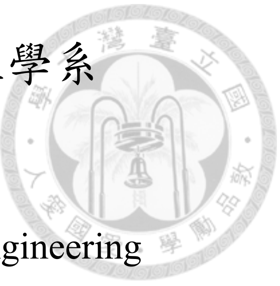
國立臺灣大學電機資訊學院資訊工程學系



碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master's Thesis

自駕車系統與時間敏感性網路的即時效能分析

Performance Profiling and Online Monitoring on Autonomous Vehicle Systems with Time-Sensitive Networking

郭泰佑

Tai-You Kuo

指導教授: 洪士灝 博士

Advisor: Shih-Hao Hung Ph.D.

中華民國 115 年 1 月

January, 2026

doi:10.6342/NTU202600098

摘要

本論文以由上而下的方法對一個完整的自駕車系統進行分析，藉由提供應用程式上層的效能指標，以及與此相關的底層系統效能資訊來分析自駕車系統的即時效能。首先，我們改進現有的 ROS 2 效能工具 CARET 使其能追蹤分散式系統，並利用改進後的工具對分散式的 Autoware 自駕車系統進行應用層級的效能分析，包括每個節點上計算與通訊所需的資源。為了進一步解析節點之間透過通訊協定所產生的交互作用，我們利用 libpcap 擷取有關數據分配服務（DDS）的網路流量，將相關的效能資訊放入時間序列資料庫，並提供數種資料庫查詢腳本，用以追蹤底層系統中通訊協定與網路架構的效能瓶頸，並且協助系統開發者找出異常發生的可能原因。最後，為了改進分散式系統的即時性，我們將時間敏感性網路（TSN）技術應用於 Autoware 系統中，並且探討其在多種應用情境負載下的效能，並與傳統的通信技術進行比較。使用簡易機器人工作負載與真實 Autoware 自駕車工作負載的評估結果顯示，TSN（特別是時間感知整形器 TAS）能在網路頻寬競爭下穩定高流量感測器數據的延遲，但同時也對控制訊息引入了週期層級的取捨。本論文所發展的效能工具、分析方法與系統改進，對於提高自駕車系統的即時效能和穩定性應具有相當的助益。

關鍵字： Autoware，數據分配服務，時間敏感網路、PCAP、分散式追蹤

doi:10.6342/NTU202600098

Abstract

This thesis employs a top-down approach to analyze autonomous vehicle systems. Providing key application performance metrics, and drilling down related underlying system performance information, we examine the real-time performance of the autonomous vehicle system. We improve the existing ROS 2 performance tool, CARET, and use the enhanced tool to perform performance analysis on a distributed autonomous vehicle system based on Autoware. To record the underlying system performance, we use libpcap to capture network traffic related to DDS, store the relevant performance information into a time-series database, and provide several database query scripts to assist users in identifying possible causes of anomalies. Finally, we apply Time-Sensitive Networking (TSN) technology to the Autoware system, evaluating its real-time performance under a specific load and comparing it with traditional communication technology. Evaluation using a simplified robotic workload and a realistic Autoware-based autonomous driving workload shows that TSN, specifically Time-Aware Shaper (TAS), stabilizes latency for bandwidth-

doi:10.6342/NTU202600098

intensive sensor data under contention, while introducing cycle-level trade-offs for control messages. These results are essential for enhancing real-time performance and stability of autonomous vehicle systems.

Keywords: Autoware, DDS, TSN, PCAP, distributed tracing

doi:10.6342/NTU202600098

Contents

Page

摘要 **i**

Abstract **ii**

Contents **iv**

List of Figures **vii**

List of Tables **ix**

Denotation **x**

Chapter 1 Introduction **1**

Chapter 2 Background **5**

2.1 SOAFEE 5

2.2 Autoware 6

2.3 ROS 2/DDS 7

2.4 Time-Sensitive Networking 8

2.4.1 Time Synchronization (IEEE 802.1AS) 8

2.4.2 Time-Aware Shaper (IEEE 802.1Qbv) 9

2.5 Related Work 9

doi:10.6342/NTU202600098

Chapter 3 Methodology **11**

3.1 Design 11

3.1.1 System Overview 11

3.1.2 Performance Implication of TSN 14

3.2 Implementation 16

3.2.1 High-level Application Performance Metrics: CARET 16

3.2.2 Low-level DDS Network Metrics: ddshark 17

3.2.3 Dissecting DDS Traffic with ddshark 19

3.2.4 Capturing Fragmented Data with ddshark 23

3.2.5 Latency Metrics Definition 24

3.2.6 Performance Data Visualization 25

Chapter 4 Evaluation **27**

4.1 Experimental Setup 27

4.1.1 Hardware and Network Topology 27

4.1.2 Software Configuration and Time Synchronization 28

4.2 Experimental Design and Scenarios 30

4.2.1 Workload and Configurations 30

4.2.2 Background Interference Models 31

4.3 Case Study I: Simple Robotic Application 32

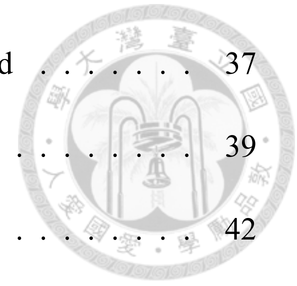
4.3.1 Analysis of Monitoring Overhead 33

4.3.2 Analysis of Large Data Message (Image) 34

4.3.3 Analysis of Small Control Message (Drive data) 36

doi:10.6342/NTU202600098

4.4 Case Study II: Realistic Autonomous Driving Workload 37
4.4.1 Analysis of LiDAR Data (Pointcloud) 39
4.4.2 Analysis of Vehicle Status data 42
4.4.3 Online Monitoring Visualization UI 43
Chapter 5 Conclusion and Future Work **45**
References **47**



doi:10.6342/NTU202600098

List of Figures

1.1 Domain and zonal architectures. 2

2.1 Architecture of SOAFEE. 6

2.2 Operating principle of Time-Aware Shaper. 9

3.1 Overview of the deployments of performance evaluation tools. 12

3.2 Overview of the setup for the monitoring framework. 13

3.3 Overview of the performance profiling workflow in CARET. 16

3.4 Overall and underlying architecture of ddshark. 18

3.5 Typical layout of an RTPS packet. 20

3.6 Capturing Fragmented Data with ddshark. 23

4.1 System setup for evaluation. 28

4.2 System setup for mapping DDS topic to TSN priority. 29

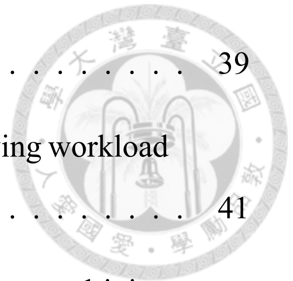
4.3 ROS 2 Node diagram for simple application. 33

4.4 Boxplot of image messages in simple application under various metric types. 35

4.5 Boxplot of drive messages in simple application under various metric types 37

4.6 The RViz Views panel of rosbag replay simulation. 38

doi:10.6342/NTU202600098



4.7 System setup for realistic autonomous driving workload. 39

4.8 Boxplot of pointcloud messages in realistic autonomous driving workload case study under various metric types. 41

4.9 Boxplot of velocity status data messages in a realistic autonomous driving workload case study under various metrics. 43

4.10 Grafana UI dashboard that monitors realistic autonomous driving workload, with x-axis as system time and y-axis as DDS message payload. . . 44

4.11 Grafana UI dashboard that monitors realistic autonomous driving workload, with the x-axis as system time and the y-axis. The y-axis of the above panel shows the latency of a complete DDS message, whereas the y-axis of the below panel shows the network bandwidth in system time. . 44

doi:10.6342/NTU202600098

List of Tables

3.1 Mapping between captured DDS traffic information and OpenTelemetry attributes. 22

4.1 Description of the computation platform used for evaluation. 29

4.2 Description of the application software used for evaluation. 29

4.3 Experimental scenarios and interference levels. 32

4.4 Description of the message sent over Ethernet in simple robotic application. 33

4.5 Statistical descriptives for overhead comparison. 34

4.6 Comparison of Theoretical vs. Measured Latency Metrics. 36

4.7 Description of the message sent over Ethernet in realistic autonomous driving workload. 40

doi:10.6342/NTU202600098

Denotation

SDV	軟體定義汽車 (Software-Defined Vehicles)
DDS	數據分配服務 (Data Distribution Service)
TSN	時間敏感網路 (Time-Sensitive Networking)
SMT	同時多執行緒 (Simultaneous Multithreading)
ECU	電子控制單元 (Electronic Control Unit)
LiDAR	光學雷達 (Light Detection and Ranging)
IMU	慣性測量單元 (Inertial Measurement Unit)
GNSS	全球導航衛星系統 (Global Navigation Satellite System)
NIC	網路介面卡 (Network Interface Controller)
DMA	直接記憶體存取 (Direct Memory Access)

doi:10.6342/NTU202600098

Chapter 1 Introduction

As the automotive industry currently shifts into the age of digital transformation, Software-Defined Vehicles (SDVs) have emerged as key technologies to enhance the functionalities of future transportation. In an SDV, software is used to determine the performance, safety features, and user experience of the vehicle. Consequently, cars are no longer just a changeless piece of hardware when leaving the factory, but an upgradeable platform that could be continually improved through innovations.

The advent of SDV marks the transition from a hardware-centric to a software-centric approach in automotive design. In this new paradigm, features and functionalities are no longer strictly coupled to the underlying hardware and components of the vehicle. In contrast, software becomes the primary driver for features and customizations. For example, if a customer subscribes to advanced functionalities related to autonomous driving, the manufacturer can simply enable this feature through over-the-air (OTA) updates. This capability significantly enables the expandability and adaptability of automotive design.

However, the transition to SDV brings challenges to the Electrical/Electronic (E/E) architecture regarding its ability to support the requirements of the software's role. In traditional E/E architectures, a vehicle typically consists of tens to a hundred Electronic

doi:10.6342/NTU202600098

Control Units (ECUs), where each ECU has its dedicated functions and connects to others on the vehicle. This architecture not only results in increased manual labor to install the wiring system but also restricts the scalability and flexibility of software functionalities.

To address this issue, the automotive industry is moving toward domain and zonal architectures [12], as shown in Figure 1.1. Domain architectures replace numerous dedicated ECUs with a smaller number of powerful domain controllers, which consolidate related functions from smaller ECUs. As a result, this simplifies software control over these grouped functionalities. Zonal architecture takes a step further by considering the physical location of each control unit, replacing multiple ECUs with a zonal ECU responsible for a specific area of the vehicle. Each zonal ECU connects to sensors based on their physical proximity and communicates with a high-performance central vehicle computer via high-speed networks. Ultimately, the Zonal E/E Architecture not only simplifies the hardware layout but also reduces the complexity of the wiring system and the weight of wiring harnesses.

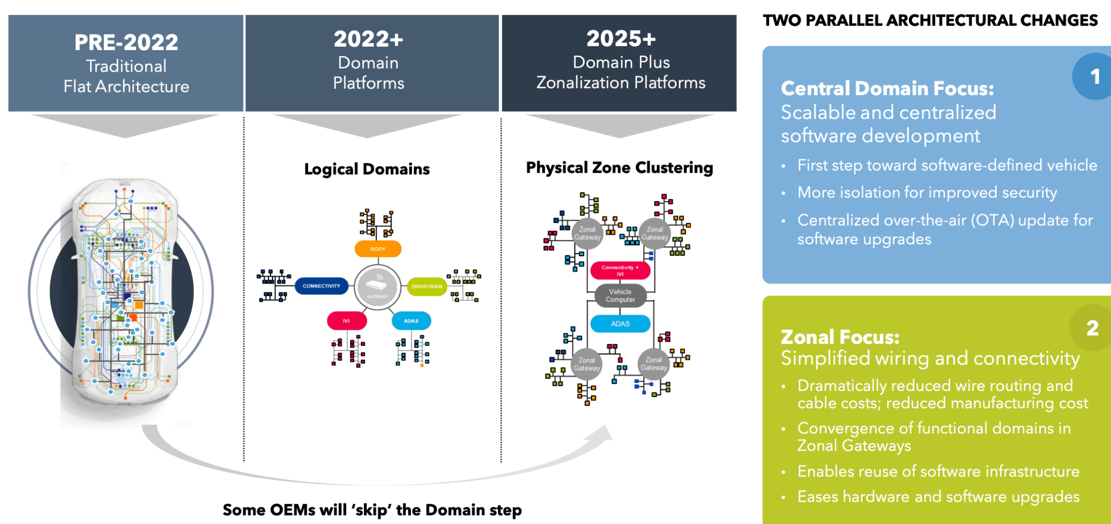
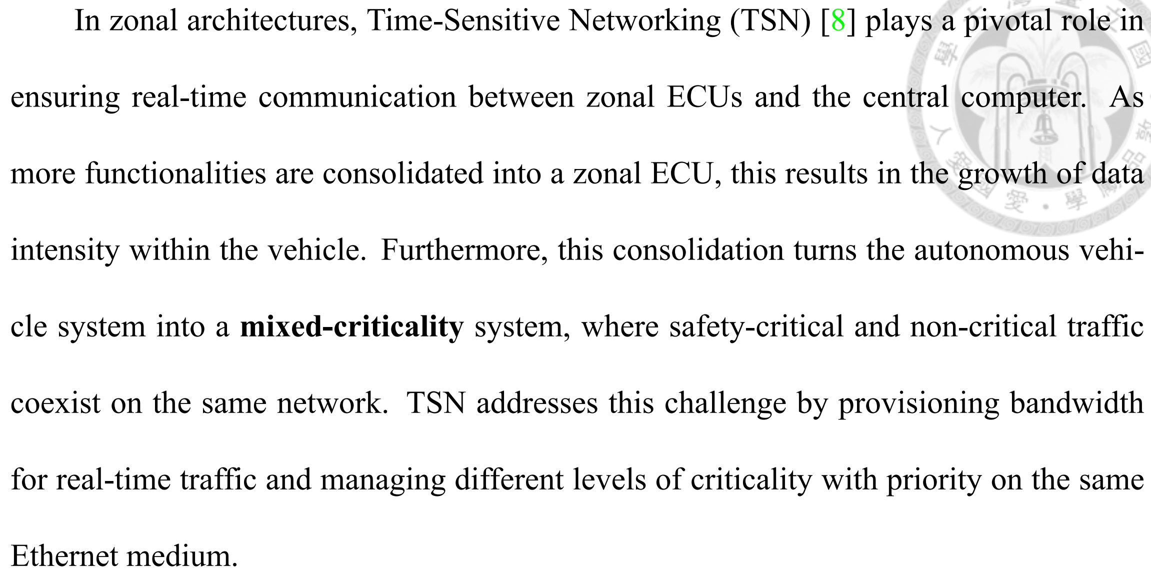


Figure 1.1: Domain and zonal architectures.

doi:10.6342/NTU202600098

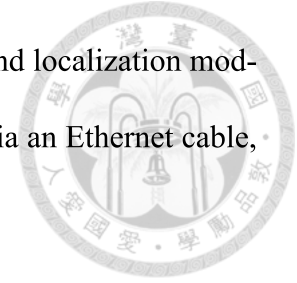


In zonal architectures, Time-Sensitive Networking (TSN) [8] plays a pivotal role in ensuring real-time communication between zonal ECUs and the central computer. As more functionalities are consolidated into a zonal ECU, this results in the growth of data intensity within the vehicle. Furthermore, this consolidation turns the autonomous vehicle system into a **mixed-criticality** system, where safety-critical and non-critical traffic coexist on the same network. TSN addresses this challenge by provisioning bandwidth for real-time traffic and managing different levels of criticality with priority on the same Ethernet medium.

Nevertheless, profiling and monitoring autonomous vehicle systems based on zonal architecture remains challenging. Zonal architecture system is a distributed system subject to real-time constraints. Currently available tools, such as CARET [10], can only profile a single host system. Meanwhile, the widely used distributed tracing framework, OpenTelemetry [15] cannot be directly applied to systems that communicate via DDS. Since understanding the interaction between high-level application logic (e.g., autonomous driving stacks) and low-level network behavior is crucial for validating performance, a more comprehensive monitoring approach is required.

In this thesis, we deploy a model of an autonomous vehicle system equivalent to a zonal architecture. This model encompasses two personal computers (PCs): the first emulates the zonal ECU, while the second simulates the central vehicle computer. The open-source software Autoware [16] is employed as a reference framework to approximate a real-world self-driving scenario. The PC mimicking the zonal ECU operates as a sensor server, responsible for generating and preprocessing sensor data, corresponding to the sensor module in Autoware. The second PC, emulating the central vehicle computer,

doi:10.6342/NTU202600098



conducts compute-intensive tasks, mainly consisting of perception and localization modules. These two PCs in the multi-host setup are directly connected via an Ethernet cable, with TSN features activated on both hosts.

We evaluate the performance of our autonomous vehicle system model using a top-down analysis. We begin with high-level application performance metrics, such as end-to-end response time. By using our modified version of CARET [10], we can trace the system distributively. For underlying system metrics, we utilize two tools to collect low-level performance data: we trace Ethernet network traffic with *ddshark*, a custom tool built on libpcap [17], and we record system-wide metrics periodically using *node_exporter* [2]. This data is stored in individual time-series databases, allowing users to inspect performance issues offline or monitor them in real time.

The remainder of the thesis is structured as follows: chapter 2 describes the background and related works for evaluating the performance of autonomous vehicle systems. chapter 3 details the methodology and the design of the tools used to analyze the system. chapter 4 evaluates the system using the proposed methodology and tools. Finally, chapter 5 concludes the thesis.

doi:10.6342/NTU202600098

Chapter 2 Background

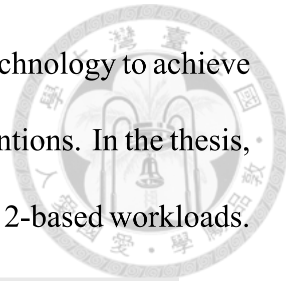
This chapter introduces the fundamental technologies of autonomous vehicle systems and related work in analyzing them. We begin with the high-level software-defined vehicle architecture (SOAFEE) and then proceed to the autonomous driving software stack (Autoware). Subsequently, we discuss communication middleware (ROS 2 and DDS) and network standards (TSN) that enable real-time performance. Finally, in this thesis, we review related work in performance profiling and observability.

2.1 SOAFEE

As SDVs functionality becomes increasingly complex, the development and deployment of software features also become challenging. The Scalable Open Architecture for Embedded Edge (SOAFEE) [3] initiative was introduced to provide a cloud-native architecture to support the development of SDVs. Figure 2.1 shows the reference architecture of SOAFEE. In the upper section of the figure, SOAFEE provides a computing environment that allows software development and validation in the cloud. Each of the workloads is running in its own containers. The bottom section of the figure, it shows the deployment of software from the cloud to the physical edge. Since the cloud and the edge both

doi:10.6342/NTU202600098

run SOAFEE architecture, developers can leverage containerization technology to achieve flexible deployment, and maintain safety requirements for specifics funtions. In the thesis, we follow the SOAFEE philosophy by using containers to deploy ROS 2-based workloads.



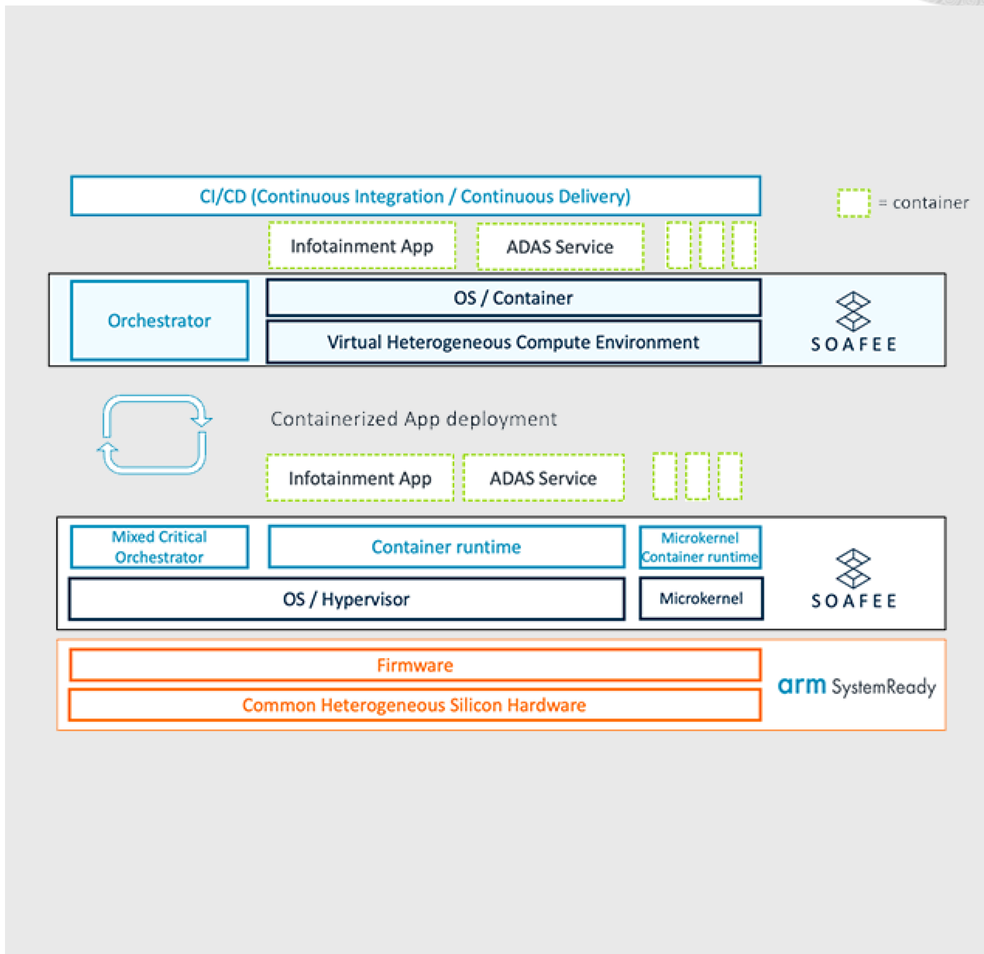


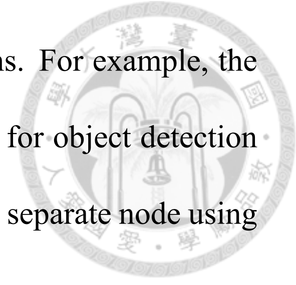
Figure 2.1: Architecture of SOAFEE.

2.2 Autoware

Autoware [16] is the leading open-source software stack for autonomous driving development. It adopts a modular design to provide essential components, including sensing, localization, perception, planning, and control. The data path begins with the sensing module, which interfaces with cameras, LiDARs, IMU, and GNSS. Then, the remaining

doi:10.6342/NTU202600098

modules use the sensor data to perform their corresponding functions. For example, the perception module uses the sensor data to perceive the environment for object detection and tracking. Within each module, each function is implemented as a separate node using ROS 2.



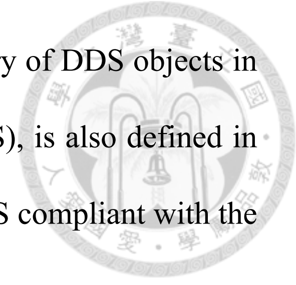
Due to its open-source nature and modular design, Autoware has been a reference for research. However, its performance heavily depends on the efficiency of the underlying middleware. In the thesis, we use a specific version of Autoware and adapt a reference simulation in a zonal architecture emulation environment. Then, we evaluate its performance for high-bandwidth pointcloud data and critical vehicle control data.

2.3 ROS 2/DDS

Autoware is based on the Robot Operating System 2 (ROS 2) [11], which serves as a framework for inter-node communication. ROS 2 is designed for building distributed real-time robotic applications. It uses Data Distribution Service (DDS) [14] as its underlying communication middleware. DDS provides decentralized data communication with a publish-subscribe paradigm, where the data are transmitted over specific topics. It abstracts the complexity for message passing between each node, allowing data exchange to occur seamlessly in a distributed environment.

The Real-Time Publish-Subscribe (RTPS) protocol [13] is the wire protocol for DDS. Developed by the Object Management Group (OMG), RTPS ensures interoperability among DDS implementations from different vendors. RTPS provides real-time, reliable message passing typically over UDP/IP. It also defines DDS objects, for example, DataWriter and

doi:10.6342/NTU202600098



DataReader, to manage data flow. The mechanism for auto-discovery of DDS objects in a distributed system, as well as support for Quality of Service (QoS), is also defined in RTPS. In this thesis, we use cycloneDDS, an implementation of DDS compliant with the latest RTPS standard.

2.4 Time-Sensitive Networking

To ensure strict real-time requirements for network transmission within a zonal architecture-based system, Time-Sensitive Networking (TSN) [8] is adopted in the vehicle network backbone. TSN is a set of IEEE 802.1 standards [6] that extend standard Ethernet to support deterministic communication. In the thesis, we specifically use the two evaluation standards: time synchronization and time-aware shaper.

2.4.1 Time Synchronization (IEEE 802.1AS)

Time Synchronization is a prerequisite in TSN. IEEE 802.1AS Generalized Precision Time Protocol (gPTP) guarantees that all machines on the same network share a common time reference. To achieve this, the protocol uses a hierarchical clock structure. The root of this hierarchy is a grandmaster, which serves as the network's ultimate clock source. The other machines operate as slaves and contionous exhange ptp packet with the grandmaster. In order to synchronize time, the machines use propagation delay across the physical network link, adjusting their NIC's hardware clock and minimizing the offset relative to the grandmaster.

doi:10.6342/NTU202600098

2.4.2 Time-Aware Shaper (IEEE 802.1Qbv)

One of the standards to ensure hard real-time transmission latency is IEEE 802.1Qbv Enhancements for Scheduled Traffic, also known as Time-Aware Shaper (TAS). The basic concept of TAS is a time-division multiple access (TDMA) scheme that allocates bandwidth to specific traffic on the Ethernet link. In the hardware level, it uses a gate control list (GCL) to open and close gates for specific traffic queues in a predefined cycle. For example, Figure 2.2 shows a TAS configuration with a cycle period of 500 microseconds. The configuration assigns real-time traffic a time interval of 50 percent of the cycle. The assignment is done by mapping real-time traffic to the traffic class (TC) with TC equal to 1 in Linux. By reserving bandwidth for high-priority traffic and blocking other traffic within a specific time interval, TAS eliminates queuing delay caused by interference, thereby guaranteeing bounded latency.

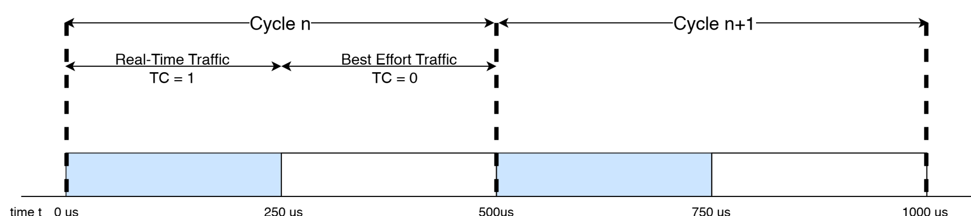


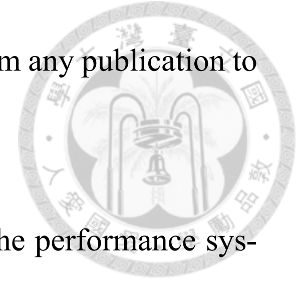
Figure 2.2: Operating principle of Time-Aware Shaper.

2.5 Related Work

There are tools available for profiling ROS 2 applications. Notably, ros2_tracing [5] provides application-level profiling, primarily utilizing LTTng tracepoints to collect performance data from the operating system and ROS 2 middleware. CARET [10] extends these capabilities by providing a comprehensive visualization of callback chains and mea-

doi:10.6342/NTU202600098

suring inter-node latency. It effectively reconstructs the data flow from any publication to subscription within the software stack in a single-host environment.



In Autoware system profiling, NXP provides a case study on the performance system with TSN enabled [7]. It uses the standard IEEE 802.1Qav Credit-Based Shaper and demonstrates that the system can maintain stable transmission latency under interference. In single-host profiling, the study [4] conducts an experiment to evaluate the performance of Autoware running with its simulator. It analyzes the performance characteristics of the underlying ROS 2 component, such as callback and timer latencies.

In the field of tracing distributed systems, OpenTelemetry [15] is widely used for systems that communicate via HTTP. By pushing metadata into the HTTP header, OpenTelemetry enables context propagation, which reconstructs causal relationships among distributed events. It also supports a wide range of backend databases for storing performance data and querying while the workload is executing. One of the popular backends is InfluxDB [9], a time-series database that supports metric querying with flexibility.

It is also important to understand system resource utilization for diagnosing performance bottlenecks. Tools such as *node_exporter* [2] expose real-time system metrics on CPU, memory, and network usage. Developers will need to configure *node_exporter* to export performance data into its corresponding time-series database. These data allow the user to correlate multiple abnormal characteristics (e.g., latency spikes) with hardware metrics (e.g., CPU saturation or bandwidth limits).

doi:10.6342/NTU202600098

Chapter 3 Methodology

This chapter describes the design and implementation of the tools that we used to evaluate the performance of the autonomous vehicle system based on Autoware. section 3.1 describes the overall design of the tools. section 3.2 describes the implementation of the tools in greater detail.

3.1 Design

3.1.1 System Overview

In this thesis, we employ top-down analysis to evaluate the performance of the autonomous vehicle system. Top-down analysis is a frequently adopted methodology to inspect system performance. It starts by collecting high-level application metrics that align with the target workload's context. Subsequently, it drills down into low-level system resource utilization to pinpoint potential performance problems. This method allows a comprehensive assessment of system operations, from a broader understanding down to specific performance elements.

Figure 3.1 illustrates the deployment of all the performance evaluation tools. First, we

doi:10.6342/NTU202600098

begin with the high-level application metrics that are closely associated with the context of the target workload; the related performance data is collected by the CARET tracing package as shown in the *caret record*. Next, to collect low-level system metrics, we deploy two tools, which include *ddshark* and *node_exporter* as stated in the figure as *Monitoring Tool*.

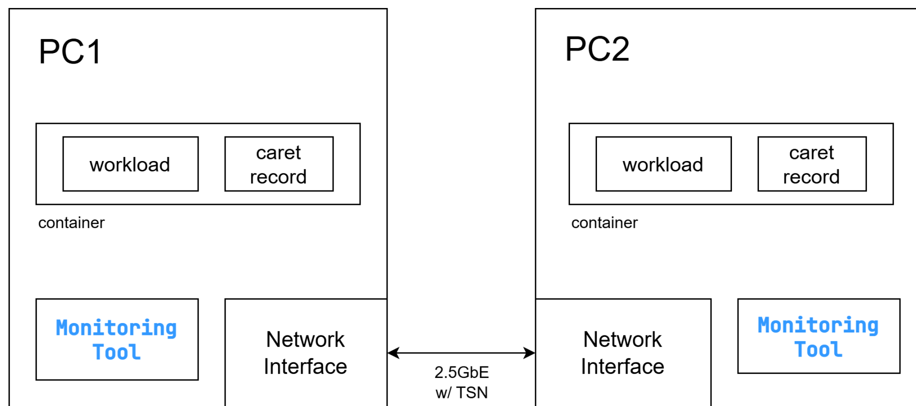


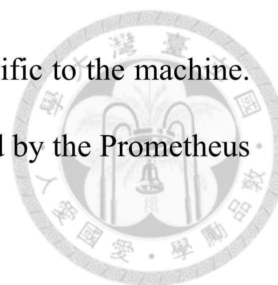
Figure 3.1: Overview of the deployments of performance evaluation tools.

When the workload is executing, both *caret record* and *Monitoring Tool* will write performance to their corresponding trace file or databases. For *caret record*, it uses LTTng as the tracing infrastructure to capture application traces by writing data to a specific directory. After the workload is complete, we can later merge the trace data from multiple machines into a single trace using our modified caret package.

As for *Monitoring Tool*, we use the setup as shown in the Figure 3.2 to store low-level performance data. For *ddshark*, it captures all the DDS traffic related to Autoware using pcap, dissects the DDS traffic, and exports the necessary information as OpenTelemetry traces. The traces will be sent to *OpenTelemetry Collector* and stored in a time series database called InfluxDB. Using *OpenTelemetry Collector* allows us to set a single endpoint to receive all the traces from any host in a distributed system. Another

doi:10.6342/NTU202600098

tool called *node_exporter* is used to collect systemwide metrics specific to the machine. *node_exporter* is a tool developed by Prometheus and will be scraped by the Prometheus time series database periodically using HTTP Pull request.



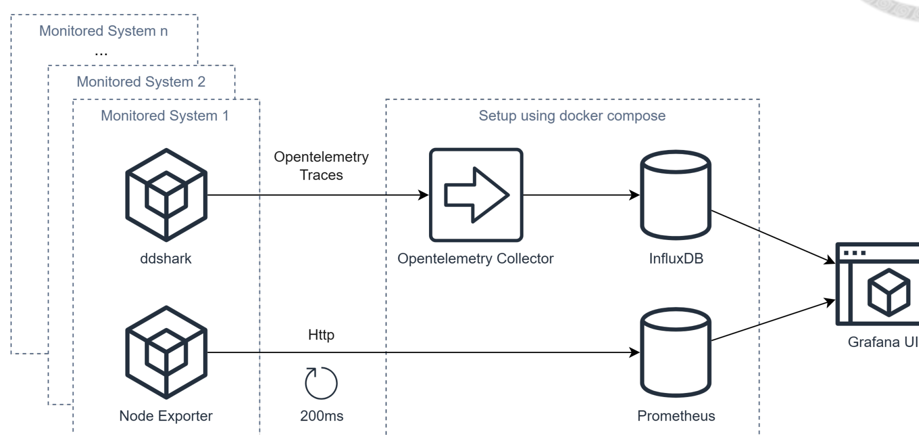
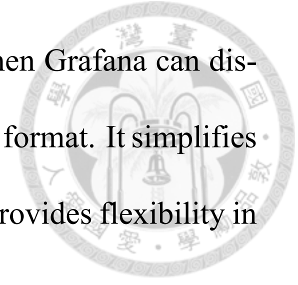


Figure 3.2: Overview of the setup for the monitoring framework.

To enhance usability, all the essential components of the monitoring framework can be established via a single command using *docker compose*, as indicated by the dotted-line enclosure in Figure 3.2. The setup is required only once, with the caveat that every host in the target system has more than one network interface. By configuring appropriate routing in the target system, the transmission of the performance data will not interfere with the workload of the application. Additionally, if developers have further needs regarding system observability, extra monitoring tools can be deployed along with their corresponding time-series databases. This demonstrates the flexibility and scalability of our design in handling complex performance monitoring requirements.

We use Grafana UI [1] as a front-end interface to visualize performance data from the two databases, as shown in the Figure 3.2. Grafana is an interactive observability platform that allows us to visualize metrics, traces, and logs in an intuitive manner. It supports a multitude of databases such as MySQL, PostgreSQL, Prometheus, and InfluxDB. Users

doi:10.6342/NTU202600098



only need to establish the connection to the target databases once, then Grafana can dispatch queries to these databases and visualize the result in a consistent format. It simplifies the process of interpreting large and complex performance data and provides flexibility in highlighting particular events.

In the thesis, Grafana UI is configured to query InfluxDB and Prometheus and delegate a portion of the data-processing tasks to these databases. It mitigates the risk of our data-processing code becoming tightly coupled to the performance data, since we use database-provided query functionality. This separation enhances the robustness and maintainability of the monitoring tools and aids in assessing system performance.

3.1.2 Performance Implication of TSN

To evaluate the impact and verify the real-world results of using TAS to provision high-priority traffic, we formulate a deterministic model to estimate worst-case DDS message latency. The model derives the arrival time denoted as T_{arrive} with the parameters defined as follows:

- P: Cycle period (in microseconds)
- D: Time interval (in percentage) allocated for transmission within the cycle period
- R: Physical link rate (in bits per second)
- X: Total size of the data to be transmitted (in bytes)

First, the effective bandwidth allocated to high-priority traffic correlates to the configured time interval D as shown in Eq. (3.1). As a result, the theoretical transmit time

doi:10.6342/NTU202600098

$T_{transmit}$ for a payload of size X can be derived in Eq. (3.2). It assumes that the network transmits the payload continuously. However, in a time-division network, the message is transmitted within an allocated time slot during each cycle. Therefore, we defined N_{cycles} in Eq. (3.3) as the number of periods required to transmit a full message. The ceiling function treats partial-cycle counts as full-cycle counts. The additional cycle is added to account for the worst-case scenario in which a message starts transmission just after the time interval has closed. Finally, Eq. (3.4) combines these factors to estimate T_{arrive}, which is the time when the complete message arrives at the receiver.

$$Bandwidth = R \times D \tag{3.1}$$

$$\begin{aligned} T_{transmit} &= \frac{8 \times X}{Bandwidth} \\ &= \frac{8 \times X}{R \times D} \end{aligned} \tag{3.2}$$

$$N_{cycles} = \left\lceil \frac{T_{transmit}}{P} \right\rceil + 1 \tag{3.3}$$

$$\begin{aligned} T_{arrive} &= N_{cycles} \times P \\ &= \left(\left\lceil \frac{T_{transmit}}{P} \right\rceil + 1 \right) \times P \\ &= \left(\left\lceil \frac{8 \times X}{R \times D \times P} \right\rceil + 1 \right) \times P \end{aligned} \tag{3.4}$$

doi:10.6342/NTU202600098

3.2 Implementation

3.2.1 High-level Application Performance Metrics: CARET

We use the CARET package to trace high-level performance data associated with the profiled application workload. The recording procedure is identical to that of a single-host setup, as shown in the Figure 3.3. In a multi-host setup, we run *caret record* on both PC 1 and PC 2. The trace data will be recorded on both machines with CARET. After the workload completes execution, we will use our modified CARET to merge the trace data collected from the two PC2s.

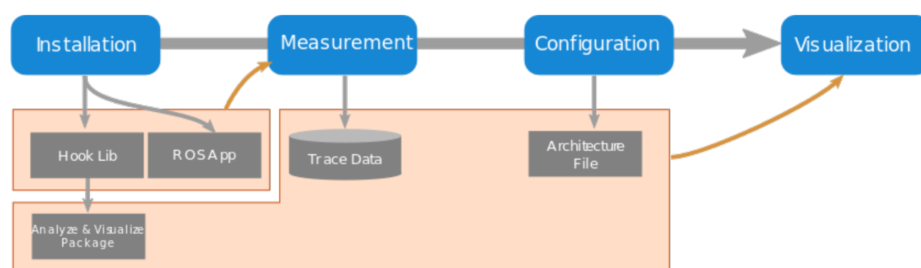
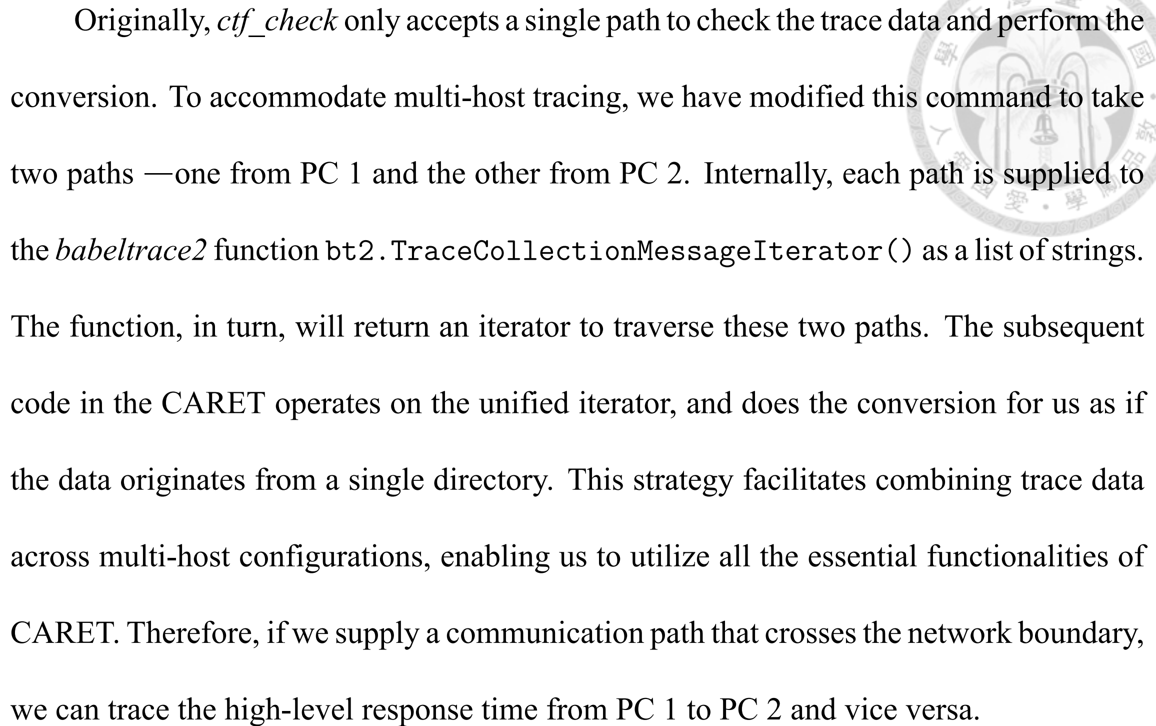


Figure 3.3: Overview of the performance profiling workflow in CARET.

The key to merging trace data from different hosts is to treat multiple underlying trace data sources as a single iterator. Since CARET relies on LTTng to write traces data, it fundamentally uses a package called *babeltrace2* for trace data handling. In CARET, it provides a subcommand called *ctf_check* to convert trace data in the format of CTF into a single file named *caret_converted*. All other analysis subcommands will examine the presence of the file *caret_converted*. In the absence of a file named *caret_converted*, the subcommand triggers trace data conversion functionalities before processing the trace data.

doi:10.6342/NTU202600098



Originally, *ctf_check* only accepts a single path to check the trace data and perform the conversion. To accommodate multi-host tracing, we have modified this command to take two paths —one from PC 1 and the other from PC 2. Internally, each path is supplied to the *babeltrace2* function `bt2.TraceCollectionMessageIterator()` as a list of strings. The function, in turn, will return an iterator to traverse these two paths. The subsequent code in the CARET operates on the unified iterator, and does the conversion for us as if the data originates from a single directory. This strategy facilitates combining trace data across multi-host configurations, enabling us to utilize all the essential functionalities of CARET. Therefore, if we supply a communication path that crosses the network boundary, we can trace the high-level response time from PC 1 to PC 2 and vice versa.

It is important to note that for this modification to function correctly, one important prerequisite is that the clocks of every host in the system must be synchronized. This is because, in some data paths, CARET relies on timestamps to resolve the causality of happen-before relationships between publication and subscription. Accurate time synchronization is achieved by PTP with the support of a hardware clock on NICs, which provides accuracy at the sub-microsecond level. We will explain the complete configuration of PTP as part of the setup within TSN in chapter 4.

3.2.2 Low-level DDS Network Metrics: ddshark

We develop *ddshark*, a versatile real-time monitoring tool designed for tracing DDS network traffic. DDShark supports exporting traces in OpenTelemetry format for each captured DDS message. Although the tool has been thoroughly tested to capture traffic generated by CycloneDDS, its design should be able to capture traffic from any DDS

doi:10.6342/NTU202600098

implementation. *ddshark* is written solely in the Rust programming language, and depends on the Cargo-based crates and projects on GitHub.

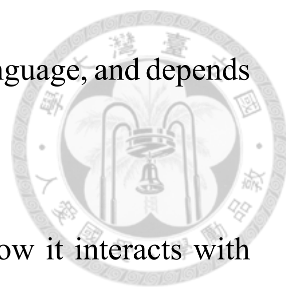


Figure 3.4 illustrates the overall architecture of *ddshark* and how it interacts with system components. *ddshark* utilizes *libpcap*, a user-space library that interacts with the `AF_PACKET` interface in the Linux kernel to provide packet-capture facilities. As depicted in the diagram, packet interception occurs at specific tap points within the Linux kernel. For egress traffic, a packet is tapped at `dev_queue_xmit_nit`, whereas for ingress traffic, it is tapped at `__netif_receive_skb_core`, right after the packet is received in the softirq `net_rx_action`.

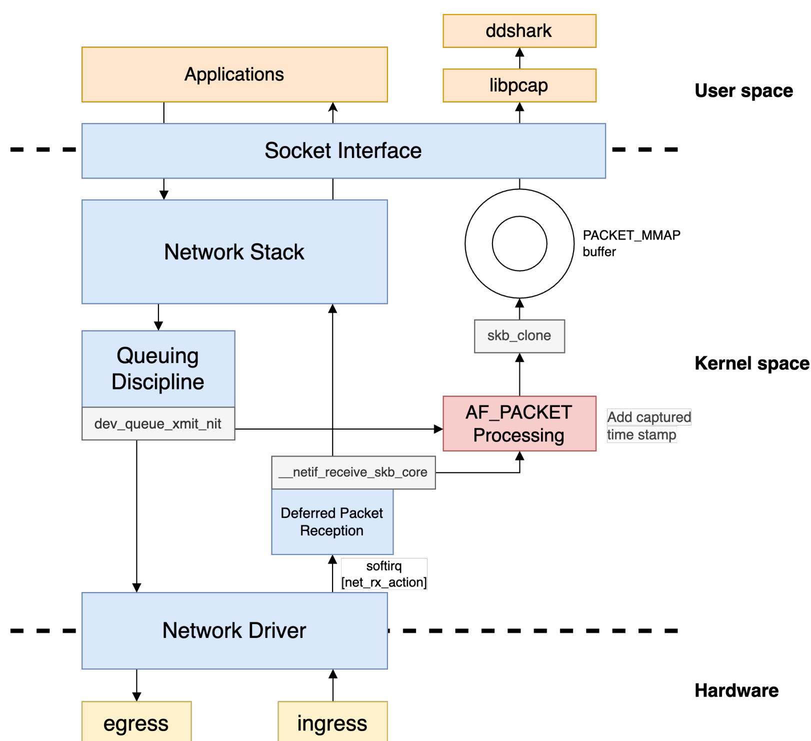
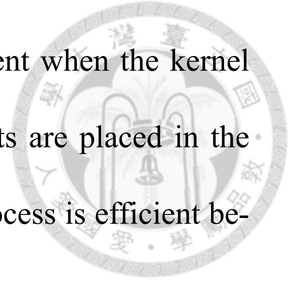


Figure 3.4: Overall and underlying architecture of ddshark.

The time-stamping mechanism is critical for this design. We employ software timestamping to record the timing of captured network events. In the *AF_PACKET Processing* block of the diagram, the timestamp is attached to the packet when the packet is cloned

doi:10.6342/NTU202600098



via skb_clone. It guarantees that the timestamps reflect the moment when the kernel packet handlers process the packets. Next, the timestamped packets are placed in the PACKET_MMAP ring buffer and then transferred to user space. The process is efficient because it avoids the memory-copying overhead between kernel and user space.

3.2.3 Dissecting DDS Traffic with ddshark

ddshark uses *libpcap* for low-level network monitoring to capture a stream of packets. Subsequently, ddshark dissects packets according to the Real-Time Publish-Subscribe (RTPS) protocol. RTPS is the wire protocol for DDS, and every standard-compliant DDS implementation is built on top of it. Figure 3.5 outlines a typical layout of an RTPS packet. Generally, DDS uses UDP as the transport layer, with the RTPS packet placed in the UDP payload. An RTPS packet consists of an RTPS header and one or more RTPS sub-messages. The presence of multiple sub-messages is attributed to the DDS decision to piggyback sub-messages in the same packet to reduce network bandwidth consumption. An RTPS packet contains a 20-byte RTPS header, composed of RTPS magic, Protocol version, Vendor ID, and GUID prefix. Among all, the GUID prefix is particularly important, as it identifies the participant (or node, in the context of ROS 2) within a DDS domain that is transmitting this RTPS packet.

doi:10.6342/NTU202600098

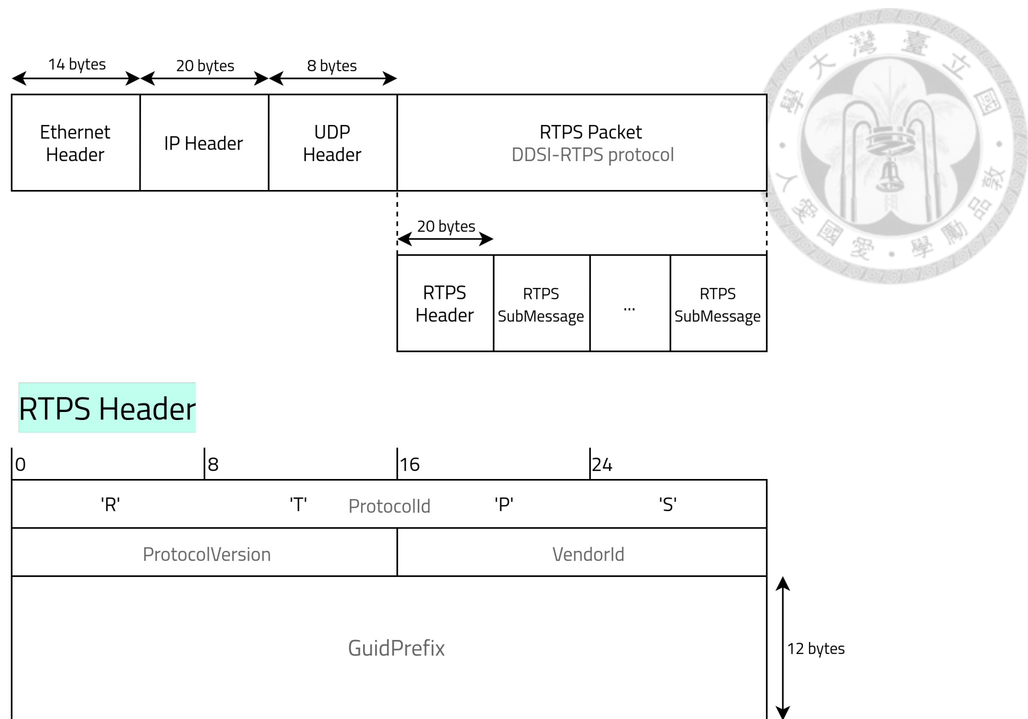
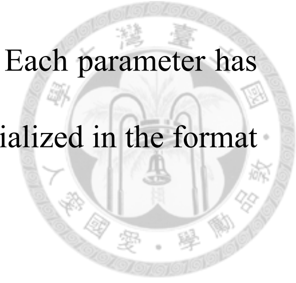


Figure 3.5: Typical layout of an RTPS packet.

In order to obtain essential information associated with the RTPS packets, for example, the topic name that is linked to each packet, *ddshark* needs to deserialize data in the payload of the packet. Specifically, in our implementation, we first filter DDS discovery data that is related to *DataWriter* (or publisher, in the context of ROS 2), and deserialize the Data submessage in the discovery data.

In RTPS, the discovery procedure is formulated as the Simple Discovery Protocol. The built-in endpoints in DDS are responsible for exchanging the necessary discovery data between participants. One important discovery data specified in the standard is *DiscoveredWriterData*. The data structure represents and exchanges information about a discovered *DataWriter*, the information includes endpoint ID, Quality of Service (QoS) settings, type of the published data, and most importantly, the name of the topic. When this information is serialized into Data Submessages, DDS uses a format called *ParameterList*

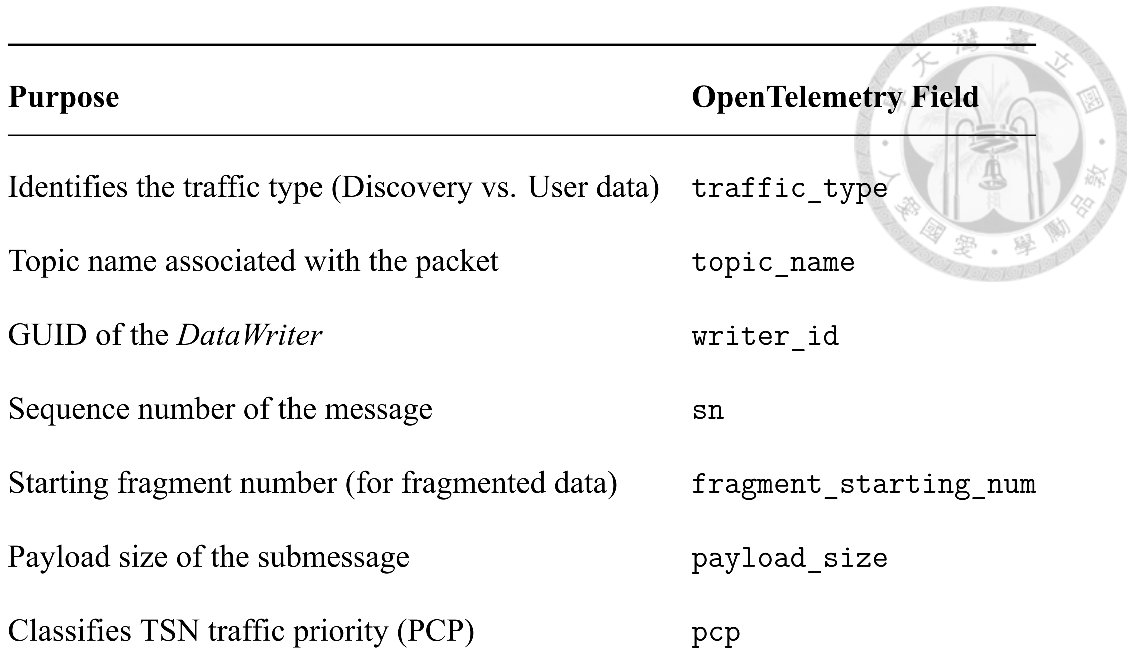
doi:10.6342/NTU202600098



and assigns each piece of information as a sequence of parameters. Each parameter has its own ID specified by ParameterId (PID). These parameters are serialized in the format of Common Data Representation (CDR).

Therefore, in ddshark, we deserialize and retrieve information from the discovery data. Utilizing the GUID prefix and endpoint ID of the *DataWriter*, we create a mapping from the GUID of *DataWriter* to discovery data. Subsequently, when an RTPS packet is captured, we can index the mapping and export the necessary information as OpenTelemetry traces. A single RTPS packet could generate multiple traces since there may be multiple submessages in an RTPS packet. We extract each of the essential DDS information and store it as attributes of an OpenTelemetry trace. A comprehensive list of the information we captured is shown in Table 3.1. The field *traffic_type* is used to differentiate the type of the DDS submessage, whether it is published from built-in endpoints or user-defined endpoints. The remaining fields are used as essential metadata required for performance metrics construction.

doi:10.6342/NTU202600098



Purpose	OpenTelemetry Field
Identifies the traffic type (Discovery vs. User data)	`traffic_type`
Topic name associated with the packet	`topic_name`
GUID of the *DataWriter*	`writer_id`
Sequence number of the message	`sn`
Starting fragment number (for fragmented data)	`fragment_starting_num`
Payload size of the submessage	`payload_size`
Classifies TSN traffic priority (PCP)	`pcp`

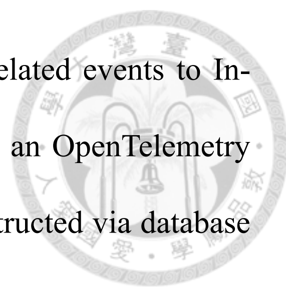
Table 3.1: Mapping between captured DDS traffic information and OpenTelemetry attributes.

The primary objective of defining these OpenTelemetry fields is to approximate context propagation in a distributed tracing environment. In distributed tracing, context propagation is a technique that is commonly used to link causally related events in a distributed environment. In the case of OpenTelemetry, context propagation is supported only with the HTTP protocol, where its SDK is capable of injection relevant tracing data, such as trace ID or span ID, onto HTTP headers. In ddshark, it cannot leverage the native support of context propagation for two reasons. First, it uses RTPS as the wiring protocol to exchange data between hosts, which is incompatible with the HTTP protocol. Second, *ddshark* uses libpcap as a passive sniffer to capture packets. It is unable to alter the original in-flight packets to add tracing headers because it records copies of packets from the kernel.

In response to these challenges, we adopt a post-processing correlation strategy. In-

doi:10.6342/NTU202600098

stead of header injection, we offload the task of linking causally related events to InfluxDB. By making *ddshark* export only the necessary metadata to an OpenTelemetry endpoint, causal relationships in the performance data can be reconstructed via database queries.



3.2.4 Capturing Fragmented Data with ddshark

In autonomous vehicle systems, sensor data such as camera images and LiDAR point clouds is often too large to send within a single transmission. Consequently, the transmission of a single DDS message involves fragmentation at multiple layers. Figure 3.6 illustrates the two-stage fragmentation process observed in DDS systems. First, at the DDS middleware layer, if the data size exceeds the maximum message size of the DDS protocol (typically 64KB for UDP-based transport), DDS splits the original data into multiple `DATA_FRAG` submessages. Second, at the network layer, each `DATA_FRAG` is further divided into smaller IP fragments to fit within the Ethernet MTU (typically 1500 bytes).

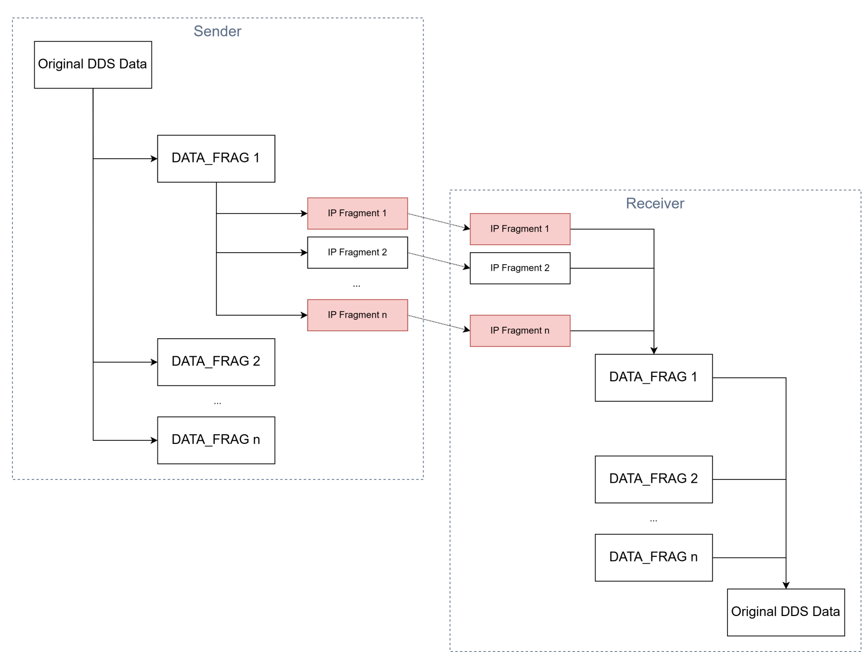


Figure 3.6: Capturing Fragmented Data with ddshark.

doi:10.6342/NTU202600098

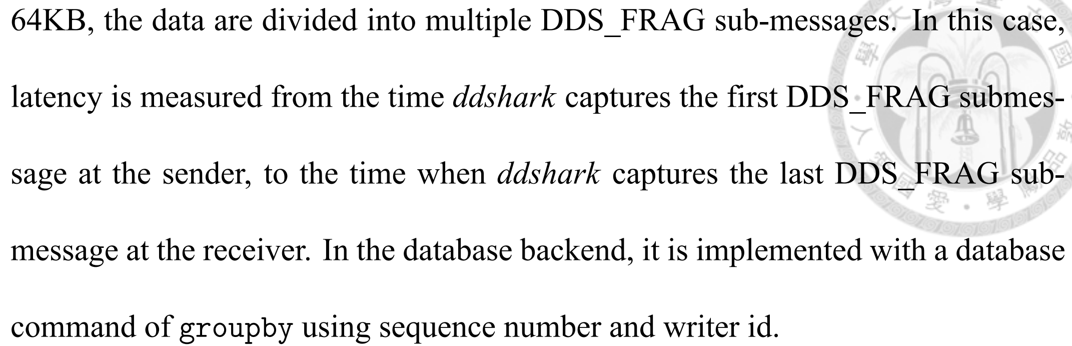
However, the performance data of DDS traffic will be large if we capture all the IP fragments within a DDS message stream. Therefore, when handling fragmented data, *ddshark* only captures the first and the last IP fragment of the `DATA_FRAG` submessage as highlighted by the red blocks in Figure 3.6. This allows for the accurate and efficient measurement of the transmission latency of such large DDS messages.

3.2.5 Latency Metrics Definition

We measure performance with our proposed framework. With the performance data collected from modified CARET and *ddshark*, we define the following metrics:

- **End-to-end:** The latency is defined as the time from the publication of the first ROS 2 node to the time when the last ROS 2 node receives the message along a specific data path. In CARET, it is referred to as path latency.
- **ROS 2 Layer Communication:** The latency is the time required to transmit messages between ROS 2 nodes. Specifically, it encompasses the time interval from the publication of a message to its receipt by a subscriber. In the paper, we specifically discuss ROS 2 latency involving transmission across Ethernet when two nodes are located on different hosts.
- **DDS Layer Communication:** The latency is defined as the time when the full DDS message is transmitted from the sender and is fully received at the receiver. Specifically, for the unfragmented DDS data (typically less than 64KB), the latency is the interval between the time when *ddshark* captures the packet at the sender and the time when *ddshark* captures the packet at the receiver. For DDS data exceeding

doi:10.6342/NTU202600098



64KB, the data are divided into multiple DDS_FRAG sub-messages. In this case, latency is measured from the time *ddshark* captures the first DDS_FRAG submessage at the sender, to the time when *ddshark* captures the last DDS_FRAG submessage at the receiver. In the database backend, it is implemented with a database command of `groupby` using sequence number and writer id.

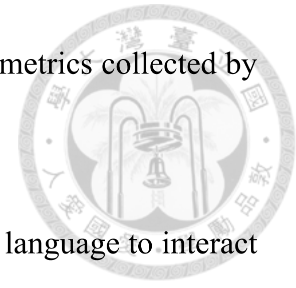
- **DDS Submessages:** The metrics apply only to DDS messages that are too large and need to be fragmented into multiple DDS_FRAG submessages. The latency is defined as the time that a DDS submessage is transmitted from the sender to the receiver. In case of IP fragmentation, the latency is defined as the time from the time when *ddshark* captures the first IP fragment at the sender, to the time when *ddshark* captures the last IP fragment at the receiver. In the database backend, it is implemented with a database command of `groupby` using sequence number, fragment starting number, and writer id.

3.2.6 Performance Data Visualization

For high-level metrics captured by CARET, the CARET tool already provides convenient tools in the *caret_analyze* package. Because we modified only the underlying performance data processing workflow, we can reuse the existing CARET visualization tools to analyze a workload running in the multi-host setup.

For DDS network metrics and system metrics, we use Grafana UI for visualization. It serves as the centralized dashboard for online monitoring. As described in the system design, there are two sources for performance data: InfluxDB for DDS network traf-

doi:10.6342/NTU202600098



fic traces collected by *ddshark* and Prometheus for system resource metrics collected by *node_exporter*.

To visualize the DDS network traffic, we utilize the Flux query language to interact with the InfluxDB database. As shown previously in the Table 3.1, ddshark exports the DDS trace data with attributes defined in OpenTelemetry. We can then construct queries to inspect specific metrics related to DDS traffic.

For the system-level metrics, we utilize PromQL (Prometheus Query Language) to retrieve time-series data on CPU utilization, memory usage, and network interface metrics for system-level analysis. The *node_exporter* broadcasts these metrics through HTTP, which Prometheus retrieves at a defined interval. Finally, our dashboard uses PromQL to query the Prometheus database and visualize and compare it with the DDS message traffic obtained from *ddshark*.

The advantage of this design and implementation is the ability to correlate multiple performance data sources. By aligning the panels of DDS traffic and system metrics on a shared time axis, we can identify potential performance bottlenecks. For example, if we observe a sudden increase in transmission latency in a DDS message, we can check for system resource saturation or network congestion to determine whether it is the culprit. This unified dashboard provides the observability required to validate the real-time performance of the autonomous vehicle system.

doi:10.6342/NTU202600098

Chapter 4 Evaluation

In this chapter, we evaluate our proposed framework using top-down analysis. We utilize two workloads—a ROS 2 simple robotics application and a real-world autonomous driving workload—to test the system's performance. The results show that the framework can profile distributed systems commonly seen in domain and zonal architectures, enabling an in-depth analysis of network latency with fine-grained metrics. Furthermore, we test the system with and without TSN under different network loads. It shows that our proposed framework can examine distinct performance characteristics in various configurations.

4.1 Experimental Setup

In this section, we show the hardware setup and software configuration that were used throughout the experiments.

4.1.1 Hardware and Network Topology

Figure 4.1 shows the system setup for evaluation. We run the workload and CARET record command on each PC within the same container. The monitoring tool, i.e., *ddshark*

doi:10.6342/NTU202600098

and *node_exporter*, runs outside the container. We use the hardware described in Table 4.1 to conduct the evaluation experiments. In order to achieve better real-time performance and repeatability for each experiment, both machines have their frequency fixed to their base frequency. That is, PC1 is fixed to 2.5GHz and PC2 to 3.5GHz. Also, both machines have their C-State fixed to C0 state, which helps prevent the overhead of transition from power save to active state in the CPU. Furthermore, SMT on both machines is also disabled to avoid interference from other processes on the same physical core.

For network connection, the two PCs are connected directly to each other without a switch or router in between. Both PCs use an Intel NIC capable of a 2.5Gbps link rate with TSN support. We enable time synchronization and TAS to conduct the experiments below. To prevent the system from synchronizing time with NTP servers, systemd-timesyncd is disabled.

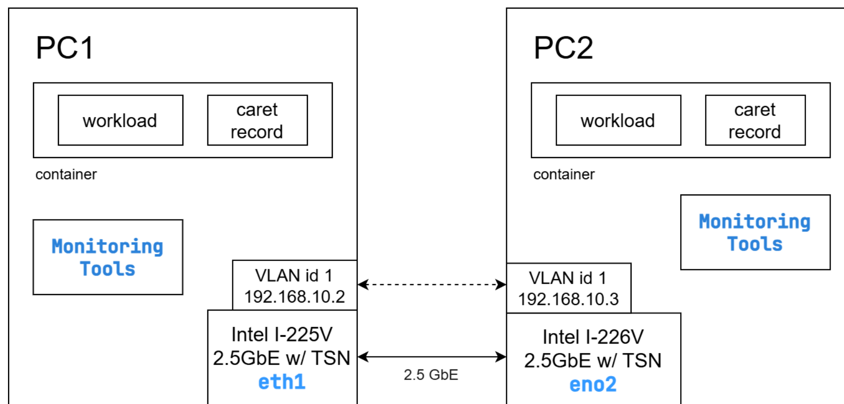
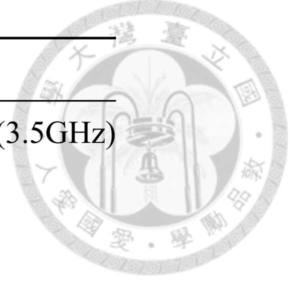


Figure 4.1: System setup for evaluation.

4.1.2 Software Configuration and Time Synchronization

The software and its version used throughout the evaluation section are shown in Table 4.2. Note that we use upstream cycloneDDS rather than the default cycloneDDS

doi:10.6342/NTU202600098



Component	PC1	PC2
CPU	Intel i7-11700 (2.5GHz)	Intel i9-13900K (3.5GHz)
Memory	64GB	128GB
GPU	RTX 3070	RTX 4070 Ti
NIC	Intel i225-V	Intel i226-V
OS	Ubuntu 22.04.2 LTS	Ubuntu 22.04.2 LTS

Table 4.1: Description of the computation platform used for evaluation.

provided with ROS 2. The distinction is crucial to the most important step in our performance tuning: mapping software-level DDS topics to hardware-level TSN queues. Unlike some DDS implementations, CycloneDDS does not support direct priority configuration. To address the challenge, we devised a two-layer mapping configuration as shown in the Figure 4.2. First, we use the `NetworkPartition` parameter available in the upstream cycloneDDS to bind specific topics to VLAN network interfaces. Next, we use the *iptable* to set the packet priority originating from these VLAN network interfaces. Thus, the traffic from a specific DDS topic can be mapped to the desired TSN hardware transmission queues.

Software	Version
ROS 2	Humble
CycloneDDS	0.10.x (commit fd9ad36e)
Autoware	tag 2023.10

Table 4.2: Description of the application software used for evaluation.

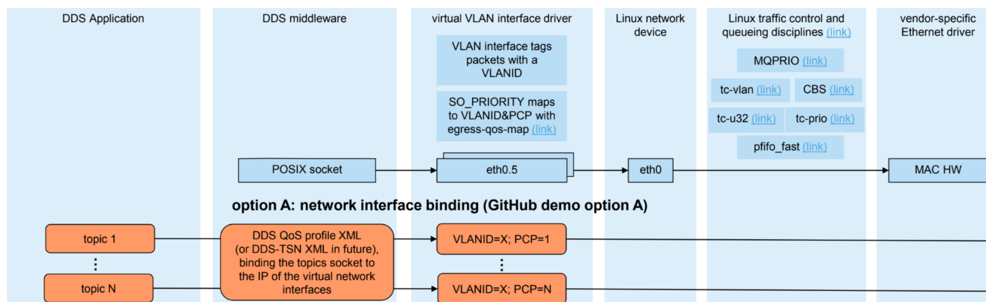
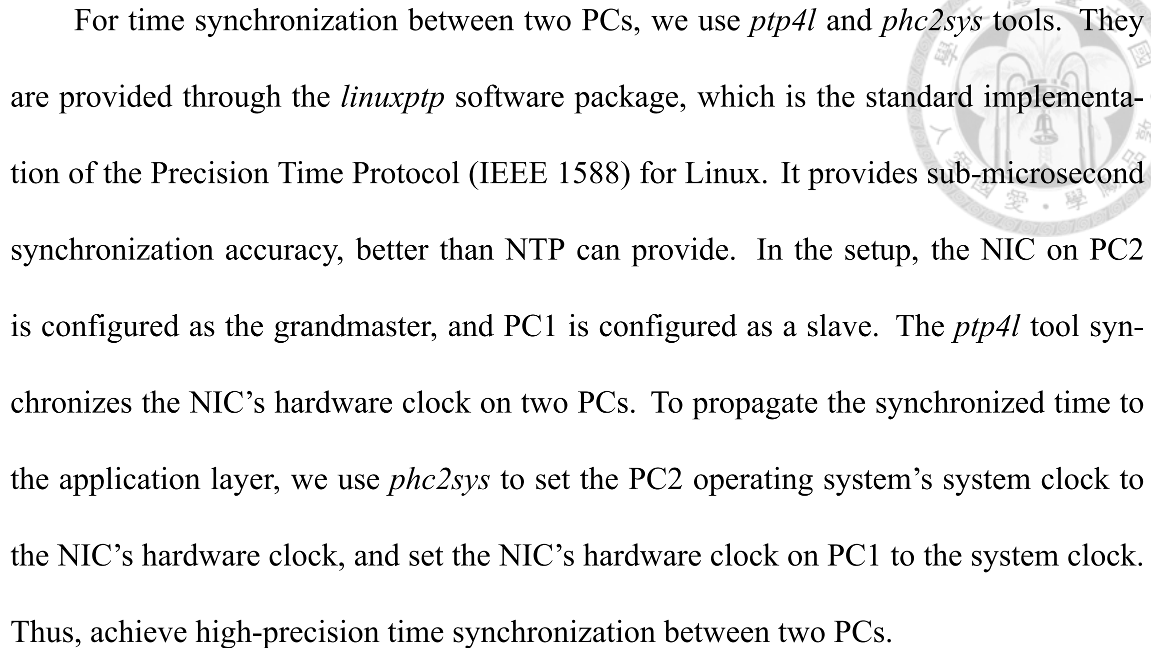


Figure 4.2: System setup for mapping DDS topic to TSN priority.

doi:10.6342/NTU202600098



For time synchronization between two PCs, we use *ptp4l* and *phc2sys* tools. They are provided through the *linuxptp* software package, which is the standard implementation of the Precision Time Protocol (IEEE 1588) for Linux. It provides sub-microsecond synchronization accuracy, better than NTP can provide. In the setup, the NIC on PC2 is configured as the grandmaster, and PC1 is configured as a slave. The *ptp4l* tool synchronizes the NIC's hardware clock on two PCs. To propagate the synchronized time to the application layer, we use *phc2sys* to set the PC2 operating system's system clock to the NIC's hardware clock, and set the NIC's hardware clock on PC1 to the system clock. Thus, achieve high-precision time synchronization between two PCs.

4.2 Experimental Design and Scenarios

4.2.1 Workload and Configurations

We utilize two workloads, simple robotic application and realistic autonomous driving workload, to evaluate our proposed framework. In each workload, we profile the transmission performance of both large and small control messages. The sample sizes are consistent across both workloads, with 300 samples for large messages and 870 samples for small control messages. Note that for small control messages, the data is transmitted directly between two nodes on different PCs through Ethernet. In addition, the data size is small enough to fit within a single DDS data. Therefore, the analysis of these messages focuses exclusively on ROS 2 and DDS-layer metrics.

- **Simple Robotic Application:** The workload models an architecture commonly

doi:10.6342/NTU202600098

seen in robotics. Aiming to run a minimal robotics system in a controlled environment, it consists of **Image Data** (Large messages) representing high-bandwidth traffic and **Drive Data** (Small messages) representing latency-critical control traffic.

- **Real-world Autonomous Driving Workload:** Based on the Autoware stack, the workload model involves real-world complexity. It features **LiDAR Pointcloud** data as the high-bandwidth data and **Vehicle Status** messages as the critical control data.

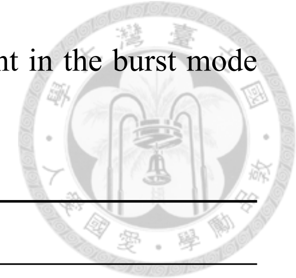
In the configuration name, we define *Baseline* where TAS is disabled and only PTP time synchronization is active. Whereas TAS-enabled configuration follows the format *TAS-XX%-YYus*. The first number indicates the percentage of time intervals set within a cycle, which is the open duration of the gate for high-priority traffic. The second number indicates the cycle time. If omitted, the cycle time defaults to 500 microseconds. Both high-bandwidth data and critical control data will be assigned to high-priority traffic when TAS is active. Specifically, for the Realistic Autonomous Driving Workload, all traffic in the sensing namespace and Vehicle Status messages are assigned to high-priority traffic.

4.2.2 Background Interference Models

To compare the performance of the system with and without TSN, we use the experimental scenarios shown in Table 4.3 to evaluate the performance under different levels of interference. The interference traffic is sent from PC1 to PC2, generated using iperf3, with the payload set to 1400 bytes and using the UDP protocol. In addition, the pacing

doi:10.6342/NTU202600098

timer is set to 15ms, which means that the interference traffic is sent in the burst mode every 15ms.



ID	Scenario	Network Load	Objective
I	No Load	0 Gbps (0%)	Establishes reference metrics under ideal conditions.
II	Moderate Interference	1.0 Gbps (40%)	Simulates high-traffic load to evaluate performance stability.
III	Heavy Interference	2.4 Gbps (95%)	Stress tests the system near link bandwidth saturation.

Table 4.3: Experimental scenarios and interference levels.

4.3 Case Study I: Simple Robotic Application

Our evaluation starts with a simple application to model an architecture commonly seen in robotics applications. The case study evaluates performance when the workload transmits and receives large messages and small control messages with and without TSN.

The application consists of six ROS 2 nodes that run on a separate machine. The graph of the nodes is shown in Figure 4.3. The nodes running on Machine PC1, which has less computing power, simulate the sensor's workload. The other machine, PC2, which has more computing power, simulates the actuator workload. In the simple application setup, the node /sensor_1 will publish a mock image data to the node filter over the topic /raw_image. After the node filter receives the data, it will publish the filtered data over the node /msg_driven on PC2. Also, a 10ms delay is added to /msg_driven node to simulate computing overhead. The end-to-end latency for the image message is defined as the time from node sensor_1 publishing a message to the time when the node /actuator receives the

doi:10.6342/NTU202600098

message.

To simulate small control message transmission, we utilize the VelocityReport data format and transmit over the topic /drive. This data is also used in Autoware to report velocity status. In the simple application setup, we create a published node /sensor and a subscriber node /monitor_node to handle the topic /drive. Table 4.4 shows the topics transmitted over the Ethernet. For both large (/image) message and small (/drive) message transmission, the ROS 2 layer communication latency is measured in the ROS 2 software layer. Similarly, the communication latency of the DDS layer is defined as the transmission latency of the underlying DDS middle layer.

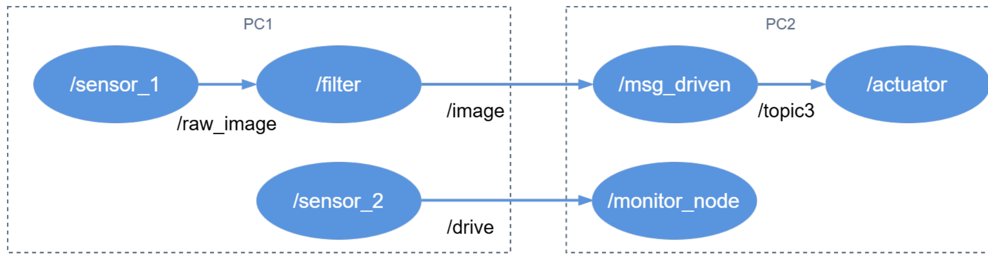


Figure 4.3: ROS 2 Node diagram for simple application.

Topic Name	Data format	Message Size	Hz	Bandwidth
/image	1280x720 (bgr8)	2.76 MB	10	27.76 MB/s
/drive	VelocityReport	36 B	29	1.21 KB/s

Table 4.4: Description of the message sent over Ethernet in simple robotic application.

4.3.1 Analysis of Monitoring Overhead

We begin by evaluating the overhead of our proposed framework. Table 4.5 shows the latency measured in the ROS 2 layer with and without active monitoring tools in milliseconds. We utilized CARET for these measurements. As established in the previous

doi:10.6342/NTU202600098

work, CARET is known to be highly efficient. Therefore, the source of the overhead is mainly attributed to libpcap and the tool's processing logic. The results suggest that the overhead is minimal and does not significantly impact the transmission of Ethernet data. This confirms the effectiveness of our proposed framework.

	ROS 2 Layer Comm.			
Configuration	**Mean**	**Std**	**P99**	**Max**
/image + w/o monitoring tools	10.980	0.086	11.270	11.549
/image + w/ monitoring tools	11.077	0.209	11.835	13.720
/drive + w/o monitoring tools	0.099	0.023	0.167	0.208
/drive + w/ monitoring tools	0.104	0.023	0.174	0.201

Table 4.5: Statistical descriptives for overhead comparison.

4.3.2 Analysis of Large Data Message (Image)

Figure 4.4 shows the performance of *Baseline* and *TAS-50%* configurations in different scenarios. We begin by establishing a performance baseline: with a minimum compute time of 10 ms and a transmission time of 8.83 ms (calculated from 2.76 MB at 2.5 Gbps), the theoretical minimum end-to-end latency is 18.83 ms. Note that ROS 2 overhead and inter-node communication on the same host are excluded. Observations show that the *Baseline* mean latency degrades significantly under load, ranging from 24 ms up to 61 ms. In contrast, *TAS-50%* yields a consistent latency of $\approx$34 ms. Applying a top-down analysis to finer metrics (ROS 2, DDS, and DDS submessage layers) confirms the root causes. While the *Baseline* performance disturbance is directly correlated with network load at middleware layers, the *TAS-50%* configuration shows stable underlying metrics. By drilling into finer metrics, it affirms that the deterministic mechanism of TAS effectively mitigates interference for large messages, resulting in a stable end-to-end distribu-

doi:10.6342/NTU202600098

tion across all load levels.

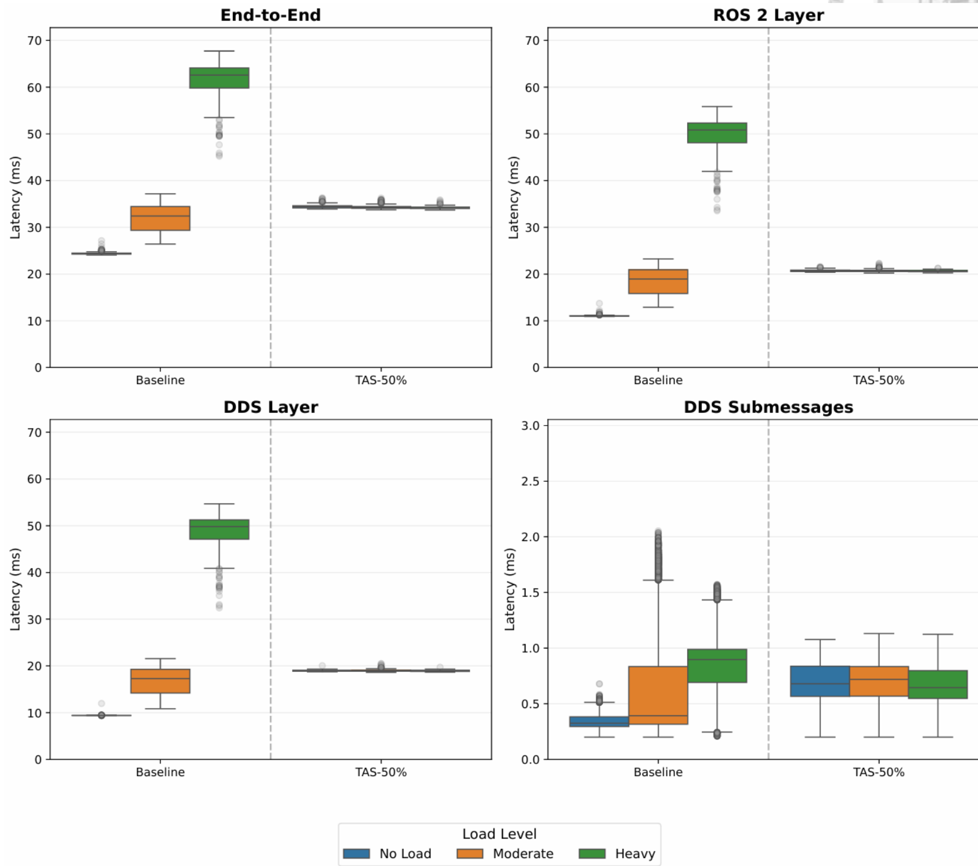


Figure 4.4: Boxplot of image messages in simple application under various metric types.

Table 4.6 shows the calculated and measured latencies using the cycle period and the percentage of the time interval as variables. The column T_{arrive} lists the calculated values with the derived formula from chapter 3, and the measured mean represents the DDS latency. We observe that the latency of large messages is highly correlated with the percentage of the time interval, and the theoretical T_{arrive} is close to the real-world measurements. However, when varying the cycle in the configuration, the measured latency remains almost unchanged. Although the derived formula remains accurate, it slightly overestimates latency when the cycle period is large or when the percentage of time interval is close to 100%.

doi:10.6342/NTU202600098

Configuration	Parameters		Latency Metrics			
	Cycle (μs)	Interval (%)	T_{arrive} (ms)	Measured (Mean) (ms)	Std Dev (ms)	Diff (%)
TAS-20%	500	20	47.00	46.99	0.18	-0.03
TAS-40%	500	40	24.00	23.87	0.14	-0.55
TAS-60%	500	60	16.00	15.84	0.12	-1.02
TAS-80%	500	80	12.50	11.87	0.08	-5.05
TAS-50%-100μs	100	50	18.70	19.00	0.10	+1.58
TAS-50%-250μs	250	50	19.00	19.25	0.10	+1.33
TAS-50%-500μs	500	50	19.50	18.97	0.13	-2.70
TAS-50%-1000μs	1000	50	20.00	18.94	0.15	-5.32

Table 4.6: Comparison of Theoretical vs. Measured Latency Metrics.

4.3.3 Analysis of Small Control Message (Drive data)

Figure 4.5 shows the results for drive data. In the *Baseline* configuration, the DDS layer latency fluctuates significantly. With TSN enabled, although the mean latency is consistently lower than the *Baseline* under heavy load, there are noticeable outliers. These outlier results from some DDS messages being blocked for one or more cycles waiting for the gate to open. This highlights a limitation of TAS: message transmission is strictly constrained by the cycle period, regardless of priority.

doi:10.6342/NTU202600098

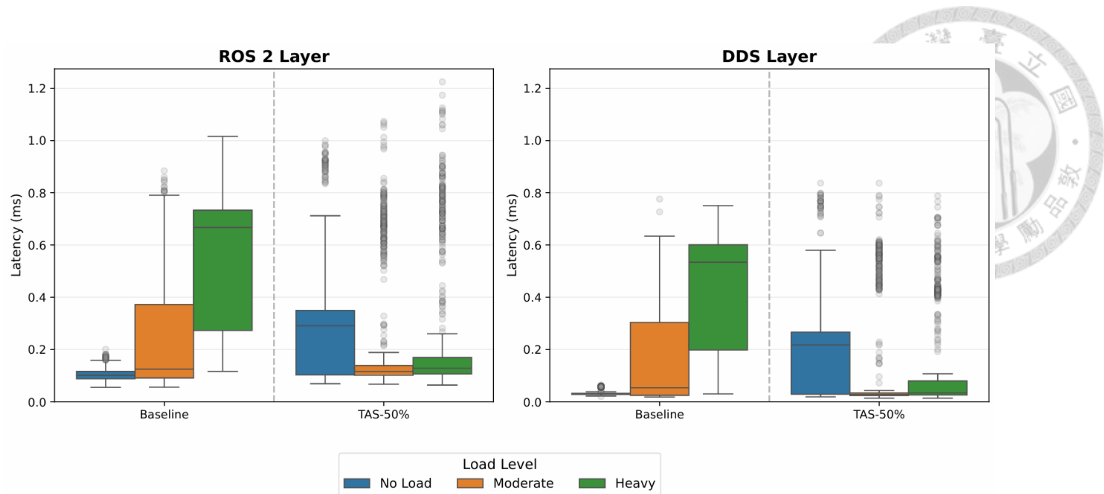


Figure 4.5: Boxplot of drive messages in simple application under various metric types

4.4 Case Study II: Realistic Autonomous Driving Workload

In this section, we use a workload called Rosbag replay simulation provided by Autoware. The workload was originally designed to verify the correctness of the modules for localization and perception. It uses prerecorded sensor data stored in a Rosbag file to simulate a self-driving environment. The execution status of the workload is visualized via RViz panels, which are shown in the Figure 4.6. Note that the workload does not directly control the self-driving car but rather validates the accuracy and reliability of the modules.

doi:10.6342/NTU202600098

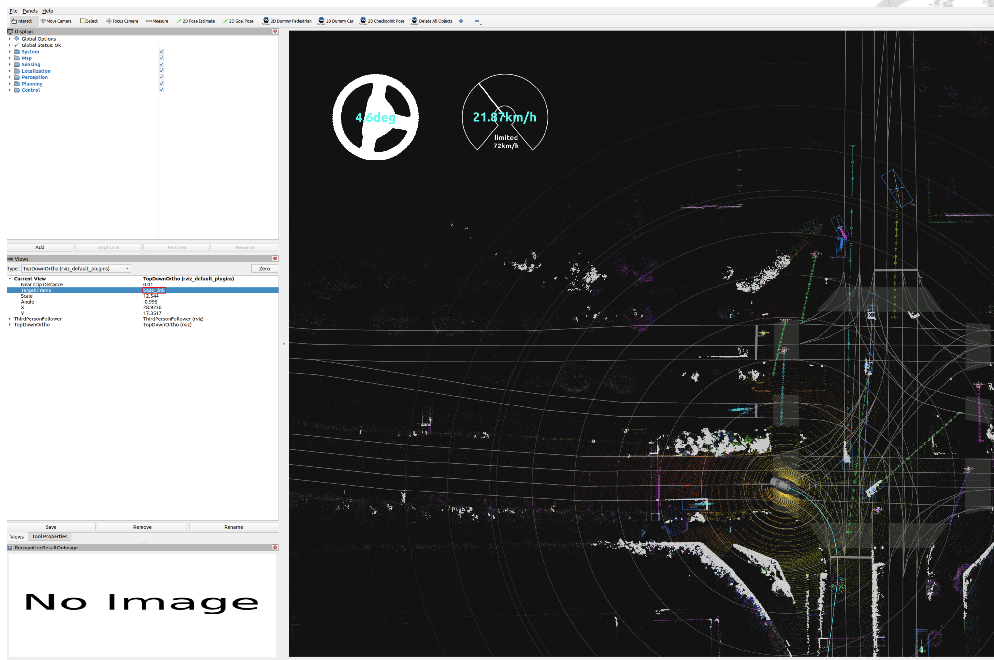


Figure 4.6: The RViz Views panel of rosbag replay simulation.

Although the workload is designed to run on a single machine, we partition it into two parts to evaluate the impact of TSN in a realistic autonomous vehicle machine setup. Figure 4.7 shows the setup for a realistic case study. The first part runs on PC1. It contains only the sensor module in Autoware. Additionally, the Rosbag replay command runs on PC1. The second part runs on PC2, which contains the remaining modules needed for the workload. This primarily includes localization and perception modules, which require sensor data to operate.

doi:10.6342/NTU202600098

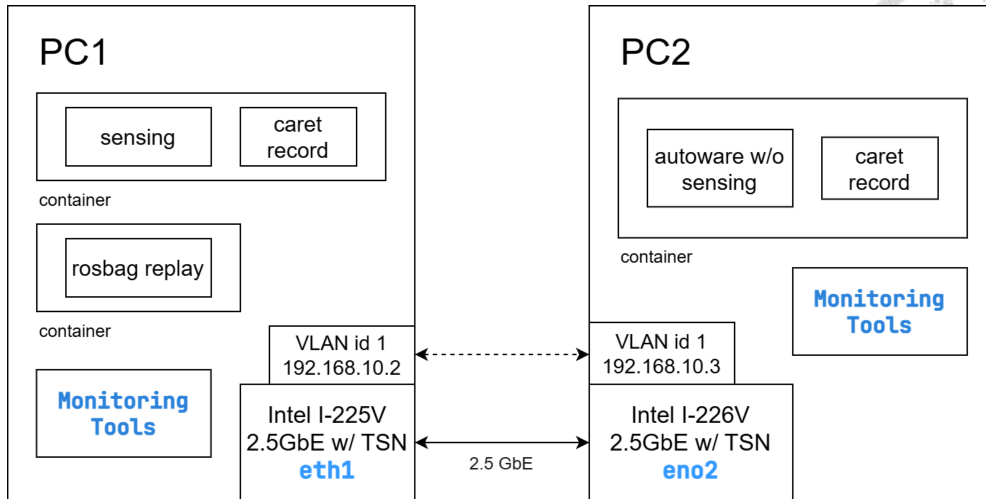


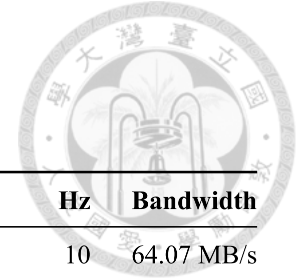
Figure 4.7: System setup for realistic autonomous driving workload.

Since PC1 will only run sensor-related modules and Rosbag, it is responsible for transmitting processed sensor data to other modules that require sensor data. The sensor data used in the case study can be divided into four groups: gnss, imu, lidar, and vehicle status. Every data within their group will have its message size and publish frequency. In addition, the size of the message may vary with the execution time of the workload. As a result, their bandwidth ranges from less than 1KB/s to tens of MB/s, as shown in the Table 4.7. Note that Hz is rounded to the nearest integer, and bandwidth is calculated with the mean message sizes times Hz before being rounded up.

4.4.1 Analysis of LiDAR Data (Pointcloud)

To examine the performance of large message data within realistic autonomous driving workload, we select a data path from the sensor module to the perception module. Specifically, the data path from the node /sensing/lidar/top/velodyne_convert_node to the node /perception/obstacle_segmentation/occupancy_grid_map_outlier_filter. The total number of nodes in the data path is 8 nodes. The data path also has an Ethernet transmis-

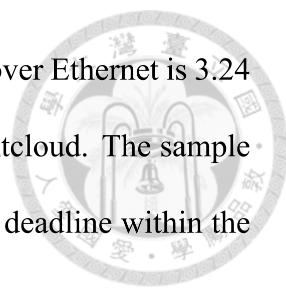
doi:10.6342/NTU202600098



Topic Name	Mean Message Size	Hz	Bandwidth
/sensing/lidar/top/pointcloud_raw	6.61 MB	10	64.07 MB/s
/sensing/lidar/concatenated/pointcloud	3.24 MB	10	31.45 MB/s
/sensing/lidar/top/outlier_filtered/pointcloud	3.00 MB	10	29.11 MB/s
/perception/obstacle_segmentation/pointcloud	1.08 MB	7	7.90 MB/s
/perception/object_recognition/detection/clustering/objects_with_feature	127.29 KB	7	935.33 KB/s
/perception/object_recognition/detection/pointcloud_map_filtered/pointcloud	99.45 KB	7	730.71 KB/s
/perception/occupancy_grid_map/map	88.00 KB	7	646.52 KB/s
/perception/object_recognition/objects	41.76 KB	10	421.22 KB/s
/api/routing/state	12 B	18519	217.01 KB/s
/perception/object_recognition/tracking/objects	15.67 KB	10	158.12 KB/s
/diagnostics	297.48 B	177	51.31 KB/s
/localization/kinematic_state	712 B	46	32.21 KB/s
/localization/acceleration	352 B	46	15.92 KB/s
/sensing/vehicle_velocity_converter/twist_with_covariance	360 B	29	10.31 KB/s
/localization/twist_estimator/twist_with_covariance	360 B	28	9.78 KB/s
/sensing/imu/tamagawa/imu_raw	328 B	29	9.18 KB/s
/sensing/imu/imu_data	320 B	29	8.96 KB/s
/tf	96 B	51	4.82 KB/s
/parameter_events	189.92 B	23	4.23 KB/s
/localization/pose_twist_fusion_filter/pose	72 B	46	3.26 KB/s
/system/emergency/control_cmd	44 B	30	1.29 KB/s
/vehicle/status/velocity_status	36 B	29	1.03 KB/s
/clock	8 B	99	790.51 B/s
/sensing/gnss/pose_with_covariance	360 B	1	372.55 B/s
/initialpose3d	360 B	1	357.47 B/s
/vehicle/status/control_mode	12 B	29	351.87 B/s
/vehicle/status/steering_status	12 B	29	351.87 B/s
/vehicle/status/gear_status	12 B	29	351.87 B/s
/autoware/state	12 B	20	241.28 B/s
/tf_static	3.28 KB	1	121.15 B/s
/system/fail_safe/mrm_state	12 B	10	120.43 B/s
/caret/status	37.65 B	1	13.12 B/s
/api/localization/initialization_state	12 B	1	4.74 B/s

Table 4.7: Description of the message sent over Ethernet in realistic autonomous driving workload.

doi:10.6342/NTU202600098



sion, similar to that in simple applications. The message transmitted over Ethernet is 3.24 MB in average, with the topic name /sensing/lidar/concatenated/pointcloud. The sample might be missed if a communication or computation failed to meet a deadline within the data path.

Figure 4.8 shows the end-to-end and pointcloud latency under various metrics. Similarly to a simple robotic application, the end-to-end latency in the *Baseline* configuration fluctuates significantly as the network load increases. By drilling down into finer metrics, we confirm that it is attributed to the DDS layer. In contrast, the *TAS-50%* configuration is stable across all load levels.

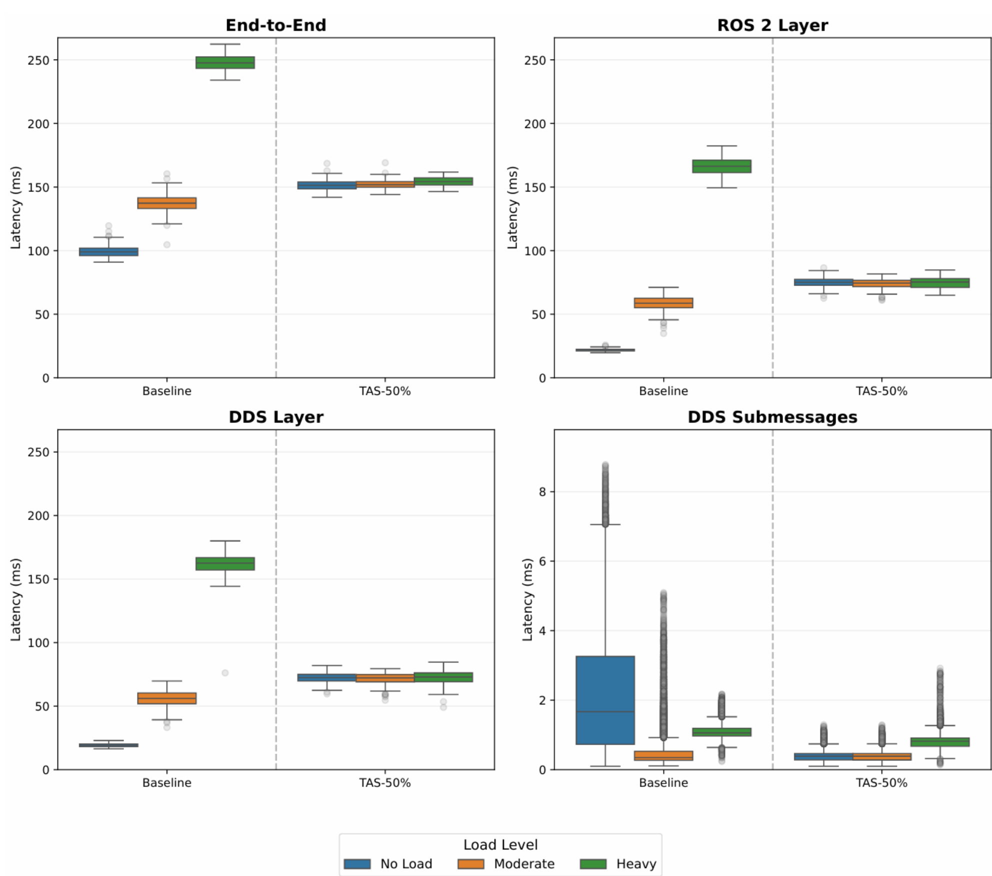
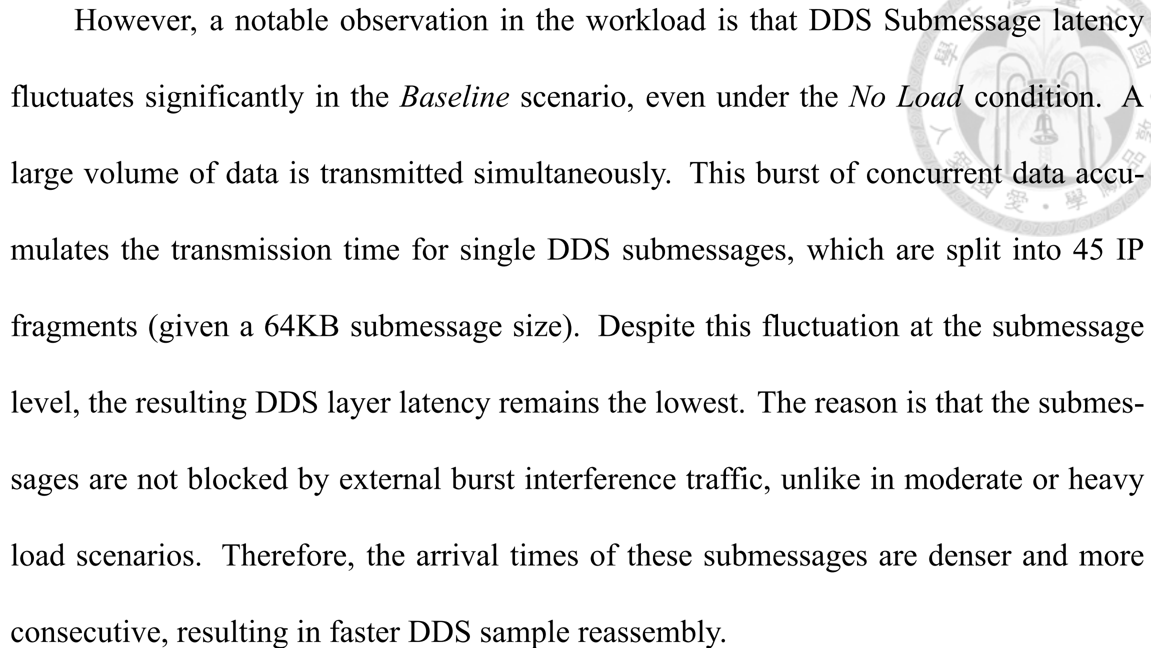


Figure 4.8: Boxplot of pointcloud messages in realistic autonomous driving workload case study under various metric types.

doi:10.6342/NTU202600098



However, a notable observation in the workload is that DDS Submessage latency fluctuates significantly in the *Baseline* scenario, even under the *No Load* condition. A large volume of data is transmitted simultaneously. This burst of concurrent data accumulates the transmission time for single DDS submessages, which are split into 45 IP fragments (given a 64KB submessage size). Despite this fluctuation at the submessage level, the resulting DDS layer latency remains the lowest. The reason is that the submessages are not blocked by external burst interference traffic, unlike in moderate or heavy load scenarios. Therefore, the arrival times of these submessages are denser and more consecutive, resulting in faster DDS sample reassembly.

4.4.2 Analysis of Vehicle Status data

Figure 4.9 shows the performance of small messages in the realistic workload. Similar to the first workload, the data flows from the vehicle status broadcast node to the monitor node across two separate PCs with the topic name /vehicle/status/velocity_status. The results parallel those of the Simple Application, where we observe outliers whenever TSN is enabled. Additionally, the mean latency increases under a heavy network. We suspect that this is caused by hardware resource contention on the NIC, involving the DMA channel or PCIe bus. The high-demand data processing of the real-world workload and the heavy network load likely result in minor latency penalties due to waiting for DMA completion or PCIe bus availability.

doi:10.6342/NTU202600098

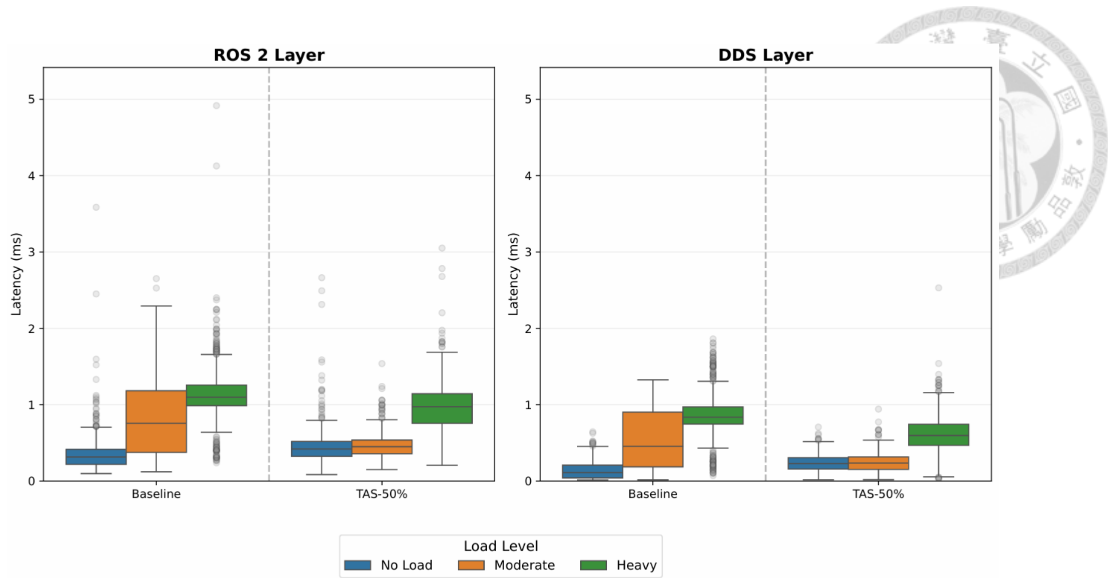


Figure 4.9: Boxplot of velocity status data messages in a realistic autonomous driving workload case study under various metrics.

4.4.3 Online Monitoring Visualization UI

All evaluation data can be monitored online via Grafana UI when the workload is executing. Note that the visualization latency depends on the specific metrics to display. Figure 4.10 shows the Grafana UI displaying the DDS message payload size sent over the Ethernet against system time. The dashboard captures three operation phases: first, we initialize Autoware modules except sensing on PC2; second, we launch the Autoware sensing module; and finally, we run Rosbag replay the published sensor data until completion. Figure 4.11 shows the Grafana UI displaying aggregated data from two sources. The panel shows the latency of a DDS message derived from *ddshark*, and the panel below shows the network bandwidth for every network interface on the system, which is collected from *node_exporter*.

doi:10.6342/NTU202600098

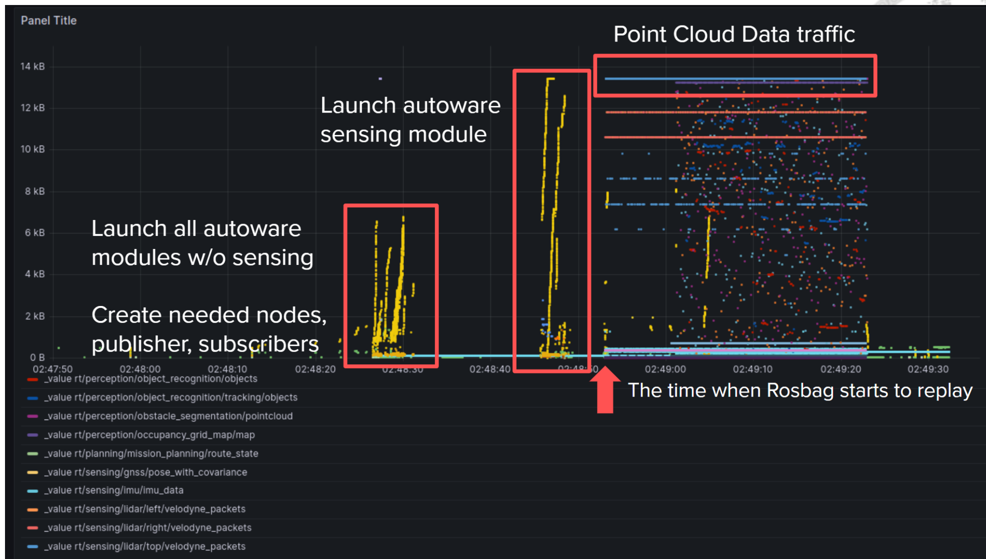


Figure 4.10: Grafana UI dashboard that monitors realistic autonomous driving workload, with x-axis as system time and y-axis as DDS message payload.

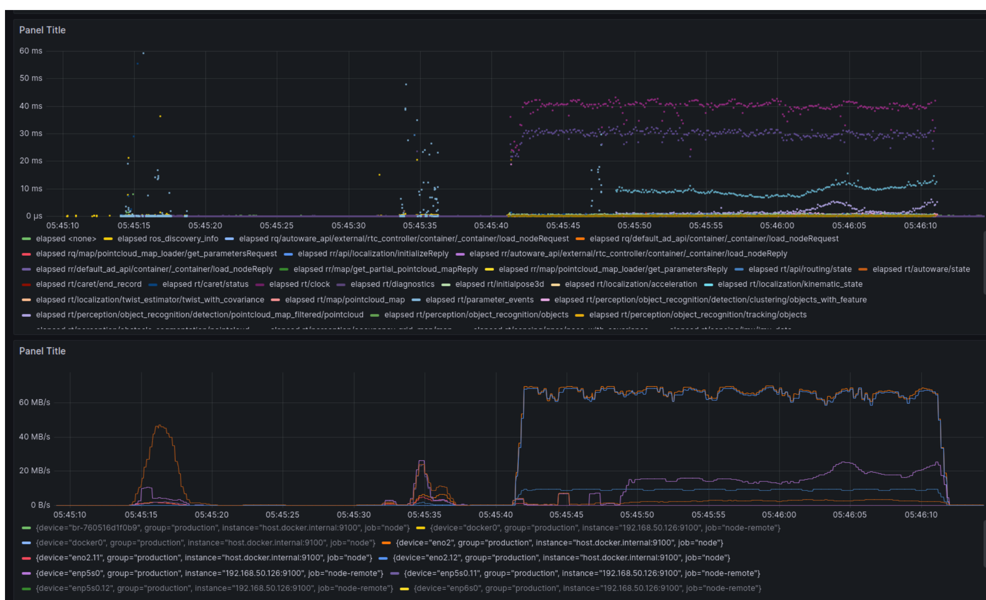


Figure 4.11: Grafana UI dashboard that monitors realistic autonomous driving workload, with the x-axis as system time and the y-axis. The y-axis of the above panel shows the latency of a complete DDS message, whereas the y-axis of the below panel shows the network bandwidth in system time.

doi:10.6342/NTU202600098

Chapter 5 Conclusion and Future Work

This paper presents a top-down performance profiling and online monitoring framework for distributed autonomous vehicle systems built on ROS 2, DDS, and TSN. By correlating application-level end-to-end latency with middleware- and network-level metrics, the proposed framework enables fine-grained inspection of performance bottlenecks in domain- and zonal-based SDV architectures. Evaluation using both a simplified robotic workload and a real-world Autoware-based autonomous driving workload demonstrates that TSN, specifically Time-Aware Shaper (TAS), effectively stabilizes end-to-end latency for bandwidth-intensive sensor data under network contention, while introducing cycle-level trade-offs for control messages. The proposed *ddshark* tool enables low-overhead, system-wide observability of DDS traffic, and supports practical online performance monitoring and diagnosis.

The tool used in the online monitoring framework is released as open source and is publicly available at: https://github.com/NEWSLabNTU/ddshark. We believe that our tool will be useful to developers for analyzing the performance of ROS 2 applications and Autoware in distributed environments. Furthermore, the proposed top-down analysis

doi:10.6342/NTU202600098

methodology provides a reference for profiling latency in the real-time distributed systems where network determinism is critical.

Future work will focus on improving timing accuracy by enabling hardware-based timestamping in libpcap capturing. This enables accurate timestamps for the time when packets enter or exit the system. We also plan to extend *ddshark* to support header-only packet tracing to further reduce overhead, and to incorporate more TSN NIC features, such as frame preemption, to enable finer-grained evaluation of mixed-criticality traffic.

doi:10.6342/NTU202600098

References

[1] Grafana UI. https://grafana.com/grafana/dashboards.

[2] node_exporter. https://github.com/prometheus/node_exporter.

[3] SOAFEE. https://www.soafee.io/.

[4] T. Betz et al. How fast is my software? latency evaluation for a ROS 2 autonomous driving software. In 2023 IEEE Intelligent Vehicles Symposium (IV), 2023.

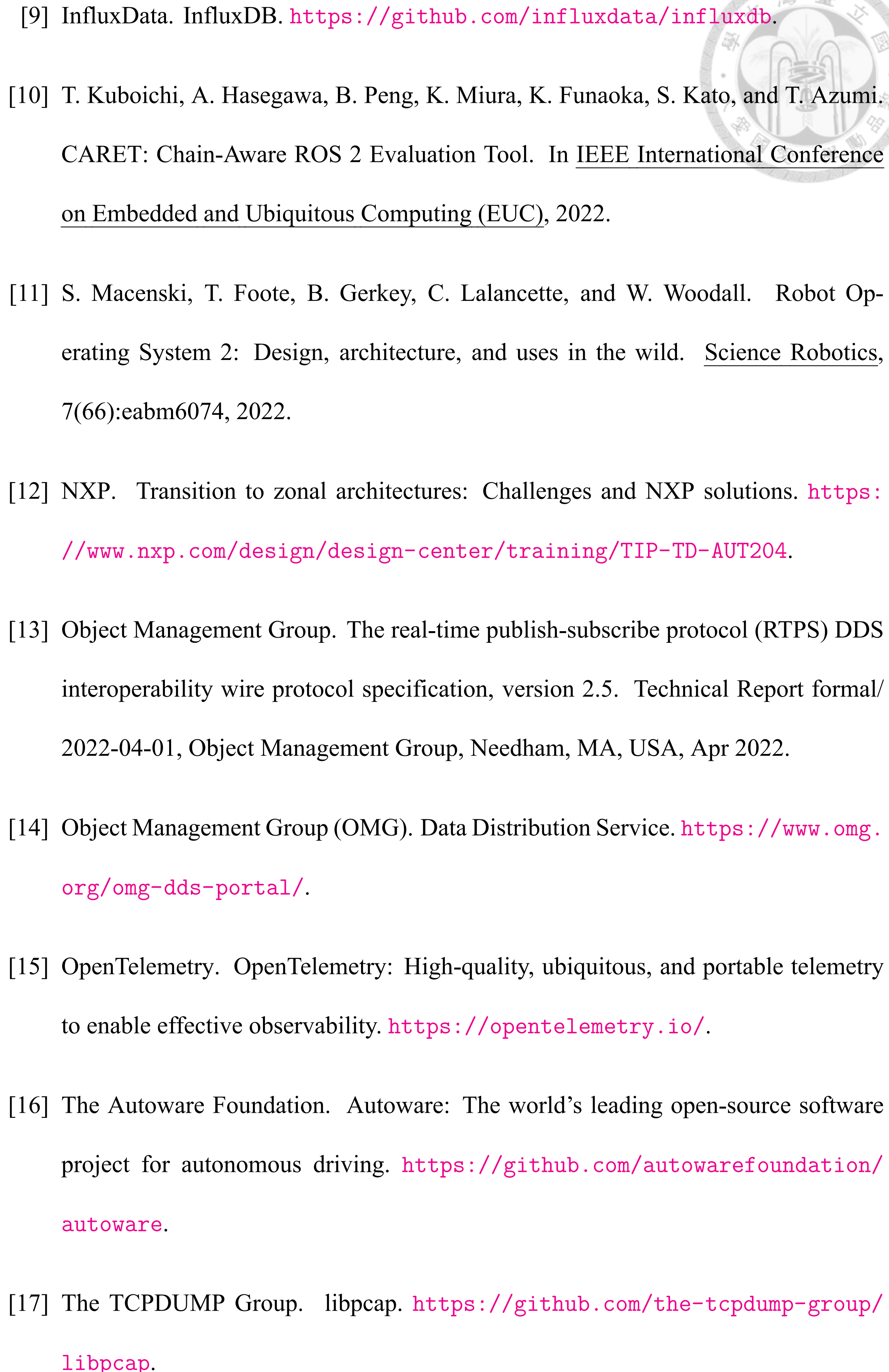
[5] C. Bédard, I. Lütkebohle, and M. Dagenais. ros2 tracing: Multipurpose low-overhead framework for real-time tracing of ROS 2. IEEE Robotics and Automation Letters, 7(3):6511–6518, 2022.

[6] F. Fejes, P. Antal, and M. Kerekes. The TSN building blocks in Linux. arXiv preprint arXiv:2211.14138, 2022.

[7] E. Guijarro Cameros and L. Chan. How DDS and TSN are driving interoperability and performance in automotive systems. ATZelectronics worldwide, 17(10):52–55, Oct 2022.

[8] IEEE Time-Sensitive Networking (TSN) Task Group. IEEE time-sensitive networking (TSN) task group. https://1.ieee802.org/tsn/.

doi:10.6342/NTU202600098



[9] InfluxData. InfluxDB. https://github.com/influxdata/influxdb.

[10] T. Kuboichi, A. Hasegawa, B. Peng, K. Miura, K. Funaoka, S. Kato, and T. Azumi. CARET: Chain-Aware ROS 2 Evaluation Tool. In IEEE International Conference on Embedded and Ubiquitous Computing (EUC), 2022.

[11] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall. Robot Operating System 2: Design, architecture, and uses in the wild. Science Robotics, 7(66):eabm6074, 2022.

[12] NXP. Transition to zonal architectures: Challenges and NXP solutions. https://www.nxp.com/design/design-center/training/TIP-TD-AUT204.

[13] Object Management Group. The real-time publish-subscribe protocol (RTPS) DDS interoperability wire protocol specification, version 2.5. Technical Report formal/2022-04-01, Object Management Group, Needham, MA, USA, Apr 2022.

[14] Object Management Group (OMG). Data Distribution Service. https://www.omg.org/omg-dds-portal/.

[15] OpenTelemetry. OpenTelemetry: High-quality, ubiquitous, and portable telemetry to enable effective observability. https://opentelemetry.io/.

[16] The Autoware Foundation. Autoware: The world's leading open-source software project for autonomous driving. https://github.com/autowarefoundation/autoware.

[17] The TCPDUMP Group. libpcap. https://github.com/the-tcpdump-group/libpcap.

doi:10.6342/NTU202600098