Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis

Deployment and Empirical Verification of Real-time Schedules

Mohammed Asjadulla

Course of Study:	Information Technology (INFOTECH)
Examiner:	Prof. Dr. Christian Becker
Supervisor:	Heiko Geppert
Commenced:	July 18, 2023
Completed:	January 18, 2024

Abstract

The rapid evolution of the Internet of Things (IoT) and the increasing prevalence of interconnected devices have significantly expanded the challenges in real-time systems, particularly concerning traffic planning for Time-Sensitive Networks (TSN). These advancements necessitate innovative solutions to ensure Quality of Service (QoS) under dynamic conditions. A critical step in addressing these challenges involves developing innovative scheduling approaches that must be thoroughly verified beyond theoretical models to ensure their practical applicability. Verifying these schedules requires creating and assessing a virtual testbed as it replicates the intricate behavior of real-time systems in a setting that offers control, flexibility, and scalability. Such a virtual environment is essential for fine-tuning traffic scheduling strategies, ensuring their effectiveness and compliance with stringent real-time constraints before implementation in actual operational contexts.

We utilize Docker containers and Linux networking functionalities to emulate real-world network scenarios to analyze traffic plans generated by TSN scheduling algorithms, particularly the Greedy Flow Heap (GFH) algorithm. We also develop a comprehensive software framework within this virtual environment that transforms theoretical scheduling concepts into practical, executable traffic flows, simplifying the intricacies associated with network experiment setups. In the setup, the Earliest TxTime First (ETF) queuing discipline (qdisc) is implemented, enabling the scheduler to replicate the timing precision necessary for real-time schedules. The delta parameter within the ETF qdisc is important as it determines the buffer time before a packet's scheduled transmission time. It acts as a "fudge factor", allowing us to accommodate the inherent latencies of a virtual environment.

We identified four key performance indicators for our experiment: frame drops, frame transmission accuracy, processing delay, and end-to-end (E2E) latency. Our results reveal a notable increase in frame loss with reduced ETF qdisc delta values, and we found an optimal delta value of $5*10^7$ ns. The empirical tests demonstrated a requisite initial stabilization period for the system, after which frame transmission accuracy achieved a minimum value of 20 µs. The research also highlighted the system's scalability, with larger network topologies showing decreased processing times due to efficient traffic distribution. The importance of network topology in influencing E2E latency is also evident, particularly concerning the number of switches a frame traverses. The observed deviation in frame transmission precision, ranging from 20 µs at best to 10 ms at worst, and the processing delays at switches reaching 25 ms in some instances, suggest that an emulation-based validation of the GFH algorithm has its limitations.

Contents

List of Figures

Listings

1 Introduction

In the modern landscape of computational and networking technologies, the idea of real-time systems has expanded from a specialized area to an essential field of research and application. Crucial to the digital transformation across various sectors, from autonomous vehicles to intelligent healthcare, real-time systems play a significant role in linking innovation with execution [CP17].

Real-time systems were traditionally confined to avionics, defense, and specific high-performance industrial applications [KS22b]. These systems require strict timing guarantees and were limited by the computational capabilities and networking paradigms of the past. However, the Internet of Things (IoT) and the growing pervasiveness of interconnected devices have significantly broadened the scope of real-time systems [SSH+18] [REC15]. Today, they support essential military operations and everyday systems like traffic control, augmented reality technology, and advanced manufacturing units.

This shift in paradigm presents numerous challenges, and the complexities of real-time systems have been compounded by the need for dynamic traffic planning that considers Quality of Service (QoS) requirements for time-triggered flows. It has historically been arduous to ensure latency and jitter bounds for traffic plans that span a network, especially in cases with rapidly changing conditions [KS22b].

Efficient traffic planning is indispensable in this context, as it ensures that time-sensitive data is accorded priority, thereby averting potential network congestion, latency spikes, or data loss scenarios [SCO18]. Such robust planning is especially critical in networks that simultaneously manage time-sensitive and non-time-sensitive data, necessitating strategies that optimize critical data's priority and overall network utilization.

Traditional techniques often face high computational overhead, the potential for reduced network utilization, or the inability to provide guarantees during traffic plan transitions. This has required developing innovative approaches to reconfigure active flows adaptively, optimizing network usage while maintaining strict QoS guarantees [FGD+22]. These advancements highlight the urgent need to bridge the gap between theoretical network structures and actual algorithmic solutions for real-time applications. Instead of relying solely on static configurations, which may lead to resource misutilization, dynamically adjusting to the current network conditions and application requirements is crucial. When implemented through middleware solutions, this dynamic approach optimizes resource utilization and ensures that applications function seamlessly without modifying their inherent structure.

The growing significance of time-critical networked systems, particularly in areas like manufacturing, which fall under the umbrella of the Industrial Internet of Things (IIoT) [KS22a], automotive industries [REC15], and other cyber-physical systems [SSH+18], reinforces this perspective. These environments, in which physical processes merge seamlessly with networked sensors, actuators, and controllers, require precision that modern systems may find challenging to achieve, and ensuring

deterministic transmissions in such environments becomes paramount. Deterministic transmission ensures that messages are delivered within a predictable time frame, and deviation from that can potentially result in system malfunctions, especially in dynamic network topologies, such as mobile robotic systems or vehicular networks [LWP+19].

The Time-Sensitive Network (TSN) standards, introduced by IEEE, represent a significant step towards addressing the challenges posed by the demand for deterministic and real-time communication [COCS16], especially in industrial and vehicular networks [BS19]. Realizing the full potential of TSN has its challenges. Modern networks' increasing complexity and heterogeneity demands innovative solutions to ensure real-time communication. One of the primary challenges lies in scheduling. Efficient scheduling algorithms are vital to guarantee that time-sensitive data is transmitted and received with strict timing constraints [COCS16]. These algorithms need to be dynamic, adjusting to the ever-changing conditions of the network and the varying traffic demands [Mes18].

Furthermore, while the focus on real-time requirements is undeniably essential, ensuring fairness in the network is equally crucial. All connected devices should have equitable access to network resources regardless of their role or importance. Balancing this fairness with efficiency, especially in a network governed by TSN standards, requires a nuanced approach and comprehensive understanding [COCS16] [Fin18].

Although the IEEE standard for TSN establishes the technological and methodological infrastructure necessary for real-time network operations, it omits specifying the schedule computation, allowing vendors and users to implement proprietary solutions that align with their unique operational requirements. This has catalyzed an upsurge in specialized research, with academic and industry experts striving to address the multifaceted problem of real-time scheduling in a dynamic digital environment.

This thesis endeavors to advance the automated deployment of real-time schedules within virtual environments, specifically focusing on developing and evaluating a virtual testbed to analyze traffic plans generated by TSN scheduling algorithms. The motivation for using such a virtual testbed stems from the growing necessity to emulate and test complex real-time systems in a controlled, flexible, and scalable manner before their deployment in the physical world. Virtual testbeds offer an invaluable platform for researchers and engineers to meticulously test and refine scheduling strategies, ensuring robustness, efficiency, and compliance with real-time constraints before real-world implementation. By employing the features of Docker containers and the networking functionalities available in the Linux kernel, the virtual testbed provides a versatile emulation environment representative of real-world conditions. The virtualized setting allows for the isolation of variables, precise control over network conditions, and the ability to consistently replicate network experiments, which is an often challenging prospect in physical testbeds.

We create a robust software framework for deploying real-time schedules within virtual environments using a structured pipeline that transforms abstract scheduling algorithms into tangible traffic flows and behaviors within the virtual testbed. This end-to-end pipeline, designed and implemented as part of the thesis, enables the automation of deployment processes, thereby reducing the complexity and time traditionally associated with the setup and execution of network experiments.

Another contribution of this work is the evaluation of the individual components comprising the virtual testbed. This research meticulously assesses each element's suitability for real-time networking applications through empirical tests and examining key performance indicators. The

evaluation extends beyond functional validation and encompasses a rigorous performance analysis under various scenarios that emulate realistic operational conditions. The findings provide valuable insights into optimizing virtual environments for real-time network testing. Finally, the comprehensive approach adopted in this research aims to bridge the gap between theory and practice, providing a foundation for future developments in real-time network scheduling and virtual testing platforms.

2 Preliminaries and Definitions

The interaction of Real-Time Systems (RTS) with networking in Linux environments and the integration of Docker Containers exemplify a dynamic interplay of complex constructs, each with its unique terminology and theoretical framework. This chapter will define and explain the key constructs relevant to our exploration to ensure a clear and concise understanding of the following discussion. Beyond the foundational exploration of RTS and Networking, this chapter will extend its analytical lens to the nuances of Linux Networking. Additionally, this chapter will examine the domain of Docker Containers.

2.1 Real-Time Systems and Networking

Real-time systems are computing systems designed to respond to input or environmental changes within a finite and specified period, often referred to as deadlines. They are crucial in various safety-critical applications, including automotive control systems, medical devices, industrial automation, and aerospace applications.

2.1.1 Characteristics of Real-Time Systems

Real-time systems have distinct features that differentiate them from non-real-time systems [KS22c]. They must be predictable and ensure fixed response times to critical events. This entails ascertaining the worst-case execution time (WCET) of tasks and ensuring the system can meet these deadlines under all feasible conditions. They are also deterministic, meaning they will always produce the same output for a given input. This is vital for applications requiring consistent performance, irrespective of how often a task is executed. Lastly, real-time systems frequently operate alongside purpose-built hardware to meet their timing constraints. Such hardware frequently consists of dedicated processors, sensors, actuators, and communication interfaces.

2.1.2 Classification of Real-Time Systems

Real-time systems are generally classified into two categories: hard real-time systems and soft real-time systems. Hard real-time systems are characterized by their absolute adherence to deadlines, where any delay or deviation can cause catastrophic results. The implications could range from property damage to endangering human lives. Common manifestations of hard real-time systems include aircraft control software, pacemakers, and nuclear reactor control mechanisms. Soft real-time systems, while still prioritizing deadlines, offer a degree of flexibility. In these systems,

occasional delays, although not ideal, would not lead to drastic consequences. They are prevalent in applications such as video streaming and VoIP, where a few milliseconds of lag might be acceptable and imperceptible to users [KS22c].

2.1.3 Networking in Real-Time Systems

Networking in real-time systems involves transmitting and receiving data packets between interconnected devices within a specified time frame. This is crucial for maintaining the determinism and predictability required by real-time applications.

Network Topologies

Network topologies dictate the structural layout of how different nodes in a network communicate. Real-time systems, depending on their requirements, might employ various topologies. The ring topology, for instance, offers a continuous looped pathway, ensuring data transmission even if one link fails, highlighting its fault-tolerant nature. In contrast, the star topology positions a central node as the primary communication hub, effectively managing and routing data among numerous peripheral nodes. This centralization facilitates easy monitoring and data routing. Meanwhile, the bus topology, which consists of a primary communication line where all nodes connect, finds applications in domains like automotive systems mainly due to its straightforward design and economic viability.

Communication Protocols

Various communication protocols are used in real-time systems to ensure timely and reliable data transmission. These protocols include Controller Area Network (CAN), Time-Triggered Protocol (TTP), and Ethernet for Control Automation Technology (EtherCAT). Each protocol has unique characteristics, advantages, and limitations that suit specific applications. For instance, CAN is extensively utilized in automotive applications because of its resilience and real-time capabilities. TTP is preferred for safety-critical applications like avionics because of its fault-tolerance capabilities. EtherCAT finds its use in industrial automation applications due to its high-speed and deterministic data transmission capabilities [KS22b].

Quality of Service

The quality of service (QoS) is a critical property of real-time networking. It entails prioritizing data packets according to their significance and the application's timing requirements. This guarantees that crucial data packets are sent and received within the necessary time frame, notwithstanding network congestion. Various QoS parameters, including bandwidth, delay, jitter, and packet loss, must be monitored and managed to ensure the determinism and predictability required by real-time applications.

Network Synchronization

Network synchronization involves aligning the clocks of different devices in a network. This is essential to ensure that all devices in the network operate in a coordinated manner and that data is transmitted and received precisely. To achieve network synchronization, various synchronization protocols, including Network Time Protocol (NTP) [Mil91] and Precision Time Protocol (PTP) [WAA+15], are used. These protocols utilize timestamps and synchronization messages to synchronize the clocks of disparate devices in a network. This synchronization is not a one-time activity but a continual process, ensuring that as devices join or leave the network or minor drifts occur in individual device clocks, the overall system remains coordinated.

2.1.4 Challenges in Real-Time Networking

Real-time networking involves several challenges that must be addressed to ensure the determinism and predictability required by real-time applications.

Network Congestion

Network congestion emerges as a significant challenge when the volume of data traffic traversing a network surpasses the network's designed capacity. Such congestion can manifest due to a surge in active devices, simultaneous data requests, or unexpected network events. When congestion takes hold, it can lead to several detrimental effects. Also, the transmission of data packets may be delayed as they queue up, waiting for their turn to be processed. This increased waiting time can subsequently increase network latency [BTT+15].

High latency can introduce unpredictable behaviors in the system, especially in real-time applications that rely on timely data exchanges. Moreover, in extreme cases where the network is heavily saturated, it might be unable to process all incoming data packets, leading to packet loss. Such loss can be catastrophic for real-time applications, as every piece of data is often crucial for system operations.

Network Jitter

Network jitter refers to the variation in the time it takes for data packets to transmit and be received across a network due to network congestion, transmission medium variations, or other factors [ZZX01]. High network jitter can cause timing uncertainties and missed deadlines, leading to degraded system responsiveness and reliability in real-time applications. Various techniques like jitter buffers and traffic shaping are employed to manage network jitter.

Network Latency

Network latency refers to the time it takes for a data packet to reach the destination after being transmitted from the source. Minimizing network latency is crucial to ensure prompt system responses. To achieve this, various strategies are employed. Route optimization identifies the quickest pathways for data packets, providing faster transmission. Similarly, priority queuing ensures critical data is processed swiftly, reducing potential wait times. By continually monitoring and optimizing for latency, real-time systems can offer the immediacy that their applications demand.

2.2 Time-Sensitive Networks

The need for deterministic data transfer has become paramount, especially for applications that require high precision and timely data delivery. Time-Sensitive Networks (TSN) offer a robust solution to these challenges. Building upon the foundational principles of traditional Ethernet, TSN introduces enhanced mechanisms that guarantee the timely and synchronized transmission of data packets [Fin18] [COCS16] [Mes18]. This deterministic approach ensures that mission-critical information is prioritized and delivered within stringent time bounds, making it indispensable for applications like industrial automation, real-time control systems, and emerging cyber-physical systems [BS19].

2.2.1 Time Synchronization

Time is a crucial operational asset in TSN rather than just a sequential measure. Achieving impeccable time synchronization across network devices is imperative. The IEEE 802.1AS protocol exemplifies this commitment by providing a robust framework that ensures all devices operate synchronously, responding to stimuli in a unified manner [GSDP17]. This synchronization is not just about consistency; it guarantees that all network actions, from transmitting to receiving data, happen within precisely defined time frames. This eliminates unpredictability that could otherwise affect system operations. Such synchronization plays a pivotal role, especially in large-scale industrial automation networks, where even slight deviations in timing can culminate in significant operational discrepancies [GGT09]. Simulation and test results have further reaffirmed the performance of IEEE 802.1AS, highlighting its capability to maintain synchronization even in challenging scenarios characterized by extensive node counts and rigorous operational demands [TG08].

2.2.2 Switch Operation in IEEE 802.1

TSN's functionality within the IEEE 802.1 switches, commonly called *bridges*, reveals that this switching process can be decomposed into three distinct operations: *ingress*, *message switching*, and *egress* [NTA+19].

The ingress phase is responsible for the initial reception of incoming data frames. During this phase, frames are classified based on specific criteria such as their priority, source, and intended destination. This classification ensures the orderly handling of incoming data and lays the groundwork for subsequent operations.

After the ingress is the message-switching phase. Here, the actual trajectory or path for the classified frames is determined. An integral component of this phase is the scheduler, which decides the precise timing and sequence for frame transmission based on algorithms and predefined policies. This ensures that critical, especially time-sensitive, frames are accorded the highest priority and are not subject to undue delays.

Concluding the process is the egress phase. This phase manages the final transmission of the frames out of the switch. It refines data dispatch, ensuring they are transmitted within stipulated time windows, especially for time-sensitive frames. Advanced queue management techniques and priority mechanisms are employed at this stage to ensure that data transmission meets the stringent requirements of real-time applications.

2.2.3 Traffic Scheduling and Management

Traffic management within TSN involves precise orchestration. Key to this orchestration is sophisticated traffic scheduling tools inherent to TSN. The Time-Aware Shapers (TAS) stand out, ensuring data packets are dispatched at exact and pre-defined intervals [NTA+19]. This determinism prevents traffic clashes and ensures that time-sensitive data enjoys unhindered passage through the network. The design intent is to instill a level of predictability counteracting potential uncertainties.

2.2.4 Scheduled Traffic - IEEE 802.1Qbv

The architectural capabilities of TSN are further illustrated in the IEEE 802.1Qbv standard. At the heart of the 802.1Qbv standard lies the gate control mechanism, which is implemented on each of the eight queues of an egress port of a TSN bridge. These gates control the flow of data packets, ensuring they are transmitted at their scheduled times without undue interference or delay.

Using a Gate Control List (GCL) gives TSN its deterministic properties. It is a predefined list that specifies the state (open or closed) of each gate at any given time, which dictates when a gate should allow data frames to pass through and when it should block them. This ensures determinism, as frames are only dispatched when their gates are open, adhering to a precise schedule [COCS16].

Another aspect of the 802.1Qbv standard is the ability to seamlessly integrate with other protocols and mechanisms within the TSN suite. For instance, the TAS leverages the gate control mechanism to prioritize and transmit time-sensitive traffic without hindrance. Furthermore, by working with stream reservation and scheduling algorithms, the 802.1Qbv ensures that the network can cater to diverse traffic types, from high-priority control messages to routine IT data.

2.2.5 VLAN Tagging

Virtual Local Area Networks (VLANs) constitute a protocol used in Ethernet networks to generate logically segmented networks within a physical network. The primary purpose of VLANs is to reduce the domain of broadcast traffic, enhance security, and facilitate flexible network management. VLAN tagging, a mechanism from the IEEE 802.1Q standard [IEE18], augments TSN's capabilities by facilitating traffic segmentation, thus enabling precise control over distinct traffic types.

VLAN tagging introduces a method to associate an Ethernet frame with a particular VLAN. This tagging mechanism extends the Ethernet frame structure without modifying its inherent format by adding a VLAN tag to the Ethernet header. This VLAN tag comprises a 12-bit VLAN Identifier (VID) that uniquely identifies the VLAN to which the Ethernet frame belongs.

The roles of VLAN tagging in the context of TSN are as follows:

- *Traffic Segmentation*: Using VLAN tags, TSN-enabled switches can quickly identify the VLAN to which traffic belongs. This facilitates segregating time-sensitive data from non-time-sensitive data, allowing for streamlined processing and forwarding critical data.
- *Priority Handling*: Along with the VLAN Identifier, the IEEE 802.1Q tag incorporates a 3-bit Priority Code Point (PCP) field, offering eight priority levels. TSN can leverage these levels to determine traffic priority, ensuring that high-priority frames receive preferential transmission and queuing treatment.
- *Enhanced Determinism*: By using VLAN tagging in conjunction with other TSN mechanisms, the determinism and reliability of Ethernet networks can be significantly augmented. This is achieved by reserving paths for specific types of traffic.

2.3 Linux Networking

Linux networking is a crucial framework in the Linux operating system that enables communication between computational nodes in isolated and disparate networks. The networking capabilities in Linux are supported by a collection of subsystems and device drivers residing within the Linux kernel. Due to its inherent capabilities, Linux is still popular for network servers, communication routers, and specialized embedded systems[Bol99].

2.3.1 Networking Subsystem

Linux's networking subsystem offers a comprehensive, layered architecture to manage complex network operations. The subsystem's functionality spans interacting with physical hardware, supervising network interfaces, and implementing various network protocols. The modular design of the Linux networking subsystem ensures both scalability and adaptability, catering to the ever-evolving nature of networking needs [Lin23].

Network Devices

Central to Linux's networking ecosystem is the concept of network devices. These comprise physical or virtual components responsible for data packet transmission and reception. The most fundamental elements in this category are the device drivers. These specialized software components facilitate the interaction between the operating system and the network hardware. Such hardware includes Ethernet adapters, Wi-Fi modules, Fibre Channel interfaces, and more.

Above the device drivers, the Linux kernel incorporates the network device subsystem. This managerial entity provides a standardized API for various networking operations. It abstracts the intricacies of hardware interactions, offering a consistent interface to higher-level subsystems. This ensures that data packet handling remains consistent, irrespective of the underlying hardware nuances.

Network Protocols

Protocol-specific subsystems are integral to the functioning of the Linux networking subsystem. They play an important role in the encapsulation, transmission, reception, routing, and validation of data packets. Some of the most widely used and recognized protocols supported by Linux include Internet Protocol (IP), Transmission Control Protocol (TCP), User Datagram Protocol (UDP), and Internet Control Message Protocol (ICMP) [Tan81].

- IP: Serves as the backbone for most network communication. IP handles packet routing among network nodes or between different networks. It employs a hierarchical addressing model, wherein each node is distinctly identifiable through a unique IP address.
- TCP: This connection-oriented protocol ensures the reliable exchange of data packets. TCP manages tasks like connection establishment, sequential data transfer, and connection termination, guaranteeing orderly and accurate data delivery.
- UDP: In scenarios where rapid data transmission precedes reliability, UDP is a suitable choice. It facilitates a connectionless, best-effort mode of data delivery, often employed in applications like video streaming or online gaming.
- ICMP: Often utilized alongside IP, ICMP aids in sending error messages and operational information. It plays a role in diagnostics, with tools like “ping” relying on ICMP echo requests and reply messages.

Network Interfaces

The concept of network interfaces in Linux lies at the intersection of software and hardware. These interfaces can be tangible, like a physical Ethernet port, or abstract, like a software-defined virtual interface, say *veth0*. Each interface possesses a distinct identifier, often following conventions such as *eth0* for the primary Ethernet interface or *wlan0* for wireless interfaces. Beyond identification, each network interface is tied to a device driver, serving as a bridge between the Linux operating system and the hardware device. Configuration and management of these interfaces are achieved through utility tools. While the legacy ***ifconfig*** [KCB+08] tool is mentioned in historical contexts, the modern ***ip*** [Lit11] command has become the de facto utility for network interface configuration. Users can adjust parameters like IP addresses, subnet masks, and gateways through these utilities. More advanced operations include VLAN configuration, MTU size adjustments, and interface bonding.

2.3.2 Network Configuration

Network configuration on Linux systems is vital for the efficient and secure movement of data packets. Adequately configured network interfaces, routing tables, and firewall rules ensure seamless communication and robust security. The ***ip*** tool is central in Linux for interface configuration, allowing for the assignment of IP addresses, determining MTU sizes, and controlling the state of interfaces. On the routing front, Linux uses tables that dictate how packets are forwarded, with tools like ***route*** [BE14] or ***ip-route*** [Lit12] being essential for administrators to view and modify these tables. Security is always at the forefront of networking considerations and is fortified using Linux's netfilter system. The ***iptables*** [Eyc23] tool interfaces with this system, providing packet filtering, NAT, and other firewall functionalities. Additionally, for Ethernet frames, Linux integrates the ***ebtables*** [Lin11] utility, allowing for filtering at the link layer level, thus giving finer control over frame transmissions.

2.3.3 Routing and Switching

Routing and switching constitute fundamental operations ensuring data packets navigate interconnected devices and networks. While intrinsically linked, these processes cater to different aspects of the data transmission protocol and play distinct roles within the networking ecosystem.

Routing

At its core, routing determines the path for data packets traveling across diverse networks. The function is carried out by routers, which serve as gateways connecting multiple networks. When a data packet needs to be transmitted, the router examines its destination IP address and consults its routing table. The routing table guides where the packet should be forwarded next, ensuring it reaches its intended destination efficiently

There are two principal categories of routing methods: *static* and *dynamic*. Static routing involves the manual configuration of routing tables, where network administrators predefined paths. In contrast, dynamic routing employs algorithms and protocols such as Routing Information Protocol (RIP) [Hed88], Open Shortest Path First (OSPF) [PSH+02], and Border Gateway Protocol (BGP) [RLH06] to automatically update and adapt routing tables in response to changes in the network topology.

Switching

Switching operates primarily within a local area network (LAN) and deals with transmitting data frames between devices on the same network. Switches champion this task. When a data frame emerges on one of the switch's ports, the switch inspects the frame's destination MAC address, consults its MAC address table, and then forwards the frame to the appropriate port leading to the destination device.

Two prevalent modes in switching include *store-and-forward* [Lam76], where switches retain the entire data frame, inspect it for errors, and then forward it, and *cut-through* [KK79], where switches begin forwarding the frame as soon as they have processed its destination MAC address, thus increasing speed but potentially allowing error-prone frames.

In modern networks, advanced switching techniques like VLAN have also gained prominence. VLANs allow for the creation of logically segmented networks within a physical network, ensuring devices within a VLAN can communicate as if they are on the same physical network, even if they are not. This brings added layers of efficiency, security, and manageability.

2.3.4 Socket Programming

Sockets serve as the default interface, bridging the application and transport layers. Fundamentally, a socket can be visualized as an endpoint in a bidirectional communication channel that operates across networked servers and clients [XZ09].

Socket Types and Characteristics

Networking paradigms typically employ three dominant socket types, each serving specific communication requirements:

- *Stream Sockets (SOCK_STREAM)*: Recognized as connection-oriented sockets, these predominantly utilize TCP. Ensuring a two-way, reliable, sequenced, and error-free data exchange, stream sockets benefit from the inherent properties of TCP, which provides mechanisms like flow control and acknowledgment.
- *Datagram Sockets (SOCK_DGRAM)*: Leaning primarily on UDP, these sockets function in a connectionless mode. The absence of an established session or connection means that data packets dispatched via datagram sockets are independent entities without assurances of ordered or reliable delivery.
- *Raw Sockets (SOCK_RAW)*: Diverging from the conventional application-layer programming paradigm, raw sockets provide direct access to transport-layer protocols. This means applications can craft custom headers and directly interface with underlying protocols, such as ICMP or IP, offering a deeper level of control and customization. This capability is essential for crafting specialized network tools, like packet sniffers or custom protocol implementations.

2.4 Docker Containers

Docker is a platform leveraging OS-level virtualization to streamline the creation, deployment, and execution of container applications. Unlike traditional virtual machines, Docker utilizes containers for a more lightweight and efficient runtime environment. These containers are derived from Docker images—immutable snapshots housing the application, its runtime, essential system tools, libraries, and configurations. By encompassing an application and its dependencies within such a container, Docker ensures consistent behavior across different deployment settings. It effectively addresses the challenges of variable environments and the pervasive “it worked on my machine” issue [And15].

2.4.1 Docker's Virtualization Technology

Docker's approach to virtualization is distinctively different from traditional VMs. While VMs run full-fledged operating system instances atop a hypervisor, adding substantial overhead, Docker containers interact directly with the host operating system. This minimizes overhead, enhances performance, and allows multiple containers to share the same OS kernel while maintaining isolated user spaces. Consequently, Docker's architecture is more streamlined and efficient than conventional VMs, making it a preferable choice for many deployment scenarios.

2.4.2 Underlying Architecture

Docker's architectural foundation is structured to enable the seamless creation, deployment, and execution of containerized applications. Key elements of this architecture include [RBA17]:

- *Docker Client*: Often referred to as the Docker CLI (Command Line Interface), the Docker Client serves as the primary user interface to Docker. Users leverage the client to interact with Docker, issuing commands to initiate, manage, or terminate Docker containers.
- *Docker Daemon*: Running in the background, the Docker daemon (*dockerd*) supervises key functionalities, managing Docker images, containers, networks, and storage volumes. The Docker Client communicates with the daemon through the Docker API.
- *Docker Images*: Docker images are static templates that instantiate containers. They are formulated using Dockerfiles, textual scripts that enumerate the requisite steps to fabricate an image. Images capture the application and its environment, ensuring portability and consistency.
- *Docker Containers*: These are live, operational instances of Docker images. A container wraps up an application alongside its environment, assuring uniform operation across distinct infrastructures.
- *Docker Registry*: As a repository for Docker images, a Docker registry facilitates storage and dissemination. While Docker Hub and Docker Cloud are renowned public registries, organizations can also establish private registries tailored to their needs.

3 Literature Review

In this chapter, we examine a range of scholarly articles that contribute to the understanding of static and dynamic traffic planning for TSN, the nuances of emulating TSN, and insights into the EnGINE Framework. The focus is critically assessing these studies to comprehend their methodologies, contributions, and interrelations, providing a cohesive overview.

3.1 Traffic Planning for TSN

In [SCO18], the authors comprehensively explore the challenges and advancements of modern cyber-physical systems and address the need for reliable real-time communication across various industries, such as automotive and industrial automation. The focus is on the IEEE 802.1Qbv standard, which aims to standardize TSN for cross-industry efficiency and enhance communication capabilities.

A solution proposed by the authors, centered around static traffic planning, uses a *communication schedule* for synchronized, time-triggered communication while minimizing transmission latency. This approach involves developing a system model that uses graphs, nodes, streams, and frames to explain network architecture and classical time-triggered communication, laying the foundation for understanding how communication schedules are created in traditional systems. SMT solvers are employed to encode networks and communication needs as it allow for the synthesis of schedules and verification of existing ones. The experimental results support the claims that varying the number of streams and windows affects schedule synthesis. Scheduling at the switch level is also introduced, allowing for more refined control over frame forwarding times and a notable enhancement over the classical model.

Routing and scheduling of time-triggered traffic in TSN is also addressed in [AHM19]. The author's approach revolves around the no-wait scheduling concept and iterated integer linear programming-based scheduling (IIS) techniques, which combines the novel Degree of Conflict (DoC)-aware streams partitioning (DASP) and DoC-aware multipath routing. The DASP technique, in particular, effectively addresses the mutual dependencies between streams through a graph-based representation. This reduces conflicts and optimizes scheduling by considering shared links, frame size, and frame period of streams. Integrating the Normalized Cut (NCut) framework further enhances the DASP technique by ensuring balanced sizes of stream groups, thereby avoiding biases toward smaller sets.

Atallah et al. also explore DoC-Aware Multipath Routing (DAMR), which generates optimized stream sets while maintaining the redundancy levels essential for fault tolerance in TSNs [AHM19]. The procedure encompasses preprocessing to generate multiple path sets, initial solution construction, and a local search for solution refinement using iterated greedy heuristics that ensures efficient and fault-tolerant routing in TSNs.

In terms of performance, the DoC-aware iterative routing and scheduling (DA/IRS) method demonstrates remarkable scalability, speed, and success rate improvements compared to existing techniques such as ILP-based and pseudo-Boolean joint routing and scheduling methods. The DA/IRS method's ability to handle networks with 21 bridges and 480 messages establishes its effectiveness, showcases its scalability, and substantially improves success rates (from 47% to 90%) compared to traditional random stream partitioning methods.

The proposed DA/IRS method opens avenues for further exploration, especially in the context of even larger and more complex network architectures. The scalability and efficiency demonstrated by DA/IRS present a significant step towards meeting the growing demands of industrial automation systems. Additionally, integrating these techniques offers a template for future studies to enhance various network infrastructures' fault tolerance and real-time capabilities.

While the discussed literature explores traffic planning in TSN, there is a concern regarding the scalability of solver-based methods. Although robust in theory, these methods are less efficient and more time-consuming in practical, large-scale implementations, which leads us to consider alternative approaches that prioritize speed and scalability.

3.2 Dynamic Traffic Planning in Networked Applications

In [FGD+22], the authors introduce an approach for dynamic traffic planning of time-triggered flows, leveraging a conflict-graph-based model. This model facilitates the reconfiguration of active flows to enhance network utilization while providing robust QoS guarantees during transitions.

The complexity of computing a network-wide traffic plan in dynamic scenarios is often likened to the NP-hard Job Shop Scheduling Problem. Previous methods predominantly addressed static scenarios, failing to accommodate the dynamic reconfigurability required in modern industrial and cyber-physical systems. The concept of "plug-and-produce" in *Industry 4.0* further highlighted the need for traffic planning that could adapt to changes, such as adding or moving devices in a network.

The proposed approach in this paper marks a departure from defensive planning, which does not alter active flows. Instead, it introduces an offensive planning approach, allowing for the reconfiguration of active flows for better resource utilization. This is complemented by a heuristic for traffic planning, focusing on efficiency and scalability.

The evaluation of a prototypical C++ implementation of the planner demonstrated its effectiveness in handling scenarios with hundreds of active flows, signifying a notable improvement over existing methods. The system model outlined in the paper, encompassing aspects such as traffic flow dynamics, application interaction, and node capabilities, provides a comprehensive framework for understanding and implementing the proposed traffic planning method. The paper also examines the impact of active flow reconfiguration on QoS, particularly in terms of jitter, and introduces methods for computing and restricting QoS degradation. This aspect is critical for maintaining deterministic real-time communication in networked systems.

The comparative analysis of the proposed *Greedy Flow Heap Heuristic (GFH)* with an integer linear programming (ILP) approach and an adapted version of Luby's maximal independent vertex set algorithm [Lub85] further validates the effectiveness of the GFH in handling more extensive

and complex scenarios. Additionally, the impact of network topology and size on the planner's performance was evaluated, indicating that the approach scales with the complexity of the traffic planning problem rather than the network size alone.

This research opens avenues for future work, including optimizing conflict graph data structures, relaxing the zero-queuing constraint, and developing error-handling protocols for traffic plan computation and deployment. Building on this dynamic traffic planning method, our thesis will specifically incorporate the schedules generated by the GFH algorithm for deployment and verification in real-time scenarios.

3.3 Emulating TSN

The paper [UAKF21] focuses on emulating TSN in virtual environments as a cost-effective alternative to specialized hardware. The authors of this paper propose a software-based emulation approach and rigorously compare it against hardware-based TSN implementations and state-of-the-art Software Defined Networks (SDNs).

To emulate TSN in virtual environments, the authors begin by integrating TSN within the Linux Kernel and Mininet. This involves enhancing the kernel with real-time capabilities for packet processing through time synchronization, traffic scheduling, and configuration. The role of Precision Time Protocol (PTP) for clock synchronization and schedulers for deterministic packet handling is especially significant. This methodology for measuring time synchronization, packet forwarding delay, jitter, and TAS gate states accuracy is comprehensive and robust, covering both software-based solutions and physical network devices that include a detailed process for time-stamping in hardware and Mininet and an evaluation of jitter and delay.

The results reveal that while Mininet offers a propagation delay comparable to that of dedicated hardware, the timing precision required for TSN Ethernet frames is not consistently achieved. The study indicates that the TAS jitter ranges from 23.71 µs in a 2-hop topology to 69.12 µs microseconds in a 20-hop topology. Furthermore, the network consists of a linear topology with only one sender and receiver, with traffic flowing in a single direction. This simplistic setup reduces processing times at the switches, representing a best-case scenario for the emulator's performance. In real-world applications, where network topologies are rarely this straightforward and traffic flows can be bidirectional and more complex, the timing precision could be expected to degrade further.

Given these findings, it is clear that while Mininet may provide an economical and flexible solution for emulating networks that do not demand stringent latency constraints, its accuracy in TSN testbed scenarios, particularly where precise synchronization is crucial, falls short. Looking ahead, the integration of machine learning algorithms into Mininet for TSN management is proposed. Such advancements could enhance Mininet's capability to handle TSN protocols by dynamically optimizing scheduling and traffic management, reducing jitter, and improving timing accuracy. However, the current margin of deviation from the required precision, especially in more complex topologies, underscores the necessity for continued research and development to meet the exacting standards of TSN Ethernet frames in virtual environments.

3.4 EnGINE Framework

Previous research shows that TSN's flexibility and determinism make it ideal for diverse domains. Each domain presents unique challenges, like varied traffic patterns and stringent latency requirements. Studies have delved into throughput requirements, the impact of traffic shaping, and the diversity of traffic types, particularly in industrial settings. The modeling of TSN using approaches like Network Calculus has helped predict performance characteristics, and simulation frameworks such as OMNeT++ [VH10] and INET [SKKS11] have enabled researchers to evaluate various TSN traffic shapers and schedulers. However, a gap persists in the literature regarding a comprehensive, reproducible methodology for TSN experimentation that combines both simulation and physical deployment insights.

[BRH+22] and [RBP+21] introduce a novel contribution in this field called the *EnGINE* framework, short for *Environment for Generic In-vehicular Networking Experiments*, facilitating scalable and reproducible TSN experiments. The EnGINE framework incorporates the Linux networking stack and leverages commercial off-the-shelf hardware and NICs. The methodology is distinctive in its comprehensive coverage of TSN experimentation phases, including network setup, traffic generation, system optimization, and performance evaluation. The research underscores the importance of replicable results in network experiments, a facet often overlooked in previous studies.

A significant contribution of the EnGINE framework lies in its detailed case study on intra-vehicular networks (IVNs). Here, the methodology is applied to assess network performance under TSN standards, focusing on system optimization techniques and evaluating qdisc parameters. The paper's results demonstrate the effectiveness of the proposed methodology in meeting TSN requirements for IVNs, albeit with some limitations. Key findings include successfully implementing qdiscs like *MQPRIO*, *ETF*, *CBS*, and *TAPRIO* in managing network traffic to adhere to Service Rate (SR) classes. However, challenges in jitter control and parameter optimization are noted. The research also reveals the need for precise time synchronization and the impact of software limitations on measurement tools like *tcpdump*.

For future work, [BRH+22] and [RBP+21] suggest extending the EnGINE framework to evaluate a broader range of TSN standards, enhancing its applicability to other domains like aerospace and industrial automation. The authors also recommend exploring automated solutions, possibly leveraging machine learning, to streamline the configuration and evaluation process of TSN experimentation. Additionally, a future research direction is proposed to address the system limitations identified, such as improving timestamping accuracy and examining clock synchronization over extended network hops.

Despite the comprehensive and innovative approach of the EnGINE framework, it only partially aligns with the specific needs of our thesis. As noted on the framework's official GitHub page, a critical limitation is its dependency on specific hardware infrastructure [rez23]. EnGINE requires a deployment environment where each node is accessible via SSH from a central management host. This necessity for specific hardware and network configurations poses a significant constraint for our project, which aims to develop and test solutions in a fully virtualized emulation environment. Our focus is creating a setup that allows for greater flexibility, reduced reliance on physical hardware, and the ability to emulate a wide range of network scenarios without needing specific hardware configurations.

4 Research Goals

This chapter outlines the primary research goals of this thesis designed to provide a comprehensive understanding of real-time schedule deployment, contribute to academic research, and offer insights for practical implementations:

- What is the accuracy of frame transmission times in a virtual testbed when deploying TSN schedules compared to expected real-world performance?
- How does the deployment strategy for TSN schedules perform across network topologies of varying sizes and complexities, and how can the precision be maintained regardless of the network scale?
- How can virtual testbeds be designed to facilitate evaluating and verifying different TSN algorithms beyond the GFH algorithm [FGD+22]?
- How can we automate the entire process pipeline from creating the virtual testbed through deploying real-time schedules to generating empirical results for analysis?
- When measured values (frame transmission accuracy, end-to-end latency, processing delay) deviate from expected outcomes, what modifications or improvements can be made to enhance accuracy?

5 Testbed Setup

This chapter outlines the experimental setup, specifically delineating the design and configuration of the virtual test bed used for our research. It delves into the reasons for selecting a virtual environment as the platform for experimentation, highlighting its alignment with the research goals. The discussion also acknowledges the inherent limitations of a virtual setup yet underscores its role in facilitating a comprehensive and controlled examination of real-time schedule deployment and verification.

5.1 Justification for a Virtual Experimental Setup

- *Controlled Environment*: A virtual test bed ensures an environment where network parameters and traffic conditions are governed with a high level of determinism, allowing scheduling policies to be finetuned. This control is indispensable for exploring real-time scheduling behaviors under various conditions that can be systematically manipulated to understand their impacts on key performance metrics.
- *Reproducibility of Results*: A virtual setting ensures high experiment reproducibility. Variations in physical hardware, transient environmental factors, and human error are mitigated, which is crucial for validating the empirical aspects of scheduling deployments.
- *Scalability and Rapid Prototyping*: Physical test beds can be resource-intensive and less scalable. Virtual environments, in contrast, allow for the emulation of extensive and complex networks without the corresponding physical resource investment, thereby supporting large-scale experiments that might otherwise be prohibitively expensive or logistically unfeasible. Virtual test beds also facilitate swift prototyping and iterative testing of scheduling algorithms. Changes can be implemented and evaluated much faster in a virtual environment than in a physical one, accelerating the development cycle and discovering potential issues.
- *Access to Advanced Technologies*: Virtual setups can incorporate the latest advancements in networking and scheduling theory, often before they become widely available in physical components. This allows for evaluating next-generation technologies and their impact on real-time scheduling. They also offer a cost-effective alternative to frequently updating or replacing physical hardware.
- *Enhanced Analytical Capabilities*: A virtual test bed can be equipped with sophisticated logging and analysis tools, which might be challenging or impractical to integrate into physical systems. Such tools are crucial for conducting thorough analyses of real-time schedules and empirically verifying their performance.

Despite the advantages mentioned above, virtual test beds do have their limitations, particularly in the realm of testing real-time schedules:

- *Timing Accuracy*: Virtual environments cannot guarantee the microsecond-level timing precision required for hard real-time deadlines due to the non-deterministic nature of software-based timing.
- *Resource Contention*: Software virtualization may introduce unpredictability due to the shared nature of computing resources.
- *Overheads*: There are intrinsic overheads associated with virtualization that can affect a system's timing and performance characteristics.

Nevertheless, it is imperative to understand that these drawbacks do not invalidate the utility of a virtual testbed. In the context of this research, virtual environments act as a close approximation of real systems, which is essential in providing valuable insights into system behavior. These insights are integral to informing and enhancing physical implementations. Furthermore, virtual test beds enable one to study the upper bounds of system performance within controlled settings, shedding light on the best-case scenarios that might be achievable. This aspect is crucial in understanding the limits and potential of real-time schedules under ideal conditions. Additionally, these environments serve as an excellent platform for the preliminary validation of concepts. They offer a safe and cost-effective space for initial experimentation, a prudent step before transitioning to more expensive and potentially riskier real-world modifications.

5.2 Virtual Test Bed Configuration

We detail the infrastructure and methodology of the virtual environment used to emulate a TSN.

5.2.1 Infrastructure and Setup

The experimental setup involves the utilization of Docker containers to emulate the devices and switches within a TSN network on a VM. Our choice for using Docker containers over network namespaces or Mininet will be discussed in Section 5.3. The emulated devices represent the end nodes, while the switches serve as intermediate nodes facilitating communication within the network.

The Docker containers are instantiated from a base Ubuntu image. This choice ensures a lightweight yet versatile Linux environment, mirroring a typical Linux-based network node that can easily be configured and manipulated. Upon instantiation, each container is initialized with the necessary networking tools and software required to emulate the devices and switches in the TSN network. This includes installing custom scripts and applications that handle the generation, transmission, and routing of Ethernet frames according to the TSN protocols and the scheduling algorithm's output.

The exact topology of the emulated TSN network is derived from an edge list file. A pair of virtual Ethernet interfaces (*veth*) is created for each connection indicated in the edge list. These pairs are then assigned to the respective containers, ensuring the virtual network accurately represents the desired topology. By automating this process through a script, the setup allows for rapid reconfiguration and scaling of the network to accommodate various topological scenarios and experiments. This approach addresses the limitations of the default *eth0* interface, which connects

to the *docker0* bridge and broadcasts frames to all containers. By using virtual Ethernet pairs, we ensure that frames are only detectable by the designated next hop, mirroring the targeted behavior of a TSN network.

The emulation extends to incorporate a Central Network Controller (CNC) through a suite of scripts and programs that orchestrate the network operation. The CNC scripts distribute the schedule files to the appropriate emulated devices. This is akin to the CNC's role in a real TSN, where schedules for time-triggered communication are disseminated to the network components to ensure synchronized data transmission without collisions. In the virtual testbed, the scripts ensure that each "Device_X" container receives its specific schedule file, which contains the timing and route information for sending frames according to the algorithm's output.

Beyond schedule dissemination, the scripts also establish the necessary *ebtables* routing rules on each "Switch_Y" container, dictating the path each frame should take based on its VLAN ID and destination. This ensures that frames traverse the emulated TSN as they would in the physical world, adhering to the planned schedules to avoid collisions and ensure timely delivery.

5.2.2 Queuing Discipline Configuration

Within our experimental setup, the Earliest TxTime First (ETF) queuing discipline (qdisc) replicates the temporal precision necessary to verify real-time schedules empirically. In a TSN, the transmission of frames is not left to chance; instead, it is a carefully orchestrated process. This level of precision is emulated in the virtual environment through the configuration of *tc qdisc*, a part of the *iproute2* package in Linux, which allows for manipulation of the queuing disciplines used by the kernel for frame scheduling.

We will look closely at qdisc configuration for a particular veth through an example. The command shown in Listing 5.1 creates a new root qdisc (parent root) of type Multiqueue Priority (MQPRIO) on the network interface *veth2_0* and sets the handle to *100:* which is an identifier for this qdisc that can be used to reference it in subsequent commands. The mqprio qdisc enables the classification of frames into different traffic classes (TCs) that map directly to a set of hardware queues. However, since we are working with virtual Ethernet devices that do not support hardware queue offloading, we configure MQPRIO in a way that relies solely on software. Here is a description of the different parameters of the command in Listing 5.1 [Fas13]:

```
1 tc qdisc add dev veth2_0 parent root handle 100: mqprio num_tc 1 map 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 queues 1@0 hw 0
```

Listing 5.1: Adding an MQPRIO qdisc to *veth2_0*

- *num_tc 1*: This specifies that only one traffic class will be used. The mqprio qdisc can support up to 16 traffic classes, but since the goal is to use ETF for time-based frame scheduling with zero-queuing rather than traffic classification, a single traffic class suffices.
- *map 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0*: This mapping parameter ensures that all 16 possible frame priorities are mapped to traffic class 0, the only one in use. This is because we are not distinguishing between priorities for different types of traffic; instead, we are focusing on timing.

- *queues 1@0*: The format count@offset specifies the range of queues for each traffic class. Here 1@0 indicates one queue starting at offset 0. This aligns with our singular traffic class and queue for this configuration.
- *hw 0*: This flag indicates that we are not using the hardware quality of service features that mqprio can map onto. This is critical because veth devices are software constructs and do not have hardware capabilities.

In Listing 5.1, the command creates a single queue MQPRIO qdisc that all traffic is mapped to, without hardware features, tailored for the subsequent application of ETF. We now configure ETF as a child qdisc to manage the transmission times for frames. We use a non-leap-second-adjusted clock and a pre-transmission buffer period, ensuring high timing accuracy for time-sensitive applications. The command in Listing 5.2 replaces an existing qdisc or adds a new one if it does not exist under the parent 100:1. Here is a description of the different parameters of the command in Listing 5.2 [SG18]:

- *clockid CLOCK_TAI*: This specifies the clock to be used by the ETF qdisc. CLOCK_TAI (International Atomic Time) is used as it does not include leap seconds, providing a consistent and monotonically increasing time source.
- *delta 10000000*: This parameter sets the "wake-up" time for frame transmission at 10 ms before their intended transmission time (*txtime*). The implications and effects of modifying the delta value will be discussed in Chapter 7.
- *skip_sock_check*: ETF would, by default, drop any frame without an associated socket or the SO_TXTIME option set. Since the frame's transmission time might be specified elsewhere in the kernel (e.g., by another qdisc), this check is skipped to avoid unnecessary frame drops.

In our virtual test bed, specific configurations of the ETF qdisc are tailored to align with the limitations and capabilities of a virtual environment, omitting options like *deadline_mode* and *offload*, which are more pertinent to hardware-based setups. When activated, the *deadline_mode* in ETF qdisc alters the handling of *txtime* from a strict transmission schedule to a deadline-oriented approach. This means that the mode changes the transmission time of a frame to the current time ("now") during the dequeue process, effectively treating *txtime* as the latest permissible transmission time rather than a fixed point in time. However, this flexibility in transmission timing does not align with the objectives of our experiment, which seeks to replicate a TSN environment with strict adherence to predefined transmission times. Therefore, we disable this option to maintain a stringent simulation of TSN scheduling behaviors.

```
1 tc qdisc replace dev veth2_0 parent 100:1 etf clockid CLOCK_TAI delta 10000000
skip_sock_check
```

Listing 5.2: Adding an ETF qdisc to *veth2_0*

The *offload* option enables time-based transmission arbitration directly within the network interface controller's hardware. This feature offloads the management of packet transmission timings to the hardware level, utilizing hardware timers for precise control. However, the virtual devices used in our testbed lack the physical hardware capabilities necessary to utilize the offload feature, rendering it irrelevant and inoperative in this context. Therefore, we exclude the offload option from our configuration, focusing solely on the software-level emulation without the influence of hardware-specific features.

5.2.3 Ethernet Frame Structure and Network Traffic Analysis

The Ethernet frames are structured with custom JSON payloads, which include the following parameters:

- *Offset*: It indicates the specific timing offset for the frame within a given cycle in microseconds, defining the precise moment the frame should be dispatched from the source device relative to the start of the cycle.
- *Period*: The period defines the recurrence interval of the frame transmission. By knowing the period, network tools and infrastructure can anticipate when the next frame is due and allocate resources accordingly, facilitating a seamless flow of periodic traffic.
- *Stream ID*: Also serves as the VLAN ID in our case. It allows network switches and routers to distinguish between different data streams, ensuring that each frame can be correctly identified and handled according to its assigned stream. This is especially useful when prioritizing traffic, implementing QoS policies, or troubleshooting specific streams without affecting others.
- *Destination Node*: This field specifies the intended recipient of the frame and enables switches to route the frames using the VLAN tag. It reduces the need for broadcast traffic and improves security by ensuring that only the designated node processes the frame.
- *Frame Size*: Knowledge of the frame size is essential for managing bandwidth. We append "0x01" characters to the JSON payload to achieve the precise frame sizes in our experiment. The choice of "0x01" is for readability as it is a non-printable character.

Utilizing tcpdump, one can intercept and analyze these Ethernet frames, extracting the JSON payloads for deeper inspection. By doing so, we can verify if the scheduling parameters are being adhered to as the frames traverse the network.

In our case, the GFH algorithm [FGD+22] generates the traffic plan. This algorithm generates a JSON file containing all the spatial and temporal information, which is processed by our CNC program and converted into a more machine-friendly format. The versatility of the virtual testbed comes to the fore in its capacity to incorporate a variety of other traffic management algorithms, highlighting its adaptability and extendibility. Although the current configuration utilizes the GFH algorithm, the underlying framework of the testbed is designed to be algorithm-agnostic. This versatility is a boon for anyone who aims to emulate and analyze different traffic scheduling and path allocation strategies, allowing for a broad exploration of potential improvements and optimizations in a network.

In this networking scheme, the first ethertype in our frame has a value *0x8100*, which is used to identify frames tagged with IEEE 802.1Q VLAN information. We set the PCP to the binary value of 111, or 7 in decimal, deliberately as our container veth interface supports only a single transmit and receive queue, as indicated by the presence of only **rx-0** and **tx-0** within the **sys/class/net/veth2_0/queues/** directory. By assigning the highest possible priority value of 7 to the frames, all frames traversing the veth are ensured to receive the highest level of QoS.

The second ethertype in our frame has been purposefully chosen as *0x88FF*. This value falls within the range of ethertypes reserved for experimental and locally defined usage *(0x88B5 - 0x88FF)*. Selecting *0x88FF* (or any other value from a specific range) minimizes the risk of conflict with

standardized ethertypes. It provides a clear distinction that the payload following the ethertype field is not an officially sanctioned protocol but a custom-defined format, which in this case is the JSON structure carrying the scheduling information. This choice aids in setting apart the operational traffic from standardized protocols, reducing the potential for confusion in automated systems and simplifying network analysis.

For the operational cycle, a sequence of frames is generated for one hypercycle—the period in which all scheduled tasks are executed at least once—and stored in a circular queue. This approach enables the experimental setup to replicate ongoing network operation, as the circular queue facilitates the reuse of the schedule for subsequent hypercycles, closely mirroring the continuous loop of real-time network traffic in a TSN system.

5.3 Rationale for Choosing Docker Containers

In the architecture of our network emulation testbed, Docker containers are employed rather than traditional network namespaces or network emulation tools like Mininet. This decision is rooted in several practical considerations beyond the capabilities of namespaces and Mininet, particularly regarding convenience and user experience.

Docker containers provide a more interactive and user-friendly interface compared to network namespaces. The Docker CLI offers intuitive commands that allow users to easily manage container lifecycle events, such as starting, stopping, and entering a container's shell, and this is apparent when working remotely over SSH. This ease of interaction streamlines the process of debugging and development. Containers can be accessed as if they were independent VMs, which is highly beneficial for troubleshooting and interactive development sessions, something that is less straightforward when working with raw network namespaces.

Moreover, Docker simplifies adding new libraries or tools within containers. Thanks to Dockerfiles and the layering of images, the installation and configuration of software within a container can be defined as code. This "infrastructure as code" approach means complex setups can be replicated consistently. In contrast, setting up similar environments with network namespaces would require manual configuration, which is less scalable.

While a Docker-based emulation of TSN may not perfectly replicate every aspect of physical hardware devices, it offers significant similarity in network behavior. This similarity is crucial for ensuring that our findings and observations within the Docker environment have practical relevance and applicability to actual TSN scenarios. The ability to approximate real TSN hardware behaviors in Docker aids in bridging the gap between experimental results and their implications in real-world deployments, enhancing the validity and reliability of our testing outcomes.

Lastly, Docker's model of containerization inherently supports portability. Containers encapsulate dependencies, which means they can be easily moved and deployed across different systems without compatibility issues. When considering the emulation of more extensive networks, Docker's scalability becomes evident. It can seamlessly scale to larger clusters, vital for stress-testing network topologies under high loads.

6 Schedule Deployment

This chapter delves into the specifics of deploying schedules generated by the GFH algorithm [FGD+22] within a virtualized test environment. We will outline the detailed configuration of the virtual setup, including hardware specifications and the software environment, followed by an in-depth explanation of the five-step pipeline designed for the automated deployment of schedules.

6.1 Host System Specifications

The host VM operates on *Ubuntu 22.04.2 LTS* and Linux kernel version *5.15.0-78-generic*. The VM employs KVM (Kernel-based Virtual Machine), backed by AMD-V technology, for efficient full virtualization. At its core lie two *AMD EPYC 7401 24-core* Processors with an *x86_64* architecture. This CPU features 48 cores, each core operating at a BogoMIPS of 3999.99, and is adept at managing high processing requirements. The VM is also equipped with 188 GB of RAM.

6.2 Generating a Traffic Plan

The foundational step in traffic plan generation is creating an edge list file. This file represents the network topology and differentiates between various types of nodes (device and switch nodes) within the network. We will use the graph topology illustrated in Figure 6.1 as a primary reference for our initial experiments and analyses. This configuration, consisting of 4 devices and 2 switches, provides an understandable framework for demonstrating the core principles and functionalities of our TSN virtual test bed. More complex network topologies and the corresponding edge list files can be generated quickly and accurately using the suite of scripts provided by the authors of the Dynamic Flow Scheduler tool [FGD+22]. Utilizing these scripts, we also create a medium-sized topology, as shown in Figure 6.2, comprising 16 devices and 16 switches, to validate our methods in a more extensive and complex network scenario.

With the network topology defined in the edge list file, the next step is to generate the network traffic scenario, which is achieved using a Python script as part of the DFS tool. The script uses a configuration file that contains key parameters to shape various traffic scenarios, such as the number of time steps, which defines the temporal scope of the emulation, the list of permissible frame sizes, and the periodicity of the frames. By adjusting the parameters in the ini file, one can generate a wide range of network conditions and traffic patterns. This flexibility is essential for testing the network's response to loads and traffic configurations.

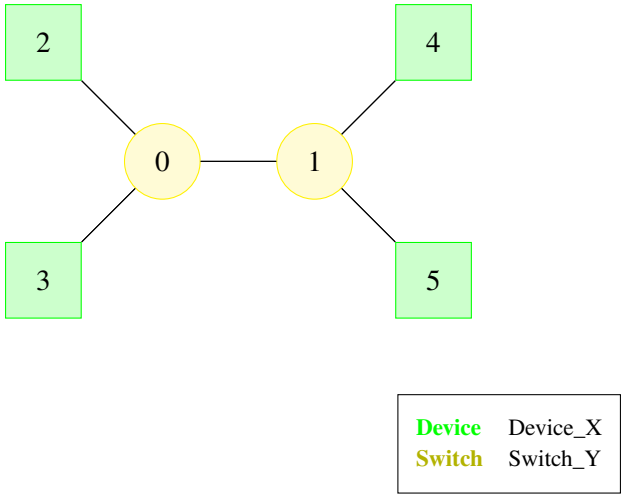

Figure 6.1: Small topology of 4 devices and 2 switches. Green-colored nodes represent end devices, and yellow-colored nodes represent switches.

The path to the edge list file is also specified within this configuration file, linking the scenario generation process directly to the defined network topology.

Once the scenario file is ready, the DFS executable is run, which is responsible for processing the scenario file and generating the final traffic files containing routing and scheduling information for TSN operations.

6.3 Workflow for Traffic Plan Deployment and Execution

This section delves into the code pipeline for deploying traffic plans and facilitating frame transmission within our virtual TSN environment. The workflow is encapsulated in a streamlined process, orchestrated by the master wrapper script. This script executes a series of steps, each contributing to the emulation, from setting up the environment to analyzing the results. Figure 6.3 shows a pictorial representation of the complete workflow and the order of execution of files in each step.

6.3.1 Step 1: Environment Setup

The setup begins with deploying Docker containers, each representing a node in the network topology. The decision on whether a container emulates a device or a switch is based on the number of connections in the network graph - a node with multiple edges is designated as a switch, while one with a single edge functions as a device. Once the containers are up and running, they have the necessary networking tools and software packages.

The connectivity between these containers is established using veth pairs. Each veth pair represents a direct link between two nodes in the network graph as shown in Figure 6.4, mirroring the physical connections in a real-world TSN. After creating the veth pairs, each container's network is configured to reflect its role in the TSN. For switches, this involves setting up internal bridges to manage traffic

Figure 6.2: The medium-sized topology used for empirical testing with 16 devices and 16 switches. Nodes numbered *0-15* are the switches and nodes numbered *16-31* are the devices.

between multiple veth interfaces. This is accomplished using the *brctl addbr* command, which creates a new bridge in each container. Once these bridges are created, the veth interfaces are added to their respective bridges using the *brctl addif* command. This process integrates the veth pairs with the bridges, mirroring physical switches' functionality in a real-world TSN network.

6.3.2 Step 2: Traffic Plan Generation and Distribution

The traffic plan in JSON format that was previously produced by the DFS executable contains the routing and scheduling information for all devices. From this traffic plan, we methodically extract information for each stream that includes parameters like offset, period, stream ID (also acting as VLAN ID), destination, frame size, and the intended route. The initial schedule generated for every device is expanded across one hypercycle, which involves adjusting the timing offsets following each frame's period and copying them to the respective device container. Furthermore, our setup permits the specification of the number of hypercycles for which the emulation will run.

We then configure the switch containers. This is achieved by constructing custom switching tables for each switch in the network, which contain information about the next-hop node correlated with each frame's VLAN ID. By doing so, we ensure that as Ethernet frames traverse a switch, they are correctly routed based on their VLAN IDs. To apply these routing rules to each switch, we generate bash scripts loaded with *ebtables* commands. These scripts are designed dynamically for each switch, reflecting the custom switching tables. This establishes the necessary forwarding rules within each switch, aligning with the predefined traffic routes.

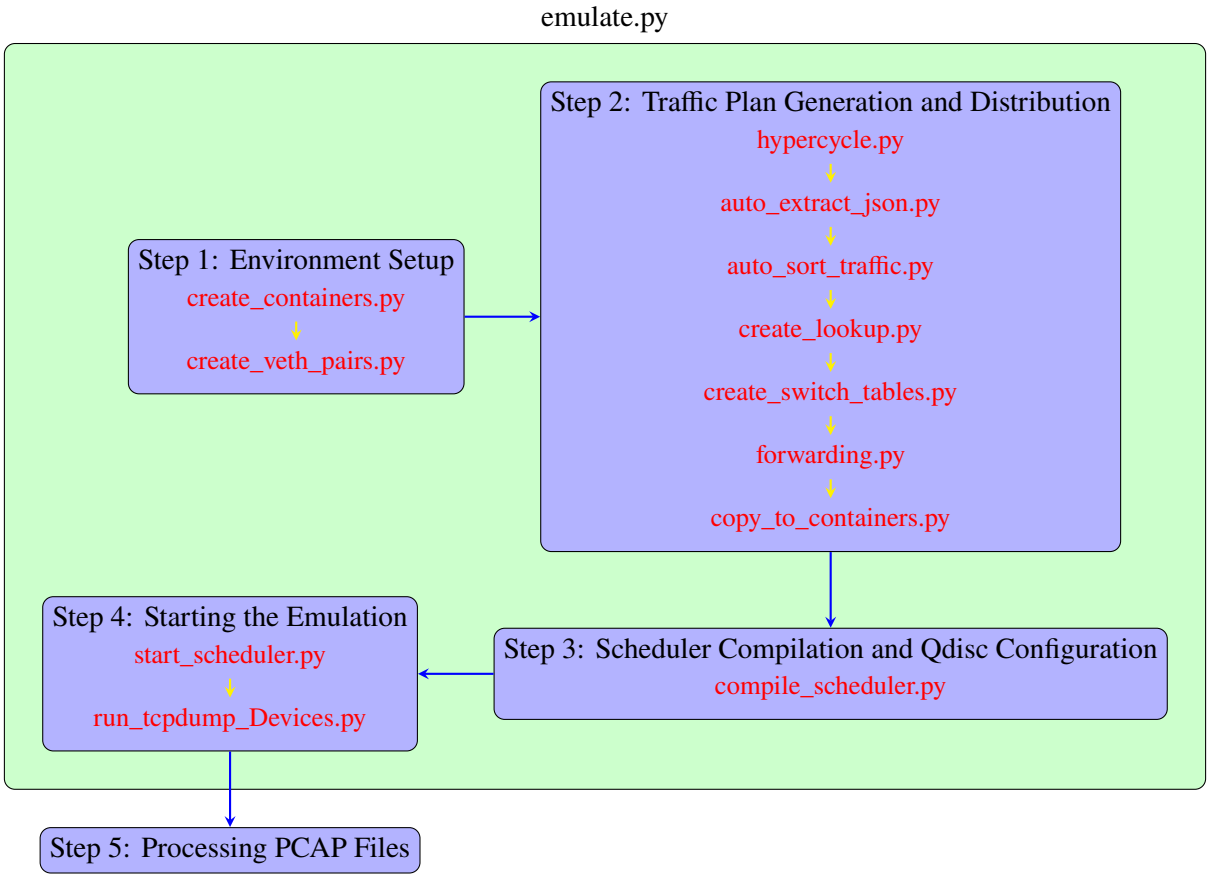

Figure 6.3: Workflow process for network configuration and data processing.

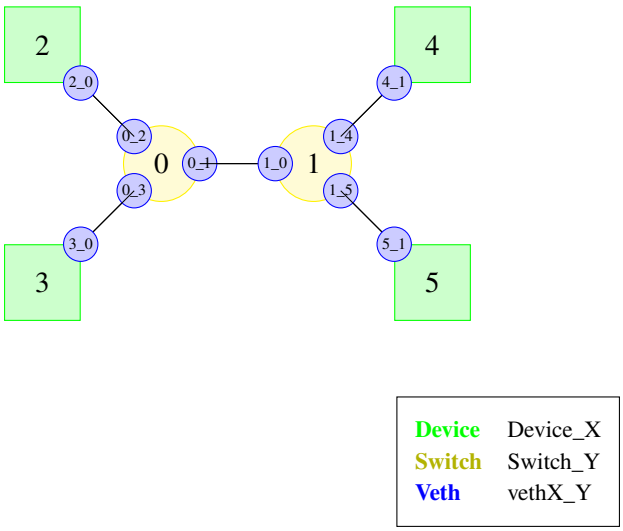

Figure 6.4: Network topology showing devices, switches, and their corresponding veth pairs after the completion of Step 1. The legend shows the naming convention used.

The scheduler carries out the creation and transmission of frames. A dynamically allocated array stores the Ethernet frames created at runtime. Each component of the Ethernet frame, ranging from the ethertype, PCP, DEI, and VLAN ID to the custom JSON-formatted payload, as well as the source and destination MAC addresses, are added to the frame through custom-developed functions. The scheduler also utilizes a circular queue for managing Ethernet frames. This design choice is strategic for cyclic transmission within a TSN, where frames are transmitted in multiple periodic hypercycles. By storing only one hypercycle's worth of frames in the queue and reiterating over the same set for subsequent hypercycles, we can achieve significant efficiency in terms of storage space. This approach is also resource-efficient and aligns with the cyclic nature of TSN schedules. However, it is essential to note that in real-world applications, where frames carry meaningful data, this method of pre-creating frames is not feasible. Nevertheless, this approach is valid and appropriate for this research, where we evaluate network behavior using dummy frames.

Since the same scheduler is copied to every device container, it must work on all of them, and the frames must be sent to the correct veth. Therefore, we must dynamically ascertain the veth interface of the device container it operates within. Before the frames are sent to the veth, we must serialize them. The serialization of Ethernet frames is a fundamental requirement in network communication – converting structured data (such as C structures) into a contiguous block of bytes suitable for transmission over a network. Serialization ensures that each frame is correctly formatted and encapsulated, preserving the integrity of the data as it traverses the virtual network.

We use Linux kernel features to control the exact transmission time of each Ethernet frame. This is achieved by creating a *struct msghdr* control message, which specifies the desired transmission time to the kernel. The *sendmsg* system call, employed for sending the frames, takes this control message and sends the data through a raw socket. This method allows for fine control over the frame transmission, specifying the exact nanosecond at which a frame should be dispatched from the veth. The accuracy of this approach will be evaluated in Chapter 7.

6.3.3 Step 3: Scheduler Compilation and Qdisc Configuration

This phase of our experimental setup is focused on compiling the scheduler program and configuring queuing disciplines on each device's veth. Qdisc configuration involves setting up MQPRIO and ETF qdiscs as described in 5.2.2.

6.3.4 Step 4: Running the Emulation

This step involves activating the scheduler in every device container with a pre-calculated epoch time, signifying the exact moment when the transmission of frames should commence. Concurrently, a *tcpdump* process is launched on each device container, which captures all incoming and outgoing Ethernet frames on the container's veth. The network traffic captured by *tcpdump* is saved into *pcap* files. These files are systematically named and stored, encapsulating detailed information about the network activities during the emulation to be used in Step 5.

6.3.5 Step 5: Processing PCAP Files

In this step, we process PCAP files to evaluate the performance of our system. This involves comparing expected and actual transmission times of frames to assess the precision of our schedule. We also measure the end-to-end latency of frames and analyze frame loss to identify optimum parameter values for qdisc configuration. We discuss and analyze the results in detail in the next chapter.

7 Performance Evaluation

In this chapter, we will discuss the evaluation metrics and Key Performance Indicators (KPIs) used to assess the system's performance, the methodologies employed for testing, and the performance of the deployed real-time scheduling system.

7.1 Evaluation Metrics and KPIs

We will establish the metrics and KPIs essential for assessing the performance of the real-time schedules deployed. The evaluation framework is designed to quantify the system's efficiency and reliability.

- *Frame Drops (FD)*: This metric measures the number of data frames lost during transmission. It indicates the network's reliability and robustness, especially in real-time systems where the loss of frames can lead to significant disruptions in data flow and system performance.
- *Frame Transmission Accuracy (FTA)*: This metric measures the deviation between the scheduled and actual transmission times of frames. It is crucial to assess the precision of schedule adherence during frame transmission.
- *Processing Delay (PD)*: This metric evaluates the latency introduced during frame switching in the network. This delay is critical in real-time systems where frame switching speed can significantly impact performance.
- *End-to-End Latency (E2EL)*: This encompasses the time a data frame takes to travel from the source to the destination. It is a comprehensive metric that includes all delays (propagation, processing, queuing, and transmission delays).

7.2 Results

Each KPI detailed in section 7.1 will be methodically explored, and the corresponding results for different network topologies and configurations will be presented in this section.

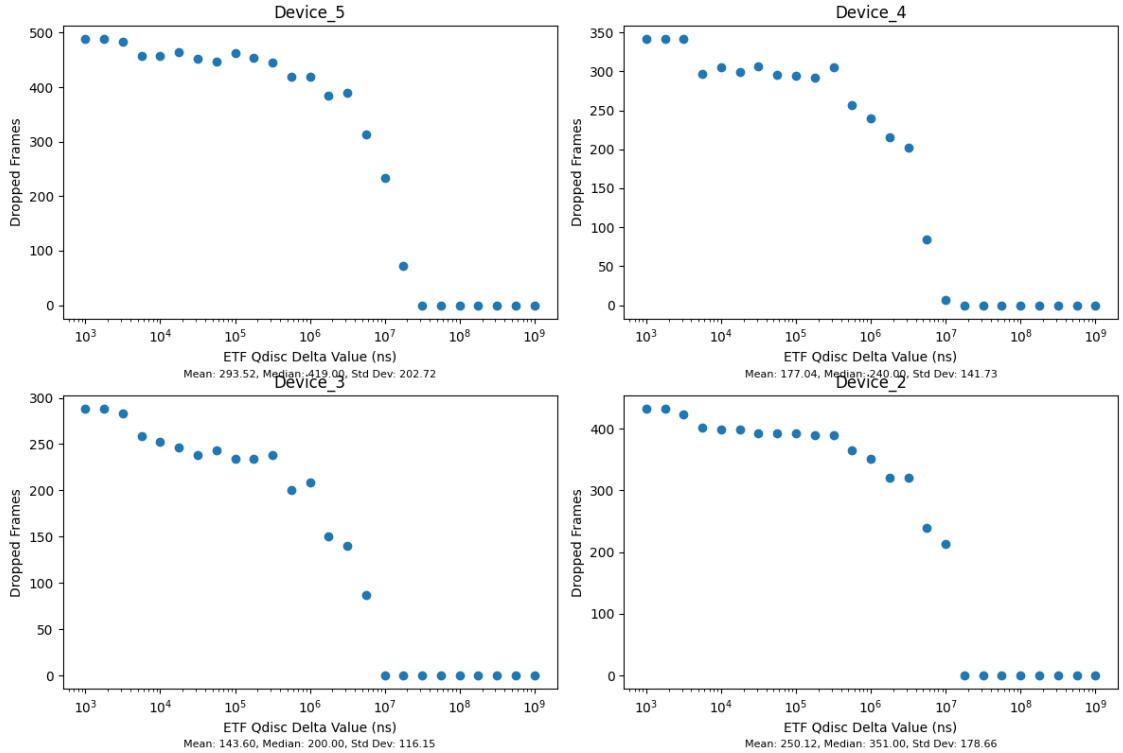

Figure 7.1: The impact of ETF qdisc delta value on frame loss for a small topology of 4 devices and 2 switches. Each plot illustrates an inverse relationship between the ETF qdisc delta value and frame loss. The frame loss is measured at a device container's *veth*. The X-axis uses the log scale.

7.2.1 Frame Drops

The ETF qdisc is designed to regulate the exact transmission time of frames from the *veth* and relies heavily on the value of the delta parameter. This parameter defines a window before the frame's scheduled transmission time, within which the system can adjust for scheduler latency.

The empirical data captured in Figures 7.1 and 7.2 displays the relationship between the ETF qdisc delta value and the number of dropped Ethernet frames on the y-axis. The ETF qdisc delta value is represented on a logarithmic scale on the x-axis as the delta value is varied across orders of magnitude ranging from 10^3 ns to 10^9 ns. The y-axis, on a linear scale, shows the number of dropped Ethernet frames. Figure 7.1 is for a small network with 4 devices and 2 switches (see Figure 6.1) and Figure 7.2 is for a medium-sized network with 16 devices and 16 switches (see Figure 6.2). Despite the differences in network topology, the pattern of frame loss variation with the ETF qdisc delta value is similar in both cases. This observation suggests that the frame loss characteristics of individual devices are primarily independent of the size of the network topology they are connected to. This is expected as the topology should not inherently affect the frame loss at a device. Thus, the frame loss as a function of the ETF qdisc delta is a device-specific behavior not significantly influenced by the complexity or scale of the underlying network topology.

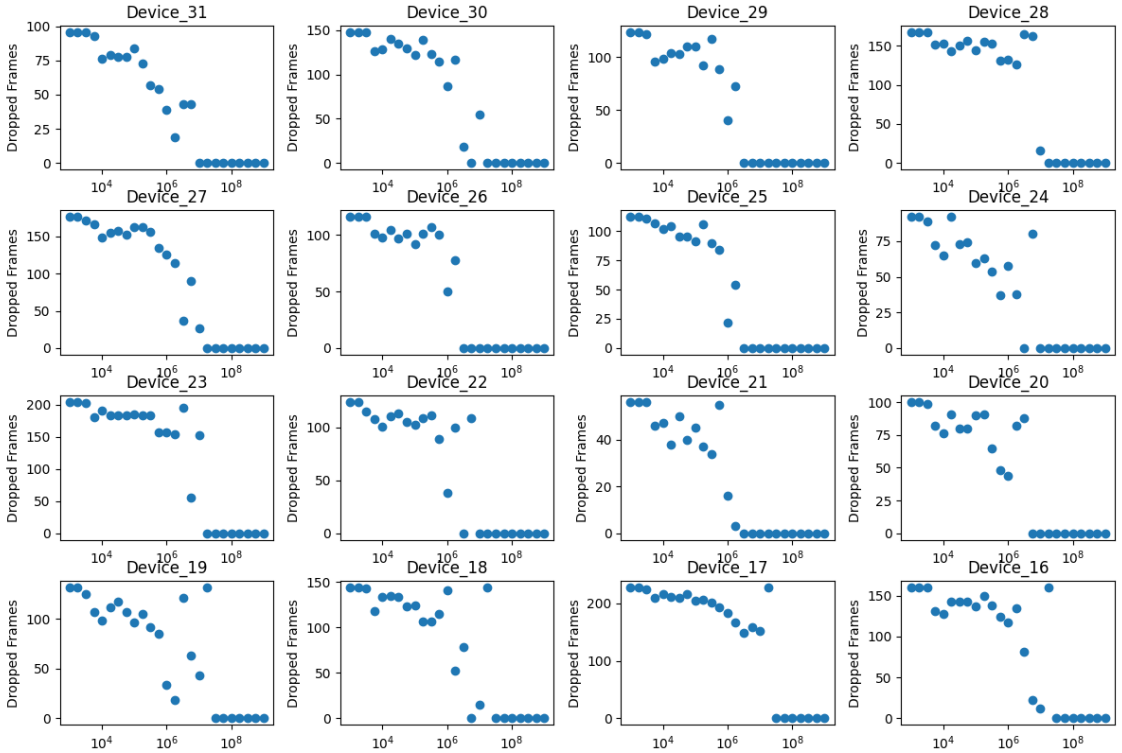

Figure 7.2: The impact of ETF qdisc delta value on frame loss for medium-sized topology of 16 devices and 16 switches. Each plot illustrates a distinct inverse relationship between the ETF qdisc delta value and frame loss. The frame loss is measured at a device container's *veth*. The X-axis uses the log scale.

The analysis reveals a pronounced increase in frame loss as the delta value decreases. With larger delta values, the system has a more considerable buffer to account for processing delays, thus mitigating the risk of frame expiration prior to dequeuing. It serves as a "fudge factor", allowing the network to accommodate the inherent latency of its traffic control mechanisms. Conversely, a smaller delta value provides a narrower timing window. This reduction in buffering window heightens the probability of frames being dropped, as the qdisc will discard any frames whose *txtime* has passed or expired while in the queue [SG18]. This is particularly evident in Figures 7.1 and 7.2, where the frame loss peaks at the lowest delta values. This suggests that the system's latency surpasses the minimal buffer provided, leading to frames being systematically dropped.

The analysis also indicates that a delta value greater than $5 * 10^7$ ns is a safe threshold value above which frame loss is virtually eliminated. This value balances the need to minimize frame loss while maintaining high frame transmission accuracy. Within this parameter range, the network's traffic control mechanisms are sufficiently buffered to manage inherent latencies and prevent frame expiration in the queue.

In a virtualized environment, the system's inability to accurately emulate the precise timing mechanisms of real hardware is the root cause of increased frame loss at lower delta values. The virtual system's intrinsic latencies, stemming from its non-deterministic nature, demand a more substantial buffer to compensate for these discrepancies. Consequently, the ETF qdisc must be configured with a larger delta value to provide the system with an adequate buffer to process and

transmit frames at the expected transmission time. Furthermore, the loss of frames at lower delta values for the ETF qdisc is an untenable outcome in real-time applications where the integrity of data transmission is paramount. This constraint necessitates the adoption of larger delta values. It is imperative to note that the delta value is not infinitely scalable. The ETF qdisc specifies an upper limit of 2 seconds for the delta value, beyond which it cannot be increased.

7.2.2 Frame Transmission Accuracy

The real-time scheduling system described in Chapter 6 was tested with a hyperperiod of 2,000 µs and an ETF qdisc delta value of $5 * 10^7$ ns for reasons mentioned in section 7.2.1. We simultaneously start the schedulers on all device containers and then record the measurements to emulate the real-world scenario.

Figure 7.3 delineates the temporal discrepancy between the anticipated and the actual frame transmission instances after running the experiment for one hypercycle. Each subfigure corresponds to a veth interface tied to a specific device container from which the data frames are dispatched. An initial observation reveals a consistent early transmission, approximately 10 ms before the prescribed schedule, during the commencement phase of the operation. This phenomenon, particularly pronounced during the initial stages of the transmission cycle, could be attributed to several factors inherent in the system's configuration and the network environment. However, as the hypercycle progresses, the difference between expected and actual transmission times appears to increase initially but eventually reduces.

The network stack may be configured to prioritize clearing buffers, which leads to an eagerness in frame transmission. This behavior can be exacerbated by configuring the network's QoS policies, which might inadvertently prioritize the early transmission of frames to avoid congestion and buffer overflow. The system compensates for non-deterministic latencies within the network by adjusting the frame release times. This proactive adjustment ensures that frames can still meet their expected release times despite unpredictable network behavior. It is important to note that in our setup utilizing the ETF qdisc, frames sent too late would be dropped rather than delayed, thereby preventing negative values on the y-axis in our data, which would indicate lateness in frame transmission [SG18].

The system is then run for multiple hypercycles, specifically 2 hypercycles in Figure 7.4 and 4 hypercycles in Figure 7.5. Initially, a noticeable temporal gap exists between the anticipated and actual transmission times of frames. However, as evidenced in Figure 7.5, this discrepancy narrows discernibly over four hypercycles. The actual transmission times of frames converge toward the expected transmission time within a margin of 20 µs, indicating high precision for a virtual testbed. This trend suggests an adaptive stabilization of the scheduling mechanism, achieving synchronization with the intended transmission timeline as the system reaches a steady operational state. The initial hypercycles serve as a calibration period during which the qdisc transmission mechanism gauges the network's performance characteristics and adjusts the transmission accordingly.

In contrast to the smaller topology previously discussed, the convergence of actual transmission time to the expected transmission time in a larger topology with 16 devices and 16 switches takes longer than four hypercycles. This extended calibration period can be seen in Figures 7.6, 7.7, and 7.8, which indicate a gradual narrowing of the temporal gap over successive hypercycles. The larger number of devices and switches introduces additional layers of complexity, leading to more significant

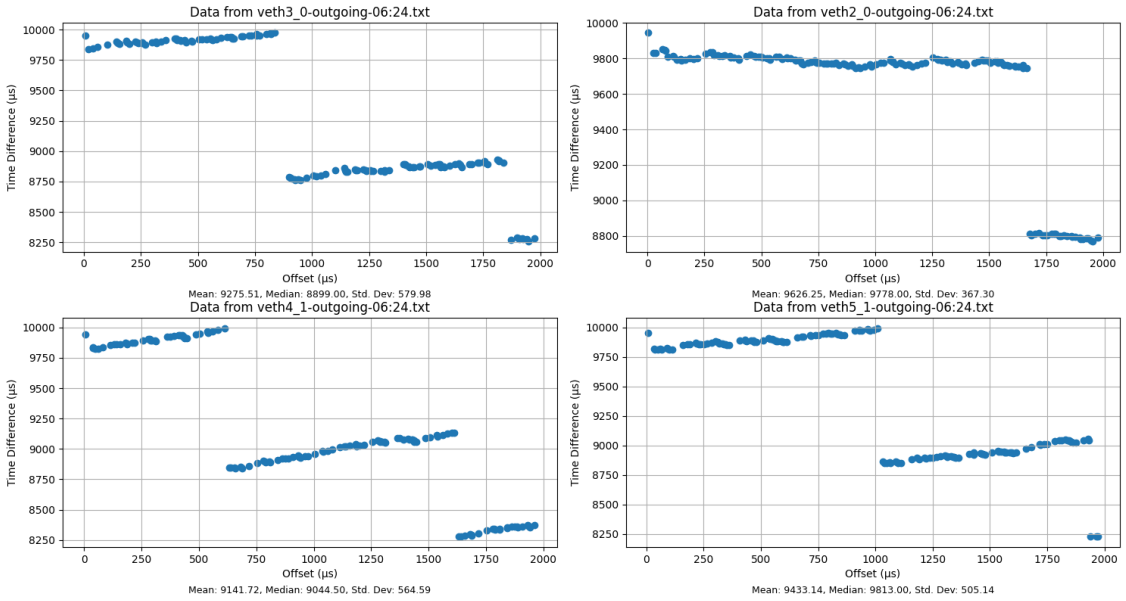

Figure 7.3: Discrepancies in frame transmission times across 4 devices. This figure illustrates the difference between expected and actual frame departure times, with each subplot corresponding to a distinct device for one hypercycle. The actual transmission time is before the expected transmission time.

initial deviations in frame release times. The convergence towards the prescribed transmission schedule exhibits a slower adaptive stabilization, suggesting the scheduling mechanism's adaptive nature is more challenged by the increased scale and complexity of the network topology. This also underscores the importance of considering network topology size and complexity when evaluating the performance and accuracy of real-time scheduling systems.

The emulation's duration was limited to four hypercycles to accommodate the constraints imposed by the ETF qdisc configuration. Notably, the delta value, set at $5 * 10^7$ ns for this series of tests, cannot be adjusted dynamically after the emulation commences, as detailed in section 7.2.1. Once set, this immutability of the delta value, combined with the requirement for a sufficiently large delta to emulate even a single hypercycle accurately, determines the fixed scope and scale of the testing period. Consequently, the empirical evaluation focused on the initial four hypercycles to observe the system's stabilization and the convergence of frame transmission times toward the expected schedule.

Suppose a 10^7 ns (10 ms) delta value is selected for a more straightforward calculation, and assume all devices are free from frame losses for this particular delta value. Frames for the first hypercycle must be queued within the ETF qdisc at least 10 ms ahead of the epoch to ensure their timely dispatch without loss. The subsequent hypercycles require frames 2 ms, which is the hyperperiod in this case, after the previous one. Thus, frames for the second hypercycle should be ready at -8 ms and the third at -6 ms, and this pattern continues.

However, given that the emulation initiates at the 0 ms mark, frames from multiple hypercycles accumulate in the buffer until this point. While a larger delta value might afford additional buffering capacity, an upper limit of 2 s for the delta value inherently limits the number of hypercycles that

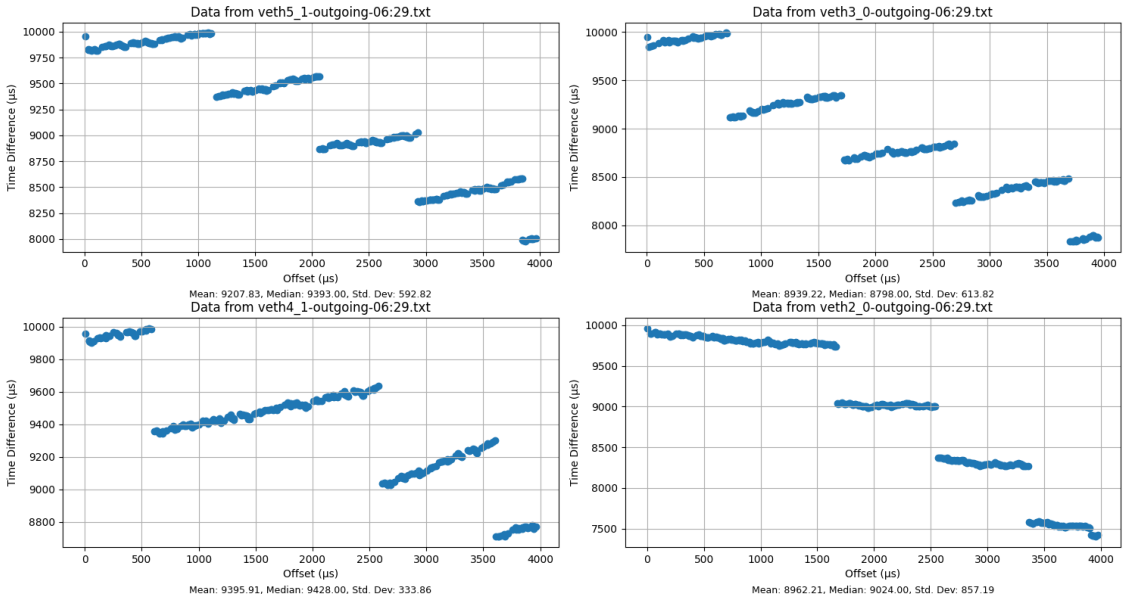

Figure 7.4: Discrepancies in frame transmission times across 4 devices. This figure illustrates the difference between expected and actual frame departure times, with each subplot corresponding to a distinct device for two hypercycles. The actual transmission time is before the expected transmission time.

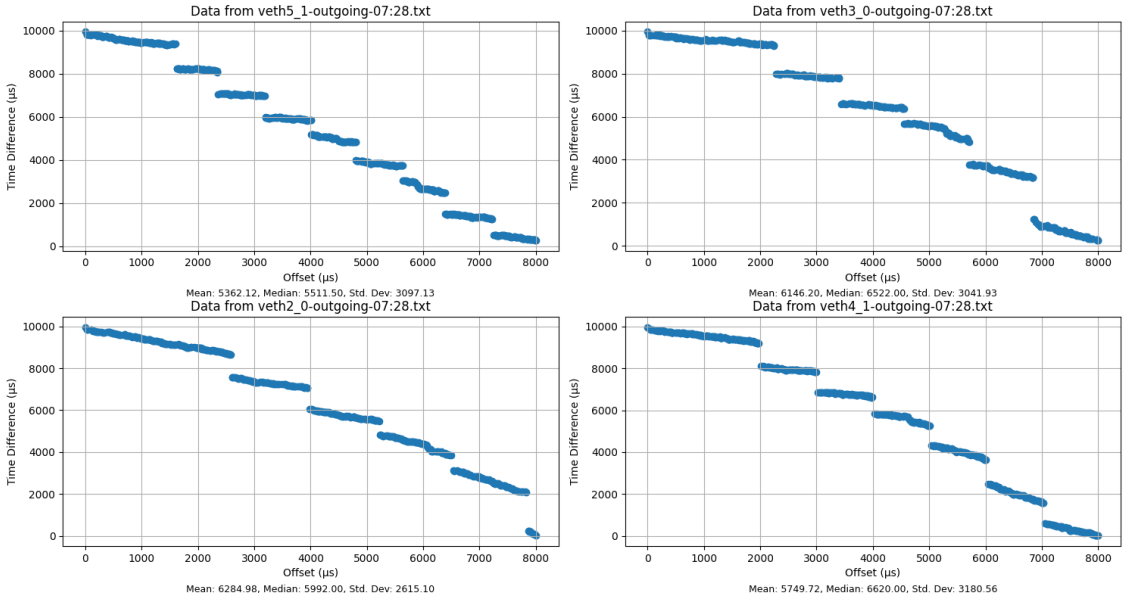

Figure 7.5: Discrepancies in frame transmission times across 4 devices. This figure illustrates the difference between expected and actual frame departure times, with each subplot corresponding to a distinct device for four hypercycles. The actual transmission time is before the expected transmission time.

can be effectively emulated. The strategy of dynamically loading frames after the commencement of the emulation has its practical limitations. Implementing a sleep interval within the scheduler program that utilizes the *sendmsg* function, managed by *nanosleep*, lacks the requisite precision for microsecond-level timing. Consequently, when the scheduler activates to dispatch new hypercycle frames, the ETF qdisc will drop them as the send time falls short of the delta threshold relative to the expected transmit time, leading to frame loss. This precision constraint necessitates a careful balance in selecting the delta value, ensuring the integrity of frame transmission and the realistic emulation of hypercycles within the virtual environment.

Figure 7.9 offers an illustrative view of the data flow from user space to the physical medium in a networking stack configured for TSN. Data frames originate from an application in user space and pass through the networking stack within the kernel space before reaching the physical layer for transmission. As frames are sent from the application using the *sendmsg* system call, they are first buffered in the socket buffer. We allocate the maximum possible size for the socket buffer using *setsockopt*. However, the system only permits a maximum socket buffer size of *33,554,432* bytes, which indicates a kernel-imposed limitation on the buffer size.

Once in the socket buffer, frames await to be processed by the MQPRIO root qdisc and the ETF child qdiscs. The ETF qdisc schedules the frames based on the txtime and holds them until it is time for their transmission. If the delta value is small, the ETF qdisc's buffering capacity will be constrained, leading to fewer frames that can be held before transmission. Conversely, a larger delta value permits a larger buffering capacity within the ETF qdisc, allowing more frames to be buffered in anticipation of their scheduled transmit times. The configured delta value thus directly influences the number of frames that can be buffered and, subsequently, the number of hypercycles that can be realistically emulated before the buffer overflows. The buffer's upper limit, influenced by the delta value, ultimately sets the practical limit for the number of hypercycles that can be preloaded in the buffer.

```
ip link set <veth_interface> txqueuelen <new_len>
```

Listing 7.1: Modifying the *qlen* of an interface buffer.

Furthermore, the interface buffer, whose size is modifiable via the command shown in Listing 7.1, also handles frames. In this experiment, the interface buffer was allocated a maximum size of *524,288* bytes. However, increasing the interface buffer's size from the default size of *1000* bytes does not affect the experimental results, as the critical factor remains the ETF qdisc's ability to schedule and buffer frames effectively according to the delta value. Thus, the restriction on socket buffer size and the impact of the delta value on the ETF qdisc's buffering capacity highlights the challenges in scheduling transmissions for multiple hypercycles in a virtual environment.

7.2.3 Processing Delay

Ethernet frames traversing a network encounter switches and experience processing delays at the switches. These delays accumulate based on the number of hops a frame takes, affecting the overall latency. Figures 7.10 and 7.12 present the network processing delays as measured across multiple hypercycles, with Figure 7.10 showcasing the results after two hypercycles and Figure 7.11 after three hypercycles for the topology in Figure 6.1. These figures illustrate the processing times within the network switches over these periods. The time-intensive nature of sequentially processing

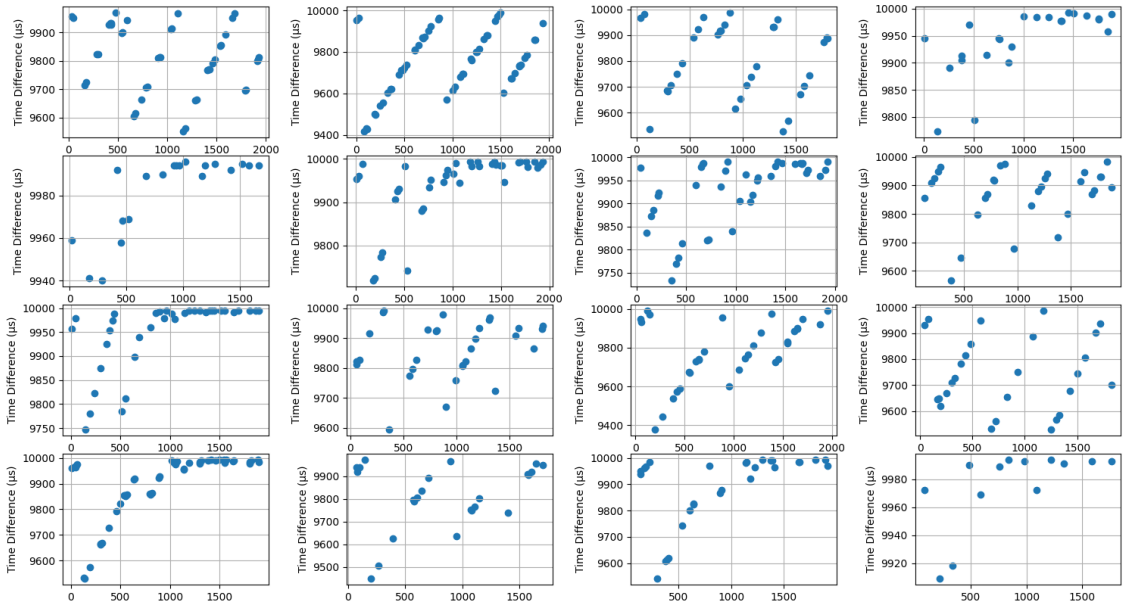

Figure 7.6: Discrepancies in frame transmission times across 16 devices. This figure illustrates the difference between expected and actual frame departure times, with each subplot corresponding to a distinct device for one hypercycle. The actual transmission time is before the expected transmission time.

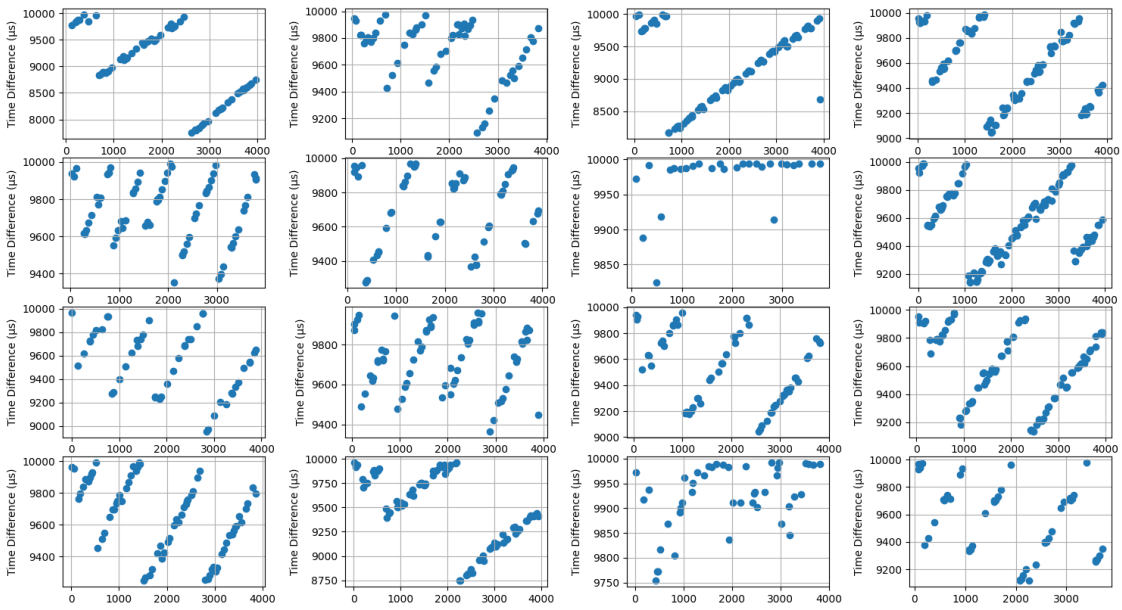

Figure 7.7: Discrepancies in frame transmission times across 16 devices. This figure illustrates the difference between expected and actual frame departure times, with each subplot corresponding to a distinct device for two hypercycles. The actual transmission time is before the expected transmission time.

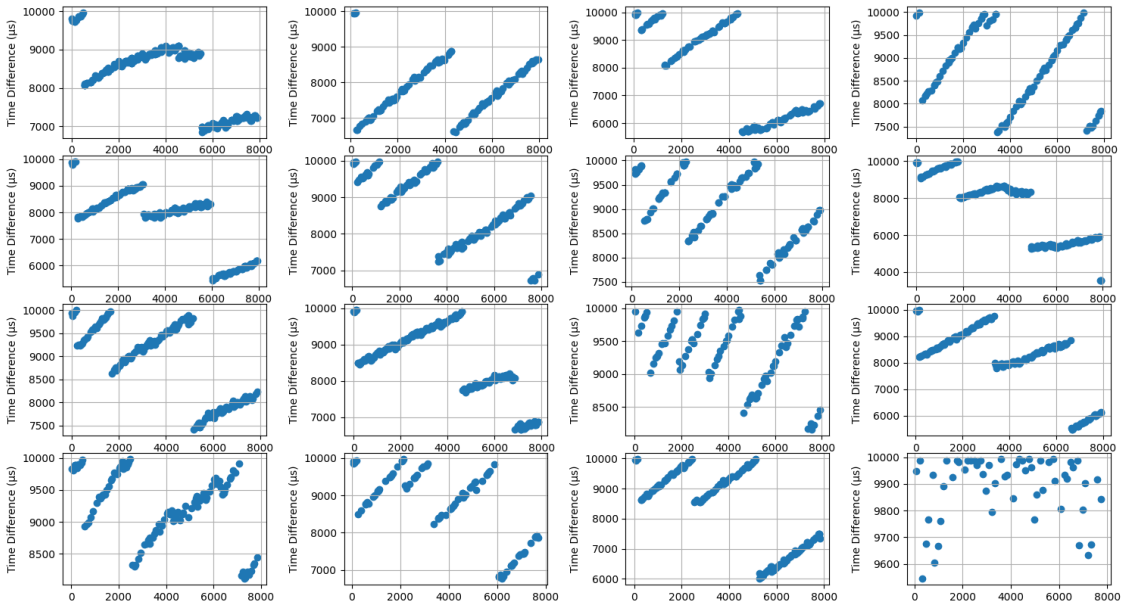

Figure 7.8: Discrepancies in frame transmission times across 16 devices. This figure illustrates the difference between expected and actual frame departure times, with each subplot corresponding to a distinct device for four hypercycles. The actual transmission time is before the expected transmission time.

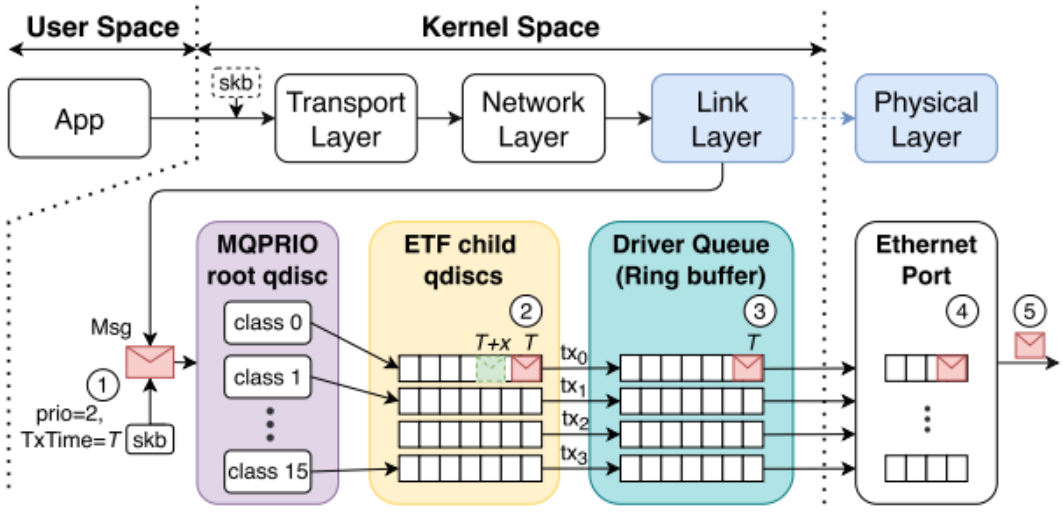

Figure 7.9: TSN data flow from application to physical layer [BRH+22].

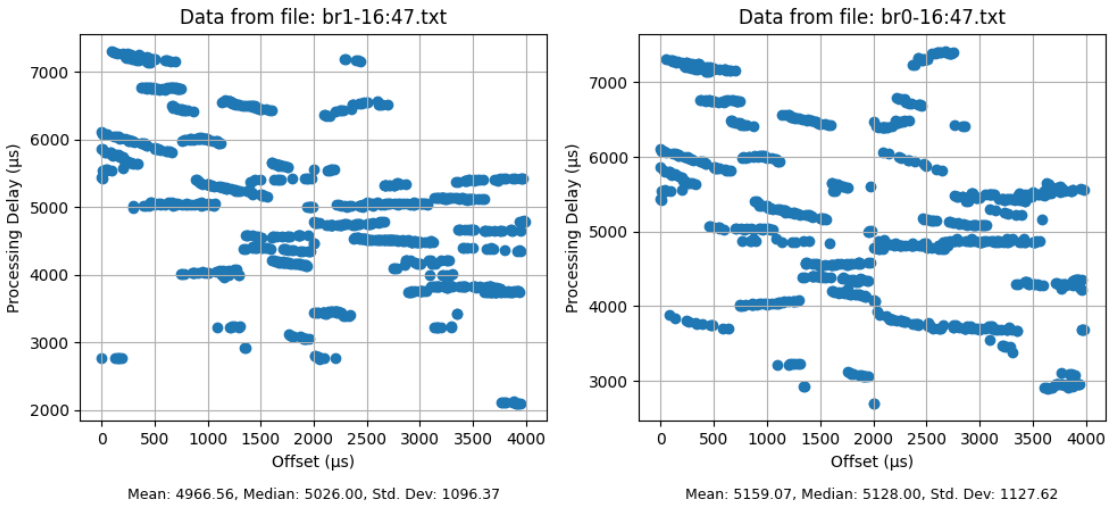

Figure 7.10: This figure depicts the processing delay experienced by frames in the switches over two hypercycles for the small topology.

ebtable rules is evident from the figures, where each frame's processing time increases due to the need to compare it against each rule until a match is found. This is especially pronounced when the rule set is extensive.

Moreover, various factors, such as hardware capabilities, the level of software optimization, the buffering strategy of the switch, and the prevailing traffic patterns, also affect the observed processing times. In Figure 7.10, elevated processing times in the order of milliseconds can be seen, which may suggest instances of network congestion or buffering within the switches as they manage the continual inflow of data frames. The same can be observed in Figure 7.11, where an additional hypercycle contributes to the complexity of the data, potentially increasing the latencies due to increased processing demand.

The distinction between Figures 7.10 and 7.11 lies in the number of hypercycles executed and the varying degrees of processing delay spread across these cycles. The data indicates that the switch processing delays stabilize somewhat as more hypercycles are executed. However, individual offsets still exhibit considerable variation, which could indicate sporadic congestion or buffering.

Figures 7.12, 7.13, and 7.14 illustrate the processing times within an expanded network topology incorporating 16 devices and 16 switches for one, two, and three hypercycles, respectively. While still in the millisecond range, the processing delays observed do not escalate linearly with the number of frames dispatched from the devices, suggesting an effective distribution of network traffic that mitigates the load on individual switches. In a larger topology, additional switches appear to offer a dispersion effect. Traffic that might otherwise congest a single switch is distributed across multiple nodes, diminishing the impact of any single set of rules on the overall latency. This decentralization of network traffic allows for a more balanced load across the network, reducing the processing delays at individual switches.

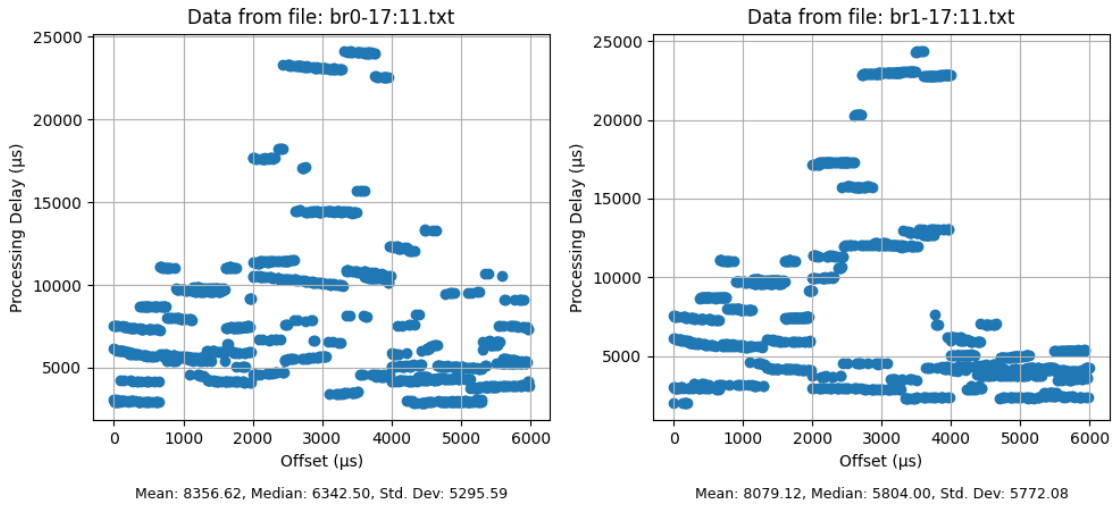

Figure 7.11: This figure depicts the processing delay experienced by frames in the switches over three hypercycles for the small topology.

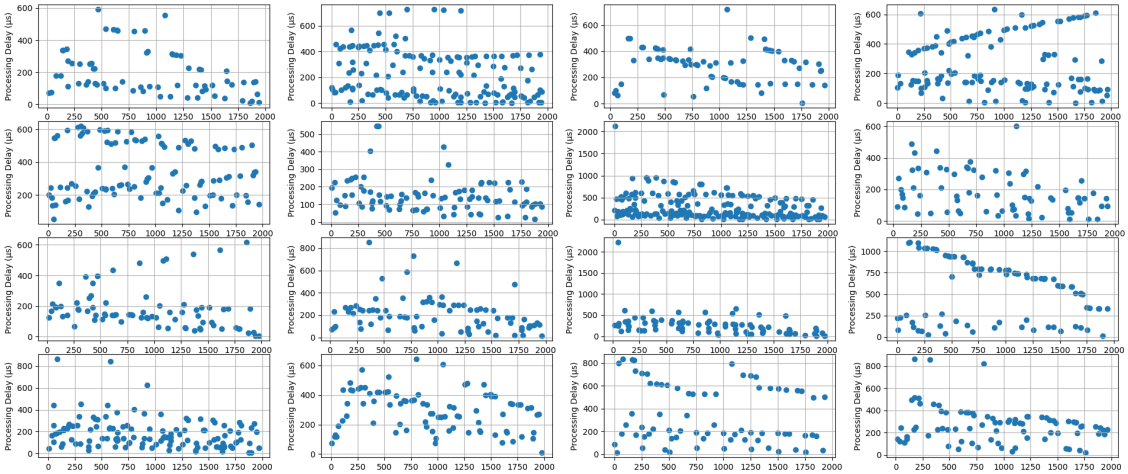

Figure 7.12: This figure depicts the processing delay experienced by frames in the switches over one hypercycle for the medium-sized topology.

7.2.4 End-to-End Latency

The E2E latency times for frames transmitted from a *veth* across a network are quantitatively represented in Figure 7.15, 7.16 and 7.17 for the topology in Figure 6.1, with Figure 7.15 showcasing the results for one hypercycle, Figure 7.16 for two hypercycles, and Figure 7.17 for three hypercycles. The observed progressive increase in E2E latency across the network reflects the compounded effect of several critical operational dynamics. Queuing delays become more pronounced as the transmission proceeds, especially as frames accumulate and network buffers near their capacity. This directly contributes to the latency buildup evident with higher offset frames.

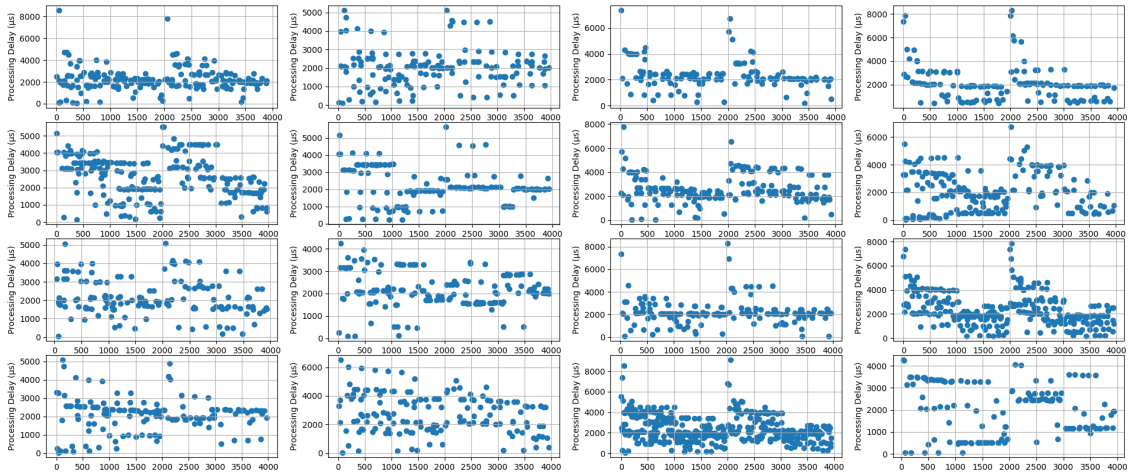

Figure 7.13: This figure depicts the processing delay experienced by frames in the switches over two hypercycles for the medium-sized topology.

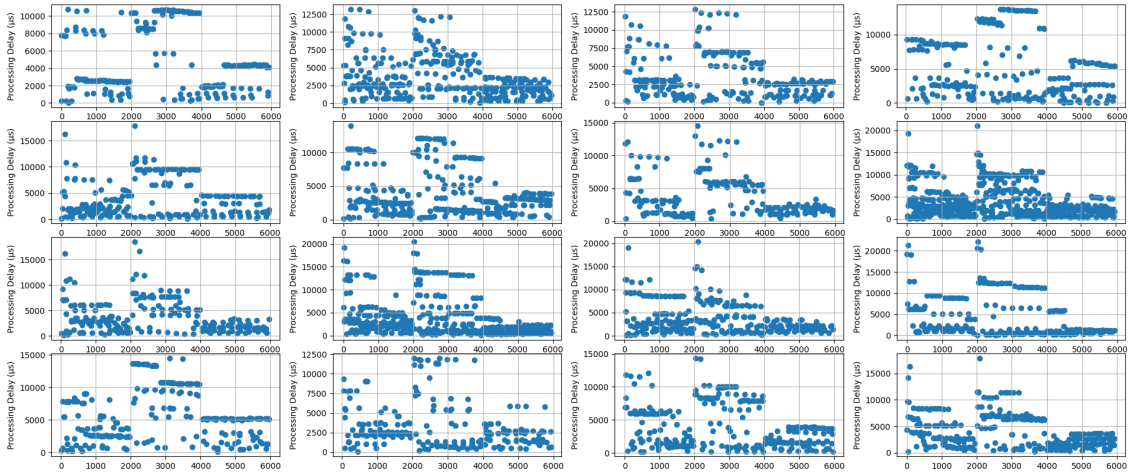

Figure 7.14: This figure depicts the processing delay experienced by frames in the switches over three hypercycles for the medium-sized topology.

Moreover, the gradual latency escalation suggests the presence of network congestion. Data frames invariably endure extended wait times as traffic increases, amplifying E2E latency. Concurrently, the processing time required by each node to handle frame headers and routing decisions accumulates, adding to the overall latency. This is exacerbated when frames traverse through variable network paths that differ in congestion levels or hop counts, introducing staggered latency increments.

Additionally, the network's approach to priority handling, possibly through QoS mechanisms, may result in varied E2E latencies depending on the assigned frame priorities. The limitations in network resources, such as the processing capabilities at intermediate nodes, intensify these latencies as the network's operational load approaches its upper thresholds. The culmination of these factors—queuing delays, transmission windows, network congestion, processing time, path variability, priority handling, and resource limitations—collectively manifest in the increased E2E latency for frames with larger offsets.

Figures 7.15, 7.16 and 7.17 elucidate two discernible trends within the plotted data, which can be fundamentally ascribed to the number of hops frames undergo. Frames navigating through a lesser number of switches, hence fewer hops, are depicted by the plots with lower latency values. This trend is attributable to the reduced processing overhead at each switch. Conversely, frames routed through a more extensive series of hops exhibit increased latency. Each additional hop through a switch introduces its latency due to processing and queuing, which, when aggregated, results in a significantly higher E2E latency, as evidenced by the plots with larger latency values. As visually segregated in the figures, this distinction in latency profiles underscores the impact of network topology and routing complexity on the performance of frame transmission within a real-time network system.

The E2E latency times for frames transmitted within an enlarged network topology present a nuanced portrait of latency dynamics, as captured in Figures 7.18, 7.19, and 7.20. The latency trends manifest a wider spectrum of outcomes influenced by the network's increased scale and complexity. While the foundational operational dynamics outlined previously still apply, the larger topology introduces additional layers of behavior that contribute to the latency profiles.

Figures 7.18, 7.19, and 7.20 reveal two distinct latency trends corresponding to the network's structure, consisting of only two switches (see Figure 6.1). Such frames encounter less cumulative processing delay overall. However, the variance between the subplots highlights another aspect. It reveals that as the number of hops for a frame increases, so does the cumulative latency. These individual delays, seemingly marginal on their own, aggregate to a considerable increase in E2E latency. This effect is illustrated by the plots showing higher latency values, correlating with the frames traversing more extensive switch sequences.

7.3 Discussion

In our detailed and comprehensive system analysis, we examined each KPI. We related them to the pertinent research questions, highlighting significant implications on the validity of using an emulation-based validation for the GFH algorithm in TSN schedules.

Beginning with frame drops, we observed a direct relationship between the ETF qdisc delta value and frame loss, where higher delta values resulted in fewer frame drops. This suggests that the system's latency exceeded the minimal buffer at lower delta values, leading to increased frame loss. A critical finding was that a delta value above $5*10^7$ ns effectively eliminated frame loss, suggesting a balance between schedule adherence and eliminating frame loss. These results have several implications for our research questions. They suggest significant inaccuracies in emulating real-world hardware timing mechanisms in a virtual testbed. This inconsistency is evident regardless of network size, raising questions about the precision of deployment strategies across various network scales.

Regarding frame transmission accuracy, we noted the early transmission of frames, indicating eagerness in frame dispatching. Over time, the actual transmission times converged toward the expected ones, demonstrating adaptive stabilization. However, this convergence was slower in larger topologies, underscoring the impact of network scale on scheduling accuracy. These observations challenge the accuracy of the virtual testbed and suggest the need for topology-specific adjustments to maintain precision.

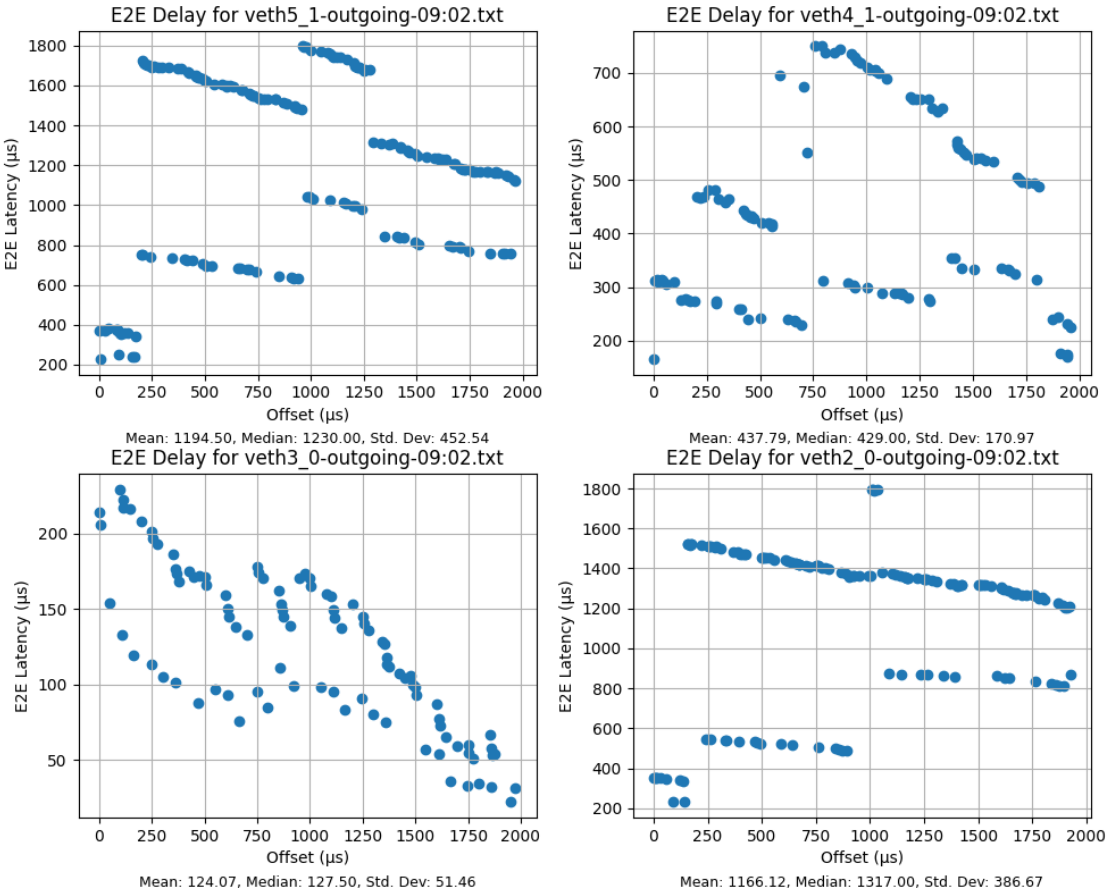

Figure 7.15: Cumulative E2E latency (E2EL) of frames transmitted from 4 devices. This figure visualizes E2EL of frames across 4 different *veths* for one hypercycle. Each data point in a subplot represents the E2EL of a frame transmitted from that particular *veth*.

Regarding processing delay, the sequential processing of *ebtable* rules in switches led to increased processing delays. Interestingly, larger network topologies distribute the traffic more effectively, mitigating some of these delays. This variation in processing delay efficiency with network topology size indicates the need for topology-specific deployment strategies.

Our results of end-to-end latency revealed that latency increased with the number of hops and network congestion and that latency profiles varied significantly with network topology. Again, this variability highlights the importance of having adaptive deployment strategies that can accommodate the scale and complexity of different network structures.

Based on these observations, several conclusions can be drawn, particularly concerning the emulation-based validation for the GFH algorithm [FGD+22]. There are notable inconsistencies in frame drops and transmission accuracy, with the emulation being unable to replicate precise timings and buffer requirements of real-world hardware. This issue is especially pronounced in larger topologies, suggesting scalability and complexity issues with the emulation-based approach. The increased processing delays and E2E latency variances, particularly in larger network topologies, further underscore the challenge of accurately emulating real-world network conditions.

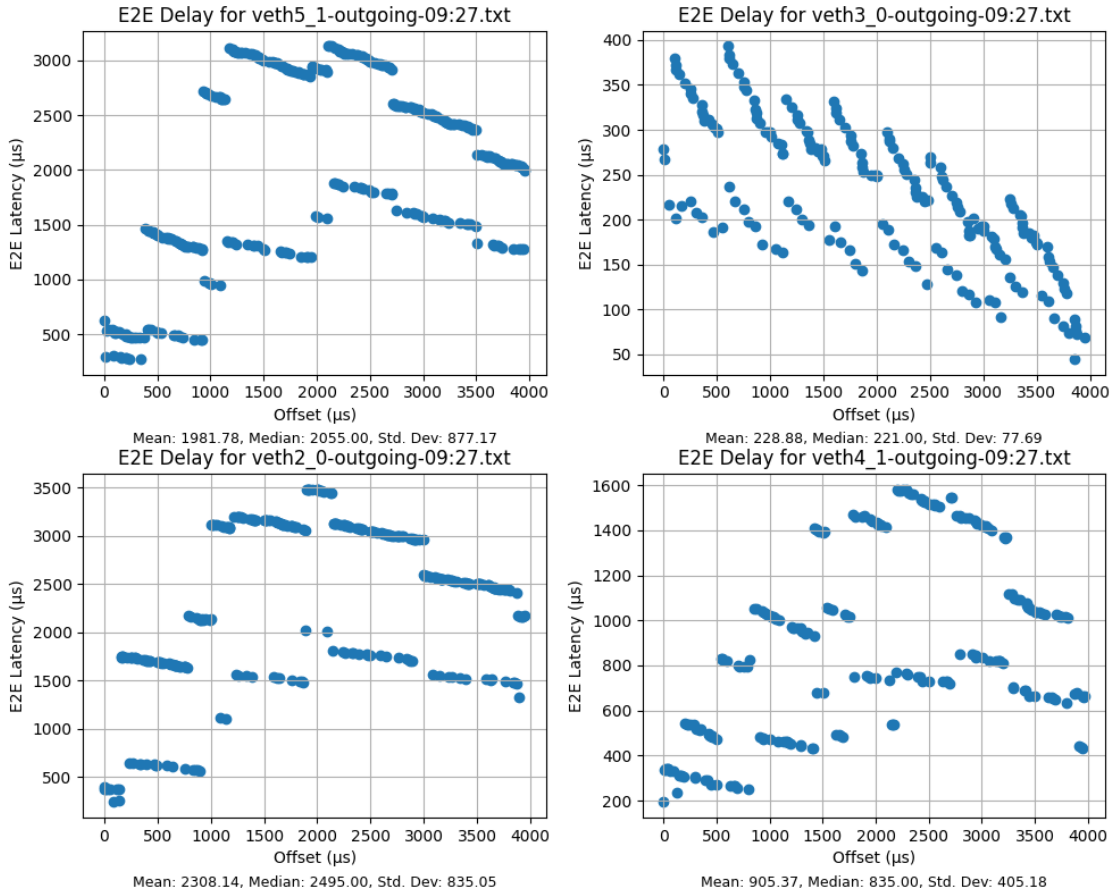

Figure 7.16: Cumulative E2E latency (E2EL) of frames transmitted from 4 devices. This figure visualizes E2EL of frames across 4 different *veths* for two hypercycles. Each data point in a subplot represents the E2EL of a frame transmitted from that particular *veth*.

There is also a discrepancy between the processing delays observed in our emulation and those reported in real-world systems. Research indicates that real-world processing delays of TSN switches range from 6.62 to 26 µs [LLM+20]. However, in our emulation, we consistently measured processing delays in the millisecond range, with some frames encountering delays as high as 25 ms. Furthermore, the GFH algorithm currently assumes fixed processing and propagation delays, which are not given in the emulation environment. Thus, the current emulation-based validation is ineffective for the GFH algorithm due to its inability to replicate real-world timing precision accurately and handle complex network topologies.

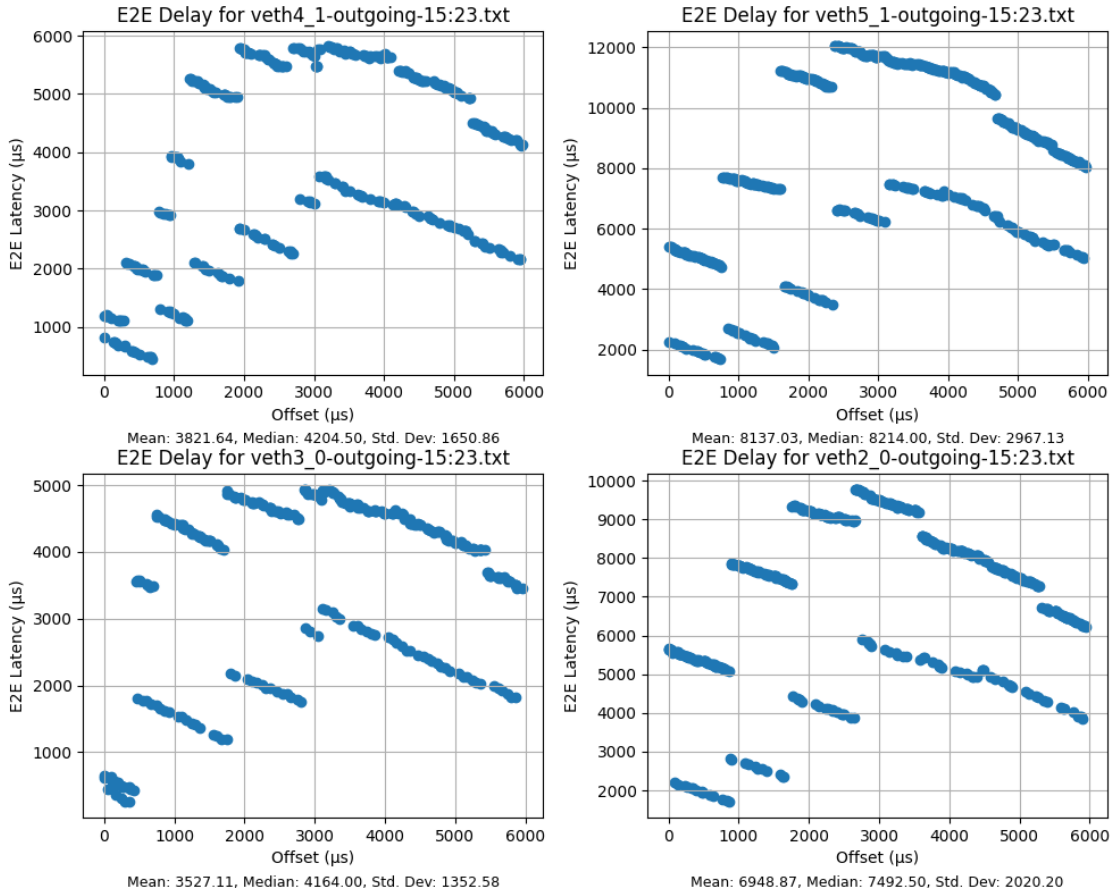

Figure 7.17: Cumulative E2E latency (E2EL) of frames transmitted from 4 devices. This figure visualizes E2EL of frames across 4 different *veths* for three hypercycles. Each data point in a subplot represents the E2EL of a frame transmitted from that particular *veth*.

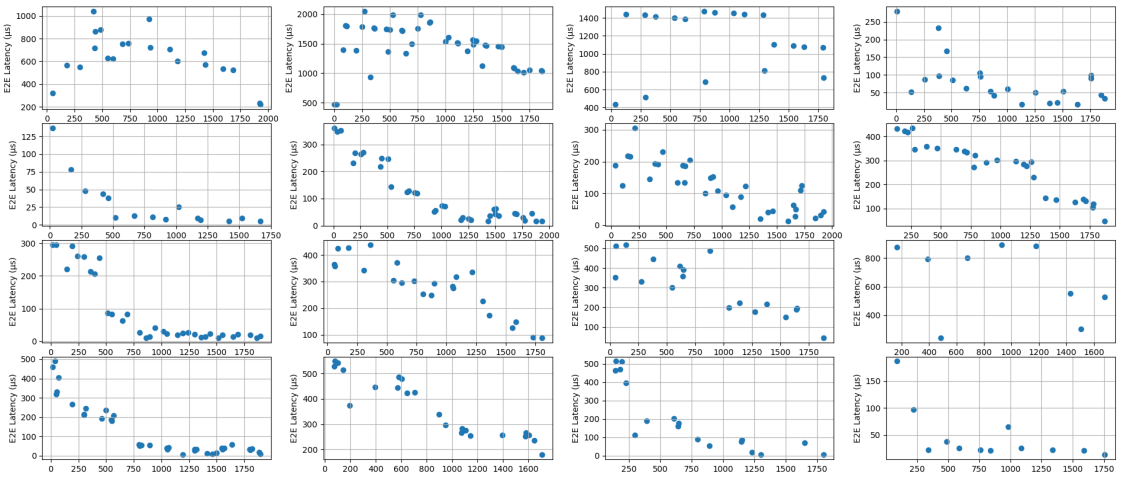

Figure 7.18: Cumulative E2E latency (E2EL) of frames transmitted from 16 devices. This figure visualizes E2EL of frames across 16 different *veths* for one hypercycle. Each data point in a subplot represents the E2EL of a frame transmitted from that particular *veth*.

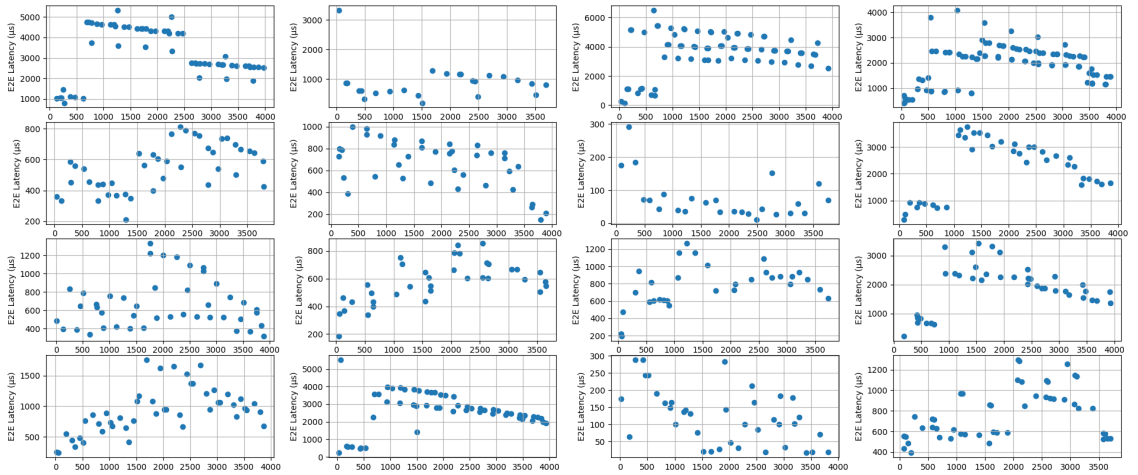

Figure 7.19: Cumulative E2E latency (E2EL) of frames transmitted from 16 devices. This figure visualizes E2EL of frames across 16 different *veths* for two hypercycles. Each data point in a subplot represents the E2EL of a frame transmitted from that particular *veth*.

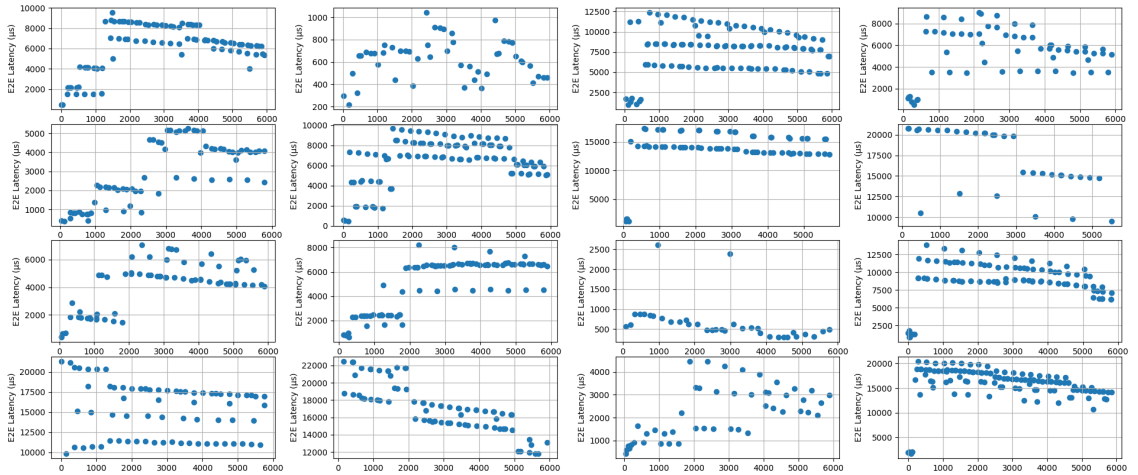

Figure 7.20: Cumulative E2E latency (E2EL) of frames transmitted from 16 devices. This figure visualizes E2EL of frames across 16 different *veths* for three hypercycles. Each data point in a subplot represents the E2EL of a frame transmitted from that particular *veth*.

8 Conclusion and Future Work

This thesis explored the intricacies of deploying and verifying real-time schedules within a virtualized environment, focusing on the automatic creation of a virtual testbed, the deployment of the schedule, and the empirical evaluation of its performance. The research tested the feasibility of replicating TSN behaviors in a virtual environment in a controlled and cost-effective manner before implementing it on real hardware.

The empirical tests revealed that the system requires a brief initial waiting period of a few hypercycles to stabilize before frames are transmitted close to their expected transmission times. Once this stabilization is achieved, the system demonstrates a precision of up to 20 µs in transmitting frames which is the best-case scenario observed after the initial stabilization period. Such a delay and variation in precision are typical characteristics of virtual testbeds and are generally considered acceptable within the constraints of virtual environments.

A larger network topology showed decreased processing times at individual switches due to the effective distribution of network traffic, highlighting the system's scalability. It is worth noting that the number of switches a frame traverses, especially in overloaded centrally located switches in a network, impacts the E2E latency significantly more than multiple hops in less congested switches located at the periphery of a network. This emphasizes the influence of network topology on overall performance.

An ETF qdisc delta value of $5*10^7$ ns was identified to minimize frame loss without compromising transmission accuracy, which is crucial for deploying real-time systems where the accuracy and reliability of frame transmission are paramount. The delta value, significantly larger than the hyperperiod used in our scheduling, introduces specific limitations. Frames must be prepared a delta time before the planned transmission, or they risk being dropped. The ETF qdisc achieves this by buffering packets until a time configurable through the delta option before their transmission deadline. This means the delta value acts as a buffer or "fudge factor" for the system's scheduler latency, allowing precise control over packet transmission timings. The qdisc will schedule its next wake-up time as the next txtime minus this delta value. This approach offers a "Launch Time" or "Time-Based Scheduling" feature, vital for shaping network traffic and ensuring data transmission precision.

In future work, enhancing the precision and applicability of virtual testbeds for real-time systems will involve a multifaceted approach. The transition to Real-Time Operating Systems (RTOS) could significantly improve timing accuracy for frame transmissions due to their efficient handling of time-critical tasks and the potential to reduce the initial stabilization period. Additionally, moving beyond *ebtables* to more efficient traffic control and management methods could address current processing speed limitations and reduce latency at the switch level. While the underlying hardware might be adequate for computational resources, integrating specialized hardware for TSN applications could be explored to refine processing and transmission efficiency further. Furthermore, longitudinal studies that involve extended empirical evaluations across varied and complex network

topologies would yield more profound insights into the scalability and robustness of these virtual testbeds. Integrating virtual testbeds with physical components to create hybrid systems could also help bridge the gap between emulation and real-world experiments. Finally, implementing and testing new and more advanced queuing disciplines and traffic management algorithms would enhance the network's capability to manage real-time traffic with high precision.

By addressing these areas, future research can build on the foundational work presented in this thesis to push the boundaries of what is achievable in virtualized testing of real-time systems, ultimately contributing to developing more reliable and efficient real-time communication networks.

Bibliography

[AHM19] A. A. Atallah, G. B. Hamad, O. A. Mohamed. "Routing and scheduling of time-triggered traffic in time-sensitive networks". In: *IEEE Transactions on Industrial Informatics* 16.7 (2019), pp. 4525–4534 (cit. on p. 25).

[And15] C. Anderson. "Docker [software engineering]". In: *Ieee Software* 32.3 (2015), pp. 102–c3 (cit. on p. 23).

[BE14] P. Blundell, B. Eckenfels. *route(8) - Linux manual page — man7.org*. https://man7.org/linux/man-pages/man8/route.8.html. 2014 (cit. on p. 22).

[Bol99] T. Bollinger. "Linux in practice: An overview of applications". In: *Software, IEEE* 16 (Feb. 1999), pp. 72–79. DOI: 10.1109/52.744572 (cit. on p. 20).

[BRH+22] M. Bosk, F. Rezabek, K. Holzinger, A. G. Marino, A. A. Kane, F. Fons, J. Ott, G. Carle. "Methodology and infrastructure for tsn-based reproducible network experiments". In: *IEEE Access* 10 (2022), pp. 109203–109239 (cit. on pp. 28, 51).

[BS19] L. L. Bello, W. Steiner. "A perspective on IEEE time-sensitive networking for industrial communication and automation systems". In: *Proceedings of the IEEE* 107.6 (2019), pp. 1094–1120 (cit. on pp. 12, 18).

[BTT+15] A. Bhatele, A. R. Titus, J. J. Thiagarajan, N. Jain, T. Gamblin, P.-T. Bremer, M. Schulz, L. V. Kale. "Identifying the culprits behind network congestion". In: *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2015, pp. 113–122 (cit. on p. 17).

[COCS16] S. S. Craciunas, R. S. Oliver, M. Chmelık, W. Steiner. "Scheduling real-time communication in IEEE 802.1 Qbv time sensitive networks". In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. 2016, pp. 183–192 (cit. on pp. 12, 18, 19).

[CP17] E. Casalicchio, V. Perciballi. "Measuring docker performance: What a mess!!!" In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. 2017, pp. 11–16 (cit. on p. 11).

[Eyc23] H. Eychenne. *iptables(8) - Linux manual page — man7.org*. https://man7.org/linux/man-pages/man8/iptables.8.html. 2023 (cit. on p. 22).

[Fas13] J. Fastabend. *tc-mqprio(8) - Linux manual page — man7.org*. https://man7.org/linux/man-pages/man8/tc-mqprio.8.html. 2013 (cit. on p. 33).

[FGD+22] J. Falk, H. Geppert, F. Dürr, S. Bhowmik, K. Rothermel. "Dynamic QoS-aware traffic planning for time-triggered flows in the real-time data plane". In: *IEEE Transactions on Network and Service Management* 19.2 (2022), pp. 1807–1825 (cit. on pp. 11, 26, 29, 35, 37, 56).

[Fin18] N. Finn. “Introduction to time-sensitive networking”. In: *IEEE Communications Standards Magazine* 2.2 (2018), pp. 22–28 (cit. on pp. 12, 18).

[GGT09] G. M. Garner, A. Gelter, M. J. Teener. “New simulation and test results for IEEE 802.1 AS timing performance”. In: *2009 International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*. IEEE. 2009, pp. 1–7 (cit. on p. 18).

[GSDP17] M. Gutiérrez, W. Steiner, R. Dobrin, S. Punnekkat. “Synchronization quality of IEEE 802.1 AS in large-scale industrial automation networks”. In: *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2017, pp. 273–282 (cit. on p. 18).

[Hed88] C. L. Hedrick. *RFC1058: Routing information protocol*. 1988 (cit. on p. 22).

[IEE18] IEEE. *802.1 Q-2018-IEEE standard for local and metropolitan area networks bridges and bridged networks*. 2018 (cit. on p. 19).

[KCB+08] F. N. van Kempen, A. Cox, P. Blundell, A. Kleen, B. Eckenfels. *ifconfig(8) - Linux manual page — man7.org*. https://man7.org/linux/man-pages/man8/ifconfig.8.html. 2008 (cit. on p. 21).

[KK79] P. Kermani, L. Kleinrock. “Virtual cut-through: A new computer communication switching technique”. In: *Computer Networks (1976)* 3.4 (1979), pp. 267–286 (cit. on p. 23).

[KS22a] H. Kopetz, W. Steiner. “Internet of Things”. In: *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Cham: Springer International Publishing, 2022, pp. 325–341. ISBN: 978-3-031-11992-7. DOI: 10.1007/978-3-031-11992-7_13. URL: https://doi.org/10.1007/978-3-031-11992-7_13 (cit. on p. 11).

[KS22b] H. Kopetz, W. Steiner. “Real-Time Communication”. In: *Real-Time Systems*. Springer International Publishing, 2022, pp. 177–200. DOI: 10.1007/978-3-031-11992-7_7. URL: https://doi.org/10.1007/978-3-031-11992-7_7 (cit. on pp. 11, 16).

[KS22c] H. Kopetz, W. Steiner. “The Real-Time Environment”. In: *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Cham: Springer International Publishing, 2022, pp. 1–29. ISBN: 978-3-031-11992-7. DOI: 10.1007/978-3-031-11992-7_1. URL: https://doi.org/10.1007/978-3-031-11992-7_1 (cit. on pp. 15, 16).

[Lam76] S. Lam. “Store-and-forward buffer requirements in a packet switching network”. In: *IEEE Transactions on Communications* 24.4 (1976), pp. 394–403 (cit. on p. 23).

[Lin11] Linux. *ebtables-nft(8) - Linux manual page — man7.org*. https://man7.org/linux/man-pages/man8/ebtables-nft.8.html. 2011 (cit. on p. 22).

[Lin23] Linux. *Linux Kernel Documentation - Networking*. 2023. URL: https://www.kernel.org/doc/html/v5.7/networking/index.html (cit. on p. 20).

[Lit11] M. Litvak. *ip(8) - Linux manual page — man7.org*. https://man7.org/linux/man-pages/man8/ip.8.html. 2011 (cit. on p. 21).

[Lit12] M. Litvak. *ip-route(8) - Linux manual page — man7.org*. https://man7.org/linux/man-pages/man8/ip-route.8.html. 2012 (cit. on p. 22).

[LLM+20] A. Larrañaga, M. C. Lucas-Estañ, I. Martinez, I. Val, J. Gozalvez. "Analysis of 5G-TSN integration to support industry 4.0". In: *2020 25th IEEE International conference on emerging technologies and factory automation (ETFA)*. Vol. 1. IEEE. 2020, pp. 1111–1114 (cit. on p. 57).

[Lub85] M. Luby. "A simple parallel algorithm for the maximal independent set problem". In: *Proceedings of the seventeenth annual ACM symposium on Theory of computing*. 1985, pp. 1–10 (cit. on p. 26).

[LWP+19] Z. Li, H. Wan, Z. Pang, Q. Chen, Y. Deng, X. Zhao, Y. Gao, X. Song, M. Gu. "An enhanced reconfiguration for deterministic transmission in time-triggered networks". In: *IEEE/ACM Transactions on Networking* 27.3 (2019), pp. 1124–1137 (cit. on p. 12).

[Mes18] J. L. Messenger. "Time-sensitive networking: An introduction". In: *IEEE Communications Standards Magazine* 2.2 (2018), pp. 29–33 (cit. on pp. 12, 18).

[Mil91] D. L. Mills. "Internet time synchronization: the network time protocol". In: *IEEE Transactions on communications* 39.10 (1991), pp. 1482–1493 (cit. on p. 17).

[NTA+19] A. Nasrallah, A. S. Thyagaturu, Z. Alharbi, C. Wang, X. Shao, M. Reisslein, H. Elbakoury. "Performance comparison of IEEE 802.1 TSN time aware shaper (TAS) and asynchronous traffic shaper (ATS)". In: *IEEE Access* 7 (2019), pp. 44165–44181 (cit. on pp. 18, 19).

[PSH+02] M. Pióro, Á. Szentesi, J. Harmatos, A. Jüttner, P. Gajowniczek, S. Kozdrowski. "On open shortest path first related network optimisation problems". In: *Performance evaluation* 48.1-4 (2002), pp. 201–223 (cit. on p. 22).

[RBA17] B. B. Rad, H. J. Bhatti, M. Ahmadi. "An introduction to docker and analysis of its performance". In: *International Journal of Computer Science and Network Security (IJCSNS)* 17.3 (2017), p. 228 (cit. on p. 24).

[RBP+21] F. Rezabek, M. Bosk, T. Paul, K. Holzinger, S. Gallenmüller, A. Gonzalez, A. Kane, F. Fons, Z. Haigang, G. Carle, et al. "EnGINE: Developing a flexible research infrastructure for reliable and scalable intra-vehicular TSN networks". In: *2021 17th International Conference on Network and Service Management (CNSM)*. IEEE. 2021, pp. 530–536 (cit. on p. 28).

[REC15] K. Rose, S. Eldridge, L. Chapin. "The internet of things: An overview". In: *The internet society (ISOC)* 80 (2015), pp. 1–50 (cit. on p. 11).

[rez23] rezabfil. *GitHub - rezabfil-sec/engine-framework — github.com*. https://github.com/rezabfil-sec/engine-framework. 2023 (cit. on p. 28).

[RLH06] Y. Rekhter, T. Li, S. Hares. *RFC 4271: A Border Gateway Protocol 4 (BGP-4) — rfc-editor.org*. https://www.rfc-editor.org/rfc/rfc4271. 2006 (cit. on p. 22).

[SCO18] W. Steiner, S. S. Craciunas, R. S. Oliver. "Traffic planning for time-sensitive communication". In: *IEEE Communications Standards Magazine* 2.2 (2018), pp. 42–47 (cit. on pp. 11, 25).

[SG18] J. Sanchez-Palencia, V. C. Gomes. *tc-etf(8) - Linux manual page — man7.org*. https://man7.org/linux/man-pages/man8/tc-etf.8.html. 2018 (cit. on pp. 34, 45, 46).

[SKKS11] T. Steinbach, H. D. Kenfack, F. Korf, T. C. Schmidt. “An extension of the OMNeT++ INET framework for simulating real-time ethernet with high accuracy”. In: *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*. 2011, pp. 375–382 (cit. on p. 28).

[SSH+18] E. Sisinni, A. Saifullah, S. Han, U. Jennehag, M. Gidlund. “Industrial internet of things: Challenges, opportunities, and directions”. In: *IEEE transactions on industrial informatics* 14.11 (2018), pp. 4724–4734 (cit. on p. 11).

[Tan81] A. S. Tanenbaum. “Network protocols”. In: *ACM Computing Surveys (CSUR)* 13.4 (1981), pp. 453–489 (cit. on p. 21).

[TG08] M. D. J. Teener, G. M. Garner. “Overview and timing performance of IEEE 802.1 AS”. In: *2008 IEEE international symposium on precision clock synchronization for measurement, control and communication*. IEEE. 2008, pp. 49–53 (cit. on p. 18).

[UAKF21] M. Ulbricht, J. Acevedo, S. Krdoyan, F. H. Fitzek. “Emulation vs. Reality: Hardware/-Software Co-Design in Emulated and Real Time-sensitive Networks”. In: *European Wireless 2021; 26th European Wireless Conference*. VDE. 2021, pp. 1–7 (cit. on p. 27).

[VH10] A. Varga, R. Hornig. “An overview of the OMNeT++ simulation environment”. In: *1st International ICST Conference on Simulation Tools and Techniques for Communications, Networks and Systems*. 2010 (cit. on p. 28).

[WAA+15] S. T. Watt, S. Achanta, H. Abubakari, E. Sagen, Z. Korkmaz, H. Ahmed. “Understanding and applying precision time protocol”. In: *2015 Saudi Arabia Smart Grid (SASG)*. IEEE. 2015, pp. 1–7 (cit. on p. 17).

[XZ09] M. Xue, C. Zhu. “The socket programming and software design for communication based on client/server”. In: *2009 Pacific-Asia Conference on Circuits, Communications and Systems*. IEEE. 2009, pp. 775–777 (cit. on p. 23).

[ZZX01] L. Zheng, L. Zhang, D. Xu. “Characteristics of network delay and delay jitter and its effect on voice over IP (VoIP)”. In: *ICC 2001. IEEE International Conference on Communications. Conference Record (Cat. No. 01CH37240)*. Vol. 1. IEEE. 2001, pp. 122–126 (cit. on p. 17).

All links were last followed on January 17, 2023.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, 18/01/2024

---

place, date, signature