Technology
Arts Sciences
TH Köln

Fraunhofer
IPT

Fraunhofer-Institut für
Produktionstechnologie IPT

Enabling Time-Aware Communication in Kubernetes through TSN-Based Networking Architecture

MASTER THESIS

by

Mahmuda Akter

submitted to obtain the degree of

MASTER OF SCIENCE (M.SC.)

at

TH KÖLN UNIVERSITY OF APPLIED SCIENCES
FACULTY OF INFORMATION, MEDIA AND ELECTRICAL ENGINEERING

Course of Studies

COMMUNICATION SYSTEMS AND NETWORKS

First supervisor: Prof. Dr. Andreas Grebe
TH Köln University of Applied Sciences

Second supervisor: Talib Sankal, M.Sc
Fraunhofer-Institut für Produktionstechnologie IPT

Cologne, May 2025

Contact details: Mahmuda Akter
(Matriculation No.: 11145749)
Theresienstrasse 64
50931 Köln
mahmuda.akter@smail.th-koeln.de

Prof. Dr. Andreas GREBE
Cologne University of Applied Sciences
Institute of Computer and Communication Technology
Betzdorfer Straße 2
50679 Köln
andreas.grebe@th-koeln.de

Talib SANKAL, M.SC
Fraunhofer-Institut für Produktionstechnologie IPT
Steinbachstraße 17
52074 Aachen, Germany
talib.sankal@ipt.fraunhofer.de

Abstract

This thesis presents a modular and standards-compliant framework for enabling deterministic, time-sensitive communication in Kubernetes-managed environments through the integration of TSN. It addresses the limitations of traditional container orchestration systems, which lack built-in mechanisms for bounded latency and jitter—requirements that are critical for Industry 4.0 and real-time control applications.

The proposed architecture introduces a Bash-based CNI plugin for configuring deterministic pod interfaces with Linux traffic control and the Time-Aware Shaper (taprio). A Python-based CNC coordinates switch-level GCL configurations using NETCONF and standardized YANG models. Synchronization across hosts and switches is achieved using the IEEE 802.1AS, implemented via LinuxPTP tools.

Experimental validation was conducted on a TSN-capable testbed, analyzing latency and jitter under various scheduling scenarios and network loads. Results showed that deterministic behavior can be achieved at the host level with taprio, skbedit but full end-to-end prioritization was limited by unreliable VLAN PCP propagation through physical interfaces and switches. These findings underscore the technical challenges of enforcing TSN guarantees across containerized and hardware layers.

Despite partial limitations, the system provides a reproducible, Kubernetes-native approach for TSN integration without relying on kernel bypass techniques like DPDK. This work contributes to bridging the gap between cloud-native orchestration and real-time industrial networking by offering both a validated prototype and insights for future research.

Keywords: Kubernetes, TSN, IEEE 802.1Qbv, IEEE802.1AS, CNC, deterministic networking, Industry 4.0, containerized systems, NETCONF/YANG, taprio.

Acknowledgements

I would like to sincerely thank Cologne University of Applied Sciences and Bonn-Rhein-Sieg University of Applied Sciences for providing an enriching academic environment. The knowledge and experiences gained from inspiring professors and collaborative peers have been invaluable to my professional and personal development.

I am especially grateful to Talib Sankal, my thesis supervisor at Fraunhofer-Institut für Produktionstechnologie IPT, for his continuous support and valuable insights throughout this research. His guidance played a key role in shaping the direction of this work. Additionally, my time at Fraunhofer IPT offered a hands-on and rewarding experience that greatly contributed to the success of this study.

I would also like to extend my appreciation to Prof. Dr. Andreas Grebe from the Institute of Computer and Communication Technology at Cologne University of Applied Sciences. His academic mentorship and constructive feedback were crucial in refining my research.

My thanks also go to Fraunhofer IPT for providing access to their lab facilities, which enabled me to conduct experiments and collect essential data, further strengthening this study.

Lastly, I am deeply thankful to my family for their constant encouragement and patience throughout this journey. A special thank you to my husband, whose unwavering support helped me stay focused and motivated. This work would not have been possible without the encouragement and assistance of those around me.

Thank you all for your invaluable contributions.

Contents

Abstract iii

Acknowledgements v

List of Figures xi

List of Tables xiii

1 Introduction 1
- 1.1 Motivation and Problem Statement . . . 1
- 1.2 Scope of the Thesis . . . 2
- 1.3 Research Objectives . . . 2
- 1.4 Thesis Structure . . . 2

2 Technical Background 5
- 2.1 Operational Technology . . . 5
- 2.2 Industry 4.0 . . . 5
- 2.3 Kubernetes . . . 7
  - 2.3.1 Kubernetes Scheduling Architecture . . . 8
  - 2.3.2 Kubernetes Networking and Container Network Interface . . . 9
- 2.4 Time-Sensitive Networking . . . 10
  - 2.4.1 Time-Sensitive Networking (TSN) Standards . . . 11
  - 2.4.2 TSN Control Architecture: CNC and CUC . . . 12
  - 2.4.3 Time Synchronization in TSN Systems . . . 13
- 2.5 Existing CNI Plugins and Their Limitations . . . 13
- 2.6 Enabling Tools and Technologies . . . 14

3 Design and Concept Methodology 15
- 3.1 Research-Driven Methodological Approach . . . 15
- 3.2 System Architecture and Integration Layers . . . 15
- 3.3 Key Design Considerations . . . 17
- 3.4 Custom CNI Strategy . . . 17
- 3.5 Time Synchronization Concept . . . 18
- 3.6 Traffic Scheduling Strategy . . . 20
- 3.7 Network Configuration and Resource Orchestration via CNC and CUC 21
- 3.8 Use Case Application and Scope . . . 21

4 Implementation Overview 23
- 4.1 Hardware Setup . . . 23
- 4.2 Experimental Platform . . . 23
- 4.3 Testbed Setup Overview . . . 24
  - 4.3.1 Scenario 1: Pod-to-Pod Communication with TSN . . . 25
  - 4.3.2 Scenario 2: Pod-to-Pod Communication without TSN . . . 25

4.3.3 Scenario 3: Device-to-Device Communication via TSN 26
4.3.4 Scenario 4: Direct Device-to-Device Communication (Bare-Metal) 26
4.3.5 Testbed Summary and Configuration 27
4.4 Kubernetes Cluster Setup 27
4.4.1 Custom CNI Plugin Implementation 28
4.5 Multus Installation and Configuration 32
4.5.1 Macvlan Setup for Secondary Interface 33
4.6 CNC and Switch Configuration via NETCONF 35
4.7 Experimental Traffic and Measurement Setup 36

5 Validation Overview **39**
5.1 Validation Goals and Scope 39
5.2 Extended Experimental Communication Scenarios 39
5.2.1 Scenario 1: Baseline VLAN Priority Communication (Without Time-Aware Scheduling) 40
5.2.2 Scenario 2: Switch-Level Time-Aware Scheduling 40
5.2.3 Scenario 3: Full TSN Scheduling (Pod + Switch Synchronization) 41
5.3 Measurement Methodology 42
5.3.1 Measurement Architecture 42
5.3.2 Measurement Tools and Software Environment 43
5.3.3 Latency and Jitter Calculation 44
Latency Calculation 44
Jitter Calculation 45
Key Metrics Evaluated 46
5.3.4 Measurement Procedure 46

6 Results and Analysis **47**
6.1 Measurement Scenarios and Goals 47
6.2 Overview of Measurement Method and Result Analysis 48
6.3 Bare-Metal Direct Connection Measurement 49
6.3.1 Latency Analysis 49
6.3.2 Jitter Analysis 50
6.3.3 Statistical Summary 52
6.3.4 Measurement Table 53
6.3.5 Interpretation 53
6.4 Scenario 1: Baseline VLAN Priority Communication (Without Time-Aware Scheduling) 54
6.4.1 Results 54
Without Background Load (iperf) 54
With Background Load (iperf) 54
6.4.2 Interpretation 54
6.5 Scenario 2: Switch-Level Time-Aware Scheduling (TAS) Enabled 54
6.5.1 Results 54
Without Background Load (iperf) 54
With Background Load (iperf) 55
6.5.2 Interpretation 55
6.5.3 Scenario 3: End-to-End Time-Aware Scheduling 55
6.5.4 Detailed Analysis of last three Results 56

7 Discussion **59**
7.1 Framwork Design Discussion 59

7.2 Testbed Reflection 60
7.3 Measurement Reflection 62
7.4 Observed Limitations and Insights 63

8 Conclusion **65**
8.1 Conclusion 65
8.2 Future Work 65

References **67**

Declaration of Authorship **71**

List of Figures

2.1 Components of a Smart Factory in Industry 4.0 6
2.2 Kubernetes Architecture Overview 8
2.4 Core Features of TSN 10
2.5 Example Amplify Build setting 11
2.6 TSN 802.1Qcc Architecture 12

3.2 Diagram of Multus, custom CNI and second CNI integration 18
3.3 Precision Time Protocol (gPTP) message exchange diagram 19
3.4 Working principle of IEEE 802.1Qbv Time-Aware Shaper 20

4.1 Front view of the testbed with two PCs, TSN switches, and non-manageable switch 24
4.2 Rear view showing of the testing platform 24
4.3 Pod-to-Pod Communication across TSN Cluster 25
4.4 Pod-to-Pod Communication without TSN 25
4.5 Device-to-Device TSN Communication 26
4.6 Direct Bare-Metal Communication Setup 26
4.7 Kubernetes Node Status 28
4.8 Master Node CNI lifecycle log 30
4.9 Worker Node CNI lifecycle log 30
4.10 CoreDNS validation 31
4.11 Multus DaemonSet status 32
4.12 macvlan interface in pod 33
4.13 macvlan pod ping to host 34
4.14 macvlan pod-to-pod connectivity 34
4.15 NETCONF read from TSN switch 35
4.16 Python CNC NETCONF patch 36
4.17 PTP grandmaster clock log 37
4.18 PTP client clock synchronization 37

5.1 Baseline VLAN Priority Communication Setup (Scenario 1) 40
5.2 Switch-Level Time-Aware Scheduling (Scenario 2) 41
5.3 Switch-Level Time-Aware Scheduling (Scenario 2) 42
5.4 iperf3 traffic generation from sender 44
5.5 iperf3 server receiving traffic 45

6.1 Latency Scatter Plot: Bare-Metal Host-to-Host 49
6.2 Latency Histogram: Bare-Metal Host-to-Host 50
6.3 Jitter Scatter Plot: Bare-Metal Host-to-Host 51
6.4 Jitter Histogram: Bare-Metal Host-to-Host 51
6.5 Statistical Summary of Latency and Jitter 52

List of Tables

5.1 Measurement tools and their purposes. 43

6.1 Latency and Jitter Statistics for Bare-Metal Direct Connection 52
6.2 Summary of Maximum Latency and Maximum Jitter Across Test Scenarios 53

List of Abbreviations

OT	**O**perational **T**echnology
IT	**I**nformation **T**echnology
IoT	**I**nternet **oT**of Things
TSN	**T**ime **S**ensitive **N**etworking
K8	**K**ubernetes
CNI	**C**ontainer **N**etworking **I**nterface
CNC	**C**entralized **N**etwork **C**onfiguration
CUC	**C**entralized **U**ser **C**onfiguration
NETCONF	**NET**work **CONF**iguration
RESTCONF	**R**epresentational **S**tate **T**sansfer **CONF**iguration
YANG	**Y**et **A**nother **N**ext **G**eneration
NIC	**N**etwork **I**nterface **C**ard
PTP	**P**recision **T**ime **P**rotocol
CNCF	**C**loud **N**ative **C**omputing **F**oundation
CRI	**C**ontainer **R**untime **I**nterface
YAML	**Y**et **A**nother **M**arkup **L**anguage
PLCs	**P**rogrammable **L**ogic **C**ontrollers
DCS	**D**istributed **C**ontrol **S**ystems
SCADA	**S**upervisory **C**ontrol **D**ata **A**cquisition
NAT	**N**etwork **A**ddress **T**ranslation
Https	**H**yper **T**ext **T**ransfer **P**rotocol **S**ecure
VxLAN	**V**irtual **E**xtensible **L**ocal **A**rea **N**etwork
VLAN	**V**irtual **L**ocal **A**rea **N**etwork
PCP	**P**riority **C**ode **P**oint
QoS	**Q**uality **O**f **S**ervice
gPTP	**g**enarilized **P**recision **T**ime **P**rotocol
GCLs	**G**ate **C**ontrol **L**ists

Chapter 1

Introduction

1.1 Motivation and Problem Statement

The adoption of Industry 4.0 and smart manufacturing concepts has created a critical need for real-time, deterministic communication between distributed industrial devices. Applications such as closed-loop control, predictive maintenance, and autonomous manufacturing processes rely on strict timing guarantees to operate reliably. However, traditional cloud-native platforms like Kubernetes, although offering scalability and resilience, inherently lack mechanisms for guaranteeing bounded latency and minimal jitter. Failures in timely communication can result in production disruptions, safety risks, and reduced operational efficiency.

Time-Sensitive Networking (TSN) extends Ethernet by introducing mechanisms for synchronized, time-aware traffic scheduling, enabling industrial networks to meet strict timing requirements without relying on proprietary technologies [Nas+19]. Integrating TSN capabilities into Kubernetes environments would allow industries to combine the flexibility of container orchestration with the determinism required for real-time control, bridging the longstanding gap between information technology (IT) and operational technology (OT) systems.

Several research efforts have addressed low-latency orchestration within containerized environments. Early work by Toka [Tok21] and Eidenbenz et al. [EPR20] focused primarily on optimizing Kubernetes scheduling strategies and CPU resource allocation for latency-sensitive edge applications, but did not address network-layer determinism. Broader studies on TSN and Deterministic Networking (DetNet) standards [Kir+23] have emphasized the importance of bounded-latency networking but have not extended these principles to containerized orchestration platforms. Attempts to enhance Kubernetes networking, such as KuberneTSN, introduced user-space system call interception for time-aware traffic control [BHS+18], bypassing the standard Container Network Interface (CNI) model and lacking tight integration with Linux kernel queuing mechanisms critical for TSN compliance. Similarly, SDN-based orchestration platforms like NEON [Pla25] focused on centralized TSN configuration but were limited to simulated control-plane demonstrations without real Kubernetes-native integration.

Consequently, despite partial solutions across different layers, a gap remains: a Kubernetes-native, standards-compliant, and hardware-validated architecture capable of enabling deterministic communication for containerized workloads synchronized with TSN-capable network infrastructure is still lacking.

This thesis aims to address this gap by extending Kubernetes with deterministic, time-sensitive communication capabilities through the integration of TSN mechanisms into its native networking model. The proposed system enables bounded-latency and low-jitter communication between containerized applications, maintaining compatibility with Kubernetes' declarative orchestration architecture. The solution is validated through empirical measurements conducted on a real hardware testbed consisting of TSN-capable nodes and network infrastructure.

1.2 Scope of the Thesis

This thesis investigates the integration of TSN mechanisms into Kubernetes-based container orchestration environments to enable deterministic communication. The system architecture focuses on enhancing Kubernetes pod networking with deterministic queuing disciplines and synchronized transmission control, validated on a real hardware testbed. A two-node Kubernetes cluster, equipped with Intel I225 TSN-capable network interface cards and Kronoton TSN switches, serves as the experimental environment. Empirical evaluation concentrates on measuring end-to-end latency, jitter. The work emphasizes a proof-of-concept implementation aligned with IEEE 802.1AS, IEEE 802.1Qbv, IEEE 802.1Qbr and IEEE 802.1Qcc standards, without extending to large-scale deployments or dynamic flow orchestration.

1.3 Research Objectives

The objectives of this thesis are as follows:

- To design a Kubernetes-native architecture for enabling deterministic, time-sensitive communication between containerized applications using TSN standards.
- To develop and integrate a custom CNI plugin that configures time-aware scheduling mechanisms during pod network attachment in Kubernetes.
- To implement a CNC that dynamically configures TSN switches using open standards such as NETCONF and YANG models.
- To establish end-to-end time synchronization across K8 nodes and TSN-capable switches based on IEEE 802.1AS.
- To empirically validate the proposed system architecture through real hardware measurements focused on end-to-end latency, jitter, and synchronization accuracy under different real-time traffic scenarios.

Additionally, this thesis aims to answer the following research questions:

  - How can Kubernetes be extended to support deterministic communication without violating its native orchestration architecture?
  - What architectural components (at pod, host, and switch level) are essential to synchronize scheduling behavior end-to-end?

1.4 Thesis Structure

The remainder of this thesis is organized as follows:

- **Chapter 2: Theoretical Background**
  This chapter provides an overview of TSN standards, Kubernetes networking models, and the existing challenges in achieving deterministic communication within containerized environments.
- **Chapter 3: Design and Concept Methodology**
  This chapter describes the architectural design of the proposed system, including the integration of TSN capabilities into Kubernetes, the development of the custom CNI plugin, the implementation of the CNC, and the conceptual framework for end-to-end deterministic communication.
- **Chapter 4: Implementation Overview**
  This chapter details the practical implementation of the system, including the Kubernetes cluster setup, configuration of deterministic networking mechanisms, TSN switch management, synchronization infrastructure, and the deployment of the measurement environment.
- **Chapter 5: Validation Overview**
  This chapter defines the validation objectives, experimental scenarios, measurement tools, and evaluation criteria used to assess the deterministic performance of the Kubernetes-TSN integrated system.
- **Chapter 6: Results and Analysis**
  This chapter presents the empirical results obtained from latency, jitter, and synchronization measurements, and analyzes the system's performance across different communication scenarios and load conditions.
- **Chapter 7: Discussion**
  This chapter critically reflects on the design choices, implementation outcomes, observed limitations, and broader applicability of the proposed architecture.
- **Chapter 8: Conclusion**
  This chapter summarizes the key findings of the thesis, discusses its contributions to the field, and outlines potential directions for future research, including dynamic flow reservation and centralized user configuration (CUC) integration.

Chapter 2

Technical Background

2.1 Operational Technology

Operational Technology (OT) refers to the hardware and software systems employed to monitor, control, and manage physical processes, devices, and infrastructure. These systems are predominantly utilized in industrial settings such as manufacturing plants, energy distribution networks, and transportation systems, where real-time responsiveness and reliability are paramount. OT systems typically comprise programmable logic controllers (PLCs), distributed control systems (DCS), supervisory control and data acquisition (SCADA) systems, and other embedded devices [SFS11].

A defining feature of OT is its deterministic behavior, which ensures the execution of operations within predefined and tightly controlled timeframes. This characteristic is crucial in industrial processes, where delays or unpredictability can lead to equipment malfunction, quality degradation, or safety risks. In contrast to information technology (IT), which is concerned with processing, data management, and user interaction, operational technology (OT) is committed to maintaining consistent and dependable control over physical systems [Pla25]. OT is vital in industrial settings because automation and real-time monitoring are necessary to ensure productivity and security [Cis25].

Traditionally, OT systems have operated in isolated closed-loop environments with proprietary protocols and minimal external connectivity. This design has ensured operational reliability but has also limited flexibility. However, the ongoing convergence of IT and OT, driven by digital transformation, has resulted in more networked and software-defined OT environments [Mic25]. This evolution facilitates greater automation and data exchange but introduces challenges related to interoperability, timing precision, and cybersecurity [SK24]. Consequently, modern OT infrastructures increasingly necessitate deterministic communication frameworks, such as Time-Sensitive Networking (TSN), to maintain reliability while enabling scalable and flexible integration.

2.2 Industry 4.0

Industry 4.0 signifies the fourth wave of industrial advancement, marked by the fusion of digital technologies with manufacturing and industrial frameworks [IBM25]. It extends the concept of automation by incorporating intelligence, connectivity, and

autonomy into production settings. This revolution introduces cyber-physical systems, the Internet of Things (IoT), cloud platforms, and sophisticated analytics to facilitate flexible, real-time, and data-driven operations [OV25][ANR25].

Core principles of Industry 4.0 encompass interoperability, decentralization, real-time capability, virtualization, and service orientation [Dav24]. A significant shift brought about by this transformation is the transition from centralized, inflexible automation structures to decentralized, adaptive, and self-organizing systems. Machines and devices are anticipated to communicate using standardized protocols, process data either locally or remotely, and autonomously adapt to changes in operations. The implementation of digital twins, predictive modeling, and edge computing further boosts the responsiveness and predictive capabilities of industrial systems [**6**].

Although Industry 4.0 enhances production efficiency and system intelligence, it imposes new requirements on communication networks [ANR25]. Traditional industrial networking technologies often fall short in terms of flexibility, scalability, and determinism needed for contemporary applications [Eng25]. To address these needs, Ethernet-based technologies like TSN have been developed to support time-synchronized, low-latency, and reliable communication. The deterministic features of TSN, when combined with container orchestration tools such as Kubernetes, provide a pathway for enabling real-time, scalable industrial applications that align with the principles of Industry 4.0 [AG23].

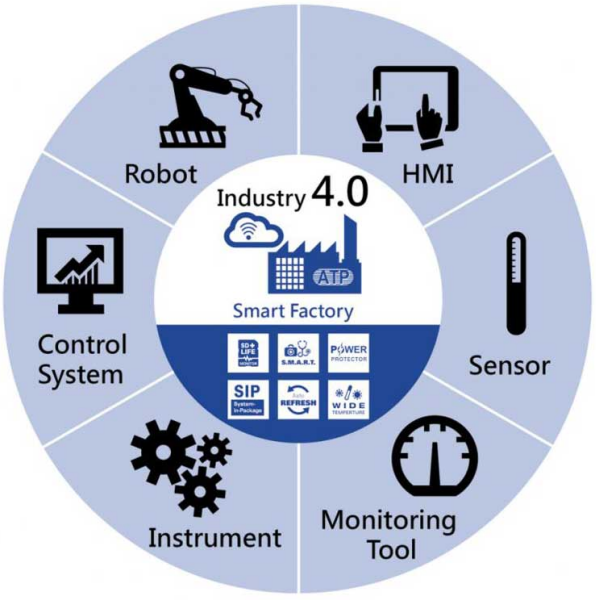


FIGURE 2.1: Core components of Industry 4.0 in a smart factory. Image taken from [Del25].

2.3 Kubernetes

Kubernetes, an open-source platform for container orchestration, was initially created by Google and is now maintained by the Cloud Native Computing Foundation (CNCF) [Fou23]. It facilitates the automation of deploying, scaling, and managing containerized applications within a distributed computing framework. By abstracting the underlying infrastructure, Kubernetes offers a declarative interface to specify application behavior, thus supporting resilient and scalable systems based on microservices [Bur+16].

At its core, Kubernetes employs a master-worker architecture. The control plane, which oversees the desired state of the cluster, comprises essential components like the API server, scheduler, controller manager, and etcd, a distributed key-value store [HBB17]. Worker nodes execute container runtimes such as Docker, CRI-O and host the actual application workloads organized into pods, the smallest deployable units in Kubernetes [HBB17].

Additionally, Kubernetes provides services for load balancing, volume management, and service discovery. Users can specify application needs and system policies through declarative configuration files written in YAML. Kubernetes continuously aligns the actual system state with the desired state, enabling features like autoscaling, rolling updates, and self-healing [Bur+16]. These capabilities have established Kubernetes as a standard in modern DevOps and edge computing.

However, Kubernetes was initially designed for cloud-native applications that operate on a best-effort basis [DS+20] and does not ensure deterministic network performance. Its default networking model lacks time-awareness, packet scheduling, and latency constraints, which are crucial for real-time industrial systems. To address these shortcomings, extensions such as custom Container Network Interface (CNI) plugins, Multus for multiple network interfaces, and traffic control tools are being investigated. Incorporating TSN features into Kubernetes-based systems is an ongoing research effort aimed at adapting Kubernetes for Industry 4.0-grade deterministic workloads.

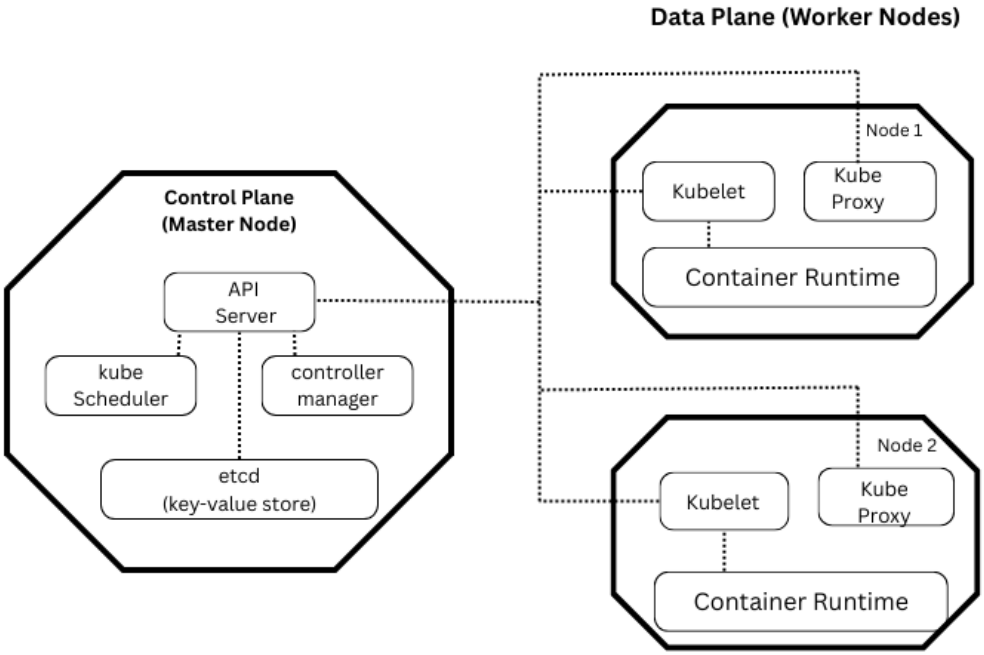


FIGURE 2.2: Kubernetes Architecture Overview

2.3.1 Kubernetes Scheduling Architecture

Scheduling in Kubernetes refers to the process of assigning pods to suitable nodes within the cluster. The kube-scheduler, a component of the control plane that monitors unscheduled pods and selects appropriate host nodes based on a range of criteria, performs this task [HBB17].

The scheduling process proceeds in two main stages: filtering and scoring [Aut23b]. In the filtering phase, nodes that do not meet the pod's requirements—such as insufficient resources, incompatible labels, or conflicting taints—are excluded. The remaining nodes are evaluated and ranked in the scoring phase based on metrics like resource balance and workload distribution. The scheduler then binds the pod to the node with the highest score [DS+20]. While this approach efficiently handles general-purpose workloads, the default scheduler lacks awareness of temporal constraints. It does not consider factors such as execution timing, network transmission schedules, or alignment with time-sensitive protocols. This limitation poses a challenge in environments that rely on precise coordination between application behavior and network-level timing—such as those incorporating TSN. TSN requires synchronization between computing tasks and deterministic network transmission. For example, the IEEE 802.1Qbv standard mandates transmission at predefined time intervals [IEE16b]. If pods are placed without regard to network timing configurations, the system may fail to meet the required latency or jitter bounds.

Although Kubernetes allows extensibility through features like affinity rules, custom schedulers, and taints and tolerations, these are primarily oriented toward resource and topology considerations. They do not provide intrinsic support for time-aware scheduling. Addressing this gap requires external coordination or enhancements to the scheduling logic to ensure that workload placement is compatible with deterministic network requirements.

2.3.2 Kubernetes Networking and Container Network Interface

Networking in Kubernetes is essential for enabling communication among distributed application components [Kub25]. Each pod is assigned a unique IP address, allowing for direct connectivity without needing network address translation (NAT). This flat network model makes finding services and communicating between pods, both inside and between cluster nodes, easier..

Networking abstractions such as services, ingress, and egress rules facilitate communication between pods, services, and external endpoints. Kubernetes Services offers load balancing and consistent IP addresses across dynamically shifting groupings of pods. Through HTTP(S) routing, ingress resources control external access, and egress configurations specify how pods can access resources outside the cluster.

Kubernetes does not implement its networking stack; instead, it uses the Container Network Interface (CNI) specification to integrate with various third-party networking solutions. CNI plugins create network interfaces, assign IP addresses, and configure container routing. This pluggable architecture allows operators to tailor Kubernetes's networking behavior to match specific requirements [Aut23a]. Various CNI plugins support different operational goals. Flannel, for instance, creates an overlay network using VXLAN encapsulation, offering a simple solution for basic connectivity [KHM20]. Calico employs Layer 3 routing and supports advanced policy enforcement, making it suitable for secure, production-grade deployments [Cal23].

In some applications, such as industrial automation or real-time communication, unique CNI plugins are required. These plugins can include TSN-compliant functionality, including traffic priority, time-aware shaping, and VLAN tagging. Deterministic and domain-specific networking behaviors in Kubernetes-based systems are thus supported, and connectivity is made possible, thanks in large part to CNI.

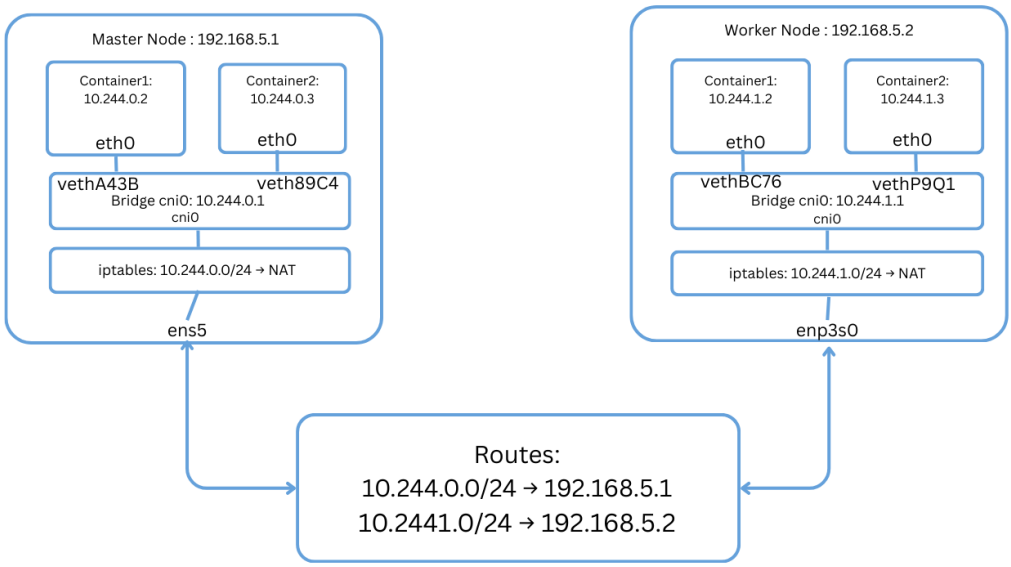


FIGURE 2.3: Kubernetes Networking Architecture

2.4 Time-Sensitive Networking

Time-Sensitive Networking (TSN) is a set of standards developed by the IEEE 802.1 working group to extend Ethernet with deterministic communication capabilities. These standards enable reliable, low-latency data delivery with bounded jitter and precise synchronization, making Ethernet suitable for applications with strict timing requirements [IEE24].

TSN overcomes the drawbacks of conventional Ethernet, which relies on best-effort and cannot guarantee delivery timeliness. TSN enables the coexistence of best-effort and real-time traffic on shared network infrastructure by implementing time synchronization, bandwidth reservation, and traffic scheduling techniques.

Rather than forming a single protocol, TSN consists of interoperable standards that can be selectively implemented. Its adoption supports the convergence of IT and OT networks by replacing legacy fieldbus systems with scalable, standards-based Ethernet, reducing cost and complexity while increasing flexibility [TGE19].

TSN is essential to Industry 4.0's communication infrastructure, as deterministic networking is required for robotics, autonomous systems, smart manufacturing, and other time-sensitive processes. New architectures that combine dynamic application deployment and real-time control are made possible by their integration with software-defined platforms. [PTF20].

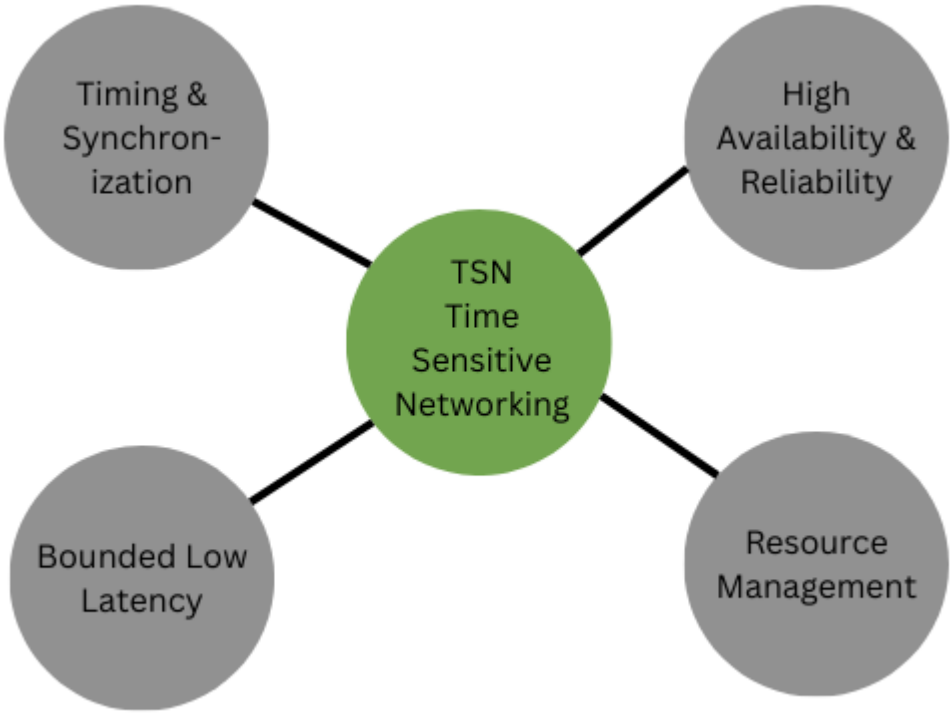


FIGURE 2.4: Illustration of the core properties of TSN

2.4.1 Time-Sensitive Networking (TSN) Standards

Deterministic communication over Ethernet is ensured via a set of characteristics defined by the TSN standards package. IEEE 802.1AS, which uses the Generalized Precision Time Protocol (gPTP) to provide time synchronization, is one of the fundamental standards. For coordinated scheduling and event alignment, this protocol makes sure that every device in the network runs on a precise, shared time base [IEE20].

IEEE 802.1Qbv introduces time-aware traffic shaping by segmenting time into recurring cycles and allocating specific transmission windows to prioritized traffic [IEE16a]. This ensures that time-sensitive packets are transmitted without delay or interference from best-effort traffic. IEEE 802.1Qbu and IEEE 802.3br introduce frame preemption, allowing urgent traffic to interrupt lower-priority frames, reducing latency in congested networks [IEE16c] [IEE16a].

IEEE 802.1Qcc provides mechanisms for stream reservation and centralized configuration of TSN flows. It supports both static and dynamic control, facilitating scalable and automated deployment in complex environments [IEE18a]. Additional standards, such as IEEE 802.1CB for redundant transmission and IEEE 802.1Qci for per-stream policing, further enhance reliability and network protection [IEE17].

Together, these standards enable Ethernet to deliver the deterministic performance required by real-time systems. Their modularity allows them to be integrated as needed, depending on the requirements of specific applications and industries.

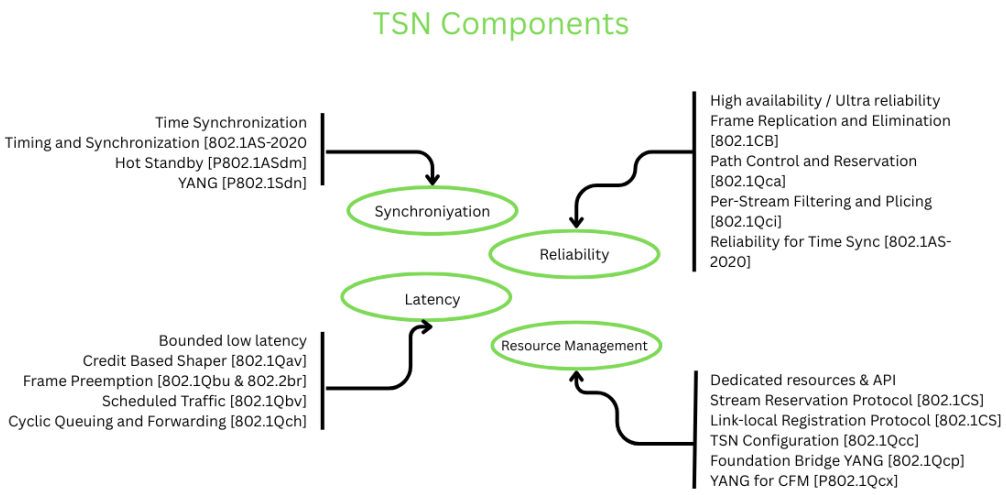


FIGURE 2.5: TSN Components

2.4.2 TSN Control Architecture: CNC and CUC

The Centralized User Configuration (CUC) and the Centralized Network Configuration (CNC) are the two main components of the TSN control architecture, which is designed to coordinate communication requirements between end devices and the network infrastructure. [IEE18a].

The CUC collects communication requirements from end systems, such as bandwidth demands, timing constraints, and endpoint identifiers. It translates these high-level application needs into network-specific configurations that the CNC can understand and implement.

The CNC is in charge of figuring out and implementing the necessary network configuration. It establishes egress queues, assigns time slots, establishes the best route for every stream, and implements rules in compliance with TSN guidelines. By monitoring the network topology and resources globally, the CNC guarantees that all accepted streams are schedulable and non-conflicting [IEE18a].

Configuration exchange between CUC, CNC, and TSN-capable switches is performed using standardized protocols such as NETCONF or RESTCONF [BCL19]. These leverage YANG-defined data models to convey configuration parameters in an organized, machine-readable manner via secure connections. This model-driven approach is ideal for dynamic and programmable networks since it facilitates automation and interoperability..

Integrating TSN with other control and orchestration platforms, such as Kubernetes, requires understanding the roles played by the CUC and CNC. These elements serve as a link between the deterministic behavior that the underlying network infrastructure enforces and the requirements of the application layer.

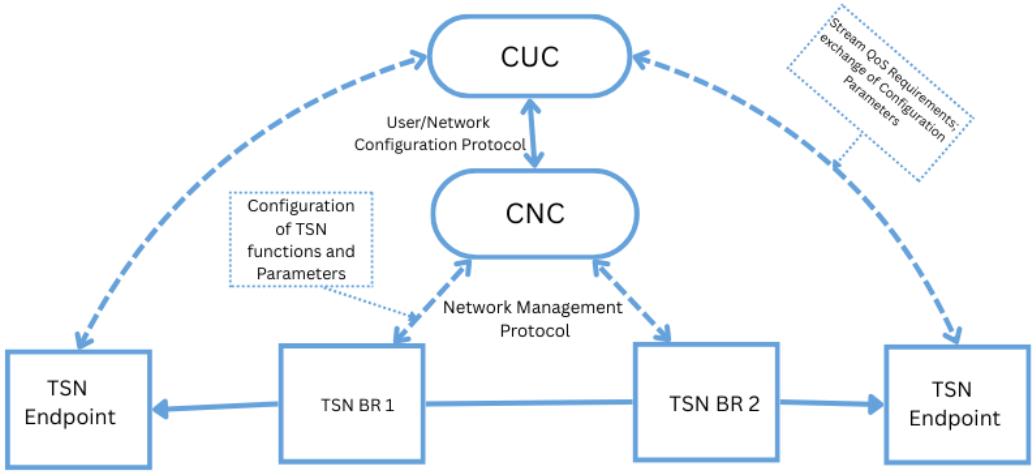


FIGURE 2.6: TSN 802.1Qcc Architecture

2.4.3 Time Synchronization in TSN Systems

Accurate time synchronization is a foundational requirement for Time-Sensitive Networking. IEEE 802.1AS provides the mechanism for distributing precise time across the network using the Generalized Precision Time Protocol (gPTP), which is derived from IEEE 1588 PTP [Unk20]. In a TSN system, one device is elected as the Grandmaster Clock, and other devices synchronize to its time using hardware timestamping. This ensures that all devices in the network maintain a shared sense of time with sub-microsecond precision. Synchronization enables features like time-aware shaping (802.1Qbv) and coordinated transmission schedules [IEE19]. In Linux-based systems, synchronization is implemented using tools such as ptp4l, which manages synchronization with the Grandmaster Clock, and phc2sys, which aligns the system clock with the hardware clock [Cor14]. These tools rely on physical layer support from Network Interface Cards (NICs) that offer hardware timestamping capabilities. Accurate synchronization ensures that traffic scheduled on one device arrives at its destination within the expected time window, supporting deterministic communication. It also prevents jitter and ensures consistency across the distributed control systems in industrial applications.

2.5 Existing CNI Plugins and Their Limitations

While Kubernetes supports a modular Container Network Interface (CNI) architecture, existing CNI plugins were primarily designed for general-purpose networking and lack features essential for time-sensitive communication. This section reviews several widely used CNIs and highlights their limitations in deterministic networking. Flannel is a simple overlay network that routes traffic between nodes using VXLAN encapsulation [Cor23a]. It offers minimal configuration overhead and ease of deployment, making it a default choice for many Kubernetes clusters. However, Flannel does not support network policies, traffic prioritization, or hardware offloading. It also lacks support for VLAN tagging and time-aware scheduling, making it unsuitable for TSN [Cor23a]. Calico provides Layer 3 networking with optional support for BGP-based routing and advanced network policies [Cal23]. It eliminates overlay overhead and offers higher throughput. Despite these benefits, Calico does not support deterministic queuing, VLAN tagging for TSN flows, or integration with hardware TSN features. Macvlan is a CNI that allows direct Layer 2 access by creating virtual interfaces bound to a physical NIC [Doc23]. It reduces latency by bypassing Linux bridges and can be used to isolate traffic at the MAC level. However, macvlan lacks mechanisms for shaping traffic based on time and does not provide integration with gate control scheduling. Multus is a meta-plugin that connects Kubernetes pods to multiple network interfaces using different CNIs [Cor23b]. This enables the separation of control-plane and data-plane traffic or time-sensitive and best-effort traffic. Multus is instrumental in combining a TSN-aware secondary CNI with a standard primary CNI but does not provide TSN features. While each plugin offers unique advantages, none natively support TSN standards such as IEEE 802.1Qbv or 802.1AS. As a result, this research introduces a custom Bash-based TSN CNI explicitly designed to integrate VLAN tagging, time-aware scheduling via Taprio, and deterministic traffic handling into Kubernetes.

2.6 Enabling Tools and Technologies

Several foundational tools and protocols are employed to support deterministic communication and orchestration in a containerized TSN environment. These technologies enable key functions such as time synchronization, traffic scheduling, network configuration, and multi-interface management . Python and Bash scripting were chosen as the primary languages for automation and integration. Python supports NETCONF-based communication through libraries like ncclient, while Bash enables low-level control over network interfaces and traffic configuration. Go implements lightweight, concurrent measurement utilities for capturing and analyzing latency and jitter in real time. YAML is used to define Kubernetes manifests and NetworkAttachmentDefinitions, supporting the declarative configuration of pods and network interfaces. CRIO is employed as the container runtime environment, enabling the deployment of isolated, reproducible workloads within the Kubernetes cluster. ptp4l and phc2sys implement IEEE 802.1AS-compliant time synchronization across hosts and TSN switches, supporting hardware timestamping and submicrosecond precision. taprio, tc, and skbedit are Linux traffic control utilities that enforce IEEE 802.1Qbv time-aware scheduling and VLAN-tagged stream prioritization at the host level. netopeer2 and NETCONF enable centralized, programmable configuration of TSN switches using YANG data models, aligning with the IEEE 802.1Qcc control architecture. Multus and macvlan are CNI extensions that allow Kubernetes pods to connect to multiple network interfaces, separating time-sensitive TSN traffic from standard communication. Flannel is a reference overlay CNI for basic pod communication used to benchmark non-deterministic baseline performance.

Chapter 3

Design and Concept Methodology

3.1 Research-Driven Methodological Approach

This thesis employs an experimental, implementation-oriented research methodology to investigate the integration of TSN with Kubernetes. The goal is to demonstrate the feasibility and performance of deterministic, real-time pod communication within a Kubernetes-managed environment. The research approach is structured around three key stages: theoretical analysis, system design, and empirical validation. In the first stage, a detailed literature review and standards investigation were conducted to define the foundational elements of TSN and Kubernetes. This included standards such as IEEE 802.1AS-2020 for time synchronization, IEEE 802.1Qbv for time-aware traffic scheduling, and IEEE 802.1Qcc for centralized configuration. K8's networking capabilities—particularly the CNI model, scheduling limitations, and support for multi-network configurations—were analyzed in parallel. The second stage involved translating theoretical insights into a modular, standards-compliant system architecture. This includes conceptualizing a custom CNI plugin to enable deterministic traffic scheduling at the pod level and a Python-based CNC to configure TSN switches using NETCONF/YANG. The K8's cluster was planned using Kubeadm and CRI-O as the container runtime, with Multus proposed to support multi-interface pods. In the final stage, empirical evaluation was conducted using a physical testbed to validate key performance metrics—latency and jitter various traffic scenarios. These experiments were designed under controlled conditions and supported by purpose-built Go-based measurement tools, Wireshark, and system monitoring utilities. This research methodology ensures replicability and practical applicability of the proposed integration model, particularly for industrial environments requiring deterministic performance within a cloud-native orchestration framework.

3.2 System Architecture and Integration Layers

The proposed architecture is structured around three primary layers: orchestration, deterministic networking, and synchronization. Each layer contributes distinct functionality and integrates with Kubernetes and TSN components to support real-time communication. The orchestration layer is centered around K8's control plane and container lifecycle management. It schedules and deploys pods based on user-defined policies. The deterministic networking layer introduces mechanisms for isolating time-sensitive traffic from general-purpose traffic. This includes physically separating network paths and using dual interfaces per pod enabled by Multus.

This layer supports scheduled traffic transmission in alignment with TSN requirements. The synchronization layer enforces a standard time reference across all nodes and TSN switches using LinuxPTP tools, as defined by IEEE 802.1AS. This layer enables consistent transmission slot alignment between host and switch components. A conceptual CNC component supports centralized coordination. This component distributes network configuration to TSN switches using standardized YANG models over NETCONF. This layer ensures that host—and switch-level behaviors are aligned under a unified control strategy. 3.1 illustrates this layered architecture and its components.

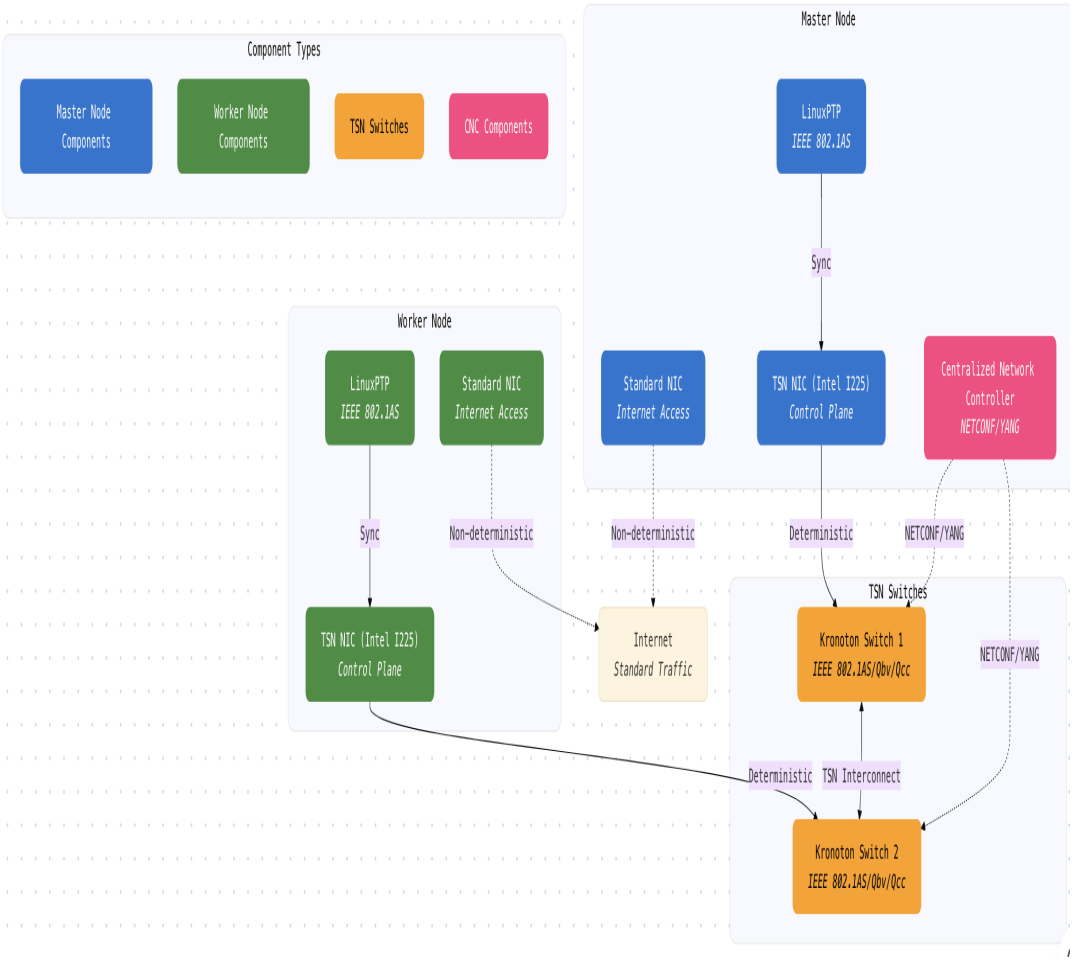


FIGURE 3.1: High-level architecture showing layered orchestration, dual-NIC node configuration, synchronization via LinuxPTP, and conceptual CNC coordination over NETCONF/YANG.

3.3 Key Design Considerations

The scalability of K8s-based systems is greatly impacted by real-time communication requirements, which call for architectural modifications to satisfy requirements for low latency and high availability [Clo25][Gro25b]. Each design choice supported deterministic networking while maintaining compatibility with open standards. Each node has one interface for general internet purposes and another dedicated to TSN. To enforce deterministic scheduling at the pod level, the system relies exclusively on native Linux kernel features, avoiding proprietary software or kernel-bypass frameworks. The taprio queuing discipline enables IEEE 802.1Qbv-based transmission scheduling [GP23] [MK23]. This decision ensures long-term maintainability and community support. Standardization was prioritized by using NETCONF and YANG to configure TSN switches. This ensures interoperability with industrial devices and aligns the system with IEEE 802.1Qcc recommendations. Time synchronization was also a core design pillar. Using ptp4l and phc2sys with hardware timestamping NICs provides sub-microsecond accuracy, enabling precise host and switch scheduling alignment. Lastly, modularity guided the architecture. Each functional unit—CNC, custom CNI, sync tools—can be independently deployed or upgraded.

3.4 Custom CNI Strategy

A tailored CNI strategy was developed to enable deterministic TSN communication within K8s-managed environments [AG23]. While K8s delegates pod networking to external CNI plugins, standard solutions such as Flannel and Calico are optimized for best-effort traffic and lack support for scheduling or prioritization. Therefore, this system adopts a layered networking approach that combines open-source plugins with a custom-developed extension to meet TSN requirements while preserving compatibility with K8's default behavior. The architecture employs Multus CNI as the first layer, enabling the attachment of multiple interfaces per pod [Gro25a]. Each pod receives two interfaces: one for deterministic pod-to-pod communication and another for best-effort traffic. This dual-interface design ensures the separation of time-critical and non-real-time traffic, facilitating independent treatment of each flow according to its performance requirements. Standard CNI plugins remain unmodified for best-effort traffic. Macvlan supports Layer 2 routing and policy enforcement. These plugins handle general traffic and coexist with the real-time extensions without disrupting core Kubernetes networking. A custom Bash-based CNI plugin provisioned the TSN interface. Upon pod instantiation, the plugin performs VLAN tagging, sets PCP fields for QoS classification, and prepares the TSN-capable interface for time-aligned scheduling. The system-level synchronization and scheduling mechanisms described in the next section manage specific queuing and scheduling configurations. This modular strategy introduces deterministic communication paths without altering Kubernetes internals. By integrating TSN configuration at the interface level, the system supports real-time networking while maintaining compatibility with standard orchestration workflows, simplifying deployment, and supporting future extensibility.

3.2 illustrating each pod receives a best-effort interface and a TSN-configured interface for deterministic communication.

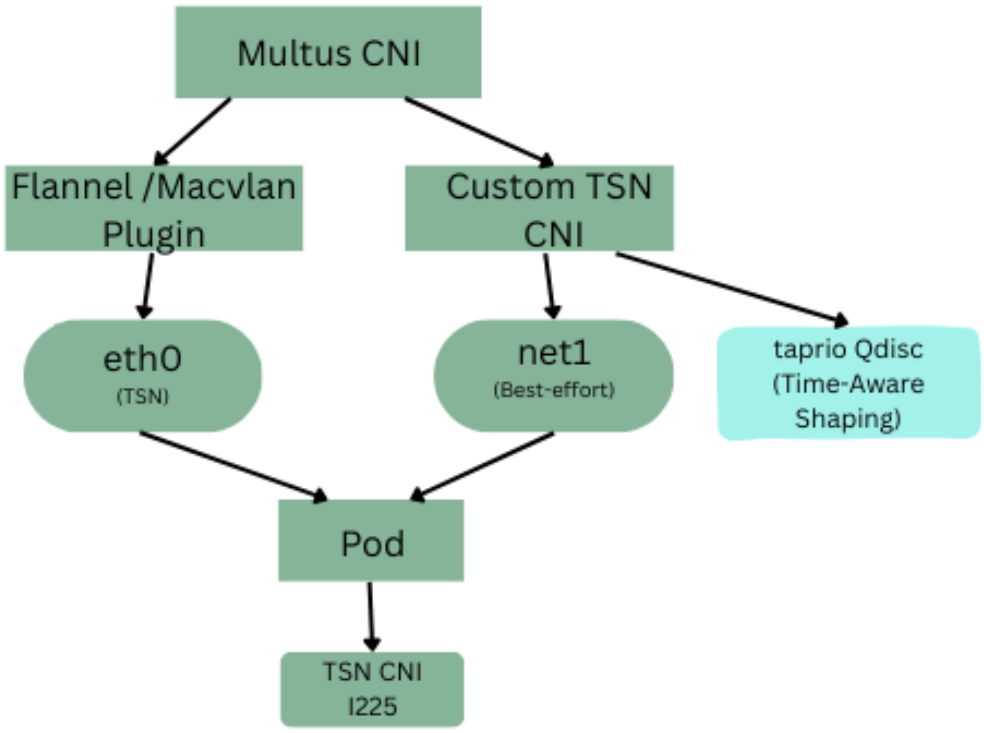


FIGURE 3.2: Diagram of Multus, custom CNI and second CNI integration

3.5 Time Synchronization Concept

Accurate time synchronization is a cornerstone for enabling TSN within distributed containerized environments [SW20]. Deterministic communication protocols such as IEEE 802.1Qbv rely on precise transmission scheduling, which demands a shared and highly accurate notion of time across all participating nodes and switches. This architecture achieves time synchronization by integrating the gPTP specified by IEEE 802.1AS-2020. Each K8's node participating in the TSN domain has a TSN-capable NIC supporting hardware timestamping. These NICs are essential for maintaining the sub-microsecond clock precision required for deterministic scheduling. The network infrastructure, composed of gPTP-aware TSN switches, extends this synchronized time base throughout the system. Typically, the TSN switch acts as the Grandmaster Clock, broadcasting synchronization messages to all connected nodes, although mastership can dynamically shift depending on network conditions and clock quality. On the host side, each node runs the LinuxPTP daemon suite, specifically ptp4l for hardware clock synchronization and phc2sys for synchronizing the system clock with the hardware clock. These daemons ensure that user-space and kernel-space processes operate under a unified, precise time reference. The hardware clock synchronization enables correct transmission scheduling through taprio-managed egress queues [Doc25], while system clock alignment supports consistent timestamping in user-space applications. The gPTP synchronization process involves periodic exchange of synchronization messages across the network links, including Pdelay Request, Pdelay Response, and Follow_Up messages [Abo18]. These messages allow devices to calculate link-specific delays and clock offsets, adjusting

their local clocks to maintain tight temporal alignment. A visual representation of this synchronization process is shown in Figure 3.3, illustrating the message flow between the Master node, intermediate TSN switch, and Slave node. This timing exchange ensures that all devices within the TSN domain maintain sub-microsecond clock precision, critical for the scheduled gate control operations defined in IEEE 802.1Qbv.

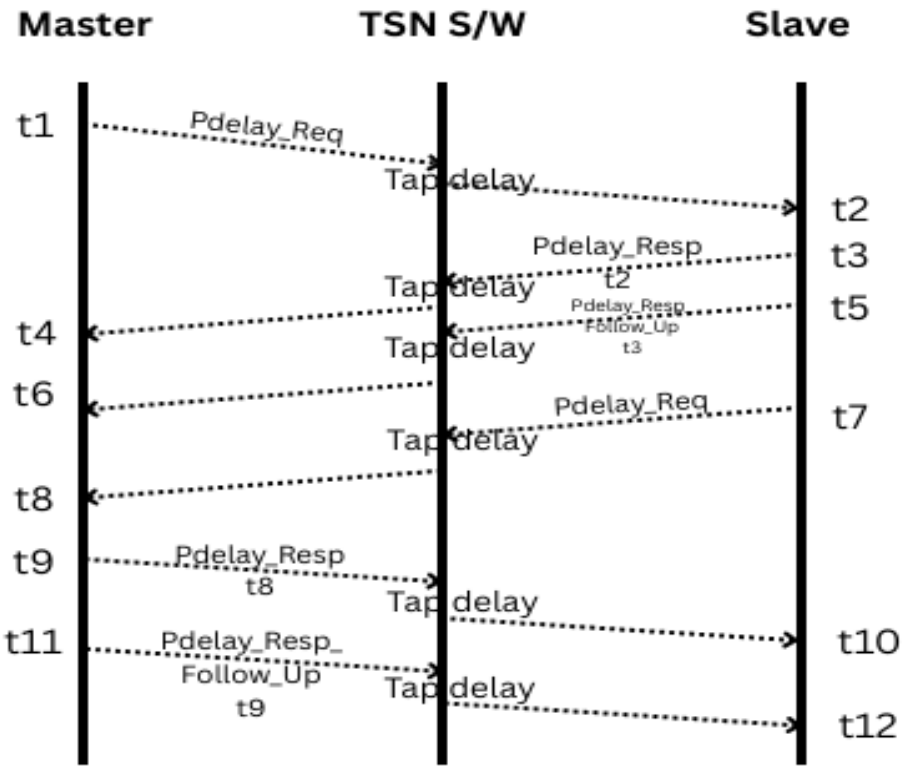


FIGURE 3.3: Precision Time Protocol (gPTP) message exchange diagram

3.6 Traffic Scheduling Strategy

The system employs a traffic scheduling strategy based on the IEEE 802.1Qbv TAS mechanism to achieve deterministic communication across the containerized and network layers [Ass25]. In this model, time is divided into repetitive cycles and specific transmission windows are allocated to distinct traffic classes. This allows critical time-sensitive frames to be transmitted at predictable intervals, fully isolated from lower-priority traffic. Outgoing traffic from Kubernetes pods is tagged at the host level with VLAN headers containing PCP fields. These PCP values are assigned according to the criticality of the traffic and are configured by the custom Bash-based CNI plugin during the pod network attachment phase [IEE18b]. Upon reaching the TSN switch, incoming frames are classified based on their PCP values and mapped into separate egress queues according to predefined priority levels. Each egress queue is associated with a gate controlled by a schedule defined by CNC. The CNC programs the GCLs into the switches through NETCONF interfaces, ensuring that queues are opened or closed at precise time intervals aligned to the cluster's synchronized clock domain. Within each scheduling cycle, only the designated queues are allowed to transmit, while all others are held closed, thus preventing contention and guaranteeing bounded latency for time-critical flows. This gate control mechanism enables scheduled traffic to transmit without competition from background best-effort traffic, even under heavy network load. Additionally, non-scheduled flows are assigned lower-priority queues with separate, non-interfering windows, protecting the timing of deterministic streams. The system further supports per-stream filtering and policing according to IEEE 802.1Qci to enforce traffic compliance and IEEE 802.1Qcc stream reservation protocols to manage dynamic network resources. **??**Time-Aware Shaping inside a TSN switch. Queued traffic is prioritized and transmitted according to a gate schedule.

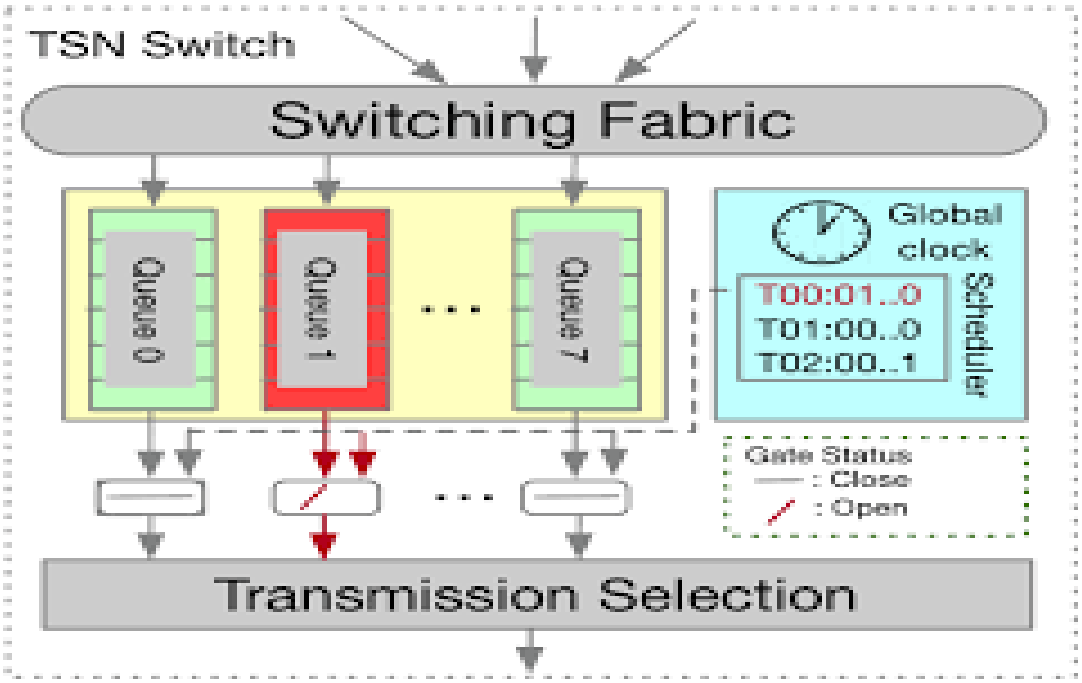


FIGURE 3.4: Working principle of IEEE 802.1Qbv Time-Aware Shaper.
Image was taken from [PMP23]

3.7 Network Configuration and Resource Orchestration via CNC and CUC

The integration of TSN into a K8s-managed environment necessitates a systematic and centralized approach to network configuration and resource orchestration. This functionality is realized through two critical architectural components: the CUC and the CNC entities. Together, they ensure that application-level communication requirements are dynamically translated into synchronized network-level configurations, enabling deterministic behavior across the entire TSN-enabled infrastructure [Gar+23]. The CUC operates as the interface between containerized applications and the underlying TSN control plane. When a pod requiring time-sensitive communication is instantiated, its specific traffic requirements—such as stream periodicity, required bandwidth, latency bounds, and priority—are either explicitly defined by the application or inferred from predefined workload profiles. The CUC aggregates this information into structured descriptors, formally known as Talker-Listener advertisements or stream reservation requests, which describe the expected communication behavior for each real-time flow. These descriptors are passed to the CNC, which maintains a global view of the network topology, resource availability, and existing stream reservations. Upon receiving stream requirements from the CUC, the CNC computes feasible transmission schedules, allocates required resources, and generates GCLs for each TSN-capable switch in the network. The CNC then pushes these configurations into the switches via standardized management protocols such as NETCONF or RESTCONF, operating over machine-readable YANG data models. This ensures that all critical traffic flows are properly prioritized and scheduled according to their deterministic timing constraints. By decoupling application-level and network-level concerns, the CUC-CNC model enables dynamic, policy-driven orchestration [WK08]. Applications remain agnostic to network details, while the CNC ensures consistent enforcement of timing guarantees even as the containerized environment evolves—such as during pod migrations, scaling operations, or service updates. This model bridges Kubernetes' flexibility in orchestrating containerized workloads with the stringent timing and reliability demands of industrial Ethernet systems based on TSN.

3.8 Use Case Application and Scope

The system architecture developed in this thesis targets industrial scenarios where real-time control commands or high-frequency sensor data must be exchanged across containerized systems with strict timing guarantees. Example application domains include smart manufacturing cells where robotic actuators require periodic control signals, predictive maintenance systems where time-correlated vibration data streams are analyzed, and machine vision platforms that demand bounded-latency transmission of image frames for defect detection.

A typical application flow involves a talker pod transmitting periodic, high-priority TSN-tagged traffic toward one or more listener devices or pods. This communication is routed through a dedicated TSN-capable network interface, configured automatically by the custom CNI plugin to enforce deterministic scheduling and VLAN-based traffic classification. The TSN switches in the network forward this traffic according to hardware-enforced GCLs, ensuring predictable transmission windows. Synchronization of all network elements is maintained using gPTP mechanisms, guaranteeing a unified clock domain across pods, nodes, and switches.

By demonstrating deterministic, time-synchronized communication in a K8s-native environment, this system addresses the critical gap between flexible cloud-native orchestration models and the stringent timing constraints of industrial Ethernet standards. The subsequent implementation chapter provides a detailed description of the deployment steps, configurations, and validation processes used to realize this architecture.

Chapter 4

Implementation Overview

4.1 Hardware Setup

4.2 Experimental Platform

The experimental platform developed for this thesis consists of two physical machines configured as a minimal Kubernetes cluster, comprising one master node and one worker node. Both machines operate on Ubuntu 24.04 LTS with the `PREEMPT-RT` real-time Linux kernel, chosen to ensure deterministic task scheduling and low-latency behavior at the operating system level.

Each machine is equipped with two independent network interface cards (NICs) to achieve strict separation between best-effort and time-sensitive traffic. The first NIC on each machine connects to a non-manageable Ethernet switch, providing internet connectivity for remote access, system maintenance, and Kubernetes component installation. This interface is isolated from the time-critical communication domain.

The second NIC on each machine is reserved exclusively for Time-Sensitive Networking (TSN) communication. These interfaces utilize Intel I225 controllers, which support key TSN features, including:

- Hardware timestamping for IEEE 802.1AS (gPTP) synchronization,
- Time-aware shaping via IEEE 802.1Qbv,
- Centralized configuration using IEEE 802.1Qcc,
- Frame preemption as per IEEE 802.1Qbr.

Each TSN NIC connects to a Kronoton TSN-capable switch—one for the master node and another for the worker node. The two switches are interconnected, forming a TSN-enabled path that spans both nodes. This setup provides a dedicated, deterministic communication channel while isolating it from background traffic. The use of separate switches reinforces domain separation, ensuring the accuracy of latency and jitter measurements under TSN constraints.

Figure **??** provides a front view of the setup, including the TSN switches and the machines designated as master and worker nodes. Figure **??** shows the rear panel of the machines, where the physical cabling of TSN and internet-facing interfaces are clearly labeled and separated.

This dual-interface topology provides a clean foundation for deterministic communication experiments. It allows for controlled testing of Kubernetes-TSN integration

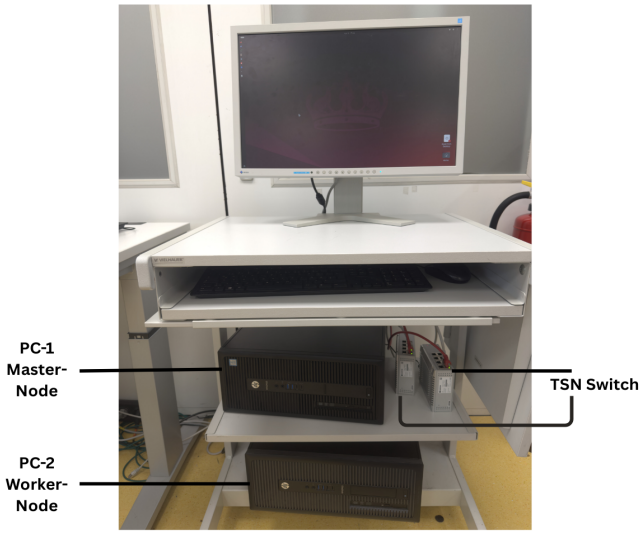


FIGURE 4.1: Front view of the testbed with two PCs, TSN switches, and non-manageable switch

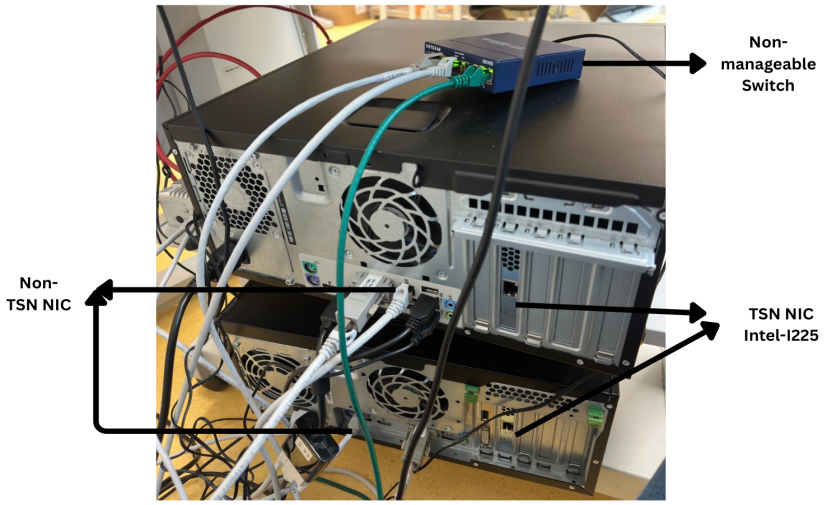


FIGURE 4.2: Rear view showing of the testing platform

while ensuring that timing-sensitive traffic remains unaffected by unrelated background communication. The separation of traffic domains is essential for validating TSN performance guarantees such as bounded latency, low jitter, and precise time synchronization.

4.3 Testbed Setup Overview

Building upon the physical setup outlined in Section **??**, the experimental testbed is structured to validate deterministic communication between containerized applications and bare-metal devices through a TSN-compliant infrastructure. The architecture is intentionally designed to represent real-world industrial topologies where Kubernetes-managed workloads interact with synchronized network devices and

endpoints. Each Kubernetes node is equipped with two physical interfaces: one standard NIC for cluster operations and internet access, and one TSN NIC interfaced with Kronoton TSN switches. These switches support IEEE 802.1AS for clock synchronization and IEEE 802.1Qbv for time-aware traffic shaping, ensuring scheduled and low-jitter communication. The testbed topology enables controlled experiments across isolated and mixed traffic domains. To analyze the behavior under diverse real-time networking conditions, four experimental scenarios were designed:

4.3.1 Scenario 1: Pod-to-Pod Communication with TSN

Two pods, hosted on separate Kubernetes nodes, communicate via the TSN NIC path. Traffic traverses Kronoton TSN switches, enabling analysis of containerized TSN communication under orchestrated conditions.

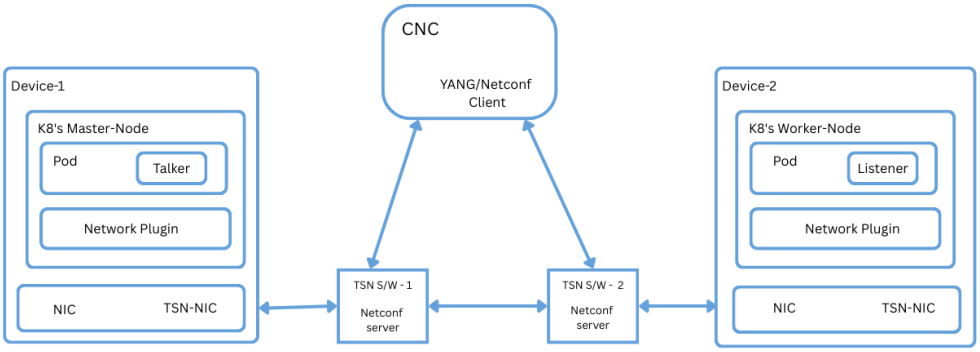


FIGURE 4.3: Pod-to-Pod communication over TSN-enabled Kubernetes cluster.

4.3.2 Scenario 2: Pod-to-Pod Communication without TSN

This scenario evaluates overlay networking via Kubernetes' default CNI plugins without TSN integration. It provides a baseline for comparison with deterministic traffic scenarios.

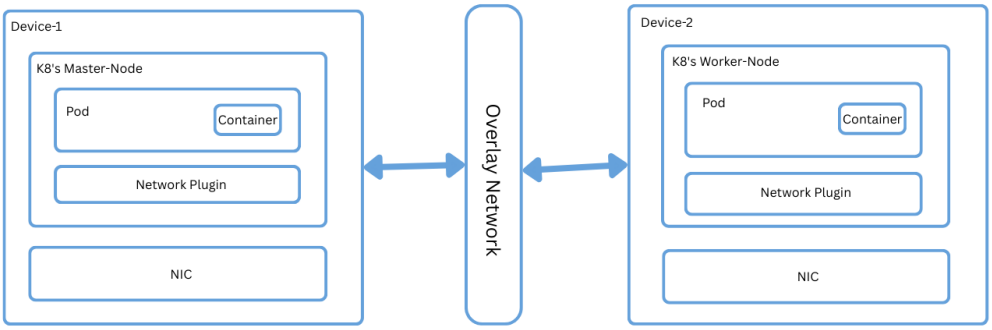


FIGURE 4.4: Overlay-based communication without deterministic network support.

4.3.3 Scenario 3: Device-to-Device Communication via TSN

Bare-metal devices equipped with TSN NICs communicate across Kronoton switches without Kubernetes involvement. This serves as a control test to measure deterministic networking performance in isolation.

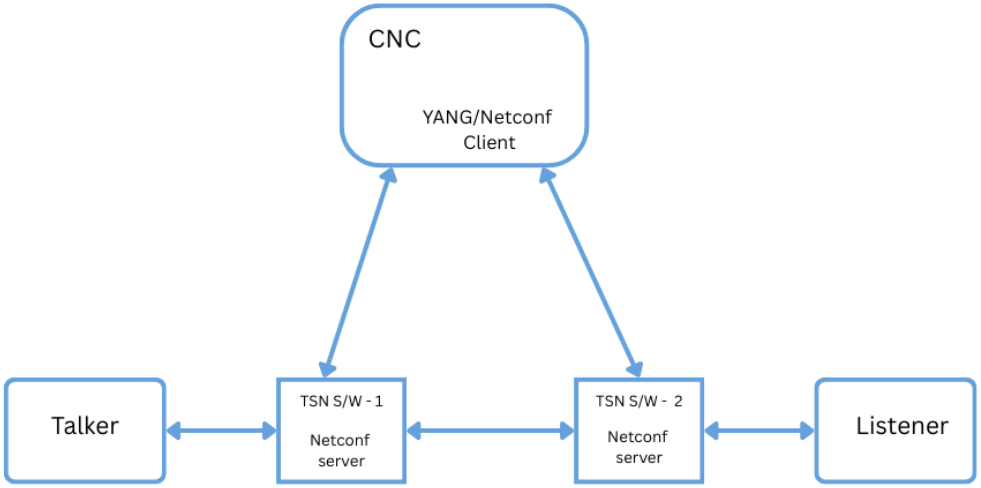


FIGURE 4.5: Bare-metal TSN communication across synchronized switches.

4.3.4 Scenario 4: Direct Device-to-Device Communication (Bare-Metal)

Two PCs equipped with TSN and standard NICs are directly connected, bypassing switches. This configuration measures point-to-point behavior, allowing assessment of scheduling effectiveness without intermediate network hardware.

FIGURE 4.6: Direct PC-to-PC connection using TSN and standard NICs.

These scenarios collectively test end-to-end determinism, validate orchestration compatibility, and highlight the contrast between conventional overlay networking and TSN-aware designs in Kubernetes-managed environments.

4.3.5 Testbed Summary and Configuration

Component	Specification / Details
Master Node	Intel i7, 16GB RAM, 2 NICs (1 for TSN, 1 for Internet)
Worker Node	Intel i5, 16GB RAM, 2 NICs (1 for TSN, 1 for Internet)
Operating System	Ubuntu 24.04 LTS
Kernel	PREEMPT-RT (Real-time Linux kernel)
Kubernetes Version	v1.27.1
Container Runtime	CRI-O
Custom CNI Plugin	Bash-based CNI plugin implementing IEEE 802.1Qbv using `taprio`
Multus Version	Installed via Helm; deployed as a DaemonSet across all nodes
Macvlan Plugin	Configured in bridge mode with static IPAM via `NetworkAttachmentDefinition`
TSN NIC Model	Intel I225 with hardware timestamping capability
TSN Switch Model	Kronoton TSN-Capable Switch
Switch TSN Features	IEEE 802.1AS (gPTP), IEEE 802.1Qbv (Time-Aware Shaping), IEEE 802.1Qcc, IEEE 802.1Qbr
Time Synchronization Tools	`ptp4l`, `phc2sys` (LinuxPTP suite)
CNC Implementation	Python-based controller using `ncclient` and NETCONF with YANG configuration models

4.4 Kubernetes Cluster Setup

The Kubernetes cluster for this system was deployed using `kubeadm`, a widely adopted tool designed to simplify the installation and management of Kubernetes components in production-grade environments. The cluster was built using Kubernetes version 1.27.1, chosen for its enhanced support for modern networking plugins, improved scheduling behavior, and proven stability in edge-computing scenarios.

The cluster consisted of two bare-metal machines running Ubuntu 24.04 LTS with a PREEMPT-RT patched kernel, allowing for low-latency task scheduling and hardware interaction. One node was configured as the master node, responsible for control plane operations, while the second was joined as a worker node, intended for scheduling pods and running application workloads.

Before initializing the Kubernetes control plane, the container runtime CRI-O was installed on both machines. CRI-O was selected over the default `containerd` because of its compliance with the Open Container Initiative (OCI) standards and its optimized performance in lightweight, real-time environments. It provides a minimal runtime layer that integrates seamlessly with Kubernetes while allowing greater control over container behavior—an important feature for TSN-based applications.

To initialize the cluster, the following command was executed on the master node:

```
kubeadm init -pod-network-cidr=10.244.0.0/16
```

This command bootstrapped the control plane and assigned the pod network CIDR range to allow communication between pods across nodes.

Once the control plane was active, a join token was generated using:

```
kubeadm token create -print-join-command
```

The worker node then joined the cluster using the provided command, which included the token and discovery certificate hash. After the nodes were successfully integrated, the `kubeconfig` file was set up to enable cluster management using `kubectl`.

The successful status of both nodes was verified using:

```
kubectl get nodes -o wide
```

Figure 4.7 shows the Kubernetes cluster after initialization, with both nodes registered and the control plane in a ready state.

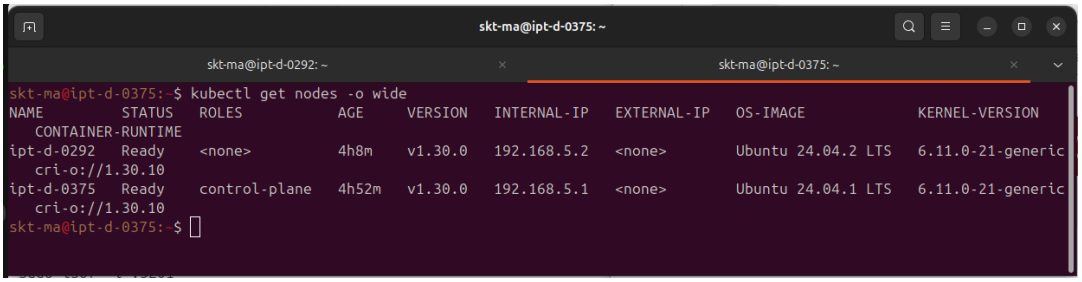


FIGURE 4.7: Node status following Kubernetes installation, displaying roles, internal IPs, and kernel runtime.

4.4.1 Custom CNI Plugin Implementation

To facilitate deterministic network behavior within Kubernetes, a custom Container Network Interface (CNI) plugin was developed using Bash scripting. This implementation diverges from conventional CNI plugins such as Flannel and Calico, which prioritize general-purpose compatibility, by specifically enabling the integration of IEEE 802.1Qbv Time-Sensitive Networking (TSN) capabilities into the Kubernetes pod network.

The custom plugin is invoked during the standard CNI lifecycle, executing logic upon pod creation and deletion. Its primary function is to configure a dedicated TSN-capable communication path for each pod. This is achieved through the allocation of a virtual Ethernet (veth) pair, wherein one end is moved into the pod's network namespace and the other is attached to the host's TSN-capable physical interface. The plugin assigns each pod a statically reserved IP address, selected from a predefined pool stored in a local tracking file, ensuring deterministic IP allocation.

To support traffic classification and time-aware scheduling, the plugin applies VLAN tagging and assigns Priority Code Point (PCP) values for Quality of Service (QoS). The queuing behavior is defined using the Linux `tc` utility by applying the `taprio` queuing discipline to the TSN interface. This configuration enables the enforcement of Gate Control Lists (GCLs), allowing time-aware transmission according to synchronized scheduling windows. The plugin also initializes multiple traffic classes and assigns corresponding queues using a fixed traffic-class mapping scheme. Gate open intervals are defined per queue using time-slot entries specified with `sched-entry` directives.

An example `taprio` configuration executed by the plugin is shown below:

```
sudo tc qdisc replace dev enp3s0 parent root handle 100 taprio \
  num_tc 4 \
  map 0 1 2 3 3 3 3 3 \
  queues 1@0 1@1 1@2 1@3 \
  base-time 0 \
  sched-entry S 0x1 250000 \
  sched-entry S 0x2 250000 \
  sched-entry S 0x4 250000 \
  sched-entry S 0x8 250000 \
  flags 0x2
```

LISTING 4.1: TAPRIO scheduling configuration

This configuration defines four traffic classes, each assigned a dedicated queue and gate interval of 250 µs. The use of hexadecimal masks determines which queues are open at specific transmission cycles, enabling time-aware deterministic delivery.

To register the plugin within the Kubernetes CNI stack, the plugin binary was placed in `/opt/cni/bin/` and a configuration file was created under `/etc/cni/net.d/`. A sample configuration file is presented below:

```
{
  "cniVersion": "0.3.1",
  "name": "mynet",
  "type": "bash-cni",
  "bridge": "cni0",
  "isGateway": true,
  "ipMasq": true,
  "subnet": "10.244.1.0/24",
  "network": "10.244.0.0/16",
  "ipam": {
    "type": "host-local",
    "subnet": "10.244.1.0/24",
    "gateway": "10.244.1.1"
  }
}
```

LISTING 4.2: Bash-based CNI plugin configuration

The plugin was named `bash-cni` and assigned the highest precedence through the file `10-custom-cni.conf`, ensuring Kubernetes would invoke it by default.

Lifecycle operations, including interface creation (ADD) and cleanup (DEL), are logged to `/var/log/bash-cni-plugin.log`. These logs serve both as operational diagnostics and as evidence of correct plugin behavior during dynamic container activity. An excerpt of such runtime logs is shown in Figure 4.8 and 4.9, illustrating the execution flow during pod provisioning and deprovisioning.

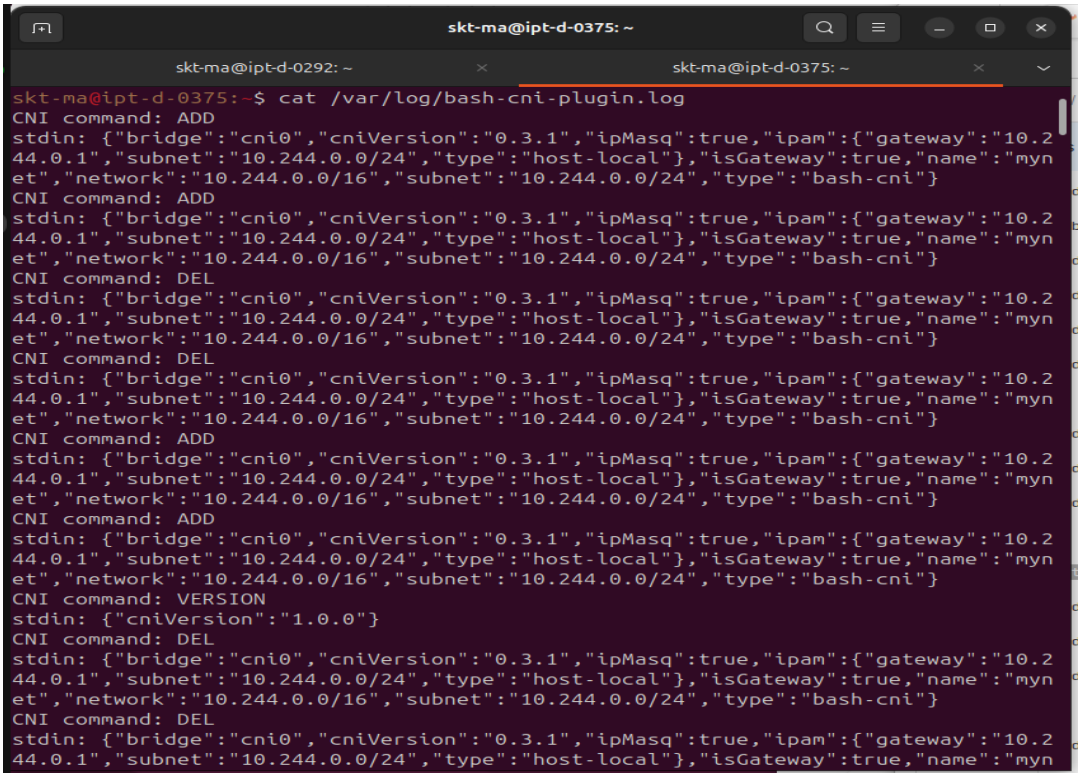


FIGURE 4.8: Master Node Runtime log showing interface creation and IP assignment.

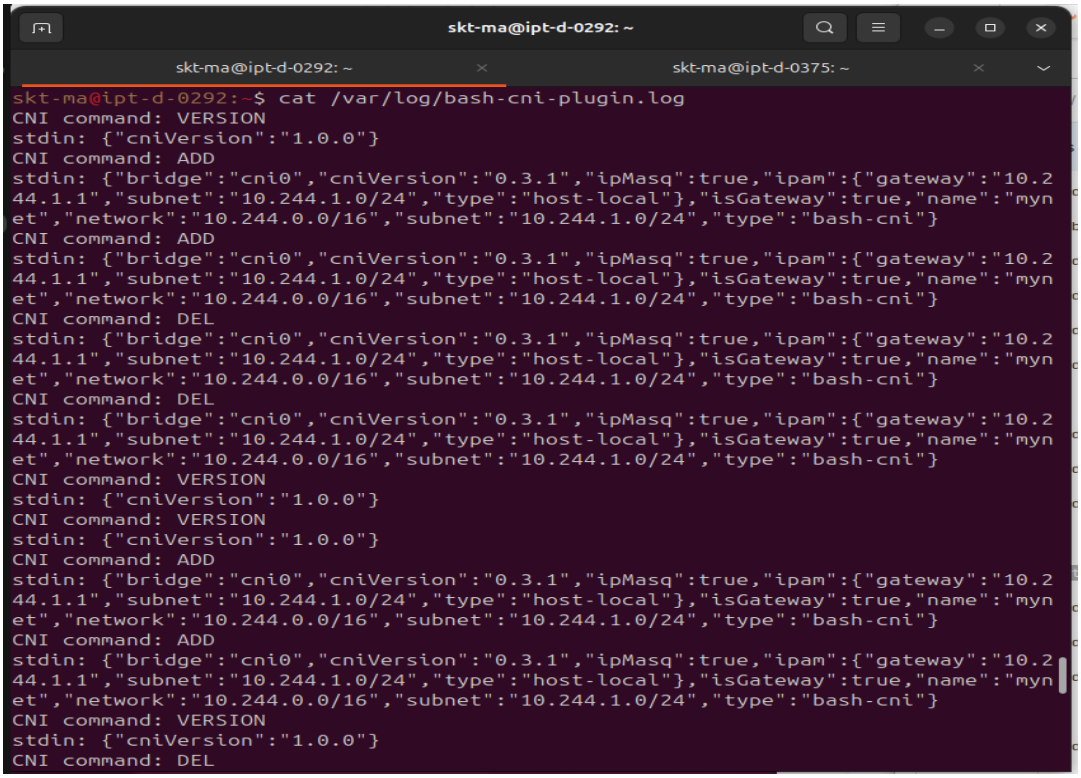


FIGURE 4.9: Worker Node Runtime log showing interface creation and IP assignment.

To validate the integration, the plugin was designated as the default CNI plugin within the Kubernetes cluster. Prior to deployment, critical system pods such as CoreDNS failed to start due to missing network connectivity. Following plugin activation, these pods transitioned to a healthy running state, thereby demonstrating the plugin's compatibility with the Kubernetes networking lifecycle and confirming its baseline correctness. This outcome is depicted in Figure 4.10.

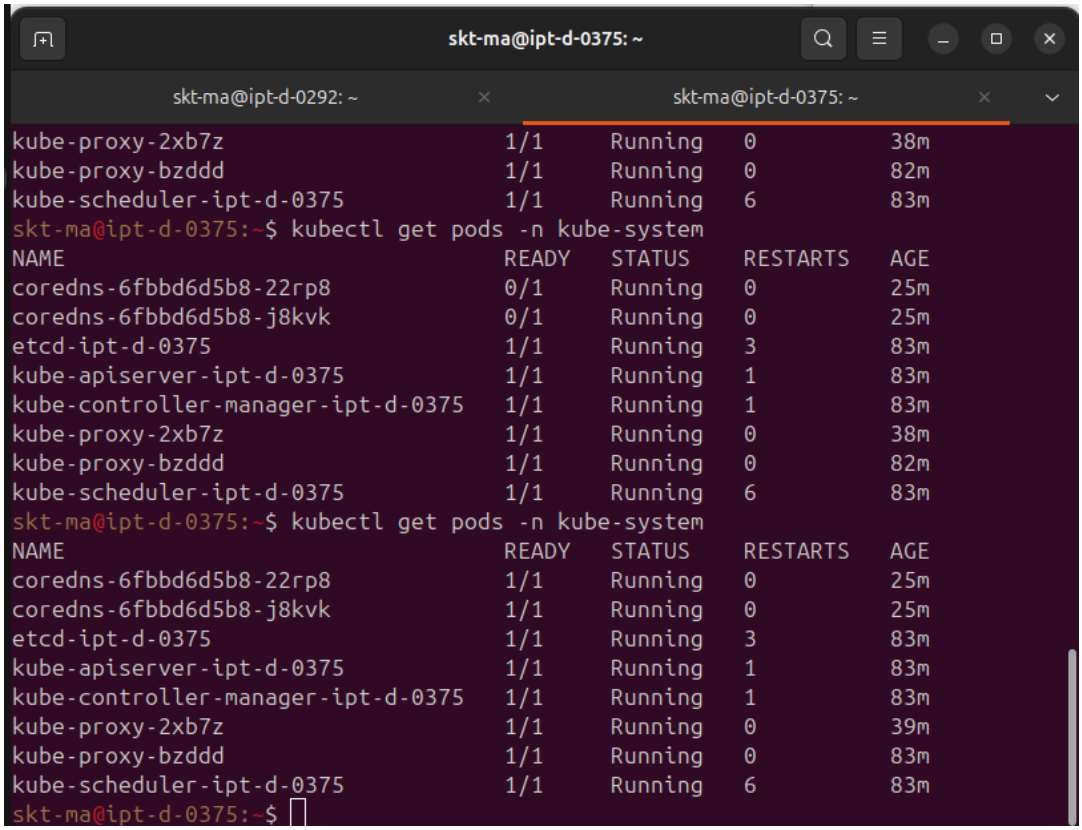


FIGURE 4.10: CoreDNS pods running after plugin activation.

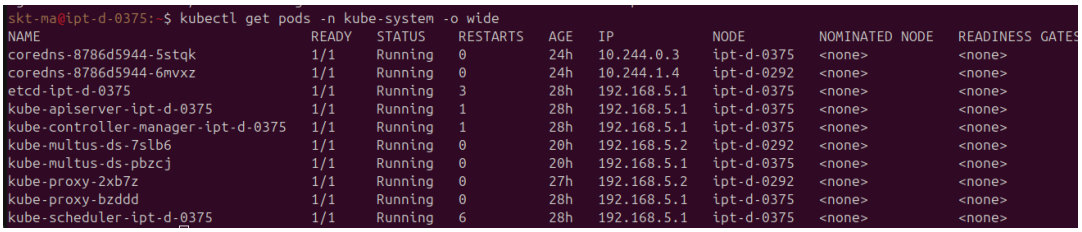
The plugin thus establishes a foundation for deterministic pod-to-pod communication in time-sensitive applications. Its Bash-based implementation, while minimalistic, provides sufficient control over Linux networking primitives to implement TSN-compliant behavior and supports flexible debugging through transparent logging.

4.5 Multus Installation and Configuration

To enable multi-interface support for pods in Kubernetes, the Multus CNI plugin was installed and configured in the cluster. Multus acts as a meta-plugin, allowing pods to attach to multiple networks by chaining additional CNI plugins through `NetworkAttachmentDefinition` (NAD) resources. This setup maintains compatibility with Kubernetes' default networking model while enabling flexible pod-level connectivity.

In this architecture, Multus was used to attach a secondary interface, `net1`, using the `macvlan` plugin. This interface facilitates Layer 2 communication between pods and TSN-capable switches for best-effort traffic. The default interface, `eth0`, is provided by the custom Bash-based CNI and is responsible for deterministic traffic aligned with TSN scheduling requirements.

Multus was installed via the official Helm charts and its deployment was verified using `kubectl`. Figure 4.11 shows the Multus DaemonSet running across all cluster nodes.



```
skt-ma@ipt-d-0375:~$ kubectl get pods -n kube-system -o wide
NAME                                   READY   STATUS    RESTARTS   AGE   IP            NODE          NOMINATED NODE   READINESS GATES
coredns-8786d5944-5stqk                1/1     Running   0          24h   10.244.0.3    ipt-d-0375    <none>           <none>
coredns-8786d5944-6mvxz                1/1     Running   0          24h   10.244.1.4    ipt-d-0292    <none>           <none>
etcd-ipt-d-0375                        1/1     Running   3          28h   192.168.5.1   ipt-d-0375    <none>           <none>
kube-apiserver-ipt-d-0375              1/1     Running   1          28h   192.168.5.1   ipt-d-0375    <none>           <none>
kube-controller-manager-ipt-d-0375     1/1     Running   1          28h   192.168.5.1   ipt-d-0375    <none>           <none>
kube-multus-ds-7slb6                   1/1     Running   0          20h   192.168.5.2   ipt-d-0292    <none>           <none>
kube-multus-ds-pbzcj                   1/1     Running   0          20h   192.168.5.1   ipt-d-0375    <none>           <none>
kube-proxy-2xb7z                       1/1     Running   0          27h   192.168.5.2   ipt-d-0292    <none>           <none>
kube-proxy-bzddd                       1/1     Running   0          28h   192.168.5.1   ipt-d-0375    <none>           <none>
kube-scheduler-ipt-d-0375              1/1     Running   6          28h   192.168.5.1   ipt-d-0375    <none>           <none>
```

FIGURE 4.11: Multus deployed as a daemonset across all cluster nodes.

A representative NAD manifest is shown below. It defines the `macvlan` configuration attached to the TSN NIC and includes static IP allocation.

```
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  name: tsn-macvlan-net
spec:
  config: |
    {
      "cniVersion": "0.3.1",
      "type": "macvlan",
      "master": "ens9",
      "mode": "bridge",
      "ipam": {
        "type": "static",
        "addresses": [
          {
            "address": "192.168.5.203/24",
            "gateway": "192.168.5.1"
          }
        ]
      }
    }
```

Pods that require this secondary interface are annotated as follows:

```
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: tsn-macvlan-net
```

This annotation enables a pod to communicate using both the primary TSN interface (`eth0`) and the `macvlan`-based best-effort interface (`net1`).

4.5.1 Macvlan Setup for Secondary Interface

The `macvlan` plugin was selected to provide Layer 2 connectivity for `net1` interfaces. Within each pod, this interface is instantiated by Multus and mapped to the TSN NIC. The configuration uses bridge mode, enabling multiple pods to share the physical NIC while maintaining separate IP identities.

Figure 4.12 shows the pod's internal interface structure, confirming the presence of both `eth0` (custom CNI) and `net1` (macvlan).

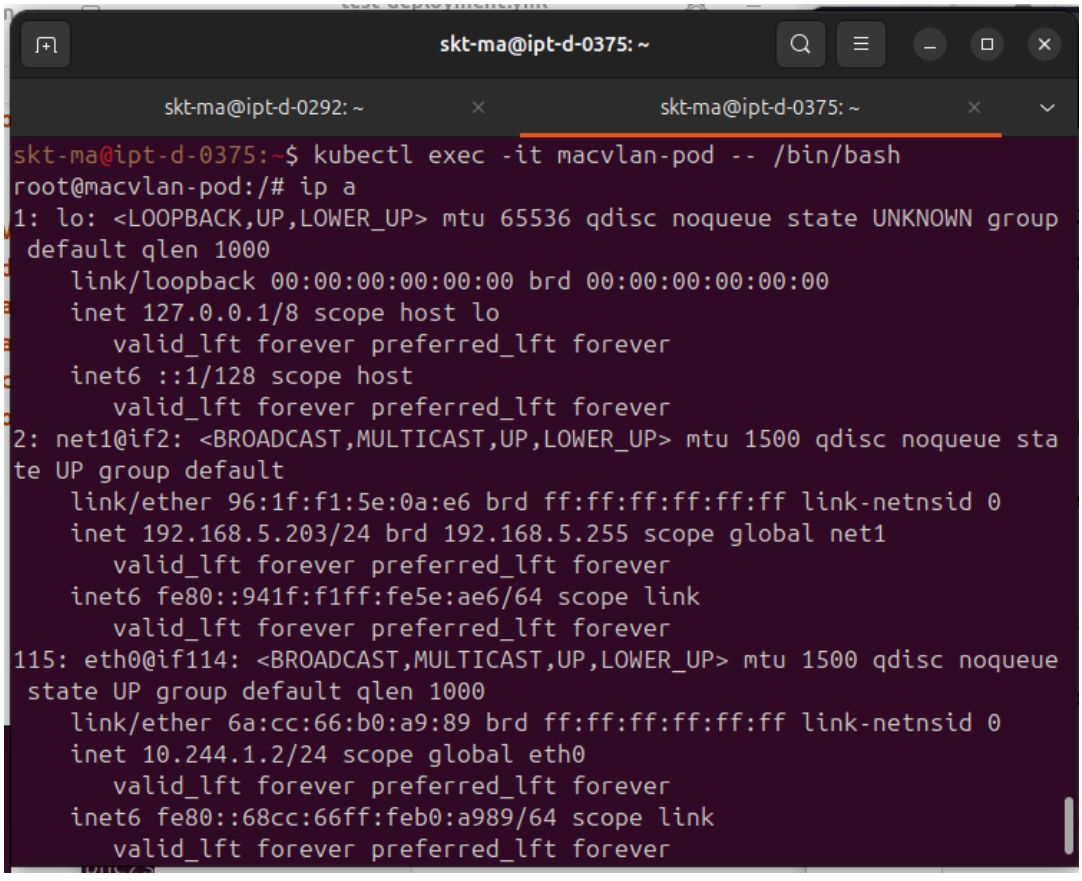


FIGURE 4.12: Pod with `net1` (macvlan) interface for Layer 2 reachability.

Connectivity between the pod and host was validated using ICMP echo requests, as shown in Figure 4.13.

Figure 4.14 demonstrates pod-to-pod communication across the TSN switches using the `macvlan` interface.

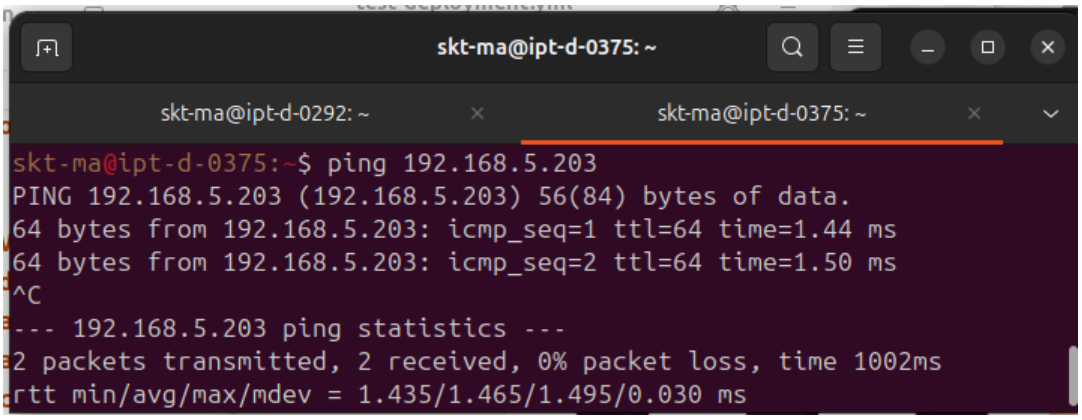


FIGURE 4.13: ICMP ping from macvlan-enabled pod to host.

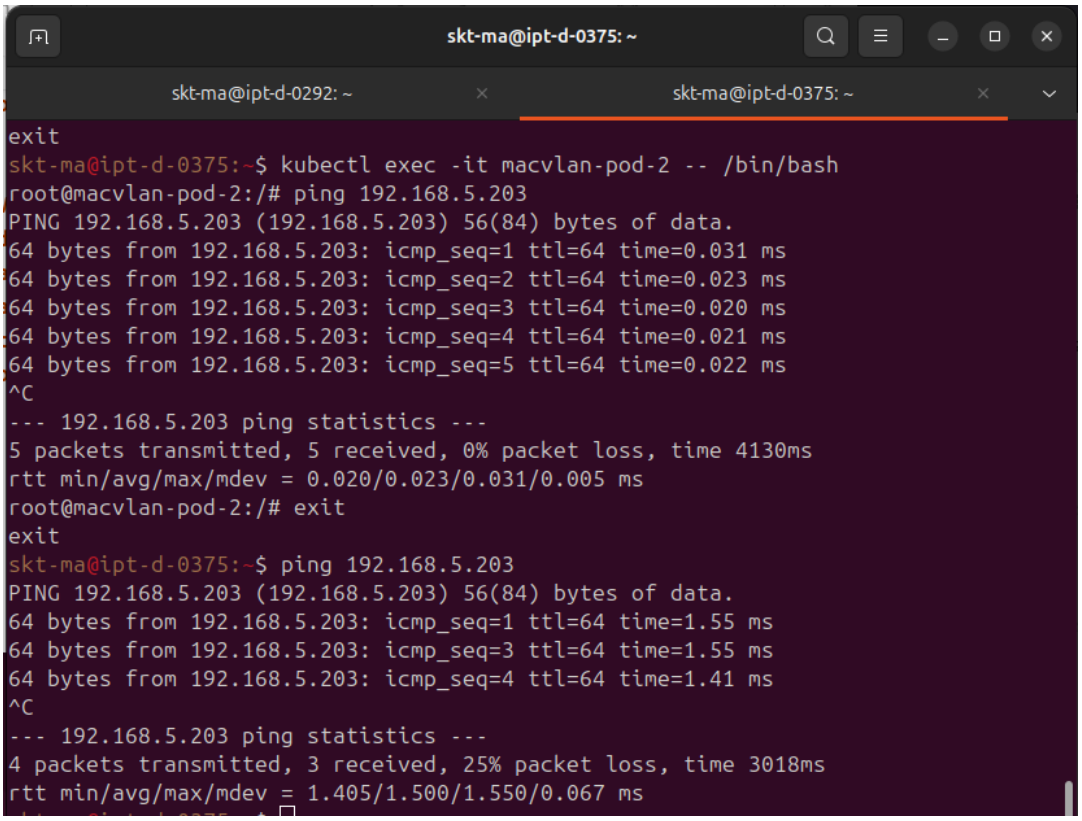


FIGURE 4.14: Ping test between macvlan-enabled pods across TSN infrastructure.

This dual-interface configuration effectively separates deterministic TSN traffic (`eth0`) from best-effort Layer 2 traffic (`net1`). It enables flexible testing and reliable evaluation of communication behaviors without interference between traffic classes.

4.6 CNC and Switch Configuration via NETCONF

To orchestrate deterministic communication across the TSN infrastructure, a CNC was developed using Python and the `ncclient` library. The CNC facilitates remote configuration of GCLs and time-aware scheduling parameters compliant with the IEEE 802.1Qbv standard.

The CNC establishes NETCONF sessions with TSN switches using XML-encoded `edit-config` operations aligned with YANG data models supported by the switch. Pre-deployment validation of NETCONF connectivity was performed using the `netopeer2-cli` command-line interface, as illustrated in Figure 4.15.

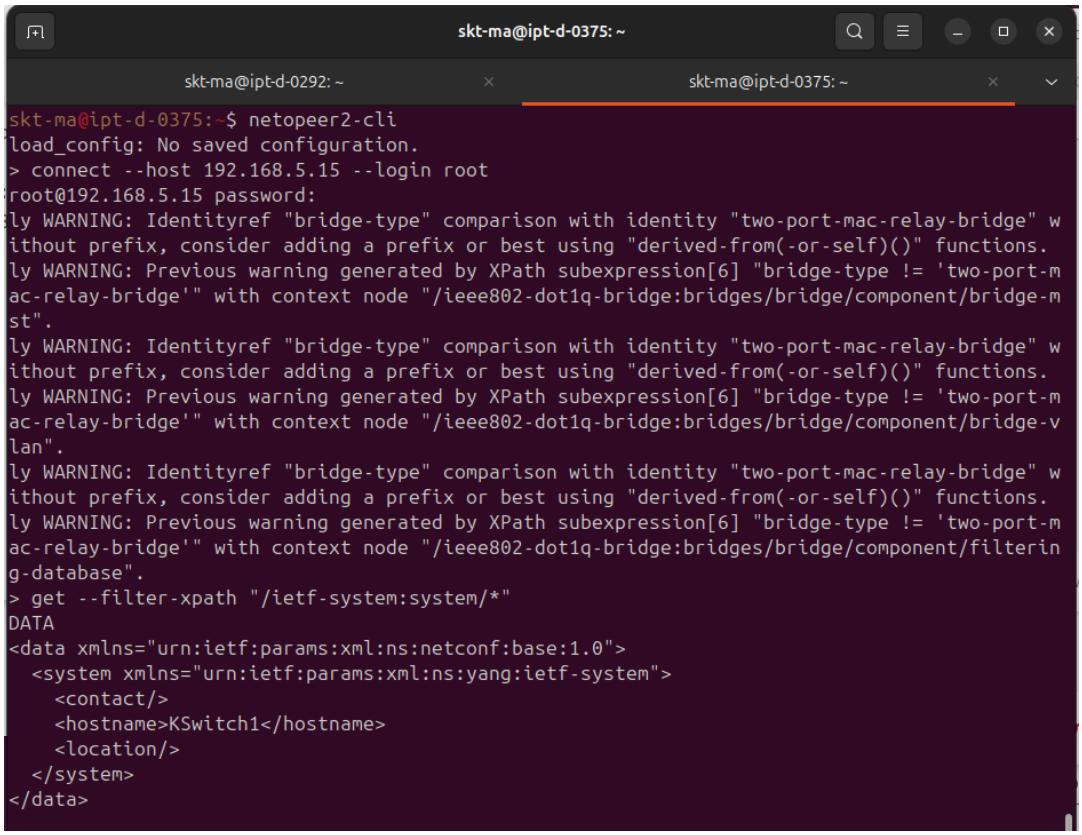


FIGURE 4.15: Using Netopeer2 CLI to retrieve system information from a TSN switch via NETCONF.

The CNC successfully executed dynamic configuration commands such as hostname updates and GCL distribution. Figure 4.16 shows the Python-based CNC applying NETCONF edits to a TSN switch during an active session.

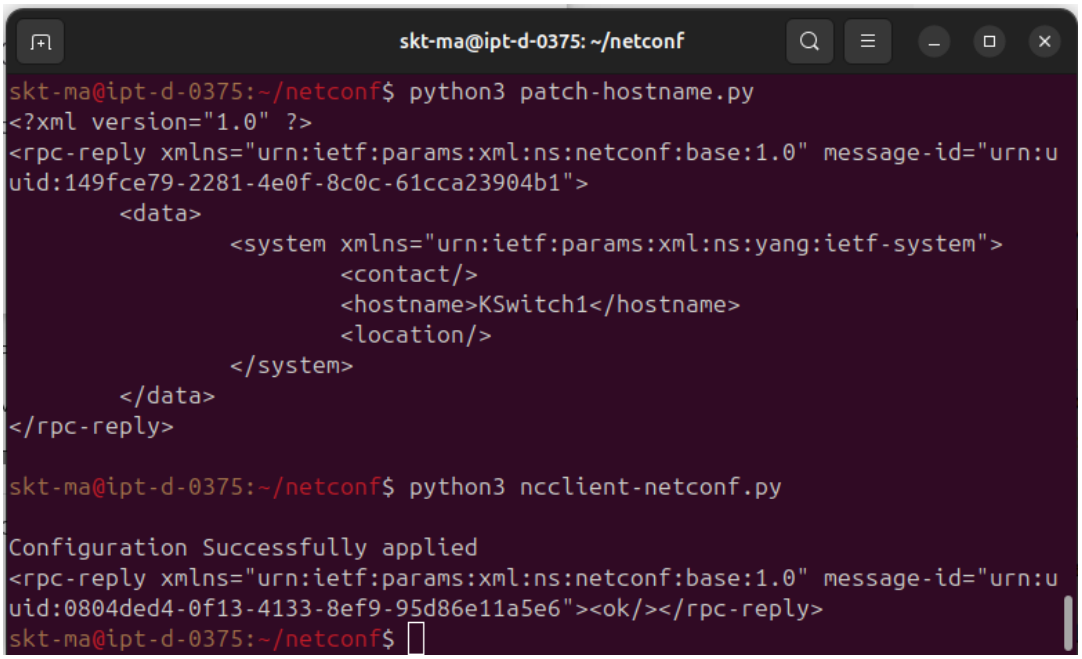


FIGURE 4.16: Python CNC pushing TSN configuration to a switch using NETCONF and YANG.

Prior to applying transmission schedules, the CNC confirms that time synchronization is established between the master and worker nodes using the PTP stack. This alignment ensures accurate gate timing for scheduled traffic transmission across TSN links.

4.7 Experimental Traffic and Measurement Setup

The testbed was designed to evaluate the performance of TSN-enabled Kubernetes communication using controlled traffic and synchronization mechanisms. It consisted of a Kubernetes master node, a worker node, and interconnected TSN switches.

Each node was provisioned with two network interfaces:

- One interface for standard Kubernetes control-plane communication
- One dedicated interface for TSN-based data exchange

Clock synchronization was achieved using the LinuxPTP toolset. On the master node, `ptp4l` was configured to serve as the Grandmaster Clock. A representative log of this service is shown in Figure 4.17.

The worker node synchronized its system clock using `phc2sys`, ensuring alignment with the hardware clock and the Grandmaster. Figure 4.18 shows a sample log of this process.

To evaluate the communication path, traffic was generated using a Go-based UDP sender and receiver deployed across the TSN interfaces. These applications exchanged timestamped packets and logged transmission latency and jitter. Tests were performed under multiple traffic conditions, with and without active scheduling or TSN-enabled paths.

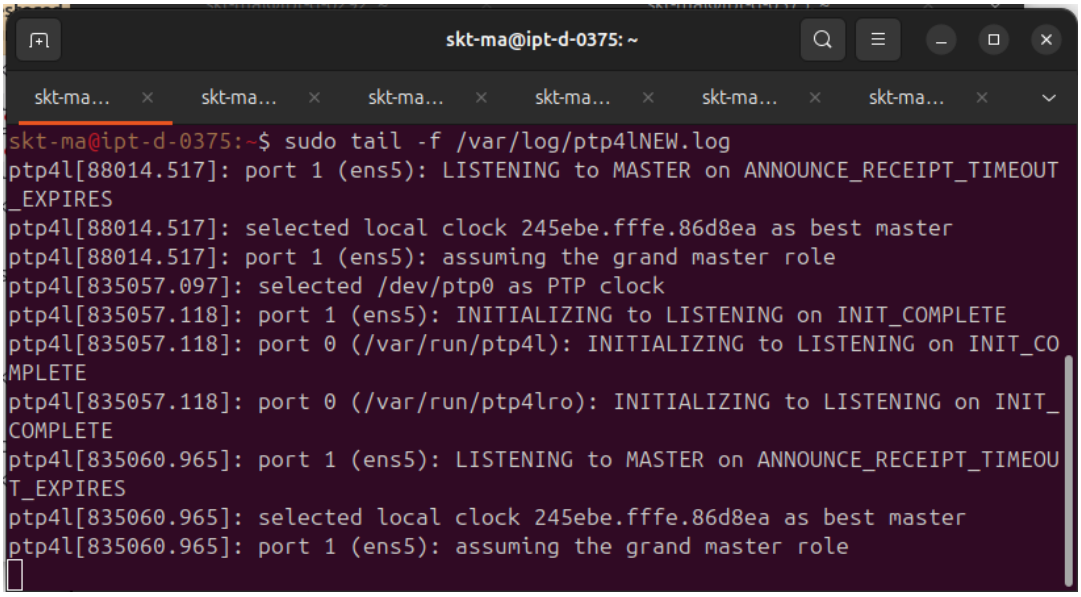


FIGURE 4.17: `ptp4l` operating on the master node to establish grandmaster clock synchronization.

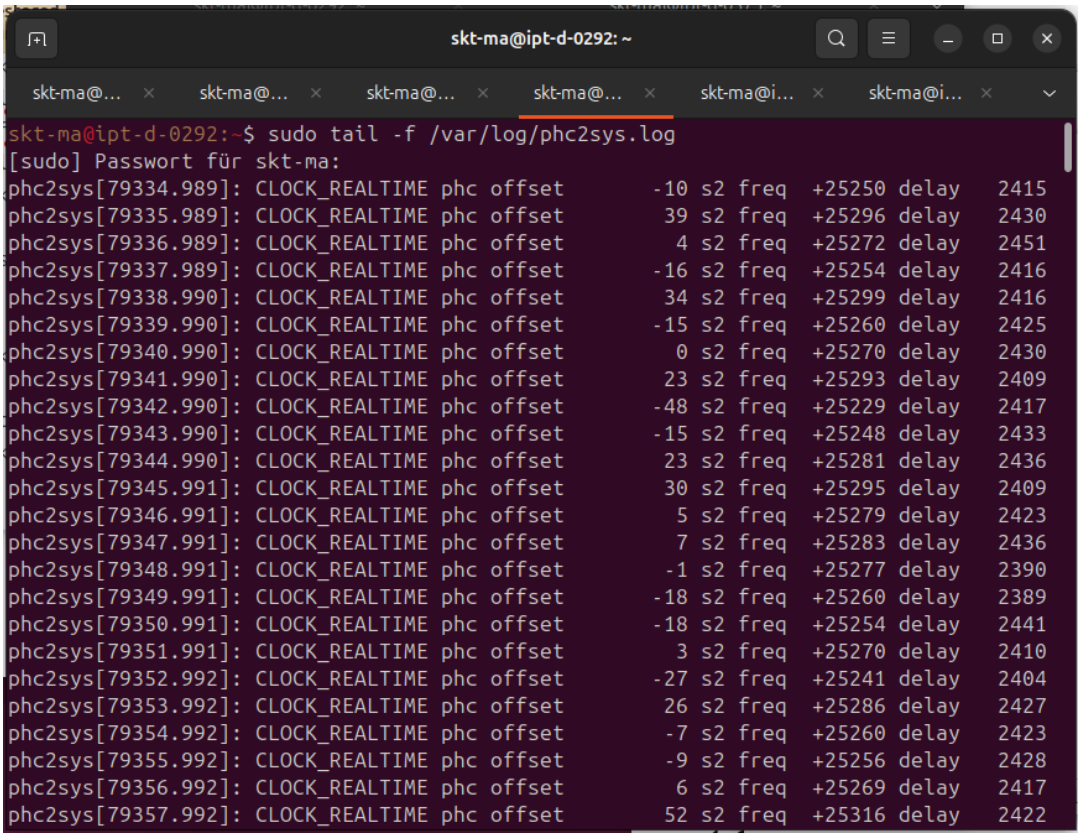


FIGURE 4.18: `phc2sys` running on the worker node to maintain synchronized time with the Grandmaster.

The experimental data collected from these measurements is analyzed and presented in Chapter 5, with a focus on evaluating the effects of synchronization, scheduling, and CNI configurations on real-time network performance.

Chapter 5

Validation Overview

5.1 Validation Goals and Scope

The purpose of the validation phase is to rigorously evaluate whether the integration of Kubernetes with TSN—as implemented through custom CNI extensions and synchronized TSN switch configurations—supports the deterministic communication requirements expected in industrial environments.

This evaluation focuses on two primary performance metrics:

Bounded End-to-End Latency: The system must ensure that latency between communicating pods remains tightly bounded and stable, even under background traffic and real-time scheduling. Based on TSN use cases in factory automation and robotics, the target latency for this system is ≤ 1 millisecond, with preference toward sub-500 µs operation where possible.

Minimal Jitter: Jitter, or variation in packet delivery intervals, should remain under 50 microseconds to support time-critical operations such as sensor-actuator loops and motion control tasks. This requirement aligns with OPC UA and IEEE TSN industrial profiles.

Although synchronization accuracy is not a primary validation metric in this study, it is a critical prerequisite. The reliability of time-aware scheduling (IEEE 802.1Qbv) depends on tight clock alignment across TSN switches and host NICs. Therefore, PTP-based synchronization was monitored to ensure sub-microsecond offset levels, enabling valid latency and jitter testing.

These metrics were initially evaluated across foundational communication scenarios described in Chapter 4, including Kubernetes pod-to-pod, pod-to-device, and device-to-pod interactions. Though extttttaprio was configured inside the custom CNI plugin and the TSN switch, the resulting measurements did not reflect effective traffic prioritization.

Therefore, the testbed was extended with three refined scenarios to consider from a measurement and performance evaluation perspective.

5.2 Extended Experimental Communication Scenarios

To systematically validate the deterministic networking capabilities of the Kubernetes–TSN integrated system, three experimental communication scenarios were designed. Each scenario incrementally applies TSN scheduling mechanisms, allowing a comparative evaluation of their impact on end-to-end latency and jitter.

Note: Throughout all measurements, background network load was generated using `iperf3` sessions between non-tested pods to simulate network congestion and stress test the deterministic capabilities of the system.

5.2.1 Scenario 1: Baseline VLAN Priority Communication (Without Time-Aware Scheduling)

In the initial baseline scenario, basic VLAN-based priority marking was applied to outgoing UDP packets, but no time-aware scheduling mechanisms were activated at either the host or switch side.

- **Host Setup:**
  - VLAN tagging was applied using Priority Code Points (PCP):
    - `PCP=7` for high-priority real-time traffic.
    - `PCP=1` for low-priority background traffic.
  - No `taprio` queuing discipline configured on the host TSN NICs.
- **Switch Setup:**
  - TSN switches operated in standard Layer 2 forwarding mode.
  - No Gate Control Lists (GCLs) applied.
- **Objective:** Establish a baseline to observe latency and jitter behavior with PCP tagging alone, without strict time-slot-based control.
- **Expectation:** Minor prioritization based on PCP, but high variability in latency and jitter due to lack of synchronized gating.
- **Measurement Notes:** UDP packets were transmitted at 1 ms, 2 ms, and 4 ms cyclic intervals. Background `iperf3` traffic was continuously active.

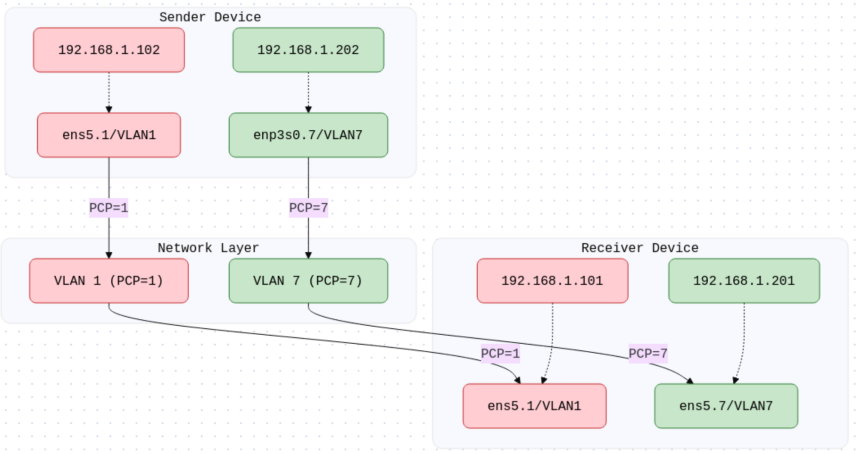


FIGURE 5.1: Baseline VLAN Priority Communication Setup (Scenario 1)

5.2.2 Scenario 2: Switch-Level Time-Aware Scheduling

In this intermediate scenario, Time-Aware Scheduling (TAS) was activated at the TSN switches to enforce deterministic transmission windows based on time.

- **Host Setup:**
  - VLAN tagging (`PCP=7` and `PCP=1`) remained active.
  - No `taprio` configuration applied at the pod or node NIC level.
- **Switch Setup:**
  - Gate Control Lists (GCLs) manually configured on TSN switches.
  - Critical traffic streams (`PCP=7`) allocated exclusive transmission windows.
- **Objective:** Validate the improvement in latency stability and jitter reduction when only switch-side TAS is applied.
- **Expectation:** Lower jitter and more predictable packet delivery for high-priority traffic compared to Scenario 1.
- **Measurement Notes:** Identical packet generation settings and background `iperf3` load maintained.

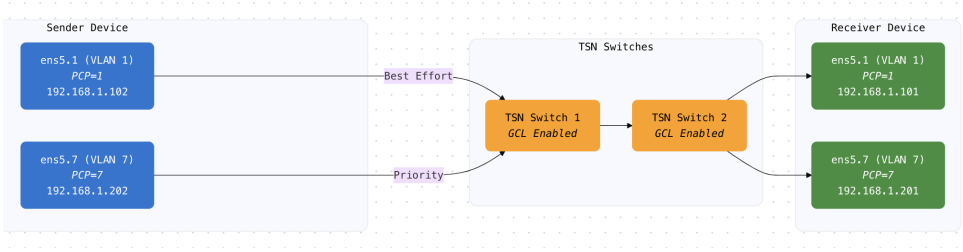


FIGURE 5.2: Switch-Level Time-Aware Scheduling (Scenario 2)

5.2.3 Scenario 3: Full TSN Scheduling (Pod + Switch Synchronization)

This final scenario represents the complete deterministic system where scheduling is tightly coordinated between Kubernetes pods, hosts, and TSN switches.

- **Host Setup:**
  - The custom Bash-based CNI plugin configured `taprio` queuing disciplines on TSN NICs.
  - Host-side taprio schedules synchronized with switch-side GCLs.
  - VLAN PCP tagging dynamically assigned based on pod annotations.
- **Switch Setup:**
  - Gate Control Lists (GCLs) dynamically pushed by the Centralized Network Controller (CNC) using NETCONF/YANG.
  - Switches operated in synchronized mode, using a gPTP-distributed clock.
- **Objective:** Demonstrate bounded low-latency and minimal jitter performance under complete TSN scheduling coordination.
- **Expectation:** Highest determinism achieved with minimal latency variation and microsecond-level synchronization.
- **Measurement Notes:** UDP traffic at 1 ms, 2 ms, and 4 ms cycle times under active background `iperf3` load.

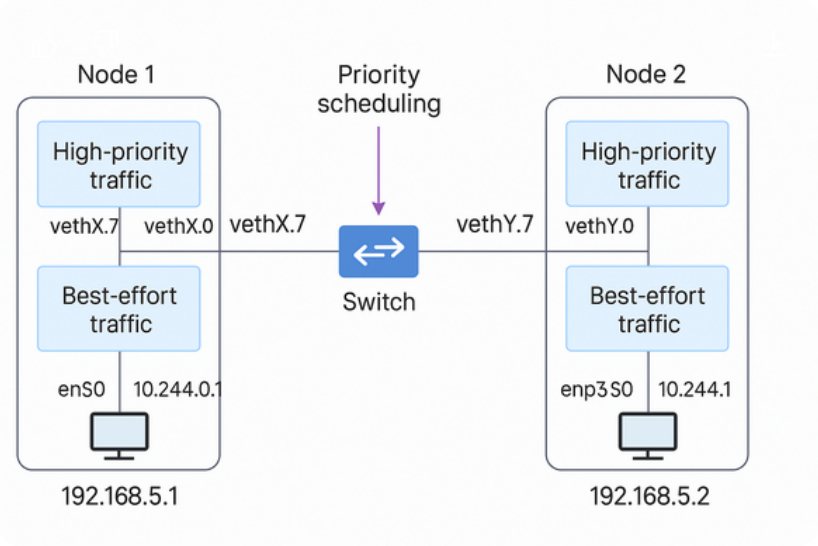


FIGURE 5.3: Switch-Level Time-Aware Scheduling (Scenario 2)

5.3 Measurement Methodology

To ensure statistical robustness and technical accuracy, a structured measurement methodology was designed and applied across all validation scenarios. This methodology involved controlled traffic generation, clock synchronization verification, precise timestamping, and standardized offline analysis of latency and jitter behavior.

5.3.1 Measurement Architecture

The experimental architecture consisted of a two-node Kubernetes cluster interconnected via TSN-capable Ethernet switches. Each node was equipped with two network interfaces:

- One interface for Kubernetes control-plane communication and Internet access.
- One dedicated interface for TSN-enabled data-plane traffic.

Time synchronization across the entire system was achieved using IEEE 802.1AS Generalized Precision Time Protocol (gPTP). Hardware timestamping was enabled on the TSN-capable Intel I225 NICs to ensure microsecond-level accuracy.

Within the Kubernetes environment:

- Pods were attached to TSN NICs using Multus and a custom Bash-based TSN CNI plugin.
- `taprio` queuing disciplines were dynamically configured where applicable (Scenario 3).

Traffic was exclusively routed through the deterministic TSN interfaces, bypassing the Kubernetes overlay network for minimized latency and jitter influence.

5.3.2 Measurement Tools and Software Environment

The following tools and frameworks were utilized during the measurement campaigns:

Tool/Framework	**Purpose**
Custom Go UDP Traffic Generator	Generation of timestamped UDP packets with configurable cyclic intervals (1 ms, 2 ms, 4 ms).
LinuxPTP (ptp4l, phc2sys)	Synchronization of hardware clocks and system clocks across nodes and switches.
iperf3	Generation of background network traffic to simulate congestion scenarios.
Wireshark	Packet capture and inspection using hardware timestamping capabilities.
Python (Pandas, Matplotlib, Seaborn)	Offline latency and jitter analysis, statistical aggregation, and graph generation.
tc (Traffic Control)	Configuration and inspection of `taprio` queuing disciplines on TSN NICs.
NETCONF and Netopeer2-cli	Switch configuration validation and GCL schedule verification.

TABLE 5.1: Measurement tools and their purposes.

To simulate congested environments, background traffic was generated using the `iperf3` tool. The sender node initiated 10 parallel TCP streams towards the receiver node during test runs. These streams saturated the standard Ethernet channel and helped evaluate the effect of background load on latency and jitter performance.

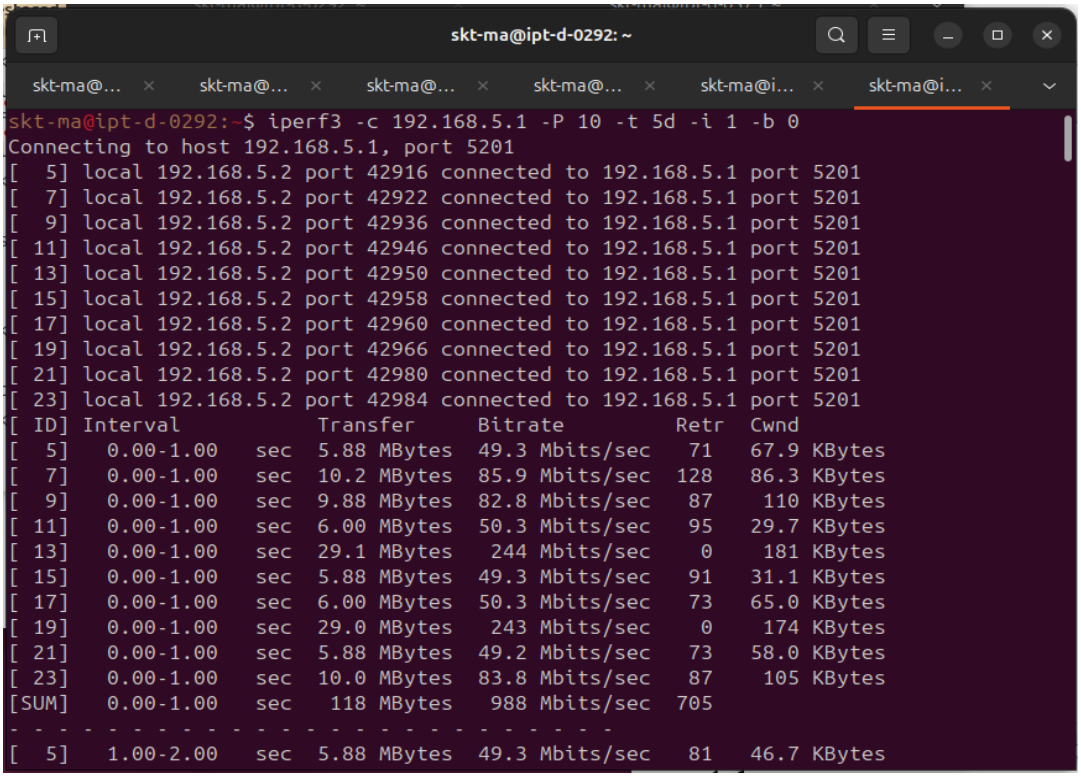


FIGURE 5.4: `iperf3` sender-side: Generating 10 parallel TCP streams to stress the network.

5.3.3 Latency and Jitter Calculation

Latency and jitter metrics were derived based on precise timestamp data collected during transmission and reception of each UDP packet.

Latency Calculation

The latency for each packet i was calculated as:

$$\text{Latency}(i) = t^{(i)}_{\text{Received}} - t^{(i)}_{\text{Sent}} \tag{5.1}$$

where:

- $t^{(i)}_{\text{Sent}}$ is the transmission timestamp,
- $t^{(i)}_{\text{Received}}$ is the reception timestamp.

The average latency across all received packets was computed as:

$$\text{Avg. Latency}_{\mu s} = \frac{1}{N} \sum_{i=1}^{N} \frac{t^{(i)}_{\text{Received}} - t^{(i)}_{\text{Sent}}}{1000} \tag{5.2}$$

where N is the total number of received packets.

Latency values were scaled appropriately to microseconds (µs) for standardized presentation.

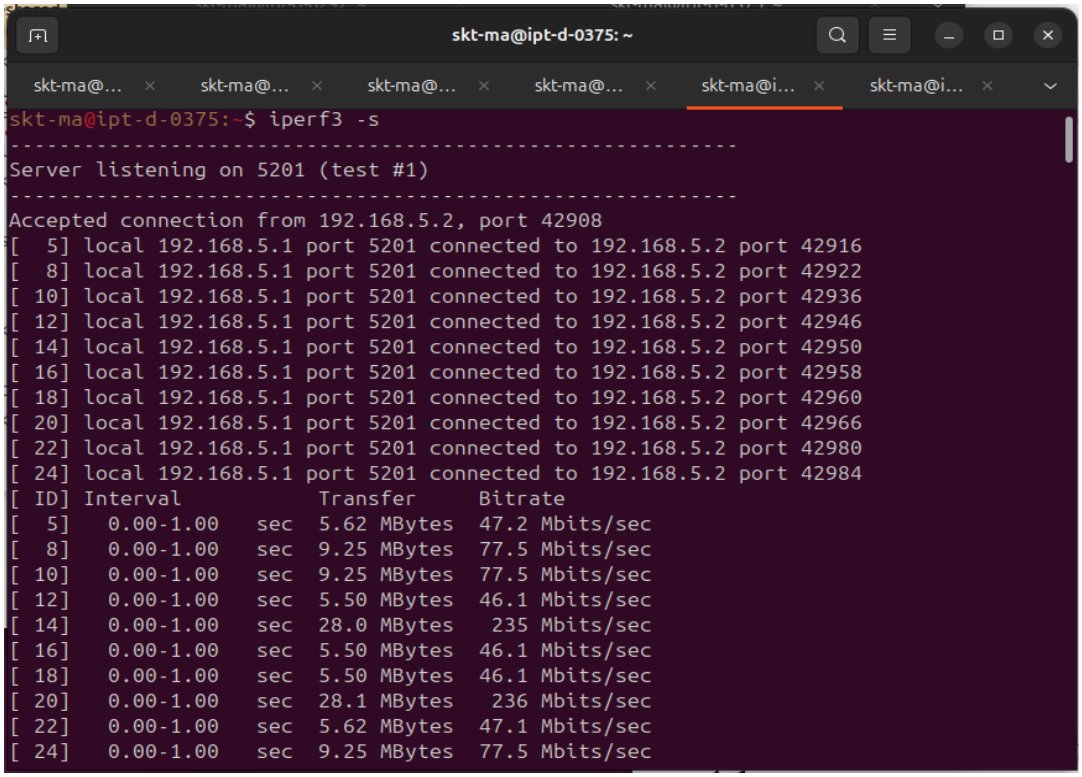


FIGURE 5.5: `iperf3` server-side: Receiving concurrent streams to simulate congestion during TSN measurements.

Jitter Calculation

Jitter was calculated as the variation from the expected cyclic packet transmission interval T (typically 1 ms):

$$\text{Jitter}(i) = |t_i - (t_{i-1} + T)| \tag{5.3}$$

where:

- t_i and t_{i-1} are consecutive packet reception timestamps,
- T is the configured cyclic interval (1 ms, 2 ms, or 4 ms depending on the test).

This method captures irregularities in packet inter-arrival timings, isolating transmission and scheduling effects from network propagation variations.

Key Metrics Evaluated

The following performance indicators were extracted for each scenario:

- Average End-to-End Latency
- Minimum and Maximum Latency
- 99th, 99.9th, and 99.99th percentile Latency
- Average Jitter
- Maximum Jitter
- Jitter Distribution Profiles

5.3.4 Measurement Procedure

Each measurement campaign followed a repeatable, scripted procedure to ensure consistency:

1. **Environment Preparation:** Kubernetes cluster re-synchronized via `ptp4l` and `phc2sys`. `taprio` queues and GCL schedules configured as per the scenario.
2. **Traffic Generation:** Go-based UDP sender and receiver applications deployed inside designated pods. Traffic generated at fixed intervals (1 ms, 2 ms, 4 ms) for approximately 100,000 packets per run. Background `iperf3` traffic initiated between auxiliary pods.
3. **Data Logging:** Transmission and reception timestamps recorded into structured CSV files. Wireshark captures collected using NIC hardware timestamping.
4. **Offline Analysis:** Latency and jitter calculated based on timestamp logs. Statistical summaries and graphs generated using Python libraries. Verification of VLAN tagging, PCP prioritization, and GCL behavior via Wireshark analysis.
5. **Repetition:** Each configuration measured in at least three independent runs to verify result stability. Averages and standard deviations reported.

Chapter 6

Results and Analysis

6.1 Measurement Scenarios and Goals

In order to validate the proposed Kubernetes-TSN integration architecture, a systematic evaluation was conducted across multiple test environments and traffic conditions. Each scenario was designed to progressively introduce more deterministic network features, allowing direct comparison between baseline and TSN-augmented setups.

The primary goals of the measurements were:

- Quantify latency and jitter under varying configurations.
- Observe the impact of TSN mechanisms via `taprio` and switch GCL scheduling) on communication performance.
- Evaluate system behavior under both idle and network-congested conditions (background `iperf` traffic).
- Benchmark improvements compared to k8 custom CNI and traditional CNI-based Kubernetes setups like Flannel and macvlan.

While recording timestamps for end-to-end latency and inter-packet jitter computation. Whenever applicable, `iperf3` was used to simulate background traffic and stress-test the network under load.

The measurements were categorized into three groups:

- Host-Level (bare-metal to bare-metal, with and without TSN switches)
- Docker Containers (container-to-container communication)
- Kubernetes Pods (Pod-to-Pod with various CNIs, including Multus and custom TSN CNI)

Results were analyzed using:

- Latency scatter plots
- Latency histograms
- Jitter scatter plots
- Jitter histograms

This multi-dimensional approach ensures a robust evaluation of deterministic behavior at all layers of the proposed system.

6.2 Overview of Measurement Method and Result Analysis

After completing each measurement run, raw timestamp data was exported to CSV format by the Go-based client/server applications. These CSV files contained per-packet send and receive times, from which latency and jitter were calculated.

The analysis workflow consisted of:

- Running a custom Python script (`analyze_application_latency.py`) to process the CSV files.
- Calculating average, minimum, maximum, and 99th/99.9th/99.99th percentile values for latency and jitter.
- Generating plots, including:
  - Scatter plot of packet latency vs packet count
  - Histogram of latency distribution
  - Scatter plot of jitter vs packet count
  - Histogram of jitter distribution

Each plot provided critical insight:

- **Latency Scatter Plot**: Highlights individual outliers and variations across packet transmissions.
- **Latency Histogram**: Shows the statistical spread and clustering of packet latencies.
- **Jitter Scatter Plot**: Reveals how packet-to-packet variation changes over time.
- **Jitter Histogram**: Aggregates jitter behavior to detect systemic patterns.

These results allow identification of:

- Maximum deviation in packet timing.
- Stability and predictability of network behavior.
- Presence of microbursts, delays, or jitter spikes.

In this chapter, a detailed explanation of one representative measurement will be provided alongside all relevant plots. Additional scenarios and measurements are summarized in tabular form for comparative analysis in Section **??**.

6.3 Bare-Metal Direct Connection Measurement

This section presents the results of the bare-metal direct connection measurements without Kubernetes or Docker overhead. Two physical hosts exchanged UDP packets directly, without passing through any TSN switch or additional network infrastructure. The test involved sending 1 million packets at 10 ms intervals using a custom Go-based UDP measurement script.

6.3.1 Latency Analysis

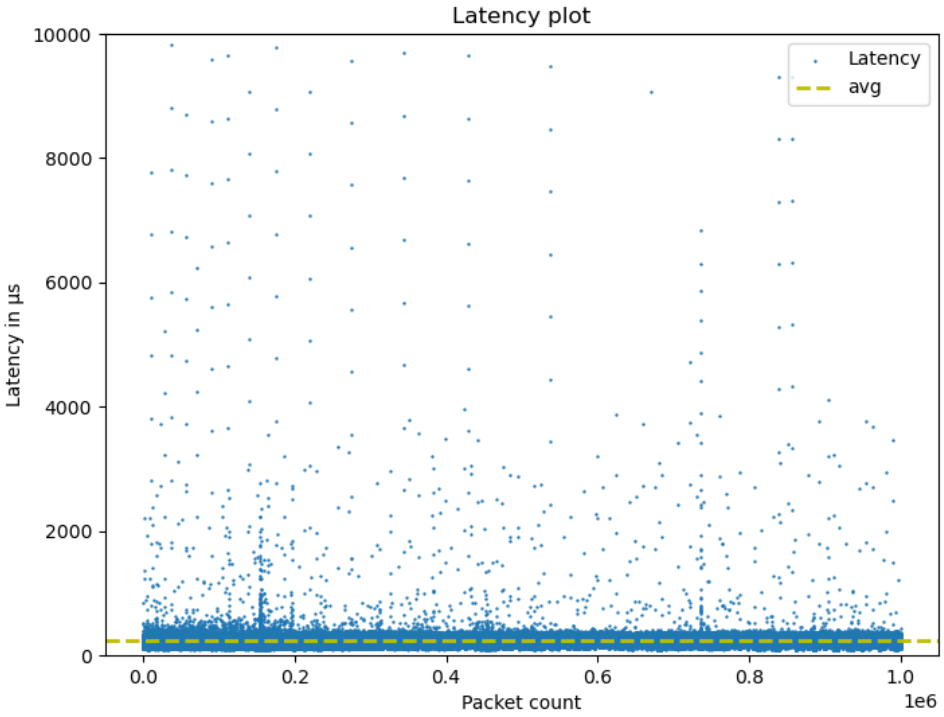


FIGURE 6.1: Latency Scatter Plot: Bare-Metal Host-to-Host

Interpretation:

- The majority of packet latencies cluster tightly around the mean value.
- Sporadic outliers are visible, reaching up to approximately 8000–9000 μs.
- The yellow dashed line indicates the calculated average latency of around 243 μs, confirming low average delay in the direct connection setup.

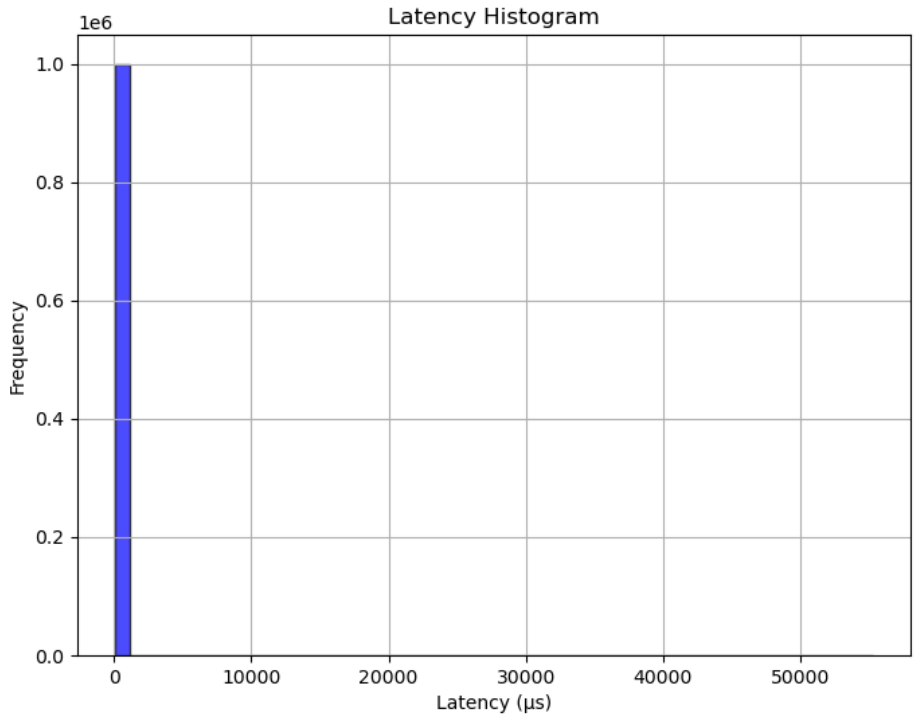


FIGURE 6.2: Latency Histogram: Bare-Metal Host-to-Host

Interpretation:

- Over 95% of the packets experienced latency below 500 μs, as indicated by the steep peak on the left side of the histogram.
- A few extreme cases are recorded beyond 1000 μs, but they are statistically negligible.

6.3.2 Jitter Analysis

Interpretation:

- The jitter values mostly remain under 1000 μs, with a small number of isolated spikes up to 8000 μs.
- The distribution remains relatively stable over the packet transmission duration, suggesting no major drift over time.

Interpretation:

- The majority of packets experienced jitter values below 200 μs.
- A minor tail towards higher jitter values indicates occasional packet bursts or delays, but the overall occurrence remains low.

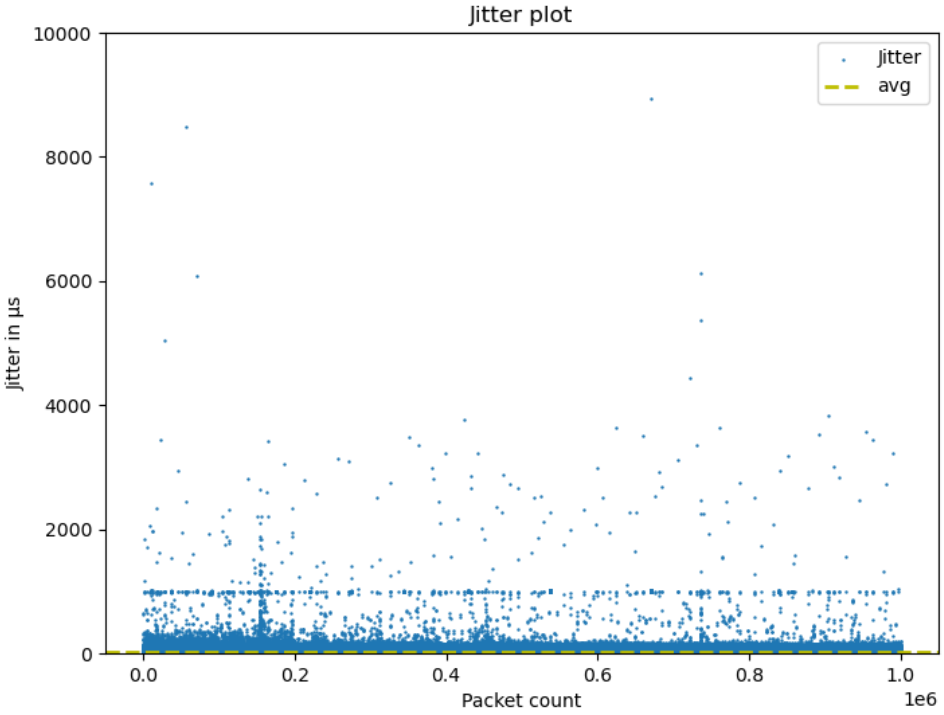


FIGURE 6.3: Jitter Scatter Plot: Bare-Metal Host-to-Host

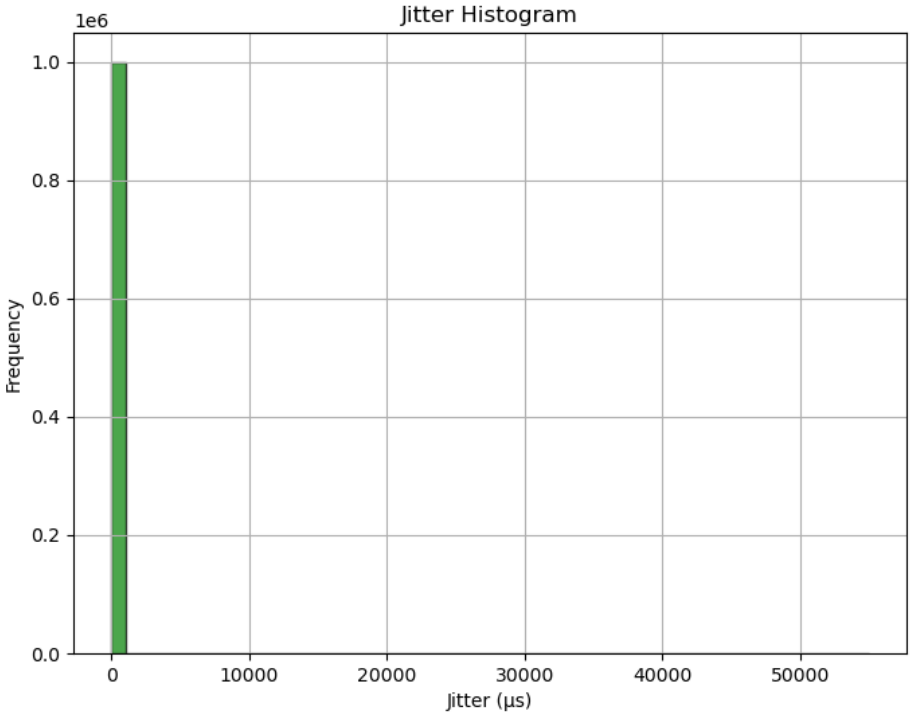


FIGURE 6.4: Jitter Histogram: Bare-Metal Host-to-Host

6.3.3 Statistical Summary

To complement the graphical analysis, Table 6.1 summarizes the key numerical statistics extracted from the measurement logs using a Python-based analysis script.

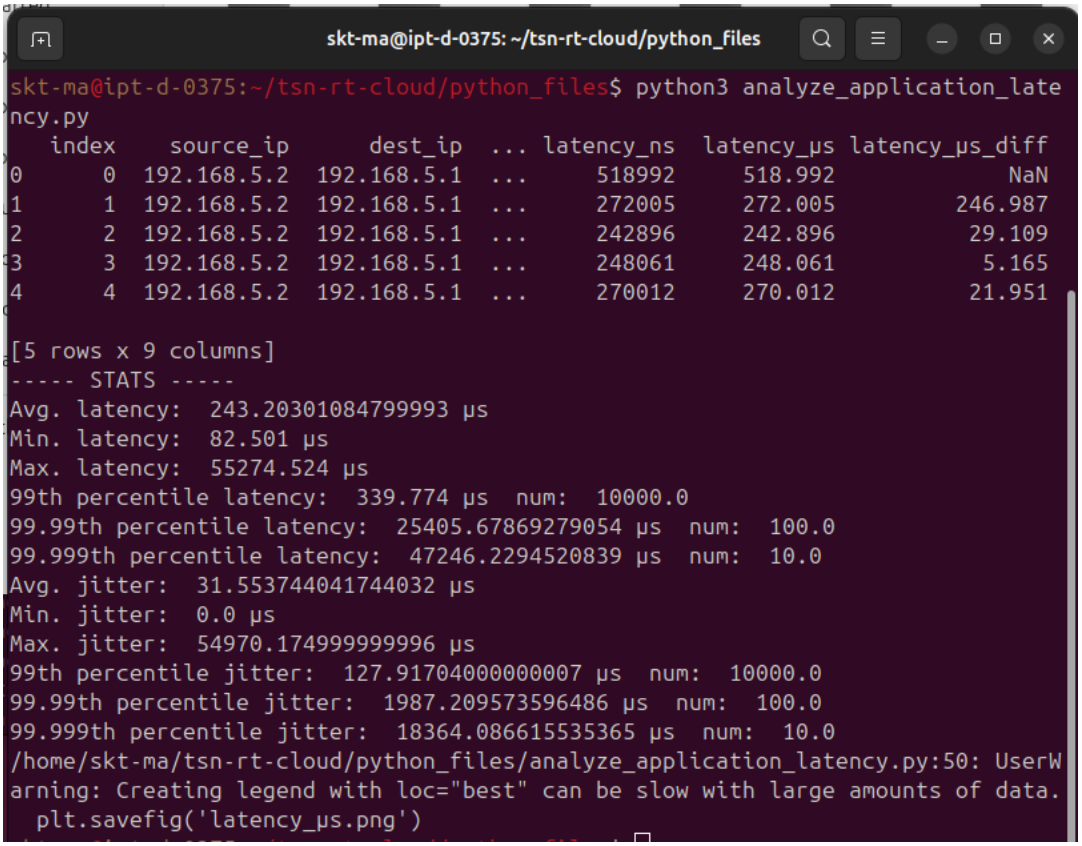


FIGURE 6.5: Python script output showing detailed latency and jitter statistics for the bare-metal direct measurement.

TABLE 6.1: Latency and Jitter Statistics for Bare-Metal Direct Connection

Metric	Value
Average Latency	243.2 μs
Maximum Latency	55,724.5 μs
99th Percentile Latency	339.8 μs
Average Jitter	31.5 μs
99th Percentile Jitter	1987.2 μs

These results confirm that, while most packets experienced minimal latency and jitter, rare outliers introduced much higher delays. Such occurrences highlight the need for tighter scheduling and queue control, motivating the application of TSN scheduling techniques in subsequent tests.

6.3.4 Measurement Table

The maximum latency and jitter results collected across different test scenarios are summarized in Table 6.2.

Test Scenario	**Max Latency (μs)**	**Max Jitter (μs)**
Bare-Metal (Direct Connection)	55725	54970
Bare-Metal via TSN Switch	5991	4605
Custom CNI without TSN Switch	46409.849	42450.39
Custom CNI via TSN Switch	35916	35719
Docker with TSN Switch	26514	26366
Macvlan Interface (Multus)	32042	31860
Flannel CNI (default)	47782	47631
Bare-Metal under Background Load (iperf3)	46829	34818
Pod 50k Packets, 1ms Cycle	35474	35232
Pod 50k Packets, 1ms Cycle with iperf3 Load	55058	60592
Pod 50k Packets, 2ms Cycle	36044	35807
Pod 50k Packets, 2ms Cycle with iperf3 Load	65894	62037

TABLE 6.2: Summary of Maximum Latency and Maximum Jitter Across Test Scenarios

6.3.5 Interpretation

The measurement results provide key insights into latency and jitter behavior across various deployment scenarios in the Kubernetes–TSN integration framework. The bare-metal direct connection scenario exhibited the highest latency and jitter, both exceeding 55,000µs, underscoring the unpredictability of unmanaged Ethernet communication in the absence of queue scheduling or transmission control. By contrast, introducing a TSN switch into the bare-metal path significantly reduced both metrics to below 6,000µs, validating the effectiveness of hardware-level Time-Aware Scheduling (TAS) even without host-side intervention. Scenarios involving the custom CNI, with and without the TSN switch, demonstrated moderate improvements but failed to consistently meet deterministic expectations—highlighting the limitations of software-only traffic shaping in containerized environments. Containerized setups using Docker and macvlan also displayed elevated jitter, suggesting that virtualization layers and interface attachment mechanisms contribute to timing instability. As anticipated, the Flannel CNI—which relies on overlay networking and lacks support for real-time traffic control—performed poorly, confirming its unsuitability for TSN-style workloads. Introducing background traffic using iperf3 imposed substantial stress across all configurations. In particular, 1ms and 2ms cyclic transmission scenarios experienced a sharp increase in latency and jitter, in some cases exceeding 60,000µs. While this exposed the system's limited ability to isolate high-priority traffic under contention, it also confirmed the realism and sensitivity of the test environment. These findings prompted the integration of skbedit filters and explicit VLAN PCP tagging within the CNI plugin to enforce traffic prioritization at the kernel level and ensure correct classification in both host and switch-side queuing hierarchies.

6.4 Scenario 1: Baseline VLAN Priority Communication (Without Time-Aware Scheduling)

In the first validation scenario, UDP traffic was transmitted between hosts with VLAN PCP tagging applied to differentiate traffic classes. However, neither the host nor the switch enforced time-aware scheduling. The aim was to observe whether basic priority tagging alone could influence packet delay characteristics under real-world conditions.

6.4.1 Results

Without Background Load (iperf)

- 1 ms cycle: Maximum latency $\approx$ 35,822 μs; jitter $\approx$ 35,677 μs
- 2 ms cycle: Maximum latency $\approx$ 42,709 μs; jitter $\approx$ 42,346 μs
- 4 ms cycle: Maximum latency $\approx$ 10,729 μs; jitter $\approx$ 10,448 μs

With Background Load (iperf)

- 1 ms cycle: Maximum latency $\approx$ 96,995 μs; jitter $\approx$ 92,896 μs
- 2 ms cycle: Maximum latency $\approx$ 189,999 μs; jitter $\approx$ 180,871 μs

6.4.2 Interpretation

Although VLAN PCP markings were applied, no strict transmission ordering was enforced. Consequently, prioritized packets were still subject to queuing delays and contention, especially under background traffic conditions. While there was some observable differentiation in delay between priority and non-priority packets, the absence of gate-based scheduling meant that these priorities could not guarantee deterministic behavior. Thus, VLAN PCP tagging alone provided relative improvement but did not ensure absolute bounded latency or eliminate jitter spikes under network congestion.

6.5 Scenario 2: Switch-Level Time-Aware Scheduling (TAS) Enabled

In the second validation scenario, the previous setup was extended by enabling Time-Aware Scheduling (TAS) at the TSN switches through Gate Control List (GCL) configurations. Hosts continued to transmit packets based on VLAN PCP tagging, but the switches now enforced strict transmission windows aligned with time cycles.

6.5.1 Results

Without Background Load (iperf)

- 2 ms cycle: Maximum latency $\approx$ 40,486 μs; jitter $\approx$ 40,187 μs
- 4 ms cycle: Maximum latency $\approx$ 9,017 μs; jitter $\approx$ 8,789 μs

With Background Load (iperf)

- **Standard TAS GCL (Q7, 100 μs cycle):**
  - Maximum latency $\approx$ 98,673 μs; jitter $\approx$ 93,101 μs
- **TAS GCL (512B SDU, 1 ms cycle):**
  - Maximum latency $\approx$ 9,156–11,309 μs; jitter $\approx$ 5,351–7,134 μs
- **TAS GCL (1024/512B SDU, mixed queues):**
  - Maximum latency $\approx$ 10,017–11,149 μs; jitter $\approx$ 4,940–9,090 μs
- **TAS GCL (1536B SDU, Q7 queue):**
  - Maximum latency $\approx$ 98,674 μs; jitter $\approx$ 93,191 μs

6.5.2 Interpretation

Activating TAS at the switch level directly improved latency and jitter distribution compared to Scenario 1. The switch enforced deterministic forwarding based on the VLAN PCP priority, offering better handling of high-priority traffic. In particular, packets marked with PCP 7 (highest priority) were consistently forwarded within reserved transmission windows, resulting in substantially tighter latency bounds at relaxed cycle times (e.g., 4 ms cycle). However, under tighter cycles (1 ms) and larger SDU sizes, background congestion still caused some transmission misalignments, particularly without synchronized host-side control. This underscores that while VLAN PCP tagging combined with switch-side TAS enhances prioritization and bounded transmission, full real-time compliance still requires time-aware transmission scheduling at the sender side as well. Thus, Scenario 2 builds upon Scenario 1 by enforcing VLAN priorities within GCL-defined transmission windows at the switch, leading to visible improvements in deterministic behavior, although not yet sufficient for strict sub-millisecond guarantees.

6.5.3 Scenario 3: End-to-End Time-Aware Scheduling

This final scenario tested the complete integration of TAS as discussed chapter 5.

Result: Due to time constraints and logistical limitations, complete quantitative measurements—such as latency, jitter, and percentile analysis—were not conducted in this scenario. Instead, qualitative validation was performed to assess whether the scheduling and classification logic were correctly applied across the stack using tools such as Wireshark, ping, and system-level logs.

Observed Behavior:

- `skbedit` filters successfully mapped high-priority packets (PCP=7) to SKB priority class 3, and best-effort traffic (PCP=1) to class 0.
- `tc -s filter show` output confirmed increasing packet counters in the respective priority queues, indicating correct queue mapping at the host level.
- `taprio` scheduling was successfully applied to the TSN NIC interfaces, as verified using `tc qdisc show dev <interface>`.

- The TSN switch was synchronized to the host via gPTP, and GCLs were confirmed to be active using `netopeer2-cli` and interface counters.
- However, VLAN prioritization did not fully propagate across the TSN switch. While intra-host prioritization and scheduling were confirmed, end-to-end differentiation between high- and low-priority traffic was not consistently observed. This issue was likely caused by missing or improperly applied VLAN tags at the pod interface, or by incorrect PCP propagation through the switch ports.

6.5.4 Detailed Analysis of last three Results

Initial trials using TSN-enabled configurations revealed substantial differences in latency patterns when varying the TSN cycle duration. In configurations using 4ms cycles under no additional background load, the maximum observed latency dropped below 11ms. At first glance, this may seem counterintuitive given that longer cycle durations generally introduce additional buffering delay. However, this behavior is consistent with expectations in a lightly loaded network: a longer gate cycle provides more tolerance for timing misalignment, especially when synchronization or packet classification is suboptimal. However, this improvement does not necessarily imply an optimized deterministic behavior. Rather, the lowered latency observed in 4ms cycles likely stems from the relaxed transmission window allowing packets to pass through gates more often—even if they arrive late. Thus, while latency values improved, deterministic behavior in terms of tightly controlled gate openings was not necessarily validated. When background IPerf traffic was introduced to simulate high network load, the system's response was not uniformly deterministic. In some cases (e.g., 1ms cycle with mixed traffic), the latency increased up to 190ms, which is far beyond the intended transmission schedule. This suggests that either (1) gate control was ineffective, or (2) traffic classification was unsuccessful. Upon closer inspection, both of these turned out to be contributing factors. Specifically, Wireshark captures and system logs revealed that although taprio scheduling and SKB-priority settings were applied via tc and skbedit, VLAN PCP tagging failed silently in many trials. The outgoing packets, as captured on egress interfaces and mirrored ports, either lacked VLAN tags entirely or carried a default PCP of 0. As a result, TSN switches, which rely on PCP to map traffic to queues and corresponding GCLs, treated all traffic as best-effort. This meant that even traffic meant to be prioritized (e.g., PCP=7) did not benefit from the deterministic treatment expected under IEEE 802.1Qbv. This phenomenon directly compromised the intended behavior of GCL-based scheduling and validates why prioritized and non-prioritized traffic experienced nearly identical latency under congestion. In the third scenario—where the sender was outside Kubernetes and the receiver resided within a Kubernetes pod—TSN prioritization was entirely undermined. Despite correct SKB-priority stamping and taprio configuration on the pod's interface, VLAN headers were either stripped or never injected. This led to a systemic breakdown in queue mapping at the switch level. All frames were received with PCP=0, rendering the GCL ineffective. Consequently, traffic arrived in the correct window only by chance, not by enforcement. This outcome reaffirms that deterministic TSN behavior depends not only on timing and scheduling, but also critically on packet marking integrity. Failure to ensure PCP propagation across veth pairs and physical interfaces resulted in loss of prioritization semantics and a return to best-effort transmission behavior. To quantify the benefit of TSN in the absence of classification errors, early

test scenarios without IPerf traffic and with VLAN tagging enabled (where successful) offer a baseline. Under these ideal conditions, 2ms and 1ms cycles achieved bounded latency values between 35–40ms and 90–100ms respectively. Although not perfect, this range was relatively stable, suggesting that when packet marking and gate control worked in tandem, the TSN architecture did enforce traffic separation. However, even in these trials, latency frequently exceeded the configured cycle time. From a TSN perspective, this is undesirable. It suggests transmission outside the gate-open window, likely caused by drift between the system clock and PHC, or misalignment between taprio schedules and switch GCLs. This hypothesis is supported by phc2sys logs, which were inconsistent across test runs and occasionally indicated synchronization errors exceeding 5µs. Since taprio is sensitive to system clock drift, such misalignment leads to gate-miss transmission.

Chapter 7

Discussion

7.1 Framwork Design Discussion

The framework developed in this thesis in chapter 4 and 5. The primary objective was to enhance Kubernetes' networking stack in a modular, standards-compliant manner, suitable for industrial applications requiring bounded latency, minimal jitter, and synchronized traffic scheduling, without altering the fundamental operation of Kubernetes itself.

One notable strength of this architecture lies in its rigorous commitment to modularity. Each key component—the network plugin, synchronization stack, traffic controller, and switch configuration utility—was individually developed and integrated through open and standardized interfaces. This modularity facilitated flexible development, simplified independent testing, and enabled straightforward extensibility. Specifically, the custom CNI plugin, implemented in Bash, leveraged the Linux tc and the taprio queuing discipline to configure deterministic traffic behaviors at the pod level. This plugin dynamically managed static IP addresses, MAC addresses, VLAN tagging, and priority classes during pod initialization. Despite being scripted in Bash, this approach provided transparency and ease of debugging, avoiding complex kernel bypass solutions such as DPDK or eBPF.

Leveraging the Kubernetes-native Multus meta-plugin allowed each pod to simultaneously manage dual network interfaces: a dedicated TSN interface (eth0) configured deterministically via the custom Bash CNI, and a secondary macvlan interface (net1) for non-critical, best-effort communication. The deliberate selection of macvlan for best-effort flows effectively bypassed Kubernetes overlay networking, thus minimizing latency and isolating deterministic flows from general cluster traffic, which was critical for maintaining low jitter.

A significant architectural decision was the use of a CNC implemented in Python. This CNC communicated with TSN-capable switches using NETCONF sessions, applying IEEE 802.1Qbv-compliant GCL configurations. Employing standard YANG data models and open communication protocols ensured interoperability and vendor neutrality, aligning switch scheduling behaviors precisely with the pod-level transmission schedules. While this CNC implementation supported static scheduling configurations effectively, future enhancements would benefit from incorporating dynamic and runtime-adjustable scheduling.

Time synchronization across the Kubernetes nodes and network switches was realized using the gPTP, compliant with IEEE 802.1AS, through LinuxPTP tools. The

tools ptp4l and phc2sys facilitated hardware and system clock synchronization, respectively. A shared, synchronized time base was crucial for ensuring deterministic scheduling consistency throughout the system. However, maintaining robust synchronization under heavy network load introduced notable challenges, particularly highlighting areas for further improvement in synchronization stability.

The reproducibility of the framework was prioritized through version-controlled management of YAML manifests, CNC scripts, synchronization tools, and measurement procedures. These structured resources were designed to be executable in isolated environments, ensuring experiments' repeatability and ease of validation. Moreover, compatibility with standard Linux distributions and standard Kubernetes components enhanced accessibility for replication in both academic and industrial contexts.

Despite these strengths, the design revealed certain limitations. Primarily, the reliance on static configuration files for scheduling logic, pod IP addresses, and queue mappings restricted adaptability to dynamically changing environments, where workloads frequently scale or undergo lifecycle changes. Additionally, K8s inherent scheduling mechanisms lack awareness of real-time constraints, necessitating manual interventions for pod placement and CPU affinity configurations—pointing to a significant gap in K8s native capabilities regarding real-time workload management.

Furthermore, the absence of comprehensive feedback mechanisms from the TSN interfaces posed challenges in real-time diagnostics. Tools such as tc and Wireshark facilitated packet-level inspection; however, the architecture lacked sophisticated runtime introspection for queue states, gate timing accuracy, or detailed packet drop analyses. This limited observability indicates a critical area for future enhancement.

Lastly, the developed framework successfully demonstrates a practical, modular approach for integrating TSN within K8s without modifying its core orchestration model. It effectively bridges cloud-native principles with industrial-grade deterministic networking requirements using established open standards, leveraging existing Linux networking capabilities and container orchestration mechanisms. While this initial implementation validates the feasibility of Kubernetes-native deterministic communications, further development focusing on dynamic scheduling, real-time observability, and improved synchronization robustness is necessary for deployment at production scales.

7.2 Testbed Reflection

The experimental testbed detailed in Chapters 4 and 5 provided the essential infrastructure for systematically evaluating the integration of Kubernetes and TSN. Although the original validation was structured around three defined scenarios, iterative experimental observations necessitated further refinements, ultimately enriching the assessment by uncovering previously unforeseen challenges and subtleties.

While the testbed architecture performed satisfactorily under controlled, idealized configurations, the measurement procedures revealed several intricate challenges. Notably, these difficulties included sustaining clock synchronization accuracy during network congestion, aligning host-to-switch transmission windows precisely, and achieving reliable traffic prioritization across multiple networking layers in containerized environments.

A significant initial challenge arose from the early reliance on ICMP-based diagnostic tools, such as ping. These tools, inherently lacking support for VLAN tagging and queuing prioritization mechanisms, provided misleading results and obscured underlying timing irregularities. This issue underscored the necessity of adopting more sophisticated, application-layer measurement approaches capable of accurately characterizing deterministic communication. Consequently, the evaluation strategy transitioned to a custom-developed Go-based UDP traffic generator, facilitating high-resolution timestamping of packet transmissions and receptions. This tool proved essential in rigorously quantifying latency and jitter characteristics, yielding data that effectively reflected the true deterministic capabilities of the network stack.

Another critical challenge involved configuring the Linux traffic control mechanism, specifically the taprio queuing discipline, on TSN-capable NICs. Correct operation of taprio required meticulous synchronization of scheduling parameters, such as base-time alignment, cycle durations, and explicit queue-to-priority mappings between hosts and switches. Minor misconfigurations or timing deviations led to unpredictable transmission behaviors. Moreover, under conditions involving large packets or intensive background network loads introduced by tools like iperf3, packets occasionally exceeded their designated transmission windows, further complicating accurate latency measurement. The absence of built-in introspection or diagnostic feedback mechanisms in the Linux kernel's TSN implementation required reliance on indirect verification methods, such as Wireshark packet captures and offline statistical analyses, to detect and correct scheduling misalignments.

Clock synchronization, a cornerstone of deterministic TSN communication, was successfully implemented using gPTP tools—namely ptp4l and phc2sys. While these tools demonstrated the ability to achieve sub-microsecond accuracy under stable conditions, their synchronization precision notably deteriorated under conditions of high network traffic generated by iperf3 stress tests. Observed synchronization drift during periods of intense network load adversely impacted gate control accuracy, disrupting predictable transmission patterns and thus compromising deterministic network performance. These results highlight the critical requirement for robust synchronization protocols that maintain high accuracy and reliability, even under stress conditions typical of industrial environments.

Additionally, complexities associated with network interface management emerged. Utilizing Multus CNI combined with macvlan facilitated effective isolation between deterministic and best-effort traffic flows. However, these configurations occasionally introduced operational anomalies such as race conditions or duplicated network interfaces, particularly during pod initialization stages involving static IP allocations. These complications, while manageable in experimental settings, suggest a clear necessity for improved error handling, interface management logic, and tighter integration within Kubernetes production deployments to ensure consistent performance and reliability.

In summary, the testbed provided a robust platform for extensive scenario validation and offered deep insights into practical constraints and complexities encountered in real-world TSN and Kubernetes integrations. The identified limitations in synchronization stability, queue scheduling alignment, interface management, and diagnostic visibility are essential findings. These observations significantly contribute to the

practical understanding of deploying deterministic containerized systems, outlining clear pathways for future enhancement and highlighting the need for improved tooling, feedback mechanisms, and dynamic configuration capabilities.

7.3 Measurement Reflection

The measurement phase revealed important insights into the practical behavior and limitations of the Kubernetes–TSN integration framework. While the architecture successfully demonstrated time-aware communication under certain configurations, the actual results—particularly in terms of latency and jitter—highlighted several technical gaps between theoretical determinism and real-world performance.

One of the most consistent patterns observed across all test scenarios was the significant variability in both latency and jitter under background network load. As shown in Table 6.2, best-case results (e.g., bare-metal via TSN switch) achieved sub-6 millisecond latency, while worst-case configurations, particularly under high contention or without proper scheduling, exceeded 90 milliseconds. These results underscored the fact that VLAN PCP tagging alone is insufficient to ensure predictable delivery in containerized environments. Without enforced scheduling (as in Scenario 1), priority information may be present but is often ignored or diluted in the forwarding pipeline, especially when standard Linux queuing mechanisms or switch-level defaults are used.

In Scenario 2, the introduction of TAS at the switch level led to noticeable improvement. GCLs helped isolate high-priority traffic into reserved transmission windows, significantly reducing jitter. However, tight cycles and large SDU payloads still showed performance degradation under stress. This suggested that switch-side TAS alone cannot fully shield critical traffic from interference, particularly when transmission initiation remains unsynchronized at the sender (host) level.

Scenario 3 provided qualitative verification of full TAS integration, including taprio scheduling at the host and synchronized GCLs at the switch. Even though numerical latency/jitter measurements were not collected due to time constraints, diagnostic tools such as tc -s filter show, netopeer2-cli, and Wireshark validated the correct setup of queuing and scheduling mechanisms. Traffic marked with PCP 7 was correctly routed through high-priority queues, and increasing counters for SKB priority classes confirmed that packets were being enqueued as expected.

Despite this alignment, one critical limitation emerged: VLAN tagging was inconsistently handled by the NIC. This led to a breakdown in end-to-end prioritization—while traffic was prioritized inside the host and classified correctly at the software layer, the switch did not consistently differentiate between high and low priority frames. The root cause likely lay in hardware-level inconsistencies, incorrect egress tagging, or misalignment between skbedit behavior and NIC offload features.

The measurement phase also highlighted the fragility of synchronization. Although LinuxPTP tools enabled gPTP-based alignment with microsecond precision under normal conditions, stress tests (e.g., iperf3 load) occasionally introduced clock drift, leading to gate misalignment. This further impacted cycle timing and added variation to transmission intervals. In a TSN system, such timing drift directly compromises deterministic behavior—even when the scheduling logic is correct.

From a tooling perspective, the decision to rely on a Go-based measurement utility proved effective. The application enabled millisecond-level cyclic UDP transmission, in-pod timestamping, and structured CSV output. Combined with Python-based analysis, this allowed for detailed time-series plots and statistical aggregation. However, validation also relied heavily on Wireshark for hardware timestamping, which was not always straightforward due to NIC driver limitations and VLAN visibility issues.

7.4 Observed Limitations and Insights

While the integrated Kubernetes–TSN system developed in this thesis successfully demonstrated deterministic communication capabilities using IEEE 802.1Qbv and time-synchronized traffic scheduling, several limitations emerged during implementation. These reflect technical and architectural boundaries affecting scalability, visibility, and operational robustness. The system exhibited high sensitivity to clock synchronization accuracy. Although ptp4l consistently synchronized the hardware clock at the NIC level, phc2sys—responsible for aligning the system clock—often showed instability, particularly under load. Since the taprio scheduler relies on the system clock, even microsecond-level drift led to desynchronized gate control windows. This misalignment occasionally resulted in traffic being transmitted outside the intended time slot, which compromises the deterministic behavior of the system. Improved tuning of phc2sys, use of real-time kernels, and CPU pinning may offer mitigation, but the fragility of software-based synchronization remains a limiting factor. Another constraint was the static nature of the switch scheduling configuration. The CNC implemented for this work used predefined GCLs to configure switches via NETCONF and YANG. However, the complementary CUC component was not implemented, which dynamically negotiates bandwidth and traffic flows. Without CUC, all stream characteristics had to be known in advance and manually incorporated into the CNC, making it challenging to accommodate changing workloads or dynamic pod orchestration. This limited the framework's flexibility and highlighted the need for runtime stream registration mechanisms in future iterations. The testbed also lacked sufficient visibility into hardware queue behavior. While VLAN PCP tagging and taprio scheduling were correctly applied using tc filters and skbedit, it was difficult to confirm whether the NIC or switch prioritized traffic as expected. Tools such as Wireshark and tcpdump provided partial evidence but did not confirm queue-level treatment or gate timing. Packets from different traffic classes appeared nearly simultaneously, raising questions about firmware behavior or incomplete isolation in the queuing pipeline. Without access to vendor-specific introspection tools or telemetry, validating deterministic enforcement at the hardware level remained inconclusive. Resource allocation and traffic scheduling configurations were statically defined. IP addresses and MAC assignments for TSN-enabled pods were hardcoded using Multus and NAD files, and traffic class mappings were manually assigned. Although this approach was manageable for a controlled test environment, it is not sustainable for large-scale or dynamic workloads. The lack of Kubernetes-native integration for traffic-aware admission control, dynamic IP allocation, or traffic class enforcement exposes a significant operational bottleneck. Bare-metal deployment further introduced maintenance complexity. Unlike virtual environments, misconfiguration in the testbed often led to full system resets, BIOS reconfiguration, or PTP drift that required manual intervention. The absence of orchestration support for TSN-specific lifecycle

events made automation difficult, and minor errors in GCL syntax or pod annotation sometimes caused critical failures. These operational challenges underscore the importance of tooling abstractions such as TSN-aware Helm charts or configuration operators to support reproducibility and scalability. Early experimentation with traffic prioritization also revealed limitations in tag visibility. VLAN PCP markers inserted via TC and egress-QoS-map were not consistently observed in packet captures. It was unclear whether NIC firmware removed tags before transmission or whether veth devices suppressed them. This ambiguity complicated the validation of priority-based scheduling and made it challenging to link observed jitter or delay patterns with configuration logic. Finally, the most systemic limitation was that Kubernetes is not inherently designed for real-time workloads. Core Kubernetes components lack awareness of timing constraints, deadline enforcement, or latency sensitivity. TSN features had to be integrated through external mechanisms: Multus for multi-networking, a custom CNI plugin for taprio scheduling, and a separate CNC for switch orchestration. This disjointed integration required precise coordination and remained fragile in the face of pod churn, system restarts, or configuration drift. For TSN to be deployable at scale within Kubernetes, native primitives for deterministic networking must be introduced at the orchestration level. Despite these challenges, the system validated the feasibility of integrating TSN within a containerized environment and highlighted critical areas for future enhancement. These include improving synchronization robustness, enabling dynamic stream negotiation, enhancing queue observability, and embedding real-time semantics directly into Kubernetes' resource model.

Chapter 8

Conclusion

8.1 Conclusion

This thesis investigated the feasibility of enabling deterministic, time-sensitive communication within a Kubernetes cluster by integrating core TSN mechanisms into the container networking stack. The architecture combined IEEE 802.1Qbv for time-aware scheduling, IEEE 802.1AS for synchronization, and NETCONF/YANG for centralized switch configuration, implemented through a custom Bash-based CNI plugin and a Python-based CNC.

Through scenario-based experiments, the system demonstrated host-level deterministic scheduling using taprio, skbedit, gPTP synchronization, and VLAN tagging to prioritize traffic. However, practical constraints—particularly inconsistent VLAN PCP propagation across virtual and physical interfaces—prevented reliable prioritization at the switch level. As a result, even correctly marked packets were often processed as best-effort, undermining end-to-end determinism under load.

Despite these limitations, the thesis contributes a reproducible and vendor-neutral framework for partially implementing TSN principles within Kubernetes. It validated that real-time scheduling is possible without relying on DPDK or kernel modifications, and it surfaced key integration challenges such as the lack of native TSN support in Kubernetes, limited visibility into TSN runtime behavior, and fragility in synchronization under stress.

This work establishes a solid foundation for future TSN-Kubernetes research, especially in areas like dynamic stream negotiation (e.g., CUC integration), enhanced monitoring, and automated adaptation to real-time constraints. It offers actionable insights for system architects seeking to converge IT and OT domains in Industry 4.0 deployments using open-source, cloud-native infrastructure.

8.2 Future Work

While this thesis has demonstrated the potential for integrating deterministic TSN capabilities within Kubernetes-managed environments, several areas of exploration remain for future research. The following points highlight the most promising and necessary enhancements to advance the integration of Kubernetes with TSN:

- The current implementation leverages a CNC for static GCL management. Introducing a Centralized User Configuration (CUC), as specified by IEEE 802.1Qcc, would allow dynamic negotiation of network resources. Future work

should focus on developing a CUC module integrated with Kubernetes' scheduler, enabling real-time updates to traffic scheduling based on changing network conditions, pod lifecycle events, and workload demands.

- The experiments revealed issues related to VLAN Priority Code Point (PCP) propagation, significantly impacting traffic prioritization. Future investigations should concentrate on enhancing the Container Network Interface (CNI) plugin or implementing kernel-level modifications to guarantee reliable VLAN tagging and PCP handling across virtual-to-physical network boundaries. This would ensure end-to-end deterministic behavior critical for real-time industrial communication.
- The existing system, using Bash-based CNI scripting and Python CNC implementations, served well for prototyping purposes but demonstrated limitations in scalability and performance. Future efforts should explore optimized and compiled languages (e.g., Rust or Go) for critical components, allowing higher throughput, lower latency, and improved reliability under high-load industrial scenarios.
- The current testbed demonstrated deterministic behavior under controlled and simulated stress conditions. Future studies should extend validation efforts to larger-scale deployments, diverse hardware configurations, and realistic industrial workloads. This would assess the robustness, resilience, and stability of Kubernetes-TSN integration under real-world operational scenarios, including transient network failures and varying load patterns.
- Current monitoring approaches were largely manual and limited in granularity. Future work should aim at developing comprehensive monitoring tools integrated directly within Kubernetes, enabling real-time visibility into TSN parameters (e.g., gate control state, queue utilization, synchronization accuracy). Advanced analytics and automated alerting systems would significantly enhance operational transparency and proactive fault mitigation capabilities.

By addressing these outlined directions, subsequent research efforts can further refine the integration of deterministic networking principles into Kubernetes, significantly bridging the gap between cloud-native orchestration and real-time industrial communication requirements.

References

[Abo18] D. Aboubacar. *Device and method of transmitting synchronization messages in a computer network*. Patent or technical document. 2018.

[AG23] Armir Bujari Luca Foschini Andrea Garbugli Lorenzo Rosa. "KuberneTSN: a Deterministic Overlay Network for Time-Sensitive Containerized Environments". In: *arXiv* (2023). URL: https://arxiv.org/pdf/2302.08398.

[ANR25] M. Sri Rama Lakshmi Reddy Yamjala Arjun Sagar Nallagondla Jyothi S. Shilpa Angotu Nageswara Rao Adilakshmi Velivela. "Machine Learning and Artificial Intelligence in IoT: Integration Techniques and Applications". In: *SpringerLink* (2025). URL: https://link.springer.com/chapter/10.1007/978-981-97-8533-9_5.

[Ass25] IEEE Standards Association. "IEEE 802.1Qbv: Time-Aware Shaping for Deterministic Networking". In: *IEEE 802.1 Working Group* (2025). URL: https://standards.ieee.org/standard/802_1Qbv-2015.html.

[Aut23a] Kubernetes Authors. *Container Network Interface (CNI) Plugins*. https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/. Accessed: 2025-04-30. 2023.

[Aut23b] Kubernetes Authors. *Scheduling Framework*. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/. Accessed: 2025-04-30. 2023.

[BCL19] D. Bogdanovic, B. Claise, and A. Lindem. *Network Configuration Protocol (NETCONF) and RESTCONF Integration with YANG*. https://www.ietfjournal.org/netconf-and-restconf-integration-with-yang/. Accessed: 2025-04-30. 2019.

[BHS+18] Siwar Ben Hadj Said et al. "SDN-Based Configuration for IEEE 802.1 TSN". In: *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. 2018.

[Bur+16] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. "Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade". In: *Communications of the ACM* 59.5 (2016), pp. 50–57. DOI: 10.1145/2898442.2898444.

[Cal23] Project Calico. *Calico for Kubernetes*. https://docs.tigera.io/. Accessed: 2025-04-30. 2023.

[Cis25] Cisco. "Understanding IT and OT Differences". In: *Cisco Blog* (2025). URL: https://www.cisco.com.

[Clo25] Google Cloud. "Plan for scalability in Google Kubernetes Engine (GKE)". In: *Google Cloud Documentation* (2025). URL: https://cloud.google.com/kubernetes-engine/docs/concepts/planning-scalability.

[Cor14] Jonathan Corbet. *Time synchronization in Linux: ptp4l and phc2sys*. https://lwn.net/Articles/572742/. Accessed: 2025-04-30. 2014.

[Cor23a] CoreOS. *Flannel Documentation*. https://github.com/flannel-io/flannel. Accessed: 2025-04-30. 2023.

[Cor23b] Intel Corporation. *Multus CNI Plugin*. https://github.com/k8snetworkplumbingwg/multus-cni. Accessed: 2025-04-30. 2023.

[Dav24] Deep Manishkumar Dave. "Industry 4.0: The Fourth Industrial Revolution". In: *IoT Business News* (2024). URL: https://iotbusinessnews.com/2024/01/30/56566-industry-4-0-the-fourth-industrial-revolution/.

[Del25] Delta T Systems. *What is Industry 4.0?* Accessed: 2025-05-04. 2025. URL: https://deltatsys.com/industry-expertise/what-is-industry-40/.

[Doc23] Inc. Docker. *Macvlan network driver*. https://docs.docker.com/network/macvlan/. Accessed: 2025-04-30. 2023.

[Doc25] Linux Kernel Documentation. "Taprio: Time-Aware Priority Scheduling in TSN". In: *Linux Kernel Manual* (2025). URL: https://www.man7.org/linux/man-pages/man8/tc-taprio.8.html.

[DS+20] Marcelo C. Da Silva, Ali Nejatian, Samuel Kounev, and Ricardo S. De Oliveira. "Performance Evaluation of Time-Sensitive Networking for Industrial Applications in Kubernetes-based Edge Clouds". In: *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE. 2020, pp. 104–113. DOI: 10.1109/ISORC49007.2020.00026.

[Eng25] Control Engineering. "How new automation technologies shape industrial networking trends". In: *Control Engineering* (2025). URL: https://www.controleng.com/how-new-automation-technologies-shape-industrial-networking-trends/.

[EPR20] Reto Eidenbenz, Yvonne-Anne Pignolet, and Andreas Ryser. "Latency-Aware Industrial Fog Application Orchestration with Kubernetes". In: *Proceedings of the Future Mobility Computing Conference (FMEC)*. IEEE, 2020, pp. 164–171.

[Fou23] Cloud Native Computing Foundation. "Kubernetes Project Journey Report". In: *CNCF* (2023). URL: https://www.cncf.io/reports/kubernetes-project-journey-report/.

[Gar+23] Andrea Garbugli, Lorenzo Rosa, Armir Bujari, and Luca Foschini. *KuberneTSN: a Deterministic Overlay Network for Time-Sensitive Containerized Environments*. 2023. arXiv: 2304.06652 [cs.NI]. URL: https://arxiv.org/abs/2304.06652.

[GP23] Panagiotis Papadimitriou George Papathanail Lefteris Mamatas. "Towards the Integration of TAPRIO-Based Scheduling With Centralized TSN Control". In: *IFIP Networking Conference* (2023). URL: ^2^.

[Gro25a] Kubernetes Network Plumbing Working Group. "Multus CNI: Enabling Multi-Homed Pods in Kubernetes". In: *GitHub Documentation* (2025). URL: ^3^.

[Gro25b] Kubernetes Scalability Special Interest Group. "Kubernetes Scalability and Performance Goals". In: *Kubernetes Community* (2025). URL: https://github.com/kubernetes/community/blob/master/sig-scalability/README.md.

[HBB17] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: Up and Running: Dive into the Future of Infrastructure*. O'Reilly Media, Inc., 2017.

[IBM25] IBM. "What is Industry 4.0?" In: *IBM Blog* (2025). URL: https://www.ibm.com/think/topics/industry-4-0.

[IEE16a] IEEE. *IEEE Standard for Ethernet Amendment 5: Specification and Management Parameters for Interspersing Express Traffic*. IEEE Std 802.3br-2016. 2016.

[IEE16b] IEEE. *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks—Amendment 25: Enhancements for Scheduled Traffic*. https://doi.org/10.1109/IEEESTD.2016.7553521. IEEE Std 802.1Qbv-2015. 2016.

[IEE16c] IEEE. *IEEE Standard for Local and Metropolitan Area Networks—Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks—Amendment 26: Frame Preemption*. IEEE Std 802.1Qbu-2016. 2016.

[IEE17] IEEE. *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks—Amendment 28: Per-Stream Filtering and Policing*. https://doi.org/10.1109/IEEESTD.2017.8057047. IEEE Std 802.1Qci-2017. 2017.

[IEE18a] IEEE. *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks—Amendment 33: Stream Reservation Protocol (SRP) Enhancements and Performance Improvements*. https://doi.org/10.1109/IEEESTD.2018.8403927. IEEE Std 802.1Qcc-2018. 2018.

[IEE18b] IEEE. *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks*. https://doi.org/10.1109/IEEESTD.2018.8403922. IEEE Std 802.1Q-2018. 2018.

[IEE19] IEEE. *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. https://standards.ieee.org/standard/1588-2019.html. IEEE Std 1588-2019. 2019.

[IEE20] IEEE. *IEEE Standard for Local and Metropolitan Area Networks – Timing and Synchronization for Time-Sensitive Applications*. https://doi.org/10.1109/IEEESTD.2020.9369850. IEEE Std 802.1AS-2020. 2020.

[IEE24] IEEE 802.1 Working Group. *IEEE 802.1 Time-Sensitive Networking (TSN) Standards Overview*. https://1.ieee802.org/tsn/. Accessed: 2025-04-29. 2024.

[KHM20] Hannah Kelsey, John Henthorn, and Mahmoud Mahmoud. *The Illustrated Children's Guide to Kubernetes Networking*. https://learnk8s.io/kubernetes-networking. Accessed: 2025-04-30. 2020.

[Kir+23] Dennis Kirsch et al. "KuberneTSN: Enabling TSN-based Real-time Communications in Kubernetes". In: *arXiv preprint arXiv:2302.08398* (2023).

[Kub25] Kubernetes Authors. *Kubernetes Official Documentation*. Accessed: 2025-04-29. 2025. URL: https://kubernetes.io/docs.

[Mic25] Microsoft. "Unlocking business value: The power of IT and OT convergence". In: *Microsoft Industry Blog* (2025). URL: https://techcommunity.microsoft.com/blog/microsoftindustryblog/unlocking-business-value-the-power-of-it-and-ot-convergence/4393493.

[MK23] Jannis Ohms Oleksandr Shulha Olaf Gebauer Manish Kumar Martin Boehm. "Evaluation of the Time-Aware Priority Queueing Discipline with Regard to Time-Sensitive Networking in Particular IEEE 802.1Qbv". In: *Open Data University Halle* (2023). URL: ^3^.

[Nas+19] Amine Nasrallah et al. "Ultra-Low Latency (ULL) Networks: The IEEE TSN and IETF DetNet Standards and Related 5G ULL Research". In: *IEEE Communications Surveys & Tutorials* 21.1 (2019), pp. 88–145.

[OV25] Marco Ottella Martin Serrano Tore Karlsen Terje Wahlstrøm Hans Erik Sand Meghashyam Ashwathnarayan Micaela Troglia Gamba Ovidiu Vermesan Roy Bahr. "Internet of Robotic Things Intelligent Connectivity and Platforms". In: *Frontiers in Robotics and AI* (2025). URL: https://www.frontiersin.org/journals/robotics-and-ai/articles/10.3389/frobt.2020.00104/full.

[Pla25] Plattform Industrie 4.0. *Plattform Industrie 4.0*. Accessed: 2025-04-29. 2025. URL: https://www.plattform-i40.de.

[PMP23] George Papathanail, Lefteris Mamatas, and Panagiotis Papadimitriou. "Towards the Integration of TAPRIO-Based Scheduling With Centralized TSN Control". In: *Proceedings of the IFIP Networking Conference (IFIP Networking)*. New York, NY, USA: Association for Computing Machinery, 2023. DOI: 10.1145/3695248. URL: https://doi.org/10.1145/3695248.

[PTF20] Kristofer Pister, David Thiele, and Norman Finn. "Time-Sensitive Networking: The Ethernet Evolution for Real-Time Industrial Communications". In: *IEEE Communications Standards Magazine* 4.3 (2020), pp. 22–28. DOI: 10.1109/MCOMSTD.001.1900003.

[SFS11] Keith A. Stouffer, Joseph A. Falco, and Karen A. Scarfone. *SP 800-82: Guide to Industrial Control Systems (ICS) Security: Supervisory Control and Data Acquisition (SCADA) systems, Distributed Control Systems (DCS), and other control system configurations such as Programmable Logic Controllers (PLC)*. NIST Special Publication 800-82. National Institute of Standards and Technology (NIST), 2011. DOI: 10.6028/NIST.SP.800-82. URL: https://doi.org/10.6028/NIST.SP.800-82.

[SK24] Amna Mirza Sachin Kumar Ajit Kumar Verma. "Digitalisation, Artificial Intelligence, IoT, and Industry 4.0 and Digital Society". In: *SpringerLink* (2024). URL: https://link.springer.com/chapter/10.1007/978-981-97-5656-8_3.

[SW20] Max Helm Stefan Waldhauser Benedikt Jaeger. "Time Synchronization in Time-Sensitive Networking". In: *Technical University of Munich* (2020). URL: ^2^.

[TGE19] D. Thiele, M. E. Gutiérrez, and R. Ernst. "TSN in Automotive and Industrial Applications – Status and Future Directions". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 1–10. DOI: 10.23919/DATE.2019.8714944.

[Tok21] L Toka. "Ultra-reliable and low-latency computing in the edge with Kubernetes". In: *Journal of Grid Computing* 19.3 (2021), pp. 1–23.

[Unk20] Author Unknown. "Time Synchronization in Time-Sensitive Networking". In: *Technical University of Munich* (2020). Accessed: 2025-04-30.

[WK08] J. Wittgreffe and K. Khan. "Orchestrating end-to-end network and IT resources according to application level service level agreements". In: *BT Technology Journal* 26.1 (2008), pp. 46–57. DOI: 10.1007/s10550-008-0087-3.

Declaration of Authorship

I hereby declare that this master thesis was independently composed and authored by myself.

All content and ideas drawn directly or indirectly from external sources are indicated as such. All sources and materials that have been used are referred to in this thesis.

The thesis has not been submitted to any other examining body and has not been published.

_____

Place, date

_____

Signed: Mahmuda Akter