CS 1980 - Computer Science Capstone Final Report

Programming a Linux Kernel Module to Control Network Traffic

Ivan Bondarenko
Madeline Powers
Gennadi Ryan

Sponsor: Dr. Luis Oliveira

Abstract

Our project implements Time Division Multiple Access (TDMA) technology by introducing a dynamic and distributed variant tailored for the Linux Kernel. By leveraging a custom Kernel Module and the *NetController* (netcntlr) userspace program, our system dynamically adjusts network parameters to accommodate changing conditions and user demands based on TDMA. We utilize NETLINK for robust inter-process communication, allowing precise control and real-time adjustments through our TDMA Scheduler. This setup maintains the flexibility of the Linux Kernel while providing advanced network traffic management capabilities directly from the Kernel.

Table of Contents

Background

The Linux Kernel

A kernel is the core component of a computer's operating system, and it acts as the intermediary between the hardware components of the computer and the software it runs. By design, the kernel is very restrictive to general users of a computer and must be carefully communicated to, so as to not disrupt any vital processes or endanger any of the hardware components on the system. To work on this project, we had to simulate a Linux machine running a custom kernel that we could modify. This was achieved by using *qemu*, an open source machine emulator and virtualizer, as well as *VirtualBox*, and building Linux Kernel version 6.7.4, which was the latest release at the time we started development. It follows then that a kernel module in Linux lets a user expand the capabilities of their current kernel, without risk of damaging the underlying system or having to recompile the kernel to include changes and new behavior. Figure 1 below demonstrates how a kernel module can communicate with the system's kernel, through the use of 'The Kernel Core' API provided by Linux.

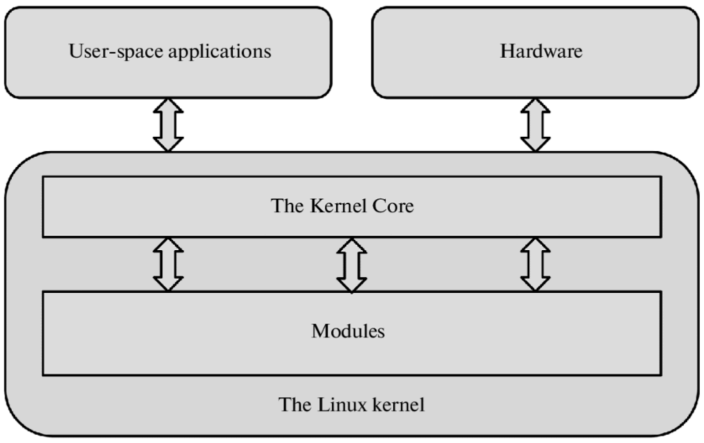

Figure 1. Linux Kernel Module Interaction

Inter-Process Communication (IPC)

Through the use of NETLINK, which is a socket family used for IPC between the Kernel and userspace processes, we can safely and efficiently send messages to the Kernel to interact with the networking hardware on our machines to implement our TDMA Scheduler. By using a high level library such as *libnl* as shown in the diagram below (Figure 2), we can pass NETLINK messages to/from the Kernel to our userspace program, *NetController* (netcntlr).

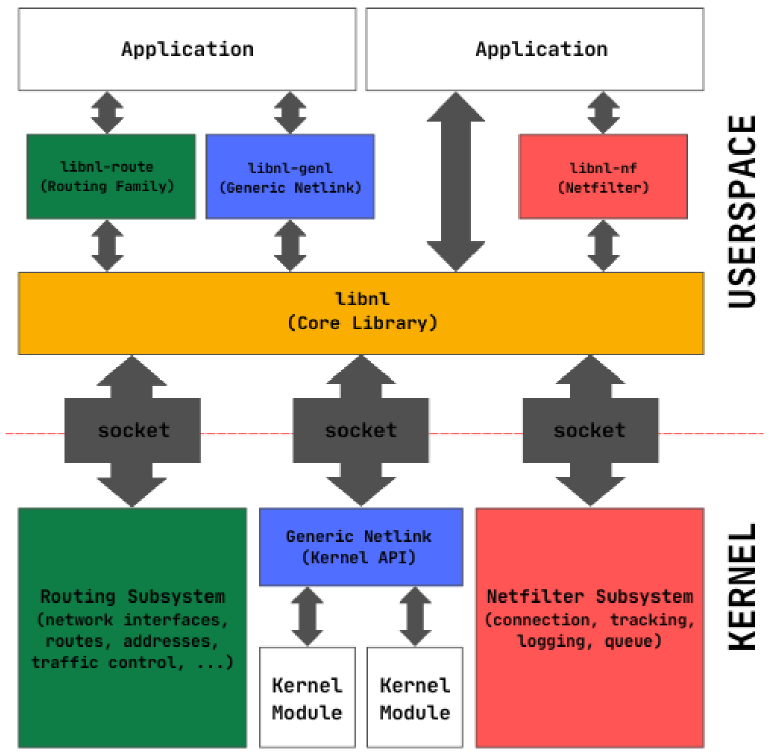
Application
Application
libnl-route (Routing Family)
libnl-genl (Generic Netlink)
libnl-nf (Netfilter)
libnl (Core Library)
socket
socket
socket
Generic Netlink (Kernel API)
Routing Subsystem (network interfaces, routes, addresses, traffic control, ...)
Kernel Module
Kernel Module
Netfilter Subsystem (connection, tracking, logging, queue)
USERSPACE
KERNEL

Figure 2. NETLINK (libnl) Interaction with Linux Kernel

Unlike other IPC approaches such as *ioctl* or BSD sockets that use defined protocols such as TCP (RFC 793), the caller (our team) has to manually create NETLINK messages, where the structure of the message is as shown below (Figure 3), and operations that handle these messages when interfacing with the socket.

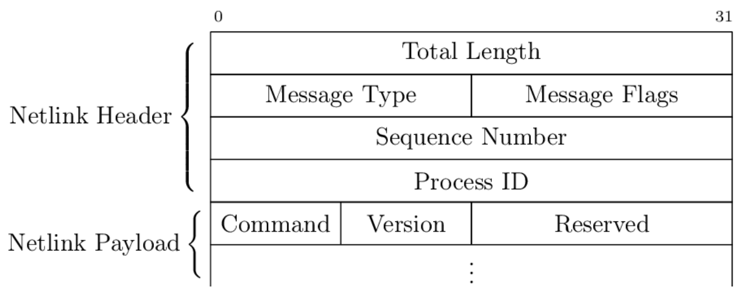

Figure 3. NETLINK Message Structure

Time Division Multiple Access (TDMA)

TDMA is a technology used in networking and telecommunications to facilitate the shared use of a single radio frequency channel among multiple users (or hosts) by dividing the signal into different time slots. In our case, we are building a distributed and dynamic version of TDMA. This means that the machines dynamically adjust their own parameters to account for network conditions, new machines joining or dropping out, as well being adjusted manually by a user through our userspace program, NetController (netcntlr). Figure 4 below shows TDMA in a single frequency channel dividing up each time slot (Frame) for the N users who need to access the network, as shown in the figure below.

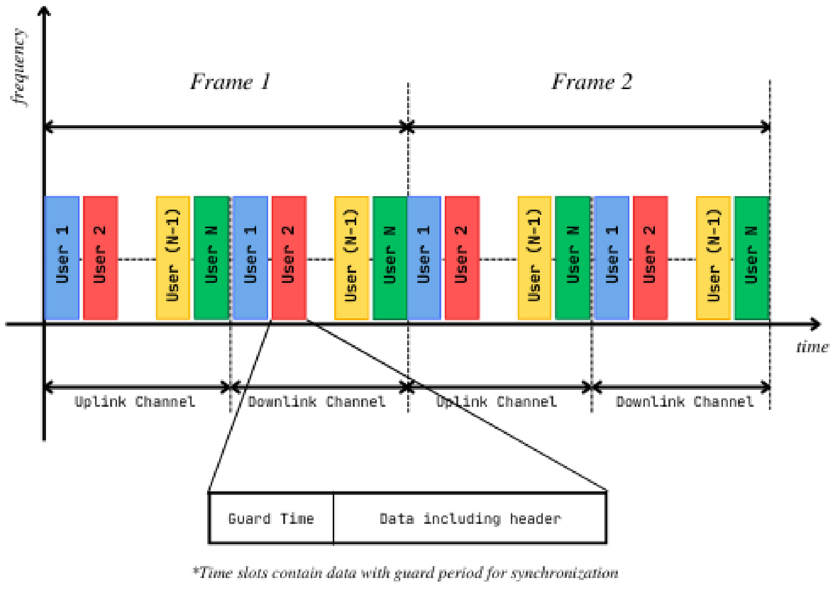

Figure 4. Time Division Multiple Access (TDMA) Diagram

While TDMA is a valuable technique in specific wireless communication and embedded systems contexts, it doesn't make much sense for the Linux Kernel to implement it due to Linux's general-purpose nature, flexibility, and portability design philosophies. As TDMA is generally implemented by higher layers of the network stack by specifically designed hardware and requires more careful and niche mechanisms, it makes sense to instead implement it as an extension to the Kernel as we have done here, through our custom Kernel Module and userspace application.

Linux Traffic Control (tc) and Quality of Service (QoS)

Linux Quality of Service refers to the Linux Kernel's built-in facilities for scheduling outgoing and incoming network packets. The main unit of QoS is the queueing discipline (QDisc), which is a kernel module/internal component responsible for throttling, or "shaping", network traffic through a variety of strategies. The core TDMA module is implemented as a custom QDisc, meaning it may be swapped seamlessly with default/built-in QDiscs. This is one of the first implementations of TDMA as a core Kernel QDisc, rather than as a hardware-bound network device driver. This allows the TDMA QDisc to interoperate with well-known Linux utilities such as *iproute2* and *tc*, allowing experienced network administrators and researchers to configure, experiment with, and modify the QDisc without making modifications to the kernel itself. Since our use case (dynamic TDMA) requires real-time updates to the QDisc parameters, tc alone is not an ideal mode for interacting with the QDisc. Nonetheless, many modern use cases call only for static TDMA, in which case tc is sufficient. Moreover, future work in this space may show that dynamic TDMA can be achieved through packet classification (e.g. via netfilter), an existing and widely supported feature of the Linux QoS stack.

Project Design & Methodology

Kernel Configuration

For this project, the team determined that in order to get the full extent of functionality that our kernel module and userspace program required, we would have to compile the kernel and set some additional configurations that are not enabled by default in standard Linux distributions. Similarly, due to the many hours spent testing different distributions, changing virtualization parameters, using different kernel configuration options, and recompiling, we wanted to preserve our original distribution's kernel as a safety net that we could reliably fall back on. In the future, other teams may aim to make this kernel module "kernel configuration agnostic", or have the module set the configurations it needs when loading it into the kernel during runtime, but for now, our approach is tested and reliable.

To build (compile) our custom kernel, we didn't need to change much from the provided distribution's kernel configuration. We had to set some options like "CONFIG_LOCALVERSION" to "-arch-1980" for distinguishing our custom kernel, "CONFIG_KERNEL_LZ4" to use LZ4 compression for more efficient compilation, turning off "CONFIG_DEBUG_INFO_BTF" (where BTF (BPF Type Format) which is the metadata format which encodes the debug info related to BPF program/map), and additionally enabling NETLINK/NETFILTER options for using *libnl* and GENERIC NETLINK. Furthermore, there are specific commands which a user can invoke when generating the kernel configuration, such as `make kvm_guest`, which copies the current machine's (the virtual machine) settings and configurations relevant to machine virtualization to the new configuration to ensure maximum compatibility with *qemu* and *VirtualBox*. Similarly, `make localmodconfig` copies the virtual machine's configuration to determine what drivers need to be built. These configurations and commands were used to build our final kernel configuration, and it has been running our module and userspace program reliably.

To use our kernel configuration, the team chose to use different Linux distributions ('flavors' of the Linux Operating System) which were emulated using *qemu* and *VirtualBox*, on the 3 main operating systems as host machines (Windows, MacOS, and Linux). While the choice of Linux distribution doesn't particularly matter since all distributions inherit essentially the same underlying kernel (with distribution specific configurations set), we wanted to ensure maximum compatibility for end users who would be using our project in the future and therefore landed on using Arch Linux x86_64 as a 'bare-bones' Linux system, and Debian for its popularity, robustness, and ease-of-use. Similarly, using both *qemu* and *VirtualBox* ensured that our build process and implementations were correct and could reliably be reproduced on whatever virtualization software the end-user is most comfortable with.

IPC Mechanism (NETLINK)

Another big design consideration for this project was the selection of an IPC mechanism to communicate with the kernel from our module and userspace program. While Linux provides many such mechanisms for this including signals, pipes, message queues, semaphores, shared memory, etc. we found that NETLINK sockets (and its higher-level libraries and subsidiaries) provided the best approach to meet our goals of efficiency, ease of programming / implementability, and design, given our time constraints. Specifically what stood out about using NETLINK was that it supports sending multicast (multiple processes can be notified), asynchronous (non-blocking, doesn't need to wait for a response), complex, structured data through a socket, and could be easily used with userspace programs, like our program *NetController* (netcntlr).

Userspace Programs

For this project, our sponsor had tasked us with two main goals outside of the kernel: creating a userspace program to handle interacting with the TDMA module, and having a graph visualization program that would show network packets and statistics in real-time going through our TDMA Scheduler. For our main application, *NetController* (netcntlr), we determined that this program should be built with the C programming language and compiled with GCC, to ensure maximum performance and compatibility with the different libraries and protocols that we needed to achieve our first goal. For creating the graph visualization program, we spent a lot of time researching available methods and implementations, and finally landed on using *matplotlib* with Python to handle the charting. We chose these technologies as it seemed to be the most straightforward approach and because we don't have many of the same strict regulations to consider as *NetController* (netcntlr).

Project Implementation

TDMA Kernel Module

The TDMA kernel module was implemented as a QDisc (queueing discipline), a configurable part of the kernel which is responsible for the scheduling of network packets, both incoming and outgoing. The kernel defines an API (net/sched/sch_api.c) that determines the structure of a valid QDisc, consisting of a specification struct that associates the QDisc's id with the QDisc's "*_sched_data" (where '*' represents a wildcard prefix) struct and its handler methods. The "*_sched_data" struct stores private variables representing the QDisc's current state, including the interface/network device it is associated with, statistics (such as number of packets/bytes transmitted/dropped), and any other internal state the QDisc needs to maintain (such as timers, rate/size tables, etc.). There are also a variety of possible handler methods, depending on the type of QDisc in question; classful QDiscs for instance (which separate packets into queues based on the "flow" they belong to or some other measure or priority) require additional handling methods corresponding to the QDisc's class/queue mapping and the state of any sub-QDiscs.

Nearly all QDiscs (whether classful or classless), including ours, implement the following five handlers: init, change, enqueue, dequeue, and destroy. Init and destroy are somewhat self-explanatory. Change, on the other hand, is responsible for processing new parameter values transmitted from userspace, and changing the state of the QDisc accordingly. The essential handlers required for all QDiscs, however, are enqueue and dequeue. When a packet is passed to a network device, the device forwards the packet directly to its root (top-level) QDisc's enqueue method, where the QDisc's internal logic determines whether to add the method to (one of) the device's transmit or receive queues, or whether to drop it. Additionally, at this stage, a QDisc may break up larger packets to ensure that they fit in the device's queue, and may record statistics regarding the packet, among other tasks. If the QDisc is the parent of a sub-QDisc, this means the packet will be filtered through sub-QDiscs as well where the same sort of tasks will be carried out. Lastly, once successfully enqueued, the device will acquire a lock and request that a packet be dequeued. Packets are not typically dropped at this stage; however, this stage is where the QDisc receives the opportunity to "shape" the network's traffic. This means the QDisc can enforce policies, such as choosing a packet selectively from among multiple queues according to a queue/priority mapping, or choosing not to dequeue a packet at all to enforce that all packets egress the network device at a particular rate (such as in TBF) or within a particular time slice (such as in our TDMA implementation). The parameters controlling how packets are enqueued and dequeued are stored in the "*_sched_data" struct and are meant to be modified by the users and network administrators from userspace or in some cases by the kernel itself (in both cases via the change method).

Due to the real-time nature of our implementation (requiring the TDMA egress QDisc to update its state dependending on the time at which certain packets are ingressed), minor parameter updates are continuously carried out automatically/programmatically, which sets this implementation apart from that of most other QDiscs. Nonetheless, larger, configuration-level updates are also made on an ad-hoc basis by users or network administrators to control everything from the size of the TDMA frame to the statistics tracked by the QDisc to the rate at which minor parameter updates are allowed to be made. These tasks are the domain of NetController. Additional exposition available at sch_api/sch_generic.

NetController (netcntlr)

NetController (netcntlr) is the main userspace program and entry-point for our project. As mentioned previously, this C program handles sending and receiving NETLINK messages that control our overall TDMA kernel module's configuration and parameters dynamically. It is a command-line program that allows users to manually configure these options by setting specific flags that are passed to the invocation of the program, or alternatively by using a configuration file that parses out these values. If desired (and set with the `--graph` flag), *NetController* (netcntlr) will also start saving summary statistics from our module, which allows our graph visualization program (graph.py) to plot the network activity under TDMA in real-time to a GUI plot.

The main technology behind both our userspace program and our kernel module is NETLINK. As shown in Figure 2, NETLINK can be used to configure many different aspects of the Linux Network stack through the use of its high-level library *libnl.* Using *libnl*, we are able to control network interfaces, routes, traffic control, transmission queues, etc. through sending specified messages from *NetController* (netcntlr). In our program, we had to define specific message types to handle updating parameters, using traffic control, start charting the statistics, etc. as well as different operations to handle exchanging different states between the userspace program and kernel.

Visual Overview

The figure below shows the flow of how our project works. The *NetController* (netcntlr) userspace program acts as the entry point for all of the TDMA scheduling and configuration during run-time. Once started, the open NETLINK socket will handle all IPC exchanges between the graph visualizer program (graph.py) and our TDMA kernel module. As you can see in Figure 5, the TDMA kernel module maintains many of core functionalities of the Linux Network Stack, however it does this in a safe and configurable way, ensuring end-users can test different implementations and configurations of TDMA network scheduling, without risk to damaging the underlying Linux Network infrastructure.

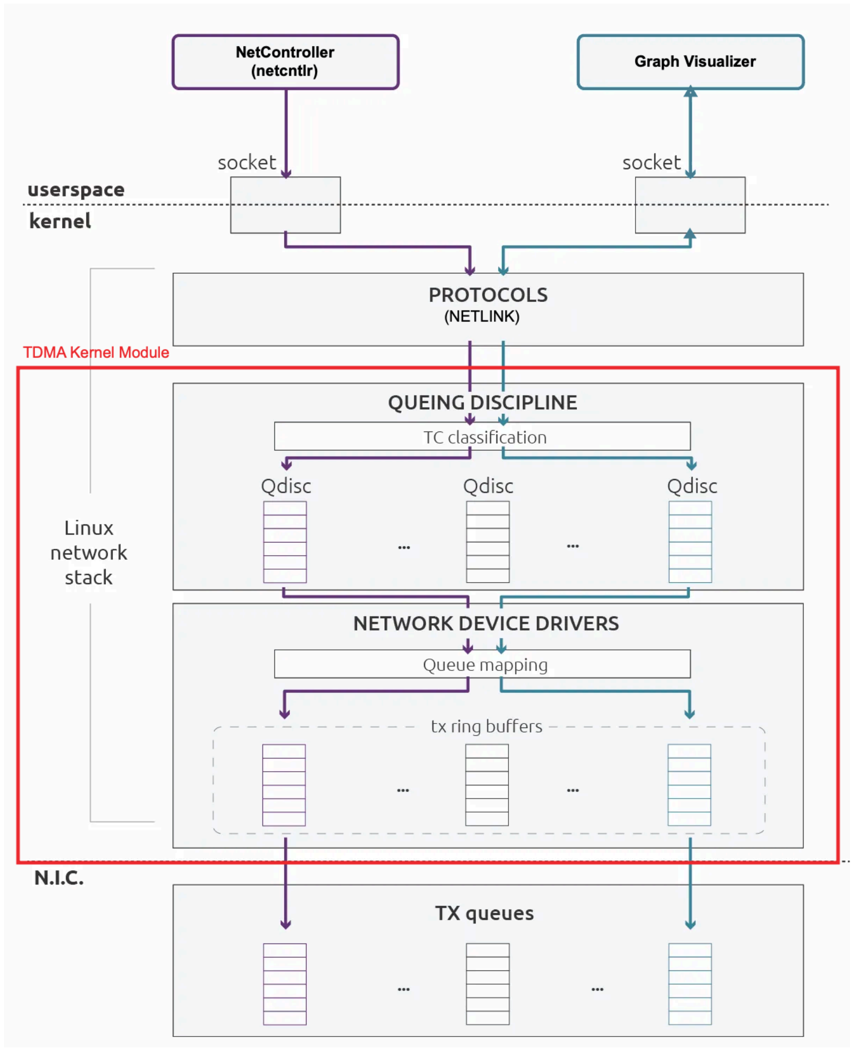

Figure 5. *NetController* (netcntlr) and TDMA Kernel Module Diagram

Retrospection

Challenges Faced

Some of the greatest challenges we faced were rather mundane, and related more to the general difficulties of kernel development rather than QDisc-specific challenges. In part due to using both qemu and VirtualBox in order to virtualize our development process, we faced difficulties in setting up a replicable environment to test the behavior of the QDisc when handling communication between >2 machines/network devices. An example of such a challenge is the need for a private network established between >2 identical machines. Since our development and testing process requires configuring the Linux kernel, this made the use of off-the-shelf containerization software (such as Docker) difficult if not impossible. Instead we aimed to achieve a similar result by manually configuring a handful of (initially) identical machines via qemu.

Once configuring a number of copies of the same machine in order to test our implementation, we faced difficulties configuring the associated private networks in order to allow the machines to communicate with one another. Qemu, in particular in its MacOS implementation, exports the ability to create virtual network devices through one of three protocols, all part of VMWare's vmnet framework for virtual machine networking.

The first of these protocols, vmnet-host, creates a shared private network on the host that is, without additional work, sealed off from the outside world (packets are not forwarded to the main network device on the host machine, but rather isolated unto themselves). Meanwhile, vmnet-shared creates a bridge network that, through NAT (network address translation), allows VMs to network as though they were just another process running directly on the host machine. However, vmnet-shared was found to be unsuitable (without quite a bit of additional work) for our purposes. When attempting to network between just two machines/network devices, it was found that ARP (the protocol mapping physical device addresses to IP addresses) failed to translate the address of one of the two machines, resulting in an apparent one-way flow of packets. This could not easily be corrected by configuring the machines' rules and routing tables.

Lastly, vmnet-bridged was found to be even more unsuitable for similar reasons, as vmnet-bridged is nothing more than a primitive version of vmnet-shared (which consists of several interconnected bridges), which would have required even more work to use in connecting two separate VMs together over a single bridge. Moreover, all of these methods suffered from the restrictions MacOS imposes on setting the DHCP configuration (assignment of local "192.*"-style IP addresses) of the virtual network devices. Ultimately, we found that vmnet-host, despite its deficiencies, was the most suitable in that it allowed us to network between more than two machines with ease and with little additional configuration (due to the machines sharing a common default gateway), even though this compromised the machines' ability to forward packets outside their local private network.

The last big challenge we faced was regarding using the NETLINK protocol to communicate with the Linux Kernel. Originally, we wanted to keep things simple and use the CLASSIC NETLINK approach and communicate via sockets to the kernel, but this seemed to cause issues when trying to deserialize the data structures when back in the TDMA module. As the userspace program and Linux Kernel are using different compilation methods and different compilation flags or options, the main variable configuration structure that we were sending through the NETLINK socket was being manipulated differently than expected. When C code is compiled, the compiler (usually GCC) will try to optimize the code to the best format that the CPU understands. This may include splitting up a large integer field (like a uint64_t, or unsigned integer of 64 bits) into multiple smaller integer types within your C struct for optimization. This proves troublesome when sending the structure over a socket because when deserializing (or getting the original data back) our defined fields may point to incorrect regions resulting in incorrect results. To combat this issue, we had to switch to the NETLINK Protocol Library Suite (libnl), which defines a different format for sending data types and structures over the NETLINK API, as well as handling the correct serialization/deserialization of the data. This involved a lot of code refactoring to get right, but in the end we were able to achieve our goal of using NETLINK to send a message to the Kernel to change our TDMA configuration during run-time.

Conclusion

Our capstone project represents a great achievement in network traffic management by integrating TDMA directly into the Linux kernel. Our solution demonstrates that dynamic TDMA can be effectively implemented in a general-purpose operating system without compromising system stability or performance. This project has laid a solid foundation for future exploration and development in this area, potentially influencing how network traffic is managed on Linux systems. Further research could explore the integration of this TDMA system with existing network management tools and expand its functionality to support more complex network environments. Additionally, the lessons learned from the challenges faced can guide future projects in this domain, contributing to the broader field of networking and Linux Kernel software development. Our project not only meets the technical requirements set out by our sponsor but also provides a platform for ongoing innovation in network traffic control within the Linux ecosystem.