# DESIGN AND IMPLEMENTATION OF A CONTAINERIZED MONITORING SYSTEM FOR PUBLISH/SUBSCRIBE NETWORKS

by

Ali Mehran

B.S., Computer Engineering, Urmia University of Technology, 2016

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Department of Computer Engineering

Boğaziçi University

2024

# ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Professor Dogan Ulus, for his invaluable guidance, support, and encouragement during both my master's studies and the preparation of this thesis. His expertise, insights, and intuitions have been instrumental in helping me understand novel concepts and navigate the complexities of my research.

I also wish to acknowledge all my professors in the Computer Engineering Department of Boğaziçi University. Their high levels of knowledge and the valuable insights they provided both in classes and in our daily conversations have significantly contributed to my education and this work.

Additionally, I want to express my sincere thanks to my family for their endless devotion, support, and patience throughout this journey. Their encouragement and belief in my abilities have been a constant source of motivation.

Lastly, I want to thank my friends Mustafa Mert Sengul, Halit Ozsoy, and Ege Baran Yalcin, who are like brothers to me. They have always been a tremendous support, sharing their bright ideas in solving difficult problems and helping me manage my time effectively so I could focus on my studies.

# ABSTRACT

## DESIGN AND IMPLEMENTATION OF A CONTAINERIZED MONITORING SYSTEM FOR PUBLISH/SUBSCRIBE NETWORKS

This work presents the design and implementation of a functional monitoring system for robotic applications. The monitoring system comprises components, tools, and processes to detect anomalies in the robot's behavior, alert users, and provide performance feedback. It also offers predictive maintenance by analyzing behavior over time and suggesting necessary repairs or parameter tunings. Developed using Robot Operating System (ROS), the monitoring system observes the behavior of a vehicle robot inside a simulation. Monitoring package includes tools, libraries, and processes to provide feedback on the robot's internal state (speed, actuator, and sensor status) and external conditions (environmental data, position, obstacles, other robots). It generates alarms, alerts, and notifications as needed, providing reports to enable robotic operations managers and operators to take appropriate action. The system is designed to be agile and efficient, capable of dynamically scaling according to domain-specific demands. Additionally, this work investigates the integration of runtime verification containers within publish/subscribe networks, which are crucial for applications like the Internet of Things (IoT) and robotics. Runtime verification techniques ensure the correctness of these highly dynamic and large-scale networks. By leveraging modern containerization technology, runtime verification tools are deployed to monitor message flow, ensuring real-time verification of network behavior. Each runtime verification container operates alongside other containerized participants, monitoring the system's integrity. This approach is benchmarked through several experiments and a real-world case study from the software-defined vehicle domain, demonstrating its performance.

# ÖZET

## YAYIN/ABONE AĞLARI İÇİN KONTEYNERLEŞTİRİLMİŞ BİR İZLEME SİSTEMİNİN TASARIMI VE UYGULANMASI

Bu çalışma, robotik uygulamalar için işlevsel bir izleme sisteminin tasarım ve uygulamasını sunmaktadır. İzleme sistemi, robotun davranışındaki anormallikleri tespit etmek, kullanıcılara bildirimde bulunmak ve performans geri bildirimi sağlamak için bileşenler, araçlar ve süreçler içermektedir. Ayrıca, davranışı zaman içinde analiz ederek ve gerekli onarımlar veya parametre ayarlamaları önererek kestirimci bakım sunar. Robot İşletim Sistemi (ROS) kullanılarak geliştirilen izleme sistemi, bir simülasyon içindeki bir araç robotunun davranışını gözlemlemektedir. İzleme paketi, robotun iç durumu (hız, aktüatör ve sensör durumu) ve dış koşullar (çevresel veriler, konum, engeller, diğer robotlar) hakkında geri bildirim sağlamak için araçlar, kütüphaneler ve süreçler içermektedir. Gerektiğinde alarmlar, uyarılar ve bildirimler üretir, robotik operasyon yöneticileri ve operatörlerinin uygun aksiyonlar alabilmesi için raporlar sunar. Sistem, alan spesifik taleplere göre dinamik olarak ölçeklenebilir şekilde çevik ve verimli olacak şekilde tasarlanmıştır. Ayrıca, bu çalışma, Nesnelerin İnterneti (IoT) ve robotik gibi uygulamalar için kritik olan yayınla/abone ol ağı içinde çalışma zamanı doğrulama konteynerıların entegrasyonunu araştırmaktadır. Çalışma zamanı doğrulama teknikleri, bu son derece dinamik ve büyük ölçekli ağların doğruluğunu sağlar. Modern konteyner teknolojisini kullanarak, çalışma zamanı doğrulama araçları mesaj akışını izlemek için konuşlandırılır, ağ davranışının gerçek zamanlı doğrulanmasını sağlar. Her çalışma zamanı doğrulama konteynerı, sistemin bütünlüğünü izleyerek diğer konteyner katılımcılarıyla birlikte çalışır. Bu yaklaşım, çeşitli deneyler ve bir gerçek dünya vaka çalışması ile değerlendirilmiştir.

# TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . iii
ABSTRACT . . . iv
ÖZET . . . v
LIST OF FIGURES . . . viii
LIST OF TABLES . . . x
LIST OF ACRONYMS/ABBREVIATIONS . . . xi
1. INTRODUCTION . . . 1
2. BACKGROUND . . . 5
2.1. Design and Implementation of a Robotic Software . . . 5
2.2. Publish/Subscribe Networks . . . 9
2.3. Modern Container Technology . . . 12
3. RELATED WORK . . . 14
3.1. Design and Implementation of a Robotic Software . . . 14
3.1.1. Design of Robotic Software: . . . 14
3.1.2. Implementation of Robotic Software: . . . 15
3.2. Runtime Verification Containers for Publish/Subscribe Networks . . . 17
4. CONTAINERIZATION . . . 19
4.1. Target Architectures . . . 19
4.2. Container Networking Modes . . . 22
4.3. Container Orchestration . . . 22
4.4. Multi-stage Container Build . . . 23
5. CONTAINERIZED NETWORK PERFORMANCE EVALUATION . . . 25
5.1. Baseline Experiments in Containers . . . 26
5.2. Runtime Verification Experiments in Containers . . . 34
6. AUTOWARE CASE STUDY . . . 39
6.1. Monitoring Node Participant . . . 41
6.2. Evaluating Deployment Setups . . . 44
7. CONCLUSION . . . 45

REFERENCES . . . . . 48

# LIST OF FIGURES

Figure 1.1. Runtime Verification Containers on a Publish/Subscribe Network. 2

Figure 2.1. Comparison of ROS 1 and ROS 2 Architectures. . . . . 8

Figure 2.2. General Autoware Design. . . . . 9

Figure 4.1. Single Pub/Sub Network for Benchmarks. . . . . 20

Figure 4.2. Single Pub/Sub Network for Apps. . . . . 20

Figure 4.3. Multi Pub/Sub Network for Benchmarks. . . . . 21

Figure 4.4. Multi Pub/Sub Network for Apps. . . . . 21

Figure 5.1. Single Pub/Sub Network for Benchmarks. . . . . 27

Figure 5.2. Multi Pub/Sub Network for Benchmarks. . . . . 27

Figure 5.3. CycloneDDS Throughput(msg/s). . . . . 29

Figure 5.4. CycloneDDS Throughput(bit/s). . . . . 30

Figure 5.5. Zenoh Throughput(msg/s). . . . . 30

Figure 5.6. Zenoh Throughput(bit/s). . . . . 31

Figure 5.7. CycloneDDS Latency. . . . . 31

Figure 5.8. Zenoh Latency. . . . . 32

Figure 5.9. DDS-to-Zenoh Throughput(msg/s). . . . . 32

Figure 5.10. DDS-to-Zenoh Throughput(bit/s). . . . . 33

Figure 5.11. DDS-to-Zenoh Latency. . . . . 33

Figure 5.12. Reelay-DDS Latency Performance. . . . . 36

Figure 5.13. Reelay-Zenoh Latency Performance. . . . . 36

Figure 6.1. Container Deployment Setup Using a Single Host Machine. . . . . 40

Figure 6.2. Container Deployment Setup Distributed Over Two Host Machines. 41

# LIST OF TABLES

Table 5.1. Runtime verification container performance on pub/sub networks. . 37

Table 6.1. Evaluation of deployment setups. . . . . 43

# LIST OF ACRONYMS/ABBREVIATIONS

| | |
|---|---|
| API | Application Programming Interface |
| CDR | Common Data Representation |
| CPU | Central Processing Unit |
| DDS | Data Distribution Service |
| GPU | Graphics Processing Unit |
| IDE | Integrated Development Environment |
| IP | Internet Protocol |
| IoT | Internet of Things |
| LTL | Linear Temporal Logic |
| MQTT | Message Queuing Telemetry Transport |
| MTL | Metric Temporal Logic |
| OCI | Open Container Initiative |
| OMG | Object Management Group |
| PREM | Predictable Execution Model |
| pub | Publisher |
| QoS | Quality of Service |
| RAM | Random Access Memory |
| ROS | Robot Operating System |
| RV | Runtime Verification |
| SAE | Society of Automotive Engineers |
| SDV | Software Defined Vehicle |
| SMT | Satisfiability Modulo Theories |
| sub | Subscriber |
| TCP | Transmission Control Protocol |
| UDP | User Datagram Protocol |
| WCET | Worst-Case Execution Time |

# 1. INTRODUCTION

The development of robots is a growing trend with applications across many domains, from domestic use to industrial automation. These robots vary in complexity depending on the tasks they execute and the environments in which they operate. A functional monitoring system, a key component of a robotic system, monitors robot performance and safety. These systems also differ in complexity, directly correlating with the composition of the robotic system and its mission. Traditional monitoring approaches do not satisfy the requirements of complex robotic systems that demand stringent correctness, agility, efficiency, reliability, and fault tolerance. These specifications impact the design and implementation phases of a functional monitoring system, necessitating the selection and integration of appropriate hardware, software, and monitoring methodologies to ensure the desired performance and reliability.

Self-driving cars exemplify highly complex systems in safety-critical applications due to their dangerous missions in environments that involve human agents. Thus, they require sophisticated design and implementation practices for a real-time monitoring system, a cumbersome task due to the system's inherent hardware, software, and algorithmic complexities. Furthermore, a good monitoring system must be scalable and uniform. Uniformity eliminates variations in existing tools and programming languages used for the monitoring system, achieving a uniform framework that simplifies further development and debugging. Scalability is essential because, unlike traditional robots, self-driving car operations are not based on simple, predefined maneuvers and perception tasks. Such systems must dynamically scale up or down depending on domain-specific requirements.

Modern software applications are increasingly distributed, with components located on edge devices and in the cloud. These components interact asynchronously with each other and external services, necessitating robust and reliable communication mechanisms. A common approach to facilitate communication within these dis-

tributed systems is the use of a publisher/subscriber (pub/sub) network architecture. In pub/sub networks, publishers disseminate messages, often containing data or event notifications, to a pool of subscribers. Subscribers express interest in specific categories of information by subscribing to relevant topics. While this loosely coupled communication pattern enables scalability and flexibility, it also introduces challenges in ensuring correct operation, maintaining software across iterations, and managing the complexity of distributed system development.

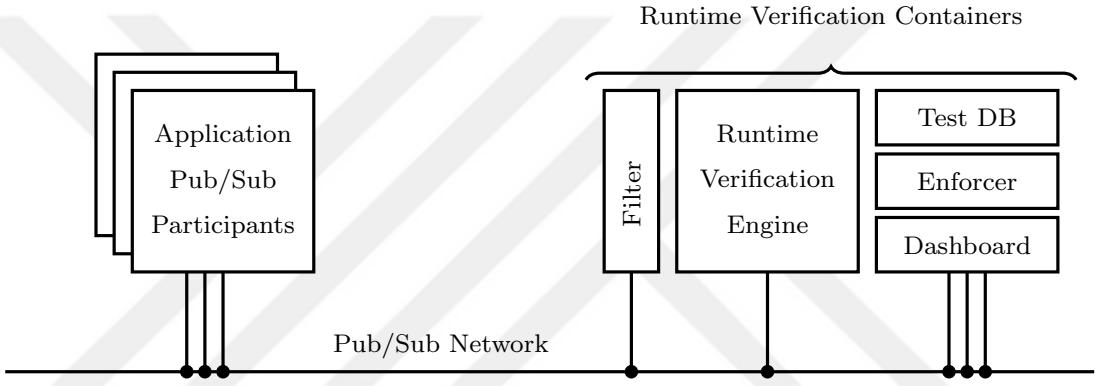



Figure 1.1. Runtime Verification Containers on a Publish/Subscribe Network.

Runtime Verification (RV) is an appealing technology for dynamically verifying and monitoring pub/sub architectures. Its key advantage is its ability to integrate into the pub/sub model by functioning as both a subscriber and a publisher. As a subscriber, online RV tools can continuously monitor message flows by subscribing to relevant topics, allowing for real-time analysis of message content and behavior, ensuring adherence to predefined properties and requirements. Additionally, RV tools can act as publishers to detect property violations and anomalies at runtime, facilitating proactive error detection and enforcement. This dual subscriber-publisher capability enables comprehensive monitoring and analysis, ensuring system-level reliability and correctness, especially in dynamic environments. However, deploying and maintaining pub/sub participants, including RV participants, is a complex task. This presents a significant challenge for creating a streamlined RV framework for pub/sub networks.

Containerization of pub/sub network and runtime verification participants offers an attractive solution for overcoming deployment and maintenance challenges. Containers simplify deployments, enhance portability, enable efficient scaling, and facilitate rapid updates, contributing to a more reliable, adaptable, and efficient communication infrastructure. Therefore, in this work, we propose and study runtime verification containers. Figure 1.1 illustrates RV containers encapsulating runtime verification functionality and receiving messages from subscribed topics. Runtime monitors can be designed to listen to message synchronizers and publish verdict messages to central test results databases or dashboards. However, the performance of containerized networks is critical for the successful implementation of this approach. Therefore, in this work, alongside the design and implementation of functional monitoring we study runtime verification containers from a performance perspective, making the following contributions in general:

- In the course of this work, we first designed and implemented a real-time functional monitoring system specifically tailored for overseeing the behavior of an autonomous vehicle within a simulated real-world environment. This monitoring system is capable of providing real-time feedback on both the internal operational states of the vehicle and the external environmental conditions in which it operates. By aligning with user-defined requirements and specifications, this system ensures that the monitoring information provided is precisely suited for testing the system under specific scenarios or for verifying the correctness of particular software modules of the vehicle.
- Building upon this, we developed a containerized approach for integrating runtime verification applications within publish/subscribe networks, facilitating seamless communication with other participants in our distributed ecosystem. To thoroughly evaluate the performance of these runtime verification tools, we conducted a series of experiments across varying levels of complexity, starting with baseline performance evaluations.
- Our initial experiments involved baseline performance evaluations of containerized pub/sub network participants, aimed at measuring their inherent networking

capabilities within a minimalistic architecture. This setup was intentionally designed to exclude additional participants, such as runtime verification tools or other applications, that could introduce networking overhead. In these experiments, we utilized two widely adopted pub/sub networking protocols commonly used in robotic applications and explored various container runtime networking modes to ensure a comprehensive assessment.

- After establishing baseline performance, we proceeded to containerize a specification-based runtime verification tool and re-evaluated its networking performance within the pub/sub network. This evaluation was critical for understanding how the inclusion of runtime verification impacts the overall ecosystem's networking efficiency. Similar to the baseline tests, we employed two different pub/sub communication protocols and explored various container runtime networking modes to ensure a thorough assessment of the runtime verification tool's performance within the pub/sub framework.
- Finally, we extended our evaluation to a real-world case study centered on autonomous driving. In this scenario, we integrated a containerized runtime verification tool with two additional participants: a containerized autonomous driving software stack and a containerized photo-realistic simulation. These components operated within two distinct pub/sub communication protocols. The primary objective of this scenario was to evaluate the performance of the runtime verification container within a system of computationally intensive processes, all working together to execute a realistic autonomous driving scenario. To achieve a thorough assessment, we carried out these experiments across two different hardware configurations: one where all containerized participants were deployed on a single host, and another where they were distributed across two separate hosts. In both configurations, we analyzed the effects of various container runtime networking modes on the overall system performance.

# 2. BACKGROUND

In this chapter, we provide a general overview of the design and implementation of robotic software and offer background information on publish/subscribe networks and software containerization.

## 2.1. Design and Implementation of a Robotic Software

Robotic systems are increasingly prevalent across a variety of domains, from domestic environments to industrial automation. These systems range in complexity based on the tasks they perform and the environments in which they operate. A fundamental component of any robotic system is a functional monitoring system, which ensures the robot's performance and safety. The complexity of these monitoring systems is directly correlated with the complexity of the robotic system and its mission.

Not all robots are autonomous; many are designed to perform specific, repetitive tasks under human supervision. However, as the field of robotics evolves, there is a growing demand for autonomous robots—machines capable of performing tasks without human intervention. Autonomous robots can operate independently, making decisions based on their perception of the environment. This autonomy introduces significant challenges in design, implementation, and verification, particularly for safety-critical applications such as self-driving cars.

For self-driving cars, levels of autonomy are defined by the Society of Automotive Engineers (SAE), known as SAE levels. The definitions of autonomy levels for self-driving cars are as follows:

- Level 0: This level does not contain any automation and the driver is fully responsible for all aspects of driving.
- Level 1: In this level the human driver is responsible for most of the driving tasks,

but the vehicle can assist with steering and acceleration/deceleration. Adaptive cruise control and lane-keeping are examples of this level of autonomy.

- Level 2: This level introduces partial automation in which the vehicle can control steering and acceleration/deceleration commands simultaneously under specific conditions, but the human driver must remain alert and monitor the environment. Tesla's Autopilot technology [1] is an example for this level of autonomy.
- Level 3: In this level, the vehicle can completely operate autonomously but the human driver must be ready to take over when necessary. As an example we can mention Audi Traffic Jam Pilot [2].
- Level 4: This level indicates high automation in which the vehicle can perform all driving tasks and also monitor the environmental factors under certain conditions and environments; for example, geofenced areas. In the broadest sense, geofencing is simply a virtual boundary around a physical area. In the world of self-driving cars, it describes a crucial subset of the operational design domain — the geographic region where the vehicle is functional. Other than these specific conditions and areas, human driver intervention is needed. Waymo's autonomous taxis [3] can operate in level 4 only in limited areas.
- Level 5: This level represents full automation in which the vehicle can perform all autonomy tasks under any condition and environment without human intervention. Full autonomous vehicles are not available for the public yet.

Such vehicles with high levels of autonomy inherit two major issues regarding test and verification of such applications. First, because these robots operating dangerous missions in human scale, environments that also include humans, they are considered as safety critical applications with high correctness requirements that must be proved to hold. Secondly, because such robots operate in complex conditions, their functional objectives are much higher than traditional autonomy objectives that only rely on the existence of predefined maneuvers or static environments which is satisfiable via low scene perception and simple control algorithms. Thus require high performance hardware/software components and more intelligent algorithms like neural networks which in turn makes it hard to analyze these components, therefore complicating the

implementation phase and also pushing the limits of traditional functional correctness checking tools and theories that complicates verification task.

In this work, we utilize widely-used and well-tested open-source frameworks to address the complexities of the implementation phase. We employ the Robot Operating System (ROS) [4] for developing our robotic software. ROS is a flexible framework that includes essential tools and libraries, simplifying the creation of complex and reliable software for various robotic platforms. Although it is not an operating system in the traditional sense, ROS provides necessary services such as hardware abstraction, low-level device control, base functionality implementations, message-passing among processes, and package management. By offering standard functionalities for accessing hardware devices, component communication, and visualization, ROS establishes a solid foundation for developing standard robot applications.

ROS is available in two versions of ROS 1 and ROS 2. As shown in Figure 2.1, these versions have significant differences. ROS 1 uses custom protocols, TCPROS and UDPROS, instead of the standard TCP/IP and UDP protocols, which are not optimized for real-time performance and scalability. In contrast, ROS 2 adopts the Data Distribution Service (DDS) [5], an industry-standard communication protocol for real-time systems. By incorporating DDS for decentralized discovery and communication among nodes, ROS 2 eliminates the need for the central master node (ROSCORE) used in ROS 1. This removal increases reliability by eliminating a single point of failure and reduces latency that a central communication point can introduce.

ROS 1 employs the Nodelet API, enabling multiple algorithms to run within the same process. ROS 2 replaces this with intra-process communication, providing better control over scheduling and prioritization, which significantly reduces latency. Additionally, ROS 2 introduces Quality of Service (QoS) policies, allowing control over communication behavior regarding latency, reliability, and priority. ROS 2 also supports a wider range of operating systems. Due to these advantages, we chose ROS 2 as the primary framework for implementing our pipeline.

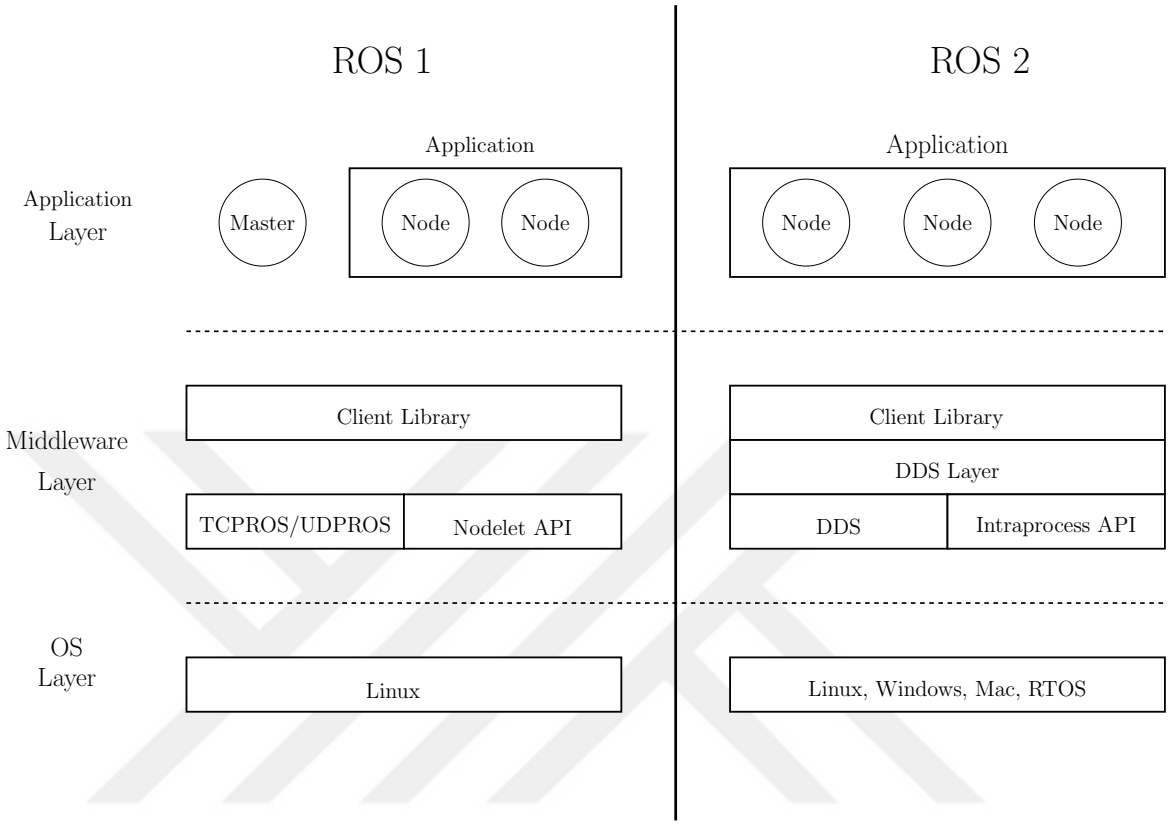



Figure 2.1. Comparison of ROS 1 and ROS 2 Architectures.

The focal point of our work is to demonstrate the application of runtime monitoring and verification within a real-world autonomous driving stack. We chose Autoware [6] as our framework because it is a comprehensive, open-source software stack built on the Robot Operating System, which serves as the foundation for self-driving car software development. As shown in Figure 2.2, Autoware includes a wide range of components and functionalities essential for autonomous driving, such as sensing, localization, perception, planning, and control. These well-defined functionalities provide a structured framework for autonomy, significantly reducing the complexity of configuring and tuning the interdependencies of the system's architecture.

Autoware's modular design, as shown in Figure 2.2, is a key advantage that allows developers to selectively use only the components needed for their specific applications and seamlessly integrate them into custom solutions. This modularity not only en-

hances flexibility but also supports scalability and adaptability, making Autoware an ideal choice for various autonomous driving projects. Additionally, being built on ROS 2, Autoware leverages the robust communication infrastructure, tools, and libraries of ROS, further enhancing its capability and reliability in real-world applications. The extensive community support and continuous contributions to the Autoware project ensure it remains at the forefront of autonomous driving technology, providing developers with cutting-edge tools and resources.

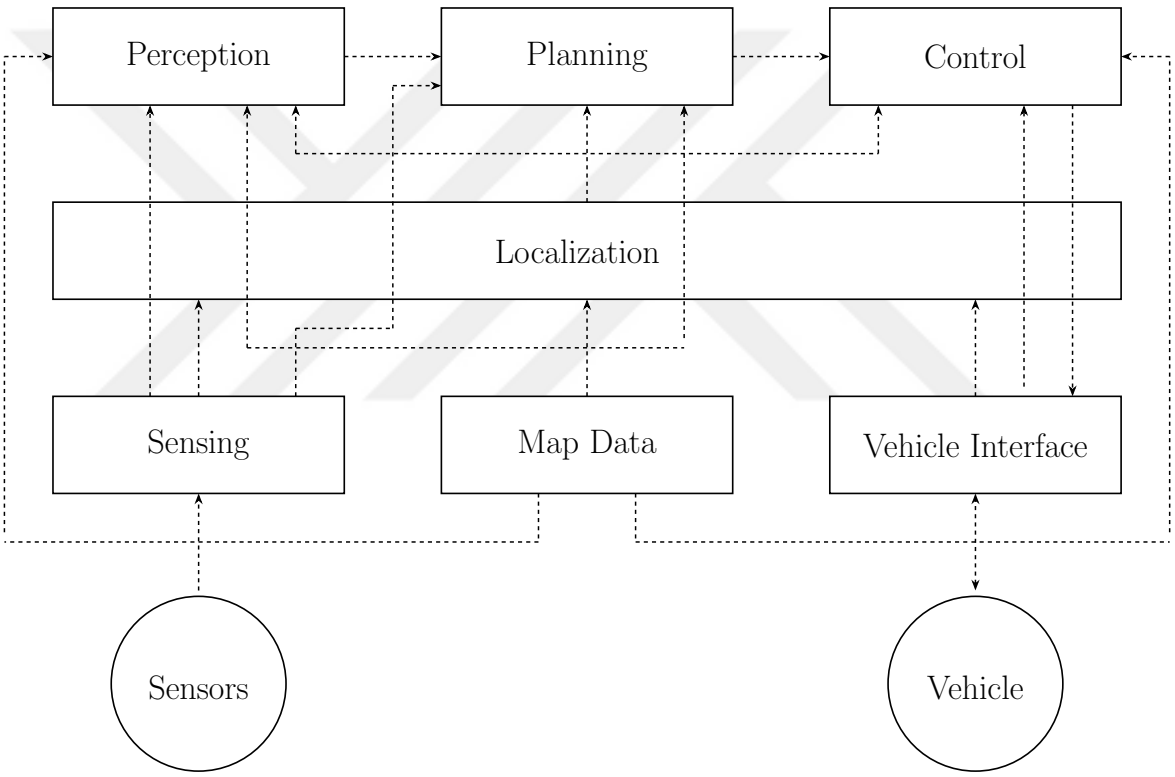



Figure 2.2. General Autoware Design.

## 2.2. Publish/Subscribe Networks

In this work, we primarily focus on DDS [5] and Zenoh [7] communication protocols, which are based on the publish/subscribe paradigm and are widely used for implementing complex real-time systems, particularly in robotics.

Data Distribution Service (DDS) is an open, data-centric publish/subscribe communication protocol for real-time and embedded systems, standardized by the Object Management Group (OMG). Unlike traditional message-passing systems, DDS features a fully decentralized architecture, allowing network participants to automatically discover each other through a dynamic discovery service, thus eliminating the need for a central broker. This decentralized approach is particularly advantageous for systems that require high availability and reliability, as it removes the single point of failure associated with central brokers. Each DDS participant, whether a publisher, subscriber, or both, independently maintains the state of the system, leading to a robust and resilient communication network.

DDS supports a wide range of Quality of Service (QoS) policies, which enable developers to specify the behavior of data flows in terms of reliability, latency, durability, and resource usage. These QoS policies allow DDS to address the stringent requirements of real-time systems, such as those found in autonomous vehicles and industrial automation. The protocol's flexibility in adjusting these parameters ensures that the communication infrastructure can be adjusted to meet specific application needs, providing the right balance between performance and resource consumption. Another key feature of DDS is its use of the Common Data Representation (CDR) format for serializing messages. CDR ensures that data is efficiently encoded and decoded, enabling high-throughput and low-latency communication. The interoperability of different DDS implementations, facilitated by adherence to this standard, allows for seamless integration of heterogeneous systems. Despite potential challenges related to the dynamic discovery service, which can become a performance bottleneck with many participants, various optimization techniques and configurations can be employed to mitigate this issue.

DDS has been widely adopted as the default middleware for the Robot Operating System (ROS), a framework extensively used in academia and industry for developing robotic applications. ROS leverages DDS to handle the communication between its distributed nodes, benefiting from the protocol's real-time capabilities and robust data

exchange mechanisms. The use of DDS in ROS proves its applicability in diverse and demanding environments, highlighting its role in advancing the state of the art in robotic systems. In general, DDS offers a powerful and flexible communication solution for real-time and embedded systems. Its decentralized architecture, comprehensive QoS policies, and interoperability make it a preferred choice for many critical applications. DDS remains a foundation of modern robotic frameworks, enabling reliable and efficient data exchange across complex and distributed systems.

Eclipse Zenoh is a recently introduced open-source communication protocol operating under the Eclipse Foundation. It leverages both the publisher/subscriber and client-server paradigms, addressing limitations found in existing protocols like DDS. Zenoh enables the creation of highly configurable network topologies that can span beyond a single physical location, facilitating the development of scalable distributed systems that operate seamlessly across cloud and edge devices. This capability is particularly important in modern IoT and robotic applications, where the integration of various devices and systems across different environments is crucial for efficient operation and data management.

Unlike other protocols, Zenoh does not impose a specific serialization format; users can choose any format, provided that publishers and subscribers agree on it. This flexibility allows developers to optimize data serialization based on their specific application needs, enhancing performance and interoperability. For instance, in scenarios requiring high-speed data transmission, lightweight serialization formats can be employed to reduce latency, while more complex formats can be used when data integrity and richness are important. The absence of a specific type of serialization format also simplifies integration with existing systems that may already use proprietary or specialized data formats, reducing the overhead associated with converting data between different representations.

Zenoh's modern, library-oriented design is a significant advantage over more established alternatives. It provides a lightweight, efficient, and easily deployable commu-

nication middleware that can be integrated into a wide range of devices, from resource-constrained edge devices to powerful cloud servers. This design philosophy aligns with the current trend towards microservices and containerized applications, where modular and reusable components are preferred. Zenoh supports various Quality of Service (QoS) policies, enabling fine-grained control over communication behavior, such as reliability, latency, and resource utilization. These features make Zenoh a good choice for developers looking to build resilient and high-performance distributed systems.

Additionally, Zenoh's ability to operate over different transport protocols, including TCP, UDP, and even shared memory, further enhances its adaptability and performance. This multi-transport support allows Zenoh to optimize communication based on the underlying network conditions and application requirements, providing a robust and efficient communication solution. The extensive documentation and active community support under the Eclipse Foundation ensure that developers have access to the resources they need to effectively implement and leverage Zenoh in their projects. In general, Zenoh's innovative features and flexible design make it a powerful tool for building next-generation distributed systems that require efficient, reliable, and scalable communication across diverse environments.

The landscape of pub/sub networks extends beyond DDS implementations and Zenoh. Other popular protocols, such as MQTT and Redis, are widely used, particularly for applications that prioritize flexibility over high performance.

## 2.3. Modern Container Technology

Containers are lightweight virtualization solutions that can be traced back to the chroot utility in early UNIX systems. The introduction of isolation features like namespaces and cgroups in the Linux kernel marked a significant milestone in the early 2000s, leading to the emergence of various containerization technologies since then [8]. Namespaces provide isolated views of system resources, ensuring that processes within a container cannot see or affect processes outside it. Meanwhile, cgroups

allow precise control over resource allocation, such as CPU, memory, and I/O bandwidth, among containers. Docker containers [9], later standardized by the Open Container Initiative (OCI), have become the most popular containerization technology, offering a vast ecosystem of user-friendly tools and conventions. Docker simplifies the process of packaging, distributing, and running applications, making it easier for developers to create reproducible environments. This standardization has driven widespread adoption across various industries, from web development to scientific computing, and has promoted a vibrant community that continually enhances and extends Docker's capabilities.

For CPU-intensive tasks, containers can achieve near-native performance. This is due to the minimal overhead introduced by containerization compared to traditional virtual machines, which require a full OS stack for each instance. Containers share the host OS kernel, resulting in more efficient use of system resources and faster startup times. It is possible to restrict computing resources individually per container, which simplifies deployment in resource-constrained environments. This granularity in resource allocation ensures that applications run efficiently and predictably, even under varying loads. However, container network isolation can introduce significant overhead for inter-container communication tasks. This overhead arises from the additional layers of abstraction required to isolate network traffic between containers. Consequently, modern container runtimes support multiple networking modes to balance network isolation and performance [10, 11]. These modes include bridge networks, host networks, and overlay networks, each offering different trade-offs between security, performance, and ease of configuration. By selecting the appropriate networking mode, developers can optimize containerized applications for their specific use cases, whether they prioritize security, simplicity, or high throughput.

# 3. RELATED WORK

## 3.1. Design and Implementation of a Robotic Software

In [12], the authors conducted a comprehensive survey on the design, implementation, and verification of embedded software for robotic applications, with a focus on mobile robots like self-driving cars. These robots fall into a high-level autonomy category and face two major challenges regarding testing and verification. First, as safety-critical applications operating in human-populated environments, they require stringent correctness guarantees. Second, their complex operational conditions demand advanced hardware and intelligent algorithms, such as neural networks, which complicate both implementation and verification. This paper highlights challenges relevant to our work in addressing these issues which is explained in following sections regarding the design and implementation phases.

### 3.1.1. Design of Robotic Software:

In this section authors emphasize the need for a unified framework(languages, IDEs, and APIs) as a synthesis of low-level autonomy modules(e.g perception, control, planning) and high-level softwares for functionalities like goal specification and fault tolerance. This unified platform can address the need for systematic design of a complex autonomous robot system which is cumbersome and many-layered. In traditional understanding two main objectives or specifications for mobile robots were to reach from location A to B while avoiding obstacles. These two specifications could be captured via methods like Linear Temporal Logic (LTL) [13] in order to generate a finite set of robot's actions over time. Later several approaches like SMT solvers or Mixed-Integer Linear Programming tools could be used to come up with robot's trajectories to complete the mission by determining low-level control polices and high-level motion plans [14–17]. On the other hand, in modern approaches for a systematic design of autonomy task, there are several designed platforms like Antlab [18] for controlling

multiple robots that streams down high level user defined tasks all the way into low-level robotic sub-tasks like controlling and planing. Antlab provides the user with a high-level programming environment in which they can specify a sequence of actions for multiple robots. These tasks are then scheduled and distributed among robots by a runtime system and then compiled to specific tasks for each of them automatically without even needing the user to know current state of these robots.

Another important aspect of a systematic design is to involve a monitoring sub-system to capture the real-time actions of the robot in order to report faulty behaviors and enforce the system to a fail-safe state. Considering the expectations from and complexity of current high-level safety-critical robotic systems and their sub-systems(e.g. perception and planning modules), the need for a more complex monitoring emerges to ensure the correct perception of environment, execution of plans and enforcement of a fail-safe behavior. Such a monitoring system can be designed via high-level specification languages like Regular Expressions and Temporal logic that can satisfy the requirements of a complex monitoring [19,20] by providing a pattern about temporal, quantitative information of parameter space of a robotic system like how the value of a sensor in different intervals has changed. One sophisticated approach is to generate the monitoring system as a network of individual modules for the sake of temporal logic [21], timed specifications [22], quantitative [23], and parametric specifications [24, 25]. However the design of a monitoring system as a network of modules provides an abstractive composition of specifications that offers better extensibility and several other advantages [26, 27], the tools that implemented such monitoring system like [28–31] are developed in different programming languages and have different interfaces that complicates the monitoring task. Therefore there is still need for a comprehensive general framework that handles all monitoring tasks at once as a whole. ROSMOP [29] is a general software monitoring framework that constitutes from a set of softwares handling all specifications together.

### 3.1.2. Implementation of Robotic Software:

Besides a fault tolerant and safe design of the system, implementational details also play a big role in the correctness of the system which can be considered by choosing

the right hardware to run the even complex code and algorithms like neural networks which are energy consuming and are of high space complexity and also need an optimized software to handle the concurrency and parallel executions. Finite precision arithmetic in which the results of calculations are limited to a finite number of significant digits can also reduce the amount of memory and time needed to perform operations which in turn makes it suitable for computer vision and robotic applications. Finite precision arithmetic can be used to implement robot's feedback controller software. As an example [32], employing an stochastic optimization approach [33], designs a controller that can minimize state and control costs alongside minimizing the error in finite precision implementation of the controller. Another application of finite precision arithmetic is to come up with an optimal compilation scheme for feedback controller software. As an example, [34] uses a genetic programming technique to minimize controller error using fixed-point arithmetic in which the range of software's output error is defining the fitness score. However the non-linear complexity of some robotic systems' controller still remains as verifying/synthesizing challenge of neural-network based controllers [35] lack stability, abstractive interpretation [36] lacks precision, and reduction to SMT problem [37] lacks scalability for complex systems.
Another issue in implementation phase of a robotic system is to come up with a reliable scheduling algorithm which is very related to the existing hardware components used in multi-core, heterogeneous computer architectures in order to meet the computationally intensive requirements of autonomy software modules. In general a safety-critical application should provide a perfect compatibility between autonomy software, underlying real-time operating system and hardware; [38] conducts a survey for this objective. Another work [39], Implements an autonomy software module along with its Predictable Execution model (PREM) that reaches less Worst-Case Execution Time (WCET) via scheduling the program into compute and memory phases. Worst-case execution time is an important issue which fluctuates depending on the hardware that algorithms are executed on; this fact emerges a need to know the behavioral patterns of individual hardware in order to have a better estimation about temporal correctness related parameters. [40], and [41] provide surveys around the issue for GPU applications in general and specifically NVIDIA GPUs respectively that facilitates the process

of choosing the right hardware for application specific purposes.
Finally, to wrap up the aspects of an efficient implementation for a safety critical robotic application, the issue of how to implement neural networks has to be mentioned. Neural networks are prevalent in robotic realm due to their good performance. Because these algorithms consume too much computational, memory, and energy resources, their implementation must remain efficient. On the other hand, from monitoring perspective regardless of algorithms' efficiency, their consumption must be estimated.

### 3.2. Runtime Verification Containers for Publish/Subscribe Networks

Various runtime verification frameworks have been proposed in the literature, offering diverse runtime monitors and checks for publish/subscribe networks, particularly for the Robot Operating System (ROS). The first version of ROS used a custom pub/sub protocol, while ROS 2 utilizes the DDS protocol. Early efforts to apply runtime monitoring to ROS-based applications include the ROSRV framework [42] and [43]. Monitoring has been identified as a promising direction for robotics [12]. The work in [44] and the framework in [45] focus on supporting components for monitoring, while code generation from requirements and specifications is explored in [46]. Temporal logic monitoring on ROS has been investigated in [47]. However, these approaches lack viable strategies for integration into industrial workflows. Recently, a comprehensive survey on runtime verification in the ROS ecosystem was presented in [48].

Containerization and microservices architectures have gained significant traction in robotic applications [49–51]. Similarly, the automotive industry is transitioning to software-defined solutions, as seen in industrial consortia like Eclipse SDV [52] and SOAFEE [53]. The overhead of system virtualization in robotic and automotive applications has been studied in [54]. The granularity of containerization is also a concern, with a comparative study presented in [55]. Real-time and mixed-criticality containers are another important topic, as standard containers do not offer real-time guarantees. In [56], the authors study real-time schedulers for Kubernetes orchestration in

robotic applications, but their container and system monitoring is limited to basic health checks, unlike the functional system properties that a runtime verification tool can provide.

# 4. CONTAINERIZATION

This chapter outlines our approach to containerizing an existing specification based runtime verification tool. Reelay is an online runtime verification library designed to monitor message streams against past-time metric and first-order temporal logic specifications [57]. Previous works have mainly focused on demonstrating the performance of online monitoring using logfile-based Timescales benchmarks [58]. In contrast, we explore several containerized architectures that integrate runtime verification functionality and evaluate their performance, as detailed below.

## 4.1. Target Architectures

The central element in our architectures is the runtime verification engine container. This container houses an executable that employs the Reelay library and the appropriate pub/sub API for the selected communication protocol, such as DDS or Zenoh. The executable subscribes to designated topics, processes each incoming message with the RV engine, and then publishes the resulting verdict message. By default, Reelay monitors read JSON strings, and both DDS and Zenoh networks support the transmission and reception of these strings. Figures 4.1, 4.2, 4.3, 4.4 depict the container architectures examined in this work for performance evaluation and case studies. We begin with a straightforward benchmarking setup shown in Figure 4.1 , where the benchmark publisher component sends simulated data messages in JSON format on a specific topic.

The second setup, shown in Figure 4.2, represents a typical application configuration involving multiple participants, with the RV container integrated into the network as an additional participant. Publish/subscribe networks generally use asynchronous communication, while runtime verification tools like Reelay require a strictly synchronized message stream. To address this, we incorporate an adapter component, which is our monitoring ROS2 node, between the RV container and the publish/subscribe

network. This monitoring node consolidates and synchronizes messages from the subscribed topics into a single topic for the RV container to subscribe to. We will elaborate on the implementation of this approach and its current limitations in Chapter 6.

Figures 4.3 and 4.4 depict our multi-network setup for benchmarking and applications. Given the widespread use of the DDS protocol and the emerging potential of Zenoh, we primarily focus on the DDS-to-Zenoh scenario where the system operates on DDS and the runtime verification component functions on Zenoh. In these setups, we utilize a bridge component to translate messages between the two protocols. While this introduces some overhead, it is a common scenario when dealing with containerized pub/sub networks. Additionally, in native Zenoh networks, participants can discover each other using multicast, a feature not available for containers. Therefore, a Zenoh router is required unless participants are aware of each other's network addresses.

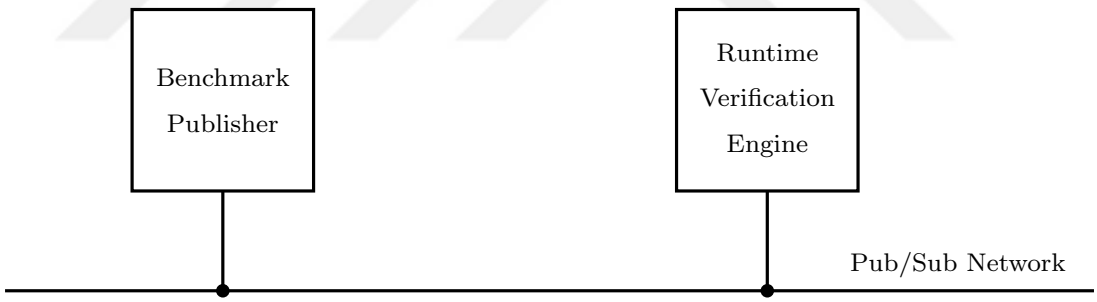



Figure 4.1. Single Pub/Sub Network for Benchmarks.

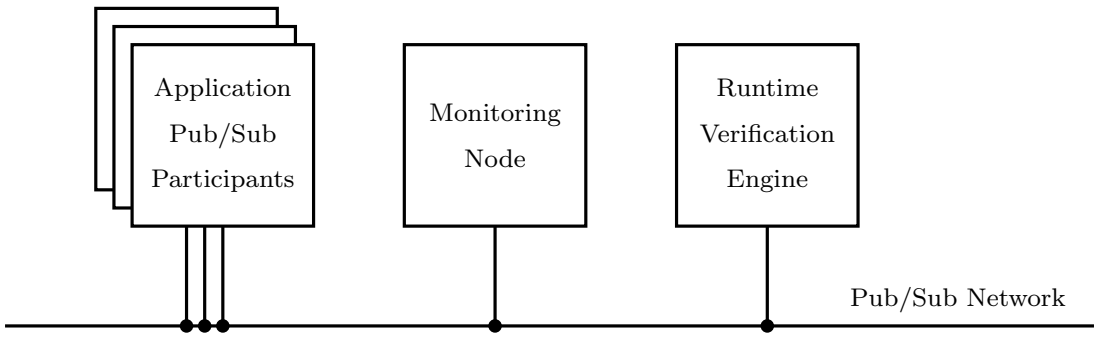



Figure 4.2. Single Pub/Sub Network for Apps.

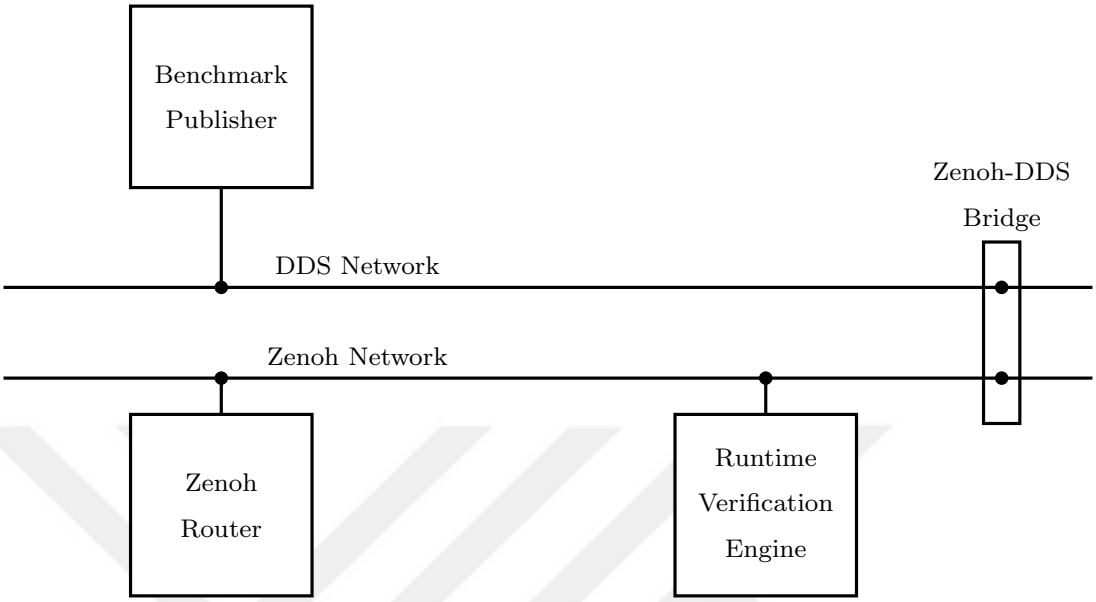



Figure 4.3. Multi Pub/Sub Network for Benchmarks.

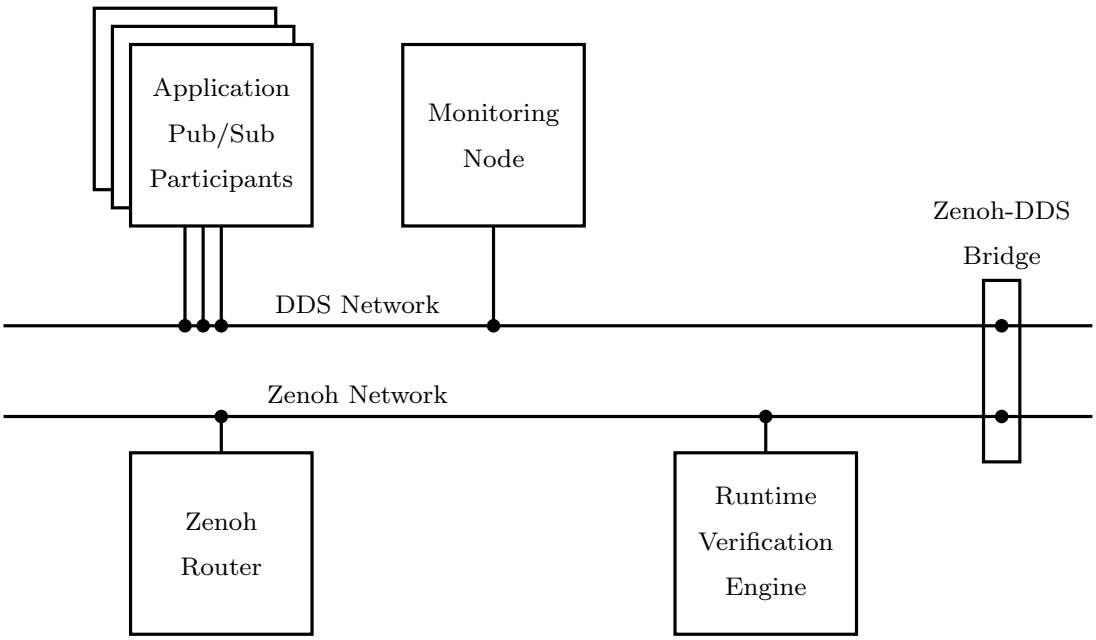



Figure 4.4. Multi Pub/Sub Network for Apps.

## 4.2. Container Networking Modes

Networking overhead between containers can be excessively high for applications that require high frequency and throughput, as the added layers of abstraction and isolation can introduce latency and reduce data transfer rates. To address this, container runtimes offer various networking modes that allow users to balance performance and isolation requirements. Docker containers, for example, support several networking modes including bridge, host, ipvlan, macvlan, and direct one-to-one connections between containers. The bridge mode, which is the default networking mode in Docker, creates a private internal network on the host where containers can communicate with each other through a virtual bridge. While it provides a good balance between usability and isolation, it may not offer the best performance for high-throughput applications. The host mode removes the network isolation between the Docker container and the Docker host, allowing the container to use the host's networking directly, thus reducing latency but at the cost of reduced isolation. Ipvlan and macvlan modes enable containers to have their own network interfaces, which can improve network performance and allow for more flexible network configurations. Direct one-to-one connections, often implemented using techniques like container-to-container networking or container-specific virtual network interfaces, can provide the highest performance by minimizing the layers of network abstraction and reducing the overhead associated with packet processing. By selecting the appropriate networking mode, developers can fine-tune their containerized applications to meet the specific needs of their workloads, whether they prioritize maximum performance, security, or a balance of both.

## 4.3. Container Orchestration

We utilize Docker Compose (docker-compose) to manage multi-container applications. This tool simplifies the orchestration of applications that depend on several containers operating together. Docker Compose allows for the declarative specification of services, network settings, data volumes, and other dependencies within a single YAML file, typically named compose.yml. This centralized configuration file makes

it easy to define and run multi-container Docker applications, reducing the complexity involved in manually managing individual containers. By using Docker Compose, we can ensure that all the necessary containers are started in the correct order, with the appropriate network configurations and shared volumes, thereby streamlining the development and deployment processes. This is particularly beneficial for handling the multiple container configurations examined in this study, as it allows for consistent and reproducible environments across different stages of development and testing. Furthermore, Docker Compose supports various features such as scaling services, defining environment variables, and setting up health checks, which enhance the robustness and flexibility of the container orchestration process. By leveraging these capabilities, we can efficiently manage the intricate dependencies and interactions between containers, ensuring optimal performance and reliability of the overall application.

### 4.4. Multi-stage Container Build

Multi-stage container builds are an effective technique for creating small and efficient container images. Given that containers are frequently downloaded online, minimizing container size is vital for optimizing deployment times and reducing bandwidth usage. This technique separates the build and runtime environments within a single container file, allowing developers to include only the necessary dependencies and artifacts in the final image while excluding unnecessary build tools and intermediate files. By discarding these elements after the build phase, multi-stage builds can significantly reduce the overall image size.

In a multi-stage build, each stage within the Dockerfile can use a different base image and have its own set of instructions, allowing for a clean and organized build process. The initial stages are typically used to compile the application, run tests, and package the application, while the final stage contains only the essential components needed to run the application. This results in a final image that is not only smaller in size but also more secure, as it contains fewer components that could potentially introduce vulnerabilities.

By leveraging multi-stage builds, developers can streamline the development workflow, as each stage can be independently optimized and maintained. Additionally, this approach promotes better practices in container image creation by encouraging the separation of concerns and modularity. The ability to create lean and efficient containers ultimately leads to faster deployments, lower bandwidth consumption, and reduced storage requirements, making multi-stage builds an essential technique for modern containerized applications.

# 5. CONTAINERIZED NETWORK PERFORMANCE EVALUATION

Throughput and latency measurements are essential metrics for evaluating the performance of networked applications. Network throughput refers to the rate at which data is successfully transmitted over a network connection, usually measured in messages per second or megabits per second. Network latency, often defined as half the total time for a message round-trip, involves sending a message to another participant and having that participant immediately respond with another message. These metrics illustrate the network's data-carrying capacity and responsiveness. Throughput and latency are influenced by various factors that affect real-world data transfer, including network congestion, protocol overhead, and processing delays.

The delivery guarantee for pub/sub messages is crucial for safety-critical runtime verification applications where data loss could have severe consequences. To address this, we use DDS with Reliable+KeepAll quality of service (QoS) settings, ensuring that all published messages are delivered to subscribing participants. Similarly, Zenoh offers reliable QoS settings, which we enable in our experiments. The protocols' versions and configurations are detailed as follows: For the DDS experiments, we utilize the Eclipse CycloneDDS implementation (cyclonedds v0.10) from distribution repositories. For the Zenoh experiments, we use the C library bindings (zenoh-c v0.10.2-rc) and the Zenoh-DDS bridge plugin of the same Zenoh version. All benchmarks are performed on a computer with the following specifications:

- CPU: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz, 12 cores
- Memory: 32GB RAM
- GPU: NVIDIA GeForce RTX 2060
- Network: Intel Corporation Wi-Fi 6 AX201 & Realtek RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller
- Kernel: Linux (5.15.0-105-generic)

- Operating System: Ubuntu Linux 22.04

We present our baseline performance experiments for the Zenoh and DDS protocols in Section 5.1. Section 5.2 details our experiments with runtime verification containers, demonstrating their performance in comparison to the baseline and non-networked settings.

## 5.1. Baseline Experiments in Containers

For our baseline DDS experiments, we use the industry-standard ddsperf utility to measure DDS performance. For the baseline Zenoh cases, we utilize custom Zenoh applications from the Zenoh C repository [59] to measure throughput and latency. These applications provide fine-grained control over message payload size and frequency, enabling comprehensive benchmarking. We containerize these applications using Docker and orchestrate them with the docker-compose utility to test various message parameters and container network modes. Below, we present the performance results from these experiments. These experiments belong to two different architectures which are shown in Figures 5.1 that depicts the single-protocol benchmarking scenario and Figure 5.2 which outlines the multi-protocol scenario in our baseline experiments combining both Zenoh and DDS protocols in the benchmarking process.

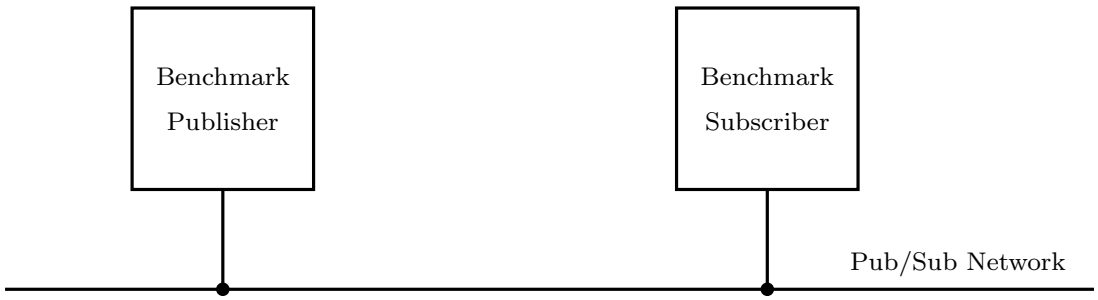



Figure 5.1. Single Pub/Sub Network for Benchmarks.

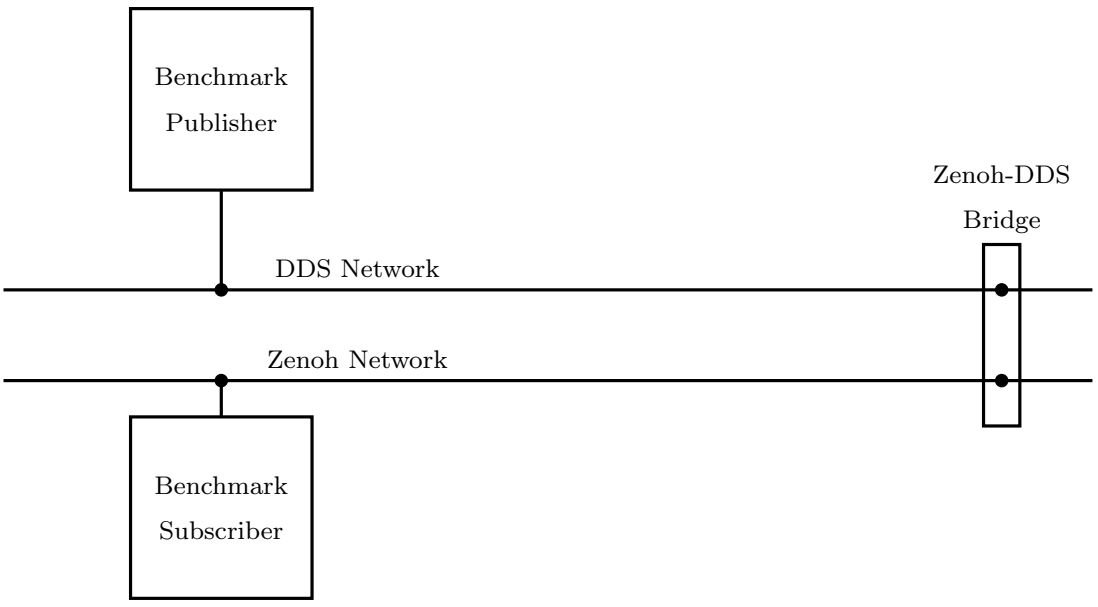



Figure 5.2. Multi Pub/Sub Network for Benchmarks.

The throughput measurement in Figure 5.3 shows the expected decrease in messages per second for the DDS protocol as the payload size increases. However, for very small messages, the performance stays constant because the protocol overhead is more important. Figure 5.4 shows the throughput in bits per second for the DDS protocol, giving another view of data transmission efficiency. Similarly, Figure 5.5 shows the throughput in messages per second for the Zenoh protocol. Zenoh follows a similar pattern to DDS, with declining message counts as payload size increases, but starts with about twice as many messages. This indicates that Zenoh can support much higher throughput for smaller message sizes. Finally, Figure 5.6 shows the throughput in bits per second for the Zenoh protocol, highlighting its ability for efficient data transmission. Notably, container network modes have a substantial impact on the throughput performance of containerized pub/sub networks in these benchmarks. Cases with no network isolation (host, direct) perform significantly better than those with full (bridge) or partial (ipvlan, macvlan) network isolation. The Zenoh protocol outperforms others, achieving rates of 4M msg/s for small messages and sustaining 2M msg/s for medium-sized messages. In comparison, CycloneDDS reached rates of up to 1M msg/s in our experiments.

The latency measurement plots in Figure 5.7 and Figure 5.8 reveal an interesting phenomenon in pub/sub networks, exhibiting two distinct behavior regimes based on message frequency—low-frequency and high-frequency. This phenomenon, previously reported in [60], is reproduced in our containerized setup across all networking modes. Latency for low-frequency messages can reach up to 300 microseconds for both DDS and Zenoh, while at higher frequencies, latency drops to around 20 microseconds. In our experiments, the threshold appears to be between 1K and 10K messages per second. The sharp change in latency is attributed to context switching within the operating system; high-frequency messaging keeps the process active, preventing the OS from putting it to sleep and avoiding the costly wake-up operation for new messages. Additionally, we observed that Zenoh protocol latency values were more consistent across different network modes. Although DDS generally outperforms Zenoh in terms of latency, this advantage diminishes under network isolation.

Finally, we present our benchmark results for the cross-protocol DDS-to-Zenoh case in Figures 5.9, 5.10, and 5.11, where Zenoh network subscribers can receive messages from DDS network participants. This setup is facilitated by an additional participant known as the Zenoh-DDS bridge, which performs the necessary translation between networks by subscribing to specified topics on one network and publishing them to the other. It's important not to confuse this with the bridge container network mode discussed earlier. This scenario is particularly relevant as many legacy systems currently use DDS-based architectures and won't transition to a new pub/sub framework like Zenoh immediately, despite its promise for new applications. This consideration also applies to our context, where we recognize the need to monitor DDS networks while developing our runtime verification application on Zenoh.

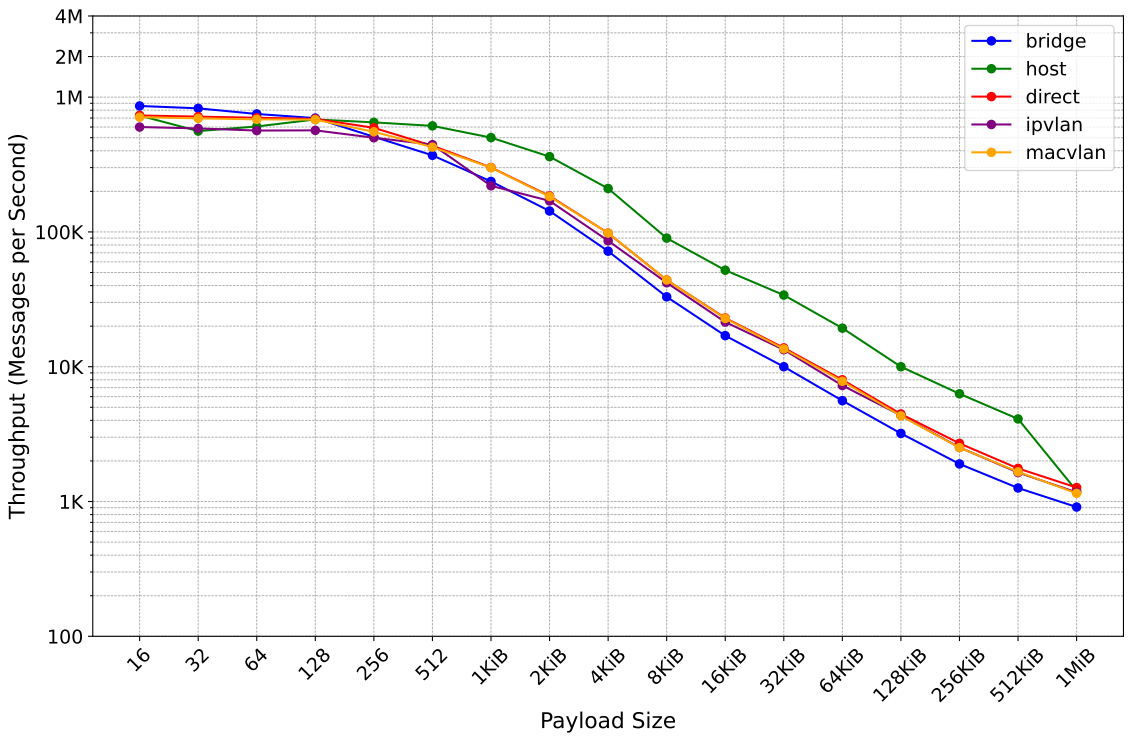



Figure 5.3. CycloneDDS Throughput(msg/s).

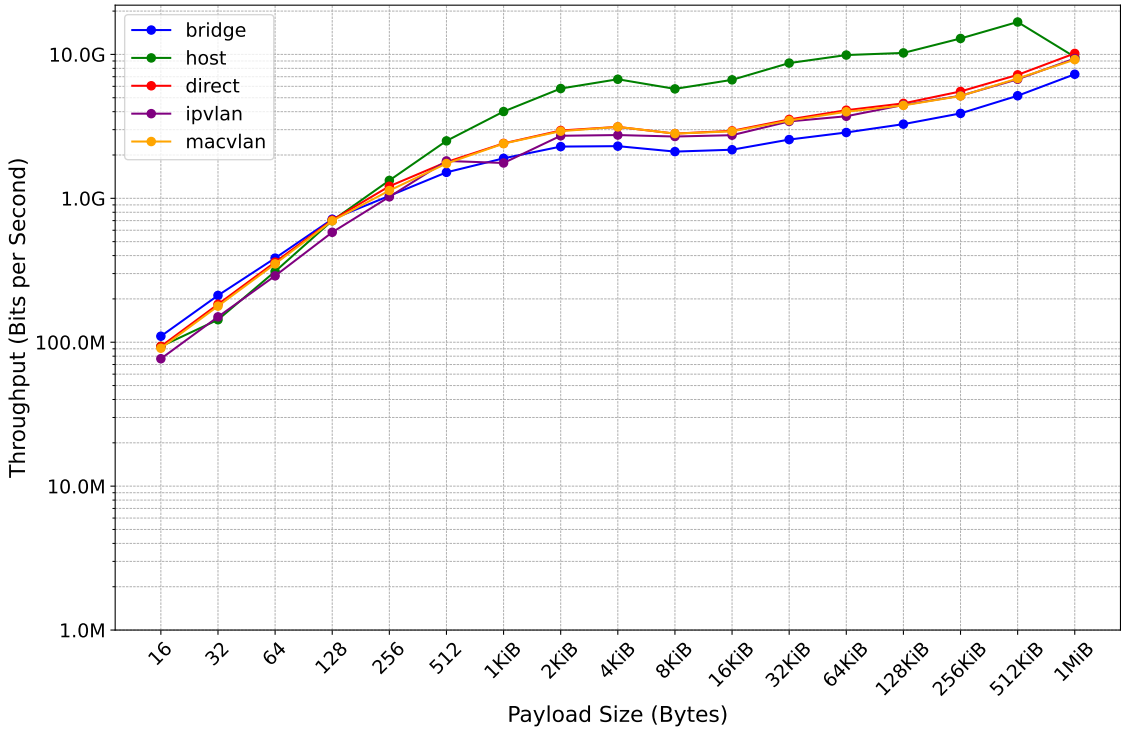



Figure 5.4. CycloneDDS Throughput(bit/s).

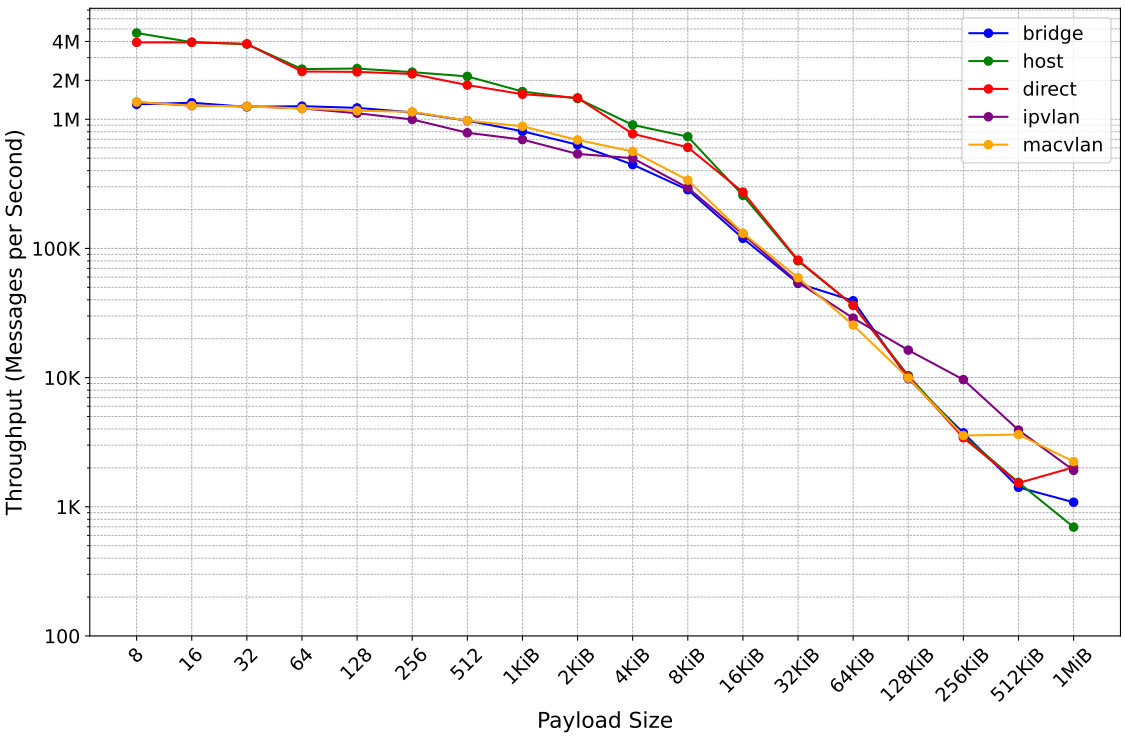



Figure 5.5. Zenoh Throughput(msg/s).

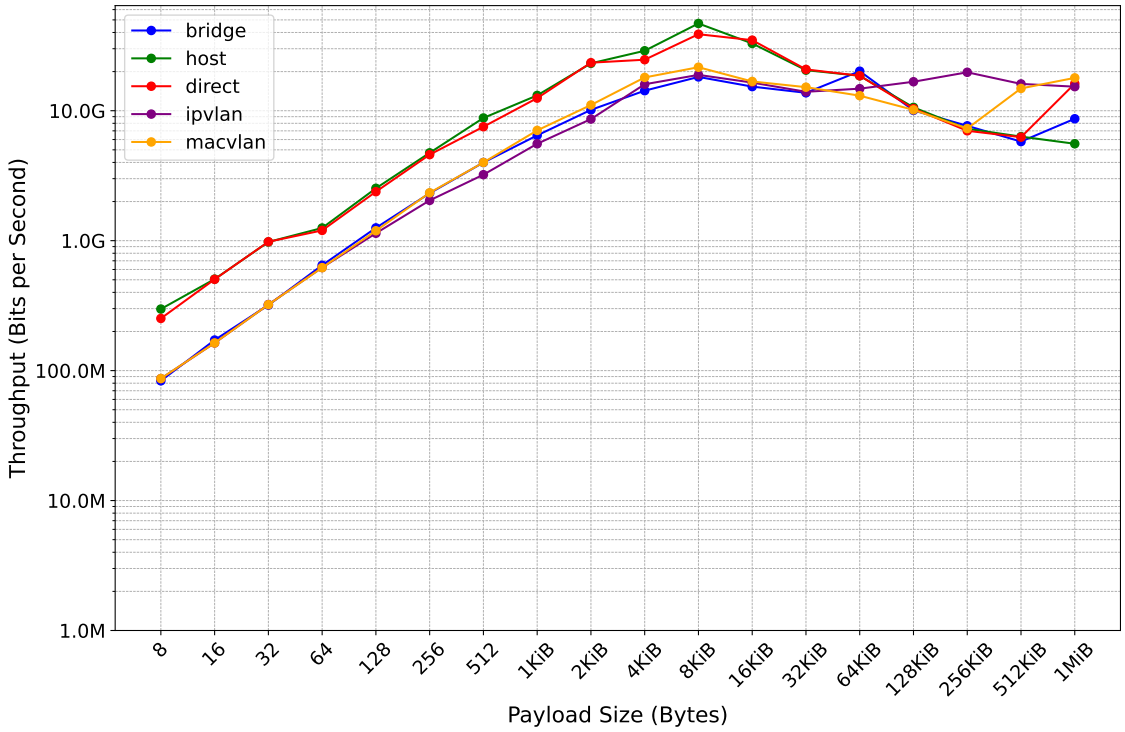



Figure 5.6. Zenoh Throughput(bit/s).

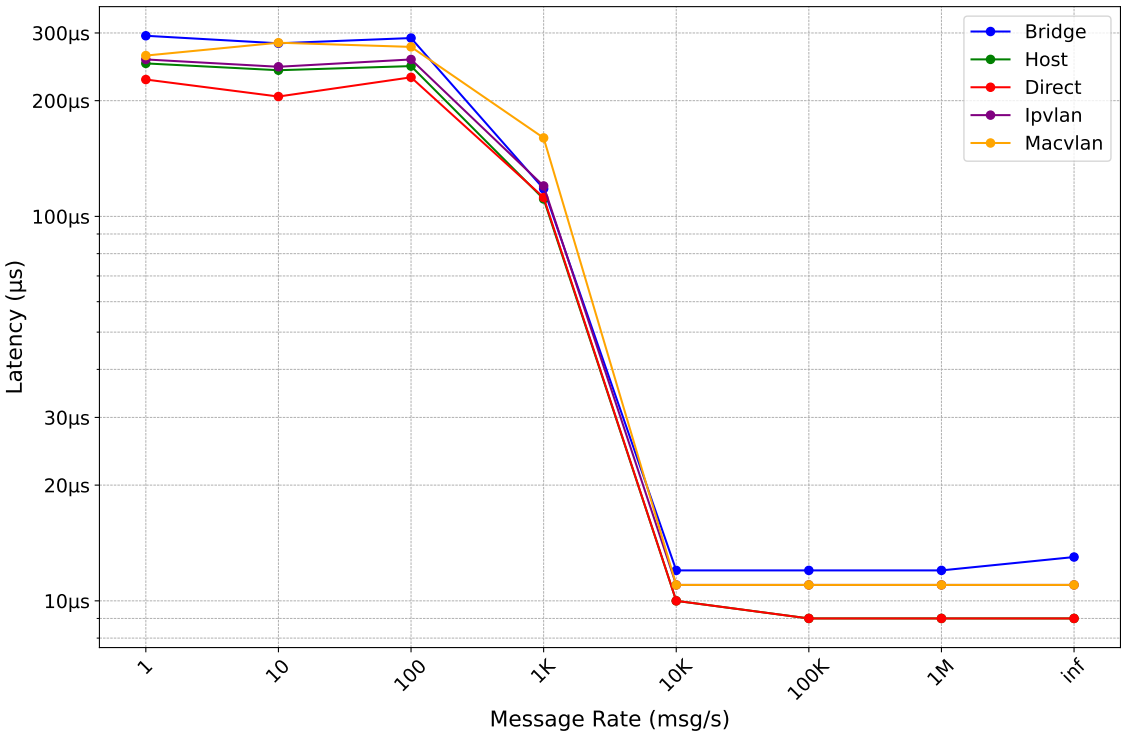



Figure 5.7. CycloneDDS Latency.

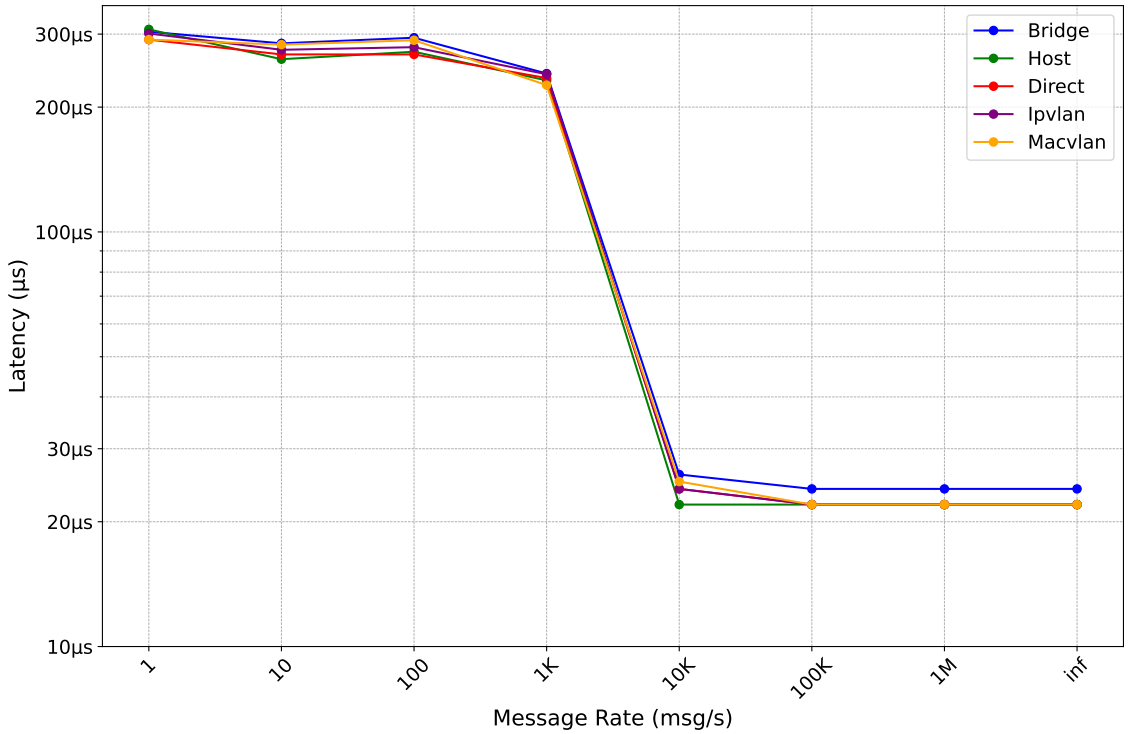



Figure 5.8. Zenoh Latency.

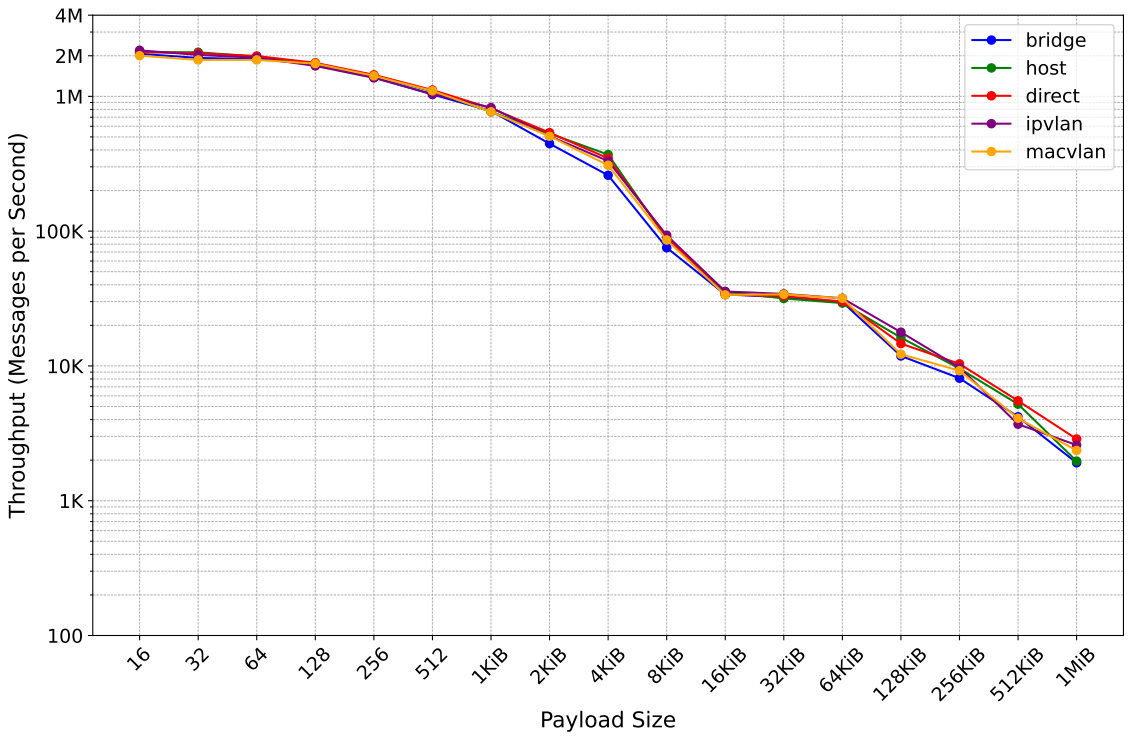



Figure 5.9. DDS-to-Zenoh Throughput(msg/s).

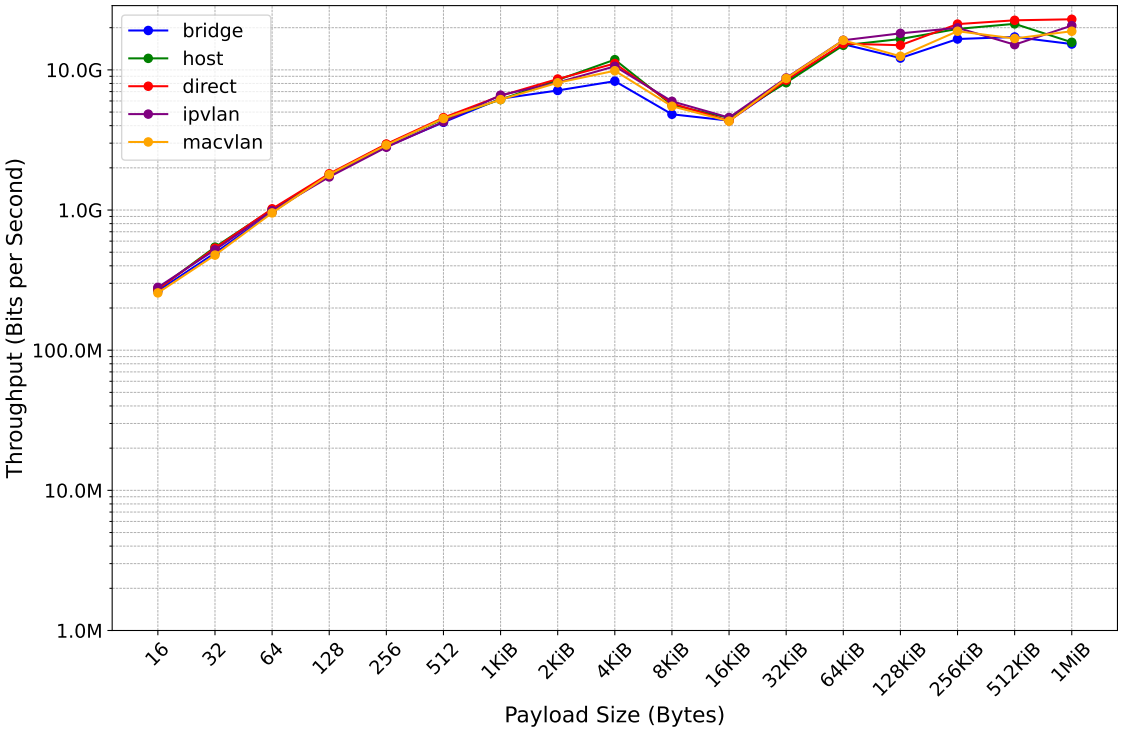



Figure 5.10. DDS-to-Zenoh Throughput(bit/s).

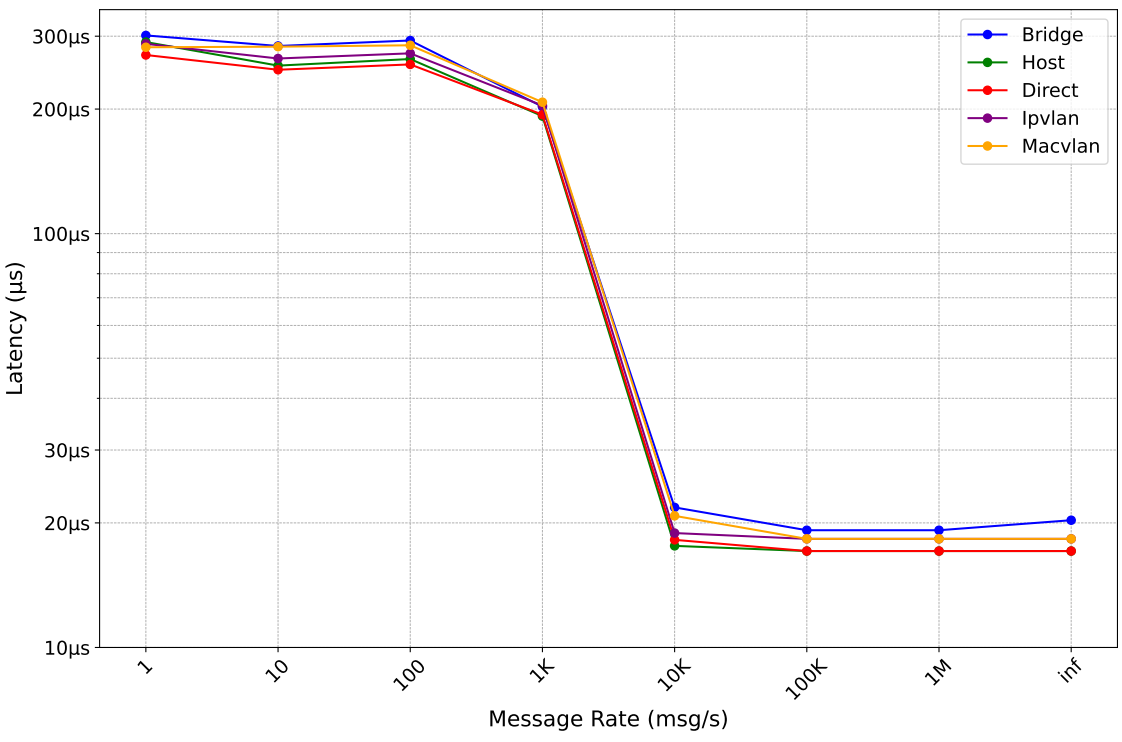



Figure 5.11. DDS-to-Zenoh Latency.

## 5.2. Runtime Verification Experiments in Containers

This section outlines our performance evaluation of containerized runtime verification applications. We utilize the Reelay library, designed for online monitoring of Metric Temporal Logic (MTL) formulas. In our experiments, Reelay is containerized as a subscriber to listen to a single pub/sub topic that streams propositional values for atomic propositions within formulas. On the publisher side, we use a custom participant to publish messages to the pub/sub network one by one, after reading them from Timescales benchmarks, which are commonly used to assess MTL monitoring tools' performance. Reelay expects JSON-formatted messages as propositional values by default, so we use JSON strings as the message payloads. Timescales messages are small, containing at most four propositional values, resulting in JSON strings no larger than 32 bytes.

We containerize Reelay as standalone CycloneDDS and Zenoh participants that subscribe to a dedicated topic and expect JSON-formatted strings as inputs. The container requires an MTL formula to operate, where the names of propositional atoms in the formula must match the names of Boolean fields in the JSON document. At startup, the Reelay library constructs a runtime monitor inside the container and updates the monitor state with each incoming message from the network. Reelay monitors are designed to require a strict time ordering of input messages, so we must ensure synchronization and ordering of these propositions if they originate from different pub/sub topics before feeding them to the runtime monitor.

We present our latency measurements for the runtime verification case in Figures 5.12, and 5.13. These measurements capture the time from message publishing to when the runtime monitor publishes the current verdict back to the network. We observe an additional wake-up delay of 100 microseconds. Notably, the actual processing delay attributable to the Reelay runtime verification functionality is likely less than 1 microsecond for these benchmarks, making it visually indistinguishable in the plots. Nevertheless, this measurement is valuable as it provides insights for accurately

predicting the overhead of runtime monitors, especially when used in feedback loops for real-time processes.

Table 5.1 illustrates the performance overhead of pub/sub networking and containerization compared to reading propositional values directly from local log files. We benchmark Reelay containers using Timescales benchmarks for DDS and Zenoh protocols across various container network modes. The table displays the total processing times when replaying each Timescales log file through the pub/sub network. The final column provides Reelay's raw performance when accessing data directly from local files, serving as a baseline for comparison. As expected, there is a noticeable increase in processing time due to pub/sub networking and containerization compared to direct local file access. Additionally, the results indicate that this overhead might be relatively lower for more complex MTL formulas, suggesting the potential for runtime verification containers to host multiple properties more efficiently.

These performance values align with our baseline performance benchmarks. In terms of throughput, Zenoh outperforms CycloneDDS and comes closer to non-networked tests. However, in these application experiments, we did not find compelling evidence to justify trading off container network isolation, as the two extreme networking modes showed similar performances. This distinction might not be significant for real-world applications, particularly for compute-intensive tasks, as discussed in Chapter 6.

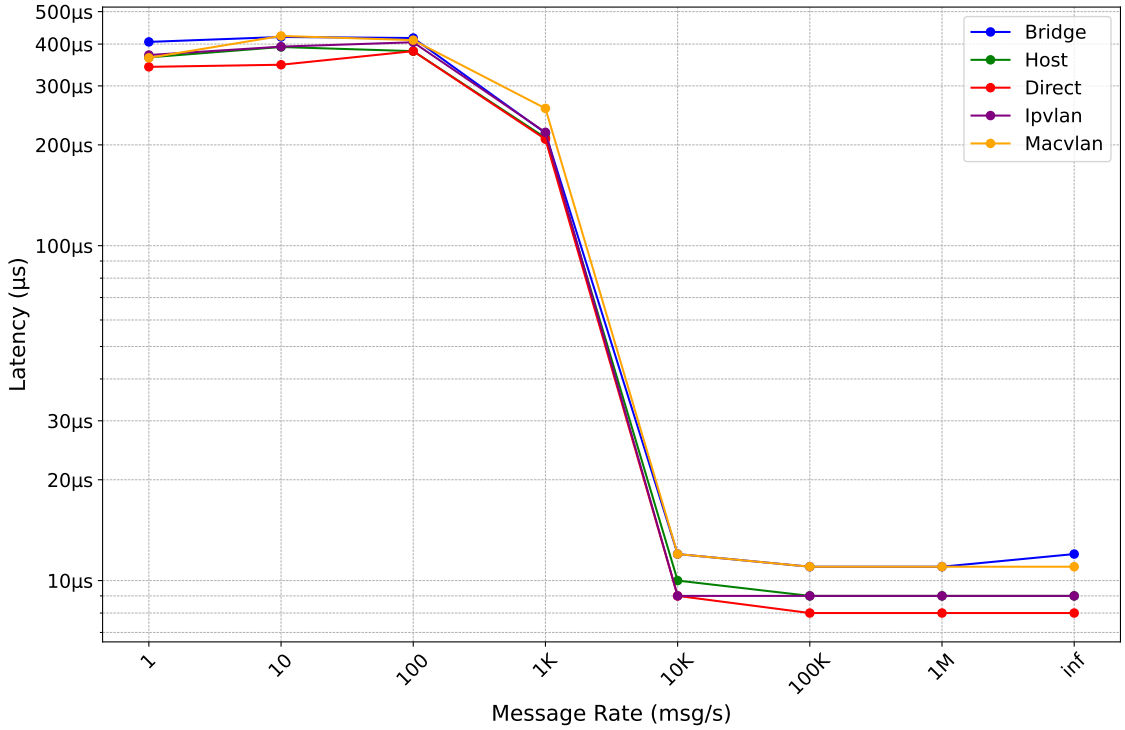



Figure 5.12. Reelay-DDS Latency Performance.

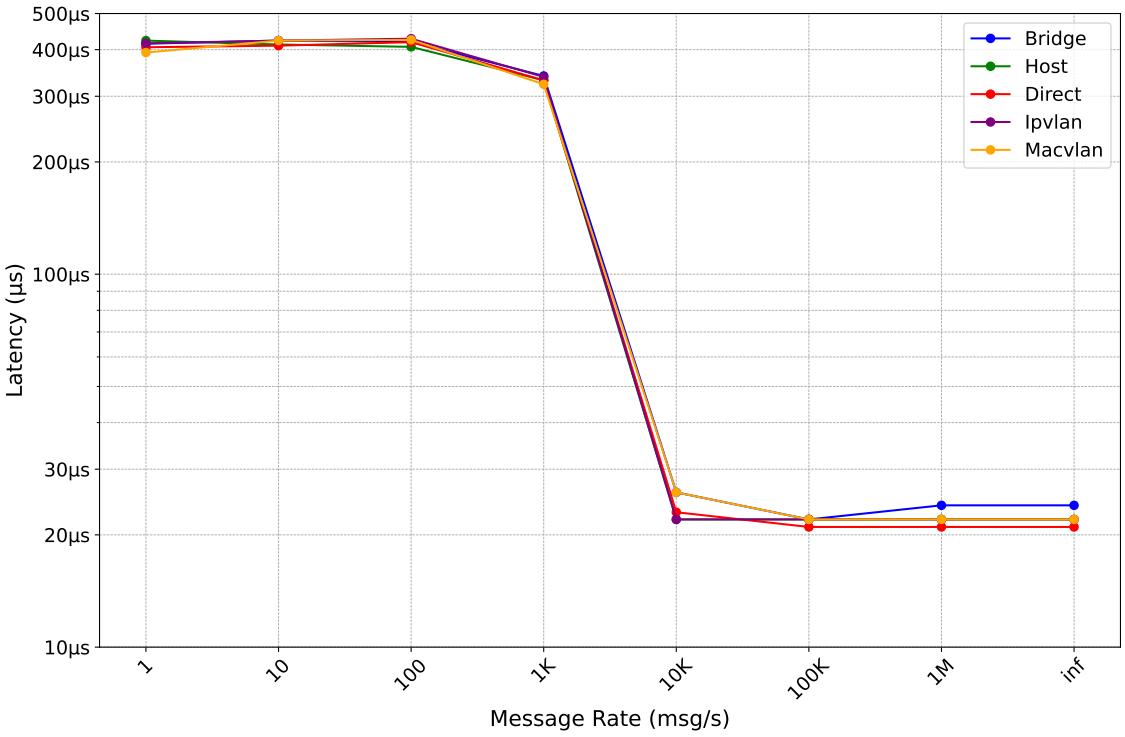



Figure 5.13. Reelay-Zenoh Latency Performance.

Table 5.1. Runtime verification container performance on pub/sub networks.

| | CycloneDDS | | Zenoh | | Filesystem |
|---|---|---|---|---|---|
| | Bridge | Host | Bridge | Host | Local |
| AbsentAQ10 | 1.56 | 1.38 | 0.71 | 0.72 | 0.39 |
| AbsentAQ100 | 1.55 | 1.35 | 0.64 | 0.67 | 0.36 |
| AbsentAQ1000 | 1.54 | 1.37 | 0.65 | 0.68 | 0.36 |
| AbsentBQR10 | 1.54 | 1.48 | 0.82 | 0.80 | 0.52 |
| AbsentBQR100 | 1.49 | 1.46 | 0.73 | 0.74 | 0.48 |
| AbsentBQR1000 | 1.50 | 1.50 | 0.74 | 0.74 | 0.48 |
| AbsentBR10 | 1.41 | 1.39 | 0.66 | 0.70 | 0.42 |
| AbsentBR100 | 1.39 | 1.37 | 0.67 | 0.73 | 0.42 |
| AbsentBR1000 | 1.40 | 1.38 | 0.66 | 0.67 | 0.42 |
| RecurGLB10 | 1.35 | 1.35 | 0.63 | 0.69 | 0.30 |
| RecurGLB100 | 1.32 | 1.31 | 0.58 | 0.61 | 0.23 |
| RecurGLB1000 | 1.34 | 1.33 | 0.56 | 0.56 | 0.22 |
| RespondGLB10 | 1.54 | 1.50 | 0.83 | 0.85 | 0.56 |
| RespondGLB100 | 1.50 | 1.47 | 0.89 | 0.91 | 0.46 |
| RespondGLB1000 | 1.51 | 1.44 | 0.80 | 0.89 | 0.45 |
| AlwaysAQ10 | 1.42 | 1.41 | 0.66 | 0.65 | 0.38 |
| AlwaysAQ100 | 1.41 | 1.43 | 0.65 | 0.63 | 0.35 |
| AlwaysAQ1000 | 1.43 | 1.39 | 0.64 | 0.64 | 0.34 |
| AlwaysBR10 | 1.42 | 1.41 | 0.67 | 0.69 | 0.40 |
| AlwaysBR100 | 1.43 | 1.40 | 0.74 | 0.74 | 0.42 |
| AlwaysBR1000 | 1.44 | 1.39 | 0.66 | 0.67 | 0.42 |
| AlwaysBQR10 | 1.58 | 1.50 | 0.74 | 0.74 | 0.51 |
| AlwaysBQR100 | 1.52 | 1.47 | 0.77 | 0.78 | 0.47 |
| AlwaysBQR1000 | 1.58 | 1.46 | 0.75 | 0.74 | 0.46 |

Table 5.1. Runtime verification container performance on pub/sub networks. (cont.)

| | CycloneDDS | | Zenoh | | Filesystem |
|---|---|---|---|---|---|
| | Bridge | Host | Bridge | Host | Local |
| RecurBQR10 | 1.59 | 1.57 | 0.79 | 0.77 | 0.62 |
| RecurBQR100 | 1.58 | 1.57 | 0.79 | 0.79 | 0.56 |
| RecurBQR1000 | 1.57 | 1.54 | 0.78 | 0.78 | 0.54 |
| RespondBQR10 | 1.79 | 1.74 | 0.93 | 0.91 | 0.84 |
| RespondBQR100 | 1.83 | 1.66 | 0.88 | 0.93 | 0.74 |
| RespondBQR1000 | 1.79 | 1.71 | 0.87 | 0.90 | 0.74 |

# 6. AUTOWARE CASE STUDY

This case study introduces two example deployment configurations to assess the feasibility of using runtime verification containers in practical applications. Specifically, we deploy the Autoware autonomous driving stack [6] and the Carla simulator [61] for our evaluation.

Autoware is a comprehensive, open-source software stack based on ROS 2, comprising over 250 ROS packages that are crucial for the sensing, perception, localization, planning, and control aspects of autonomous vehicles. A typical Autoware setup can create a network with nearly 100 DDS participants engaged in compute-intensive tasks such as object detection and collision avoidance. In contrast, Carla is an autonomous driving simulator built on Unreal Engine. The simulator inherently uses a client-server architecture and does not employ a pub/sub network internally. However, it can interact with external programs via its C++ and Python APIs. Both software stacks are resource-intensive in terms of compilation, deployment, and execution, posing challenges for setting up an autonomous driving testing environment. Consequently, containers and containerization technologies prove to be highly beneficial in managing this complexity.

Our initial deployment setup, depicted in Figure 6.1, involves running Autoware, the Carla simulator, and the Reelay runtime verification engine on a single host machine. This straightforward configuration minimizes network overhead, making it advantageous for early development and testing phases where performance optimization is not yet critical. However, this setup necessitates a robust workstation and may not scale effectively for more complex scenarios with high resource demands. Additionally, the DDS network has a participant limit of 120, meaning this configuration is already close to its capacity. This setup is belonging to the architecture depicted in Figure 4.2.

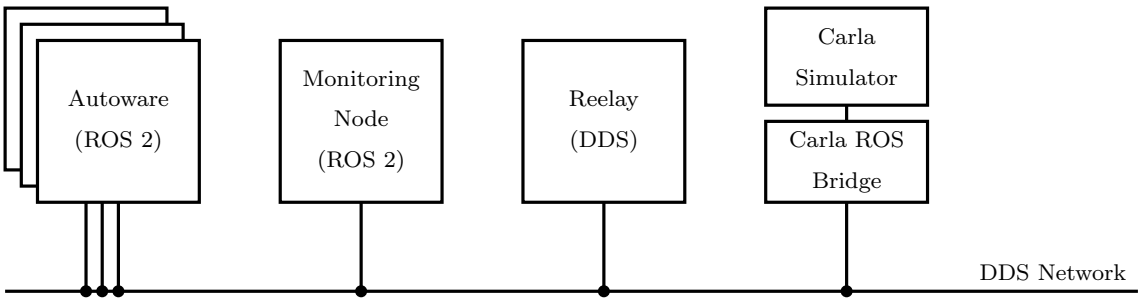



Figure 6.1. Container Deployment Setup Using a Single Host Machine.

Alternatively, we investigate a second deployment setup, depicted in Figure 6.2, where Autoware and the simulation/verification components are run on separate host machines. Although this setup is more complex and introduces additional networking overhead, it offers the advantage of isolating the system under test (Autoware) from the testing environment. This separation can be particularly beneficial or necessary when the deployment platform differs from the development environment. To facilitate this, we incorporate the Zenoh-DDS bridge on the first host, leveraging Zenoh's flexible configuration options to finely control message flow between hosts. This allows us to permit or restrict specific topics, giving the system architect greater control and potentially reducing network bandwidth usage. This setup is belonging to the architecture depicted in Figure 4.4.

The final element of these architectural setups is the monitoring node, which is also responsible for synchronizing messages. This will be elaborated in the next section.

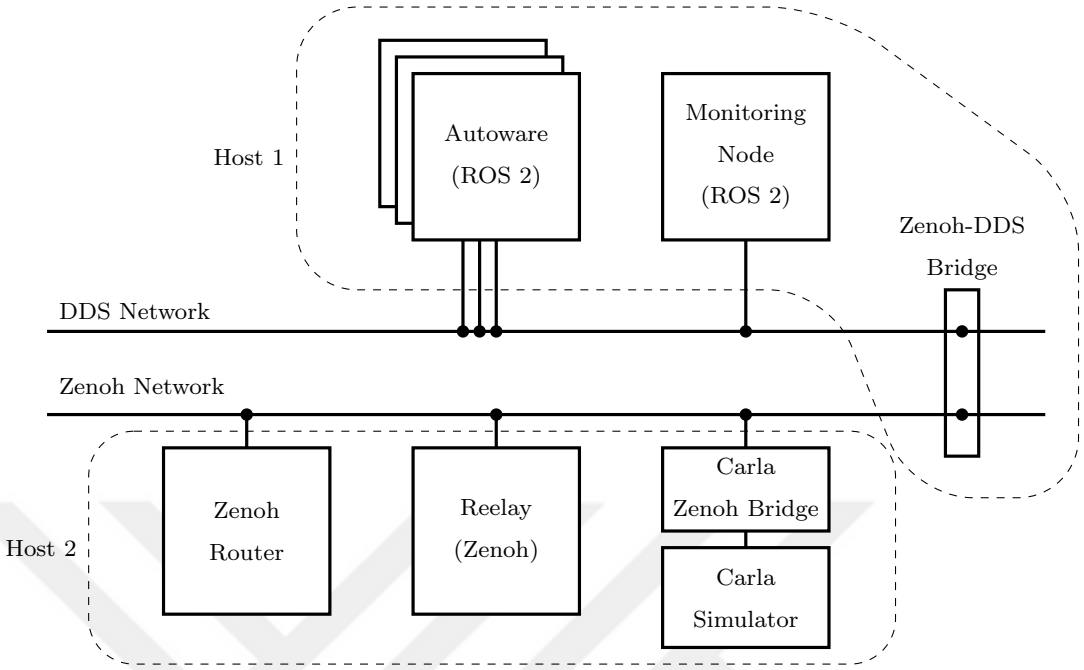



Figure 6.2. Container Deployment Setup Distributed Over Two Host Machines.

### 6.1. Monitoring Node Participant

A significant challenge in Autoware and similar distributed software stacks is the absence of synchronization across message streams. As pub/sub participants operate independently and exchange data asynchronously, it is difficult to ensure that all participants share a unified system view. This is especially problematic for runtime verification tools like Reelay, which require a synchronized input message stream.

To tackle this challenge, we developed a ROS Monitoring Node participant responsible for message synchronization that can be deployed alongside other Autoware nodes. This node plays a vital role in facilitating runtime verification by dynamically analyzing system behavior, ensuring adherence to specified predefined properties and requirements throughout execution.

Our Monitoring ROS package encapsulates two different ROS sub-packages: one for creating a custom ROS message for our monitoring node to publish to reflect the behavior of our simulated vehicle under experiment, and the other for the functionality

of the node itself. Inside the custom message file, we can include any set of messages belonging to either built-in ROS or Autoware messages. In our case, we include nine messages from the autoware_auto_vehicle_msgs and tier4_external_api_msgs packages. These message topics and their corresponding packages are listed in Table 6.1.

In the second sub-package, we implement the monitoring node itself. Upon initialization, the monitoring node subscribes to relevant ROS topics to gather pertinent data for verification by capturing a snapshot of the vehicle's operational status. Alongside these topics, the monitoring node also subscribes to the clock topic from the rosgraph_msgs package to encapsulate the current simulation time, providing the timestamp of the message to be published by the node.

After collecting data, the monitoring node synchronizes messages from different topics using the ApproximateTime policy from the message_filters library. Unlike the ExactTime policy, ApproximateTime does not require messages to have identical timestamps. Instead, it synchronizes messages with slightly varying timestamps, regardless of their arrival time, ensuring that messages experiencing arbitrary network or processing delays can still be synchronized. This policy identifies the best match for a set of messages based on their timestamps without needing a predefined time difference threshold. Each message is used only once, creating contiguous synchronization sets with no unused time between consecutive sets, thereby ensuring tight synchronization and minimal delay. However, a limitation of the ApproximateTime policy is that it can synchronize a maximum of nine messages by design. Consequently, our synchronizer subscribes to nine different messages from two distinct Autoware packages, generated and published by the Autoware stack, to verify the vehicle's current operational status in simulation. After synchronization, these messages are unified and published under a dedicated runtime verification topic in the network.

Users can select any subset of messages from Table 6.1 for subscription, with the collected data publishable as either a ROS custom message or serialized into a flat JSON format. These specific messages were chosen because they encompass crucial

Table 6.1. Evaluation of deployment setups.

| Package Name | Message | Setup1 | Setup2 |
|---|---|---|---|
| autoware_auto_vehicle_msgs | SteeringReport | 25.10 | 23.55 |
| autoware_auto_vehicle_msgs | GearReport | 25.02 | 23.69 |
| autoware_auto_vehicle_msgs | ControlModeReport | 24.95 | 22.59 |
| autoware_auto_vehicle_msgs | TurnIndicatorsReport | 24.88 | 22.68 |
| autoware_auto_vehicle_msgs | HazardLightsReport | 25.07 | 23.70 |
| autoware_auto_vehicle_msgs | Engage | 9.95 | 7.95 |
| tier4_external_api_msgs | FailSafeStateStamped | 9.95 | 7.84 |
| tier4_external_api_msgs | Emergency | 10.05 | 8.90 |
| tier4_external_api_msgs | VehicleStatusStamped | 4.99 | 2.97 |

vehicle state and control parameters such as steering angle, gear position, control mode, turn indicators, hazard lights, and engagement status. These parameters are vital for assessing the vehicle's maneuverability, control state, and signaling system.

Safety-related messages like FailSafeStateStamped and Emergency are included to monitor the vehicle's response to potential faults or hazardous conditions, ensuring appropriate handling of unexpected situations. The VehicleStatusStamped message offers a comprehensive overview of the vehicle's overall status.

By synchronizing and analyzing these messages, the monitoring node can accurately depict the vehicle's operational state in real-time. This capability is particularly valuable for testing and verifying complex maneuvers, safety-critical decision-making, and the vehicle's interactions with dynamic environments. For instance, during an emergency braking scenario, the synchronizer ensures that the Emergency message is correctly correlated with the vehicle's braking and signaling actions.

The publishing rate of the monitoring node is significantly influenced by the frequency of the messages it subscribes to, which can be as low as three msg/s, as indicated in Table 6.1. Due to limitations imposed by simulation speeds, we have not conducted extensive performance tests like those in Chapter 5. However, our

observations suggest that the overhead of the Reelay verification tool on communication is minimal at these publishing rates.

## 6.2. Evaluating Deployment Setups

In our deployment experiments, we tested both setups. The first setup resulted in a very low simulation speed, dropping to as low as five frames per second on the machine specified in Chapter 5. While the setup remains functional at these speeds, it raises concerns about the practicality of the simulation. Early-stage simulation-based testing relies heavily on visualization, where engineers visually validate scenarios. Such low performance hinders these development workflows.

For the second setup, we implemented a distributed approach using an additional machine with specifications identical to the initial one. This setup offloads some computationally intensive tasks to the second machine, resulting in enhanced simulation speed and a smoother visual experience for the engineer. As anticipated, simulations run more smoothly, and the runtime verification engine subscribes to the synchronized topic. Containerization is vital for ensuring flexibility and modularity within these deployment configurations. Moreover, migrating between these configurations would be significantly more challenging without containerization.

# 7. CONCLUSION

In this thesis, we presented a comprehensive exploration of containerized runtime verification solutions within publish/subscribe networks, particularly focusing on their application to complex real-world systems like autonomous driving stacks. Our research began with the design and implementation of a real-time functional monitoring system capable of observing the behavior of an autonomous vehicle in a simulated environment. This system provides real-time feedback on both internal vehicle states and external environmental conditions, aligned with user-defined specifications, thereby ensuring targeted and effective monitoring for various testing scenarios.

We then introduced a containerized approach to integrating runtime verification applications within pub/sub networks, facilitating seamless communication with other participants in our distributed ecosystem. To thoroughly assess the performance of these runtime verification tools, we conducted a series of experiments across varying levels of complexity. Our initial baseline performance evaluations focused on measuring the inherent networking capabilities of containerized pub/sub network participants in a minimalistic setup, without additional participants that could introduce networking overhead. These evaluations provided a foundational understanding of the network's performance under standard conditions.

Building on this, we containerized a specification-based runtime verification tool and re-evaluated its performance within the pub/sub network. This step was crucial in understanding how the inclusion of runtime verification impacts the overall ecosystem's networking efficiency. Finally, we conducted a real-world case study centered on autonomous driving, where we integrated the runtime verification tool along with a containerized autonomous driving software stack and a containerized photo-realistic simulation. These experiments were executed across different hardware setups to assess the performance of runtime verification within a computationally intensive environment, providing a comprehensive evaluation of the entire system's performance under

realistic conditions.

Key findings from our research highlight the significant efficiency gains achieved through the use of runtime verification containers. By grouping related properties into cohesive containerized units, we enhanced synchronization efficiency and reduced overall system complexity. This approach proved particularly beneficial in high-participant environments, such as those found in autonomous driving systems, where the deployment of multiple containers to manage functional properties and verify system behavior in real-time became both feasible and advantageous.

Our exploration also revealed the potential advantages of pull-based subscriptions within pub/sub networks, which could offer increased efficiency and reliability for runtime verification components. While our experiments were limited to push-based subscriptions, the potential for pull-based approaches to enhance network throughput and reduce unnecessary data transfer suggests a promising area for future research.

Furthermore, we underscored the importance of flexible message formats, such as protobuf and flatbuffers, in optimizing verification processes. Zenoh's support for diverse wire formats, in contrast to the more rigid DDS implementations, provides an opportunity to improve performance and resource utilization within the runtime verification framework.

Containerization has proven to be a powerful tool for managing and verifying complex software stacks, offering significant benefits in terms of deployment flexibility, management simplification, and adaptability to evolving verification requirements. As demonstrated in our case studies, containerized environments enhance both the development and operational phases, making them a critical component in the future of real-time monitoring and verification systems.

Future work should focus on refining these methods and addressing current limitations while exploring the potential of emerging technologies. Key areas for further

investigation include developing and documenting support for pull-based subscribers to optimize network throughput and reduce unnecessary data transfer, and integrating more complex and varied temporal logic specifications into containerized runtime verification. Additionally, expanding the application of containerized monitoring systems to other domains, such as industrial automation and IoT, will be crucial for broadening the impact of these technologies in increasingly complex and distributed systems.

# REFERENCES


1. *Tesla Autopilot*, `https://www.tesla.com/autopilot`, accessed on June 29, 2024.

2. *Audi AI Traffic Jam Pilot*, `https://magazine.audi.com.au/article/audi-ai-traffic-jam-pilot`, accessed on June 29, 2024.

3. *Waymo - Self-Driving Cars - Autonomous Vehicles - Ride-Hail*, `https://waymo.com/`, accessed on June 29, 2024.

4. *ROS: Home*, `https://www.ros.org/`, accessed on June 29, 2024.

5. *OMG DDS*, `https://www.omg.org/dds/`, accessed on June 23, 2024.

6. *Autoware*, `https://autoware.org/`, accessed on June 23, 2024.

7. *Zenoh*, `https://zenoh.io`, accessed on June 23, 2024.

8. Bentaleb, O., A. S. Belloum, A. Sebaa and A. El-Maouhab, "Containerization Technologies: Taxonomies, Applications and Challenges", *The Journal of Supercomputing*, Vol. 78, No. 1, pp. 1144–1181, 2022.

9. Merkel, D., "Docker: Lightweight Linux Containers for Consistent Development and Deployment", *Linux J*, Vol. 239, No. 2, p. 2, 2014.

10. Claassen, J., R. Koning and P. Grosso, "Linux Containers Networking: Performance and Scalability of Kernel Modules", *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*, pp. 713–717, IEEE, Istanbul, Turkey, 2016.

11. Kozhirbayev, Z. and R. O. Sinnott, "A Performance Comparison of Container-Based Technologies for the Cloud", *Future Generation Computer Systems*, Vol. 68,

pp. 175–182, 2017.

12. Abbas, H., I. Saha, Y. Shoukry, R. Ehlers, G. Fainekos, R. Gupta, R. Majumdar and D. Ulus, "Special Session: Embedded Software for Robotics: Challenges and Future Directions", *2018 International Conference on Embedded Software (EMSOFT)*, pp. 1–10, IEEE, Turin, Italy, 2018.

13. Baier, C. and J.-P. Katoen, *Principles of Model Checking*, MIT Press, Cambridge, MA, 2008.

14. Hung, W. N., X. Song, J. Tan, X. Li, J. Zhang, R. Wang and P. Gao, "Motion Planning with Satisfiability Modulo Theories", *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 113–118, IEEE, Hong Kong, China, 2014.

15. Nedunuri, S., S. Prabhu, M. Moll, S. Chaudhuri and L. E. Kavraki, "SMT-Based Synthesis of Integrated Task and Motion Plans from Plan Outlines", *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 655–662, IEEE, Hong Kong, China, 2014.

16. Saha, I., R. Ramaithitima, V. Kumar, G. J. Pappas and S. A. Seshia, "Automated Composition of Motion Primitives for Multi-Robot Systems from Safe LTL Specifications", *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1525–1532, IEEE, Chicago, IL, USA, 2014.

17. Wolff, E. M., U. Topcu and R. M. Murray, "Optimization-Based Trajectory Generation with Linear Temporal Logic Specifications", *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5319–5325, IEEE, Hong Kong, China, 2014.

18. Gavran, I., R. Majumdar and I. Saha, "AntLab: A Multi-Robot Task Server", *ACM Transactions on Embedded Computing Systems (TECS)*, Vol. 16, No. 5s, pp.

1–19, 2017.

19. Bartocci, E., J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Ničković and S. Sankaranarayanan, "Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications", *Lectures on Runtime Verification: Introductory and Advanced Topics*, pp. 135–175, 2018.

20. Havelund, K., G. Reger and G. Roşu, "Runtime Verification: Past Experiences and Future Projections", *Computing and Software Science: State of the Art and Perspectives*, pp. 532–562, 2019.

21. Havelund, K. and G. Roşu, "Synthesizing Monitors for Safety Properties", *Tools and Algorithms for the Construction and Analysis of Systems: 8th International Conference, TACAS 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8–12, 2002 Proceedings 8*, pp. 342–356, Springer, Grenoble, France, 2002.

22. Basin, D., F. Klaedtke and E. Zălinescu, "Algorithms for Monitoring Real-Time Properties", *Acta Informatica*, Vol. 55, pp. 309–338, 2018.

23. Dokhanchi, A., B. Hoxha and G. Fainekos, "On-Line Monitoring for Temporal Logic Robustness", *International Conference on Runtime Verification*, pp. 231–246, Springer, Toronto, Canada, 2014.

24. Basin, D., F. Klaedtke, S. Müller and E. Zălinescu, "Monitoring Metric First-Order Temporal Properties", *Journal of the ACM (JACM)*, Vol. 62, No. 2, pp. 1–45, 2015.

25. Havelund, K., D. Peled and D. Ulus, "First-Order Temporal Logic Monitoring with BDDs", *Formal Methods in System Design*, Vol. 56, No. 1, pp. 1–21, 2020.

26. Ulus, D., "Sequential Circuits from Regular Expressions Revisited", *arXiv Preprint arXiv:1801.08979*, 2018.

27. Annpureddy, Y., C. Liu, G. Fainekos and S. Sankaranarayanan, "S-Taliro: A Tool for Temporal Logic Falsification for Hybrid Systems", *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 254–257, Springer, Saarbrücken, Germany, 2011.

28. Basin, D., M. Harvan, F. Klaedtke and E. Zălinescu, "Monpoly: Monitoring Usage-Control Policies", *Runtime Verification: Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers 2*, pp. 360–364, Springer, San Francisco, CA, USA, 2012.

29. Meredith, P. O., D. Jin, D. Griffith, F. Chen and G. Roşu, "An Overview of the MOP Runtime Verification Framework", *International Journal on Software Tools for Technology Transfer*, Vol. 14, No. 3, pp. 249–289, 2012.

30. Reger, G., H. C. Cruz and D. Rydeheard, "MarQ: Monitoring at Runtime with QEA", *Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21*, pp. 596–610, Springer, London, UK, 2015.

31. Ulus, D., "Montre: A Tool for Monitoring Timed Regular Expressions", *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*, pp. 329–335, Springer, Heidelberg, Germany, 2017.

32. Majumdar, R., I. Saha and M. Zamani, "Synthesis of Minimal-Error Control Software", *Proceedings of the Tenth ACM International Conference on Embedded Software*, pp. 123–132, Tampere, Finland, 2012.

33. Kennedy, J. and R. Eberhart, "Particle Swarm Optimization", *Proceedings of ICNN'95-International Conference on Neural Networks*, Vol. 4, pp. 1942–1948, IEEE, Perth, Australia, 1995.

34. Darulova, E., V. Kuncak, R. Majumdar and I. Saha, "Synthesis of Fixed-Point Programs", *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pp. 1–10, IEEE, Montreal, Canada, 2013.

35. Hwangbo, J., I. Sa, R. Siegwart and M. Hutter, "Control of a Quadrotor with Reinforcement Learning", *IEEE Robotics and Automation Letters*, Vol. 2, No. 4, pp. 2096–2103, 2017.

36. Goubault, E., S. Putot, P. Baufreton and J. Gassino, "Static Analysis of the Accuracy in Control Systems: Principles and Experiments", *Formal Methods for Industrial Critical Systems: 12th International Workshop, FMICS 2007, Berlin, Germany, July 1-2, 2007, Revised Selected Papers 12*, pp. 3–20, Springer, Berlin, Germany, 2008.

37. Barrett, C., R. Sebastiani, S. A. Seshia and C. Tinelli, "Satisfiability Modulo Theories", *Handbook of Satisfiability*, pp. 1267–1329, IOS Press, Amsterdam, Netherlands, 2021.

38. Burgio, P., M. Bertogna, N. Capodieci, R. Cavicchioli, M. Sojka, P. Houdek, A. Marongiu, P. Gai, C. Scordino and B. Morelli, "A Software Stack for Next-Generation Automotive Systems on Many-Core Heterogeneous Platforms", *Microprocessors and Microsystems*, Vol. 52, pp. 299–311, 2017.

39. Forsberg, B., D. Palossi, A. Marongiu and L. Benini, "GPU-Accelerated Real-Time Path Planning and the Predictable Execution Model", *Procedia Computer Science*, Vol. 108, pp. 2428–2432, 2017.

40. Yang, M., "Avoiding Pitfalls When Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems", *Proceedings of the 30th Euromicro Conference on Real-Time Systems*, Barcelona, Spain, 2018.

41. Amert, T., N. Otterness, M. Yang, J. H. Anderson and F. D. Smith, "GPU Schedul-

ing on the NVIDIA TX2: Hidden Details Revealed", *2017 IEEE Real-Time Systems Symposium (RTSS)*, pp. 104–115, IEEE, Paris, France, 2017.

42. Huang, J., C. Erdogan, Y. Zhang, B. Moore, Q. Luo, A. Sundaresan and G. Rosu, "ROSRV: Runtime Verification for Robots", *Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings 5*, pp. 247–254, Springer, Toronto, ON, Canada, 2014.

43. Adam, S., M. Larsen, K. Jensen and U. P. Schultz, "Towards Rule-Based Dynamic Safety Monitoring for Mobile Robots", *Simulation, Modeling, and Programming for Autonomous Robots: 4th International Conference, SIMPAR 2014, Bergamo, Italy, October 20-23, 2014. Proceedings 4*, pp. 207–218, Springer, Bergamo, Italy, 2014.

44. Ferrando, A., R. C. Cardoso, M. Fisher, D. Ancona, L. Franceschini and V. Mascardi, "ROSMontoring: A Runtime Verification Framework for ROS", *Towards Autonomous Robotic Systems: 21st Annual Conference, TAROS 2020, Nottingham, UK, September 16, 2020, Proceedings 21*, pp. 387–399, Springer, Nottingham, UK, 2020.

45. Stadler, M. and M. Vierhauser, "ROMoSu: Flexible Runtime Monitoring Support for ROS-Based Applications", *2023 IEEE/ACM 5th International Workshop on Robotics Software Engineering (RoSE)*, pp. 53–60, IEEE, Melbourne, Australia, 2023.

46. Perez, I., A. Mavridou, T. Pressburger, A. Will and P. J. Martin, "Monitoring ROS2: From Requirements to Autonomous Robots", *arXiv Preprint arXiv:2209.14030*, 2022.

47. Yamaguchi, T., B. Hoxha and D. Ničković, "RTAMT–Runtime Robustness Monitors with Application to CPS and Robotics", *International Journal on Software Tools for Technology Transfer*, Vol. 26, No. 1, pp. 79–99, 2024.

48. Caldas, R., J. A. P. García, M. Schiopu, P. Pelliccione, G. Rodrigues and T. Berger, "Runtime Verification and Field Testing for ROS-Based Robotic Systems", *arXiv Preprint arXiv:2404.11498*, 2024.

49. Xia, C., Y. Zhang, L. Wang, S. Coleman and Y. Liu, "Microservice-Based Cloud Robotics System for Intelligent Space", *Robotics and Autonomous Systems*, Vol. 110, pp. 139–150, 2018.

50. Lumpp, F., M. Panato, F. Fummi and N. Bombieri, "A Container-Based Design Methodology for Robotic Applications on Kubernetes Edge-Cloud Architectures", *2021 Forum on Specification & Design Languages (FDL)*, pp. 01–08, IEEE, Antibes, France, 2021.

51. Dawarka, V. and G. Bekaroo, "Building and Evaluating Cloud Robotic Systems: A Systematic Review", *Robotics and Computer-Integrated Manufacturing*, Vol. 73, p. 102240, 2022.

52. *Eclipse SDV*, `https://sdv.eclipse.org/`, accessed on June 23, 2024.

53. *SOAFEE*, `https://www.soafee.io/`, accessed on June 23, 2024.

54. Wen, L., M. Rickert, F. Pan, J. Lin and A. Knoll, "Bare-Metal vs. Hypervisors and Containers: Performance Evaluation of Virtualization Technologies for Software-Defined Vehicles", *2023 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1–8, IEEE, Anchorage, AK, USA, 2023.

55. Betz, T., L. Wen, F. Pan, G. Kaljavesi, A. Zuepke, A. Bastoni, M. Caccamo, A. Knoll and J. Betz, "A Containerized Microservice Architecture for a ROS 2 Autonomous Driving Software: An End-to-End Latency Evaluation", *arXiv Preprint arXiv:2404.12683*, 2024.

56. Lumpp, F., F. Fummi, H. D. Patel and N. Bombieri, "Enabling Kubernetes Orchestration of Mixed-Criticality Software for Autonomous Mobile Robots", *IEEE*

*Transactions on Robotics*, 2023.

57. Ulus, D., "Online Monitoring of Metric Temporal Logic Using Sequential Networks", *arXiv Preprint arXiv:1901.00175*, 2019.

58. Ulus, D., "Timescales: A Benchmark Generator for MTL Monitoring Tools", *Proceedings of the Conference on Runtime Verification (RV)*, pp. 402–412, Springer, Porto, Portugal, 2019.

59. *Zenoh-C on GitHub*, `https://github.com/eclipse-zenoh/zenoh-c`, accessed on June 23, 2024.

60. Liang, W.-Y., Y. Yuan and H.-J. Lin, "A Performance Study on the Throughput and Latency of Zenoh, MQTT, Kafka, and DDS", *arXiv Preprint arXiv:2303.09419*, 2023.

61. *CARLA Simulator*, `https://carla.org/`, accessed on June 23, 2024.