Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

European Master in Software Engineering

Master Thesis

Integration and Tutorial of pmacctd Toolset on FABRIC Testbed

Author: Pilar Fernández Gayol

September, 2024

Master Thesis

European Master in Software Engineering

Integration and Tutorial of pmacctd Toolset on FABRIC Testbed.
September / 2024

Author: Pilar Fernández Gayol

Supervisor:

Oscar Dieste Tubio

Associate Professor

Lenguajes y Sistemas Informáticos e Ingeniería del Software / Escuela Técnica Superior de Ingenieros Informáticos

Universidad Politécnica de Madrid

Co-supervisor:

Nik Sultana

Ph.D. Computer Science

Trinity College, University of Cambridge

College of Computing

Illinois Institute of Technology

Abstract

FABRIC Testbed is an international infrastructure designed to support cutting-edge large-scale experimentation and research in multiple disciplines such as networking, cybersecurity, distributed computing systems and machine learning, among others.

Pmacctd, on the other hand, is a versatile open-source set of tools which collects and analyses network traffic data.

This Master's Thesis investigates the integration of the pmacctd toolset into the FABRIC testbed, enabling enhanced network monitoring and experimentation. The goal of this project is to develop a comprehensive tutorial and series of experiments to demonstrate how pmacctd can be used within FABRIC.

By creating a Jupyter Notebook that integrates both tools and automates the use of pmacctd using Python, this work offers an educational resource for computer science and engineering students, enabling them to gain hands-on experience in network monitoring, Python-based network topology creation, and network experimentation. The project covers detailed experiments on protocols and tools such as IPFIX, ICMP, and TCP/UDP, with a focus on providing practical, step-by-step examples.

The results demonstrate the powerful capabilities of pmacctd when integrated with FABRIC, while also highlighting the challenges posed by the complexity of the tool. Ultimately, this thesis contributes a valuable resource for future students and researchers in network experimentation.

Table of Contents

Table of Figures

1 Introduction

This document covers the report on the Master's Thesis entitled Integration of the Pmacctd Toolset within the FABRIC Testbed. The primary goal of this project is to obtain a successful integration between FABRIC testbed and pmacctd, to enhance network monitoring capabilities while providing an educational platform for computer science students and anyone who is interested in this topic.

FABRIC is an innovative infrastructure designed for large-scale network research and experimentation, offering a dynamic environment for testing technologies. On the other hand, pmacctd is a versatile, open-source tool for passive network traffic monitoring and analysis. By merging the functionalities of both, this project seeks to develop a comprehensive set of experiments that demonstrate how pmacctd can be effectively used for real-time, detailed network traffic monitoring within the FABRIC environment and how it could be an interesting dynamic learning resource.

In addition to the integration of these tools, another objective of this project is to expand my knowledge of Python, learning how to apply it in an entirely different field from what I usually work in — network management and monitoring. This project has provided an opportunity to explore how Python can be used to automate network experiments and interact with advanced networking infrastructures like FABRIC.

This document shows a first section on the state of the art, which is focused on describing in detail all the technologies that have been used throughout the project, as well as their main features and real applications.

Subsequently, the next section describes the process and methodology that has been followed while this master thesis has been carried out, pointing out its planning and additional key aspects that contributed to its success.

The following section describes the implementation of the integration of the tool in FABRIC, how this process has been automated using Python using a Jupyter Notebook within FABRIC and the detailed description of the experiments performed to test the potential of this combination, which includes experiments on PCAP files, IPFIX flows, and multiple protocols such as ICMP, TCP or UDP, with detailed instructions on how to configure and run these experiments within FABRIC.

Each experiment consists of two variations. The first is a simpler version using a PCAP file as the data source. The second, and more advanced version, uses live traffic capture for real-time data analysis, which is more closely related to real-world experimentation. The flexibility of FABRIC's network topologies allows for the creation of various interconnected nodes, enabling complex and realistic network scenarios

Finally, this document concludes with two distinct sections: Conclusion and Future Work, where the results are discussed, assessing whether the project has been successful and, most importantly, its potential impact on the academic community it targets. Additionally, it outlines how this project could be extended in the future, thanks to the publication of the notebook in the official FABRIC repository.

Although one of the main objectives of this project is the automation and integration of the use of these two tools in combination, the primary objective is

to create a valuable learning resource that simplifies the complex process of network monitoring and data collection using pmacctd.

This project not only showcases the integration of pmacctd into a large-scale testbed but also addresses the need for clear, accessible educational resources in network monitoring. The results and insights gained from this project contribute to both the academic and professional development of students interested in networking technologies.

2 State of the Art

The FABRIC testbed [1] is an online platform that serves as a central hub for large-scale experimentation and research in networking, providing an environment for researchers and developers to explore innovative networking technologies and solutions. On the other hand, pmacctd [2] is a set of widely used, free, and open-source, multi-purpose passive monitoring tools for data networks created by Paolo Lucente.

The integration of pmacctd into the FABRIC testbed infrastructure is expected to enhance the monitoring capabilities of the environment, enabling detailed, real-time monitoring of network traffic in an advanced research setting. Future computer science students could benefit from this integration as well, as it will give them practical experience with innovative network monitoring technologies and provide them for paths in network research and development.

2.1 Technologies

For this master's thesis, many technologies have been employed alongside pmacctd and FABRIC. This section outlines the key technologies used for the project's development, while other tools crucial for running the integration experiments, will be discussed in the experimental section. Additional technologies and tools involved in the integration, though secondary to the core focus of this study, will also further be explained upon in the experimental part of the document.

2.1.1 FABRIC

FABRIC was founded by the National Science Foundation of The United States with the collaboration of other international research centres. Its infrastructure is distributed in different collocation spaces, national labs and campuses all around the United States.

There is a total of 29 different FABRIC sites, each one equipped with a significant amount of computing and storage capacity, and they are all connected by fast, dedicated optical lines.

It offers a highly valuable solution for students or researchers who lack access to extensive network resources, as they can leverage the vast infrastructure provided by FABRIC. This allows them to broaden their experiments and studies, significantly enhancing their capabilities.

The following picture shows the map that displays the different nodes available in FABRIC and how they are interconnected.

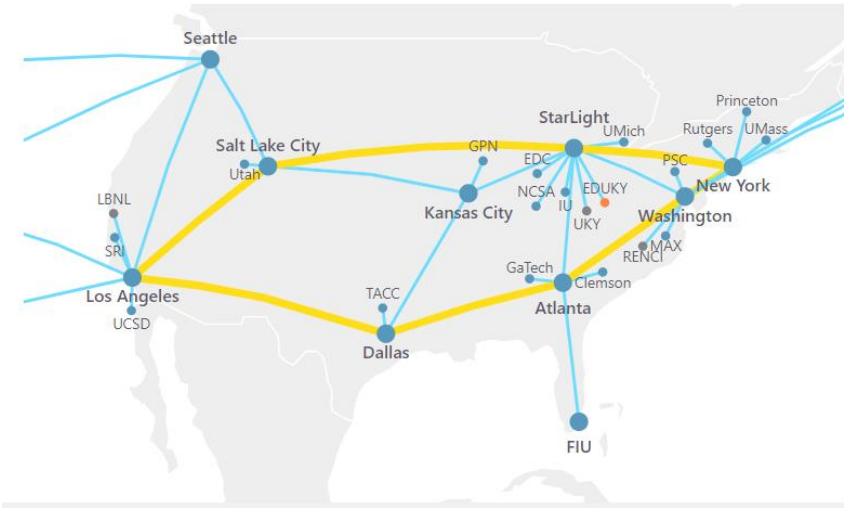

Figure 1: FABRIC node's map

In addition, FABRIC interfaces with the Internet, high-performance computing centres, and specific testbeds like 5G/IoT PAWR, providing this way a huge range of experimental operations.

It counts with 28.207 total slices created, more than 1 thousand users registered and more that 195 active projects. Additionally, a new extension called FABRIC Across Borders has been created which connects the core infrastructure in North America to four nodes in Asia, Europe and South America, enabling international cooperation in scientific investigations.

2.1.1.1 Features

The key features of FABRIC are:

- Programmable infrastructure: it enables the opportunity to define and program the behaviour of a network in different layers, allowing the customization of protocols and algorithms.
- Scalability: FABRIC enables both horizontal and vertical scalability, allowing this way the validation of experiments in diverse environments, being capable of simulate a network experiment in the real world.
- Diverse Network Topologies: it enables the creation of customizable topologies, allowing researchers and students to model real world scenarios.

2.1.1.2 Applications

The main application of FABRIC is network research and experimentation. However, FABRIC can also be used in a wide range of other different areas including cybersecurity, IoT technologies or Artificial Intelligence [3].

- Cybersecurity: testing new cybersecurity protocols and mechanisms in realistic simulation scenarios, facilitating researchers to evaluate the effectiveness of them in a controlled but dynamic environment
- IoT Technologies: experimentation with IoT architectures, enabling the simulation of a growing number of connected devices. This could help to

understand the performance, reliability and scalability of the architecture.
- Artificial Intelligence: examining AI models and algorithms in network-related contexts, investigating their uses in anomaly detection and network optimization.

2.1.2 Jupyter Hub

Jupyter Hub is a multi-user server for Jupyter Notebooks which is integrated in the FABRIC platform. It provides each user with a personalized environment, which enables the use of Jupyter Notebooks in a secure and scalable way [4].

For the integration of pmacctd and the subsequent experimentation, I will leverage Jupyter Notebook, using its open-source capabilities to create live code documents.

A key advantage of Jupyter Notebook is its seamless integration with FABRIC, allowing me to develop and run all experiments in Python. Specifically, I will use the `FablibManager`, which simplifies the interaction with FABRIC's resources and infrastructure directly from Python.

Using Jupyter Hub guarantees a customized workspace, storing progress in the cloud over time, and facilitates resource management and experiment execution. It also enhances the reproducibility of the experiments, allowing me to document, record, and rerun experiments easily, which is crucial for the long process of this thesis.

2.1.3 pmacctd

Pmacctd, is a network monitoring and traffic accounting software tool, which is designed for collecting, analysing and storing network traffic data.

Network engineer Paolo Lucente started working on it in 2003 with the goal of developing an open-source program that could manage different network protocols and offer in-depth, real-time insights into network traffic.

Since that moment the development of the tool has been driven by the community contributions. Nowadays is starting to be widely used in research, academic and industry environments, since it supports multiple protocols such as NetFlow or IPFIX.

2.1.3.1 Features

The key features of pmacctd are:

- Real-Time Monitoring: it enables the opportunity to monitor on the different traffic patterns and network performance by offering real time network data processing and collection.
- Data Aggregation and Filtering: it allows the aggregation, and the application of multiple filers based on a diverse criterion, so that specific types of traffic can be captured.
- Diverse Storage Options: it can store gathered data in message queues, SQL databases, NoSQL databases, and plain text files, among other backend systems.

- IPV4/IPV6 support: it fully supports IPv6 as well as IPv4 networks, ensuring compatibility with modern environments and architectures.

2.1.3.2 Applications

Since pmacctd is a versatile tool, it contains many applications but some of the main are:

- Network Traffic Accounting and Analysis: it collects network data in real-time, so it may help administrators to check bandwidth usage and traffic patterns.
- Security Monitoring: it can be used to identify unusual traffic patterns, detecting potential threads.
- Performance Monitoring and Troubleshooting: it helps with resource optimization by tracking performance bottlenecks, analyzing latency, and identifying network problems through real-time traffic monitoring.

3 Methodology

The project was implemented over six months, spanning the Spring and Summer semesters. The objective of the first semester was to familiarize myself with FABRIC and the tool I would be integrating into it. This involved identifying its main features and determining which aspects I should prioritize for the project.

Initially, the selected tool was not pmacctd, but ILANDS, a traffic data analysis software tool. I spent a couple of weeks studying the tool but encountered issues that prevented further progress. To compile it, it required a library that the provider had removed from the internet, and despite contacting them, I never received a response.

Subsequently, pmacctd was chosen as the tool for the project development due to its extensive features and functionalities, as outlined in section 2.1.5 of this document. This caused significant changes in the project plan, as it meant restarting the process from scratch.

The investigation process to evaluate the use of pmacctd took several weeks. I needed to ensure the tool was compiling and functioning correctly, study how to automate its installation on an Ubuntu operating system, and, most importantly, assess whether its integration with FABRIC would be worthwhile.

Once the tool was installed, I began by reading its documentation and conducting small experiments. These initial experiments were challenging because, although the documentation is detailed, it is tailored to very specific use cases. Additionally, there are a few examples available online, which extended this part of the process.

However, these challenges made the idea of accompanying the integration with a tutorial more appealing, as it could serve future students or professionals who want to start using pmacctd.

Next, I had to become familiar with FABRIC, researching how to create network architectures using Python. This step was relatively straightforward because I was taking a course where all practical exercises focused on using Python with FABRIC.

After acquiring the necessary knowledge, I began having weekly in-person meetings with Nik Sultana, who guided me on the next steps to take. These meetings were essential for ensuring the project's progress and alignment with the research objectives. Each session provided valuable insights and direction, helping to overcome any challenges and refine the implementation strategy.

As the project advanced, I began to integrate the tool with FABRIC, focusing on optimizing the network monitoring capabilities and validating the results through various tests. This phase required meticulous attention to detail and iterative adjustments to ensure the seamless functioning of the integrated system.

Gradually, together with my co-supervisor, we began creating more complex experiments, concentrating on tools widely used in real-world scenarios, both in industry and research, such as IPFIX and NetFlow.

3.1 Objectives

1. **Familiarize with FABRIC.**
   Develop a thorough understanding of the FABRIC platform, focusing on learning how to create and manage slices and using Python for network-oriented tasks.

2. **Understand and evaluate the selected tool.**
   Conduct an in-depth study of pmacctd to fully understand its capabilities, features, and potential applications. This includes assessing its compatibility with the project's goals and determining which aspects of the tool are most relevant for network monitoring and analysis within the FABRIC environment.

3. **Set up and test pmacctd in a testing environment.**
   Successfully install, compile, and verify the functionality of pmacctd in a test environment. This stage ensures that the tool can be reliably implemented and that any configuration issues are resolved before full integration.

4. **Automate the installation process of pmacctd on FABRIC.**
   Develop an automation of the entire process using Python to streamline the installation and configuration of pmacctd on an Ubuntu operating system inside a FABRIC's node.

5. **Design and conduct network experiments.**
   Create and execute network experiments that leverage pmacctd within the FABRIC environment, with a focus on enhancing network monitoring capabilities.

6. **Automate experiment workflows.**
   Build a Python Jupyter Notebook to automate the configuration, execution, and data collection for network experiments. This will allow experiments to be deployed and run within FABRIC efficiently, enabling consistent and repeatable testing scenarios.

7. **Develop a user-friendly tutorial for future use in a Jupyter Notebook.**
   Create comprehensive documentation that guides future users through the process of installing, configuring, and using pmacctd in FABRIC.

3.2 Tasks

1. **Learn and Setup FABRIC.**
   This task involves familiarizing myself with the FABRIC platform and its network architecture. To get ready for tool integration and testing, I will also set up the initial FABRIC environments. Learning how to set up network environments is part of this.

2. **Study pmacctd Tool.**
   The next task is to conduct an in-depth study of pmacctd. This will include understanding its most relevant features and functionalities. I will need to evaluate how compatible pmacctd is with FABRIC and

identify the necessary configurations to make it work effectively in the network environment.

3. **Weekly Progress Meetings.**
   This task involves holding weekly progress meetings with my co-supervisor, Nik Sultana. The purpose of these meetings is to monitor the progress of the project, discuss implementation details, and identify any obstacles or challenges that may arise. During these meetings, I will receive some feedback on my work and make any necessary adjustments to the implementation plan. These regular check-ins will help ensure the project stays on track and aligns with the research objectives.

4. **Set up pmacctd in Test Environment.**
   Once I am familiar with the tools, the next task will be to install and compile it on an Ubuntu system, setting it up in a controlled test environment on a virtual machine. During this phase, I will verify that pmacctd works as expected by testing its basic features and configurations.

5. **Automate pmacctd Installation.**
   In this task, I will automate the installation and configuration of pmacctd on FABRIC nodes. This involves writing the necessary code to automate the process of installing pmacctd. The automation should ensure that the installation process is consistent and reliable each time it is run, and I will need to validate that the setup works correctly by testing it on various nodes within FABRIC.

6. **Conduct Network Experiments Using pmacctd.**
   This task is broken into several sub-experiments, each with a specific focus:

   6.1 Traffic Monitoring with PCAP Files

   6.2 Traffic Monitoring with Live Traffic

   6.3 IPFIX Protocol Implementation

   6.4 NetFlow Protocol Analysis

7. **Automate Experiment Execution.**
   This task involves writing Python code to automate the entire workflow for the experiments. This includes automating the setup of network slices, the configuration of pmacctd, the execution of experiments, and the collection of results.

8. **Create a Tutorial.**
   In this task, I will create a detailed tutorial that guides future users through the installation, configuration, and usage of pmacctd within the FABRIC environment. The tutorial will include step-by-step instructions, practical examples, best practices, and troubleshooting tips based on the experiments I have conducted. This tutorial will serve as a valuable resource for students, researchers, or professionals looking to integrate pmacctd into their own network monitoring projects.

9. **Presentation Elaboration.**
   This task involves preparing a clear and concise presentation summarizing the project, its methodology, results, and key insights.

10. **Document Elaboration.**
    This task involves the of the design, structuring and elaboration of this document.

3.3 Gantt Chart

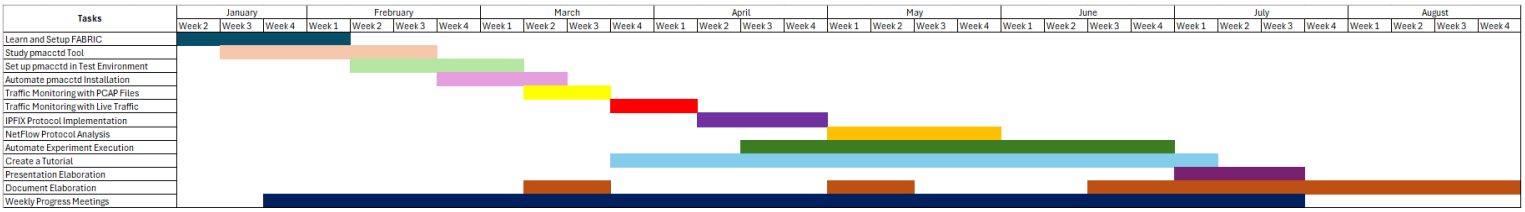

Figure 2: Gantt Chart of the project

4 Implementation

The implementation of this project consists of two phases: the integration phase and the experimentation phase. These phases are interrelated, as the experimentation is essential for validating the integration and ensures that pmacctd's deployment is effective for real-time network monitoring within FABRIC.

The tangible outcome of this implementation, as described previously, is an interactive Python Jupyter Notebook that serves as a hands-on tutorial. This interactivity is achieved through the creation of experiments that allow both users, by executing them, and myself, by designing them, to understand networking concepts and explore real-world protocols.

Both phases converge into a final phase that focuses on automating the entire process. This phase introduces the programming aspect of the project, where Python is used to create and configure FABRIC slices, install pmacctd on the nodes within the slice, set up necessary libraries and dependencies, automate file transfer between FABRIC nodes and the workspace, and manage the automation of the various experiments that form part of the tutorial.

Finally, the entire process is documented in detail, ensuring the tutorial is complete and includes all necessary information. This allows users to execute and understand each phase without needing to consult external resources.

4.1 Integration

The integration of the pmacctd tool into the FABRIC environment is described in detail in this section. It illustrates the steps involved in configuring the FABRIC slice and its nodes where pmacctd will be integrated, establishing network connections, and installing the required libraries and dependencies in order to make pmacctd to work.

4.1.1 Creation of FABRIC slice

The integration of pmacctd into FABRIC has been carried out using a Jupyter Notebook within the FABRIC environment. In this implementation, I created a slice named "`pmacctd_slice`" with two Ubuntu nodes (n1 and n2) running on the MAX site. These nodes are equipped with 2 cores, 4GB RAM, and 50GB of disk space. Each node is connected to a Layer 2 network through a network interface.

The network topology used for this project is illustrated in the following figure:

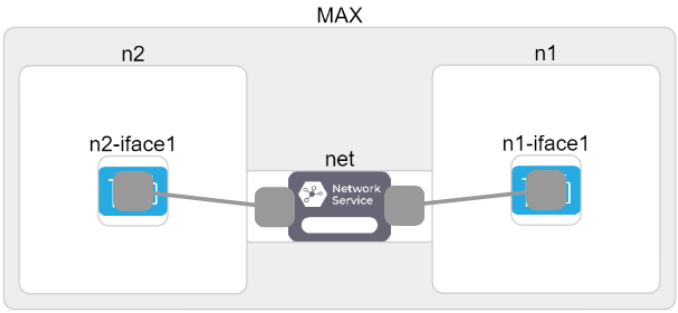

Figure 3: pmacctd Network Slice

The following figures show the characteristics of the network, and the interfaces created. This figures are generated with the python command used in the Jupyter Notebook "`slice.show()`":

Networks

ID	Name	Layer	Type	Site	Subnet	Gateway	State	Error
138e5de3-a65a-48ff-89ff-99a9f1a5d661	net	L2	L2Bridge	MAX	None	None	Active	

Figure 4: Slice's Network data

Interfaces

Name	Short Name	Node	Network	Bandwidth	Mode	VLAN	MAC	Physical Device	Device	IP Address	Numa Node
n1-iface1-p1	p1	n1	net	100	config		12:CF:6D:B3:EB:75	enp7s0	enp7s0	None	6
n2-iface1-p1	p1	n2	net	100	config		16:CC:F6:4F:22:4C	enp7s0	enp7s0	None	6

Figure 5: Slice's Interfaces data

The process that I have followed, and the code used for creating the slice is shown below.

Firstly, I have imported the FABRIC library and initialized the "`FablibManager`", which is responsible for managing and interacting with FABRIC slices [5].

If no slice exists with the given name (`pmacctd_slice`), a new one is created with the nodes and resources. This step is very interesting because if the slice is already created it just creates the necessary variables again so that they can be used in the rest of the notebook without creating a new one, which may take time.

The "`default_ubuntu_20`" image is used for the nodes, this has allowed me to integrate the pmacctd tool inside the node and interact with it, since the node has also an ubuntu terminal inside of it, as you can see on the following figure:

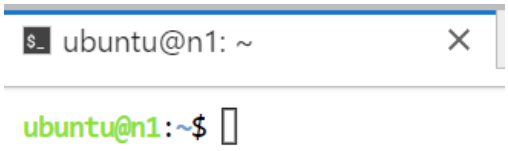

Figure 6: n1 ubuntu terminal

Once the nodes are created with the specified configurations, the slice is submitted using the "`submit()`" function, which sends the slice configuration to FABRIC and initiates the resources. By default, the slice lease lasts 24 hours but can be extended up to two weeks each time an extension is requested.

This is the code used for the creation of the slice:

```
from fabrictestbed_extensions.fablib.fablib import FablibManager as
fablib_manager

# Initialize the FABRIC library manager
fablib = fablib_manager()

# Define the slice and node names
slice_name = "pmacctd_slice"
n1_name = "n1"
n2_name = "n2"

# Try to find an existing slice with the given name
slice = None
for slice_candidate in fablib.get_slices():
if slice_candidate.get_name() == slice_name:
slice = slice_candidate
break

# If the slice does not exist, create a new one
if None == slice:
image = 'default_ubuntu_20'
site = 'MAX'
slice = fablib.new_slice(name=slice_name)

# Add the first node to the slice with specified resources
n1 = slice.add_node(name=n1_name, image=image, cores=2, ram=4, disk=50,
site=site)
n1_iface1 = n1.add_component(model="NIC_Basic",
name="iface1").get_interfaces()[0]

# Add the second node to the slice with specified resources
n2 = slice.add_node(name=n2_name, image=image, cores=2, ram=4, disk=50,
site=site)
n2_iface1 = n2.add_component(model="NIC_Basic",
name="iface1").get_interfaces()[0]

# Create a layer 2 network connecting both nodes
net = slice.add_l2network(name="net", interfaces=[n1_iface1, n2_iface1],
type="L2Bridge")
slice.submit()

# Retrieve the nodes from the slice
n1 = slice.get_node(name=n1_name)
n2 = slice.get_node(name=n2_name)
```

```
# Display the slice information
slice.show()
```

4.1.2 Installation of pmacctd

For the correct installation and usage of pmacctd, some necessary development and networking libraries are installed into the ubuntu image of both nodes. These include `build-essential` for software compilation [6], `net-tools` for network management [7], and `python3-scapy` for network packet manipulation [8]. With these dependencies in place, the environment was ready for the installation.

Following the installation of the necessary tools, the network interfaces of the nodes were configured with IPv6 addresses, and connectivity between the nodes was tested using ICMPv6 ping commands. This ensures that the nodes can communicate over the network. This is necessary since some experiments capture real time traffic that will be distributed over the network.

The IPv6 addresses chosen for n1 and n2 are fd3f:f209:c712::1 and fd3f:f209:c712::2 respectively.

Additionally, SSH connectivity was verified for both nodes, allowing for secure remote access. I have used ssh commands generated to connect into the terminals of both nodes in a secure way.

To facilitate the experiments, two functions were defined to upload and download files between the Jupyter workspace and the nodes. The functions use the "`scp`" Ubuntu command for the transmission of files. This allows the necessary files for the pmacctd tool to be transferred. And this facilitates the process for downloading the results obtained from the tool. The code of the functions is described below:

```
def cmd_upload_file_to(n, filename):
    scp_pre_command = (n.get_ssh_command()).split(" ")
    scp_pre_command[0] = 'scp'
    scp_pre_command[5] = scp_pre_command[5].split("@")
    scp_pre_command[5] = scp_pre_command[5][0] + "@[" + scp_pre_command[5][1]
+ "]:~/"
    scp_pre_command.insert(5, filename)
    return ' '.join(scp_pre_command)

def cmd_download_file_from(n, filename):
    scp_pre_command = (n.get_ssh_command()).split(" ")
    scp_pre_command[0] = 'scp'
    scp_pre_command[5] = scp_pre_command[5].split("@")
    scp_pre_command[5] = scp_pre_command[5][0] + "@[" + scp_pre_command[5][1]
+ "]:~/" + filename
    scp_pre_command.append(".")
    return ' '.join(scp_pre_command)
```

The pmacctd tool and the Jansson library, which is required for obtaining JSON output, one of the most interesting outputs supported formats, were then uploaded to the nodes, using the upload and download functions described.

Once the files were transferred, both were extracted and installed, along with their necessary dependencies, enabling the nodes to run the pmacctd tool. This installation sets the foundation for conducting detailed network analysis in the subsequent sections.

```
ubuntu@n1:~/pmacct-1.7.8$ sudo pmacctd -h
Promiscuous Mode Accounting Daemon, pmacctd 1.7.8-git (RELEASE)
Usage: pmacctd [ -D | -d ] [ -i interface ] [ -c primitive [ , ... ] ] [ -P plugin [ , ... ] ] [ filter ]
       pmacctd [ -f config_file ]
       pmacctd [ -h ]
```

Figure 7: Evidence of the installation of pmacctd on n1

4.2 Experimentation

In this section, I will present various experiments conducted to test the tool's functionalities and verify its integration within FABRIC. Initially, these experiments were performed on an Ubuntu system within a virtual machine and subsequently integrated into one or more Ubuntu nodes within the created slice.

The experiments were designed with a progressive increase in complexity and required knowledge. This approach aims to train future students in programming and networking topics effectively.

4.2.1 Configuration Files

Before starting to describe the different experiments performed, it is necessary to explain how to customize the execution of pmacctd. This is done using configuration files. These configuration files contain a set of parameters that specify the behavior of the pmacctd, such as which protocols to monitor, which fields of the captured data to save, how to handle collected data, and where to store or export it and in what format.

These configuration files will be used to perform the different experiments carried out with pmacctd.

Therefore, for each experiment, I had to design a configuration file tailored to its specific purpose. This required testing various options for each experiment, as each one is unique, and there is a lack of documentation or examples to guide

the process. Particularly in the beginning, this was a challenging task that demanded considerable effort.

For their use the file upload function to the corresponding node described above is used.

The following figure shows a simple example of a pmacctd configuration file:

```
daemonize: false
debug: true
pcap_savefile: gmail.pcapng.cap
aggregate: src_host, dst_host, src_port, dst_port, proto,
plugin_buffer_size: 4096
plugins: print
print_output_file: result_pcap_formatted.txt
print_history_roundoff: m
```

Figure 8: Configuration File example

As you can see the configuration file shows different fields. In this case it is specified that its demon mode is deactivated and that debug is activated, so that during its execution it is printed by the standard output what is occurring during execution, that the capture data must be obtained from a .pcap file, in this case specifically from "`gmail.pcapng.cap`" and that the results must be stored in the file "result_pcap_formatted.txt", no other specific format is specified.

4.2.2 PCAP Files

Objective: Evaluate the ability to analyze and process network traffic from pre-captured PCAP (Packet Capture) files. The goal of this experiment was to test pmacctd's capability to extract flow data (such as IP addresses, protocols, and port information) from PCAP files, which simulate real network traffic. This was aimed at assessing how well it can handle offline traffic data analysis. In addition, to test its versatilityand customization by using different output formats or filters.

The main functionality of pmacctd is capturing network traffic information. This information can be obtained from two different paths, from a .pcap file or from live network traffic through one or more networks interfaces, being the .pcap file the simplest one.

These two ways of obtaining information for future processing are exclusive, i.e. if it is indicated that the data will be obtained through an input file of type. pcap, it will not capture any other type of information. This is also since when pmacctd is not given an input file, it acts as a daemon. It runs and does not stop until it is told to do so, since it must be running to obtain all the information in real time, as will be seen later in this document.

A .pcap file or Packet Capture File is a data file which stores network traffic packet captures during a network session, usually created with packet-sniffing tools such as Wireshark [9] or tcpdump [10]. It is a standardized format, which enables multiple tools to read and interpret them [11].

Each PCAP file contains a header followed by a series of packet records. The records of a packet that are contained on a .pcap file are shown on the following figure.

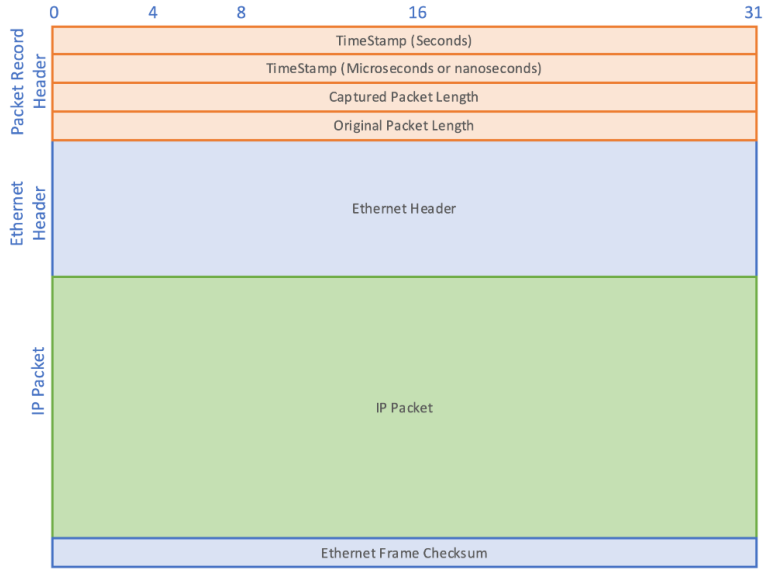

Figure 9: PCAP File Packet Records

In the following figure it can be seen the content of a .pcap file using Whireshark , specifically one of the files used for the experiments created by me by using the Python library scapy: “pfernandezgayol.pcap”.

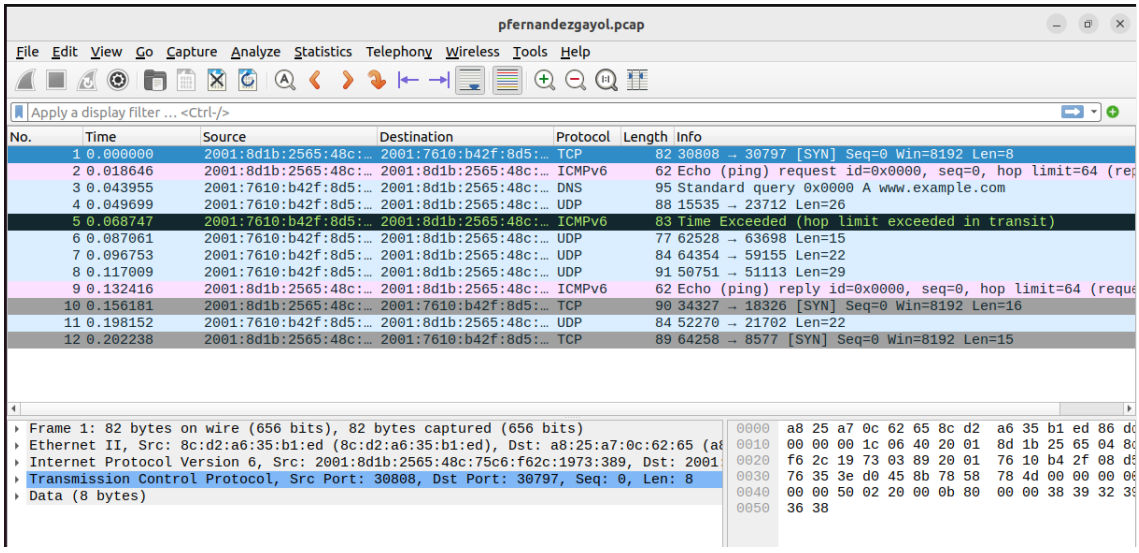

Figure 10: pfernandezgayol.pcap

The fact that so much diversity of data can be stored on a network package is very important because one of the benefits of using pmacctd is that it allows the customization of results, to work only with the data necessary for each situation.

4.2.2.1.1 Formats

In this section I am going to perform three different experiments using the same .pcap file but with different configurations of the tool, showing the results obtained from processing the pmacctd tool on a .pcap file in three different formats: CSV, JSON and formatted (.txt).

For the execution of the tool a configuration file is needed, in this section the following fields are necessary:

- daemonize: this field indicates whether the process should run in the background or not. In this case, will set to false, meaning the process will run in the foreground.
- debug: this field controls the debug mode. When set to true, it enables debugging output, which is helpful for these first experiments.
- pcap_savefile: this field specifies the path where the tool obtains the data. It will set to `$PATH_of_pmacct_tool/pmacct-1.7.8/file.pcap`. The file contains captured network traffic data.
- aggregate: this field specifies the fields by which to aggregate the captured data. In this case, it will aggregate data based on source host, destination host, source port, destination port, and protocol.
- plugin_buffer_size: this field defines the buffer size used by plugins.
- plugins: This field specifies the plugins to be used. In this example, the print plugin is specified, which likely indicates that the captured data will be printed.
- print_output: this field specifies the output format for the printed data. It will be set to csv for the first experiment since I want the output in this format.
- print_history_roundoff: this field specifies the rounding-off interval for historical data.

The following image shows an example of the configuration file used for these experiments.

```
daemonize: false
debug: true
pcap_savefile: gmail.pcapng.cap
aggregate: src_host, dst_host, src_port, dst_port, proto,
plugin_buffer_size: 4096
plugins: print
print_output: csv
print_output_file: result_pcap_csv.csv
print_history_roundoff: m
```

Figure 11: Configuration file used for experiments using .pcap files

The notebook automates using Python all the process of uploading the configuration files and the .pcap file to the node, creating the specific format result file on the node and move all the files to the correct path so that pmacctd can execute correctly and the download process of the results once everything is completed successfully.

The following three images show a piece of the results of the first three experiments:

	SRC_IP	DST_IP	SRC_PORT	DST_PORT	PROTOCOL	PACKETS	BYTES
1	192.168.1.101	178.123.13.120	42559	26895	udp	1	95
2	208.117.231.17	192.168.1.101	443	56561	tcp	71	102240
3	192.168.1.101	208.117.231.17	56562	443	tcp	47	2212
4	208.117.231.17	192.168.1.101	443	56562	tcp	54	77760
5	192.168.1.101	208.117.231.17	56561	443	tcp	53	2444
6	208.117.231.17	192.168.1.101	443	56563	tcp	63	90720
7	178.123.13.120	192.168.1.101	26895	42559	udp	1	98
8	192.168.1.101	208.117.231.17	56563	443	tcp	45	2188
9	93.184.221.133	192.168.1.101	80	56668	tcp	15	20212
10	192.168.1.100	239.255.255.250	54714	1900	udp	2	322

Figure 12: Results of the CSV pcap experiment


```
{"event_type": "purge", "ip_src": "192.168.1.101", "ip_dst": "178.123.13.120", "port_src": 42559, "port_dst": 26895, "ip_proto": "udp", "packets": 1, "bytes": 95}
{"event_type": "purge", "ip_src": "208.117.231.17", "ip_dst": "192.168.1.101", "port_src": 443, "port_dst": 56561, "ip_proto": "tcp", "packets": 71, "bytes": 102240}
{"event_type": "purge", "ip_src": "192.168.1.101", "ip_dst": "208.117.231.17", "port_src": 56562, "port_dst": 443, "ip_proto": "tcp", "packets": 47, "bytes": 2212}
{"event_type": "purge", "ip_src": "208.117.231.17", "ip_dst": "192.168.1.101", "port_src": 443, "port_dst": 56562, "ip_proto": "tcp", "packets": 54, "bytes": 77760}
{"event_type": "purge", "ip_src": "192.168.1.101", "ip_dst": "208.117.231.17", "port_src": 56561, "port_dst": 443, "ip_proto": "tcp", "packets": 53, "bytes": 2444}
{"event_type": "purge", "ip_src": "208.117.231.17", "ip_dst": "192.168.1.101", "port_src": 443, "port_dst": 56563, "ip_proto": "tcp", "packets": 63, "bytes": 90720}
{"event_type": "purge", "ip_src": "178.123.13.120", "ip_dst": "192.168.1.101", "port_src": 26895, "port_dst": 42559, "ip_proto": "udp", "packets": 1, "bytes": 98}
{"event_type": "purge", "ip_src": "192.168.1.101", "ip_dst": "208.117.231.17", "port_src": 56563, "port_dst": 443, "ip_proto": "tcp", "packets": 45, "bytes": 2188}
{"event_type": "purge", "ip_src": "93.184.221.133", "ip_dst": "192.168.1.101", "port_src": 80, "port_dst": 56668, "ip_proto": "tcp", "packets": 15, "bytes": 20212}
{"event_type": "purge", "ip_src": "192.168.1.100", "ip_dst": "239.255.255.250", "port_src": 54714, "port_dst": 1900, "ip_proto": "udp", "packets": 2, "bytes": 322}
```


Figure 13: Results of the JSON pcap experiment


```
SRC_IP          DST_IP          SRC_PORT DST_PORT PROTOCOL PACKETS BYTES
192.168.1.101   178.123.13.120  42559    26895    udp      1       95
208.117.231.17  192.168.1.101   443      56561    tcp      71      102240
192.168.1.101   208.117.231.17  56562    443      tcp      47      2212
208.117.231.17  192.168.1.101   443      56562    tcp      54      77760
192.168.1.101   208.117.231.17  56561    443      tcp      53      2444
208.117.231.17  192.168.1.101   443      56563    tcp      63      90720
178.123.13.120  192.168.1.101   26895    42559    udp      1       98
192.168.1.101   208.117.231.17  56563    443      tcp      45      2188
93.184.221.133  192.168.1.101   80       56668    tcp      15      20212
```


Figure 14: Results of the formatted pcap experiment

Pmacctd also offers an important characteristic, which enables the user to choose which fields want to obtain from the captured data. This can be done using the "`aggregate`" field on the configuration file.

Some of them may not be supported by some packets, in that case the output will be a 0, but I made an example to test some of them.

You can target characteristics of network traffic by changing the aggregate field, which can significantly improve the efficiency of data gathering and the applicability of the metrics obtained.

However, it is important to note that the availability of these fields may vary based on the packet types and the specific network configuration being used.

In this case the aggregate field of the configuration file looks like this:

```
aggregate: src_host, dst_host, src_port, dst_port, proto, src_mac, dst_mac, etype, in_iface, out_iface
```

The same process was followed as the last. pcap experiments in Python inside the Jupyter notebook, obtaining the following results:

	SRC_MAC	DST_MAC	ETYPE	IN_IFACE	OUT_IFACE	SRC_IP	DST_IP	SRC_PORT	DST_PORT	PROTOCOL	PACKETS	BYTES
1	00:14:0b:33:33:27	d0:7a:b5:96:cd:0a	800	0	0	192.168.1.101	178.123.13.120	42559	26895	udp	1	95
2	d0:7a:b5:96:cd:0a	00:14:0b:33:33:27	800	0	0	208.117.231.17	192.168.1.101	443	56561	tcp	71	102240
3	00:14:0b:33:33:27	d0:7a:b5:96:cd:0a	800	0	0	192.168.1.101	208.117.231.17	56562	443	tcp	47	2212
4	d0:7a:b5:96:cd:0a	00:14:0b:33:33:27	800	0	0	208.117.231.17	192.168.1.101	443	56562	tcp	54	77760
5	00:14:0b:33:33:27	d0:7a:b5:96:cd:0a	800	0	0	192.168.1.101	208.117.231.17	56561	443	tcp	53	2444
6	d0:7a:b5:96:cd:0a	00:14:0b:33:33:27	800	0	0	208.117.231.17	192.168.1.101	443	56563	tcp	63	90720
7	d0:7a:b5:96:cd:0a	00:14:0b:33:33:27	800	0	0	178.123.13.120	192.168.1.101	26895	42559	udp	1	98
8	00:14:0b:33:33:27	d0:7a:b5:96:cd:0a	800	0	0	192.168.1.101	208.117.231.17	56563	443	tcp	45	2188
9	d0:7a:b5:96:cd:0a	00:14:0b:33:33:27	800	0	0	93.184.221.133	192.168.1.101	80	56668	tcp	15	20212

Figure 15: Results of the custom pcap experiment example

4.2.2.1.2 Filters

The tool also allows filtering the data captured using the `pcap_filter` field on the configuration file. This allows users to narrow down the traffic they want to monitor, improving efficiency and reducing irrelevant data.

This filter syntax must be written in the same format as tcpdump uses [10], to work properly.

To test this filter, an experiment is conducted where only data from a pcap file using the TCP protocol is captured.

This approach ensures that the test focuses solely on TCP-based traffic, allowing for a more precise analysis of the filter's performance under controlled conditions.

	SRC_IP	DST_IP	SRC_PORT	DST_PORT	PROTOCOL	PACKETS	BYTES
1	208.117.231.17	192.168.1.101	443	56561	tcp	71	102240
2	192.168.1.101	208.117.231.17	56562	443	tcp	47	2212
3	208.117.231.17	192.168.1.101	443	56562	tcp	54	77760
4	192.168.1.101	208.117.231.17	56561	443	tcp	53	2444
5	208.117.231.17	192.168.1.101	443	56563	tcp	63	90720
6	192.168.1.101	208.117.231.17	56563	443	tcp	45	2188
7	93.184.221.133	192.168.1.101	80	56668	tcp	15	20212
8	192.168.1.101	90.84.59.130	56671	443	tcp	4	161
9	192.168.1.101	93.184.221.133	56668	80	tcp	11	501
10	192.168.1.101	173.194.35.53	56643	443	tcp	21	5083

Figure 16: Results of the filter pcap experiment example

A wide range of filters can be employed, from simple ones like the one currently in use, to highly specific filters. This feature is especially useful in fields like cybersecurity, where threat detection, network monitoring, and forensic

investigations can all be substantially improved by having precise data capture and analysis capabilities.

Advanced filters are an effective tool for spotting weaknesses or predicting possible attacks since they may help pointing suspicious patterns or anomalies in network traffic.

4.2.3 Live traffic

Objective: test pmacctd's ability to monitor and capture real-time network traffic within the FABRIC environment. The objective was to ensure pmacctd could monitor dynamic network conditions and provide live insights without significant latency or packet loss, checking its efficiency and integration.

As described earlier in this document, one of pmacctd's key features is its ability to capture live network traffic directly from the network interfaces. This can be tested through an experiment since the node where pmacctd was installed is part of a network slice which is connected to other nodes through its interface.

The following picture shows the network configuration of the node n1 which will be the one that will be used for these experiments.

```
enp3s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 9000
        inet 10.30.6.153  netmask 255.255.254.0  broadcast 10.30.7.255
        inet6 fe80::f816:3eff:fe0f:263e  prefixlen 64  scopeid 0x20<link>
        inet6 2001:468:c00:ffc4:f816:3eff:fe0f:263e  prefixlen 64  scopeid 0x0<global>
        ether fa:16:3e:0f:26:3e  txqueuelen 1000  (Ethernet)
        RX packets 383321  bytes 409233621 (409.2 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 81676  bytes 8755266 (8.7 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

enp7s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::10cf:6dff:feb3:eb75  prefixlen 64  scopeid 0x20<link>
        inet6 fd3f:f209:c712::1  prefixlen 48  scopeid 0x0<global>
        ether 12:cf:6d:b3:eb:75  txqueuelen 1000  (Ethernet)
        RX packets 24  bytes 2264 (2.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 232  bytes 17248 (17.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 1182  bytes 121624 (121.6 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1182  bytes 121624 (121.6 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 17: Network configuration of the node n1

This feature is achieved by using the `pcap_interface` field in its configuration, which defines the network interface from which the tool will listen for traffic. The program allows for flexibility in the type and structure of the data captured by capturing packets in real time and processing them according to the parameters specified in the configuration file.

By leveraging this live capture functionality, pmacctd can monitor a wide range of protocols, including ICMP, TCP, and UDP, across any number of network interfaces. The traffic can be aggregated and filtered in multiple ways, such as

by source or destination IP, protocol, or ports. Once the data is captured, pmacctd can output it in various formats, like CSV or JSON. The formats will vary on the different experiments of this section.

4.2.4 ICMP

In this section, I am going to perform an experiment using live traffic generated by sending ICMP packets between n1 and n2 nodes via the ping command. I will capture and process this traffic using pmacctd and demonstrate the results in a .txt format. For this experiment, I will use the following configuration fields:

```
pcap_interface: eth3s0
```

The `pcap_interface` field specifies the network interface from which traffic is captured. This can be set to "any" to capture from all available interfaces, or a specific interface can be named, as in this specific case.

For this experiment I have had to use the threading library from python for complete automation of the execution of the tool. This occurs because pmacctd acts as a demon, so once that you execute it, it keeps capturing all the traffic data from the interface specified, until it is stopped manually.

Two threads were created to run functions "`run_pmacctd`" on nodes n1 and n2, executing commands one for starting the daemon and the second which creates ICMP traffic from n2 to n1, respectively. This allows simultaneous operation of network monitoring (pmacctd) and IPv6 ping testing. The use of the "`join()`" function ensures synchronization, waiting for completion of the second thread before ending execution.

The following picture shows the results of the experiment:

	SRC_IP	DST_IP	SRC_PORT	DST_PORT
	PROTOCOL PACKETS BYTES			
2	fd3f:f209:c712::2	fd3f:f209:c712::1	0	0
	ipv6-icmp 3 312			
3	fd3f:f209:c712::1	fd3f:f209:c712::2	0	0
	ipv6-icmp 3 312			
4	fe80::f816:3eff:febb:9db7	ff02::1:ffe3:fe9f	0	0
	ipv6-icmp 1 72			
5	2001:400:a100:3000:f816:3eff:febc:8269	2610:1e0:1700:202::7	22	49796
	tcp 2 416			
6	2610:1e0:1700:202::7	2001:400:a100:3000:f816:3eff:febc:8269	49796	22
	tcp 1 72			
7	127.0.0.1	127.0.0.6	55448	4739
	udp 1 216			
8	127.0.0.1	127.0.0.4	40777	4738
	udp 1 212			
9	127.0.0.4	127.0.0.1	0	0
	icmp 2 480			
10	127.0.0.1	127.0.0.4	43598	4739
	udp 1 212			

Figure 18: Results of capturing ICMP traffic between n1 and n2

As can be observed, the tool captures more packets than expected, as it collects all the traffic reaching node n1. To address this, I can apply a filter to capture only the packets originating from node n2 that use the ICMPv6 protocol.

This feature is also quite useful when working with live traffic, as it enables you to focus on specific data of interest or identify anomalies. By applying filters,

you can streamline your data analysis, making it easier to pinpoint relevant information.

I am going to filter by the host's IP address and the type of protocol. Additionally, I can remove these parameters from the aggregation field as the collected data will already include this information uniformly.

To achieve this, I used the following filter:

```
pcap_filter: icmp6 and src host fd3f:f209:c712::2
```

The result can be observed in the following picture:

	ETYPE	IN_IFACE	OUT_IFACE	SRC_PORT	DST_PORT	PACKETS	BYTES
1	86dd	0	0	0	0	17	1728

Figure 19: Filtered results of capturing ICMP traffic between n1 and n2

4.2.5 TCP and UDP

In this section I am going to continue to use live traffic to test the pmacctd tool, but this time I am going to use iPerf [12]. iPerf is a network performance testing tool used to measure bandwidth between two hosts over a TCP or UDP connection. iPerf works by establishing a client-server model where one instance of iPerf acts as the server and another as the client.

The installation of iPerf is automatically handled within the nodes via Python, as the tool is readily available on Ubuntu. I have used iPerf3, which is better suited for IPv6 addresses—the format used by our nodes.

The objective of this is to test the pmacctd tool this time with UDP and TCP packets, rather than the ICMP packets used in the previous experiment.

This time have used the same multi-thread approach to test the tool but using iPerf. Node n1 acted as the server node, while n2 acted as the client, but first I had to execute the daemon tool on the server node, which finishes when the last command completes.

In this scenario, I have had three threads: the first one starts the pmacctd tool, the second configures node n1 as the server, and the third configures node n2 as the client. All nodes finish when the third thread completes its task.

The following images displays the results of the iperf3 and pmacctd executions:

```
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
Accepted connection from fd3f:f209:c712::2, port 35218
[  5] local fd3f:f209:c712::1 port 5201 connected to fd3f:f209:c712::2 port 35228
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec  1.52 GBytes  13.0 Gbits/sec
[  5]   1.00-2.00   sec  1.51 GBytes  13.0 Gbits/sec
[  5]   2.00-3.00   sec  1.47 GBytes  12.6 Gbits/sec
[  5]   3.00-4.00   sec  1.51 GBytes  13.0 Gbits/sec
[  5]   4.00-5.00   sec  1.58 GBytes  13.6 Gbits/sec
[  5]   5.00-6.00   sec  1.57 GBytes  13.5 Gbits/sec
[  5]   6.00-7.00   sec  2.47 GBytes  21.3 Gbits/sec
[  5]   7.00-8.00   sec  2.46 GBytes  21.1 Gbits/sec
[  5]   8.00-9.00   sec  1.97 GBytes  17.0 Gbits/sec
[  5]   9.00-10.00  sec  2.73 GBytes  23.4 Gbits/sec
[  5]  10.00-10.00  sec   690 KBytes  21.7 Gbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-10.00  sec  18.8 GBytes  16.2 Gbits/sec                  receiver
Connecting to host fd3f:f209:c712::1, port 5201
[  5] local fd3f:f209:c712::2 port 35228 connected to fd3f:f209:c712::1 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  1.53 GBytes  13.1 Gbits/sec  312   1.38 MBytes
[  5]   1.00-2.00   sec  1.51 GBytes  13.0 Gbits/sec   38    785 KBytes
[  5]   2.00-3.00   sec  1.47 GBytes  12.6 Gbits/sec    0    844 KBytes
[  5]   3.00-4.00   sec  1.51 GBytes  13.0 Gbits/sec    1    898 KBytes
[  5]   4.00-5.00   sec  1.58 GBytes  13.6 Gbits/sec    0    918 KBytes
[  5]   5.00-6.00   sec  1.57 GBytes  13.5 Gbits/sec    0    922 KBytes
[  5]   6.00-7.00   sec  2.47 GBytes  21.2 Gbits/sec  560   1.41 MBytes
[  5]   7.00-8.00   sec  2.46 GBytes  21.1 Gbits/sec  434   1.46 MBytes
[  5]   8.00-9.00   sec  1.97 GBytes  17.0 Gbits/sec  171   1.48 MBytes
[  5]   9.00-10.00  sec  2.73 GBytes  23.4 Gbits/sec   30   1.74 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  18.8 GBytes  16.2 Gbits/sec  1546             sender
[  5]   0.00-10.00  sec  18.8 GBytes  16.2 Gbits/sec                  receiver

iperf Done.
```

Figure 20: iperf3 execution results

2600:2701:5000:a902::c	2001:468:c00:ffc4:f816:3eff:fe0f:263e	33516	22	tcp	24	4392
2001:468:c00:ffc4:f816:3eff:fe0f:263e	2600:2701:5000:a902::c	22	33516	tcp	21	5986
2600:2701:5000:a902::c	2001:468:c00:ffc4:f816:3eff:fe0f:263e	33522	22	tcp	22	4168
2001:468:c00:ffc4:f816:3eff:fe0f:263e	2600:2701:5000:a902::c	22	33522	tcp	17	6066

Figure 21: Results of the iperf experiment

4.3 IPFIX

Objective: The objective of the remaining experiments is to understand the practical applications of various protocols, such as IPFIX and NetFlow, commonly used in real-world scenarios, particularly within the corporate environment. This knowledge is intended to benefit not only my own understanding but also future users of the proposed tutorial, equipping us with relevant skills and insights for professional development. By becoming familiar with these protocols, users will be better prepared to apply these tools and techniques in industry contexts. In addition, another goal is to test pmacctd's compatibility with IPFIX collectors and verify that it could reliably export flow data

IPFIX (IP Flow Information Export) is a standardized protocol for exporting detailed information about IP traffic flows from network devices such as routers and switches. These flows capture crucial metadata, including packet counts, timestamps, protocol types, ports, source and destination IP addresses, and more. This data is essential for tasks like security monitoring, traffic analysis,

and overall network performance evaluation. IPFIX plays a critical role in detecting anomalies, optimizing performance, and ensuring network security.

IPFIX is a more advanced and flexible version of NetFlow, offering greater customization and richer detail in flow information export [13] [14].

In this section, I will produce and analyze IPFIX flows using pmacctd with both pcap files and live traffic. The process will be conducted in four phases with incremental difficulty:

1. Printing the result on a terminal.
2. Logging the output into a Log File.
3. Generating an .ipfix File and Analyzing with Tshark.
4. Sending the IPFIX Flows to a Collector.

Each step is gradually more advanced, from local traffic capture and analysis to exporting flows to a collector for more complex network monitoring and analysis tasks. Through this experiment, I will explore the full potential of pmacctd in using IPFIX flows in monitoring and analyzing network traffic.

In addition, it is also highly valuable to include this type of experiment, as IPFIX and NetFlow are widely used in the corporate world today. Since the major goal of this project is to serve as a learning tool for students, incorporating these technologies makes the content even more practical for real-world applications.

4.3.1 Terminal Output

In this first experiment, I will run a simple test with a pmacctd configuration file that will contain the fields explained below. The objective is to record network flow data and illustrate how the tool can export this data in an IPFIX format. The objective of it is to test the pmacctd interactions with IPFIX flows.

The different special configuration file fields used for this experiment are:

```
plugins: nfprobe, print
nfprobe_version: 10
```

- nfprobe: this plugin is used for exporting network flow data using the NetFlow/IPFIX protocol.
- print: this plugin is used for printing the collected data, in this case to standard output since I have not specified an output file.
- nfprobe_version: 10 refers to IPFIX, which is also known as NetFlow version 10.

4.3.1.1 Pcap

First, the experiment was conducted using a .pcap file. The results were satisfactory, as the output displayed the flows generated from the captured packets, demonstrating that the tool successfully processed the traffic data.

SRC_IP	PACKETS	BYTES
192.168.1.101	366	30583
208.117.231.17	239	344160
178.123.13.120	1	98
93.184.221.133	15	20212
192.168.1.100	2	322
90.84.136.33	9	6020
173.194.35.53	26	13704
68.232.35.139	4	191
173.194.66.84	11	4881
157.56.106.184	1	137
fe80::6d49:f772:ed9b:d16a	3	283
74.125.206.125	1	66
173.194.35.39	5	2195

Figure 22: Results of the IPIFX .pcap experiment 1

First, the flows were generated from the captured network traffic. These flows contain metadata. Once the flows were created, the tool packages them into IPFIX packets, as you can see in the following figures.

These packets could be then prepared for export, allowing the detailed flow information to be analyzed or sent to a collector for further processing. But these will be considered in the following experiments, since the objective of this one is just printing all the information on the terminal.

```
 INFO ( default/core ): Promiscuous Mode Accounting Daemon, pmacctd 1.7.8-git (RELEASE)
INFO ( default/core ):  '--enable-jansson' '--enable-l2' '--enable-traffic-bins' '--enable-bgp-bins' '--e
nable-bmp-bins' '--enable-st-bins'
INFO ( default/core ): Reading configuration file '/home/ubuntu/pmacct-1.7.8/ipfix_1.conf'.
WARN ( default_print/print ): defaulting to SRC HOST aggregation.
INFO ( default/core ): [,0] link type is: 1
INFO ( default_nfprobe/nfprobe ): plugin_pipe_size=4096000 bytes plugin_buffer_size=10240 bytes
INFO ( default_nfprobe/nfprobe ): ctrl channel: obtained=212992 bytes target=3200 bytes
INFO ( default_print/print ): plugin_pipe_size=4096000 bytes plugin_buffer_size=10240 bytes
INFO ( default_print/print ): ctrl channel: obtained=212992 bytes target=3200 bytes
INFO ( default_nfprobe/nfprobe ): NetFlow probe plugin is originally based on softflowd 0.9.7 software, C
opyright 2002 Damien Miller <djm@mindrot.org> All rights reserved.
INFO ( default_nfprobe/nfprobe ):           TCP timeout: 3600s
INFO ( default_nfprobe/nfprobe ):  TCP post-RST timeout: 120s
INFO ( default_nfprobe/nfprobe ):  TCP post-FIN timeout: 300s
INFO ( default_nfprobe/nfprobe ):           UDP timeout: 300s
INFO ( default_nfprobe/nfprobe ):          ICMP timeout: 300s
INFO ( default_nfprobe/nfprobe ):       General timeout: 3600s
INFO ( default_nfprobe/nfprobe ):      Maximum lifetime: 604800s
INFO ( default_nfprobe/nfprobe ):       Expiry interval: 60s
INFO ( default_nfprobe/nfprobe ): Exporting flows to [127.0.0.1]:2100
INFO ( default/core ): PCAP capture file, sleeping for 2 seconds
INFO ( default_print/print ): cache entries=16411 base cache memory=54878384 bytes
WARN ( default_print/print ): no print_output_file and no print_output_lock_file defined.
DEBUG ( default_nfprobe/nfprobe ): ADD FLOW seq:1 [178.123.13.120]:26895 <> [192.168.1.101]:42559 proto:1
7
DEBUG ( default_nfprobe/nfprobe ): ADD FLOW seq:2 [192.168.1.101]:56561 <> [208.117.231.17]:443 proto:6
DEBUG ( default_nfprobe/nfprobe ): ADD FLOW seq:3 [192.168.1.101]:56562 <> [208.117.231.17]:443 proto:6
DEBUG ( default_nfprobe/nfprobe ): ADD FLOW seq:4 [192.168.1.101]:56563 <> [208.117.231.17]:443 proto:6
DEBUG ( default_nfprobe/nfprobe ): ADD FLOW seq:5 [93.184.221.133]:80 <> [192.168.1.101]:56668 proto:6
DEBUG ( default_nfprobe/nfprobe ): ADD FLOW seq:6 [192.168.1.100]:54714 <> [239.255.255.250]:1900 proto:1
7
```

Figure 23: Results of the IPIFX .pcap experiment 1

```
DEBUG ( default_nfprobe/nfprobe ): Building IPFIX packet: offset = 134, template ID = 1024, total len = 1
18
DEBUG ( default_nfprobe/nfprobe ): Building IPFIX packet: offset = 248, template ID = 1024, total len = 2
32
DEBUG ( default_nfprobe/nfprobe ): Building IPFIX packet: offset = 362, template ID = 1024, total len = 3
46
DEBUG ( default_nfprobe/nfprobe ): Building IPFIX packet: offset = 476, template ID = 1024, total len = 4
60
DEBUG ( default_nfprobe/nfprobe ): Sending NetFlow v9/IPFIX packet: len = 476
```

Figure 24: Results of the IPIFX .pcap experiment 1

4.3.1.2 Live traffic

The same experiment was performed using the live traffic captured by node n1, this experiment was also successful. However, since the IPFIX flows were generated as pmacctd captured the network packets, the output per terminal is

very messy, so we will continue with the experiments since, using the next approaches this does not occur.

```
DEBUG ( default_nfprobe/nfprobe ): ADD FLOW seq:20 [fe80::f816:3eff:feb5:f142]:0 <> [ff02::16]:0 proto:58
DEBUG ( default_nfprobe/nfprobe ): ADD FLOW seq:21 [fe80::f816:3eff:fef7:a8f]:0 <> [ff02::16]:0 proto:58
  DEBUG ( default_nfprobe/nfprobe ): ADD FLOW seq:22 [fe80::f816:3eff:fe2c:1282]:0 <> [ff02::2]:0 proto:58
DEBUG ( default_nfprobe/nfprobe ): ADD FLOW seq:23 [fe80::f816:3eff:fedc:b34]:0 <> [ff02::2]:0 proto:58
 2001:468:c00:ffc4:f816:3eff:fe0f:263e          15                    9088
2600:2701:5000:a902::c                          5                     360
2610:1e0:1700:202::7                            9                     648
fe80::f816:3eff:fe28:f3f6                       2                     256
fe80::f816:3eff:fe00:26d3                       2                     152
fe80::f816:3eff:fe6a:e567                       2                     152
fe80::f816:3eff:fe8c:84c6                       2                     152
fe80::f816:3eff:fe3a:f04                        2                     152
fe80::f816:3eff:fef2:a942                       2                     192
fe80::f816:3eff:fee0:c123                       2                     152
fe80::f816:3eff:fe0f:263e                       2                     152
fe80::f816:3eff:fea9:c748                       2                     152
fe80::f816:3eff:fe80:4724                       2                     152
fe80::f816:3eff:fe09:66a6                       2                     152
fe80::f816:3eff:feb5:f142                       2                     152
fe80::f816:3eff:fef7:a8f                        2                     152
fe80::f816:3eff:fe2c:1282                       1                     48
fe80::f816:3eff:fedc:b34                        1                     48
 INFO ( default_print/print ): *** Purging cache - START (PID: 131930) ***
  INFO ( default_print/print ): *** Purging cache - END (PID: 131930, QN: 18/18, ET: 0) ***
  INFO ( default_print/print ): *** Purging cache - START (PID: 131935) ***
 2001:468:c00:ffc4:f816:3eff:fe0f:263e          10                    5656
2610:1e0:1700:202::7                            6                     432
fe80::f816:3eff:fe28:f3f6                       2                     256
2600:2701:5000:a902::c                          3                     216
```

Figure 25: Results of the IPIFX live experiment 1

4.3.2 Log File

In this second approach, the `print_output_file` field is used to specify the log file where the captured results will be exported. Two experiments were conducted: one using .pcap files and the other capturing live traffic from node n1, following the same multi-threading approach described earlier.

```
print_output_file: ipfix_2.log
```

In the results, we observe that the output only displays packets grouped by their source IP address, indicating the number of packets for each source and the number of bytes they occupy.

While this approach is less messy, it provides limited information. The figures below show the results obtained from both experiments.

4.3.2.1 Pcap

SRC_IP	PACKETS	BYTES
192.168.1.101	366	30583
208.117.231.17	239	344160
178.123.13.120	1	98
93.184.221.133	15	20212
192.168.1.100	2	322
90.84.136.33	9	6020
173.194.35.53	26	13704
68.232.35.139	4	191
173.194.66.84	11	4881

Figure 26: Result of IPIFX .pcap experiment 2

4.3.2.2 Live traffic

SRC_IP	PACKETS	BYTES
2001:468:c00:ffc4:f816:3eff:fe3b:6ec0	88	26026
2610:1e0:1700:202::7	19	1368
fd3f:f209:c712::2	21	2144
fd3f:f209:c712::1	21	2144
fe80::4cc:fcff:feb8:a782	3	208
fe80::9b:aeff:fe2a:2143	3	208
2600:2701:5000:a902::c	80	15001
fe80::f816:3eff:fe28:f3f6	2	256
fe80::f816:3eff:fe2d:a5ae	2	152

Figure 27: Result of IPIFX live experiment 2

4.3.3 IPFIX File

In this next experiment, we aim to achieve more interesting and comprehensive results. The objective is to generate two .IPFIX files: one from capturing data using a .pcap file with pmacctd, and another from capturing live traffic within our network slice.

For this, I will configure the following field of the configuration file:

```
nfprobe_receiver: 127.0.0.1:4739
```

nfprobe_receiver: it configures pmacctd to send the collected network flow data to a local collector on IP address 127.0.0.1 (local) running on port 4739.

The first step in this process involves redirecting traffic from port 4739 to a file. We accomplished this by using the Netcat utility [15]. First, we have ensured that Netcat is installed, then execute the following command to redirect the IPFIX flows to a specified file.

So, we first need to download it and then use the following command to redirect the ipfix flows created to it:

```
nc -u -l 4739 > pmacct-1.7.8/ipfix_flows_3.ipfix.
```

This command sets up Netcat to listen for incoming UDP connections on port 4739 and redirects the output to the file ipfix_flows_3.ipfix. By doing this, we can capture the IPFIX flow data generated by pmacctd.

This process is automated within Python in the Jupyter Notebook. The following code illustrates how traffic is redirected and how pmacctd is subsequently executed:

```
import threading

def run_iperf(node, command):
    result = node.execute(command)
    print(result)
command_1 = "nc -u -l 4738 > pmacct-1.7.8/ipfix_flows_3.ipfix"
command_2 = "cd pmacct-1.7.8 && sudo pmacctd -f ipfix_3.conf"

thread_1 = threading.Thread(target=run_iperf, args=(n1, command_1))
thread_2 = threading.Thread(target=run_iperf, args=(n1, command_2))

thread_1.start()
thread_2.start()

thread_2.join()
```

TShark is a potent network protocol analyzer that we must download in order to examine the content of the flows in the result file. Since the script cannot automatically obtain the necessary permissions, this step requires manually downloading TShark.

After installing TShark, we can use it to examine our IPFIX file's contents and see the different flows that were recorded within. To obtain a summary of the packets that were captured, the packet summary data from the IPFIX file can be read and displayed using the following command:

```
n1.execute("tshark -r pmacct-1.7.8/ipfix_flows_3.ipfix")
```

In the following command, -c 1 specifies that only the first packet should be displayed. This parameter can be modified to analyze different packets by executing the command again with the desired packet number.

```
n1.execute("tshark -r pmacct-1.7.8/ipfix_flows_3.ipfix -x -c 1")
```

4.3.3.1 Pcap

The obtained results of the experiment using a .pcap format file are shown below:

```
[42]: n1.execute("tshark -r pmacct-1.7.8/ipfix_flows_3.ipfix")

    1    0    →    CFLOW 408 IPFIX flow ( 408 bytes) Obs-Domain-ID=    0 [Data-Template:1024] [Data-Template:1025] [Data-Template:2048] [Data-Template:2049] [Data:1024]
    2    0    →    CFLOW 476 IPFIX flow ( 476 bytes) Obs-Domain-ID=    0 [Data:1024]
    3    0    →    CFLOW 476 IPFIX flow ( 476 bytes) Obs-Domain-ID=    0 [Data:1024]
    4    0    →    CFLOW 488 IPFIX flow ( 488 bytes) Obs-Domain-ID=    0 [Data:1024] [Data:2048] [Data:1024] [Data:2048]
    5    0    →    CFLOW 476 IPFIX flow ( 476 bytes) Obs-Domain-ID=    0 [Data:2048] [Data:1024]
    6    0    →    CFLOW 508 IPFIX flow ( 508 bytes) Obs-Domain-ID=    0 [Data:2048] [Data:1024]
    7    0    →    CFLOW 452 IPFIX flow ( 452 bytes) Obs-Domain-ID=    0 [Data:2048] [Data:1024]
    8    0    →    CFLOW 476 IPFIX flow ( 476 bytes) Obs-Domain-ID=    0 [Data:1024]
    9    0    →    CFLOW 476 IPFIX flow ( 476 bytes) Obs-Domain-ID=    0 [Data:1024]
   10    0    →    CFLOW 476 IPFIX flow ( 476 bytes) Obs-Domain-ID=    0 [Data:1024]
   11    0    →    CFLOW 420 IPFIX flow ( 420 bytes) Obs-Domain-ID=    0 [Data:1024]
   12    0    →    CFLOW 476 IPFIX flow ( 476 bytes) Obs-Domain-ID=    0 [Data:1024]
   13    0    →    CFLOW 420 IPFIX flow ( 420 bytes) Obs-Domain-ID=    0 [Data:1024]
   14    0    →    CFLOW 476 IPFIX flow ( 476 bytes) Obs-Domain-ID=    0 [Data:1024]
   15    0    →    CFLOW 308 IPFIX flow ( 308 bytes) Obs-Domain-ID=    0 [Data:1024]

[42]: ('    1    0    →    CFLOW 408 IPFIX flow ( 408 bytes) Obs-Domain-ID=    0 [Data-Template:1024] [Data-Template:1025] [Data-Template:2048] [Data-Template:2049]
[Data:1024]\n    2    0    →    CFLOW 476 IPFIX flow ( 476 bytes) Obs-Domain-ID=    0 [Data:1024]\n    3    0    →    CFLOW 476 IPFIX
flow ( 476 bytes) Obs-Domain-ID=    0 [Data:1024]\n    4    0    →    CFLOW 488 IPFIX flow ( 488 bytes) Obs-Domain-ID=    0 [Data:1024] [Data:2048] [Data:1024]
[Data:2048]\n    5    0    →    CFLOW 476 IPFIX flow ( 476 bytes) Obs-Domain-ID=    0 [Data:2048] [Data:1024]\n    6    0    →    CFLO
W 508 IPFIX flow ( 508 bytes) Obs-Domain-ID=    0 [Data:2048] [Data:1024]\n    7    0    →    CFLOW 452 IPFIX flow ( 452 bytes) Obs-Domain-ID=    0 [Data:2048]
[Data:1024]\n    8    0    →    CFLOW 476 IPFIX flow ( 476 bytes) Obs-Domain-ID=    0 [Data:1024]\n    9    0    →    CFLOW 476 IPFIX
flow ( 476 bytes) Obs-Domain-ID=    0 [Data:1024]\n   10    0    →    CFLOW 476 IPFIX flow ( 476 bytes) Obs-Domain-ID=    0 [Data:1024]\n   11    0
→    CFLOW 420 IPFIX flow ( 420 bytes) Obs-Domain-ID=    0 [Data:1024]\n   12    0    →    CFLOW 476 IPFIX flow ( 476 bytes) Obs-Domain-ID=    0 [Dat
a:1024]\n   13    0    →    CFLOW 420 IPFIX flow ( 420 bytes) Obs-Domain-ID=    0 [Data:1024]\n   14    0    →    CFLOW 476 IPFIX flow
( 476 bytes) Obs-Domain-ID=    0 [Data:1024]\n   15    0    →    CFLOW 308 IPFIX flow ( 308 bytes) Obs-Domain-ID=    0 [Data:1024]\n',
 '')
```

Figure 28: Result of IPIFX .pcap experiment 3

```
0000  00 0a 01 98 66 f6 db d0 00 00 00 00 00 00 00 00   ....f...........
0010  00 02 00 44 04 00 00 0f 00 99 00 08 00 98 00 08   ...D............
0020  00 01 00 08 00 02 00 08 00 3c 00 01 00 0a 00 04   .........<......
0030  00 0e 00 04 00 3d 00 01 00 08 00 04 00 0c 00 04   .....=..........
0040  00 07 00 02 00 0b 00 02 00 05 00 01 00 06 00 01   ................
0050  00 04 00 01 00 02 00 44 04 01 00 0f 00 99 00 08   .......D........
0060  00 98 00 08 00 01 00 08 00 02 00 08 00 3c 00 01   .............<..
0070  00 0a 00 04 00 0e 00 04 00 3d 00 01 00 08 00 04   .........=......
0080  00 0c 00 04 00 07 00 02 00 0b 00 02 00 05 00 01   ................
0090  00 06 00 01 00 04 00 01 00 02 00 44 08 00 00 0f   ...........D....
00a0  00 99 00 08 00 98 00 08 00 01 00 08 00 02 00 08   ................
00b0  00 3c 00 01 00 0a 00 04 00 0e 00 04 00 3d 00 01   .<...........=..
00c0  00 1b 00 10 00 1c 00 10 00 05 00 01 00 07 00 02   ................
00d0  00 0b 00 02 00 06 00 01 00 04 00 01 00 02 00 44   ...............D
00e0  08 01 00 0f 00 99 00 08 00 98 00 08 00 01 00 08   ................
00f0  00 02 00 08 00 3c 00 01 00 0a 00 04 00 0e 00 04   .....<..........
0100  00 3d 00 01 00 1b 00 10 00 1c 00 10 00 05 00 01   .=..............
0110  00 07 00 02 00 0b 00 02 00 06 00 01 00 04 00 01   ................
0120  04 00 00 78 00 00 01 47 b3 23 0c 28 00 00 01 47   ...x...G.#.(...G
0130  b3 23 0a ee 00 00 00 00 00 00 00 bf 00 00 00 00   .#..............
0140  00 00 00 04 04 00 00 00 00 00 00 00 00 00 44 e8   ..............D.
0150  23 8b c0 a8 01 65 01 bb dd 53 00 19 06 00 00 01   #....e...S......
0160  47 b3 23 0c 28 00 00 01 47 b3 23 0a ee 00 00 00   G.#.(...G.#.....
0170  00 00 00 00 97 00 00 00 00 00 00 00 03 04 00 00   ................
0180  00 00 00 00 00 00 00 c0 a8 01 65 44 e8 23 8b dd   ..........eD.#..
0190  53 01 bb 00 1d 06 00 00                           S.......
```

Figure 29: Result of IPIFX .pcap experiment 3

4.3.3.2 Live traffic

This process is also automated using the threading library in Python, like the approach taken in the previous experiment. However, in this case, three different threads were required. The following code illustrates this process:

```
import threading

def run_iperf(node, command):
    result = node.execute(command)
    print(result)
command_1 = "nc -u -l 4738 > pmacct-1.7.8/ipfix_flows_live_3.ipfix"
command_2 = "cd pmacct-1.7.8 && sudo pmacctd -f ipfix_live_3.conf"
command_3 = "ping -c 20 " + n1_ip

thread_1 = threading.Thread(target=run_iperf, args=(n1, command_1))
```

```
thread_2 = threading.Thread(target=run_iperf, args=(n1, command_2))
thread_3 = threading.Thread(target=run_iperf, args=(n2, command_3))

thread_1.start()
thread_2.start()
thread_3.start()

thread_3.join()
```

The obtained results of the experiment capturing live traffic packets from n1 are shown below:

```
0000  00 0a 00 b8 66 f6 e0 95 00 00 00 5a 00 00 00 00   ....f......Z....
0010  08 00 00 a8 00 00 01 92 34 5a 96 c5 00 00 01 92   ........4Z......
0020  34 5a 95 f2 00 00 00 00 00 00 08 c9 00 00 00 00   4Z..............
0030  00 00 00 12 06 00 00 00 00 00 00 00 00 00 20 01   .............. .
0040  04 68 0c 00 ff c4 f8 16 3e ff fe 0f 26 3e 20 01   .h......>...&> .
0050  06 7c 15 62 00 00 00 00 00 00 00 00 00 22 00 81   .|.b.........".."
0060  a2 01 bb 1f 06 00 00 01 92 34 5a 96 c5 00 00 01   .........4Z.....
0070  92 34 5a 95 f2 00 00 00 00 00 00 51 61 00 00 00   .4Z........Qa...
0080  00 00 00 00 0f 06 00 00 00 00 00 00 00 00 00 20   ...............
0090  01 06 7c 15 62 00 00 00 00 00 00 00 00 00 22 20   ..|.b........."
00a0  01 04 68 0c 00 ff c4 f8 16 3e ff fe 0f 26 3e 00   ..h......>...&>.
00b0  01 bb 81 a2 1b 06 00 00                           ........
```

Figure 30: Result of IPIFX live experiment 3

4.3.4 Collector

Objective: the objective of these last experiments was to test the tool's capability to capture and export network flow data using the NetFlow protocol, evaluating pmacctd's ability to support and export flows in NetFlow format, enabling integration with NetFlow-compatible collectors.

In this final experiment, we will send the flows generated by pmacctd to a collector. To accomplish this, we will utilize the NetFlow Python library, which can be found online and will be installed on node n1.

Additionally, to automate the process, it will be necessary to install Python3 and pip on the node to ensure the library functions correctly. We also need to install the `netflow` Python package for proper operation.

For this experiment, we will use the following updated version of the `nfprobe` plugin:

nfprobe_version: 9, which refers to NetFlow. Since we are using its Python implementation, specifying this version is essential.

This last experiment provided us with the most complete information on the flows generated in the data capture.

The results were obtained on a .tar file, containing a JSON file placed on the `netflow` folder as you can see on the following picture:

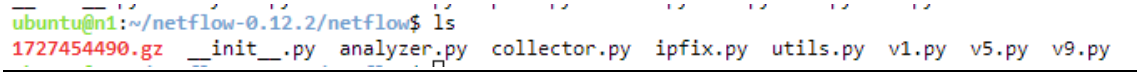

Figure 31: Folder which contains the .gz results of the IPIFX experiment 4

Once these results were extracted using the gunzip component, the results JSON file could be accessed. The following image shows the results of the first IPFIX stream.

We can see the great improvement in the usefulness of the information from the first experiment where the results were simply printed on the terminal to this one where a multitude of fields of the flow information are specified.

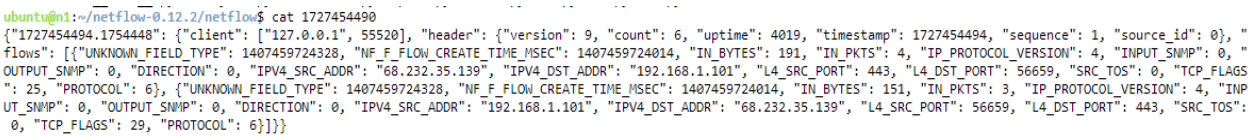

Figure 32: Result example of IPIFX .pcap experiment 4

The result shows the following fields of two flows. The first flow shows traffic from the server to the client, and the second shows the client's response.

Client Information:

- IP Address: 127.0.0.1
- Port: 55520

Header:

- Version: 9 (This indicates the use of the NetFlow Version 9 protocol, which allows for flexible and extensible flow information export.)
- Flow Count: 6
- Uptime: 4019 seconds
- Timestamp: 1727454494
- Sequence: 1
- Source ID: 0

Flow Details:

- Flow 1:
    - Incoming Bytes: 191
    - Incoming Packets: 4
    - IP Protocol Version: 4
    - Source IP Address: 68.232.35.139
    - Destination IP Address: 192.168.1.101
    - Source Port: 443
    - Destination Port: 56659
    - TCP Flags: 25 (this represents a combination of TCP flags, including SYN and ACK flags).
    - Protocol: 6 (this refers to the Transmission Control Protocol (TCP).

- Flow 2:

    - Incoming Bytes: 151
    - Incoming Packets: 3
    - IP Protocol Version: 4
    - Source IP Address: 192.168.1.101
    - Destination IP Address: 68.232.35.139
    - Source Port: 56659
    - Destination Port: 443
    - TCP Flags: 29 (This indicates that multiple TCP flags were set, such as PSH and ACK)
    - Protocol: 6 (Indicating that this flow also uses the TCP protocol.)

This experiment has been also tested using live traffic instead of using a .pcap file.

Figure 33: Results example of IPIFX live experiment 4

5 Results and Conclusions

Throughout this master thesis process, I have successfully achieved nearly all the experiments proposed by my supervisor. In addition, pmacctd has been proven to be a powerful tool with excellent functionalities.

However, the tool has very extensive documentation, as it has a multitude of different capabilities. In addition, this documentation contained some examples of use but very specific in the use of certain technologies such as Kafka. The examples and information about this tool on the Internet are also very limited. So, this turned out to be a longer learning curve, using trial and error and more time spent configuring the tool for each specific experiment.

The limited number of online examples tended to concentrate on very specific features. Considering this, I think that creating a comprehensive Jupyter Notebook, containing a variety of examples ranging from basic to more advanced, and explaining each step clearly, could be considered as a valuable outcome of this project.

For example, the progression from steps 1 to 4 in the IPFIX experiments highlights the importance of thorough research. Initially, the results were limited to an extensive and disorganized terminal output. However, by the final step, the output was well-structured, providing high-quality, actionable information about the packets. This demonstrates the necessity of deep research and understanding for better results, especially when working with complex and evolving tools like pmacctd.

This project has considerably enhanced my knowledge of both networking and network-related programming in Python. For instance, I have gained a deep understanding of how to use Python to create and manage different network topologies with the use of FABRIC. At the beginning of this project, I had very little knowledge of networking, but through this work, I have become familiar with several tools, libraries, and components that are widely used in both academic and professional settings such as NetFlow, IPFIX or iPerf.

As for the general conclusions of the project, I consider that both tools are very interesting to use separately. But I think that being able to use pmacctd in an environment that can simulate a real one thanks to the extensive topologies that can be created in FABRIC, is very interesting, since both are very customizable.

However, this high degree of customization limits the number of users who can use them, as significant prior knowledge is required. This makes it even more interesting to direct the project toward education, as it can provide a more dynamic approach to learning both Python programming for creating network topologies in FABRIC, as well as gaining a deeper understanding of networking itself.

6 Future Work

This master thesis focuses on the integration of pmacctd with FABRIC, but this is not its ultimate goal. The aim is to create an opportunity for future students and individuals interested in computer science, especially in the areas of software engineering and networking, to experiment and learn in a dynamic way by using the combined capabilities of these two tools.

Therefore, this notebook can be expanded, given that the tool is open source and is continually evolving. All these experiments can be enhanced by creating more extensive network architectures using FABRIC, combining tools for other fields such as cybersecurity or simply exploring the numerous functionalities that pmacctd offers, including for example, with its connection to relational databases.

In the future, a more extensive tutorial could be developed to cover advanced use cases and experiment variations, allowing users to not only learn network monitoring but also apply it to other real-world scenarios, like for example detecting network anomalies. This could involve integrating pmacctd with machine learning tools to develop predictive models based on network traffic, which could be particularly useful for research in areas like security analysis, or resource optimization.

To make this possible, both my co-supervisor, Professor Nik Sultana, and I contacted the FABRIC testbed administrators to include this Jupyter Notebook in the official repository of FABRIC usage examples. This process was successful, allowing anyone with a FABRIC account and license to access, execute, and modify it [16].

The repository containing the notebook used throughout this project can be found at the official [FABRIC GitHub repository].

Below is a screenshot of the repository:

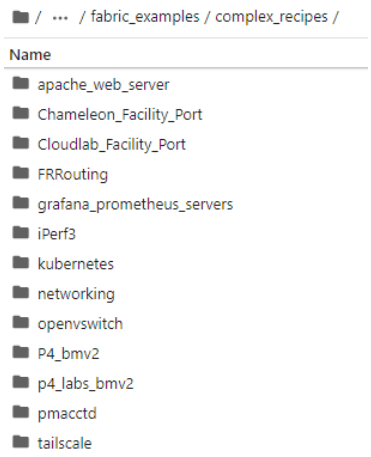

Figure 34: Official frabric_examples repository

7 References

[1] [Online]. Available: https://portal.fabric-testbed.net/.

[2] P. Lucente, "pmacct project," [Online]. Available: http://www.pmacct.net/.

[3] "About FABRIC," [Online]. Available: https://portal.fabric-testbed.net/about/about-fabric.

[4] JuoyterHub, [Online]. Available: https://jupyter.org/hub.

[5] FABRIc, «FablibManager,» [En línea]. Available: https://learn.fabric-testbed.net/docs/fablib/fablib.html.

[6] Java Point, «Ubuntu Build Essential,» [En línea]. Available: https://www.javatpoint.com/ubuntu-build-essential.

[7] Debian, «Package: net-tools (2.10-1.1 and others),» [En línea]. Available: https://packages.debian.org/sid/net-tools.

[8] Scapy, [En línea]. Available: https://scapy.net/.

[9] WIRESHARK, [En línea]. Available: https://www.wireshark.org/.

[10] «TCPDUMP &LIBPCAP,» [En línea]. Available: https://www.tcpdump.org/.

[11] endance, «What is a PCAP file?,» [En línea]. Available: https://www.endace.com/learn/what-is-a-pcap-file.

[12] «iPerf - The ultimate speed test tool for TCP, UDP and SCTP,» [En línea]. Available: https://iperf.fr/.

[13] IBM, «IPFIX overview,» [En línea]. Available: https://www.ibm.com/docs/en/npi/1.3.1?topic=insight-ipfix-overview.

[14] S. Kumarsamy, «IP Flow Information Export (IPFIX) vs. NetFlow,» 17 September 2019. [En línea]. Available: https://blog.gigamon.com/2019/09/17/ipfix-vs-netflow/.

[15] GeeksforGeeks, «Introduction to Netcat,» [En línea]. Available: https://www.geeksforgeeks.org/introduction-to-netcat/.

[16] «FABRIC Testbed,» github, [En línea]. Available: https://github.com/fabric-testbed.

8 Annexes

This chapter shows some examples of configuration files used for each experiment described in this document.

Configuration File 1:

This configuration file was designed for performing a customized capture in PCAP format, using the file gmail.pcapng.cap as input and exporting the results in CSV format. The configuration aggregates multiple fields of interest, enabling customized analysis.

```
daemonize: false
debug: true
pcap_savefile: gmail.pcapng.cap
aggregate: src_host, dst_host, src_port, dst_port, proto, src_mac, dst_mac,
etype, in_iface, out_iface
plugin_buffer_size: 4096
plugins: print
print_output_file: result_custom.csv
print_history_roundoff: m
```

Configuration File 2:

This configuration file was designed for capturing PCAP format data from the file gmail.pcapng.cap, applying a filter to extract only TCP packets, and exporting the results in JSON format.

```
daemonize: false
debug: true
pcap_savefile: gmail.pcapng.cap
pcap_filter: tcp
aggregate: src_host, dst_host, src_port, dst_port, proto,
plugin_buffer_size: 4096
plugins: print
print_output: csv
print_output_file: result_pcap_filter.csv
print_history_roundoff: m
```

Configuration File 3:

This configuration file was developed for performing a live traffic capture, applying a filter to extract only ICMPv6 packets with a specific source host (fd3f:f209:c712::2), and exporting the results in CSV format.

```
daemonize: false
debug: true
pcap_interface: any
pcap_filter: icmp6 and src host fd3f:f209:c712::2
aggregate: src_port, dst_port, etype, in_iface, out_iface,
plugin_buffer_size: 4096
plugins: print
print_output: csv
print_output_file: result_live_filter.csv
print_history_roundoff: m
```

Configuration File 4:

This configuration file is designed to perform a capture using the gmail.pcapng.cap file as input. The configuration uses the plugin nfprobe for generating IPFIX flows. The output is exported to a log file named ipfix_2.log, and the IPFIX version is set to 10 for compatibility with specific flow collectors.

```
daemonize: false
debug: true
pcap_savefile: gmail.pcapng.cap
plugins: nfprobe, print
nfprobe_version: 10
print_output_file: ipfix_2.log
```

Configuration File 5:

This configuration file is designed to perform a capture using the gmail.pcapng.cap file as input file and sending them to a specified receiver. The nfprobe plugin is used to generate and export IPFIX flow records to a collector located at 127.0.0.9 on port 4738. The IPFIX version is set to 10 for compatibility with modern flow analysis tools.

```
daemonize: false
debug: true
pcap_savefile: gmail.pcapng.cap
plugins: nfprobe
nfprobe_receiver: 127.0.0.9:4738
nfprobe_version: 10
```

Configuration File 6:

This configuration file is designed to generate IPFIX flows from the gmail.pcapng.cap file and export them using the nfprobe plugin. The IPFIX flows are sent to a collector at 127.0.0.14 on port 4739, using version 9 of the IPFIX protocol.

```
daemonize: false
debug: true
pcap_savefile: gmail.pcapng.cap
plugins: nfprobe
nfprobe_receiver: 127.0.0.14:4739
nfprobe_version: 9
```

Configuration File 7:

This configuration file is configured for live traffic capture on interface `enp3s0`. It uses both the nfprobe plugin for IPFIX flow generation and the print plugin for logging the results. The IPFIX flows are exported to a collector on localhost at port 4739, using version 10.

```
daemonize: false
debug: true
pcap_interface: enp3s0
plugins: nfprobe, print
nfprobe_version: 10
nfprobe_receiver: localhost:4739
```

Configuration File 8:

This configuration file is configured for live traffic capture on all network interfaces. It uses the nfprobe plugin to generate IPFIX flows, exporting them to a collector at 127.0.0.4 on port 4738, using IPFIX version 10.

```
daemonize: false
debug: true
pcap_interface: any
plugins: nfprobe
nfprobe_receiver: 127.0.0.4:4738
nfprobe_version: 10
```

Configuration File 9:

This configuration file is also set up for live traffic capture on all network interfaces. It uses the nfprobe plugin to generate IPFIX flows and sends them to a collector at 127.0.0.16 on port 4739, using version 9 of the IPFIX protocol.

```
daemonize: false
debug: true
pcap_interface: any
plugins: nfprobe
nfprobe_receiver: 127.0.0.16:4739
nfprobe_version: 9
```