Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Institute of Computer Science
Department XVII
Robotics
Prof. Dr. Andreas Nüchter

ROS2 Over Wide-Area-Networks: Teleoperation of Mobile Robots in an E-Learning Environment

Master Thesis in Computer Science
submitted by

Daniel Schott

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Institute of Computer Science
Department XVII
Robotics
Prof. Dr. Andreas Nüchter

ROS2 Over Wide-Area-Networks: Teleoperation of Mobile Robots in an E-Learning Environment

Master Thesis in Computer Science
submitted by

Daniel Schott
born September 26, 1994 in Crailsheim

Written at
Chair of Robotics (Computer Science XVII)
Bavarian Julius–Maximilians–University of Würzburg

Supervisor:
Prof. Dr. Andreas Nüchter
Prof. Dr.-Ing. Sergio Montenegro

Submission on:
September 30, 2025

Danksagung

Ich möchte mich herzlich bei allen bedanken, welche mich bei der Erstellung dieser Masterarbeit unterstützt haben.

Mein besonderer Dank gilt hierbei Prof. Dr. Andreas Nüchter für die gegebene Möglichkeit, diese Arbeit zu verfassen, sowie Prof. Dr.-Ing. Sergio Montenegro für die Übernahme des Zweitgutachtens.

Des Weiteren gilt mein besonderer Dank Dr. Christian Herrmann und MSc. Lakshminarasimhan Srinivasan für die fachliche Betreuung meiner Arbeit. Ebenso danke ich Dr. Michael Bleier und MSc. Christoph Liebender für wertvolle Hinweise und Anregungen. Die Betreuung, wie auch die gegebenen Anregungen, haben einen wichtigen Beitrag zum Gelingen dieser Arbeit geleistet.

Zuletzt gilt mein Dank der EduArt Robotik GmbH, insbesondere Markus Fenn und Christian Wendt, für ihre hilfreiche Unterstützung bei der Lösung von Problemen mit den Eduard Robotern, die im Rahmen dieser Masterarbeit eingesetzt wurden.

Zusammenfassung

Mobile Roboter gewinnen in Industrie, Forschung und Alltag zunehmend an Bedeutung, wobei sich das Robot Operating System 2 (ROS2) als de-facto Industriestandard für die Entwicklung einer Vielzahl von Robotern etabliert hat. Infolgedessen gewinnt ROS2 auch im Bereich der akademischen Lehre zunehmend an Bedeutung, wobei der Fokus insbesondere darauf liegt, wie mobile Roboter mithilfe von ROS2 implementiert und wie Interaktionen mit diesen gestaltet werden können, um komplexe Aufgaben wie etwa Simultaneous Localization and Mapping zu bewältigen. Vor diesem Hintergrund ergibt sich eine starke Motivation, das Telelab, eine vom Lehrstuhl für Robotik der Universität Würzburg betriebenen E-Learning Plattform, auf ROS2 zu migrieren, um nicht nur die Vermittlung von ROS2-bezogenen Inhalten zu ermöglichen, sondern zugleich die Teleoperation von ROS2-basierten Robotern zu realisieren.
Um dieses Ziel zu erreichen bestand der erste Schritt in einer eingehenden Analyse der Kommunikations-Middleware von ROS2, des Data Distribution Service (DDS) in Verbindung mit dem Real-time Publish-Subscribe Protocol (RTPS), das jedoch in seiner nativen Ausprägung auf den Einsatz in einem Local-Area-Network (LAN) beschränkt ist, und die daraus resultierende Notwendigkeit, eine Lösung zu entwickeln, welche die Kommunikationsfähigkeit von ROS2 auf ein Wide-Area-Network (WAN) ausweitet. In diesem Zusammenhang wurden existierende Ansätze wie eProsima's DDS Router sowie Rosbridge hinsichtlich ihrer Eignung evaluiert, mit dem Ergebnis, dass beide Lösungen aufgrund erheblicher Performanzdefizite ungeeignet sind. Vor diesem Hintergrund wird im Rahmen dieser Arbeit eine neue Lösung für die WAN-basierte ROS2 Kommunikation vorgestellt: ROS2 Connect, welches auf WebSockets basiert und das gesamte Funktionsspektrum von ROS2, einschließlich Topics, Services, Actions, tf sowie Zeitsynchronisation unterstützt. Performanzmessungen haben darüber hinaus verdeutlicht, dass ROS2 Connect sämtliche evaluierten Alternativen sowohl hinsichtlich Stabilität als auch Latenz deutlich überlegen ist und somit die robuste Teleoperation ROS2-basierter Roboter über ein WAN im Kontext des Telelabs ermöglicht.
Darüber hinaus wurden zwei neue ROS2-basierte mobile Roboter mit unterschiedlichen kinematischen Konfigurationen in das Telelab integriert. Diese Integration schafft damit einen ersten, grundlegenden Schritt einer umfassenden Transformation des Gesamtsystems hin zu ROS2, welche zukünftige Erweiterungen sowie die curriculare Entwicklung von Remote-Lernmodulen und simulierten Umgebungen ermöglicht.
Zusammenfassend stellt diese Arbeit nicht nur eine leistungsstarke Lösung für die ROS2 Kommunikation über ein WAN bereit, sondern leistet darüber hinaus einen wesentlichen Beitrag zur Weiterentwicklung des Telelabs zu einer modernen, ROS2-zentrierten E-Learning Plattform für die Ausbildung im Bereich der mobilen Robotik.

Abstract

Mobile robots are becoming increasingly relevant in industry, research, and everyday life, with the Robot Operating System 2 (ROS2) emerging as a de facto industry standard for the development of a wide variety of robots. As a result, ROS2 is gaining greater presence in education, with a focus on how mobile robots can be implemented using ROS2 and how they can be interacted with in order to solve various tasks, such as Simultaneous Localization and Mapping. For this reason, there is strong motivation to migrate the Telelab, a robotics e-learning platform operated by the Chair of Robotics at the University of Würzburg, to ROS2, thereby enabling not only the teaching of ROS2-related content but also the teleoperation of ROS2-based robots.

To achieve this objective, the initial step was the examination of ROS2's communication middleware, Data Distribution Service (DDS) in conjunction with the Real-time Publish-Subscribe Protocol (RTPS), which is natively restricted to work across a Local-Area-Network (LAN), and the resulting necessity of developing a solution that extends the communication capabilities of ROS2 to a Wide-Area-Network (WAN). To this end, existing approaches such as eProsima's DDS Router and Rosbridge were evaluated with respect to their suitability, revealing that both are unsuitable due to significant performance drawbacks. In response, this work introduces a new solution for the problem of ROS2 over WAN, ROS2 Connect, which is based on WebSockets and support the full ROS2 spectrum including topics, services, actions, tf and time synchronization. Performance evaluations further demonstrated that ROS2 Connect significantly outperforms all other evaluated solutions in terms of both transmission stability and latency, thereby enabling robust teleoperation of ROS2-based robots over a WAN in the context of Telelab.

In addition, two new ROS2-based mobile robots, with different kinematic configurations, have been integrated into Telelab. This integration constitutes the initial step of the transformation of the Telelab infrastructure to ROS2, thereby establishing a sustainable foundation which enables future extensions and the curricular development of remote learning modules and simulated environments.

Overall, this work provides a high-performant solution for ROS2 communication over a Wide-Area-Network and further develops Telelab into a modern, ROS2-centered e-learning platform for mobile robotics education.

Contents

Chapter 1

Introduction

Mobile robots are ubiquitous in today's world and play an increasingly important role in our every day lives, whether in industrial manufacturing plants, fully autonomous warehouse systems, or in our homes in the form of robotic vacuum cleaners. Although these systems may at first appear to be simple devices, they are in fact sophisticated technological systems. A large proportion of autonomous robots today are implemented in a standardized manner, thereby ensuring a uniform way of interacting with them. At the core of this implementation lies the Robot Operating System 2 (ROS2), developed by the Open Source Robotics Foundation, which furthermore plays a significant role in both research and education.

Consequently, there is a growing interest in expanding the Telelab of the Chair of Robotics at the University of Würzburg, which is operated in collaboration with the Virtual University of Bavaria, not only by incorporating ROS2-related learning content but also by migrating its infrastructure to a ROS2 foundation. This Telelab, which has been previously described in works by Schott, is an e-learning platform designed to provide students from arbitrary geographical locations with the opportunity to deepen their knowledge in the field of mobile robotics by enabling the teleoperation of mobile robots over the internet.

However, this refocusing of Telelab with ROS2 as its foundation is subject to a fundamental challenge, which arises from the communication middleware Data Distribution Service (DDS) in conjunction with the Real-time Publish-Subscribe Protocol (RTPS), on which ROS2 is based. The underlying reason of this challenge is rooted in the fact that DDS, respectively, RTPS is specified only for communication within a Local-Area-Network (LAN), thereby impeding teleoperation of ROS2-based mobile robots from any geographical location over a Wide-Area-Network (WAN).

For this reason, the objective of this master thesis is to identify a performant and seamless solution to the problem of ROS2 communication over WAN and to integrate it into the existing Telelab infrastructure, with the goal of enabling the teleoperation of mobile robots. The teleoperating student should be able to make full use of ROS2 functionality when interacting with the robot, thereby creating the impression of being connected to the robot through a shared LAN. Furthermore, the integration of two new ROS2-based robots necessitates the partial redevelopment or modification of existing infrastructural Telelab components to establish a ROS2 foundation for

Telelab. Through these two achievements, the Telelab will provide the basis for the creation of ROS2-related learning content, as exemplified by the planned course "Introduction to ROS", with the added possibility of teleoperating the two newly integrated ROS2-based robots over a Wide-Area-Network.

Accordingly, chapter 2 first presents the theoretical background of ROS2, DDS and RTPS, as well as a detailed description of the issues that prevent ROS2 from functioning over a WAN natively. This is followed by a discussion of two existing solutions, both of which share the advantage of being applicable in the context of Telelab with only minimal configuration or implementation effort. Finally, a new solution for achieving ROS2 communication over WAN, ROS2 Connect, is introduced, which not only outperforms the existing solutions but can also be seamlessly integrated into the Telelab infrastructure.

Chapter 3 then introduces the two new robots to be integrated, together with their properties, as well as the hardware and software modifications made to them. Finally, the implemented and modified infrastructural components are presented, which serve the purpose of completing the integration of the robots into the Telelab and thereby realizing the overall objective.

Chapter 2

ROS2 over Wide-Area-Networks

This chapter proposes a method for achieving Robot Operating System 2 (ROS2) communication between two host systems over a Wide-Area-Network (WAN). Enabling ROS2 WAN communication is a key element of this work, as it is essential for the teleoperation of mobile robots.
To achieve this goal and to show why an explicit implementation of ROS2 over WAN is necessary at all, a background on ROS2 and its communication middleware Data Distribution Service (DDS) as well as the Real-time Publish-Subscribe Protocol (RTPS) is first provided.
On this basis, the problem is then defined in detail and various existing solutions are discussed and evaluated in terms of their suitability for the intended use in the context of Telelab.
The method developed in this work is then presented, its implementation explained and its suitability and performance compared to the existing solutions already presented.

2.1 Background

2.1.1 Data Distribution Service (DDS)

Data Distribution Service (DDS) is a specification by the Object Management Group that describes a Data-Centric Publish-Subscribe (DCPS) model [1]. This model finds application in the context of distributed applications, wherein it facilitates asynchronous communication across a network infrastructure [1]. In this scenario, an entity has the ability to publish data of a specific type and another entity can subscribe to it if it is interested in that data [1]. Given that ROS2 employs DDS as its communication middleware, which facilitates scalable multi-robot communication, real-time behavior, zero-copy mechanisms, and best-in-class security, the following will outline some of the fundamental concepts of DDS, as they will be found later in the context of ROS2 [1, 2].

2.1.1.1 Publisher and Subscriber

Since DDS describes a DCPS model, the concept of Publisher & Subscriber exists for asynchronous data exchange between entities, see Fig. 2.1. In this context, the responsibility for making data from a given topic available falls upon the Publisher. In

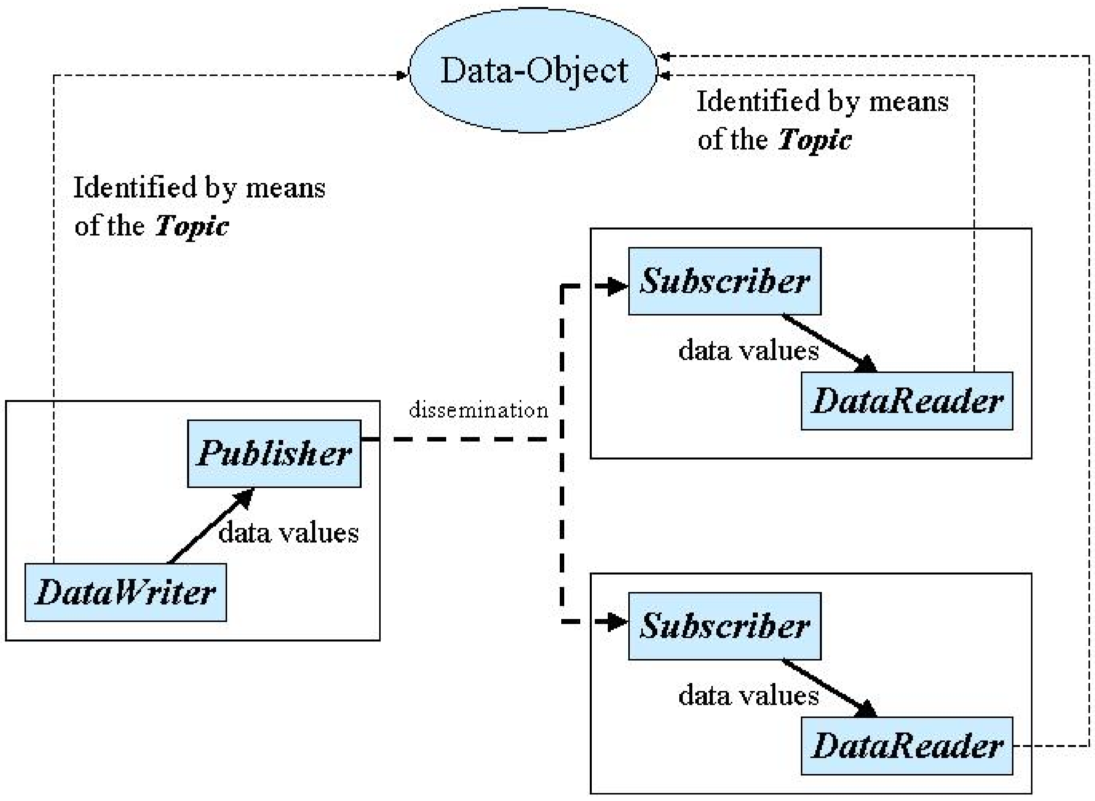


Fig. 2.1: DDS Publisher & Subscriber Concept [1]

contrast, the responsibility for receiving data of a given topic falls upon the Subscriber [1].
A Publisher is assigned a set of Quality of Service (QoS) policies and is associated with a DataWriter, which is associated with a topic [1]. To publish data, a user application must communicate with the DataWriter, which in turn passes the data to be published to the Publisher, which subsequently publishes the data according to its QoS policies [1]. The definition of the data and its type is derived from the topic.
A Subscriber, too, is assigned a set of QoS policies; however, it is linked with a DataReader, which, like the DataWriter, is associated with a topic [1]. To access the received data, the user application must communicate with the DataReader [1].

In order to receive or send data, user application must therefore create a Subscriber and a DataReader or a Publisher and a DataWriter, both for a certain (and common) topic [1].

2.1.1.2 Topic

The concept of a topic is used to allow the distinct identification of data objects between the Publisher and Subscriber [1]. A topic is defined as a combination of a name that is unique with respect to the domain, a data type and a set of QoS policies [1]. The QoS policies of the topic are combined with the policies of the Publishers and Subscribers in order to influence their behavior [1]. Topics are defined through the Interface Definition Language (IDL), specified by the Object Management Group, a purely descriptive language that defines grammatical constructs used to describe data types and interfaces [3]. IDL, in combination with DDS, provides a mapping between platform and definition, so that a user application can use different DDS implementations, since the definition of the topic is independent of the DDS implementation [1, 3].

2.1.1.3 Quality of Service (QoS)

DDS employs a set of QoS policies that, when utilized in conjunction, affect the behavior of DDS entities such as Publishers and Subscribers [1]. A set of these QoS policies is referred to as QoS [1]. However, in this thesis, the term "QoS Definition" is employed when referring to a set of QoS policies. It is important to note that not all QoS policies are compatible [1]. To ensure a seamless exchange of data between a Publisher and Subscriber, they both must use compatible or, in the best case, identical QoS policies [1].
An example is the DurabilityQosPolicy [1]. This policy dictates the retention of published data, ensuring that Subscribers who join the network at a later point can access the data published earlier ("white board" principle) [1]. However, if a Publisher sets the DurabilityQosPolicy to VOLATILE_DURABILITY_QOS, yet a Subscriber expects TRANSIENT_LOCAL_DURABILITY_QOS, the Subscriber who joins the network at a subsequent point in time will be unable to access any previously published data. [1] However, in instances of incompatibility between Publisher and Subscriber QoS policies, complete prevention of communication is also possible [1]. One such example is the ReliabilityQosPolicy, which specifies whether data should be retransmitted in the event of loss or corruption [1].

2.1.1.4 Domain

A domain is defined as a logical communication separation layer [1]. Publishers, Subscribers and Topics are attached to a DomainParticipant (see Fig. 2.2), denoting their affiliation with a particular domain [1]. In this context, the DomainParticipant functions as a factory for Publishers, Subscribers, and Topics [1]. The communication between entities such as Publishers and Subscribers is only possible within the context of the same domain [1]. This concept facilitates the isolation of DDS communication from multiple user applications that share a common physical network; the domain functions as a Virtual Private Network (VPN) [1]. To achieve this, each domain is identified by a unique domainId which is defined as an integer value, generally confined to the range of 0 to $232 - 1$ due to the ephemeral port range [1, 2].

2.1.2 Real-time Publish-Subscribe Protocol (RTPS)

As DDS only describes a DCPS model, but not how data is exchanged between Publishers and Subscribers, the Real-time Publish-Subscribe Protocol (RTPS), specified

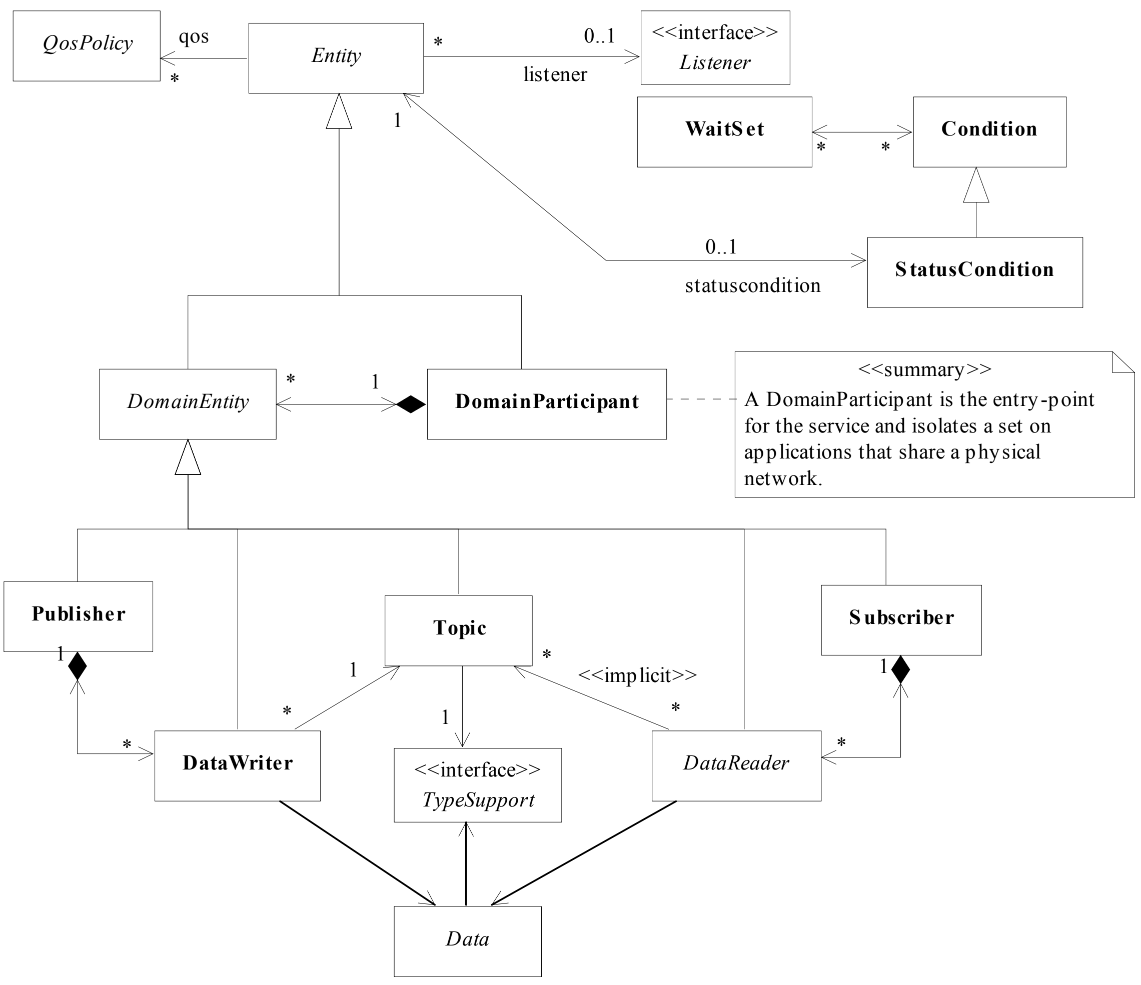


Fig. 2.2: DDS DomainParticipant [1]

by the Object Management Group, will also be mentioned here shortly. Real-time Publish-Subscribe Protocol (RTPS) is a DDS wire protocol that facilitates communication among vendor specific DDS implementations, optimizing the utilization of Quality of Service [4]. The two most common DDS implementations in the context of ROS2, eProsima's FastDDS and the Eclipse Foundation's CycloneDDS, both implement RTPS.

RTPS is mapped to DDS via the Writer and Reader structures, which represent endpoints, see Fig. 2.3, that are capable of communicating with each other by transmitting RTPS Messages [4]. RTPS Messages are hereby composed of a header and a series of Submessages, see Fig. 2.4, which also include a header and several Submessage-Elements [4].
In the context of RTPS, Discovery is defined as the process by which participants independently discover the existence of other participants through metatraffic and obtain information about their configuration and as a result have the ability to configure their local Writers and Readers to enable the exchange of data [4]. RTPS specifies a discovery protocol for this purpose, which is divided into the Participant Discovery

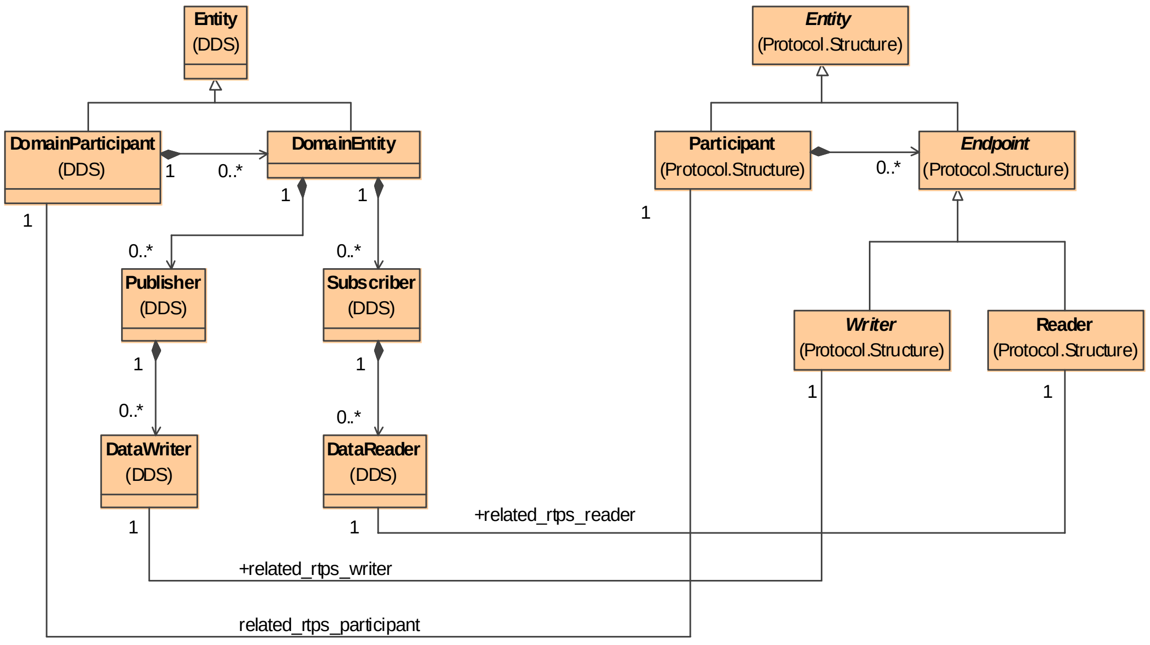


Fig. 2.3: Mapping between RTPS and DDS [4]

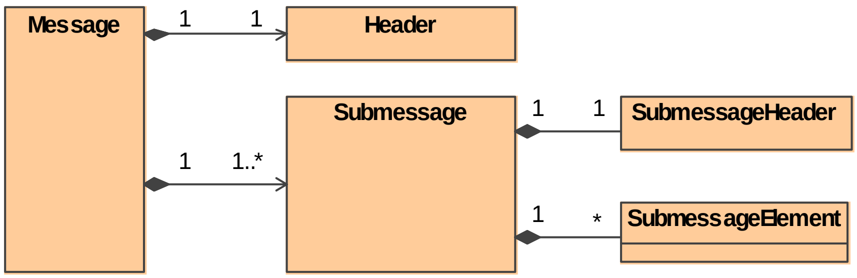


Fig. 2.4: RTPS Message structure [4]

Protocol (PDP), which specifies how Participants discover each other on the network, and the Endpoint Discovery Protocol (EDP), which specifies how discovered Participants exchange information about their endpoints [4]. However, given the possibility for implementing multiple vendor-specific PDPs and EDPs, each RTPS implementation must implement the Simple Participant Discovery Protocol (SPDP) and the Simple Endpoint Discovery Protocol (SEDP) in order to operate across implementations [4]. RTPS proposes the utilization of multicast for Discovery while utilizing UDP/IP for data transport [4]. Despite this, it should be noted that certain DDS implementations, including FastDDS, also offer the utilization of unicast for discovery and TCP/IP for transport [5].

2.1.3 Robot Operating System 2 (ROS2)

The Robot Operating System, despite its name, is best described as a robotics framework, as a set of libraries and tools or as a middleware that allow robots of all kinds

to be implemented in a unified way without having to reinvent the wheel for every robot.
The development of ROS began before 2007 at the Stanford Artificial Intelligence Laboratory by Eric Berger and Keenan Wyrobek out of a need to create a standardized solution that would allow any robot to be integrated into projects without having to spend a lot of time programming the robots [6, 7]. In November 2007, the development moved to Willow Garage, a research laboratory for the development of robotics hardware and software, with the first early release of ROS 0.4 Mango Tango in February 2009 and the subsequent release of ROS 1.0 in January 2010 under the BSD License [6–9]. Even then, ROS 1.0 contained sophisticated algorithms for navigation, mapping, processing of sensor data such as cameras and laser scanners, and visualization tools such as RViz [9]. In 2012, Willow Garage founded the Open Source Robotics Foundation (OSRF) with the mission to "support the development, distribution, and adoption of open source software for use in robotics research, education, and product development" [10], under which the development of ROS has taken place from 2013 to the present [11].
Due to changes in requirements resulting from the development of new technologies in robotics, ROS1 was no longer able to optimally cover all use cases. It was limited in terms of security, reliability in non-traditional environments, and real-time capabilities [2]. As a result, development began on ROS2, which was released in December 2017 with the first ROS2 release Ardent Apalone under the Apache License 2.0 [12]. Unlike ROS1, ROS2 is based on DDS, which it uses as its communication middleware.

ROS2 releases, also known as ROS2 distributions in reference to Linux, are linked to the Ubuntu release cycle [12]. A new ROS2 release is issued with each new .04 Ubuntu release, and Ubuntu Long Term Support (LTS) releases are also ROS2 LTS releases [12].
This thesis focuses on the ROS2 release Jazzy Jalisco, released in May 2024, for Ubuntu 24.04 (LTS). Support for the release Kilted Kaiju, released in May 2025, is also ensured.

2.1.3.1 ROS2 Architecture

As mentioned above, ROS2 uses DDS as its communication middleware, upon which several abstraction layers are built, see Fig. 2.5.
The lowest level is the direct DDS abstraction layer, the ROS Middleware (rmw), which provides abstractions to the actual DDS implementation by offering a communication interface [2]. The implementations for this interface are designed to be interchangeable and exist for various DDS implementations, including two of the most prominent ones: ePosima's FastDDS and the Eclipse Foundation's CycloneDDS [2].
The ROS Client Libraries (rcl) builds upon the rmw. The rcl, implemented in the C programming language, serves as an intermediate interface for which bindings can be implemented in different programming languages to provide ROS2 functionalities [2]. The two most prominent bindings are ROS Client Libraries for C++ (rclcpp) and ROS Client Libraries for Python (rclpy).

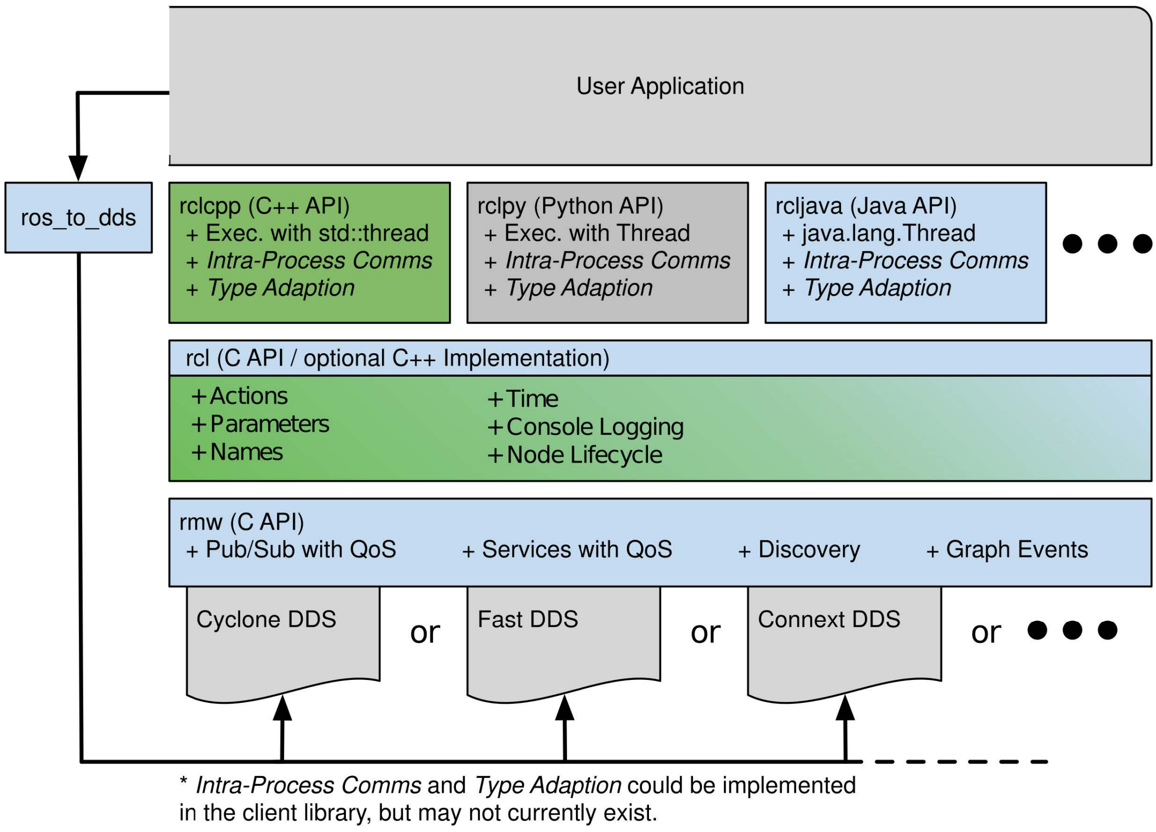


Fig. 2.5: ROS2 abstraction layers [2]

Based on the language specific bindings of the rcl, user applications can be implemented that can communicate with all levels of the ROS2 architecture and make use of the DDS nature of ROS2.

As this thesis discusses implementations in C++, only the rclcpp binding is used in the following.

2.1.3.2 ROS2 Concepts

A significant number of concepts in ROS2 originate from the underlying DDS, such as the concepts of Topics, Publisher and Subscriber, and Quality of Service. Additionally, ROS2 incorporates the concept of self-discovery, which is based on the Discovery concept of RTPS. However, ROS2 also introduced new concepts, such as the concept of a Node, see Fig. 2.6. In the following, the already known concepts are briefly addressed in the context of ROS2, and the ROS2-specific concepts are explained in more detail. It should be noted that this thesis cannot cover every ROS2 concept. Consequently, the focus is on the important ones in the context of this thesis. For a comprehensive overview of all concepts, see the papers by Macenski et al. [2, 13], as well as the official ROS2 Jazzy Jalisco documentation [12].

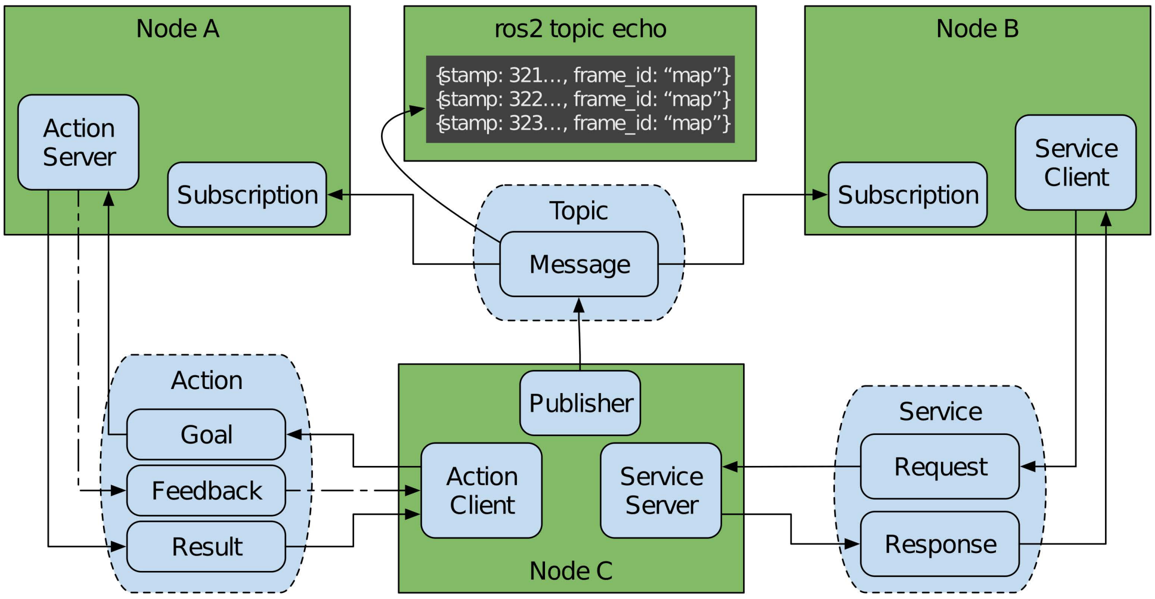


Fig. 2.6: ROS2 concepts [2]

Node

In the context of ROS2, a Node represents a participant in the ROS2 graph and functions as a unit of processing, typically designed to accomplish a single logical objective [12, 13]. A Node has the ability to communicate with other Nodes by the use of the rcl and is capable of discovering other Nodes on its own [12, 13]. To facilitate this communication, a Node employs Publishers and Subscribers, along with Services and Actions [12, 13]. A Node thus groups different and often several instances of ROS2 concepts under one namespace [12]. Additionally, Nodes utilize parameters, thereby enabling configuration at startup and external agents to modify their behavior during runtime [12]. In the context of DDS semantics, a Node thus corresponds to a DomainParticipant (see section 2.1.1.4) [14]. However, it should be noted that certain DDS implementations employ shared resources, thereby resulting in the mapping of multiple ROS2 Nodes to a single DDS DomainParticipant [14].
In C++, a Node is implemented as a class that inherits from `rclcpp::Node`. Consequently, Nodes are executables and are organized into packages. Such packages are typically managed by colcon, the ROS2 meta-build tool, which utilizes cmake respectively ament_cmake.

Publisher and Subscriber, QoS, Topics, Messages and Interfaces

The concepts of Publishers and Subscribers, as well as QoS, Topics, Messages and their definition, were adopted from DDS and RTPS (see section 2.1.1 and 2.1.2) and abstracted by the rmw and rcl.
ROS2 employs the ROS1 Message definition to specify the topic interface (data type), which is subsequently converted into IDL definitions (.idl files) [14]. The ROS1 Message definitions specify exactly how a Message is structured, i.e. what data fields it

has, e.g. [12]

```
uint32 field1
string field2
```

A topic can then use this definition as its data type. Consequently, a Publisher can utilize a topic to publish Messages of the Topics type, or a Subscriber can utilize a topic to receive Messages [12]. Both can be influenced in their behavior by assigned QoS policies [12].
In rclcpp, two types of Publishers and Subscribers are distinguished: typed Publishers and Subscribers, which hold received or outgoing Messages in C-structures and of which the type must be known at compile time; and generic Publishers and Subscribers, which hold Messages as raw byte vectors and of which the type must only be known at run time [14, 15].

The ROS2 Publishers and Subscribers are abstracted by the rmw and correspond to a DDS DataWriter, Publisher and Topic, respectively a DDS DataReader, Subscriber and Topic, and originate from a DDS Participant [14]. The rcl subsequently offers two interfaces: one for publishing data of a topic through the `publish` method call of a Publisher, and the other for subscribing to a topic by defining a callback function which is called upon data arrival [12, 15].

Services and Actions

The concept of Services and Actions in ROS2 enables the remote execution of code in form of a pre-defined procedure [12]. While both accept request and return parameters, a Service is executed synchronously, i.e. blocking, while an action is executed asynchronously, allowing its execution to be interrupted and providing continuous

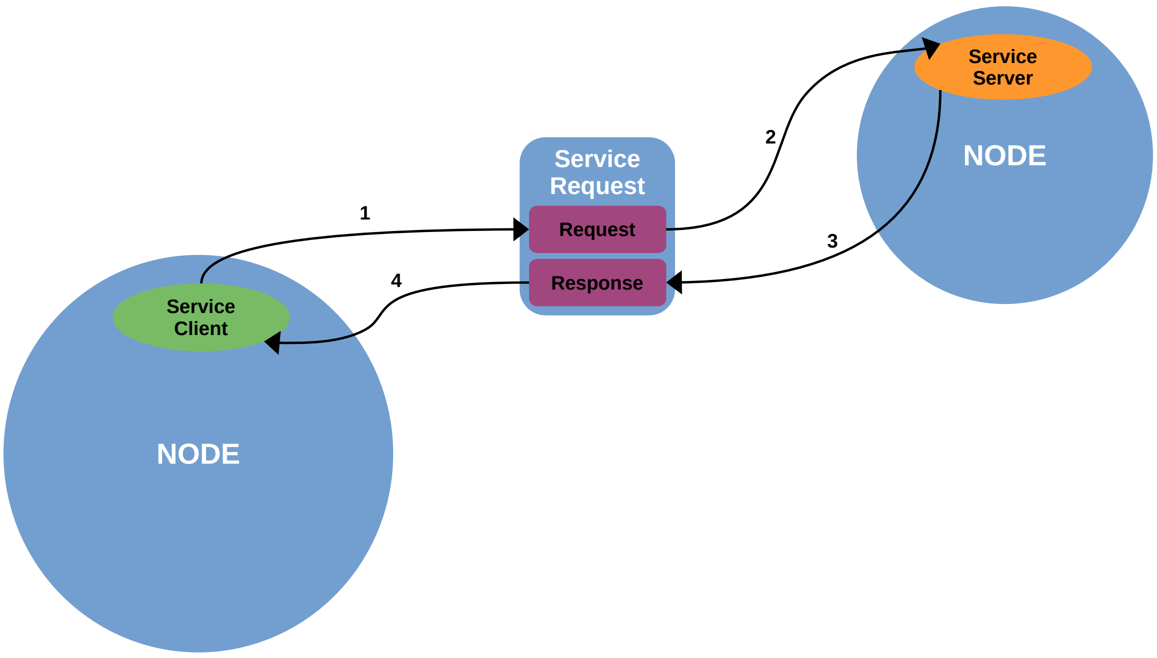


Fig. 2.7: ROS2 Service, based on [12]

feedback during its execution [12]. Services, therefore, should never be used for longer-lasting procedures; an example of a Service use is querying sensor values [12]. On the other hand, an Action is ideal for longer-lasting procedures and is often understood literally as an action, such as autonomously moving and docking a robot to a charging station [12]. Services and Actions are identified like Topics by a unique name and their interface is defined in the already known style of ROS1 Message definitions, e.g. (left for a Service and right for an Action) [12]:

```
int32 request   int32 goal
– – –           – – –
int32 response  int32 result
                – – –
                int32 feedback
```

ROS2 Services and Actions do not have a direct equivalent in DDS. Consequently, they are implemented internally via topics that are hidden from the user [14]. A Service is composed of one or many Service Clients, which initiate a request and wait for the response, and a single Service Server, which advertises a Service, receives a request, executes the defined procedure, and responses with a result (see Fig. 2.6 and 2.7) [12].
An Action is also composed of one or more Action Clients and a single Action Server with the same role distribution (see Fig. 2.6 and 2.8) [12]. The Client and Server utilize three Services internally [14]. The first Service facilitates the transmission of the request / goal (Goal Service); the second Service facilitates the transmission of the response / result (Result Service); and the third Service provides the capability to cancel the processing of a goal (Cancel Service) [14]. Furthermore, two topics are established on which the Action Server publishes and the Action Client subscribes, thus providing continuous feedback (Feedback Topic) and information about accepted goals (Status Topic) [14].

It should be noted that, in ROS2 Jazzy Jalisco, only Service Clients exist in a generic variant in rclcpp. Service Server as well as Action Servers and Clients do not, thus always need to be implemented with their type known at compile time [14, 15].

Executors and CallbackGroups
As previously described, DDS specifies an asynchronous DCPS model, which necessitates the processing of asynchronous callback handlers such as those from Subscribers, Service and Actions or Timers. In ROS2, this is accomplished through the utilization of Executors and CallbackGroups. These mechanisms facilitate the coordination of the order and timing of pending and asynchronous tasks, in addition to the execution of these tasks through on or multiple threads [12, 13]. For this purpose, a wait set is used to inform the executor thread(s) about available new messages or events [12, 13]. These messages or events are held in the middleware until they are processed in order to respect their respective QoS definitions [12]. Upon invoking the `spin()` function on the executor instance, it proceeds to evaluate the wait set and subsequently invokes the user-defined callback functions associated with the messages or events [12, 13].

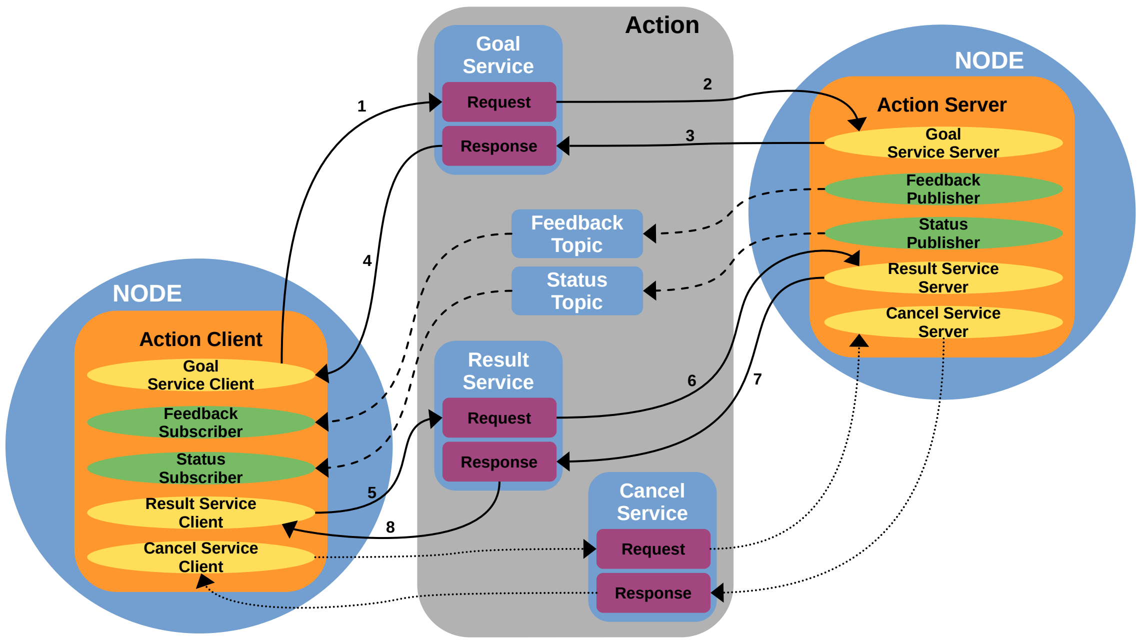


Fig. 2.8: ROS2 Action, based on [12]

At present, there are three distinct implementations of Executors in the ROS Client Libraries for C++ (rclcpp), with new implementations already under development [13]. By default, ROS2 utilizes the SingleThreadedExecutor, which operates using a single thread to process all messages and events [12, 13]. Furthermore, the StaticSingleThreadedExecutor demands that all Subscriptions, Timers and Service and Action Servers, and Clients to be created during Node initialization and remain static thereafter [12]. In contrast to the other two Executors, this approach eliminates the StaticSingleThreadedExecutor's need to continuously scan the Node for changes to its structure [12].
The third implementation is the MultiThreadedExecutor, which utilizes a thread pool with a predetermined number of threads [12, 13]. Consequently, the MultiThreadedExecutor is capable of processing an equivalent number of callbacks concurrently as the number of its threads [12, 13]. Furthermore, callbacks can be organized into CallbackGroups, which exist as mutually exclusive and reentrant types [12, 13]. The execution of callbacks from disparate callback groups can be executed concurrently [12]. Within a reentrant CallbackGroup, callbacks can be executed in parallel with respect to the group [12]. In contrast, within a mutually exclusive CallbackGroup, callbacks can only be executed sequentially with respect to the group [12].

Launch

In a typical ROS2 setup, several nodes usually run across multiple processes, all of which require different configuration parameters at startup [12]. In order to simplify the startup and configuration of such a rapidly growing and complex setup, ROS2 offers its own launch system named "launch". This concept enables the execution

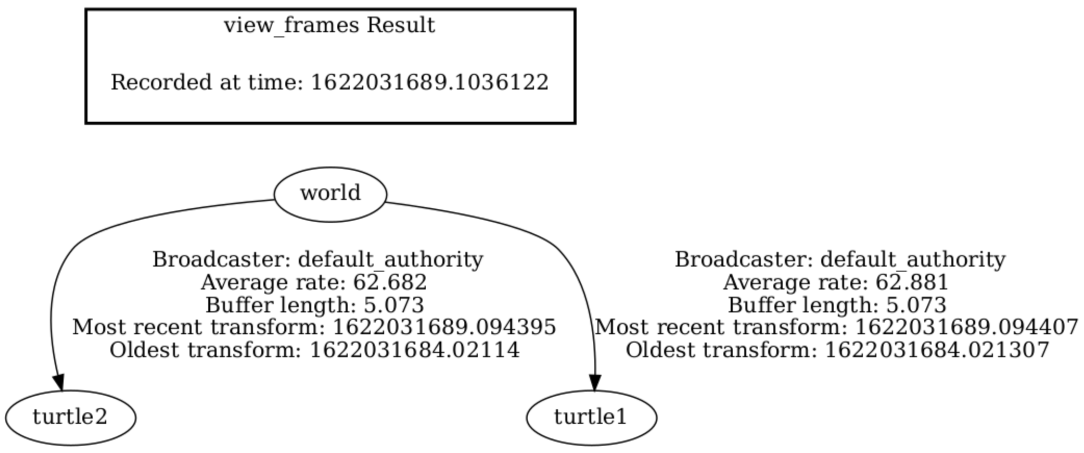

view_frames Result
Recorded at time: 1622031689.1036122
world
Broadcaster: default_authority
Average rate: 62.682
Buffer length: 5.073
Most recent transform: 1622031689.094395
Oldest transform: 1622031684.02114
Broadcaster: default_authority
Average rate: 62.881
Buffer length: 5.073
Most recent transform: 1622031689.094407
Oldest transform: 1622031684.021307
turtle2
turtle1

Fig. 2.9: tf2 tree recorded with tf2_tools [12]

of a single command that can initiate and configure the parameters of an arbitrary number of nodes [12]. In order to achieve this objective, a launch file is created that specifies the nodes to be initiated and the method by which they are to be started, in addition to the parameter values to be assigned to them [12]. Launch also facilitates the monitoring of processes executing the nodes, enabling a response to various events, such as the termination of a started Node [12]. Launch also implements signal handlers for Unix signals, including Signal Interrupt (SIGINT) and Signal Terminate (SIGTERM) [16]. These signal handlers forward the received signals to the individual processes / nodes initiated by launch [16]. As a result, a controlled and graceful shutdown of the started nodes is achieved.

Tf2

Tf2 is a core library of ROS2 that facilitates the storing of distinct coordinate frames and their interrelationships [17]. This enables the transformation of data recorded in one coordinate frame into another, without the necessity for every component of the overall system to be aware of the precise relationship between the coordinate frames [17]. In order to achieve this objective, tf2 maintains an acyclic and directed tree internally, in which the coordinate frames are represented by the nodes and the transformations are represented by the edges [17]. This facilitates a rapid search for the relationship between two coordinate frames, thereby enabling the calculation of their respective transformation [17]. Given that the tree is directed, it follows that the inverse transformation must be applied in order to ascend a path between the coordinate frames [17]. Fig. 2.9 shows an exemplary tf2 tree.
In tf2, each transformation is associated with a timestamp (`StampedTransform`) and has limited validity [17]. Consequently, each request for data from the tf2 tree must be timestamped [17]. This guarantees that the transformation corresponding to the request timestamp is returned [17].

Tf2 employs the concepts of the transformation Broadcaster (internally a Publisher) and Listener (internally a Subscriber) [17, 18]. Additionally, the method call `lookup Transform(...)` facilitates an explicit transformation lookup [12, 18]. Tf2 differentiates between two types of transformations: dynamic transformations and static transformations [18]. Dynamic transformations are defined as those transformations between two coordinates frames that can change over time. A notable example of this would be the transformation between the base coordinate frame of a robot and a map in which the robot moves. In contrast, static transformations are defined as those transformations between two coordinate frames that remain constant over time. An example of this would be the transformation between the base coordinate frame of a robot and its laser scanner. Two topics are employed by tf2 for this purpose. The /tf topic is designated for dynamic transformations, while /tf_static is used for static transformations [18]. The distinction between these two topics is rooted in the QoS policies, particularly with respect to the Durability policy [18]. /tf_static employs TRANSIENT_LOCAL, while /tf utilizes VOLATILE (see also 2.1.1.3) [18].

rosbag2
Rosbag2 is a command line tool that facilitates the recording of data published by Publishers, enabling its replay in its original form at a subsequent point in time [12]. To this end, rosbag2 accumulates the published data on one or more topics in a file, also known as a bag [12]. Subsequently, the recorded file (bag) can be replayed with rosbag2, which will result in the creation of all the recorded Publishers, which then publish their respective data with the same timing as was recorded [12]. This facilitates the archiving and sharing of data results from experiments.

Rosbag2 also has the capability to record Services and Actions; however, this functionality is conditional upon the presence of nodes that facilitate Service / Action introspection [12].

2.1.4 WebSocket Protocol

As WebSockets are to play a significant role in this thesis, they will be mentioned briefly here. WebSockets, better known as the WebSocket protocol, are a bidirectional communication protocol based on TCP/IP, which is standardized by the Internet Engineering Task Force (IETF) in Request for Comments (RFC) 6455 [19]. The WebSocket protocol is independent of the HTTP protocol, yet it employs it for the initial opening handshake, which is utilized for the connection upgrade [19]. Subsequent to the opening handshake, the two agents, one designated as server and the other as client, are capable of exchanging messages independently of each other via a single TCP/IP connection [19]. In the context of the WebSocket protocol, these messages are referred to as frames. The uniform header, which defines an Operation Code (opcode) among other things, and the actual payload data are the two important components of the frame [19]. A categorization of frames is possible based on the opcode, which distinguishes between text frames, binary frames, continuation frames, and control frames [19]. Text frames and binary frames are utilized for the transfer of payload data, whereby in the case of text frames, the payload is interpreted as UTF-8, and in the case of binary frames, the interpretation is left to the application that sends or receives the payload [19]. Continuation frames are used for fragmentation of larger payload data into multiple frames while control frames, on the other hand, do not carry and payload data but signal various events at the protocol level, such as closing the connection or ping/pong messages to maintain the connection and verify the viability of the connection partner [19].

Finally, it should be noted that payload data transmitted from a server agent to a client agent is not masked, however, payload data from a client agent to a server agent is required to be masked with an unpredictable 32-bit key [19].

2.2 Problem and Goal Definition

As previously outlined, teleoperation necessitates ROS2 communication between two host systems over a Wide-Area-Network (WAN). In this scenario, ROS2 nodes would be executed on a robot, thereby providing hardware abstraction. A user would then operate the robot from their computer, with both distributed over a WAN and being behind two or more Network Address Translation (NAT) routers.

This scenario presents an inherent challenge to the self-discovery mechanism of ROS2 nodes, which is based on DDS / RTPS discovery mechanisms. The underlying reason for this challenge is that DDS / RTPS utilizes multicast over UDP/IP for the Simple Participant Discovery Protocol (SPDP) and Simple Endpoint Discovery Protocol (SEDP) [4]. However, multicast utilizes the address range 224.0.0.0/24 (local network control block) by default, for which the specification states that incoming packages in this address range must not be forwarded by routers [20]. Additionally, the Time-To-Live (TTL) is configured to be 1 by default for multicast datagrams, resulting in the datagram's death at the first hop [21]. In summary, this results in multicast functioning exclusively across a local subnet by default. Consequently, its functionality is not applicable to networks distributed across a WAN behind NAT routers.

Therefore, it is necessary to explicitly implement ROS2 communication over WAN in order to achieve the desired teleoperation of robots within the context of Telelab. Three approaches are viable for achieving this objective: The discovery of ROS2 nodes can be facilitated through the utilization of vendor-specific DDS discovery mechanisms, the DDS or ROS2 messages can be intercepted and manually transmitted via an established network connection, or the two networks can be bridged in the form of a Virtual Private Network (VPN).

The goal hereby is to facilitate the transmission of messages from all Publishers and Subscribers, while concurrently enabling Service and Action calls. Also, it is essential to enable comprehensive tf2 functionality, encompassing the transfer of the tf2 tree and all its StampedTransformations. Moreover, the implementation of time synchronization mechanisms is crucial since, as previously outlined in section 2.1.3.2, tf2 transformations are always timestamped and have a limited validity [17]. At the same time, it is imperative to ensure that the overhead with regard to the connection is minimized to facilitate the efficient transfer of data at optimal speed while concurrently transferring multiple data packages in parallel. Finally, for the context of the Telelab, Authentication, Authorization and Accounting (AAA) mechanisms must also be implemented to ensure that only one user at a time (who has a corresponding reservation) has access to a specific robot and can only view explicitly permitted data and perform explicitly allowed operations.

The subsequent sections of this thesis thus present various existing solutions and evaluates their applicability within the use of Telelab. Furthermore, a new solution is proposed, its implementation explained and its suitability and performance compared to the existing solutions evaluated before.

2.3 Existing Solutions

This section presents existing solutions for the problem of ROS2 over WAN. A common characteristic of the presented solutions is that they require either minimal configuration or limited programming effort for achieving the previously defined objective. In total, four potential solutions are considered here. Of these, two are discussed in detail, while the remaining two are excluded due to their lack of suitability for the intended use case or externally imposed limiting factors.
It is important to note that numerous other approaches exist. However, these were excluded from consideration based on factors such as incompatibility with the objective to achieve, poor code quality, or lack of active maintenance.

In the subsequent section, the term "client" is employed to denote a user who intends to operate a robot with a computer and is thus part of the WAN behind a NAT router. Furthermore, eProsima's FastDDS is designated as "FastDDS", and the Eclipse Foundation's CycloneDDS as "CycloneDDS". Additionally, the term "DDS" is used to refer to a comprehensive DDS implementation that includes the implementation of RTPS.

2.3.1 VPN and Vendor-Specific Discovery

A Virtual Private Network (VPN) facilitates the integration of participants into a shared non-physical network [22]. In this scenario, the client and robot would be in a shared subnet, thereby enabling multicast and, consequently, ROS2 discovery to function without further configuration [22]. In principle, a VPN would therefore be a very favorable solution for ROS2 over WAN. The implementation of AAA mechanisms and sufficient network security to prevent the client from operating outside the subnet or addressing other hosts than the robot in the subnet would be sufficient.
Nevertheless, given the network operator's aversion to accommodate a VPN, this solution must be dismissed.

Another option to consider is the utilization of DDS vendor-specific discovery and transport mechanisms. As previously mentioned in section 2.1.2, DDS respectively RTPS implementations are required to support SPDP and SEDP to ensure interoperability. However, they are free to implement supplementary discovery and transport protocols or mechanisms. The two DDS implementations, FastDDS and CycloneDDS, both support unicast over UDP/IP or TCP/IP with fixed peers as an alternative to multicast [5, 23]. In contrast to multicast, unicast utilizes standard IP routing [24]. Consequently, unicast traffic is routed by routers across a WAN. However, NAT routers pose a significant challenge due to their tendency to block incoming unsolicited connections. In such cases, the utilization of port forwarding or other NAT traversal mechanisms becomes a necessity, yet these cannot be realized by all clients, for example, due to a lack of access to the NAT router or a network topology in which multiple NAT routers operate in series. Also, despite its theoretical functionality and implementation, CycloneDDS, presents challenges in its ability to function seamlessly with unicast. Consequently, unicast is not a viable solution for ROS2 discovery over WAN in the context of Telelab.
Another discovery mechanism is central discovery, as implemented by FastDDS's Discovery Server. This mechanism enables nodes to communicate with each other subsequent to the exchange of discovery information with a central server that is accessible to all participants [2, 5], see Fig. 2.10. Nevertheless, given that this discovery mechanism imposes the use of FastDDS on all participants, which may not be desirable, central discovery via the FastDDS Discovery Server is also not a viable solution. However, eProsima's DDS Router is based on a similar concept and is discussed in more detail in section 2.3.2.
There are also other vendor-specific discovery and transport mechanisms, nevertheless, all of these mechanisms either impose a binding to an explicit DDS implementation or have problems with NAT routers, therefore needing NAT traversal mechanisms. For this reason, vendor-specific discovery mechanisms of the two common DDS implementations, FastDDS and CycloneDDS, are not viable solutions and will not be considered further.

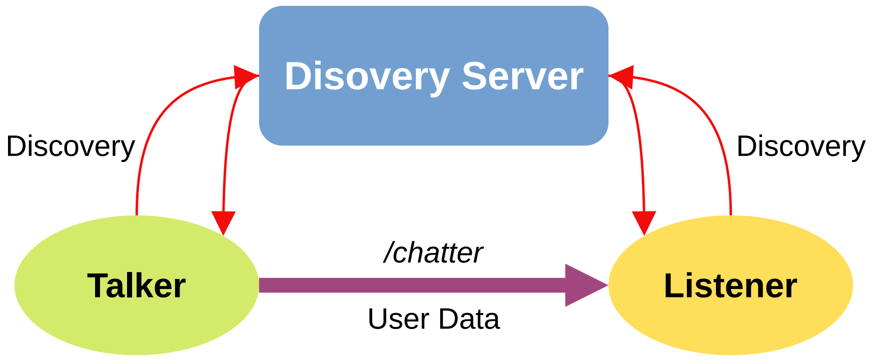


Fig. 2.10: FastDDS Discovery Server topology, based on [5]

2.3.2 eProsima's DDS Router

As the name suggests, eProsima's DDS Router is a software router for DDS respectively RTPS messages. The DDS Router facilitates the transmission of DDS / RTPS messages among DDS entities, irrespective of their geographical distribution or network configuration [25]. This enables DDS communication across different domains within a host system or subnet, or across a Wide-Area-Network [25]. And, as ROS2 employs DDS as its communication middleware, it enables ROS2 over WAN.
The DDS Router offers two topologies to fulfill this objective [25]:

1. In the first scenario, the DDS Router software is deployed on the two host systems, robot and client, which act as edge devices within the network. These instances then communicate directly with each other over a WAN, facilitating the exchange of DDS traffic.

2. In the second scenario, the DDS router software is deployed on the two host systems, robot and client, in addition to a third DDS Router instance on a host system that is public accessible to both, see Fig. 2.11. This public host functions as a Traversal Using Relays around NAT (TURN) repeater, thereby facilitating NAT traversal.

In the case of Telelab, the presence of multiple NAT routers between the robot and the client necessitates the selection of the topology via an additional TURN router.
In addition to the option of transmitting all DDS traffic via UDP/IP, DDS Router also offers the option of transmitting traffic via TCP/IP, which is preferable for WAN communication due to its higher reliability [25]. In this particular context, the DDS Router also provides the option of TLS over TCP for the purpose of encrypting DDS traffic during transport [25]. Furthermore, DDS Router incorporates built-in topic filtering, enabling configuration through allow and block lists, thereby restricting client interaction to approved topics [25]. Logging and monitoring are also provided, facilitating the documentation of client connections and the transmission of data [25].

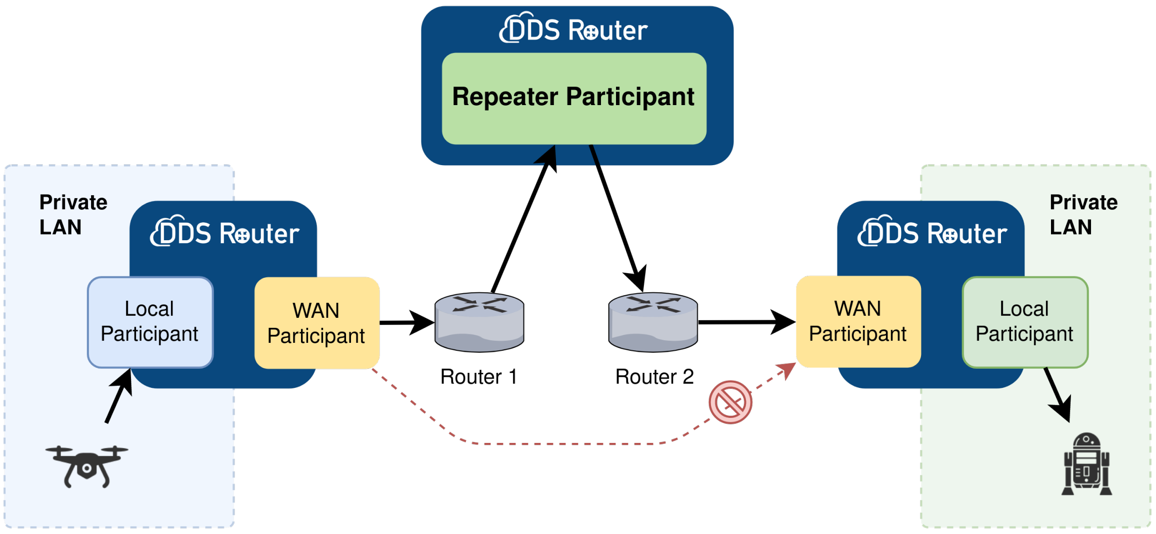


Fig. 2.11: DDS Router topology with TURN repeater [25]

Therefore, mechanisms for authorization and accounting are available. However, the implementation of the authentication mechanism must still be executed manually. Options here would either involve the implementation of targeted ad hoc configuration of firewall policies, or restriction of the connection option to selected clients via TLS certificates.
It should also be noted that the DDS Router functions optimally when used in conjunction with FastDDS, as both are developed by eProsima. Nevertheless, the DDS Router is also compatible with CycloneDDS, although this may potentially disrupt some vendor-specific functionalities of both.

2.3.2.1 Setup and Deployment

Since the DDS Router is a ready-made software solution, apart from authentication mechanisms, no programming effort is required, only minimal configuration.
According to the specifications outlined in the documentation [25], the code provided by eProsima must be compiled on all three hosts A, B and C (see Fig. 2.12) at the beginning in order to facilitate setup and deployment. The utilization of colcon, the ROS2 meta-build-tool, is recommended in this context. When attempting to compile with a GCC (GNU compiler collection) version 12.x, the error “maybe-uninitialized” may be encountered. This error can be converted into a warning by utilizing the cmake argument `-DCMAKE_CXX_FLAGS="-Wno-error=maybe-uninitialized"` to prevent the compilation process from aborting. However, this has no effect on the functionality of DDS Router an does not occur with GCC version 13.x.

Following the compilation of the code on all three hosts, a total of three configurations in YAML format must be created. Hereby, the configuration files from host A and B are identical, with the exception of potential variations in the local domainId. The configurations consist of defining two domain participants. On the one hand, a local participant is defined, thereby enabling the DDS Router to subscribe and publish the DDS traffic of the corresponding domain on the corresponding host. On the other hand, a WAN participant is defined, thus enabling the locally received DDS traffic to be transported (respectively relayed) via TCP to the other host(s) or received from them.

The configuration examples employed for the subsequent evaluation are documented in appendix A.1, as well as the commands needed to start the DDS Router instances. Within the scope of DDS Router, WAN participants that do not function as repeaters are uniformly designated as clients.

In the context of Telelab, it would later suffice to provide the user with instructions on how to compile the DDS Router and how to execute it with the correct configuration file.

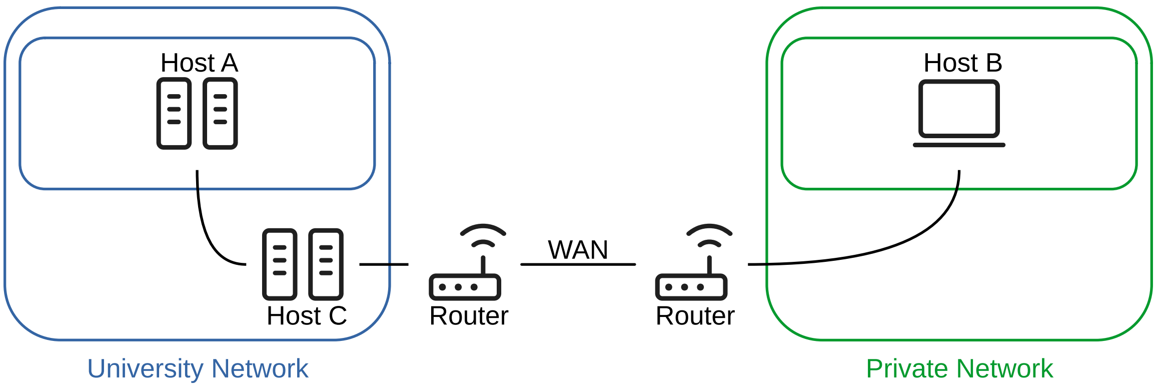


Fig. 2.12: Illustrated network topology for time measurements (Icons © Google (Apache 2.0) [26])

2.3.2.2 Performance Evaluation

A standardized procedure is employed for the evaluation, which is also applied in the remainder of this thesis and explained here. The objective of this procedure is to measure the time required for a ROS2 message of variable size to be transmitted from host A to host B and back. The measured transmission time can then be compared with the other ROS2 over WAN solutions presented in this thesis, as well as with the TCP data throughput between host A and B, which is determined using the iperf3 software. The procedure facilitates the transmission of ROS2 messages in both sequential and parallel. In the context of sequential transmission, this approach enables the extraction of performance metrics and overhead comparisons between the implementation and conventional TCP packets (iperf3). Conversely, in the context of parallel transmissions, conclusions can be drawn regarding the scalability of the implementation. This is particularly important in the context of Telelab, as a substantial amount of data, including laser scans, image data, status data and localisation data, must be transmitted in parallel or quasi-parallel.

The procedure is carried out as follows, whereby host A is located in the university network, while host B is located in a private network external to the university, both networks connected via a WAN. Host C is also located in the university network and considered a publicly accessible host (network edge). The topology is illustrated in Figure 2.12.

1. Firstly, host A and B are time-synchronized through the Network Time Protocol (NTP). Host C does not require explicit time synchronization because it does not record timestamps itself. Host A is synchronized with an NTP time server located in the same network which is synchronized with the Physikalisch-Technische Bundesanstalt (PTB). Host B is synchronized directly with the NTP time server of the PTB. The procedure presented here involves a round trip of the transmitted ROS2 message, enabling the calculation of errors in the time synchronization between host A and B as uncertainty provided that the clock of host B is monotonic during a measurement.

In the case of host A, a small clock skew of less than 1 millisecond to the NTP server is to be expected, as demonstrated by Novick et al. [27]. In the case of host B, however, the skew can vary between 1 millisecond and tens of milliseconds, as demonstrated by Novick et al. [27] and Mills [28]. However, the time measurements carried out in this thesis demonstrated a median time synchronization error of 0 ms between host A and B. The maximum time synchronization error was consistently below 1 ms.

2. Subsequently, the bandwidth of the connection between host B and host C is determined using iperf3. It is not necessary to measure the bandwidth of the connection between host A and C, as they are connected via an internal network with a steady bandwidth of 1 Gbit/s. This bandwidth is significant higher than the bandwidth between hosts B and C in all cases.

   The bandwidth test is initiated by launching iperf3 in server mode on host C. Subsequently, host B executes the measurement in client mode for a duration of 60 seconds per direction under the deactivation of the Nagle's Algorithm, enabling to identify asymmetries in connection speed.

   Alg. A.4 of Appendix A.2 shows the shell commands that must be executed on host B and C for the iperf3 bandwidth measurements.

3. Now the actual evaluation begins with sequential measurement. In order to execute this process, it is necessary to initiate the ROS2 Node *primary*, which has been developed for the purpose of time measurement, on host B. Concurrently, the ROS2 Node *secondary* is to be initiated on host A. The time measurement can be described as follows:

   primary publishes a new message of type "time_measurement/msg/Time Measurement" on the topic /primary, which is defined as follows:

```
# variable payload
uint8[] payload

# count to identify the message
int32 count
```

   These messages contain the value `count`, which facilitates their identification, as well as a variable `payload`. If the payload is empty (empty array), the size of the message serialized as a byte vector is 12 Bytes.

   secondary subscribes to the topic /primary, receives the published message, and subsequently republishes it on the topic /secondary, to which *primary* subscribes. Hereby, *primary* records two timestamps: "A", which corresponds to the moment at which a message is published, and "NN", which corresponds to the moment at which the message is received again. *secondary* records the timestamp "GG" at the time a message is received and "H" at the time it is republished. The timestamps are associated with the `count` value of the message,

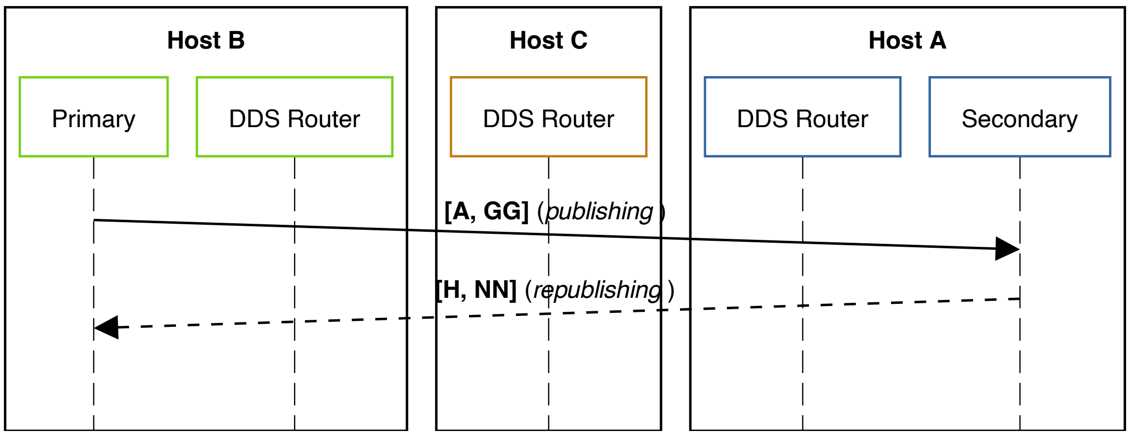


Fig. 2.13: DDS Router Time Measurement Sequence

thereby enabling the aggregation and subsequent evaluation of the timestamps recorded by the two systems.

The designation of the timestamps is such that subsequent evaluations may incorporate additional timestamps. However, given the nature of the DDS Router as a "closed system", it is only possible to capture the described four of the total 30 possible timestamps. Fig. 2.13 shows a sequence diagram with the time measurement process described above.

A total of 1000 messages are published by *primary* sequentially, and the respective timestamps are recorded, which is repeated for various different payloads. Subsequently, *primary* and *secondary* export the recorded time stamps to a JSON file, which is then subjected to statistic evaluation through the utilization of a Python script with the help of the Pandas library. This statistic evaluation includes the calculation of the arithmetic mean / average, median, minimum, and maximum transmission times as well as the standard deviation and Coefficient of Variation (CV), which is calculated by:

$$c_v = \frac{\sigma}{\mu} = \frac{\sqrt{\frac{1}{N} \cdot \sum_{i=1}^{N} (x_i - \mu)^2}}{\mu} \tag{2.1}$$

where:

c_v	coefficient of variation
σ	standard deviation
μ	arithmetic mean / average
N	sample size
x_i	the i-th measurement value

Hence, the Coefficient of Variation is a dimensionless statistic defined as the standard deviation divided by the arithmetic mean / average. Therefore, according to Abdi [29], this statistic expresses the magnitude of variability relative to the typical value for the time measurement of a certain payload size. In this work, CV functions as a scale-independent “relative jitter” metric, enabling direct comparison of latency stability across payload sizes whose absolute delays differ by orders of magnitude, as described by Malm et al. [30]. To this end, a rating scale is employed to interpret the value, which was derived from the works of Sundaresan et al. [31] and Lee at. al [32], with a CV from 0 to 10% indicating low variability, 10 - 30% indicating moderate variability, 30 - 50% indicating high variability, and > 50% indicating a very high variability.

The code and Python scripts developed and utilized for time measurement can be found on the data medium accompanying this thesis, see appendix C.

4. Subsequent to the execution of sequential time measurement, parallel time measurement is performed in a similar manner. Here, time measurements are performed for selected payload sizes with varying degrees of parallelization. As previously outlined, the above-mentioned time stamps are recorded, exported and subsequently subjected to statistical analysis.

   For this sake, parallel time measurement employs the two ROS2 Nodes *pprimary* and *psecondary*. These are characterized by their ability to concurrently process $n \in \mathbb{N}$ ROS2 time measurement messages as defined above. To this end, both nodes utilize a MultiThreadedExecutor (see section 2.1.3.2) with $n + 1$ threads to process the n incoming messages in parallel, and an additional thread for any additional events that may arise. Furthermore, n topics are used. *pprimary* publishes on the topics /primary_0 to /primary_$n-1$ and *psecondary* republishes on the topics /secondary_0 to /secondary_$n-1$. The distribution of roles among publications remains consistent with the sequential time measurements.

   As for the sequential measurement, a total of 1000 messages are published by *pprimary*, distributed over the n available Publishers in parallel, and the respective timestamps are recorded, which is repeated for selected payload sizes and, per payload size, with different degrees of parallelization, hence, different values for n.

Figure 2.14 presents the results of the sequential time measurement, whereby time measurements were carried out for messages with a total size ranging from 12 B to 200 kB. The graph displays the total message size on the logarithmic (base 10) X-axis and the total transmission time of a message (round trip) on the linear Y-axis. The average transmission time for each measured message size is indicated in blue, the median transmission time is indicated in orange, the range between the minimum and maximum measured transmission time is shaded in green, and the theoretically optimal transmission time based on the measured bandwidth (uplink 18.5 Mbit/s, downlink 58.6 Mbit/s) is indicated in gray.
It can be observed that the difference between the average measured transmission times and the theoretically optimal transmission times remain relatively constant, ranging from 38 ms to 43 ms for message sizes up to 6 kB. However, for message sizes exceeding 6 kB a steady increase of the difference can be observed, and for message sizes exceeding 60 kB, there is a pronounced increase in the difference, reaching a maximum at about 147 ms. These longer transmission times for larger messages can

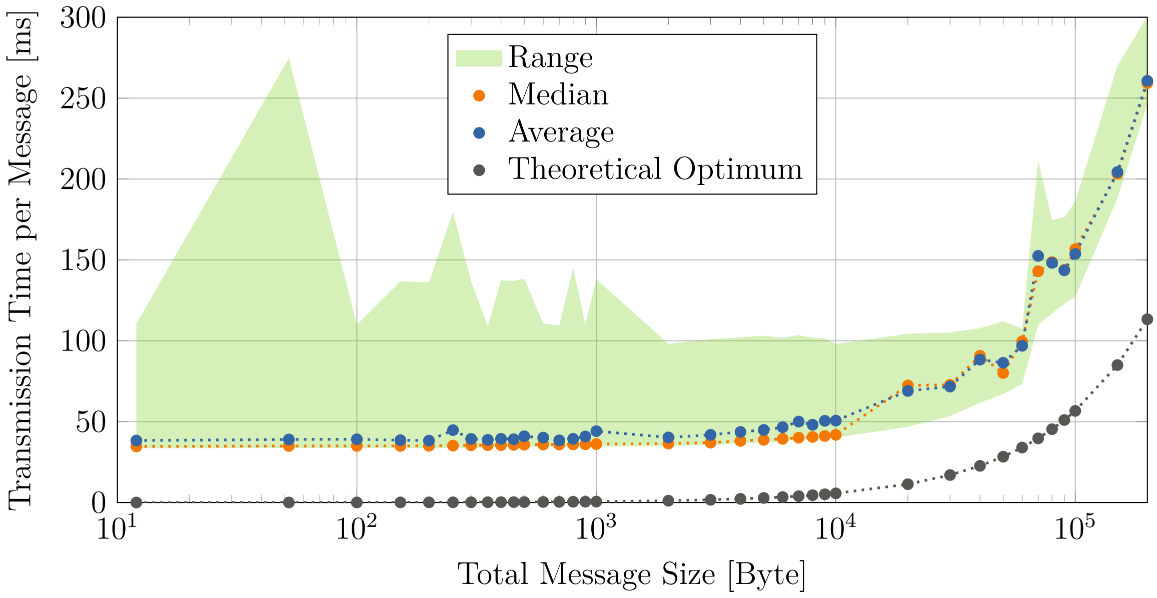


Fig. 2.14: Sequential Time Measurement of eProsima's DDS Router

be attributed to transmission / serialization delay and has also been observed in the ROS2 over WAN implementation introduced in this thesis as well as for the subsequent existing solution. In contrast to the new ROS2 over WAN solution presented in this thesis, ROS2 Connect, the observed increase in transmission times is not influenced by the default congestion control algorithm employed be the Linux kernel (see section 2.4.10).

Furthermore, it can be seen that average and median transmission times are very close to each other and are close to the recorded minimum transmission times, despite the substantial range (i.e., the difference between the minimum and maximum recorded transmission times) observed among the individual measurements, which can reach up to 200 ms. This shows a distribution of the measured transmission times that is skewed towards the left.

Figure 2.15 illustrates this again by a histogram with 20 bins depicting the 1000 measured values of a time measurement for messages with a total size of 2 kB on the left and an X-Y line plot of these 1000 measured values on the right.

The histogram indicates that nearly 900 of the measured 1000 values are concentrated around 36 ms. However, the presence of outliers is also evident, with a notable concentration in the 70 ms range, and a smaller number also present in the 100 ms range. The X-Y plot provides a visual representation of these outliers, as it reveals the presence of substantial spikes, with the Coefficient of Variation of the measured values being 28.29% indicating moderate variability to high variability.

The mean CV over all measured message sizes was also in this range at 29.53% (median at 31.12%), with a minimum of 2.85% at 200 kB and a maximum of 52.22% at 252 B. Given that a high CV was not measured for the subsequent existing ROS2 over WAN solution or the new solution introduced in this thesis, and that a lower value could not be measured by repeated measurements even with the client being in different

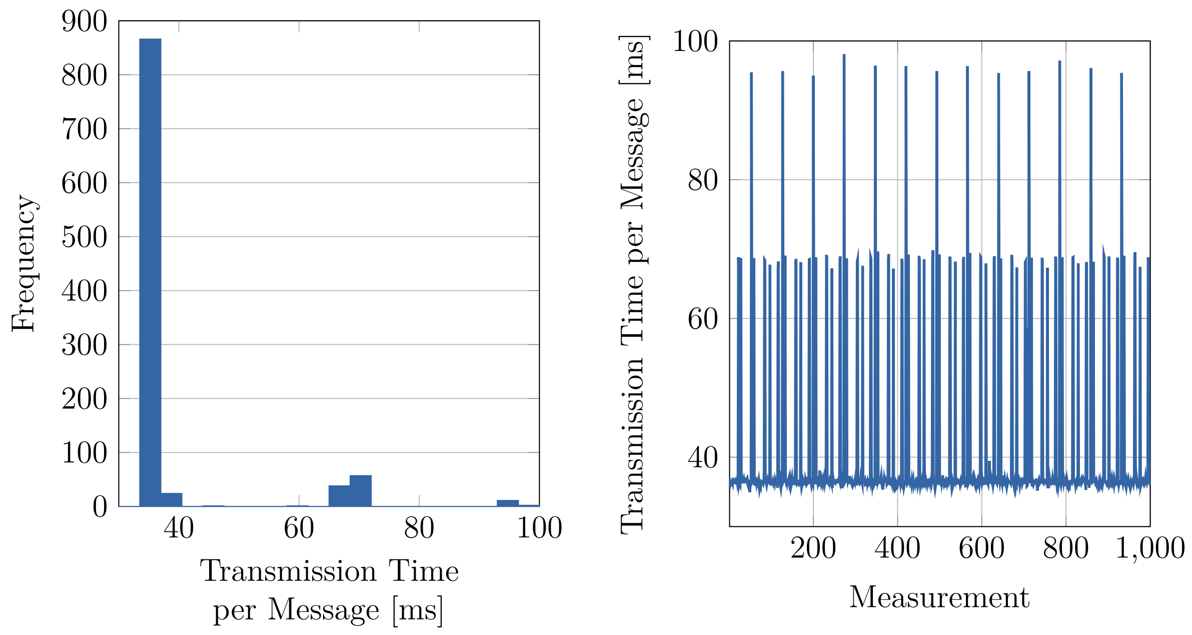


Fig. 2.15: Left: Distribution of time measurement values, Right: Plot of time measurement values, both for 2 kB total message size

networks, it must be assumed that this high variability of the round-trip times is an implementation-related phenomenon of the DDS Router.

Another phenomenon that can be observed is an increase in the average transfer time when the DDS Router instances are run for an extended period and several time measurements are conducted consecutively without restarting the instances after each run, which is illustrated by Fig. 2.16.
In this case, 10 time measurements were performed in succession for a total message size of 1 kB. A steep increase in the average and median transmission time was observed with the sixth measurement. The eighth time measurement showed a slight decrease in the median transmission time, while the average remained relatively constant. The minimum transmission time exhibited a negligible increase, while the maximum time demonstrated fluctuations that were already previously observed in Fig. 2.14. The CV was merely constant at around 38%. This indicates a shift in the distribution, favoring the outliers that had previously been observed, which become increasingly prevalent with the sixth measurement. This phenomenon again is not attributable to measurement errors or network issues, and can be reliably reproduced.

Another behavior of the DDS Router is that messages with a total size of 100 kB or more cause significant problems. In this scenario, the DDS Router instance of host B repeatedly stops responding and is unable to transfer the entire message, and can only be terminated via a Signal Kill (SIGKILL). Messages exceeding 200 kB in size were not transmitted at all.

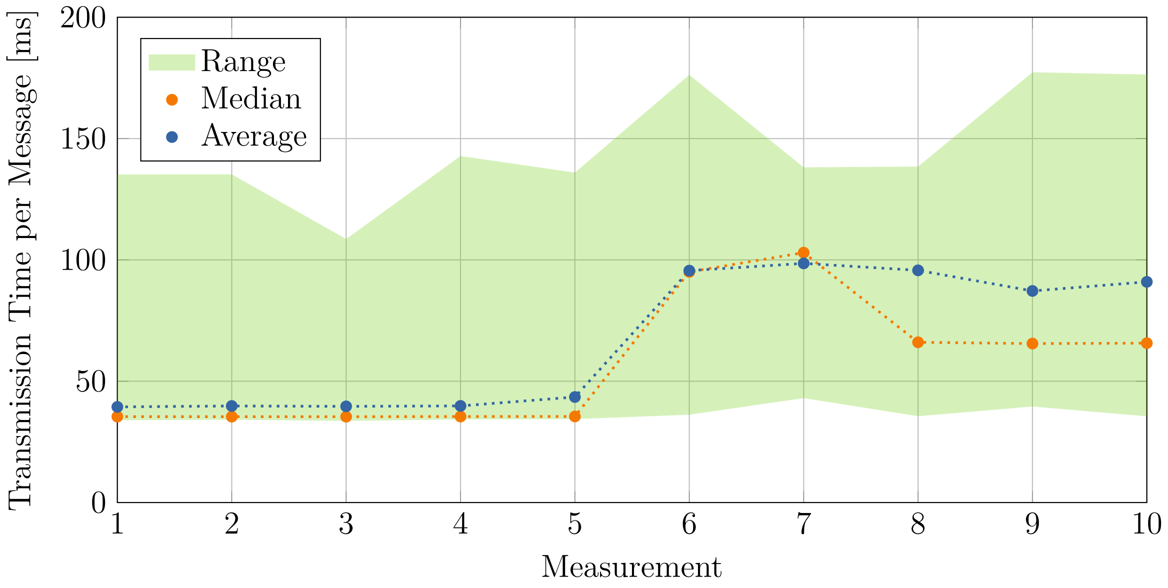


Fig. 2.16: Repeated Sequential Time Measurement of eProsima's DDS Router with 1 kB

It is likely that the phenomena described above are a result of the memory management, buffer sizes, and Nagle's algorithm of the DDS Router. However, this hypothesis could not be verified due to the lack of successful reconfiguration of the DDS Router respectively FastDDS in regard of the DDS participants. Nevertheless, it is evident that these phenomena do not result from measurement errors or other side effects, as other ROS2 over WAN solutions evaluated in this thesis do not manifest this behavior under equivalent measurement conditions.

Finally, parallel time measurement was conducted, the results of which are presented in Figure 2.17, whereby time measurements were carried out for messages with a total size ranging from 12 B to 10 kB with a degree of parallelization from 1 to 10. The plot illustrated the degree of parallelization on the X-axis, defined as the number of messages transmitted in parallel, and the total transmission time of these messages on the Y-axis. The average transmission times of the different messages sizes are depicted in relation to the degree of parallelization.
It can be seen that the transmission time of two parallel messages is about 1.75 as long as of a single message. A further sharp increase in transmission time is observed with three parallel messages. In instances involving four or more messages, an additional phenomenon emerged: for messages of less or equal than 1 kB, a subsequent yet less pronounced increase in the average transmission time was observed, which subsequently stabilized. In contrast, for messages exceeding 1 kB, the average transmission time exhibited a substantial decrease, approaching the transmission time of a single message.
As before, this phenomenon can be replicated and is not attributable to measurement errors or fluctuations in network quality.
As has been previously observed, the problem arose again that the DDS Router

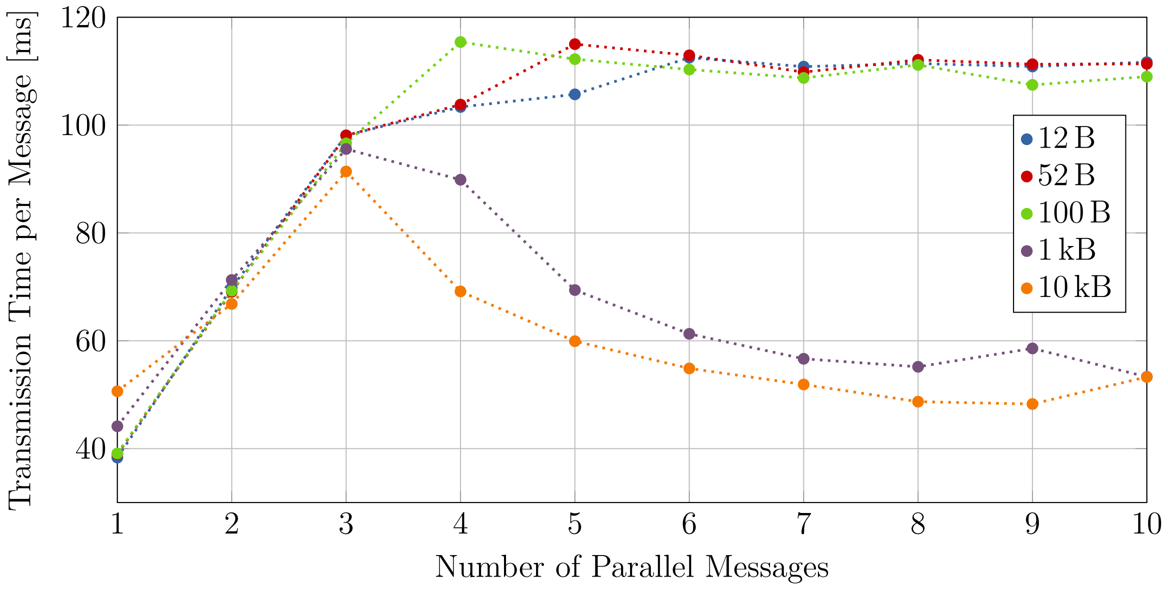


Fig. 2.17: Parallel Time Measurement of eProsima's DDS Router

instance on host B becomes unresponsive, could no longer transmit messages, and must be terminated by a SIGKILL. In this instance, however, the issue manifested with messages exceeding 20 kB in size and exhibiting a degree of parallelism of 2.

2.3.2.3 Suitability

Finally, the suitability of eProsima's DDS Router for ROS2 over WAN in the context of Telelab will be assessed. The fact that the DDS Router is an almost ready-made solution that only requires to be expanded to include authentication mechanisms speaks in favor for its use. However, there exist several arguments against its use. On the one hand, DDS Router exhibits a fixed dependency on a third party, a circumstance that has the potential to result in complications over the course of several years of operation. On the other hand, it cannot be completely ruled out that operation with DDS implementations other than FastDDS is possible on a permanent basis and whether this makes sense at all with regard to possible incompatibilities due to the elimination of vendor-specific functionalities. Moreover, the time measurements have revealed several phenomena. In particular, the slowing down of the average transmission time during longer operation (Fig. 2.16) should be emphasized here, which, however, would be the normal use case. Additionally, the issues associated with the transmission of larger messages, wherein the DDS Router ceases to function and necessitates termination by SIGKILL, would compromises e.g. the transmission of camera images or parallel transmissions in general. Parallel transmission also comes with longer average transmission time since DDS Router does not seem to implement dedicated mechanisms for handling several topics in parallel; in Telelab, however, several parallel topics will have to be transmitted.

In summary, the DDS Router is not a viable solution for ROS2 over WAN in the context of Telelab. Its performance, reliability, and compatibility are insufficient for the intended use case.

2.3.3 Rosbridge

The Rosbridge respectively the ROS2 node *rosbridge_websocket* of the package *rosbridge_server* is part of the meta-package *rosbridge_suite*, which is developed by RobotWebTools [33]. This node provides a JavaScript Object Notation (JSON) Application Programming Interface (API) for ROS2 [33]. Its primary function is to facilitate access to ROS2 functionalities and interaction with ROS2 via JSON, for instance, for JavaScript integration, which enables web clients to display ROS2 data, such as laser scans, or to initiate service calls [33]. Rosbridge offers a variety of transport protocols to facilitate this process, including the utilization of WebSockets which are NAT friendly [33].
Despite the fact that Rosbridge was not developed as a ROS2 over WAN solution, it can still be used for this purpose. For this, a Rosbridge instance is initiated on the client and robot, respectively, utilizing WebSockets as a transport mechanism. Subsequently, a ROS2 node, which has been developed for this purpose and is called *rosbridge_websocket_bridge*, is initiated on the client. In this configuration, the developed *rosbridge_websocket_bridge* node establishes a WebSocket connection to the Rosbridge instance that is running on the client and to the instance running on the robot, see Fig. 2.18. *rosbridge_websocket_bridge* hereby functions as a bridge itself, facilitating the transmission of incoming messages and events from the robot to the client and vice versa. Consequently, the implementation of a ROS2 over WAN solution can be facilitated via the utilization of WebSockets, employing the JSON API of the Rosbridge. Given that WebSockets are utilized as the transport mechanism, which can be secured using TLS and are also NAT friendly, no NAT traversal mechanisms need to be implemented.
However, Rosbridge does not implement any AAA mechanisms apart from rudimentary accounting mechanisms. Therefore, it is necessary to implement all AAA mechanisms. One potential implementation would be another node similar to the *rosbridge_websocket_bridge* node. This new node would be executed on the robot and

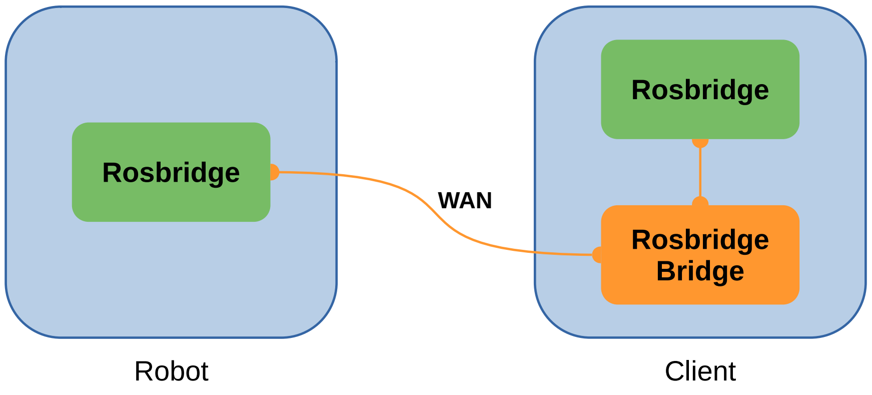


Fig. 2.18: Rosbridge Topology

establish a connection with the local Rosbridge instance, thereby facilitating the provision of a WebSocket server. Consequently, the *rosbridge_websocket_bridge* node on the client would not establish a direct connection the the Rosbridge instance on the robot, but rather to the newly created node. This node functions as a bridge too, facilitating the transmission of messages in both directions. Furthermore, this node is capable of implementing all AAA mechanisms.
It should also be noted that the Rosbridge ROS2 over WAN solution functions independently of the DDS implementation. This is due to the fact that Rosbridge, in contrast to the DDS Router, utilizes the messages at the ROS2 abstraction level instead of the DDS abstraction level, which is independent of the latter, see section 2.1.3.1. This approach even enables the utilization of two distinct DDS implementations on both the client and the robot.

2.3.3.1 Setup and Deployment

Rosbridge respectively the *rosbridge_server* package, is a pre-made software solution that has been built by the package maintainers of the ROS2 repositories for each ROS2 version. These builds are made available via the repositories, enabling Ubuntu users to install the *rosbridge_server* package through package management. The package built for ROS2 Jazzy Jalisco is designated as "ros-jazzy-rosbridge-server". For other Linux distributions, such as Debian or Fedora / RHEL, the *rosbridge_server* package and its dependencies must be built manually according to the documentation [33].
The *rosbridge_server* package is equipped with a launch file designed as `rosbridge_websocket_launch.xml`, which facilitates the execution of a local instance of the *rosbridge_websocket* node, utilizing default parameters [33].

Following Rosbridge has been installed and can be executed locally using the launch file, the self-developed node *rosbridge_websocket_bridge* must be built on the client side, which uses the Boost library (see 2.4.5 for why the Boost library was chosen) to establish the WebSocket connections to the Rosbridge instances. This node is part of a ROS2 colcon package, and consequently, can be built using colcon, the ROS2 meta-build-tool. The detailed implementation of the *rosbridge_websocket_bridge* will not be discussed here, as its development status is adequate for time measurements, but not for production use, and its precise implementation is not significant at this point. The complete code of the *rosbridge_websocket_bridge* can be found on the data medium accompanying this thesis, see Appendix C.

In the context of Telelab, it would therefore be sufficient to provide users with the *rosbridge_websocket_bridge* code and instructions on how to make it executable, as well as instructions on how to install the ROS2 package *rosbridge_server*. It can be assumed that the *rosbridge_websocket_bridge* would likely utilize configuration parameters in a production setting, which could be distributed to the user via a dedicated launch file, which would start all necessary nodes (*rosbridge_websocket* and *rosbridge_websocket_bridge*) thereby providing the needed configuration parameters.

2.3.3.2 Performance Evaluation

Performance evaluation is carried out through sequential and parallel time measurements using the procedure previously described in section 2.3.2.2. In the case of Rosbridge, a Rosbridge instance is initiated on both host A and B. On host B, the *rosbridge_websocket_bridge* node is also initiated. Host C functions as a reverse proxy (Apache HTTP Server) to enable a connection between host A and B from outside the university network. The traffic between host B and C is secured via TLS; the traffic between host A and C is not. Hence, the *rosbridge_websocket_bridge* establishes a connection with the locally running Rosbridge instance and the reverse proxy on host C, which functions as a router for traffic to and from the Rosbridge instance running on host A via an internal network.
The designation of the timestamps remains constant. However, given *rosbridge_websocket_bridge*'s presence as an additional node in the communication path, the recording of additional timestamps is now possible, as illustrated in Fig. B.15 of appendix B. Consequently, timestamps BB and II can now be recorded when a new message arrives in the *rosbridge_websocket_bridge*. Within the *rosbridge_websocket_bridge*, intra-processing times can also be recorded using the timestamps C, CC, and D, as well as J, JJ and K.

Figure 2.19 presents the results of the sequential time measurements, whereby time measurements were carried out for messages with a total size ranging from 12 B to 250 kB. As with the DDS Router, the graph displays the median and average transmissions times of a round trip, in addition to the range of the measured transmission times and the theoretical optimum based on iperf3 bandwidth measurements. The graph also shows, for comparison, the average transmission times previously determined for the DDS Router.

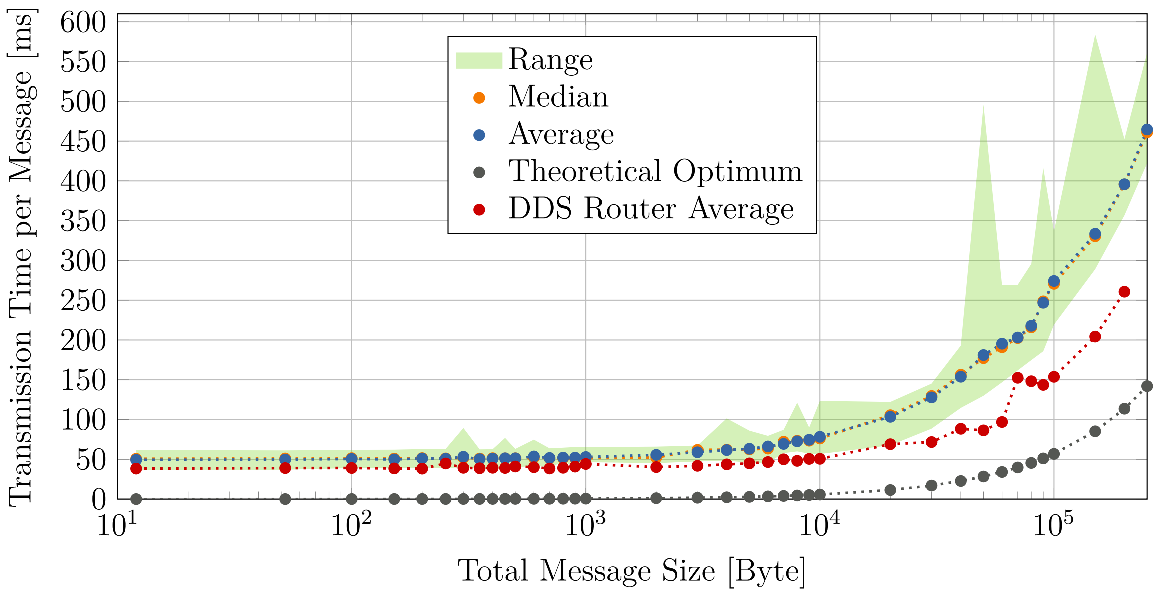


Fig. 2.19: Sequential Time Measurement of Rosbridge

It can be observed that the average transmission time remains relatively constant up to total message sizes of 1 kB with a difference to the theoretical optimum ranging from 49 ms to 53 ms. However, the transmission times are already marginally larger than that of the DDS Router. For messages with a total size larger than 1 kB, a steady increase of the transmission time can be observed reaching up to 323 ms in difference to the theoretical optimum for messages with a total size of 250 kB. These longer transmission time for larger messages can be attributed to transmission / serialization delay, as previously observed with DDS Router. However, an approximate twofold increase in transmission times was observed for Rosbridge, in contrast to DDS Router, which can be attributed to the utilization of the JSON API by Rosbridge. This is due to the fact that the ROS2 messages require conversion to and from JSON, which incurs additional time consumption during transmission. As with the DDS Router and in contrast to the new ROS2 over WAN solution presented in this thesis, ROS2 Connect, the observed increase in transmission time is not influenced by the default congestion control algorithm employed by the Linux kernel (see section 2.4.10).
Furthermore, it can be observed that, in contrast to the DDS router, the average and median transmission times of the Rosbridge are found to be close to the middle of the range of transmission times. In general, the range of transmission times is also significantly smaller than was measured in the case of the DDS Router. The mean CV over all measured message sizes was at 7.59% (median at 7.08%), with a minimum of 3.51% at 200 kB and a maximum of 21.63% at 50 kB. The average and median CV indicate low variability, while the maximum CV indicates moderate variability. This observation signifies that, in contrast to DDS Router, Rosbridge

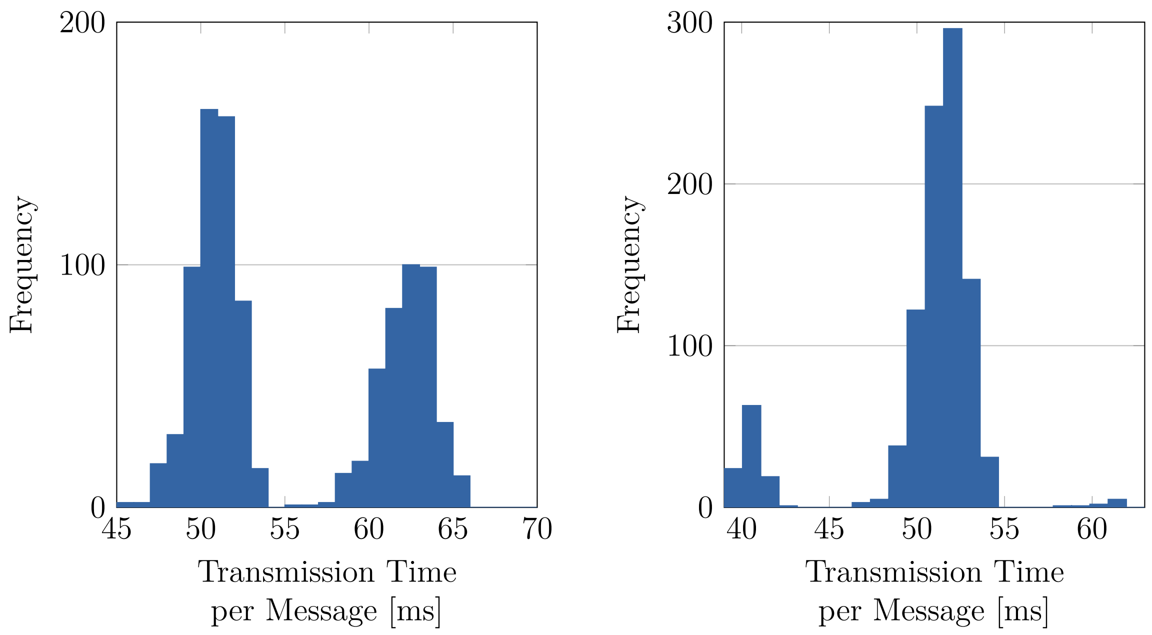


Fig. 2.20: Left: Distribution of time measurement values for 2 kB total message size, Right: Distribution of time measurement values for 100 B total message size

produces significantly more stable round-trip times whose variability is typical or even better for a WAN connection, according to Lee et al. [32].
As illustrated in Fig. 2.20, a histogram with 25 bins is presented on the left, which displays the distribution of the 1000 measured values of the time measurement for a message with a total size of 2 kB. The measured values are observed to accumulate to the left and right of the mean between the minimum and maximum transmission time, indicating a bimodal distribution in the case of 2 kB, with a CV of the measured values being 10.53%. On the right in Fig. 2.20 is another histogram, this time with 23 bins, showing the distribution of the 1000 measured values for a message with a total size of 100 B. The majority of the measured values are concentrated around the mean value between the minimum and maximum transmission time, however, with more values accumulated to the right of the mean. Furthermore, a secondary yet comparatively minor concentration is observed in the region of the minimum. This indicates a slightly right skewed distribution in the context of 100 B, with a CV of the measured values being 7.39%.
This phenomenon can be explained by examining the individual time intervals illustrated in Fig. 2.21. An initial examination will be conducted of the two intervals [A, BB] and [K, NN]. While both intervals are measured on host B, they exhibit significant differences. [A, BB] is found to be relatively smooth, while [K, NN] is observed to be highly scattered and, on average, longer than [A, BB]. This can be attributed to two primary factors: Firstly, [A, BB] is a transfer from a WebSocket server to a WebSocket client, while [K, NN] is a transfer from a client to a server. As previously outlined in section 2.1.4, transfers from clients to servers must be masked, which requires additional computing time. Furthermore, [A, BB] involves a conversion from a ROS2 to JSON message representation, while [K, NN] involves a conversion

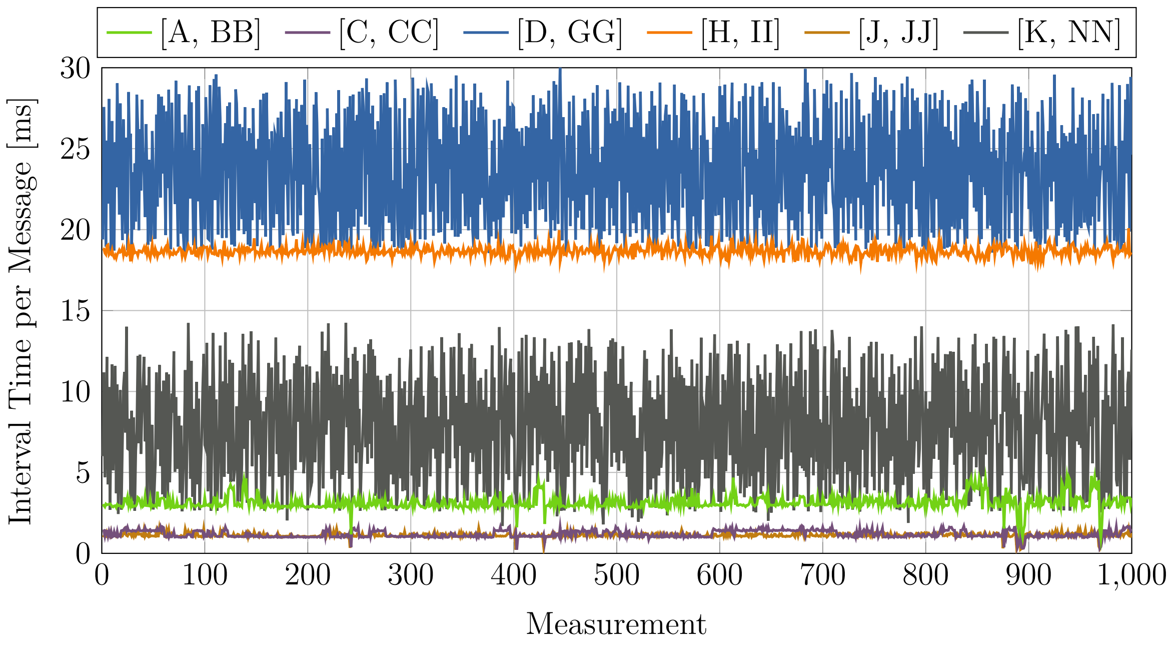


Fig. 2.21: Time Intervals of sequential time measurement for 2 kB total message size

from a JSON to ROS2 representation. This difference could also have implications for the time intervals, in addition to the effects of WebSocket frame masking. A similar conclusion can be derived by examining the intervals [D, GG] and [H, II]. In this case as well, the transmission from the WebSocket client to the server ([D, GG]), which encompasses the conversion from JSON to ROS2, exhibits greater scattering and is longer on average. In the case of [D, GG] and [H, II], however, an additional factor contributing to the observed differences is an asymmetry in the bandwidth between host A and B (uplink 18.5 MBit/s, downlink 58.6 MBit/s).
In summary, this indicates the presence of two distinct processes during transmission, one characterized by a slower rate and the other by a faster rate. Notably, both processes demonstrate increased scattering in their proportional transmission time. Depending on how this scattering is distributed, the measured values may exhibit a slight right-skewed, left-skewed, or more modal distribution. The scattering is the result of additional computationally intensive tasks that depend on operating system (OS) resources and are therefore subject to OS scheduling, which leads to the observed variations in processing time. However, this scattering is in general significantly less pronounced in comparison to the DDS Router.

Finally, parallel time measurement was conducted, the results of which are presented in Figure 2.22, whereby time measurements were carried out for messages with a total size ranging from 12 B to 100 kB with a degree of parallelization from 1 to 10. The outcome is in clear contrast to the results observed in the DDS Router. The results for the message sizes tested demonstrate that for messages up to 10 kB in size, the transmission time remains constant despite the simultaneous transmission of up to 10 messages. This outcome is consistent with the Rosbridges's utilization of a MultiThreadedExecutor [33] which allows the handling of several topics in parallel (see

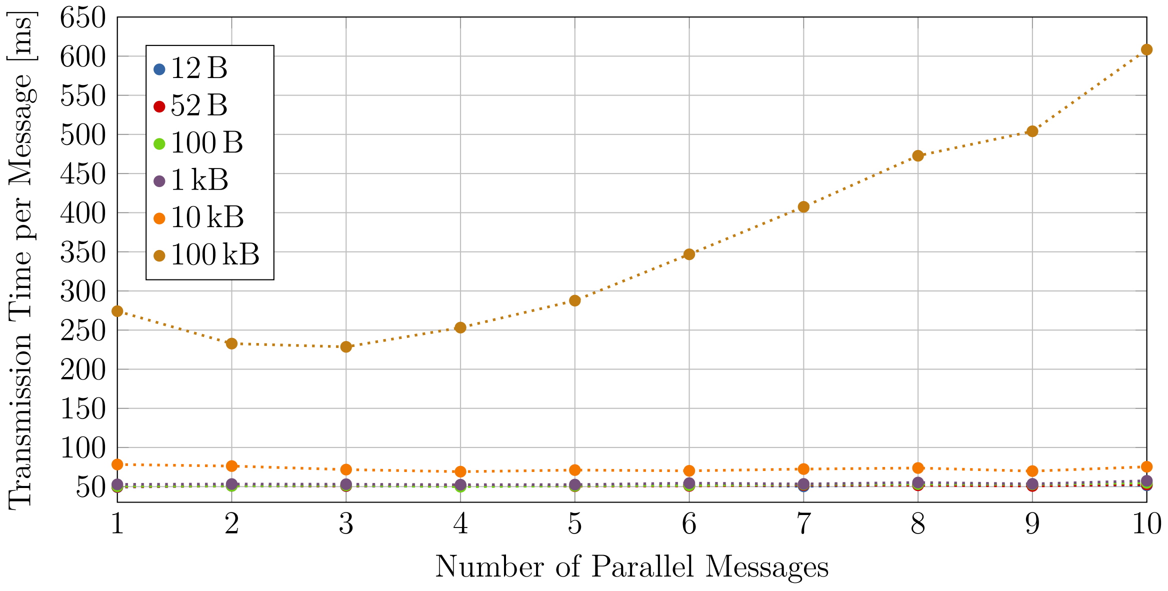


Fig. 2.22: Parallel Time Measurement of Rosbridge

section 2.1.3.2). For messages with a total size of 100 kB, there is a steady increase in the average transmission time per message with a parallelism of four and more messages. This can be attributed to the nature of the WebSocket connection, which is designed to transfer a single message at a time. However, in scenarios involving concurrent transmission of multiple messages, the network experiences congestion, leading to an increase in the average transmission time. Nevertheless, this phenomenon can be mitigated by employing multiple instances of Rorsbridge in parallel, thereby facilitating the establishment of numerous discrete WebSocket connections. To illustrate, the transmission of larger image data could occur through a dedicated connection, while smaller sensor data could be transmitted via a secondary connection, which would prevent starvation during the transmission of the smaller sensor data.

2.3.3.3 Suitability

Finally, the suitability of Rosbridge for ROS2 over WAN in the context of Telelab will be assessed. The performance evaluation demonstrated that Rosbridge shows significantly more stable behavior compared to the DDS Router. Rosbridge functioned without exhibiting issues with large messages, slowdowns in transmission time during prolonged operation, or massive spikes in transmission time. The time measurements were stopped at a total message size of 250 kB, as the transmission time for this particular message size had already reached a substantial duration of around 500 ms on average. However, it was verified that the transmission of messages was possible for messages with a total size up to 2 MB. Rosbridge has been designed from the ground up to process multiple topics simultaneously. The utilization of WebSockets as the transmission mechanism also ensures seamless connectivity, eliminating potential issues associated with NAT routers. The programming effort necessary to establish a connection between the Rosbridge instances on one hand and to implement AAA mechanisms on the other is limited.
Conversely, Rosbridge exhibits a substantially slower average transmission time in comparison to DDS Router. This difference can be attributed to Rosbridge's utilization of JSON, wherein all ROS2 messages are required to be converted to and from JSON, which necessitates an additional computational expense.
Another issue with Rosbridge that has not yet been addressed relates to its behavior when a subscriber attempts to access a topic that has not yet been published but is subsequently published at a later point in time. In such a scenario, Rosbridge is unable to transmit ROS2 messages for this topic. Therefore, it is necessary to first delete the existing subscription and then create a new one. This behavior stands in contrast to the one on which ROS2 manages subscriptions. In the context of native ROS2, a subscription to a topic that has not yet been published is provisioned with messages as soon as a corresponding publisher becomes available and publishes data.
Another notable issue with using Rosbridge for ROS2 over WAN is the absence of a method by Rosbridge itself to monitor the number of local subscribers of the local publishers of the Rosbridge instance. In the context of native DDS / RTPS as used by ROS2, the transmission of RTPS messages from a publisher is suspended in the absence of a subscriber seeking to receive them [4]. However, given that Rosbridge

does not monitor local publishers with regard to the number of subscribers and only offers a rudimentary system for subscribing to a topic, the data of all available topics would be continuously transmitted, even in the absence of a “recipient” for this data. Consequently, this results in the WebSocket connection being in a constant state of activity, which is particularly problematic in scenarios involving large messages, such as camera images.

In summary, although Rosbridge performs better than the DDS Router in most aspects, it is not an optimal solution for ROS2 over WAN in the context of Telelab. The average transmission time is notably prolonged, failing to meet the requirements for the intended use case. Furthermore, the data of all available topics is transmitted continuously, even in the absence of a designated recipient. Nevertheless, the fundamental concept underlying Rosbridge provides a solid foundation for the implementation of an own solution.

2.4 ROS2 Connect
A new solution to ROS2 over WAN

Following the conclusions of the preceding sections, which demonstrated the absence of a satisfactory solution to the problem of ROS2 over WAN, a new solution was developed as part of this thesis: the ROS2 Connect package. It implements mechanisms to cover the entire spectrum of ROS2, including subscribers, publishers and QoS, tf2 transformations, service and action calls, as well as native ROS2 discovery. The Connect package is inherently optimized for processing parallel topics, supports per-topic compression, and implements the AAA mechanisms. Notably, the authentication mechanisms is integrated directly into the existing Telelab infrastructure.

The Connect package provides two nodes, *server* and *client*, which, like Rosbridge, establish a WebSocket connection to transmit all ROS2 traffic. However, unlike Rosbridge, the Connect nodes transmits raw serialized binary data, which is significantly more efficient than the JSON API employed by Rosbridge.
The subsequent section describes the fundamental components and design decisions of the Connect nodes by methodically examining the code base, beginning with the launch and configuration, progressing to the WebSocket implementation, connection establishment, authentication mechanisms, message representations and concluding with the transmission of ROS2 messages and the execution of service and action calls. Finally, as before in this thesis, the performance of the Connect nodes is evaluated by measuring the transmission times.

The data medium accompanying this thesis contains a class diagram that provides an overview of all implemented classes and their relations to each other (see appendix C). Due to the considerable dimensions of this diagram, its inclusion in this thesis has been considered unfeasible.

2.4.1 Setup and Deployment

The Connect package is managed as a ROS2 colcon project and implemented in the C++ programming language, using the C++23 standard. It has dependencies on various ROS2 components, such as rclcpp, rclcpp_action, and pluginlib. Additionally, it incorporates the Boost library with a minimum version of 1.75, along with OpenSSL, lz4, and z-lib. Subsequent to the installation of these dependencies on the host system, the Connect package can be built using the ROS2 meta-build-tool colcon. The result of this build is two nodes / binaries, *server* and *client*. These two nodes are associated with the two WebSocket roles, server and client. Consequently, the *server* node is deployed on the robot, while the *client* node is deployed on the client which wants to connect to the robot.

However, within the context of Telelab, the *server* node is ultimately deployed in a slightly different way. It is not executed directly on the robot; rather, it is executed on an additional host system / server located within the university's internal network.

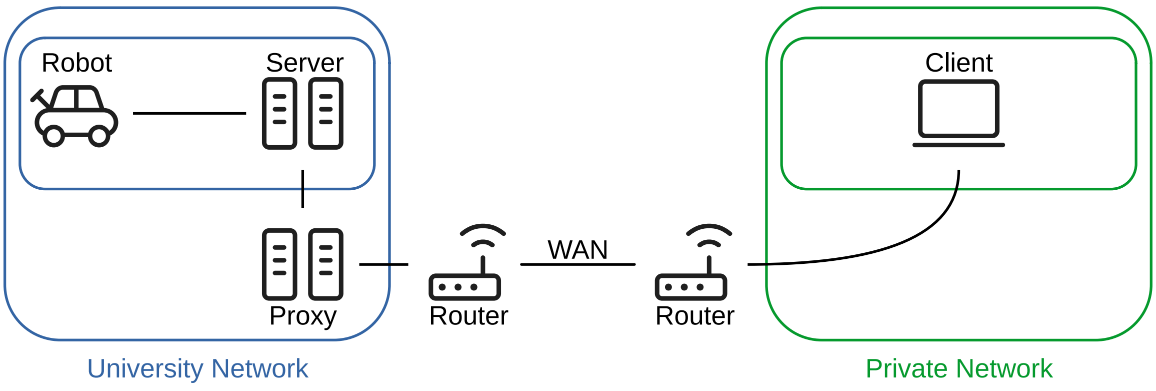


Fig. 2.23: Illustrated network topology of Telelab (Icons © Google (Apache 2.0) [26])

The underlying reason for this approach is, on the one hand, the limited computing capacities of the robot, and on the other hand, that additional data needs to be collected and transmitted, such as camera images from the overhead camera, which can be used to observe the robot during experiments. However, this has no impact on the development of the *server* node or the Connect package in general, as the additional host and the robot are on a shared internal network, therefore the additional host can receive all ROS2 messages from the robot. Also, as with Rosbridge, a proxy server is implemented at the edge of the university network. The client thus establishes a connection with the proxy server, secured by TLS, which then relays the client's traffic to the additional server via an internal network connection. The topology is illustrated schematically in Fig. 2.23.

In the context of Telelab, it would therefore be sufficient to provide users with the code of the Connect package and instructions on how to install the needed dependencies and on how to build the package using colcon. Furthermore, each user has the ability to download a customized parameter file from the Telelab portal. This parameter file contains all the configuration parameters necessary to launch the Connect *client* node, as well as a unique and user-specific `user_key` that is employed for authentication purposes.
In the future, establishing a Personal Package Archive (PPA) for Ubuntu, which would offer pre-built binaries, is considered. This would enable Ubuntu users to install the Connect package and all of its needed dependencies via the package manager. However, the parameter file containing the user-specific `user_key` would still need to be downloaded from the Telelab portal.

2.4.2 Conceptual Overview

As previously described, ROS2 Connect compiles to two nodes: *server*, which is deployed on the robot or the additional host, respectively, and *client*, which is deployed on a client's host that wants to teleoperate the robot. To this end, a bidirectional WebSocket connection is established between the two nodes *server* and *client*, which functions as a ROS2 communication bridge, see the following section 2.4.5. Further-

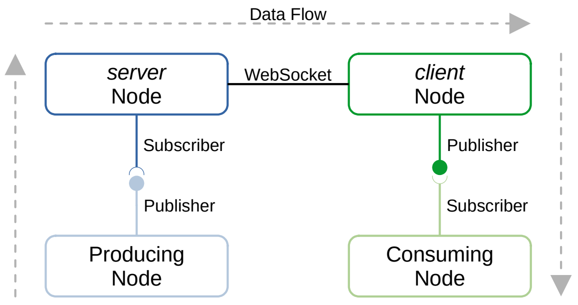


Fig. 2.24: Illustration of data flow of topic data

more, the two nodes implement ROS2 subscribers and publishers to receive data from other ROS2 nodes and re-publish it, see the following section 2.4.6.
Figure 2.24 provides an illustrative example:
A producing node is running on the robot, which provides status data on e.g. the topic /robot_status, and which the client wants to receive by the means of a consuming node. In order to execute this process, it is necessary for the *server* to be configured at the time of startup to have a subscription to the topic /robot_status. Additionally, the *client* must be configured to have a ROS2 publisher for this topic. When the consuming node subscribes to the topic /robot_status, this is registered by the *client* node and a control message (opcode) is sent to the *server* node. Subsequently, the *server* node creates a ROS2 subscriber to the topic /robot_status, receives its data from the producing node, and sends it to the *client* node via the WebSocket connection. Thereafter, the *client* node re-publishes the received data on its publisher, thereby enabling the consuming node to receive this data.
The *client* and *server* nodes are therefore configurable through their parameters in the topics they relay; there is no need to change the code of ROS2 Connect if a new topic or topic type is to be forwarded.
In addition to the implementation of ROS2 subscribers and publishers, ROS2 Connect also implements ROS2 action and service server and clients, see the following section 2.4.9. This enables the client to call a dedicated action or service, of which the actual action or service server is executed on the robot, as illustrated in Figure 2.25. In this scenario, the *server* node must implement an action or service client, and the *client* node an action or service server, for a dedicated action or service. This enables an action or service client implemented by the consuming node, which is executed on the

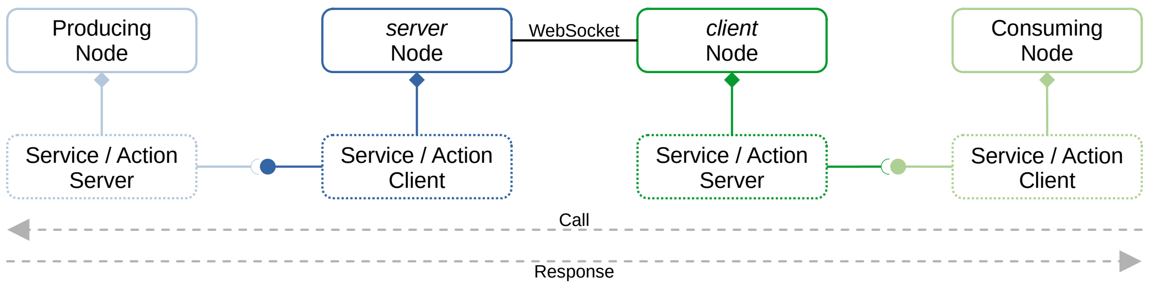


Fig. 2.25: Illustration of data flow of action / service call

client, to call the action or service server of the *client* node. This request (or goal) is subsequently transmitted to the *server* node through the WebSocket connection. The action or service client implemented by the *server* node then calls the dedicated action or service server implemented by the producing node, receives the response, and transmits it back to *client* node. Therefore, the client's action or service call is answered by the actual action or service server running on the robot.

The subsequent sections will elaborate on these and other implementation details of ROS2 connect.

2.4.3 Code Review Launch and Configuration

While the two nodes *server* and *client* can be executed directly without a launch file, launch files are employed for the execution of the two nodes, which are documented in appendices A.3.1 and A.3.3. These launch files are utilized to load the extensive parameter files which serve to initially configure the two nodes, and are documented in appendices A.3.2 and A.3.4.
In addition to the connection parameters, the `user_key`, and its verification, the parameter files define, above all, the QoS profiles, compression profiles, and most importantly, the publishers and subscribers, as well as the service and action servers and clients which are utilized by the *server* or *client*. This specification of subscribers and publishers determines the topics that the *client* or *server* instance transmits to the other party and the topics that the other party can transmit to it. This implements the authorization mechanism (AAA) since the *server* exclusively accepts data from topics for which it has defined a publisher, or only forwards data from subscribers defined by it to the *client*. This principle similarly applies to actions and services. In this context, the *client* is only permitted to execute actions and services for which the *server* has defined a corresponding service or action client.

The YAML Ain't Markup Language (YAML) format was selected for the structure of the parameter files. Despite the fact that this format supports complex arrays, i.e., it offers the possibility to define arrays in which each entry is a complex type consisting of several values, the ROS2 YAML parser used in the ROS2 launch process does not currently support the parsing of complex arrays [12]. Consequently, the definition of QoS and compression profiles, subscribers and publishers, as well as service and action clients and servers is achieved through the utilization of arrays of JSON strings.

The parsing and validation of all these parameters and JSON arrays is conducted within the abstract `ConnectBase` class, which inherits from `rclcpp::Node`, of the *server* and *client* node, ensuring the integrity and consistency of the parameters. This class also computes the number of threads necessary for the MultiThreadedExecutor, which can be calculated as follows:

$$\begin{aligned} &n = 4 + a + x + y \\ &x = \begin{cases} 1 & \text{, if parameter “tf/publish” or “tf/subscribe” is true} \\ 0 & \text{, else} \end{cases} \\ &y = \begin{cases} 1 & \text{, if parameter “clock/publish” is true} \\ 0 & \text{, else} \end{cases} \end{aligned} \tag{2.2}$$

where:

n	number of threads assigned to the MultiThreadedExecutor
a	number of subscriber, publisher, service server, service client, action server, and action client which have defined “useOwnThread” as true

Therefore, the number of threads is a minimum of four: one thread for the mutually exclusive callback groups of shared subscribers, services, and actions, and one additional thread for the default callback group used by the publishers.

The abstract class `ConnectBase` is inherited by the two classes `ConnectServer` and `ConnectClient`, which represent the entry points for the *server* and *client* nodes, respectively, see Fig. B.2 of appendix B. These classes perform the initialization of the ROS2 context, the parsing and validation of the parameters that are made available globally via a read-only (by contract) class, `GlobalConfig`, and the instantiation of the actual WebSocket implementation.

2.4.4 Code Review Message Representations and Compression

As previously described, the *client* and *server* nodes facilitate the transmission of data between each other in the form of raw serialized binary data. To achieve this objective, a uniform message representation was developed, encompassing both the transport process and the handling of messages within the nodes, including payload compression. This message representation is defined by the abstract class `MessageBase`, for which there are three implementations: `VectorMessage`, `RclMessage`, and `ServiceActionMesssage`, see Fig. B.3 of appendix B.
This message representation defined by `MessageBase` is structured as follows:

channel	compressor	uncompressed size (*only if compressed*)	payload
1 Byte	1 Byte	4 Byte	n Byte

Consequently, each message is composed of two components: a header and a payload. The header always contains the fields “channel” and “compressor”. “channel” encodes a topic in the value range of 0 to 249 (see section 2.4.6), in addition to opcodes in the value range of 250 to 255, wherein the following opcodes are defined:

Value	Definition
250	Indicates a new subscription, see section 2.4.6
251	Message is a `ServiceActionMessage`, see below and section 2.4.9
252	Data of /`tf`, see section 2.4.7
253	Data of /`tf_static`, see section 2.4.7
254	Data of /`clock` topic, see section 2.4.8
255	Authentication: Confirmation or `user_key`, see section 2.4.5

The “compressor” field serves to indicate whether the payload is compressed, and if so, which compression algorithm was employed to compress the payload. A total of three distinct compression algorithms are available:

Value	Compression Algorithm
0	**None**
1	lz4 default
2	lz4 hc
3	z-lib deflate

If compression has been applied, the “compressor” field is followed by four additional bytes containing the size of the uncompressed payload, which is a necessity for decompression. The reason behind the selection of three distinct implementations for compression can be attributed to their different characteristics, which is why a performance evaluation was carried out to assess them. To this end, three distinct inputs were compressed and decompressed under varying settings of the algorithms. The inputs consisted of a byte array containing solely the value “0”, representing a highly compressible input, a byte array containing randomized values, signifying a challenging input to compress, and a camera image from the Telelab’s overhead camera (see Fig. B.1 of appendix B). The settings that were tested for the algorithms included maximum compression, minimum compression, the default values, and the average between maximum and minimum compression. The results presented in Fig. 2.26, which show the compression rate in reference to the total compression and decompression time, can be interpreted as follows:

- The z-lib deflate algorithm consistently demonstrated best compression with the highest compression rate. In the case of the lowest settings, the time required is greater than that of lz4 hc. However, for middle or higher settings, it is more efficient than lz4 hc, while producing superior compression rates.
- The lz4 default algorithm demonstrated the optimal compression time across all algorithms and settings. The duration of the compression and decompression process is consistently brief, lasting only a few milliseconds. Nevertheless, it yields inferior compression rates than z-lib deflate.
- The lz4 hc algorithm demonstrated merely equivalent compression rates to those of z-lib deflate. However, it is only faster than deflate in the case of lowest settings.

The results of this evaluation demonstrate that lz4 default compression can be utilized without significant time consumption while achieving acceptable compression rates.

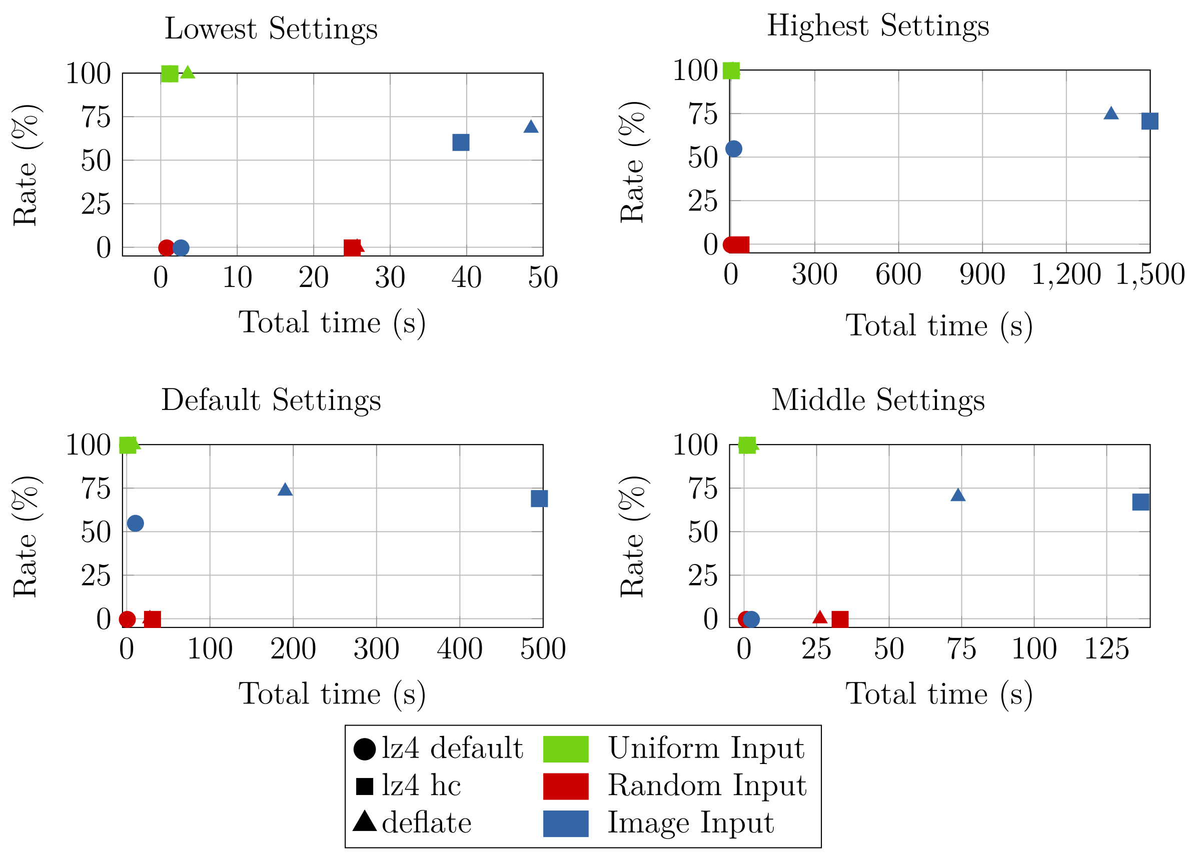


Fig. 2.26: Compression and decompression performance evaluation, executed on Intel i7-8565U

In instances of uncompressable data, lz4 default compression will halt early. The utilization of lz4 hc compression is only recommended for lowest settings. Conversely, z-lib deflate can be employed in scenarios where high compression rates are required, however, this comes with a greater time consumption.

The two classes `VectorMessage` and `RclMessage` implement this message representation and compression. They are employed for data from topics, as well as for the opcodes described above. Nevertheless, they differ in their respective implementation for storing the binary data. While `VectorMessage` employs a `std::vector<uint8_t>` to store its data, `RclMessage` utilizes an `rclcpp::SerializedMessage` as its underlying data structure, which originates from ROS2. The reason behind the development of these two distinct implementations is to minimize the necessity of data replication. In nearly all instances, data originating from a subscriber is transferred via the WebSocket connection to the other party. A `std::vector<uint8_t>` is more suitable for this purpose, which is why a `VectorMessage` is always used for data originating from subscribers. For data originating from the WebSocket connection, the data is published via an `rclcpp::GenericPublisher` (see section 2.4.6) in nearly all cases. It is important to note than an `rclcpp::SerializedMessage` is required for this process; therefore, an `RclMessage` is employed for data originating from the WebSocket connection. This approach is intended to minimize the number of necessary copies along the processing and communication pipeline.

Conversely, the `ServiceActionMessage` is employed exclusively for communication between the service and action server and client, which implies that the "channel" for the message is invariably 251 (see above). It also utilizes a `std::vector<uint8_t>` as the data structure for the purpose of storing the serialized binary data. However, in contrast to the message representation defined by the `MessageBase`, the `ServiceActionMessage` extends the definition with an additional header:

header	service action channel	service action opcode	goal id Size	goal id	payload
2-6 Byte	1 Byte	1 Byte	1 Byte	m Byte	n Byte

The fields "service action channel" and "service cction opcode" define the channel of the service or action within the value range of 0 to 255 and the type of the message, as multiple message types are possible for a service or action, respectively, see section 2.4.9. The "goal id size" field specifies the number of bytes allocated for the "goal id", which facilitates the concurrent execution of a service or action.

As with `VectorMessage` and `RclMessage`, the payload is the only part that undergoes compression. The header information is necessary for routing of messages within the *client* and *server* nodes.

2.4.5 Code Review WebSocket Implementation

The WebSocket implementation is the most essential component of ROS2 Connect, as it implements and provides the transport mechanism which enables ROS2 over WAN. The selection of WebSockets as the transport mechanism is driven by various factors. WebSockets are inherently NAT-friendly, as the establishment of the connection is initiated by the client and subsequently upgraded to a bidirectional connection, which circumvents the necessity of integrating explicit NAT traversal mechanisms [19]. Additionally, WebSockets demonstrate superior performance in comparison to pure TCP connections [34], and have already enabled successful teleoperation in the context of Telelab [35, 36]. Consequently, WebSockets were implemented as the transport mechanism. However, in contrast to Rosbridge, they transmit raw serialized binary data, which is considerably more efficient in the context of ROS2. This is due to the fact that `rclcpp::GenericSubscriber` can deliver serialized binary data directly, while `rclcpp::GenericPublisher` can republish this data directly (see section 2.4.6) [15]. This WebSocket implementation is based on Boost.Beast, a C++ header-only library that is part of the Boost library, build on Boost.Asio, and is primarily developed by Vinnie Falco [37]. However, during the development process of ROS2 Connect, two other WebSocket libraries were considered: WebSocket++, also known as libwebsock-

Tbl. 2.1: Properties of WebSocket libraries

Property	**Boost.Beast**	**WebSocket++**	**Simple-WebSocket-Server**
Development status	Under active development	Latest release in 2020	Under active development
Complexity of integration	Very complex to integrate; poor documentation, but examples and a strong community are available	Easy to integrate; poor and incomplete documentation; examples are available	Very easy to integrate; good documentation and examples; developer responds to tickets & questions
Possibility to configure	Very high; implements full WebSocket standard	Low to none	Low to none
Deployment	Part of Boost library; Linux package management	Linux package management	Needs to be compiled; deployed through ROS2 Connect

etpp and primarily developed by Peter Thorson [38], as well as Simple-WebSocket-Server (SWS) developed by Ole Christian Eidheim [39]. These two WebSocket libraries are also based on Boost.Asio [38, 39], nevertheless all three libraries differ in their properties, which are summarized in table 2.1. While Boost.Beast necessitates a dedicated implementation for establishing connections and transmitting data, it does provide the option for extensive configuration. Conversely, SWS demands minimal implementation effort while offering a very limited array of configuration options. WebSocket++ stands out primarily due to its lack of active development.
To further assess the performance of these libraries, an evaluation was conducted, in which the performance of the three WebSocket libraries was recorded. As previously, this was accomplished by employing the time measurements described in section 2.3.2.2. Consequently, binary data messages of varying size between 1 B and 1 MB were transferred from a WebSocket client to the server and back, while the round-trip time was measured. Figure 2.27 presents the outcomes of these time measurements for all three WebSocket libraries, showing the average round-trip times along with the additional theoretical optimum as determined by iperf3 bandwidth measurements. This plot reveals that there is hardly any difference between the performance of the three libraries for message sizes up to 10 kB. However, there is an observable increase in round-trip time for WebSocket++ above and equal 10 kB in comparison to Boost.Beast and SWS. For messages larger than 100 kB, there is also a noticeable increase for SWS compared to Boost.Beast. Nevertheless, this increase is more subtle than that observed for WebSocket++. As the message size continues

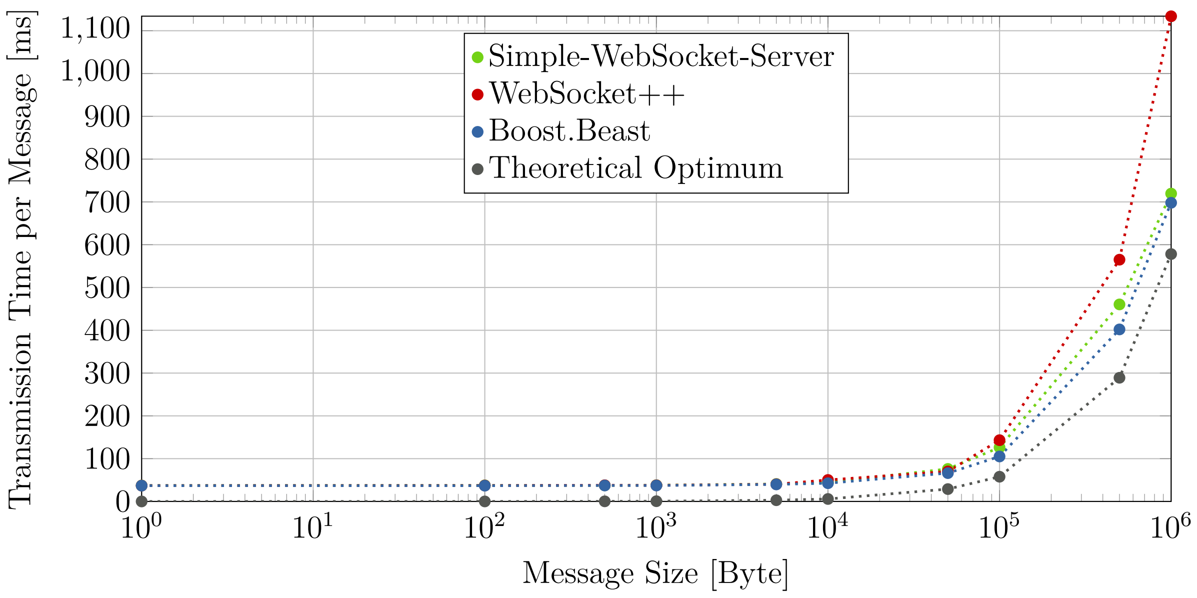


Fig. 2.27: Time Measurement of WebSocket libraries

to increase up to 1 MB, SWS performs slightly less efficiently than Boost.Beast. The average time difference for messages with a size of 1 MB is about 21 ms. However, WebSocket++ exhibits substantially poorer performance, demonstrating an average difference in round-trip time of around 436 ms. The average difference between Boost.Beast and the theoretical optimum remains constant at around 37 ms for message sizes up to 50 kB. For message sizes up to 1 MB, however, the difference increases to about 120 ms.
In summary, this demonstrated that WebSocket++ is the least suitable option due to its inactive development and poor performance. In the case of Boost.Beast and SWS, the decision was made in favor of Boost.Beast. This is attributable to its slightly superior performance and considerably more extensive configuration options. These options include the ability to implement ping / pong frames (see [19]) to maintain the WebSocket connection and detect unresponsive connection participants.

The WebSocket implementation of ROS2 Connect is distributed across multiple source files and classes. The connection is established by the `Client` and `Server` classes, which are instantiated by the `ConnectClient` and `ConnectServer` classes, respectively, see Fig. B.4 and B.5 of appendix B. The `Server` class implements a `boost::asio::ip::tcp::acceptor`, which accepts the initial HTTP connection initiated by the `Client` class. The `Client` class has been implemented with the capacity to utilize TLS / SSL for the purpose of establishing an outgoing connection. However, it should be noted that the `Server` class does not implement TLS / SSL, which is due to the fact that, in the context of Telelab, the *server* node always runs behind a reverse proxy server.
The handshake for upgrading the HTTP connection to a WebSocket connection is implemented by the two classes `WebsocketClientConnection` and `WebsocketServer`

`Connection`, both of which inherit from `WebsocketConnectionBase`, see Fig. B.6 of appendix B. Subsequent to the successful establishment of the handshake and the WebSocket connection, messages are not exchanged immediately; at this point, the `WebsocketServerConnection` discards the vast majority of incoming messages. The reason behind this behavior is that the WebSocket implementation also implements the authentication mechanism (AAA). As a result, subsequent to the establishment of the connection, the `WebsocketClientConnection` is required to transmit the `user_key` to the `WebsocketServerConnection`. In the event that this does not occur, the `WebsocketServerConnection` will automatically close the connection after a configured time period. The connection is also closed if the `user_key` cannot be associated with a user who currently has a valid reservation. Once the `WebsocketServerConnection` successfully authenticates the other party, it enables all communication. The `WebsocketConnectionBase` class implements asynchronous writing and reading on the WebSocket connection for this purpose. Consequently, both the `WebsocketClientConnection` and `WebsocketServerConnection` implement the same code for writing and reading. Incoming messages are routed to their intended destination, which may be, e.g., the PublisherManager (see section 2.4.6) or the ServiceActionManager (see section 2.4.9), based on their channel (see section 2.4.4). Outgoing messages are asynchronously sent from a FIFO queue.
Additionally, `WebsocketConnectionBase` implements ping / pong frames, which are standardized in the WebSocket protocol [19]. In this context, the *server* transmits a ping frame to the *client*, which responds with a pong frame. This protocol ensures the persistence of the connection by maintaining communication between both parties. Furthermore, it guarantees the vitality of both parties by automatically closing the connection if either a ping or pong frame is not received within a defined timeframe.

2.4.6 Code Review Subscriber and Publisher

The implementation of subscribers and publishers is the second most essential component of ROS2 Connect, as subscribers and publishers are the ROS2 (DDS) concepts that enable data capture and subsequent republishing. Therefore, ROS2 Connect implements the following mechanism: first, it subscribes to topics to receive data from producing nodes; second, it transmits this data to the other party using the WebSocket implementation; and third, it uses publishers to republish the data (see Fig. 2.24). As previously outlined in section 2.4.3, publishers and subscribers are implemented in a generic and dynamic manner in ROS2 Connect. This objective was accomplished by employing `rclcpp::GenericSubscription` and `rclcpp::GenericPublisher`, which, as previously outlined in section 2.1.3.2, do not require the topic type to be known at compile time, but rather at runtime [15]. Furthermore, these generic subscribers and publishers utilize raw serialized binary data [15]. Consequently, an `rclcpp::Generic Subscription` directly delivers binary data that can be transmitted via the WebSocket connection without further modification and can be republished directly by the other party using an `rclcpp::GenericPublisher` [15].

In ROS2 Connect, these generic subscribers and publishers are encapsulated by the abstract classes `SubscriberBase` and `PublisherBase`, respectively. These two abstract classes delegate the subscription callback and the actual publishing of data to their implementations. In case of `SubscriberBase`, these are: `ThreadedSubscriber` and `SharedSubscriber`; in case of `PublisherBase`, these are: `ThreadedPublisher` and `SharedPublisher` (see Fig. B.7 and B.8 of appendix B). These two implementations differ in their threading model. While the `Threaded...` implementations utilize a discrete thread for the purpose of processing topic data, the `Shared...` implementations utilize a shared thread. The implementation of these two distinct threading models offers several advantages: On the one hand, `SubscriberBase` and `PublisherBase` have the ability to utilize a discrete thread in scenarios where high-frequency or large topic data needs to be processed, as to avoid the unnecessary obstruction of the shared thread and, consequently, the entire ROS2 Connect processing pipeline, for an extended period. Conversely, `SubscriberBase` and `PublisherBase` also have the ability to utilize a shared thread in scenarios where low-frequency or small topic data necessitates processing, as to circumvent an excessive increase in the total number of threads employed by ROS2 Connect. Additionally, the per-topic compression of data, as described in section 2.4.4, has been implemented by the `Threaded...` implementations. This enables the compression and decompression of topic data to be executed in isolation in a distinct thread, thereby avoiding the obstruction of the entire ROS2 Connect processing pipeline.

The dynamic management and instantiation of these generic subscribers and publishers is implemented in ROS2 Connect by the two classes `SubscriberManager` and `PublisherManager` (see Fig. B.9 and B.10 of appendix B), which are instantiated themselves by the WebSocket implementation (`WebsocketConnectionBase`). These two classes utilize the values of the ROS2 parameters “subscriber” and “publisher” (further elaboration on this below), to create the various instances of `SubscriberBase` and `PublisherBase` at runtime. In the context of the instantiation of `PublisherBase` instances by the `PublisherManager`, this process occurs immediately subsequent to the successful authentication of the other party. Consequently, all publishers defined by the ROS2 parameter “publisher” are instantiated directly and subsequently maintained through the entire runtime of ROS2 Connect. However, in the context of the instantiation of `SubscriberBase` instances by the `SubscriberManager`, a different scheme is implemented. During the runtime of connected *client* and *server* nodes, the `PublisherBase` instances monitor their encapsulated `rclcpp::GenericPublisher` instance for the number of local subscribers / consuming nodes. Upon the occurrence of an increase in this number from 0 to at least 1 or a decrease from at least 1 to 0, the other party is notified via the WebSocket connection that either there is no longer a consumer or that there is now a consumer requiring the data of a topic or, more precisely, a channel (see also message representation; section 2.4.4). This notification (opcode value 250) is then forwarded by the WebSocket implementation to the `SubscriberManager`, which subsequently performs one of the two actions: it either destroys its `SubscriberBase` instance or it instantiated a `SubscriberBase` instance for the corresponding topic / channel according to the subscribers defined by the ROS2 parameter “subscriber”. This scheme facilitates substantial optimization by ensuring that only data for which there are actual consumers needs to be collected and transmitted via the WebSocket connection. This scheme is founded on the DDS and RTPS specifications, which similarly specify that data should be transferred exclusively under the condition that there are consumers for the data in question [1, 4].

An important restriction concerning publishers, subscribers, and hence the topics exchanged between two host systems or domains via ROS2 Connect is that neither a *client* nor a *server* instance my simultaneously act as both a publisher and a subscriber for the same topic. Allowing both instances to fulfill both roles for a given topic would result in an endless loop of topic data exchanged between the *client* and *server*.
While it is possible to extend the implementation of ROS2 Connect to avoid this behavior, such an approach introduces additional overhead in the publishing and subscription process. In particular, each message received by a subscriber would need to be verified against its originating publisher to ensure that messages published through ROS2 Connect are not re-captured and retransmitted to the other party.
In the specific context of Telelab, there is no requirement for bidirectional transmission of topics. Consequently, this extended implementation of ROS2 Connect was not adopted, as the unidirectional approach enables reduced transmission latency.

The aforementioned ROS2 parameters “subscriber” and “publisher” are both string arrays, where each entry is a JSON string, thereby enabling the definition of arrays of which each member constitutes a complex type (see section 2.4.3). Each member, and thus each publisher and subscriber definition for a topic, contains the following attribute-value pairs:

- **channel**
  The attribute channel relates to the channel of the message representation previously described in section 2.4.4 and therefore is an `uint8_t` value in the range of 0 to 249. Consequently, this attribute serves the function of linking data between a subscriber and a publisher, concurrently linking a channel to a topic type. Therefore, for example, if the *server* has a subscriber for channel 3, the *client* has a publisher for channel 3, both of which are for a topic with the same type while the name can differ. This enables the *client* or *server* to internally forward incoming data from the WebSocket implementation to the publisher for the designated channel. At the same time, this implementation also ensures that a *client* can only transmit data to a *server* for which the *server* has defined a publisher and that data transmitted from a *client* to a publisher of the *server* must be conform to the expected type for the designated channel. This fulfills the authorization mechanism of the AAA mechanisms.
- **topic**
  The attribute topic holds the name of the topic for which a `rclcpp::GenericSubscription` or `rclcpp::GenericPublisher` is to be instantiated by a `SubscriberBase` or `PublisherBase` in order to capture data from a producing node or to

republish it. The name does not need to be identical across a channel; hence, ROS2 Connect allows for the renaming of a topic.

- **type**
  The attribute type holds the type of the topic for which a `rclcpp::GenericSubscription` or `rclcpp::GenericPublisher` is to be instantiated by a `SubscriberBase` or `PublisherBase`. The value of type is required to be identical across a channel, otherwise the data transmitted via the WebSocket implementation cannot be republished. In the case of a subscriber, the value of type must correspond to the topic type of the topic to be forwarded to the other party.
- **useOwnThread**
  The attribute useOwnThread specifies (as `bool` value) wether a subscriber or publisher should use an exclusive, dedicated thread or the shared thread. This attribute, therefore, correlates with the two implemented threading models, hence, the two implementations `Threaded...` and `Shared...` for the two classes `SubscriberBase` and `PublisherBase`.
- **qos**
  The attribute qos contains the ID (as `uint8_t` value) of a QoS profile that has been defined by the independent ROS2 parameter "qos", which is also a string array in which each entry is a JSON string, thus constituting a complex type. These so-defined QoS profiles specify at least the QoS policies history, depth, reliability, and durability, while deadline, livespan, liveliness, and liveliness_lease_duration are optional, with default values being used if they are not defined. Therefore, these QoS profiles correspond in their attributes and possible values to the C++ struct `rmw_qos_profile_t` defined by rclcpp [15]. The parameter files utilized for the performance evaluation of ROS2 Connect are documented in appendices A.3.2 and A.3.4 and contain defined QoS profiles for reference.
- **compression**
  The attribute compression contains the ID (as `uint8_t` value) of a compression profile that has been defined by the independent ROS2 parameter "compression", which is also a string array in which each entry is a JSON string, thus constituting a complex type. These so-defined compression profiles specify wether and, if so, which compression algorithm should be used and with which settings to compress topic data (see section 2.4.4). It is important to note that the compression profile is specified exclusively for subscribers, since, as described in section 2.4.4, a compressed message contains all the necessary information for decompression by a publisher. Additionally, for the purpose of compressing data from a given topic, it is necessary to set the attribute "useOwnThread" to true for both the subscriber and the publisher. The parameter files utilized for the performance evaluation of ROS2 Connect are documented in appendices A.3.2 and A.3.4 and contain defined compression profiles for reference.
- **permanent** and **eager**
  The attributes permanent and eager enable the override of the default behavior of subscribers and publishers, which operates under the principle that data is transferred and republished exclusively under the presence of a consuming node. The **permanent** attribute, applicable exclusively to subscribers, facilitates the

immediate creation of the corresponding `SubscriberBase` instance following successful authentication of the other party, ensuring its persistence throughout the entire runtime of ROS2 Connect. Consequently, the `SubscriberBase` instance is considered permanent. The **eager** attribute, applicable exclusively to publishers, facilitates the republishing of topic data by the corresponding `PublisherBase` instance even in the absence of a consuming node. Consequently, the `PublisherBase` instance exhibits eager behavior. This option to override the default behavior enables the transfer of topics that employ a QoS profile with the durability policy `RMW_QOS_POLICY_DURABILITY_TRANSIENT_LOCAL`. As previously outlined in the section 2.1.1.3, the durability policy specifies wether topic data is volatile or wether it is cached. This ensures that nodes that join the network at a subsequent point in time can still receive the most recently published data (transient local) [1]. In such instances, the data from these topics must be collected, transmitted, and republished by ROS2 Connect, even in the absence of a consuming node. An example of a topic that uses this durability policy is given by tf2, as described in the next section 2.4.7.

The parameter files utilized for the performance evaluation of ROS2 Connect are documented in appendices A.3.2 and A.3.4 and contain exemplary definitions for the two parameters “subscriber” and “publisher” for reference.

The mechanisms and implementations described in this section in conjunction with the ROS2 parameter “subscriber” and “publisher” enable the dynamic and generic nature of ROS2 Connect, insofar as no code changes are required for any topic to be relayed by ROS2 Connect.

2.4.7 Code Review tf2 Transformations

As previously described in section 2.6, the semantics of the ROS2 core library tf2 is characterized by the roles of broadcaster and listener for the storage and lookup of both static and dynamic coordinate frame transformations, as well as their associated relationships [17, 18]. However, tf2 employs the two topics /tf_static and /tf for this purpose [18]. Accordingly, the tf2 functionality can be achieved between two distributed host systems connected via ROS2 Connect by forwarding the messages of these two topics. This objective is realized by the implementation of predefined subscriber and publisher for the two topics by ROS2 Connect with the type tf2_msgs/msg/TFMessage, which can be enabled using the boolean parameters “tf/subscribe” for the subscriber and “tf/publish” for the publisher. These subscribers and publishers do not use compression of their payload. Furthermore, they utilize predefined QoS profiles, which are identical to those that tf2 uses by default for the two topics. This ensures that the forwarding of the two topics by ROS2 Connect does not lead to QoS conflicts for consuming nodes.
The /tf_static topic is published by tf2 with the QoS durability policy “transient local” and no periodic messages are published on this topic; only one is immediately published after subscription, as well as when a static transformation is removed or added

[18]. Accordingly, the subscriber and publisher in ROS2 Connect for the /tf_static topic must

- both use the QoS durability policy "transient local" and a depth of 1,
- the publisher must be "eager",
- and the subscriber must be "permanent".

In the case of the topic /tf, the default QoS durability policy "volatile" is employed [18]. However, in contrast to the default ROS2 QoS profile, the depth in this case is set at 100 instead of 10 [18]. Consequently, the subscribers and publishers in ROS2 Connect for the /tf topic must

- both use the default ROS2 QoS profile, however, with a depth of 100,
- the publisher must be "eager",
- and the subscriber must be "permanent".

Furthermore, it should be noted that the restriction mentioned in the previous section 2.4.6 also applies to the two topics of tf2, namely that neither the *server* nor the *client* node may both have a publisher and subscriber for one or both of the two topics of tf2, as this would result in an endless transmission loop. Consequently, either the parameter "tf/publish" or "tf/subscribe" may be enabled. In the context of Telelab, however, this does not constitute an actual limitation, as only the coordinate frame definitions and their relationships need to be transferred from the robot's nodes to the client. Therefore, the *server* sets "tf/subscribe" to true, while the *client* sets "tf/publish" to true.

2.4.8 Code Review
ROS2 Time Synchronization

Time synchronization is a critical component in the context of ROS2 Connect, as it enables the execution of ROS2 nodes on distributed systems that communicate with each other. This is particular relevant in scenarios where ROS2 messages are transmitted which hold a timestamp, such as tf2 transformations [18]. ROS2 nodes utilize ROSTime as a time reference, which is a ROS2 time abstraction that, if not configured explicitly, uses the operating system's system time as its time source [14]. Hence, two options are available for time synchronization of the two hosts:

1. As with time measurements employed for performance evaluation in this thesis, time synchronization can be achieved at the host operating system level using the Network Time Protocol (NTP). The time measurements demonstrated that synchronization can be attained by synchronizing the host systems within the university network with the NTP server within the university network, which is synchronized with the PTB's NTP server, and by synchronizing the client host directly with the PTB's NTP server. The maximum observed skew between the hosts in the university network and the client was less than 1 ms with a median skew of 0 ms. The advantage of this solution is that no further adjustments need to be made to ROS2 for time synchronization at the operating system level, as

ROSTime uses the system time directly. A notable disadvantage is the potential for misconfiguration, particularly for inexperienced Linux users. Furthermore, although this was not observed with the time measurements carried out as part of this thesis, there is a possibility that an increase in the skew between the hosts in the university network and the client may arise, as demonstrated by Novick et al. [27] and Mills [28].

2. The second option is to directly synchronize the ROSTime of the client with that of the robot, through the means of ROS2 Connect. This is possible because each ROS2 node can be configured using the "use_sim_time" parameter to use the `/clock` topic, on which the current time is periodically published, as the time source for ROSTime [14]. In order to achieve this objective, the *server* node implements the `ClockSubscriber` class (see Fig. B.11 of appendix B) which is instantiated by the `SubscriberManager` class and can be enabled using the boolean parameter "clock/subscribe", while the *client* node implements a statically predefined publisher for the `/clock` topic with the type `rosgraph_msgs/msg/Clock`, which can be enabled using the boolean parameter "clock/publish". This configuration enables the *server* node to transmit its current ROSTime to the *client* node at an interval of 10 ms through the WebSocket connection. Subsequently, the *client* node can make this time information available to other nodes via the designated `/clock` topic. This topic can then be utilized by these other nodes as a time source for ROSTime, thereby achieving time synchronization with the robot. As previously outlined in section 2.4.6, the `ClockSubscriber` of the *server* node and the dedicated statically predefined publisher for the `/clock` topic functions equally to any conventional subscriber and publisher within the context of ROS2 Connect, which means that data collection and transmission occur exclusively in the presence of a local subscriber for the published `/clock` topic. The advantage of this solution is its independence from NTP, utilizing the robot as a time source to circumvent the possible skew that can arise with NTP time synchronization. However, a notable disadvantage is that each ROS2 node must be explicitly started with the "use_sim_time" parameter on the client. In the event that this is forgotten, time synchronization will not take place.

In the context of Telelab, the second solution, i.e., synchronization of ROSTime via ROS2 Connect, is the preferred solution. The reason for this is that any NTP time skews can be bypassed, and users do not have to change the NTP configuration of their operating systems, which could lead to misconfiguration. Consequently, the parameters "clock/subscribe" for the *server* and "clock/publish" for the *client* are set to true in the deployed parameter file (see sections A.3.2 and A.3.4). In the tutorials later created in the context of Telelab for the tasks to be completed by the users, the use of the "use_sim_time" parameter for starting ROS2 nodes will be explicitly pointed out for each task.

2.4.9 Code Review Service and Action

The final set of fundamental ROS2 concepts concern services and actions, both of which are fully supported by ROS2 Connect. Consequently, ROS2 Connect is capable of forwarding service or action calls in their entirety, thereby enabling, for example, a *client* to initiate a service call on a remote system. In order to facilitate the execution of a service call on a remote system by a *client*, ROS2 Connect is required to populate the service server that is running on the remote system on the system of the *client*. This process is facilitated by the *server* instance on the remote host, which implements a corresponding service client, and the *client* instance, which implements a corresponding service server. When the service server of the *client* receives a call, it's goal is transmitted, along with its potential parameters, via ROS2 Connect to the service client of the *server*, which in turn repeats the call to the locally running, actual service server, waits for the results, and then transmits them back vie ROS2 Connect to the service server of the *client*, which then returns the results to the caller (see Fig. 2.25).

In contrast to the implementation of subscribers and publishers and thus the forwarding of topics, which is entirely generic, the implementation of services and actions breaks the generic nature of ROS2 Connect. This is due to the fact that, in contrast to topics for which generic implementation is possible using `rclcpp::GenericSubscription` and `rclcpp::GenericPublisher`, no generic implementation for service and action server and clients in the ROS2 version Jazzy Jalisco is possible due to the absence of counterparts for `rclcpp::GenericSubscription` and `rclcpp::GenericPublisher` for service and action server and clients. In Jazzy Jalisco, only the generic service client (`rclcpp::GenericClient`) exists [15]. However, the latest ROS2 version Kilted Kaiju has already additionally incorporated a generic service server (`rclcpp::GenericService`) and a generic action client (`rclcpp_action::GenericClient`), although the generic action server remains to be developed [40]. This progression between Jazzy Jalisco and Kilted Kaiju suggests that a complete generic implementation for services and actions will be possible in the future.

However, for ROS2 Connect, this necessitates that, given the current development state of the ROS2 versions, the type of services and actions must be known at compile time. Consequently, the type must be explicitly implemented in the code, which in turn implies that each service or action that is to be forwarded by ROS2 Connect must be explicitly implemented. In order to facilitate the provision of services and actions in ROS2 Connect without the necessity of modifying the ROS2 Connect code itself for each service or action, the implementation of services and actions has been outsourced to plugins, which encapsulate the service and action type-specific code and can be dynamically loaded at runtime by ROS2 Connect. Consequently, no explicit code changes to ROS2 Connect itself are necessary; only the implementation of a plugin. These plugins are loaded at runtime by the C++ library pluginlib, which is a ROS2 package and, as such, is part of ROS2 [12]. It facilitate the organization and implementation of plugins in a discrete package within the context of ROS2 semantics, which compile to shared libraries and thus, shared objects (so) in the case of Unix-like systems and dynamic-link libraries (dll) in the case of Windows systems [12]. Consequently, these libraries can be loaded dynamically, eliminating the requirement for explicit linking.

The ROS2 package ROS2 Connect-Plugins was created for the implementation of service and action server and clients. Each service or action server or client implements on of the abstract classes: `service::ServiceServer`, `service::ServiceClient`, `action::ActionServer`, or `action::ActionClient` (see Fig. B.12 and Fig. B.13 of appendix B). These classes are defined and exported by ROS2 Connect in headers which encompass the instantiation of the type-specific action or service client (`rclcpp_action::Client<ActionT>` or `rclcpp::Client<ServiceT>`) or server (`rclcpp_action::Server<ActionT>` or `rclcpp::Service<ServiceT>`) as well as the type-specific conversion from service or action calls (goals) and results to `Service ActionMessages` (see section 2.4.4), which contain raw serialized binary data, and vice versa. ROS2 Connect is thus only responsible for receiving `ServiceActionMessages` from the plugins or transmitting them to the plugins. It subsequently has the capacity to compress them if specified (see below) and forward them to the other party via the WebSocket connection. This renders the explicit implementation of the service or action server or client irrelevant for ROS2 Connect, provided that they fully implement the header exported by ROS2 Connect and deliver or receive `ServiceActionMessages` accordingly. It is essential that implementations correctly record the "goal id" of a call, and consequently the goal, within the `ServiceActionMessage` instance (see section 2.4.4). The "goal id", which is uniquely assigned to each goal, servers as the identifier of that goal and is therefore critical for ensuring that `ServiceActionMessages` exchanged between the *client* and *server* can be consistently associated with a specific call. This mechanism guarantees that the result of a remove service or action is delivered to the corresponding caller.
Additionally, each implemented plugin in ROS2 Connect-Plugins is assigned a type, which is utilized by pluginlib to export the plugins from ROS2 Connect-Plugins and to dynamically load them later in ROS2 Connect. For instance, a service server that facilitates the addition of two integers would be of the type "connect_plugins::AddTwoInts ServiceServer" and would extend the base class `service:: ServiceServer`.

In ROS2 Connect, dynamic loading, instantiation, and management of these plugins are implemented by the class `ServiceActionManager` (see Fig. B.14 of appendix B), which is itself instantiated by the WebSocket implementation (`WebsocketConnection Base`). As was the case previously with subscriber and publisher, this class utilizes ROS2 parameters, which define the service and action servers and clients, to load the necessary plugins at runtime utilizing pluginlib. The plugins are loaded and instantiated by the `ServiceActionManager`, through the means of four `pluginlib::Class Loader` instances [12], immediately following successful authentication of the other party and are maintained throughout the entire runtime of ROS2 Connect. As with subscribers and publishers, two threading models are implemented for services and actions: one using a shared thread and the other using a dedicated thread per service or action server or client, whereby the threads are part of the MultiThreadedExecutor. The shared thread employs a callback group of type MutualExclusive, while the dedicated thread has the option of utilizing a callback group of type Reentrant (see section 2.1.3.2 and below). Consequently, the entirety of service and action servers and clients that utilize the shared thread are constrained to processing a single call to one of the service or action server or clients per time unit. Therefore, dedicated threads must be utilized to enable parallel service and action calls.

Similar to the constraints imposed on subscribers and publishers, services and actions are also subject to restrictions in order to prevent the occurrence of infinite loops between the *client* and *server* nodes. Specifically, in the context of services and actions, a *client* and a *server* instance cannot simultaneously act as both a service / action server and client under the same service / action name.

The aforementioned ROS2 parameters to define the service and action server and clients are “service/server”, “service/client”, “action/server”, and “action/client”, which are all string arrays, with each entry being a JSON string, thereby enabling the definition of arrays of which each member constitutes a complex type (see 2.4.3). Each member, and thus each service or action server or client definition, contains the following attribute-value pairs:

- **channel**
  The attribute channel relates to the “service action channel” of the message representation previously described in section 2.4.4 and therefore is an `uint8_t` value in the range of 0 to 255. This is a crucial difference to subscribers and publishers, where the channel attribute corresponds to the channel of the message representation. However, given that services and actions utilize the extended message representation `ServiceActionMessage`, the channel of the message representation in invariably 251, and the value of the channel attribute is stored in the extended header of the `ServiceActionMessage`. Additionally, values ranging from 0 to 255 are available for both, actions and services, individually.
  Despite these differences, however, the same properties as for subscriber and publisher apply to the channel attribute. Thus, the channel attribute provides a link between a service (or action) server and client, as well as a link between the channel itself and a service (or action) type (see section 2.4.6).
- **type**
  The attribute type holds the type of the plugin, implementing a service or action server or client, which is to be dynamically loaded and instantiated by the `ServiceActionManager`. This also contrasts with subscribers and publishers, where the type attribute directly uses the topic type. However, given that the implementation of services and actions is outsourced to plugins that are dynamically loaded at runtime, it follows that the type of these plugins must be known for services and actions. This type is the type with which the plugins are exported

by pluginlib. To reiterate the aforementioned example, an example type would therefore be “connect_plugins::AddTwoIntsServiceServer”.

- **useOwnThread**
  The attribute useOwnThread specifies (as `bool` value) wether a service or action server or client should use an exclusive, dedicated thread or the shared thread. This attribute, therefore, correlates with the two implemented threading models, whereby services or actions that are to be executed in parallel with other services or actions must use an exclusive, dedicated thread.
- **allowSimultaneous**
  The attribute allowSimultaneous specifies (as `bool` value) wether simultaneous calls to a service or action should be permitted. However, this necessitates the utilization of an dedicated thread for the service or action in question. In the event that the allowSimultaneous attribute is set to true, the thread employs a callback group of type Reentrant, thereby enabling the processing of concurrent calls. In the event that the allowSimultaneous attribute is set to false, the thread employs a callback group of type MutualExclusive, which facilitates the serialization of calls. However, it should be noted that the majority of nodes that implement and thus offer services or actions do not have an implementation that enables the processing of simultaneous service or action calls.
- **serviceQos**, **feedbackQos**, and **statusQos**
  The attributes serviceQos, feedbackQos and statusQos contain the ID (as `uint8_t` value) of QoS profiles that have been defined by the independent ROS2 parameter “qos”. The attribute serviceQos is required for service and action servers and clients; the attributes feedbackQos and statusQos are exclusively required for action servers and clients.
  The attribute serviceQos, denotes the QoS profile that is applied to the service server or client. In the event of an action, this QoS profile is applied to the services utilized by the action. As previously described in section 2.1.3.2, this services facilitate the transfer of the goal and result between the action server and action client as well as offer the possibility to cancel a goal [14]. Accordingly, the attribute serviceQos is equivalent to the attribute qos of subscribers and publishers.
  Furthermore, the attribute feedbackQos corresponds to the QoS profile designated for the feedback topic of an action, through which the action server is capable of providing the action client with continuous feedback regarding the current execution status of the goal [14]. The attribute statusQos corresponds to the QoS profile assigned to the status topic of an action, through which the action server is capable of providing the action client with information on the goals it has accepted [14].
- **resultTimeout**
  The attribute resultTimeout, which is exclusively applicable to action servers, defines (as `int32_t` value) the duration in seconds for which a result calculated by an action server for a specific goal is valid before it is discarded. Consequently, the attribute determines whether a result should be cached for a particular goal and the duration of its caching, whereby a value of -1 signifies that the calculated result maintains its validity without expiration [14]. It is important to note that

the result of a goal is linked to the "goal id", rather than its parameters. Therefore, to retrieve the cached result, it is necessary to make a call using the corresponding "goal id" with the `async_get_result` method of an action client [14, 15]. The default value in rclcpp is 10 s and is also recommended for utilization in ROS2 Connect, unless there exists a particularly compelling reason to utilize a different value [15].

- **maxExecTime**
  The attribute maxExecTime (max execution time), which is exclusively applicable to service servers, defines (as `uint32_t` value) the duration in seconds following which the processing of a service call is aborted and, consequently the resources allocated for processing the call in ROS2 Connect are released. The creation of this attribute was necessitated by the absence of a cancellation option for service calls in ROS2 [2, 12]. In the event that a service server relayed by ROS2 Connect ceases to respond to a call, the caller is unable to cancel the call, which would, in instances where the service server and client in ROS2 Connect employ the shared thread, lead to the blocking of the shared thread. Consequently, the caller would be incapable of initiating a call to another service or action that also utilizes the shared thread. Therefore, ROS2 Connect implements a mechanism that allows for the cancellation of a service call if the actual, relayed service server does not respond within the max execution time. This results in the release of the shared thread and its associated resources. Given that the purpose of services is to facilitate prompt responses to calls, it is recommended that a max execution time (maxExecTime) of 10 s is to be established. However, it is possible to set a time limit for each service in order to enable services that may require more time.
- **compression**
  The attribute compression contains the ID (as `uint8_t` value) of a compression profile that has been defined by the independent ROS2 parameter "compression". This attribute is therefore identical in meaning, use, and possible values to the attribute of the same name for subscribers and publishers. Consequently, it defines whether the payload of a `ServiceActionMessage` should be compressed, and if so, which compression algorithm should be employed. However, in the case of services and actions, the compression profile can be specified for both servers and clients, whereby the compression profile always affects outgoing `ServiceActionMessage`s. This enables a service or action server and client communicating on a common channel to use two distinct compression profiles. In contrast to subscribers and publishers, it is possible to enable compression for services and actions even if useOwnThread is set to false. This is due to the fact that the shared thread is only capable of processing one service or action call per time unit, as described above.

The parameter files utilized for the performance evaluation of ROS2 Connect are documented in appendices A.3.2 and A.3.4 and contain exemplary definitions for the four parameters "service/server", "service/client", "action/server", and "action/client" for reference. The data medium accompanying this thesis contains the corresponding example ROS2 Connect-Plugin plugin implementations, see appendix C.

The mechanisms and implementations described in this section in conjunction with the ROS2 parameters "service/server", "service/client", "action/server", and "action/-client" enable the dynamic (and generic) nature of ROS2 Connect, insofar as no code changes of ROS2 Connect itself are required for any service or action to be relayed by ROS2 Connect.

2.4.10 Performance Evaluation

Performance evaluation is carried out through sequential and parallel time measurements using the procedure previously described in section 2.3.2.2. In the case of ROS2 Connect, a *server* node instance is initiated on host A while a *client* node instance is initiated on host B. Host C functions as a reverse proxy (Apache HTTP Server) to enable a connection between host A and B from outside the university network. The traffic between host B and C is secured via TLS; the traffic between host A and C is not. This corresponds to the planned deployment of ROS2 Connect, as previously outlined in section 2.4.1.
The designation of the timestamps remains constant. However, given that the entirety of the communication path is now covered by ROS2 Connect itself, all 30 aforementioned possible timestamps can be recorded, as illustrated in Fig. B.16 of appendix B. These 30 timestamps facilitate a comprehensive insight into the intra-processing times, thereby enabling, for example, the identification of congestion within ROS2 Connect and the evaluation of the performance of the compression algorithms as part of the entire communication path.
Figure 2.28 presents the results of the sequential time measurements, whereby time measurements were carried out for messages with a total size ranging from 12 B to 500 kB. As with Rosbridge and the DDS Router, the graph displays the median and

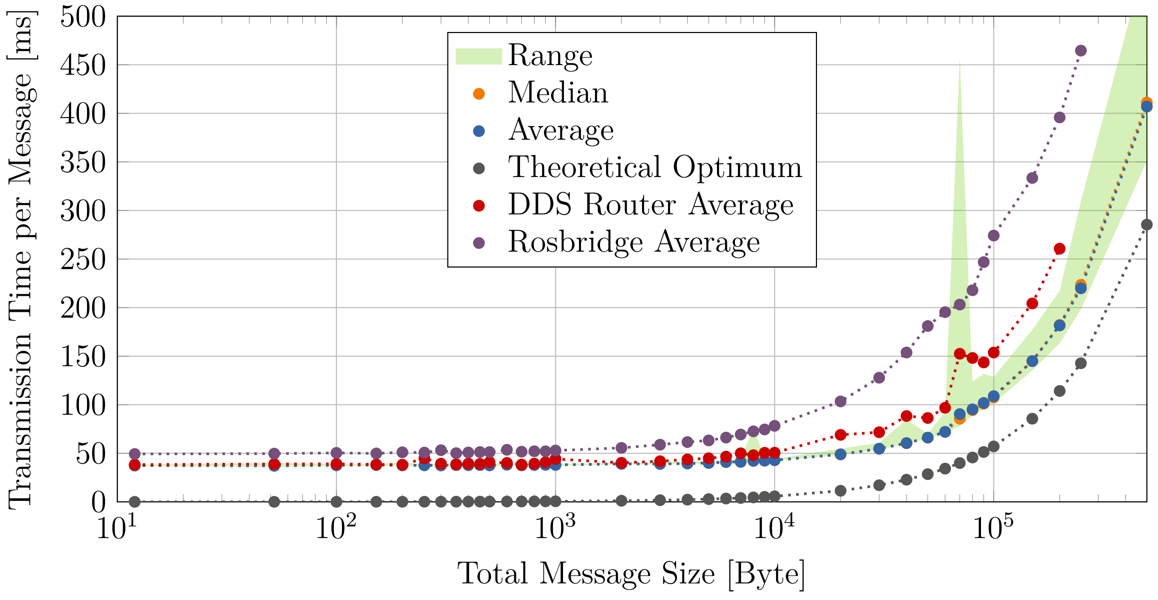


Fig. 2.28: Sequential Time Measurement of ROS2 Connect

average transmission times of a round trip, in addition to the range of the measured times and the theoretical optimum based on iperf3 bandwidth measurement. The graph also shows, for comparison, the average transmission times previously determined for Rosbridge and the DDS Router.

It can be observed that the average transmission time remains relatively constant up to total message sizes of 60 kB with a constant difference to the theoretical optimum of about 37.3 - 37.7 ms. For message sizes larger than 60 kB, the difference increases, reaching a maximum of 121.6 ms for a message size of 500 kB. In contrast to Rosbridge and DDS Router, ROS2 Connect demonstrates a stable difference to the theoretical optimum for significantly larger messages. In the case of the DDS Router, the difference was only stable up to 6 kB, similar in magnitude to ROS2 Connect. In the case of Rosbridge, the difference was only stable up to 1 kB and was more than 10 ms greater. The comparison of the maximum difference, which can only be carried out for messages up to 200 kB (as this is the largest message size that could be transmitted by the DDS Router), shows that ROS2 Connect also has the lowest at 67.5 ms, followed by the DDS Router with a maximum that is already more than twice as large at 147.4 ms and Rosbridge with 282.2 ms. Furthermore, ROS2 Connect has been demonstrated to exhibit the most minimal average transmission times. For a message with a size of 200 kB, ROS2 Connect requires only 181.7 ms, while DDS Router required 260.8 ms and Rosbridge 395.8 ms. Additionally, ROS2 Connect is able to transmit messages with a size of 500 kB faster than Rosbridge transmits messages with a size of 250 kB. This outcome demonstrates that ROS2 Connect exhibits reduced transmission times and enhanced stability in terms of difference from the theoretical

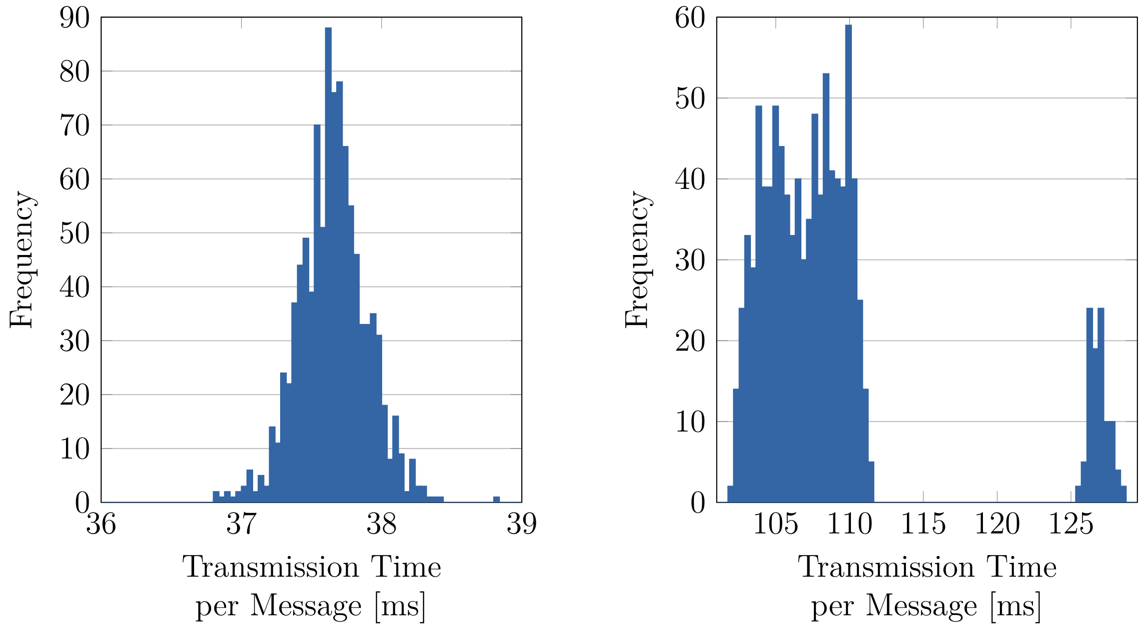


Fig. 2.29: Left: Distribution of time measurement values for 100 B total message size, Right: Distribution of time measurement values for 100 kB total message size

optimum when compared with DDS Router and Rosbridge.
Furthermore, the graph reveals that the mean and median transmission times are nearly identical across all measured messages sizes. With an exception of a substantial deviation observed for a message size of 70 kB, the range between the minimum and maximum transmission times per message size is considerably more dense compared to Rosbridge and the DDS Router. This is also reflected in the CV, which averages 2.8% across all measurements (median 0.7%) with a maximum of 42.1% for 70 kB and a minimum of 0.5% for 9 kB. This indicates, apart from the observed exception at 70 kB, a very low variability of the round-trip times which is even better than a typical WAN connection, according to Lee at al. [32].
As illustrated in Fig. 2.29, two histograms with 75 bins are presented for further evaluation of the distribution of the measured values. The histogram on the left depicts the transmission times for a message size of 100 B, while the histogram on the right illustrates the transmission times for a message size of 100 kB. In the case of 100 B, which is among the message sizes with a minimal range of transmission times in Fig. 2.28, the measured values are approximately normal distributed with an average value of 37.6540 ms and a median of 37.6505 ms. The very small difference of 3.5 µs, with the mean value that exceeds the median, indicates a subtle right skew. The CV of the measured transmission times is minimal at 0.644%, as is the standard deviation at 0.834 ms. In the case of 100 kB, which shows a clearly recognizable range in Fig. 2.28, yet the mean and median appear almost identical and are at the lower end of the range, the histogram reveals a bimodal distribution with a large mode at approximately 101-112 ms and a smaller mode at approximately 125-129 ms with an average of 108.84 ms and a median of 107.47 ms. The CV of the measured transmission times is at 5.93% (which remains withing satisfactory limits according to Lee at al. [32]) and the standard deviation is at 6.45 ms.
An explanation for this distribution of transmission times in case of 100 kB can be found by examining certain time intervals of the time measurement in the plot shown in Fig. 2.30. This plot shows the actual sending times from *client* to *server* [D, DD] and from *server* to *client* [K, KK], as well as the additional intra-processing times [A, CC], [E, JJ] and [L, NN]. It can be observed that the intra-processing times are minimal, and when compared with the actual sending times, contribute negligibly to the total transmission time. The average and median intra-processing times are 1.6 ms, of which the majority, with an average of 1.04 ms, is attributable to the intra-processing time of host A (interval [E, JJ]), which has significantly less computing capacity than host B. Examining the actual sending times reveals two findings:

1. First, the sending time from *client* to *server* (interval [D, DD]) exhibits a regularly repeating pattern. This observed pattern demonstrates a pronounced increase in the sending time, which is then succeeded by a sharp yet less intense decline. This is subsequently followed by a gradual and consistent decrease in the sending time reaching a local minimum. This pattern then repeats itself, creating a typical sawtooth pattern. This phenomenon can be observed in messages exceeding 60 kB in size, which is precisely the message size at which an increasing difference between the average transmission time to the theoretical optimum can be observed.

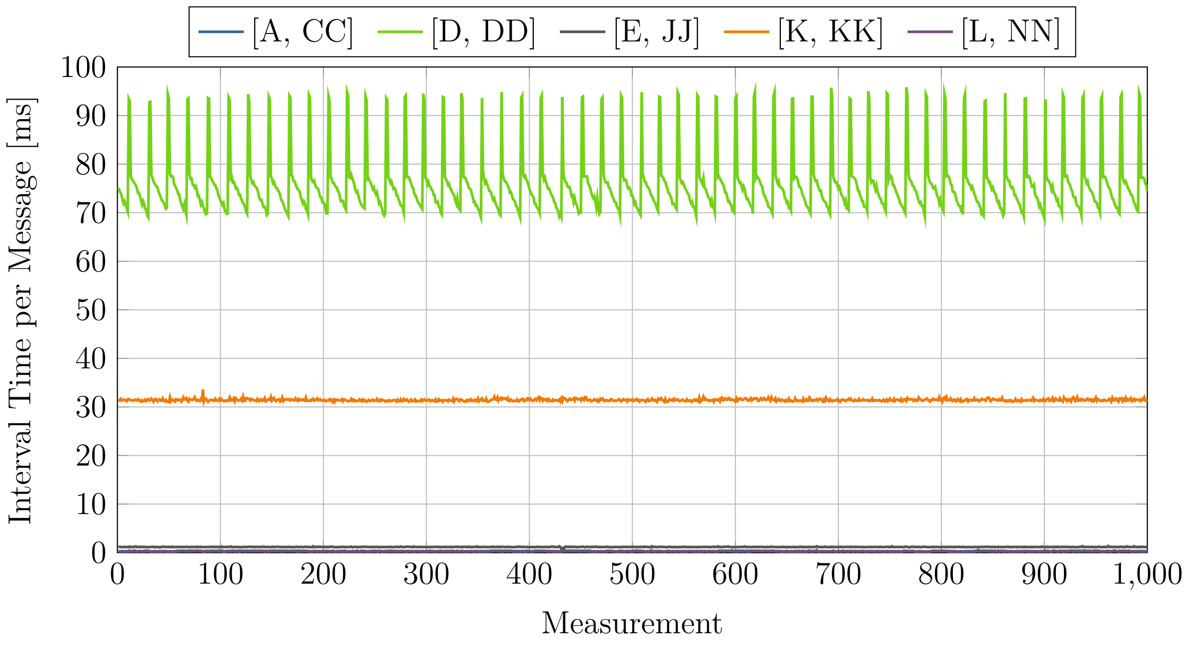


Fig. 2.30: Time Intervals of sequential time measurement for 100 kB total message size

The origin of this phenomenon is caused by the congestion control algorithm used by default by the Linux kernel since version 2.6.19, which is the TCP CUBIC algorithm [41], in conjunction with the low uplink bandwidth of 18.45 MBit/s. The CUBIC algorithm is a loss-based algorithm, which means that congestion is signaled as soon as packet loss occurs [42]. The occurrence of a packet loss, for example, due to a Traffic Policer (see [43]), results in CUBIC cutting the congestion window size to 70% [42]. This causes the window size to become insufficient to what is needed for the transmission of a 100 kB message, necessitating the execution of one or more acknowledgment (ACK) round trips by the *client* to facilitate the transmission of the complete message [42]. This corresponds to the observed sudden increase in sending time. Subsequently, the CUBIC algorithm initiates an increase in the congestion window size, characterized by an initial concave rapid increase, and succeeded by a convex slower increase [42]. This corresponds to the sudden drop in sending time followed by the slower but steady decrease. Consequently, the observed sawtooth pattern is attributable to the manner in which the TCP CUBIC congestion control algorithm responds to packet loss, which occurs regularly on the lower bandwidth link for messages exceeding 60 kB. Furthermore, this sawtooth pattern is the underlying reason of the two modes observed in the right histogram of Fig. 2.29.

To illustrate this correlation, sequential time measurement was conducted again for a message size of 100 kB. In this particular instance, however, the Linux kernel was reconfigured to use the Bottleneck Bandwidth and Round-trip propagation time (BBR) congestion control algorithm on host A, B, and C. This congestion control algorithm operates in a fundamentally different way. Rather than emphasizing packet loss, it models the path by continuously measuring the bottleneck bandwidth (r_b) and the minimum round-trip-time of the path (t_{prop}) [44]. BBR then maintains the in-flight-amount of data at approximately

$$b = r_b \cdot t_{prop} \tag{2.3}$$

where:

b	bandwidth-delay product in bit
r_b	bottleneck bandwidth in bit/s
t_{prop}	minimum round-trip-time in seconds

with BBR transmitting packets at precisely this rate using a pacing timer [44]. Consequently, BBR ensures that the path is constantly filled, thereby maintaining a stable measured rate, which prevents significant rate drops caused by individual packet losses [44].

Figure 2.31 shows the result of this time measurement with BBR as congestion control algorithm. The sawtooth pattern previously observed under TCP CUBIC has completely disappeared under BBR. The sending time from *client* to *server* (interval [D,DD]) is consistent, exhibiting minimal variations. Additionally, a notable decrease in sending time can be observed under BBR. While no values below 68.85 ms were recorded under TCP CUBIC, minimal sending time values of 63.36 ms were measured under BBR. The average sending time exhibited decline from 75.82 ms (median 74.48 ms) to 63.85 ms (median 63.86 ms).

However, the sending time from *server* to *client* remains unaffected under BBR. Only a few minor spikes are visible, but these cannot be definitely attributed to BBR and are likely attributable to a transient variation in bandwidth.

The histogram of Fig. 2.31 indicates that the total transmission time, with the exception of a few outliers, is once again approximately normally distributed with a subtle left skew. The CV of the total transmission time decreased from 5.93% to 0.41%.

This finding indicates that the observed sending time pattern is attributable to the TCP CUBIC congestion control algorithm used by default by the Linux kernel. Despite this finding, no absolute recommendation for the utilization of BBR can be made. The reasons for this are, on the one hand, that BBR is currently still undergoing active development [44]. On the other hand, compared to TCP CUBIC, BBR sometimes distributes resources unfairly [45, 46]. The use of BBR or alternative congestion control algorithms should therefore be explicitly investigated in order to determine the best choice in the context of Telelab for ROS2 Connect.

2. Second, the sending time from *server* to *client* (interval [K, KK]) is significantly shorter than the time from *client* to *server* (interval [D, DD]). This difference results from, on the one hand, the asymmetric bandwidth between hosts A and B (uplink 18.45 MBit/s, downlink 58.3 MBit/s). On the other hand, as already

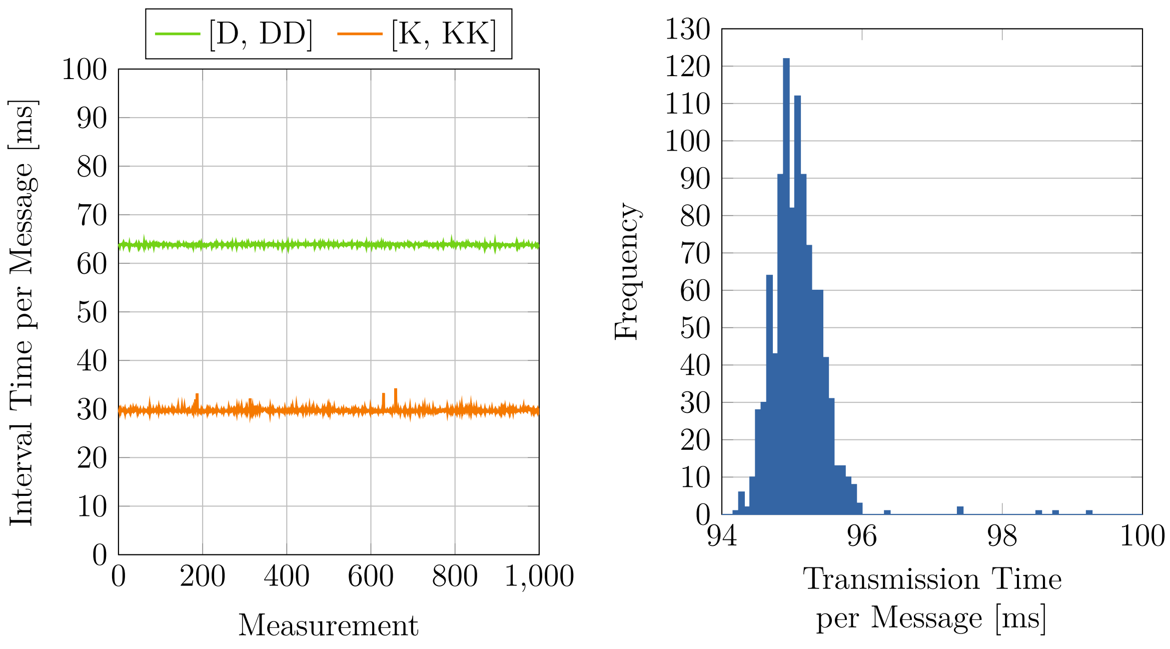


Fig. 2.31: Time Intervals and distribution of measurement values for 100 kB total message size using BBR as congestion control algorithm

outlined in section 2.1.4, the fact, that messages from a WebSocket client to a WebSocket server must be masked according to the WebSocket specification [19], while messages from a server to a client do not have to be. To further elaborate the impact of data masking on sending time, sequential time measurement was conducted again for a message size of 100 kB. In this particular instance, both the *client* and *server* node operated on the same host, both in separate ROS2 / DDS domains, with the objective of ensuring their isolation. This configuration enables time measurement with very high bandwidth between *server* and *client* which is not subjected to relevant asymmetries. Since ROS2 Connect facilitates the recording of all possible intra-processing times, the effect of this masking on the actual transmission time can be measured.

Figure 2.32 shows the result of this time measurement. The uncorrected sending times [D, DD] and [K, KK] are represented in blue and gray, respectively, which also demonstrate a significantly longer sending time from *client* to *server*, with an average of 0.17 ms, as compared to an average of 0.08 ms from *server* to *client*. Hence, the transmission from *client* to *server* is 112.5% longer than from *server* to *client*. However, since the time required for a call to Boost.Beast `async_write` was recorded using the timestamps "CW" and "SW", the time required to transfer the data to be sent to the Linux kernel buffer was measured on the one hand, and the time required for Boost.Beat to mask the data in the case of *client* to *server* was measured on the other hand. This time ([D, CW] and [K, SW]) can then be subtracted from the corresponding intervals [D, DD] and [K, KK] to obtain a corrected sending time, which is shown in Fig. 2.32 in green and orange. The corrected sending times now demonstrate nearly

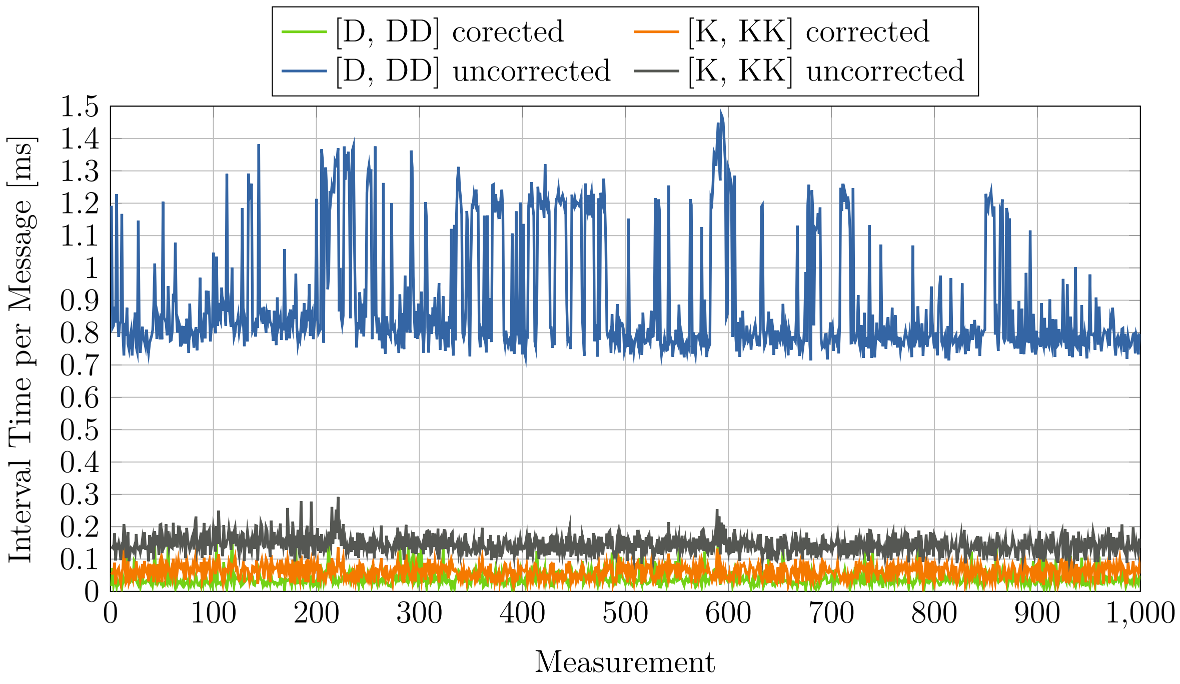


Fig. 2.32: Uncorrected and corrected actual sending times of sequential time measurement on localhost for 100 kB total message size

complete overlap, with an average corrected sending time from *client* to *server* of 0.05 ms and from *server* to *client* of 0.04 ms.

This finding indicates that the impact of data masking on sending time is negligible. However, the time measurement reveals that the data masking leads to elevated time variability (uncorrected [D, DD] CV: 27.12%, corrected [D, DD] CV: 15.56%), with the cause most likely being operating system scheduling. Consequently, the previously observed difference in sending times is attributable to the asymmetry of the bandwidth between host A and B.

Finally, parallel time measurement was conducted, the results of which are presented in Figure 2.33, whereby time measurements were carried out for messages with a total size ranging from 12 B to 100 kB with a degree of parallelization from 1 to 10. This parallel time measurement was performed using ThreadedSubscriber and ThreadedPublisher (see section 2.4.6) to enable ROS2 Connect to process up to 10 parallel topics truly in parallel via its MultiThreadedExecutor. The result indicates that, similar to the case of Rosbridge, the transmission time remains constant even with 10 parallel topics for messages up to 10 kB in size. For messages with a size of 100 kB, a linear increase in transmission time can be observed, beginning with three parallel messages. This phenomenon is again similar to the behavior exhibited by Rosbridge which also utilizes a MultiThreadedExecutor. As previously described, this phenomenon illustrates the behavior of WebSockets, which are constrained to transmitting a single message per time when confronted with network congestion. In order to prevent topics that only publish small messages from starving while topics that publish large messages constantly “block” the connection, it is advisable to run

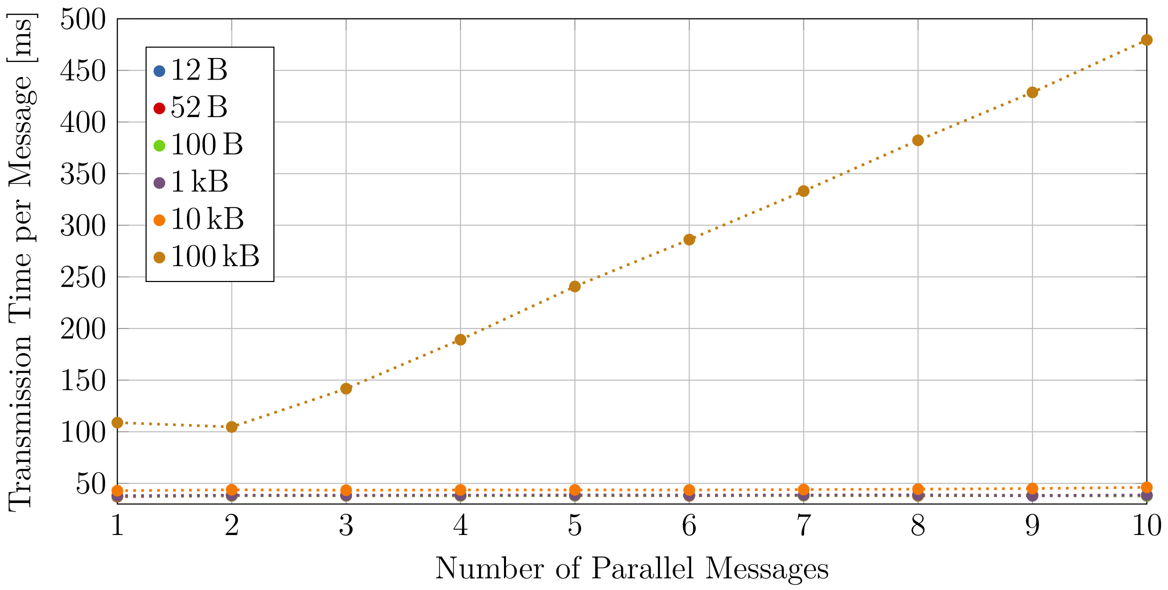


Fig. 2.33: Parallel Time Measurement of ROS2 Connect

multiple *server* and *client* instances in parallel. This will allow for the distribution of data across multiple WebSocket connections. For instance, the transfer of large image data can be facilitated through an independent WebSocket connection, while smaller sensor data employs a distinct connection for transmission. ROS2 Connect allows the parallel execution of multiple instances. However, caution must be taken to ensure that, for instance, in the case of two parallel *client* instances, there is no subscriber and publisher for the same topic across both instances, as this can lead to endless transmission loops (see section 2.4.6).

2.4.11 Suitability, Conclusion and Outlook

Finally, the suitability of ROS2 Connect for ROS2 over WAN in the context of Telelab will be assessed. Given that ROS2 Connect is a completely new development as part of this thesis, it is particularly well-suited for integration into Telelab.

ROS2 Connect's comprehensive implementation of Authentication, Authorization and Accounting mechanisms facilitates direct connectivity with the Telelab reservation system, thereby enabling the authentication of incoming connections. The authorization process is executed through the dynamic definition of publishers, subscribers, service clients, and action clients, within the parameter files. Therefore, ROS2 Connect is designed to prevent users from injecting data, or triggering unwanted services or actions. Additionally, all user actions are documented and can be traced.

Performance evaluation has demonstrated that ROS2 Connect exhibits a superiority over both Rosbridge and DDS Router. It demonstrated significantly higher stability in terms of transmission times, as well as significantly shorter transmission times when compared to Rosbridge and DDS Router. The distribution of transmission times is approximately normal, with the exception of effects caused by the TCP CUBIC congestion control algorithm. Despite the absence of any explicit mention in this thesis, ROS2 Connect imposes no limitations on the maximum message size. Messages up to the tested 2 MB in size can be transmitted without any abnormalities. Furthermore, ROS2 Connect implements a variety of compression algorithms to compress large messages with, for instance, lz4 default compression, without significant time losses before sending. As with Rosbridge, ROS2 Connect employs WebSockets as transport mechanism, which are inherently NAT friendly, thereby ensuring seamless connectivity without any issues related to NAT routers along the connection path.

In summary ROS2 Connect is an optimal solution for ROS2 over WAN, applicable beyond the context of the Telelab.

Nevertheless, there are still further possibilities for optimization and further development:

- As previously mentioned during the performance evaluation, further investigations should be carried out to assess the impact of different congestion control algorithms. While BBR demonstrated favorable behavior in the context of time measurements, it has the potential to result in unfairness in real-world application if multiple applications compete in parallel for the available bandwidth [45, 46]. However, a comparative analysis of congestion control algorithms and an evaluation of their impact on ROS2 Connect under real-world conditions may yield more suitable alternatives to TCP CUBIC in regard of smoothing out and lowering transmission time.
- As stated in section 2.4.9, the implementation of service and action server and clients cannot be executed in a completely generic manner under ROS2 Jazzy Jalisco or ROS2 Kilted Kaiju [15, 40]. However, the development progression from Jazzy Jalisco to Kilted Kaiju suggesst the feasibility of achieving this objective in the future. Consequently, upon the release of a ROS2 distribution that facilitates the generic implementation of service and action servers and clients, it is advisable to extend the implementation of ROS2 Connect to enable generic service and action servers and clients, in addition to generic subscribers and publishers. This will enable the definition of service and action servers entirely through the parameter file, thereby utilizing a ROS2 parameter, eliminating the necessity for implementing a ROS2 Connect plugin, as is currently required.
- With regard to the definitions of subscribers and publishers (as well as service and action servers and clients), the current development status of ROS2 Connect is such that under normal conditions, the *server* and *client* have mirrored ROS2 parameters. Consequently, if the *server* has a subscriber for topic `/a`, the *client* has a publisher for topic `/a`. One potential optimization in this scenario would involve the *server* being the sole source for the definitions, with the *client* receiving them after a successful connection has been established. This approach would eliminate the need for duplicate parametrization of the *client*.

In the context of Telelab, the addition of new or the removal of existing topics, as well as the exchange of QoS profiles, could then be accomplished without necessitating the download of a new parameter file by users.

- The final point mentioned here regarding the optimization of ROS2 Connect concerns compression. As previously mentioned in section 2.4.6, the compression of topic data necessitates the utilization of ThreadedSubscriber / -Publisher, and the compression algorithms and its settings must be defined prior to the execution of ROS2 Connect. However, this significantly restricts its application because to truly benefit from compression of topic data, real-time metrics must be known about the current bandwidth, the compressibility of the data, and the performance of the host systems running the *server* and *client* node with regard to compression. A possible extension of ROS2 Connect would therefore be that

  1. after startup, the *client* and *server* compress predefined data-sets in the background, measuring the time required for this process. This enables the estimation of the performance of the host system.
  2. The *client* and *server* evaluate the compressibility and size of the data to be transferred at runtime to estimate the potential benefits of compression for a certain topic.
  3. The *client* and *server* perform time measurements for individual topics during runtime to estimate the bandwidth of the connection.

  This bundled information can then be exchanged between the *client* and *server* and used to decide at runtime whether to compress the data to be transferred for a specific topic, selecting the most suitable compression algorithm with the most suitable settings. To accomplish this objective, the *client* or *server* would then communicate the decision to compress to the other party so that both can switch to a ThreadedSubscriber / -Publisher if necessary to begin transferring compressed data. For the *client* and *server*, a ROS2 parameter would be employed to specify the maximum number of additional threads that can be utilized for compressed topics to limit the number of threads. This extension would enable the maximum potential of topic data compression to be achieved.

Chapter 3

Integration of two Eduard Robots into Telelab

This chapter provides a brief overview of the integration, of two Eduard robots from the EduArt Robotik GmbH [47] into the Telelab e-learning infrastructure. The integration of these two robots is intended to convert the Telelab infrastructure from an in-house robot control software (see Schott [36]) to ROS2, thereby enabling the provision of ROS2-related learning content on the Telelab platform.

To achieve this objective, a series of modifications were implemented in both the hardware and software components of the Eduard robots. The robots were integrated into the university network via wireless connectivity, thereby enabling seamless ROS2 communication and data exchange. In addition, multiple ROS2 nodes were developed, while the database system and the pre-existing Telelab user interface were adapted to facilitate full integration.
Furthermore, ROS2 Connect, described in detail in the preceding chapter, enables remote clients to teleoperate the Eduard robots over the Wide-Area-Network.

3.1 Eduard Robots

The Eduard research & development platform describes robots that, at the time of writing, possess four wheels and are classified into two distinct configurations: one configuration equipped with fixed wheels and one configuration fitted with mecanum (Swedish) wheels with rollers oriented at 45° [47]. Given that all four wheels are driven in both configurations, Eduard implements two kinematic models: the skid steer model, in which the wheels on one side are always driven at the same speed; and the omni-directional kinematic model based on four independently driven mecanum wheels [48]. It therefore follows that an Eduard robot equipped with four fixed wheels, with the wheels on one side driven at the same speed, has a degree of steerability of $\delta_s = 0$, a degree of mobility of $\delta_m = 2$, and thus a degree of maneuverability of $\delta_M = \delta_s + \delta_m = 2$ [49, 50]. However, since the configuration space involves three degrees of freedom (x, y, θ), while the Eduard robot in the four-fixed-wheel configuration can independently control only two of them (x, θ) through its two control inputs (the angular velocities of the left and right wheels), it follows that the Eduard robot with four fixed wheels is a non-holonomic robot [50]. In contrast, an Eduard robot equipped with four mecanum wheels, each of which can be driven independently, has a degree of steerability of $\delta_s = 0$, a degree of mobility of $\delta_m = 3$, and thus a degree of maneuverability of $\delta_M = \delta_s + \delta_m = 3$ [50, 51]. Since the configuration space involves three degrees of freedom (x, y, θ), and the four-mecanum-wheel configuration provides independent control of these three degrees of freedom via appropriate combinations of the four wheel angular velocities (with the wheel angular velocity of one of the wheels being constrained by the kinematic model), it follows that the Eduard robot with four mecanum wheels is a holonomic robot [50].

In the context of Telelab, two Eduard robots are integrated: one with fixed wheels and skid-steer kinematics, and another with mecaum wheels and omni-directional mecanum kinematics (see Fig. 3.1). The next sections discuss the robots' installed hardware and its extension for integration, as well as the internal and EduArt-developed software.

3.1.1 Hardware

The Eduard robots are constructed from a base plate of milled aluminum, on which most of the hardware is mounted, including the four motors, the embedded computer (Siemens SIMATIC IPC BX-21A), the so-called "Free Kinematics Kit", a wireless router (tp-link TL-WR802N), a 6 degrees of freedom inertial measurement unit (IMU), and the battery (4.5 Ah at 17.5 V to 30 V). The base plate also supports a structure made of MakerBeam XL aluminum profiles that holds the side panels and top plate, on which the charging port, the on / off switch, and an e-stop are mounted. The aluminum profile structure also holds two Time of Flight (TOF) sensors at the front and rear (ST VL53L8CX), which are embedded in circuit boards that also contain LEDs for signaling various states and illuminating the direction of travel.
The "Free Kineamtics Kit" is a stack of circuit boards that integrate various functions and is connected to the Eduard robot's embedded computer via an Ethernet adapter board, see Fig. 3.2. The circuit boards implement several important functions: The first circuit board implements the Ethernet interface and a Controller Area Network (CAN) bus, which connects the circuit boards that hold the TOF sensors

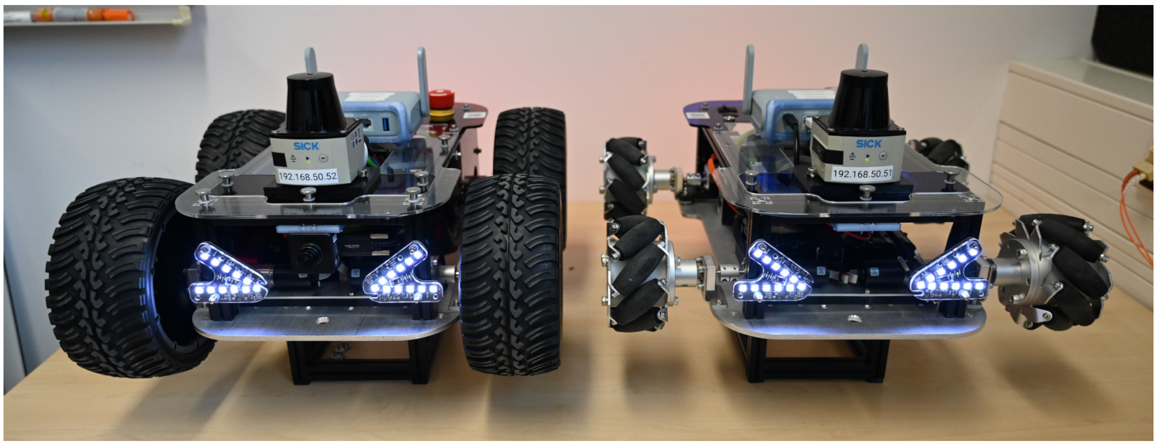


Fig. 3.1: "Eduard Red" with fixed Wheels, and "Eduard Blue" with mecanum wheels

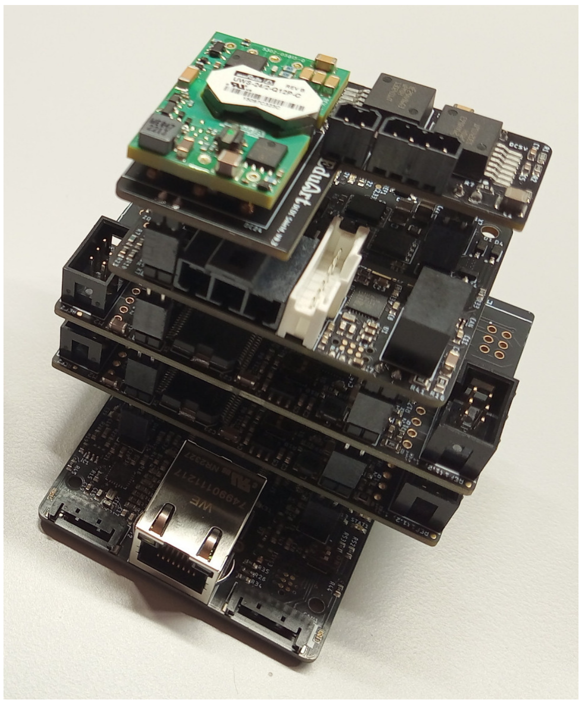


Fig. 3.2: Free Kinematics Kit (cropped from [52])

and the LEDs [52]. The second circuit board implements motor controllers (drivers), which generate voltages for the motors and simultaneously read the values of the encoders embedded in the motors [52]. The third circuit board controls battery charging, monitors its status, and evaluates the e-stop (power management module) [52]. Yet another board implements several DC / DC converters to generate 24 V for the embedded computer, as well as freely available 5 V and 12 V (auxiliary power supply module) [52].
Connecting the “Free Kinematics Kit” to the embedded computer allows the computer to read the various sensor and status values and to send control commands to the motor controllers to drive the robot. It is important to note that the “Free Kinematics Kits” is not connected directly to the embedded computer via Ehternet; rather, it is connected via an Ethernet-to-USB converter. This is because the Ethernet controller of the “Free Kinematics Kit” only supports link mode 10base-T half duplex, whereas the Network Interface Controllers (NICs) of the embedded computer only support 10base-T, 100base-T, or 1000base-T full duplex.

In summary, the Eduard robots can, out-of-the-box, detect their immediate surrounding thanks to their TOF sensors, which enable, for example, collision avoidance. The encoders detect wheel movements, which enables odometry and the implementation of controllers, e.g. PID, for velocity regulation and motion control.
To integrate the robots into the Telelab for their intended use, two additional optical sensors were added: a camera facing the forward direction and a 2D Light Detection and Ranging (LiDAR) scanner. In addition the wireless router was replaced.

The camera, a See3CAM_24CUG by e-con Systems®, possess a sensor resolution of 2.3 MP and is capable of capturing Full HD footage at a rate of up to 120 frames per second [53]. It is equipped with a global shutter mechanism and offers a field of view (FOV) of 128.2° diagonally, 104.6° horizontally, and 61.6° vertically [53, 54]. An important feature of the camera is its USB Video Class (UVC) compatibility, which facilitates the transmission of compressed and encoded MJPEG images [53]. This enables the direct capturing and processing of camera images without the need for the embedded computer to encode the raw image data first, which significantly reduces computing demand. The subsequent section 3.1.2.1 details the ROS2 device driver, developed for the purpose of image capture and publication.
The camera facilitates several key functions. Primarily, it provides visual feedback in the direction of travel during teleoperation. Furthermore, the camera is intended to be integrated into a variety of Telelab experiments planned for the future, including, but not limited to, camera calibration aimed at determining the intrinsic camera parameters and subsequent correction of the camera image distortion. Moreover, the camera is envisioned as an essential component in the development of autonomous docking strategies for the robot with a charging station.
For this purpose, the camera is mounted in a horizontally central position on the front-facing aluminum profile of the Eduard robot using a custom 3D-printed adapter (see Fig. 3.3) and connected to the embedded Computer via its USB 3.2 interface.
The 2D LiDAR scanner, a TiM571 by SICK AG, operates based on the High Definition Distance Measurement (HDDM) principle, which is a statistical pulse time-of-flight technique developed by SICK AG [55]. To this end, the TiM571 employs a stationary near-infrared (NIR) laser diode with a wavelength of 850 nm and a rotating mirror, thereby achieving a horizontal field of view of 270 ° at a scan frequency of 15 Hz, and an angular resolution of 0.33 °, with a measurement range spanning from 0.05 m to 25 m [56]. These properties enable the capture of 811 measurement points per

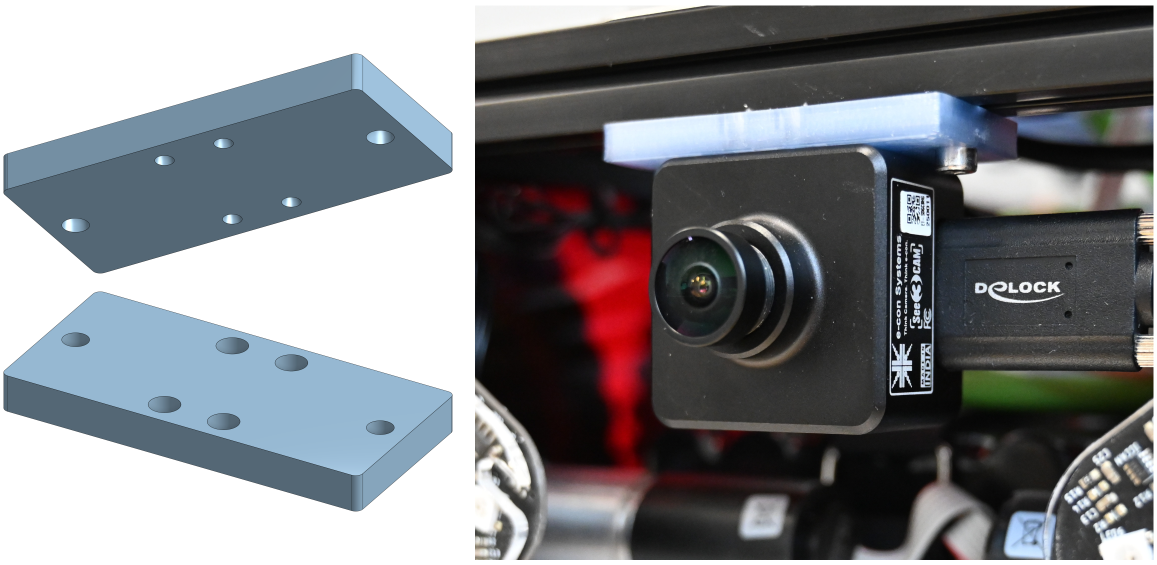


Fig. 3.3: Left: 3D-Model of mounting adapter for See3CAM_24CUG, Right: Mounted See3CAM_24CUG

scan, with each measurement point being associated with two values: the distance in millimeters and the Received Signal Strength Indicator (RSSI) as a dimensionless 8-bit number [55].
The utilization of this LiDAR scanner enables a wide range of applications. On the one hand, the scanner is intended to be integrated into a variety of Telelab experiments planned for the future focusing on mapping, navigation, and point cloud generation, thereby supporting the implementation of algorithms such as Iterative Closest Point (ICP). Moreover, as the LiDAR scanner is a key technology for mapping and navigation, it is envisioned as an essential component in the development of an autonomous charging strategy for the robot to precisely approach the charging station before docking.
For this purpose, the LiDAR scanner is mounted in a horizontally central position on a plate embedded in the top plate of the Eduard robot (see Fig. 3.5, where the embedded plate is shown in gray) using a custom 3D-printed adapter (see Fig. 3.4) and connected to the embedded Computer via its Ethernet interface. Furthermore, the precise position of the laser scanner, particularly the exit aperture of the laser beams, was measured in relation to the robot's base coordinate system. This facilitates the transformation of measurement points in the scanner's coordinate system into the robot's coordinate system through the utilization of the ROS2 library tf2. The LiDAR scanner is powered with 12 V of electricity, which is supplied by one of the DC / DC converters of the auxiliary power module of the "Free Kinematics Kit", and is protected by an 800 mA fuse.

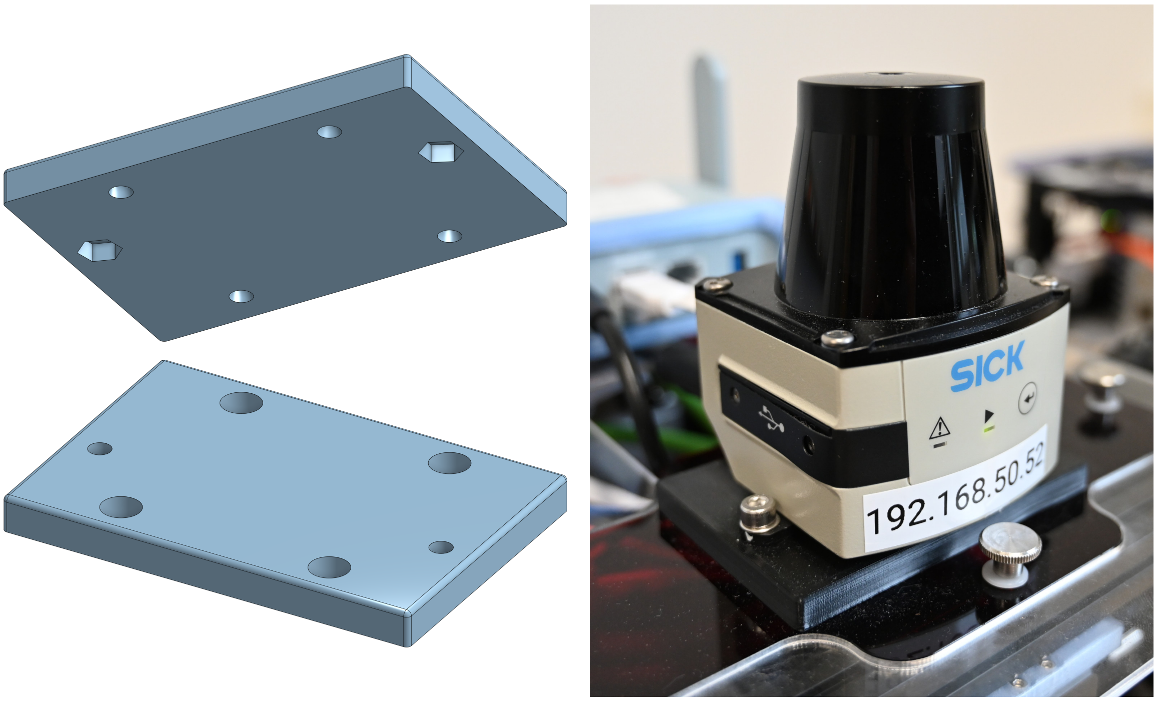


Fig. 3.4: Left: 3D-Model of mounting adapter for SICK TiM571, Right: Mounted TiM571

Additionally, section 3.1.2.2 details the ROS2 device driver, developed for the purpose of configuration of the TiM571 and the acquisition of scan data.

Finally, the extension and modification, respectively, of the Eduard robots with regard to the built-in wireless router is described. The initially installed tp-link TL-WR802N was substituted by a GL.iNET GL-MT3000 (Beryl AX). The reason behind this substitution is that the original tp-link router offers only a limited bandwidth capacity, with a maximum of 100 Mbit/s in bridge mode, i.e., when receiving a wireless signal and subsequently bridging it to the Ethernet port [57]. In contrast, the Beryl AX supports a bandwidth capacity of up to 2402 Mbit/s in the same configuration [58]. Details regarding the integration of the Eduard robots into the university network, facilitated by the Beryl AX router, is described in the subsequent section 3.2.
While the original tp-link router was enclosed within the Eduard robot's body, the Beryl AX was mounted on the top plate, both to free up internal space and to ensure a direct, unobstructed line of sight between the router and its access point. In order to implement this modification, it was necessary to modify the top plate of the Eduard robot, both to provide a larger opening for cable routing and to create mounting holes for attaching a holder for the Beryl AX. The original version of the top plate (left) [47] and its modified counterpart (right) are depicted in Figure 3.5. The cut-out was extended in the vertical and horizontal direction, and two new holes for mounting the Beryl AX holder were added above the extended opening. Furthermore, two of the original holes, which had been utilized to attach a carrying handle for the Eduard robots, were removed. This modified Eduard robot top plate was precisely manufactured from 3 mm-thick transparent Plexiglas® XT using a laser cutter. The

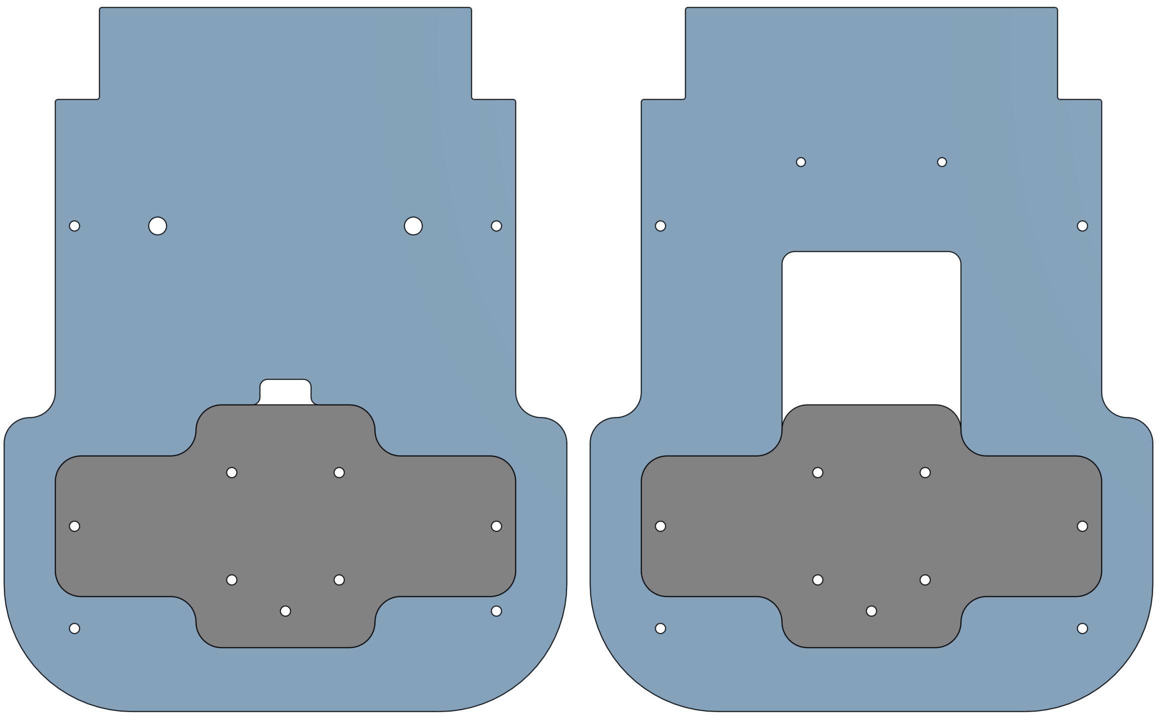


Fig. 3.5: Left: Original Eduard top plate [47], Right: Modified Eduard top plate

gray plate, which partially covers the cutout, remains unchanged and, as previously described, is employed for mounting the SICK TiM571 LiDAR scanner.
The 3D-printed holder for the Beryl AX, which is mounted using the two newly created holes in the top-plate (see Fig. 3.6), is based on an adapted 3D-Model originally published on Thingiverse by dunkndonuts under the Creative Commons (CC) Attribution-ShareAlike (BY-SA) 3.0 license [59]. The original model was modified by enlarging the mounting holes to a diameter of 5 mm to facilitate the use of M4 screws for installation. Furthermore, 5 mm spacers were incorporated into the underside of the holder to enhance its ventilation, thereby facilitating effective cooling for the Beryl AX. The Beryl AX itself is powered with 5 V of electricity, which is supplied by one of the DC / DC converters of the auxiliary power module of the "Free Kinematics Kit", and is protected by an 3.1 A fuse.

3.1.2 Software

The Eduard robots are supplied directly by EduArt Robotik GmbH with extensive software. This involves the implementation of complete hardware abstraction via ROS2 nodes, interaction options such as control via commercially available joysticks, including the PlayStation DualSense® Wireless Controller, and ready-to-use Gazebo simulations [60]. In the context of Telelab, two of the software components developed by EduArt are currently in use.

The first component utilized is the *eduard-ethernet-gateway-bot* node, provided by the ROS2 package "edu_robot", which is executed directly on the robot. This node implements the full hardware abstraction of the robot, evaluating the data of all

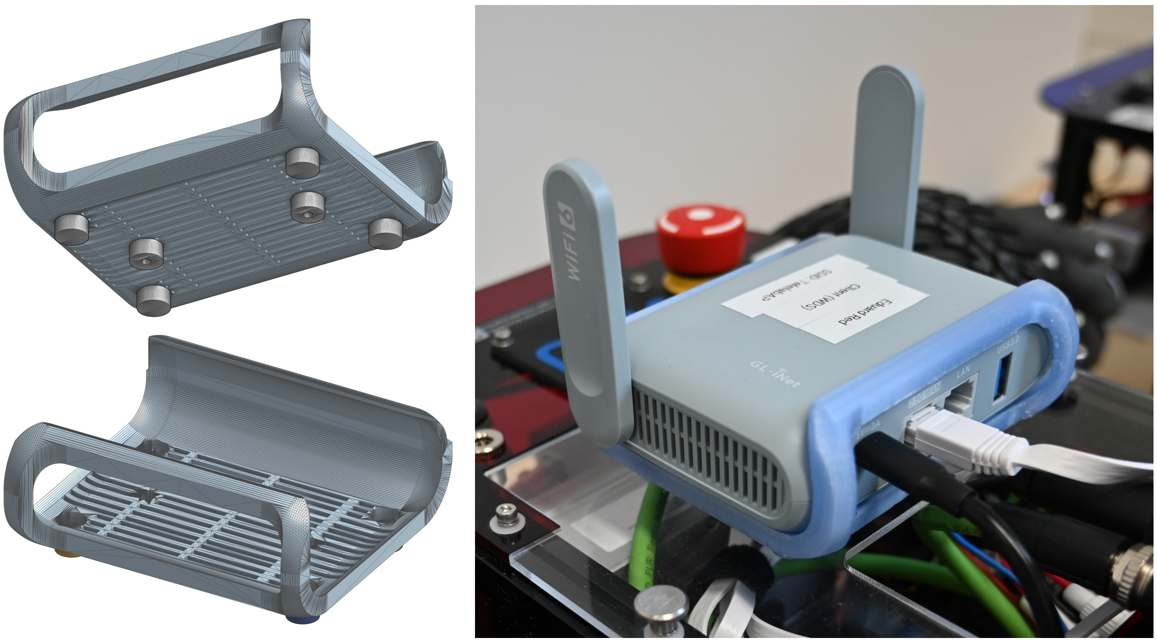


Fig. 3.6: Left: Adapted 3D-Model of mounting adapter for Beryl AX, Right: Mounted Beryl AX

installed sensors and publishing them to ROS2 topics [60]. It also includes the implementation of the two kinematic models, enabling control either by specifying `geometry_msgs/Twist`, i.e., spatial velocities, or by directly setting the individual angular velocities of the motors, which enables the implementation of custom control algorithms [60]. In addition, the node provides odometry, tf2 transformations between various defined frames, collision avoidance based on the TOF sensors, the periodic publication of the robot's status, and diagnostic insights [60]. For this purpose, the *eduard-ethernet-gateway-bot* node establishes a serial connection via the Ethernet interface to the "Free Kinematics Kit", which enables *eduard-ethernet-gateway-bot* to transmit commands to the motor controllers or CAN devices and to acquire the data of all installed sensors [60]. Furthermore, this connection also allows *eduard-ethernet-gateway-bot* to implement watchdogs to ensure, that all motors are operating correctly, and that a node sending spatial or motor velocities is still active, as this node must repeat the spatial or motor velocity command once within a defined time period [60].
By default, EduArt Robotik GmbH deploys this node as a Docker container on the embedded Siemens SIMATIC IPC BX-21A computer, which runs Debian Stable and uses CycloneDDS as the communication middleware. However, this setup is considerably less efficient than running the node natively due to the overhead introduced by Docker. Therefore, the following setup was employed: Ubuntu 24.04 was installed on the embedded computer, and ROS2 Jazzy Jalisco was installed natively, while CycloneDDS remains in use as communication middleware. Furthermore, given that EduArt Robotik GmbH only updates the launch and parameter files in the context of the Docker images, a new ROS2 meta-package, designated "edu", was created. This meta-package exclusively bundles launch and parameter files utilized in the context of Telelab and originating from EduArt Robotik GmbH . For the ROS2 node *eduard-ethernet-gateway-bot*, the "edu" meta-package contains the files `robot.py`, `robot.yaml`, and `robot_diagnostic.yaml`, which can be found on the data medium accompanying this thesis, see appendix C.

The second component utilized are kinematic tree models in the Unified Robot Description Format (URDF) of the Eduard robots in both wheel configurations. These models, in conjunction with the joint states published by *eduard-ethernet-gateway-bot*, enable the visualization of both the robot's current state and its sensor values in RVIZ2, the ROS2 3D visualization tool [61], see Figure 3.7. To this end, EduArt Robotik GmbH provides a dedicated launch file, in addition to the URDF models, which is part of the ROS2 package "edu_robot_control" and executes the ROS2 node "robot_state_publisher", which is part of the ROS2 core stack [60]. As with the launch file for *eduard-ethernet-gateway-bot*, this launch file (`robot_description.py`) has been incorporated into the ROS2 meta-package "edu" and can also be found on the data medium accompanying this thesis, see appendix C.

In addition to utilizing the two software components from EduArt Robotik GmbH , two device drivers were developed for the See3CAM_24CUG camera and the LiDAR TiM571, which are outlined in the following sections.

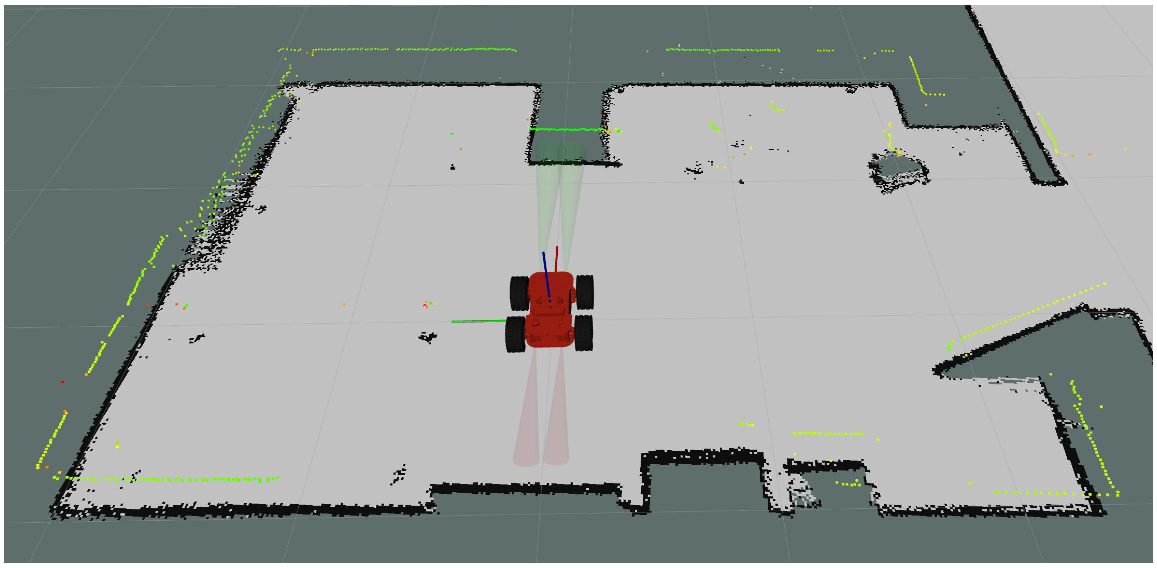


Fig. 3.7: RVIZ2 view of the fixed-wheel URDF model with TOF sensors, laser scan, and base frame in a recorded office map

3.1.2.1 ROS2 device driver for See3CAM_24CUG

Although there are existing ROS2 device drivers for USB UVC cameras that could theoretically be used for the See3CAM_24CUG, a new ROS2 driver for USB UVC cameras was implemented in the context of this work to enable integration of the camera into the Eduard robots. The implementation of this new driver is driven by two primary motivations:

1. All tested existing drives that utilize Video4Linux (V4L) as a backend demonstrated deficiencies in implementing camera configuration, resulting in settings such as exposure time and white balance failing to function properly with the See3Cam_24CUG. This renders these drivers unusable for the intended application in the context of Telelab, as correct configuration of the various camera parameters is essential to ensure that the camera image is consistent in different lightning situations or that images can be captured with constant parameters. Consequently, drivers without the support for the configuration of camera parameters, e.g. due to the utilized backend, are also not a viable option.

2. All tested existing drivers utilize the ROS2 package "image_transport" for the purpose of publishing images. "image_transport" is a plugin-based library that provides mechanisms for efficiently and flexibly publishing and subscribing to camera images via various transmission types, e.g. uncrompressed, compressed, or video-based, without the producing and consuming ROS2 nodes being tied to a specific data format [62]. This means that a producing node, i.e., a camera driver, instantiates an "image_transport" publisher, which is employed to publish raw, uncompressed images [62]. A consuming node can subsequently subscribe to the topics advertised by "image_transport" and, in turn, receive either the raw data or, i.e., the compressed and encoded image [62]. In the

latter case, the respective “image_transport” plugin of the producing node compresses and encodes the raw images prior to their publication [62].

While this approach generally offers many advantages, the use of “image_transport” in the context of Telelab has a major disadvantage, as only compressed images are intended to be transmitted to the teleoperating client due to excessive bandwidth requirements for raw image data. However, when employing “image_transport”, raw image data is first required on the producing node, where it must be compressed prior to publication. This introduces additional computational overhead on the embedded computer of the Eduard robots.

However, the See3CAM_24CUG cameras integrated into the Eduard robots are capable of directly providing compressed and encoded images (MJPEG), as outlined above. These pre-compressed images can be retrieved directly by a device driver and published without any additional processing.

For these two reasons, a new ROS2 device driver, “usb_cam”, was implemented. It is based on the VideoCapture interface from OpenCV (a computer vision library), which utilizes V4L as a backend and enables, on the one hand, full configuration of all parameters of the See3CAM_24CUG, and, on the other hand, direct acquisition of the camera’s pre-compressed MJPEG images which are subsequently published as `sensor_msgs/CompressedImage`.
Compared to the “image_transport” approach, this method significantly reduces the computational load on the embedded computer of the Eduard robots while ensuring efficient image transmission during teleoperation. The source code for this device driver can be found on the data medium accompanying this thesis, see appendix C.

3.1.2.2 ROS2 device driver for TiM571

Despite the availability of an existing device driver for the SICK TiM571, including ROS2 bindings, developed by SICK AG and designated as sick_scan_xd [63], a standalone ROS2 device driver was developed for the integration of the TiM571 in the context of Telelab. Strictly speaking, this developed ROS2 device driver is a port of an existing ROS1 device driver, utilized at the chair of robotics of the University of Würzburg, which was developed under Stefan Stiene and Jan Elseberg primarly for the SICK LMS100. In the course of this work, it was ported to ROS2, incorporating extensions and adaptions that facilitate its compatibility with a wide range of SICK LiDAR scanners.

The porting of this device driver was motivated by the fact that the sick_scan_xd device driver lacks a function essential for operating the TiM571 in the context of Telelab. This is due to the fact that the TiM571 is permanently supplied with power by the electrical installation on the Eduard robot as long as it is switched on via the main switch, regardless of whether the embedded computer is turned on or off. Therefore, it is necessary to explicitly transition the TiM571 to standby mode, in which the motor driving the rotating mirror and the laser diode are deactivated. This standby mode not only reduces the power consumption of the TiM571 but also decreases the operating hours of the motor and laser diode, thereby significantly extending their service life. Nevertheless, the device driver should remain active and maintain the connection to the TiM571 (as long as the embedded computer is switched on) in order to facilitate the driver's ability to reactivate the LiDAR from standby mode, for example, as a result of a ROS2 service call.

However, this essential function is not provided by the sick_scan_xd driver [63]. Additionally, an instance of the sick_scan_xd driver terminates automatically if no measurement data telegrams are transmitted from the LiDAR scanner within a specified period of time [63]. Nevertheless, the sick_scan_xd driver facilitates the transition of the driven SICK LiDAR scanner into the aforementioned standby mode upon termination [63]. Nonetheless, during the course of experimental trials with the sick_scan_xd driver, it was observed that when the LiDAR scanner is to be transitioned into standby mode, segmentation faults occur repeatedly or the driver becomes unresponsive during the process of shutdown.

Consequently, the existing ROS1 SICK LMS100 device driver was ported to ROS2 via the ROS2 package "sick" with the node *sicksensor*, and its functionality was expanded and adapted. This driver establishes a serial connection to the TiM571 via the Ethernet interface, through which it enables the configuration of all measurement parameters, the initiation and termination of measurements, and transitioning the TiM571 into standby mode, using the telegrams defined by SICK AG. To this end, the driver utilizes the ASCII-based communication format (CoLa A), inherited from the original ROS1 driver and, in contrast to the sick_scan_xd driver, which utilizes the binary communication format (CoLa B) [63, 64]. Furthermore, the driver exclusively evaluates responses of the "LMDscandata" telegram type, which contain measurement values of a scan, with respect to their first pulse response, i.e., it processes only DIST1 and RSSI1 [64], which are subsequently published as `sensor_msgs/LaserScan`. This design choice is motivated by the fact that the TiM571 records only a single pulse per singular distance measurement, and the original ROS1 driver likewise evaluated only one pulse response, albeit limited to DIST1. Additionally, the ROS2 driver implements two ROS2 services: one for starting a measurement while concurrently exiting the standby mode of the scanner, and another for stopping a measurement with the option of transitioning the scanner into standby mode.
Compared to the sick_scan_xd approach, this method enabled the TiM571 to be transitioned into and out of standby mode via ROS2 services while keeping the node active. Nevertheless, sick_scan_xd offers an efficiency advantage through its utilization of the binary CoLa B communication format. Nonetheless, the *sicksensor* driver was observed to be able to evaluate the LMS100's "LMDscandata" telegrams, which are not only significantly larger than those of the TiM571 but are also recorded at a higher rate at 50 Hz, utilizing the ASCII-based CoLa A communication format, without any reduction in data rate. The source code for this device driver can be found on the data medium accompanying this thesis, see appendix C.

3.2 Network Integration

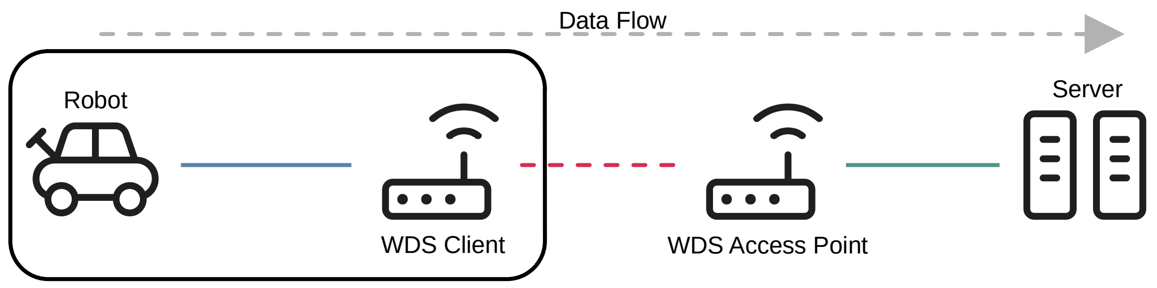
The wireless integration of the Eduard robots into the university network represents a crucial aspect in their overall integration into the Telelab infrastructure. However, this integration is challenging because ROS2 relies on DDS, and consequently on RTPS, as its communication middleware. The difficulty arises, on the one hand, from the use of the Simple Participant Discovery Protocol (SPDP) and Simple Endpoint Discovery Protocol (SEDP) defined by RTPS, which are based on multicast communication that by default does not extend beyond a local subnet (see section 2.2), and, on the other hand, from the absence of an integrated WLAN module in the embedded computer of the Eduard robots in conjunction with the planned network topology, which was already briefly introduced in section 2.4.1 and illustrated in figure 2.23. This network topology establishes that the Eduard robots do not run ROS2 Connect directly, but rather that an additional server executes ROS2 Connect *server* instances, which then tunnel the traffic between the network edge and thus the WebSocket connection to a teleoperating client, and the robots. The reason for this topology is twofold: on the one hand, it reduces the computational load on the embedded computer of the Eduard robots, and on the other hand, it enables the additional server to acquire and tunnel further data from other participants, such as an overhead camera. However, this necessitates the ability of the additional server to receive the DDS, respectively, RTPS messages from the robots and to transmit DDS / RTPS messages back to the robots. Consequently, as SEDP and SPDP rely on multicast, the additional server and the Eduard robots must be located within the same subnet.

However, as mentioned above, the robot's embedded computer (Siemens SIMATIC IPC BX-21A) lacks an integrated WLAN module. Therefore, either a WLAN-to-USB dongle or an on-board router acting as a transparent ISO/OSI Layer 2 bridge must be employed to achieve a wireless integration of the Eduard robots. In the first case, the embedded computer acts as a wireless client itself through the WLAN-to-USB dongle. In the latter case, the router receives the WLAN signal from the university network and forwards it, without the use of routing protocols or Network Address Translation (NAT), to its ethernet interface, and thus to the connected embedded computer of the Eduard robot. Consequently, the router acts as a transparent bridge which, in accordance with the requirements defined by Tanenbaum and Wetherall, does not interfere with the operation of the existing university Local-Area-Network (LAN), rendering the bridged LAN indistinguishable from a single, unbridged one [22]. As a result, multicast and other dynamic control traffic, such as Dynamic Host Configuration Protocol (DHCP), can seamlessly flow between the university network and the Eduard robots.
As indicated in the preceding section, the approach based on an on-board router (GL.iNET GL-MT3000 Beryl AX) was selected for integrating the Eduard robots into the university network. This decision is based on the fact that USB is a host-controlled, polled bus, whereas an ethernet NIC signals the CPU via hardware interrupts when packets are received [65, 66]. Consequently, in the context of a WLAN-to-USB dongle, the embedded computer depends on the USB host controller's polling

schedule for the detection of new data, which introduces additional overhead and reduces the effective data rate [65]. Conversely, when the embedded computer is connected to the network through its ethernet NIC with the on-board router, packets are transferred directly into memory via Direct Memory Access (DMA) and / or the CPU is notified by interrupts [66]. Furthermore, the utilization of an on-board router that functions as a transparent bridge facilitates Wake-on-LAN, thereby enabling the startup of the embedded computer of the Eduard robots via a magic packet.

The final setup therefore consists of a total of three GL.iNET Beryl AX routers, one of which is installed on each of the two Eduard robots, while the third functions as an access point for the two routers installed on the robots. In order to achieve the transparent bridging of the university network, or more precisely the subnet shared by the robots with the additional server, Wireless Distribution System (WDS) was employed. The Wireless Distribution System is a MAC layer mechanism which is originally described in the IEEE 802.11-1999 specification for constructing 802.11 frames using the 4-address format [67, 68]. These frames contain the source address (SA) and destination address (DA), but also the transmitter address (TA) and receiver address (RA) [67]. Therefore, they carry the addresses for the wireless hop between the robot's on-board router (WDS Client) and the WDS access point while retaining the original source and destination MAC addresses of the end hosts (e.g., the Eduard robot and the additional server) [68]. Consequently, the 4-address format enables the implementation of transparent ISO/OSI Layer-2 bridging with WDS extending the networks distribution system while preserving the original frame addressing, rather than performing routing or NAT [68], as illustrated in Figure 3.8. Since a 4-address WDS bridge forms a single Layer-2 domain, multicast frames, DHCP control traffic, and Wake-on-LAN magic packets traverse the bridge unmodified [67, 68].
However, since the IEEE 802.11-1999 specification only describes the 4-address format but not its implementation by a device, the same firmware for a client and access point should be used in a WDS setup [68, 69]. For this reason, the OpenWrt Linux operating system was installed on all three GL.iNET Beryl AX routers. OpenWrt supports the 4-address format, enabling a seamless WDS setup consisting, in this context, of one access point and two clients [69]. The routers were configured to operate according to the IEEE 802.11ax (Wi-Fi 6) standard in the 5 GHz band with a channel bandwidth of 160 MHz. Transparent bridging via the WLAN radio module is restricted to the 2.5 Gbit/s ethernet port while the 1.0 Gbit/s port is reserved for out-of-band management access.

Using this setup, the two Eduard robots were seamlessly integrated into the university network, enabling unhindered traffic flow between the additional server, or other hosts, and the robots. As a result, multicast messages can be transmitted without restriction, enabling RTPS and consequently ROS2 to function wirelessly. Since the robots are equipped with powerful GL.iNET Beryl AX routers, with one additional unit serving as an access point, all of which support the 802.11ax (Wi-Fi 6) standard in the 5 GHz band with a 160 MHz channel bandwidth, the full wireless capacity of the Beryl AX can be utilized. When a single robot is in operation, a connection



3-address format		4-address format		3-addres format	
TA = SA	Robot	SA	Robot	SA	Robot
RA	WDS Client	TA	WDS Client	TA	WDS Access Point
DA	Server	RA	WDS Access Point	RA = DA	Server
		DA	Server		

Fig. 3.8: WDS Layer-2 bridge employing 4-address format (Icons © Google (Apache 2.0) [26])

speed of up to 2402 Mbit/s to the university network can be achieved, whereas simultaneous use of both robots divides the available bandwidth equally. This ensures that the robots network integration does not constitute a bottleneck in the overall communication path between the robots and a teleoperating client.

3.3 Telelab ROS2 Integration

Given that the robots are now equipped with the necessary hardware and software to operate within the context of the Telelab, integrated wirelessly into the university network, respectively, a dedicated subnet, and that ROS2 Connect has been developed to enable their teleoperation over WAN, the remaining step is their integration into the existing Telelab infrastructure. This integration of the two Eduard robots involves two aspects: first, the continuous recording of their current status, and second, the ability to start and stop the robots both manually and automatically according to their booked reservations.

3.3.1 Existing Telelab Infrastructure Extension

In order to achieve this objective, the existing Telelab infrastructure was first extended. That infrastructure currently consists of several web frontends, with the index frontend offering students the opportunity to view and work on tasks, submit inquiries or feedback to administrators via a ticket, and book reservations for a robot assigned to a task. Conversely, administrators have the ability to assign tasks and oversee both users and robots. This index frontend communicates with a PHP

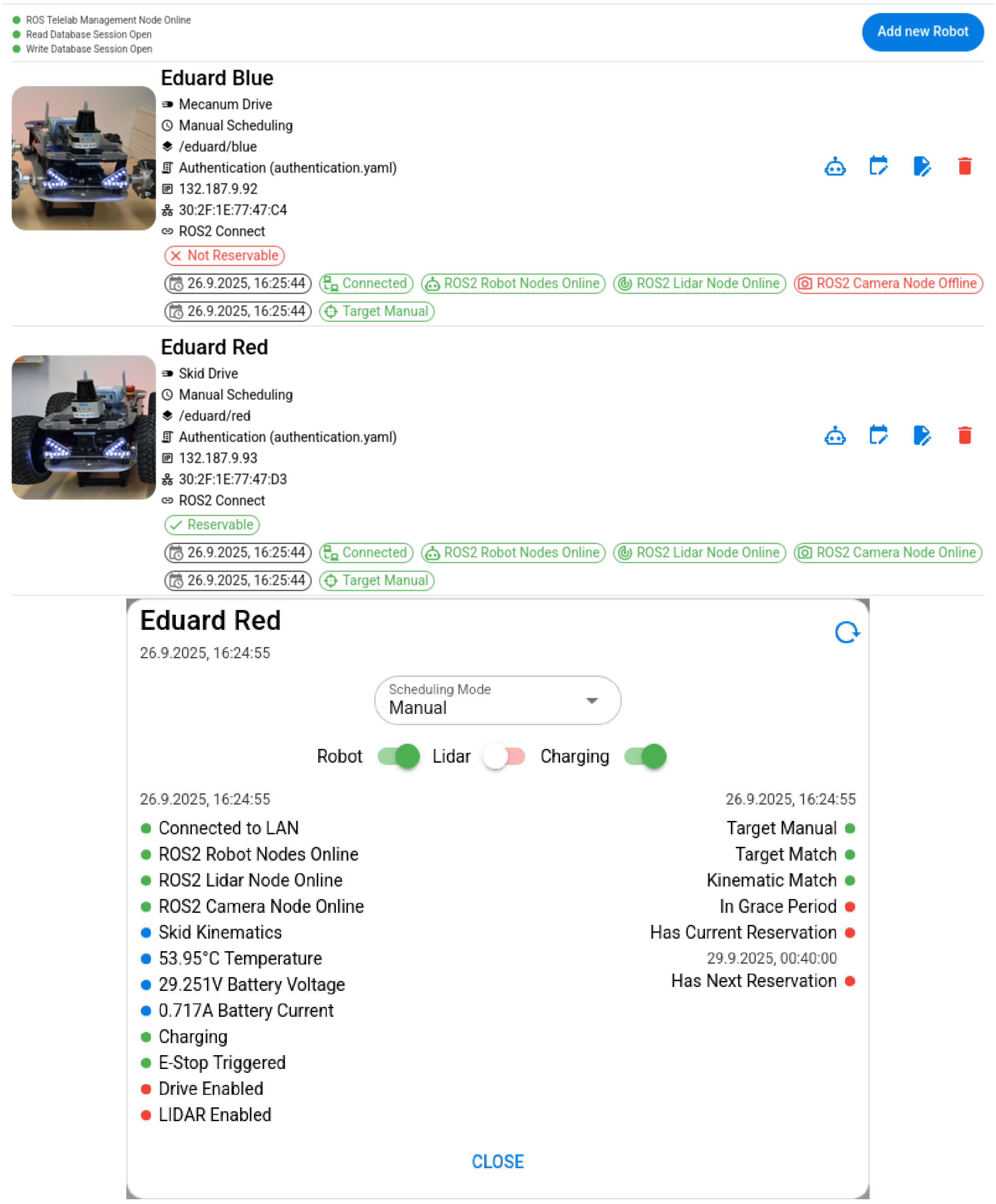
8.2 backend, which in turn utilizes a MariaDB database to facilitate the persistent storage of all data and states.

For the integration of the Eduard robots, this database, or rather its underlying model, was extended, with an excerpt of the extensions of this model shown in Fig. B.17 of appendix B. This extension of the database model encompasses the augmentation of the Robot table with the properties *ip* (IP address), *mac* (MAC address), *mode* (scheduling mode: automatic or manual), *ros_namespace*, and *launch_parameter_template*. The properties *ip*, *mac*, *mode*, and *ros_namspace* are utilized by the ROS2 Telelab implementation (see subsequent section 3.3.3), which is responsible for monitoring the robots and managing their startup and shutdown. Furthermore, the properties *ros_namespace* and *launch_parameter_template* are utilized by ROS2 Connect to authenticate the users and validate their reservation for a robot, whereby the property *launch_parameter_template* functions as a foreign key to an entry in the novelly created table Launch_Parameter_Template. This table houses templates for ROS2 Connect launch parameter files that can contain placeholdes, for instance, the user_key (see section 2.4.1). In addition, two new tables, Robot_Status and Telelab_Robot_Status, were introduced, with each entry designated to a specific robot. The Robot_Status table is designed to store various explicit status values of the robot, including its online status and its current charging level. These values correspond to those reported by the robot itself and are recorded historically. Conversely, Telelab_Robot_Status encompasses a range of status-related information concerning the robot, generated by the aforementioned ROS2 Telelab that monitors the robot's operations. This information, however, is not recorded historically; only the most recent entry is stored.

Consequently, as a result of the expansion of the database model, both the PHP backend and the JavaScript-based web index frontend underwent extension. These extensions facilitate the configuration of the robots according to their new properties, the visualization of the newly recorded robot data (see Fig. 3.9), and the manipulation of their current state, which is achieved through the PHP backend communicating with ROS2 Telelab via XMLHttpRequests.

3.3.2 ROS2 Teleab-Companion and Telelab-Interfaces

In order for the ROS2 Telelab implementation, as referenced above and presented in the following section 3.3.3, to be able to record the status of the Eduard robots and manipulate them in their state, uniform interfaces were developed and defined in the ROS2 package Telelab-Interfaces. These interfaces are implemented by the ROS2 node *eduard* of the Telelab-Companion package, which runs on the Eduard robots and establishes a bridge between the interfaces defined in the "edu_robot" package, developed by EduArt and hosting the *eduard-ethernet-gateway-bot* node, as described in section 3.1.2, and the interfaces provided by the Telelab-Interfaces package.
This approach, i.e., defining and implementing new interfaces, was not absolutely necessary for ROS2 Telelab to communicate with the Eduard robots, since both robots,



● ROS Telelab Management Node Online
● Read Database Session Open
● Write Database Session Open

Add new Robot

Eduard Blue

Mecanum Drive
Manual Scheduling
/eduard/blue
Authentication (authentication.yaml)
132.187.9.92
30:2F:1E:77:47:C4
ROS2 Connect

Not Reservable

26.9.2025, 16:25:44 | Connected | ROS2 Robot Nodes Online | ROS2 Lidar Node Online | ROS2 Camera Node Offline

26.9.2025, 16:25:44 | Target Manual

Eduard Red

Skid Drive
Manual Scheduling
/eduard/red
Authentication (authentication.yaml)
132.187.9.93
30:2F:1E:77:47:D3
ROS2 Connect

✓ Reservable

26.9.2025, 16:25:44 | Connected | ROS2 Robot Nodes Online | ROS2 Lidar Node Online | ROS2 Camera Node Online

26.9.2025, 16:25:44 | Target Manual

Eduard Red

26.9.2025, 16:24:55

Scheduling Mode: Manual

Robot ● Lidar ○ Charging ●

26.9.2025, 16:24:55	26.9.2025, 16:24:55
● Connected to LAN	Target Manual ●
● ROS2 Robot Nodes Online	Target Match ●
● ROS2 Lidar Node Online	Kinematic Match ●
● ROS2 Camera Node Online	In Grace Period ●
● Skid Kinematics	Has Current Reservation ●
● 53.95°C Temperature	29.9.2025, 00:40:00
● 29.251V Battery Voltage	Has Next Reservation ●
● 0.717A Battery Current	
● Charging	
● E-Stop Triggered	
● Drive Enabled	
● LIDAR Enabled	

CLOSE

Fig. 3.9: Visualization of a robot and its state in the Telelab web index frontend

despite their kinematic differences, execute the same code (*eduard-ethernet-gateway-bot*) and therefore already expose uniform communication interfaces. Nevertheless, the definition of new interfaces provides two advantages:

1. The interfaces defined by EduArt present a challenge to humans in terms of interpreting the parameters and results. This complicates the direct evaluation of the robot's status as well as the configuration of the robot to a particular

mode, such as altering the kinematics or activating the motors. The ROS2 Telelab-Interfaces, which have been defined to address these issues, facilitate the manipulation of the robot's state or the interpretation of its status without the necessity of prior knowledge. This is of particular significance in the context of teleoperation, wherein a student, for instance, must initiate motor activation prior to issuing driving commands.

2. The interfaces implemented in "edu_robot" are specific to Eduard robots. However, in the event that the Telelab were to be expanded at a subsequent point in time to incorporate additional robots not originating from EduArt, these would offer entirely distinct interfaces, thereby resulting in incompatibility with a ROS2 Telelab implementation based on interfaces defined by EduArt. The Telelab-Interfaces implemented for each robot type by Telelab-Companion have thus created a robot-independent communication interface that offers straightforward expandability of the Telelab to include any robot in the future.

As a result, the topic types (messages) *RobotStatus*, *TelelabRobotStatus*, and *Telelab-Status* were defined in Telelab-Interfaces, as well as the service definitions *SetMode* and *SetLidar* (see section A.4 of appendix A).
RobotStatus is a topic type published by ROS2 Telelab-Companion on the topic telelab/robot_status. It holds various states of the robot, such as battery voltage, and corresponds in its definition to the database table of the same name, with the exception of two properties. *TelelabRobotStatus* is a topic type utilized by ROS2 Telelab to represent the management status of a robot, with its definition likewise corresponding to the database table of the same name. *TelelabStatus* is a topic type published by ROS2 Telelab on the topic telelab/status. It contains information about the internal state of ROS2 Telelab as well as the management status of all monitored robots, provided in an array of type *TelelabRobotStatus*.
SetMode and *SetLidar*, in turn, represent two service definitions with the former enabling manipulation of the robot's state, such as activating the motors, while the latter allows manipulation of the installed LiDAR, specifically placing the scanner into or out of standby mode. These two services are provided by ROS2 Telelab-Companion under telelab/set_mode and telelab/set_lidar. Furthermore, ROS2 Telelab-Companion provides the two services telelab/shutdown_robot, which schedules a shutdown of the robot within 5 s, and telelab/reset_odometry, which resets the robot's current pose as determined by odometry, both of which make use of the standardized ROS2 service definition *std_srvs/Trigger*.

In summary, several uniform topic types and services were defined using ROS2 Telelab-Interfaces, which provide uniform communication interfaces. Additionally, ROS2 Telelab-Companion guarantees that ROS2 Telelab has explicitly specified topics and services available for the purpose of capturing a robot's status and manipulating its state. These topics and services are implemented by contract, meaning that every node of the Telelab-Companion package is required to implement the following topics and services:

- Topic telelab/robot_status with topic type *telelab_interfaces/RobotStatus*
- Service telelab/shutdown_robot with service type *std_srvs/Trigger*
- Service telelab/reset_odometry with service type *std_srvs/Trigger*
- Service telelab/set_mode with service type *telelab_interfaces/SetMode*
- Service telelab/set_lidar with service type *telelab_interfaces/SetLidar*

3.3.3 ROS2 Telelab

To complete the integration of the two Eduard robots into Telelab, and thus bridge the gap between ROS2 and the existing infrastructure compromising of a database, a PHP backend, and the index web frontend, the aforementioned ROS2 Telelab package was developed, which is executed on the additional server that also executes ROS2 Connect. This package implements, in the node *telelab_node*, both the monitoring of the robots and the targeted manual or automatic manipulation of their state through the communication interfaces provided by ROS2 Telelab-Interfaces and ROS2 Telelab-Companion, with the prerequisite that both robots operate within the same ROS domain, while their topics, services, and tf frames are isolated by their respective ROS namespace, for example, “eduard/blue” and “eduard/red”.

In order to accomplish this objective, ROS2 Telelab initiates two connections (sessions) to the Telelab’s database using the C++ database access library SOCI, which are employed to enable concurrent read and write operations. Consequently, it is capable of retrieving the list of all available robots, including for each robot its ROS namespace, IP and MAC address, as well as the robot’s scheduling mode (automatic or manual), which is implemented by the C++ class `RobotManager`. According to the information obtained, which is periodically retrieved from the database, ROS2 Telelab is now capable of monitoring all robots that contain a non-empty ROS namespace. This is facilitated by the `RobotManager`, which instantiates the C++ class `Robot` for each robot definition, with each instance establishing a subscription to the robot’s status that is consistently published by the respective Telelab-Companion under the designated topic /<ros-namespace>/telelab/robot_status.
With the arrival of each *RobotStatus* message, or at least every 10 s (in the case where the robot is switched off and therefore does not publish its status), the management loop implemented in the `Robot` class is executed. In the event that a *RobotStatus* message has been received, the loop first writes it to the Telelab’s database. In case not, the loop checks whether the robot is connected to the network be sending an Internet Control Message Protocol (ICMP) “Echo Reply” (ping) and subsequently writing a corresponding *RobotStatus* entry to the database.
If the robot is operating in automatic scheduling mode, the database is subsequently queried to retrieve all of its reservations to determine its target state. Accordingly, the robot is required to be switched on if it has a current reservation or if its next reservation will begin within the next minute, while it is required to be switched off it no current reservation exists and none has occurred within the previous minute. Depending on the determined target state, the robot is subsequently either started by sending a Wake-on-LAN magic packet or shut down by invoking its

/<ros-namespace>/telelab/shutdown_robot service, which is provided by its Telelab-Companion instance. Since all ROS2 nodes on the corresponding robot are launched via systemd, the robot publishes its status immediately after connection to the network and can then be configured through its services. This enables the management loop, in its next iteration, to verify, for example, wether the LiDAR scanner is actively measuring and, if not, trigger it via the *telelab_interfaces/SetLidar* service. In the future, the management loop will also be responsible for initiating and terminating the autonomous charging process of the robot.
As the final step of the management loop, which is also executed when the robot operates in manual scheduling mode, a *TelelabRobotStatus* instance is created that is both written to the database and stored in internal memory, which maintains the information and evaluations of the *RobotStatus* collected by the management loop. This enables ROS2 Telelab to report unexpected states or errors, such as the robot being in an incorrect state during automatic scheduling or the robot reporting a kinematic via its *RobotStatus* that does not correspond to the expected, in the database stored, value.

In addition, ROS2 Telelab implements an XMLHttp API based on Boost.Asio, which is part of the Boost library. This API enables both the transmission of signals to the `Robot` instance of a specific robot, for example, to initiate startup or shutdown in the case of manual scheduling, and the transmission of a signal to the `RobotManager`, triggering it to renew its context by retrieving the robot definitions from the database and updating the managed robot instances accordingly. This XMLHttp API is exclusively invoked by the Telelab PHP backend, either as a result of user interactions with the web frontend, e.g., when a user initiates the startup of a manually scheduled robot, or when such interactions modify the data stored in the database that ROS2 Telelab relies on, for instance when reservations for a robot are updated. Accordingly, the API is protected by an IP filter and by a binding to a specific IP address and port combination within an exclusive local subnet that connects the additional server running ROS2 Telelab (as well as ROS2 Connect) with the server hosting the web frontend and PHP backend.

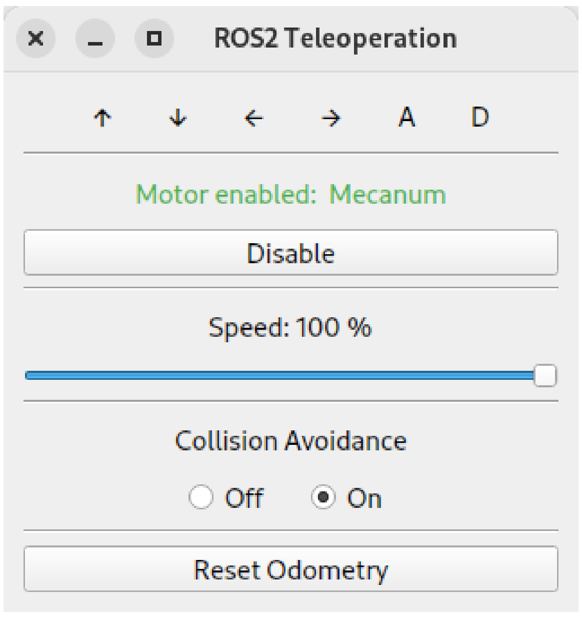
Finally, ROS2 Telelab itself publishes *telelab_interfaces/TelelabStatus* messages on the telelab/status_topic, which serve to indicate, on the one hand, whether the two database connections (sessions) are open and, on the other hand, include an array of *TelelabRobotStatus* instances that provide the collected information and evaluations from each management loop of every `Robot` instance for the monitored robots.

In summary, through the implementation of ROS2 Telelab in combination with ROS2 Telelab-Interfaces and ROS2 Telelab-Companion, a generic mechanism for monitoring and manipulating arbitrary robots, independent of their hardware and software, was realized. This implementation enables the ability to start monitoring a robot simply by registering it in the database via the web frontend, provided that the robot correctly implements ROS2 Telelab-Companion as described in section 3.3.2 and thereby exposes its status and control interfaces by contract within its dedicated ROS namespace. As result, Telelab can be expanded to incorporate new robots with minimal effort, independent of the robot type, embedded hardware, or software stack. Furthermore, given that the topic types (messages) *RobotStatus* and *TelelabRobotStatus* are newly developed and independent definitions, they can be extended at any time with additional properties. This enables new attributes to be displayed to users within the web frontend and, at the same time, to be monitored for anomalies, for example, the frame rate of the camera or the scan rate of the LiDAR to ensure that these devices and their respective drivers are functioning correctly.

3.3.4 ROS2 Teleop

Although ROS2 and EduArt provide a variety of ready-to-use solutions for robot control, a dedicated solution for the teleoperation of robots, the ROS2 Teleop package, was developed in the context of integrating the Eduard robots into the Telelab infrastructure. While existing solutions, such as the RViz Teleop plugin or the ROS2 teleop_twist_keyboard package, can be utilized for the purpose of controlling the Eduard robots, they lack the capacity to configure them in their entirety due to their generic nature.

Consequently, the ROS2 Teleop package was developed, which offers two options to control an Eduard robot during teleoperation. On the on hand, through the *key* node, which is a graphical Qt6 user interface that utilizes mouse interactions and captures the arrow, "A", and "D" keys on the keyboard (see Fig. 3.10), and on the other hand, through the *joy* node, which utilizes a conventional joystick, such as the PlayStation DualSense® Wireless Controller or the Logitech F710. While the *key* node is an independent node, directly publishing velocity commands (`geomentry_msgs/Twist`)



Motor enabled: Mecanum

Speed: 100 %

Collision Avoidance: Off [✓ On]

Fig. 3.10: Graphical Qt6 user interface of ROS2 Teleop

based on the captured keystrokes, the *joy* node, in contrast, necessitates the *joy_linux_node* from the “joy_linux” package (or an equivalent) to obtain information regarding user interactions with the joystick. Furthermore, the *joy* node necessitates an explicit, joystick-specific mapping for assigning the joystick’s axes and buttons to an action to be triggered.
In contrast to the already existing generic robot control solutions, the two newly developed *key* and *joy* nodes, both offer the possibility to enable the motors of the Eduard robots, which is explicitly necessary given that they are monitored by various watchdogs and will be deactivated if a watchdog condition, such as the requirement to receive at least one velocity command every 500 ms, is no longer met. Furthermore, they also offer the possibility to configure collision avoidance, which may be revoked in a future incremental development of ROS2 Teleop. Additionally, the *key* node allows the odometry to be reset via its graphical user interface, while the *joy* node does not implement this functionality to prevent unintended resetting, given its inability to prompt user confirmation, in contrast to the *key* node. These three operations are achieved by the *key* and *joy* nodes through specifically calling the SetMode, respectively, the ResetOdometry service of the Eduard robot, which is defined by the ROS2 Telelab-Interfaces and implemented by the ROS2 Telelab-Companion.
Consequently, if an Eduard robot is to be controlled by one of the existing, generic solutions for robot teleoperation, the SetMode service must be called manually in order to activate the motors. Additionally, it must be ensured that the utilized solution sends or repeats a velocity command at least once within 500 ms to reset the previously mentioned watchdogs.

Chapter 4

Conclusions and Outlook

The objective of this master thesis was to develop a solution that enables ROS2 communication over a Wide-Area-Network, with the intention of deploying this solution in the context of the Telelab e-learning platform. In this way, and through the integration of two new robots, Telelab is thereby enabled to provide not only ROS2-related learning content but also the teleoperation of the newly integrated ROS2-based robots.

In order to achieve this objective, chapter 2 first established a common foundation by presenting the background for the necessity of an explicit solution for ROS2 over WAN, which originates from ROS2's use of DDS, and consequently RTPS, as its communication middleware. Subsequently, potential solutions were discussed that could, in principle, resolve the problem of ROS2 over WAN, among which two, eProsima's DDS Router and Rosbridge, which would require minimal integration effort to implement in the context of Telelab, were evaluated with regard to their suitability. While DDS Router operates at the DDS abstraction layer and relies on TCP/IP connections, performance evaluation using a specifically developed method revealed several significant phenomenon, including a slowdown in transmission time after prolonged operation, which render its utilization as a ROS2 over WAN solution infeasible. In contrast, Rosbridge operates at the ROS2 abstraction layer and utilizes a persistent WebSocket connection, with its performance evaluation indicating a more stable behavior compared to DDS Router. However, due to its utilization of JSON for serializing each ROS2 message prior to transmission, Rosbridge exhibited significantly longer transmission times, which renders its utilization as solution for ROS2 over WAN equally infeasible.
Consequently, the new ROS2 over WAN solution, ROS2 Connect was developed. This solution also utilizes WebSockets but transmits raw binary data and is characterized by its generic implementation, which enables the transmission of arbitrary topics, services, actions, and tf frames, as well as time synchronization, for which it incorporates per-topic or per-service / action compression. Moreover, as a newly developed solution, it achieves optimal integration into the Telelab infrastructure through its connection to the Telelab robot reservation system. In the context of performance evaluations, ROS2 Connect exhibited superior performance in comparison to both DDS Router and Rosbridge, where it achieved the most stable transmission rates while concurrently delivering the lowest average transmission times. Furthermore, it is optimized for the parallel transmission of multiple topics.

In summary, ROS2 Connect was developed as an optimal solution to the problem of ROS2 over WAN, which can be applied far beyond the context of Telelab.

Since ROS2 Connect thus provides a solution for the teleoperation of robots in the context of Telelab, chapter 3 subsequently described the integration of two new Eduard robots from EduArt Robotik GmbH, along with the restructuring of the existing Telelab infrastructure. The integration of the robots initially involved upgrading the robots hardware with an on-board camera facing in the direction of travel and a 2D LiDAR scanner. These additional optical sensors will be utilized for tasks to be solved by students, such as Simultaneous Localization and Mapping (SLAM) or camera calibration, and they enable teleoperation by providing perception of the environment. Furthermore, the integration of the robots into the university LAN was facilitated via the WDS standard, thereby enabling seamless ROS2 communication. Finally, with ROS2 Telelab in combination with ROS2 Telelab-Companion, a generic solution was developed that not only migrates the existing Telelab infrastructure to ROS2 and enables the integration of the two Eduard robots into existing infrastructural components, but also facilitates future extensions with arbitrary mobile robots.

Consequently, the objectives of this work were achieved, enabling the teleoperation of ROS2-based robots over a Wide-Area-Network, while the Telelab infrastructure was restructured to ROS2 and extended by the integration of two new robots. However, several open issues and opportunities for optimization remain, which will need to be addressed in future work.

As already outlined in section 2.4.11, ROS2 Connect demonstrates numerous possibilities for further development. In particular, the complete generic implementation for services and actions should be highlighted again, which is, at the time of writing, not yet possible, as even the latest ROS2 distribution Kilted Kaiju does not implement a generic action server. However, with a future ROS2 release that introduces such support, ROS2 Connect should be further developed to implement completely generic services and actions. Also, further investigations should be conducted to assess the impact of different congestion control algorithms of the Linux kernel, in order to identify a potentially more suitable algorithm than the default TCP CUBIC in the context of utilizing ROS2 Connect. Finally, considerations should be given to extending ROS2 Connect with support for the application of dynamic compression of topics, with the objective of minimizing the overall transmission time of messages of a designated topic.

With regard to the integration of the two Eduard robots, several outstanding issues also remain. Of these, the most significant is the development of an autonomous charging strategy for the two mobile robots. This strategy envisions two permanently installed charging stations within the designated areas traversed by the robots, which the robots can approach and dock to autonomously, by employing the ROS2 Navigation Framework and System `nav2` and utilizing camera-based guidance. This development is essential, as manual monitoring of charge levels and charging of the robots is not feasible. As a result, it is imperative to implement the planning of charging cycles and, subsequently, the planning of available reservation times for the robots. This co-planner can benefit from the historically recorded charging states of the robots, which can be used to develop a machine learning-based approach, that aims to maximize the available reservation times while simultaneously ensures optimal preservation of the batteries installed in the robots. A potential implementation could be based on logic programming, for instance using Prolog, whereby usage patterns of the robots, the battery capacity required per task, and the robot's charging performance are learned in order to satisfy both optimization objectives.

Finally, the curricular development of the Telelab needs to be the addressed, with the objective of creating tutorials that, one the one hand, provide the theoretical foundations of ROS2, on the other hand, introduce ROS2 Connect and its use within the Telelab, and ultimately define the tasks to be completed by the students. To this end, Gazebo simulations of the Telelab environment with the two Eduard robots will also be developed, ensuring that tasks can be offered even beyond the physical availability of the robots.

Nevertheless, despite the remaining aspects for extension and optimization, this work made a substantial contribution to the structural transformation of Telelab through the development of a solution for the teleoperation of ROS2-based robots over a Wide-Area-Network, ROS2 Connect.

Appendix A

Code, Commands and Configuration

A.1 eProsima's DDS Router

A.1.1 Execute DDS Router

```
DDS-Router
├── build
├── install
│   ├── ddsrouter_tool
│   │   └── bin
│   │       └── ddsrouter
│   └── setup.sh
├── log
├── src
├── client.yaml
├── ddsrouter.repos
└── repeater.yaml
```

Algorithm A.1: Execute DDS Router

```
cd /path/to/DDS-Router

source ./install/setup.sh

# Client / Host A and B
./install/ddsrouter_tool/bin/ddsrouter -c client.yaml

# TURN Repeater / Host C
./install/ddsrouter_tool/bin/ddsrouter -c repeater.yaml
```

A.1.2 Client / Host A and B Configuration

Algorithm A.2: client.yaml

```yaml
version: v5.0

participants:

  - name: DDS_CLIENT
    kind: local
    domain: <local domainId>

  - name: Router_Client
    kind: wan
    connection-addresses:
      - ip: <public ip of host C>
        port: <public port of host C>
        transport: tcp
```

A.1.3 TURN Repeater / Host C Configuration

Algorithm A.3: repeater.yaml

```yaml
version: v5.0

participants:

  - name: DDS_REPEATER
    kind: local

  - name: Router_Repeater
    kind: wan
    repeater: true
    listening-addresses:
      - ip: <public ip of host C>
        port: <public port of host C>
        external-port: <public port of host C>
        transport: tcp
```

A.2 iperf3

Algorithm A.4: iperf3 bandwidth measurement

```bash
# To be executed on host C
iperf3 -s -p 5100

# To be executed on host B
# Measurement of bandwidth from host B to host C
iperf3 -c <ip of host C> -p <port of host C> -t 60 -N
# Measurement of bandwidth from host C to host B
iperf3 -c <ip of host C> -p <port of host C> -t 60 -N -R
```

A.3 ROS2 Connect

A.3.1 Connect *client* launch file

Algorithm A.5: client.py

```python
from launch import LaunchDescription
from launch_ros.actions import Node
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration,
    PathJoinSubstitution
from launch_ros.substitutions import FindPackageShare

def generate_launch_description():
  # Declare a launch argument for the parameter file
  param_file_arg = DeclareLaunchArgument(
    'params_file',
    default_value=PathJoinSubstitution([FindPackageShare('connect'),
      'launch','client.yaml'])
  )

  # Node definition, loading parameters from the specified file
  client = Node(
    package='connect',
    executable='client',
    name='client',
    parameters=[LaunchConfiguration('params_file')]
  )

  return LaunchDescription([
    param_file_arg,
    client
  ])
```

A.3.2 Connect *client* parameter file

Algorithm A.6: client.yaml

```yaml
client:
  ros__parameters:
    # host, path, and port of server instance to connect to
    # if SSL = true, transport encryption will be applied
    host: 'telelab.informatik.uni-wuerzburg.de'
    path: '/ws/ros'
    port: '443'
    ssl: true

    # create a pre-defined ThreadedPublisher which publishes the ros
      -time over topic /clock
    # this allows ROS2 time synchronization between the 'server' and
       'client'
    # this will only work, if the 'server' has the parameter clock/
      subscribe = true
    clock/publish: true

    # create a pre-define ThreadedPublisher which publishes tf2
       transformations over topic /tf and /tf_static
    # this will allow tf2 transformations forwarding from 'server'
       to 'client'
    # this will only work, if the 'server' has the parameter tf/
       subscribe = true
    tf/publish: true

    # enable / disable per-message fragmentation during websocket
       transport
    # using the given fragmentation size if enabled
    fragmentation/enable: false
    fragmentation/size: 4096

    # the maximum size of a message which is accepted to receive
    # 16 * 1024B * 1024B
    max_message_size: 16777216

    # unique and user-specific user key for authentication
    user_key: 'secret'

    # quality of service definitions
    # the policies history, depth, reliability and durability are
       mandatory
    # the policies deadline, lifespan, liveliness and
       liveliness_lease_duration are optional and set to default if
       missing
    #
    # id:                          unique id ... to be used for
       subscriber, publisher, service & action server & clients ...
       range from 0 to 255
    #
    # history:
       RMW_QOS_POLICY_HISTORY_SYSTEM_DEFAULT,
       RMW_QOS_POLICY_HISTORY_KEEP_LAST,
       RMW_QOS_POLICY_HISTORY_KEEP_ALL
    # depth:                       * a positive integer *
    # reliability:
       RMW_QOS_POLICY_RELIABILITY_SYSTEM_DEFAULT,
       RMW_QOS_POLICY_RELIABILITY_RELIABLE,
       RMW_QOS_POLICY_RELIABILITY_BEST_EFFORT
    # durability:
       RMW_QOS_POLICY_DURABILITY_SYSTEM_DEFAULT,
       RMW_QOS_POLICY_DURABILITY_TRANSIENT_LOCAL,
       RMW_QOS_POLICY_DURABILITY_VOLATILE
    #
    # deadline:                    RMW_QOS_DEADLINE_DEFAULT,
       RMW_QOS_DEADLINE_BEST_AVAILABLE, *{seconds, nanoseconds}* e.g
       . *{123, 456000000}*
    # lifespan:                    RMW_QOS_LIFESPAN_DEFAULT, *{seconds
       , nanoseconds}* e.g. *{123, 456000000}*
    # liveliness:
       RMW_QOS_POLICY_LIVELINESS_SYSTEM_DEFAULT,
RMW_QOS_POLICY_LIVELINESS_AUTOMATIC, RMW_QOS_POLICY_LIVELINESS_MANUAL_BY_TOPIC, RMW_QOS_POLICY_LIVELINESS_BEST_AVAILABLE
# liveliness_lease_duration: RMW_QOS_LIVELINESS_LEASE_DURATION_DEFAULT, RMW_QOS_LIVELINESS_LEASE_DURATION_BEST_AVAILABLE, *{seconds, nanoseconds}* e.g. *{123, 456000000}*
#
# 1: default profile, 2: action-status profile
qos:
  - '{"id": 1, "history": "RMW_QOS_POLICY_HISTORY_KEEP_LAST", "depth": 10, "reliability": "RMW_QOS_POLICY_RELIABILITY_RELIABLE", "durability": "RMW_QOS_POLICY_DURABILITY_VOLATILE"}'
  - '{"id": 2, "history": "RMW_QOS_POLICY_HISTORY_KEEP_LAST", "depth": 1, "reliability": "RMW_QOS_POLICY_RELIABILITY_RELIABLE", "durability": "RMW_QOS_POLICY_DURABILITY_TRANSIENT_LOCAL"}'

# compression profile definitions with default compression rate value
#
# id:          unique id ... to be used for subscriber, publisher, service & action server & clients ... range from 0 to 255
#
# NONE:        no compression
# LZ4_DEFAULT: very fast but low compression
# LZ4_HC:      faster than ZLIB for low compression rates, slower than ZLIB for high compression rates, always larger compressed size than ZLIB
# ZLIB:        lowest compressed size overall
#
# NONE:        rate is ignored
# LZ4_DEFAULT: rate is understood as accelerator, the bigger the faster ... range from 1 to LZ4_ACCELERATION_MAX (65537) .. default 1
# LZ4_HC:      rate is understood as rate, the bigger the slower ... range from 1 to LZ4HC_CLEVEL_MAX (12) ... default 9
# ZLIB:        rate is understood as rate, the bigger the slower ... range from 1 to Z_BEST_COMPRESSION (9) ... default 6
compression:
  - '{"id": 1, "compressor": "NONE",        "rate": 0}'
  - '{"id": 2, "compressor": "LZ4_DEFAULT", "rate": 1}'
  - '{"id": 3, "compressor": "LZ4_HC",      "rate": 9}'
  - '{"id": 4, "compressor": "ZLIB",        "rate": 6}'

# topics to which the 'client' subscribes locally
# these topics can be requested by the 'server'
#
# channel:      channel ... must be unique over all subscriber and publisher ... range from 0 to 249
# topic:        topic name
# type:         topic type definition
```

```
  # useOwnThread: if true, an own processing thread plus executor thread is used
  # permanent:    if true, subscriber won't be destroyed when other side stopped requesting for the data ... necessary for RMW_QOS_POLICY_DURABILITY_TRANSIENT_LOCAL
  # qos:          id of the quality of service profile to apply
  # compression:  id of the compression profile to apply ... needs useOwnThread = true
  subscriber:
    - '{"channel": 1, "topic": "/primary", "type": "time_measurement/msg/TimeMeasurement", "qos": 1, "compression": 1, "useOwnThread": false, "permanent": false}'

  # topics which are published by the 'client' locally
  # messages from these topics can be send by the 'server'
  #
  # channel:      channel ... must be unique over all subscriber and publisher ... range from 0 to 249
  # topic:        topic name
  # type:         topic type definition
  # useOwnThread: if true, an own processing thread
  # eager:        if true, data is published even when there is no local subscriber ... necessary for RMW_QOS_POLICY_DURABILITY_TRANSIENT_LOCAL
  # qos:          id of the quality of service profile to apply
  # compression:  id of the compression profile to apply ... needs useOwnThread = true
  publisher:
    - '{"channel": 2, "topic": "/secondary", "type": "time_measurement/msg/TimeMeasurement", "qos": 1, "useOwnThread": false, "eager": false}'

  # service server which are run by the 'client'
  # these service server mirror actual service server which are run on the 'server'
  #
  # channel:           channel ... must be unique over all service server & client ... range from 0 - 255
  # type:              service server type as defined in plugins.xml (pluginlib)
  # useOwnThread:      if true, an own executor thread is used
  # allowSimultaneous: if simultaneous calls to the service server should be processed simultaneously or sequentially ... needs useOwnThread = true
  # service-qos:       id of the quality of service profile to apply
  # maxExecTime:       max execution time in seconds after which a call is interrupted and stopped
  # compression:       id of the compression profile to apply (also possible for shared thread, but not recommended)
  service/server:
    - '{"channel": 1, "type": "connect_plugins::AddTwoIntsServiceServer", "service-qos": 1, "useOwnThread": false, "allowSimultaneous": false, "maxExecTime": 10, "
```

```
        compression": 1}'

    # action server which are run by the 'client'
    # these action server mirror actual action server which are run
      on the 'server'
    #
    # channel:            channel ... must be unique over all action
      server & client ... range from 0 - 255
    # type:               action server type as defined in plugins.
      xml (pluginlib)
    # useOwnThread:       if true, an own executor thread is used
    # allowSimultaneous: if simultaneous calls to the action server
      should be processed simultaneously or sequentially ... needs
      useOwnThread = true
    # service-qos:        id of the quality of service profile to
      apply for goal handling
    # feedback-qos:       id of the quality of service profile to
      apply for publishing feedback message
    # status-qos:         id of the quality of service profile to
      apply for broadcasting goal statuses
    # result-timeout:     timeout in seconds after which the goal
      result will no longer be available; -1: indefinitely, 0:
      immediately
    # compression:        id of the compression profile to apply (
      also possible for shared thread, but not recommended)
    action/server:
      - '{"channel": 1, "type": "connect_plugins::
        FibonacciActionServer", "service-qos": 1, "feedback-qos":
        1, "status-qos": 2, "useOwnThread": true, "
        allowSimultaneous": false, "resultTimeout": 10, "
        compression": 1}'
```

A.3.3 Connect *server* launch file

Algorithm A.7: server.py

```
from launch import LaunchDescription
from launch_ros.actions import Node
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration,
   PathJoinSubstitution
from launch_ros.substitutions import FindPackageShare

def generate_launch_description():
  # Declare a launch argument for the parameter file
  param_file_arg = DeclareLaunchArgument(
    'params_file',
    default_value=PathJoinSubstitution([FindPackageShare('connect'),
       'launch','server.yaml'])
  )

  # Node definition, loading parameters from the specified file
  server = Node(
    package='connect',
    executable='server',
    name='server',
    parameters=[LaunchConfiguration('params_file')]
  )

  return LaunchDescription([
    param_file_arg,
    server
  ])
```

A.3.4 Connect *server* parameter file

Algorithm A.8: server.yaml

```
server:
  ros__parameters:
    # host and port to which the websocket server should bind to
    host: '0.0.0.0'
    port: '9090'

    # host and port of the Telelabs PHP server
    # if SSL = true, transport encryption will be applied
    php/host: 'telelab.informatik.uni-wuerzburg.de'
    php/port: '443'
    php/ssl: true

    # the Telelabs PHP server endpoint to perform a user_key /
       reservation check against
    reservation_check/endpoint: '/apiv2/index.php/reservation/get/
       user-key'
    # if the user_key / reservation check should be omitted
    reservation_check/omit: false
    # timeout after which the 'server' closes the connection if no
    # user_key was send by the 'client'
    reservation_check/timeout: 10

    # create a pre-defined ClockSubscriber which sends the ros-time
       over topic /clock
    # this will allow ROS2 time synchronization between the 'server'
        and 'client'
    clock/subscribe: true

    # create a pre-define ThreadedSubscriber which subscribes to tf2
        transformations from topic /tf and /tf_static
    # this will allow tf2 transformations forwarding from 'server'
       to 'client'
    # this will only work, if the 'client' has the parameter tf/
       publish = true
    tf/subscribe: true

    # enable / disable per-message fragmentation during websocket
       transport
    # using the given fragmentation size if enabled
    fragmentation/enable: false
    fragmentation/size: 4096

    # the maximum size of a message which is accepted to receive
    # 16 * 1024B * 1024B
    max_message_size: 16777216

    # quality of service definitions
    # the policies history, depth, reliability and durability are mandatory
    # the policies deadline, lifespan, liveliness and liveliness_lease_duration are optional and set to default if missing
    #
    # id:                        unique id ... to be used for subscriber, publisher, service & action server & clients ... range from 0 to 255
    #
    # history:                   RMW_QOS_POLICY_HISTORY_SYSTEM_DEFAULT, RMW_QOS_POLICY_HISTORY_KEEP_LAST, RMW_QOS_POLICY_HISTORY_KEEP_ALL
    # depth:                     * a positive integer *
    # reliability:               RMW_QOS_POLICY_RELIABILITY_SYSTEM_DEFAULT, RMW_QOS_POLICY_RELIABILITY_RELIABLE, RMW_QOS_POLICY_RELIABILITY_BEST_EFFORT
    # durability:                RMW_QOS_POLICY_DURABILITY_SYSTEM_DEFAULT, RMW_QOS_POLICY_DURABILITY_TRANSIENT_LOCAL, RMW_QOS_POLICY_DURABILITY_VOLATILE
    #
    # deadline:                  RMW_QOS_DEADLINE_DEFAULT, RMW_QOS_DEADLINE_BEST_AVAILABLE, *{seconds, nanoseconds}* e.g. *{123, 456000000}*
    # lifespan:                  RMW_QOS_LIFESPAN_DEFAULT, *{seconds, nanoseconds}* e.g. *{123, 456000000}*
    # liveliness:                RMW_QOS_POLICY_LIVELINESS_SYSTEM_DEFAULT, RMW_QOS_POLICY_LIVELINESS_AUTOMATIC, RMW_QOS_POLICY_LIVELINESS_MANUAL_BY_TOPIC, RMW_QOS_POLICY_LIVELINESS_BEST_AVAILABLE
    # liveliness_lease_duration: RMW_QOS_LIVELINESS_LEASE_DURATION_DEFAULT, RMW_QOS_LIVELINESS_LEASE_DURATION_BEST_AVAILABLE, *{seconds, nanoseconds}* e.g. *{123, 456000000}*
    #
    # 1: default profile, 2: action-status profile
    qos:
      - '{"id": 1, "history": "RMW_QOS_POLICY_HISTORY_KEEP_LAST", "depth": 10, "reliability": "RMW_QOS_POLICY_RELIABILITY_RELIABLE", "durability": "RMW_QOS_POLICY_DURABILITY_VOLATILE"}'
      - '{"id": 2, "history": "RMW_QOS_POLICY_HISTORY_KEEP_LAST", "depth": 1, "reliability": "RMW_QOS_POLICY_RELIABILITY_RELIABLE", "durability": "RMW_QOS_POLICY_DURABILITY_TRANSIENT_LOCAL"}'

    # compression profile definitions with default compression rate value
    #
    # id:          unique id ... to be used for subscriber, publisher, service & action server & clients ... range from 0 to 255
    #
    # NONE:        no compression
    # LZ4_DEFAULT: very fast but low compression
    # LZ4_HC:      faster than ZLIB for low compression rates, slower than ZLIB for high compression rates, always larger compressed size than ZLIB
    # ZLIB:        lowest compressed size overall
    #
    # NONE:        rate is ignored
    # LZ4_DEFAULT: rate is understood as accelerator, the bigger the fast ... range from 1 - LZ4_ACCELERATION_MAX (65537) .. default 1
    # LZ4_HC:      rate is understood as rate, the bigger the slower ... range from 1 - LZ4HC_CLEVEL_MAX (12) ... default 9
    # ZLIB:        rate is understood as rate, the bigger the slower ... range from 1 - Z_BEST_COMPRESSION (9) ... default 6
    compression:
      - '{"id": 1, "compressor": "NONE",        "rate": 0}'
      - '{"id": 2, "compressor": "LZ4_DEFAULT", "rate": 1}'
      - '{"id": 3, "compressor": "LZ4_HC",      "rate": 9}'
      - '{"id": 4, "compressor": "ZLIB",        "rate": 6}'

    # topics which are published by the 'client' locally
    # messages from these topics can be send by the 'server'
    #
    # channel:      channel ... must be unique over all subscriber and publisher ... range from 0 to 249
    # topic:        topic name
    # type:         topic type definition
    # useOwnThread: if true, an own processing thread
    # eager:        if true, data is published even when there is no local subscriber ... necessary for RMW_QOS_POLICY_DURABILITY_TRANSIENT_LOCAL
    # qos:          id of the quality of service profile to apply
    # compression:  id of the compression profile to apply ... needs useOwnThread = true
    publisher:
      - '{"channel": 1, "topic": "/primary", "type": "time_measurement/msg/TimeMeasurement", "qos": 1, "useOwnThread": false, "eager": false}'

    # topics to which the 'client' subscribes locally
    # these topics can be requested by the 'server'
    #
```

```
# channel:      channel ... must be unique over all subscriber and publisher ... range from 0 to 249
# topic:        topic name
# type:         topic type definition
# useOwnThread: if true, an own processing thread plus executor thread is used
# permanent:    if true, subscriber won't be destroyed when other side stopped requesting for the data ... necessary for RMW_QOS_POLICY_DURABILITY_TRANSIENT_LOCAL
# qos:          id of the quality of service profile to apply
# compression:  id of the compression profile to apply ... needs useOwnThread = true
subscriber:
  - '{"channel": 2, "topic": "/secondary", "type": "time_measurement/msg/TimeMeasurement", "qos": 1, "compression": 1, "useOwnThread": false, "permanent": false}'

# service client which are run by the 'server'
# these service client re-call the service call of the 'client' to an actual service server which are run on the 'server'
#
# channel:           channel ... must be unique over all service server & client ... range from 0 - 255
# type:              service client type as defined in plugins.xml (pluginlib)
# useOwnThread:      if true, an own executor thread is used
# allowSimultaneous: if simultaneous calls by the service client should be processed simultaneously or sequentially ... needs useOwnThread = true
# service-qos:       id of the quality of service profile to apply
# compression:       id of the compression profile to apply (also possible for shared thread, but not recommended)
service/client:
  - '{"channel": 1, "type": "connect_plugins::AddTwoIntsServiceClient", "service-qos": 1, "useOwnThread": false, "allowSimultaneous": false, "compression": 1}'

# action client which are run by the 'server'
# these action client re-call the action call of the 'client' to an actual action server which are run on the 'server'
#
# channel:           channel ... must be unique over all action server & client ... range from 0 - 255
# type:              action client type as defined in plugins.xml (pluginlib)
# useOwnThread:      if true, an own executor thread is used
# allowSimultaneous: if simultaneous calls by the action server should be processed simultaneously or sequentially ... needs useOwnThread = true
# service-qos:       id of the quality of service profile to apply for goal handling
# feedback-qos:      id of the quality of service profile to apply for publishing feedback message
```

```
    # status-qos:         id of the quality of service profile to
      apply for broadcasting goal statuses
    # compression:        id of the compression profile to apply (
      also possible for shared thread, but not recommended)
    action/client:
      - '{"channel": 1, "type": "connect_plugins::
        FibonacciActionClient", "service-qos": 1, "feedback-qos":
        1, "status-qos": 2, "useOwnThread": true, "
        allowSimultaneous": false, "compression": 1}'
```

A.4 ROS2 Telelab-Interfaces

A.4.1 Topic Type (Message) Definitions

Algorithm A.9: RobotStatus.msg

```
std_msgs/Header header

# these values are part of the database model
bool online_lidar
bool online_camera
float64 temperature
float64 battery_voltage
float64 battery_current
bool charging
bool drive_enabled
bool lidar_enabled
bool e_stop
uint8 kinematic

# these values are not part of the database model but useful
bool collision_avoidance
bool autonomous

# CONSTANTS: kinematic
uint8 KINEMATIC_SKID = 0
uint8 KINEMATIC_MECANUM = 1
uint8 KINEMATIC_DIFFERENTIAL = 2
uint8 KINEMATIC_ACKERMANN = 3
```

Algorithm A.10: TelelabRobotStatus.msg

```
int32 robot_id
string robot_name
uint8 target_state
bool is_target_state
bool kinematic_match
bool has_current_reservation
bool has_next_reservation
bool is_in_grace
builtin_interfaces/Time current_reservation_end
builtin_interfaces/Time next_reservation_start
```

```

# CONSTANTS: TARGET_STATE
uint8 TARGET_STATE_OFF = 0
uint8 TARGET_STATE_ON = 1
uint8 TARGET_STATE_MANUAL = 2
```

Algorithm A.11: TelelabStatus.msg

```
std_msgs/Header header

bool read_database_session_open
bool write_database_session_open
TelelabRobotStatus[] robots
```

A.4.2 Service Definitions

Algorithm A.12: SetMode.srv

```
bool enable
bool autonomous
bool collision_avoidance
---
bool success_enable
bool success_collision_avoidance
```

Algorithm A.13: SetLidar.srv

```
bool enable
---
bool success
```

Algorithm A.14: std_srvs/Trigger.srv

```
---
bool success
string message
```

Appendix B

Additional Images

This appendix contains additional images, which have been rendered at their largest possible size. However, for certain images, the page dimensions may not be sufficient to guarantee optimal legibility of text. Each image displayed here is part of the data medium and can therefore be viewed in full resolution.

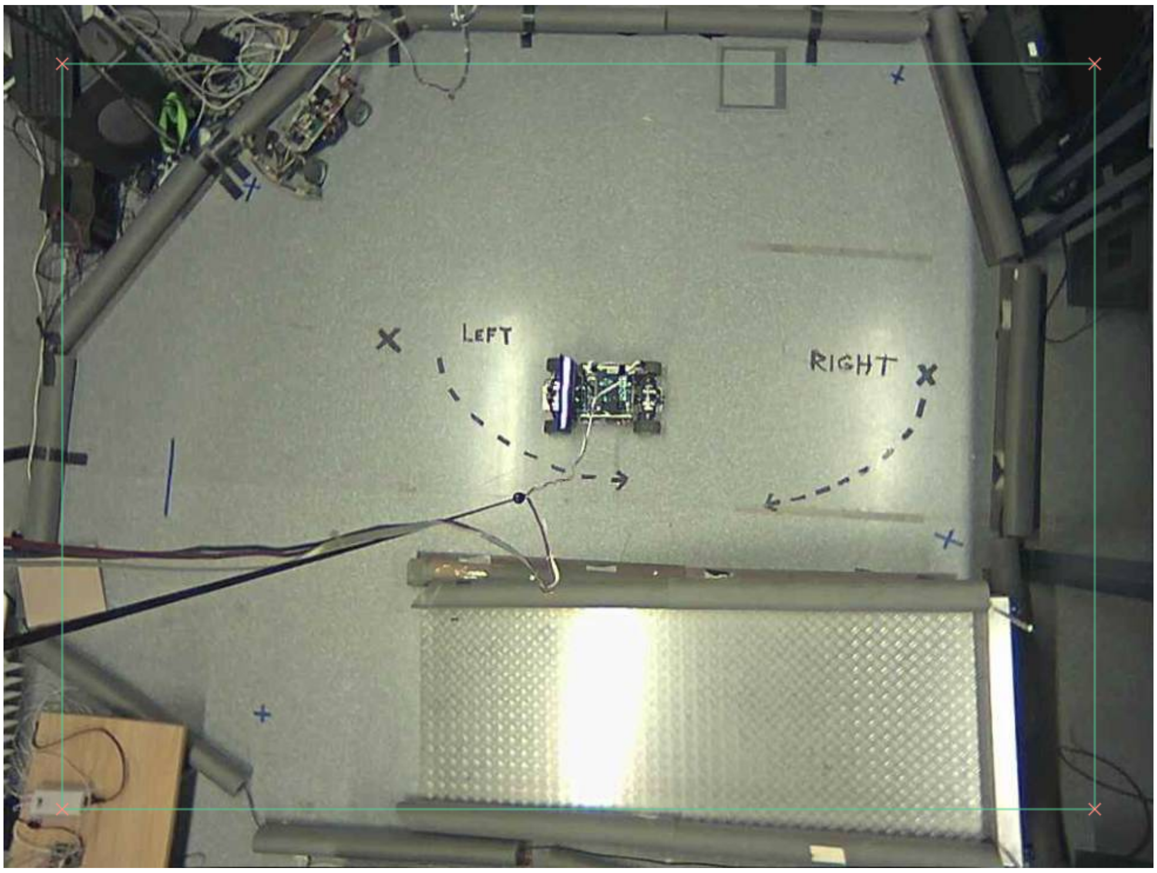


Fig. B.1: Overhead camera image used for compression evaluation

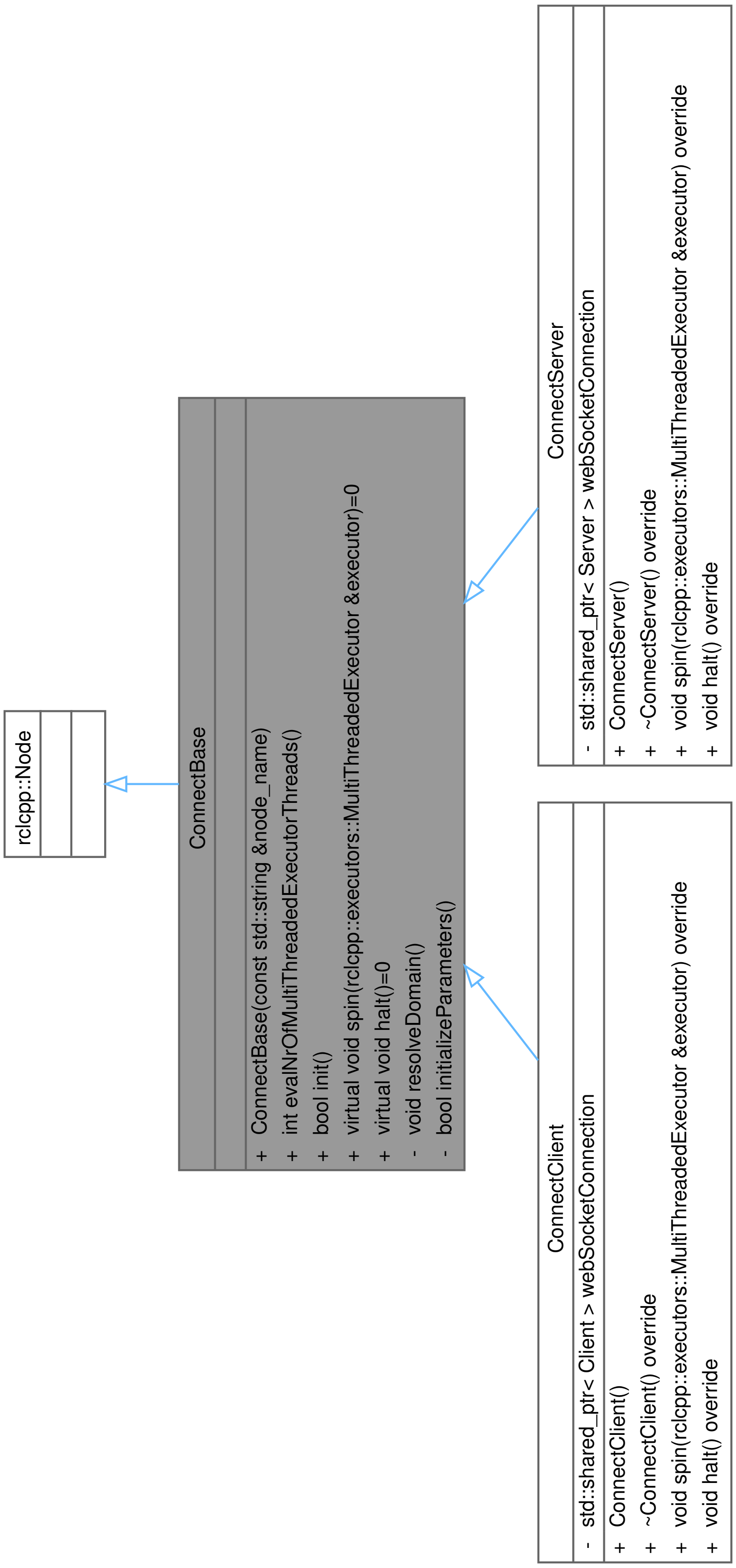
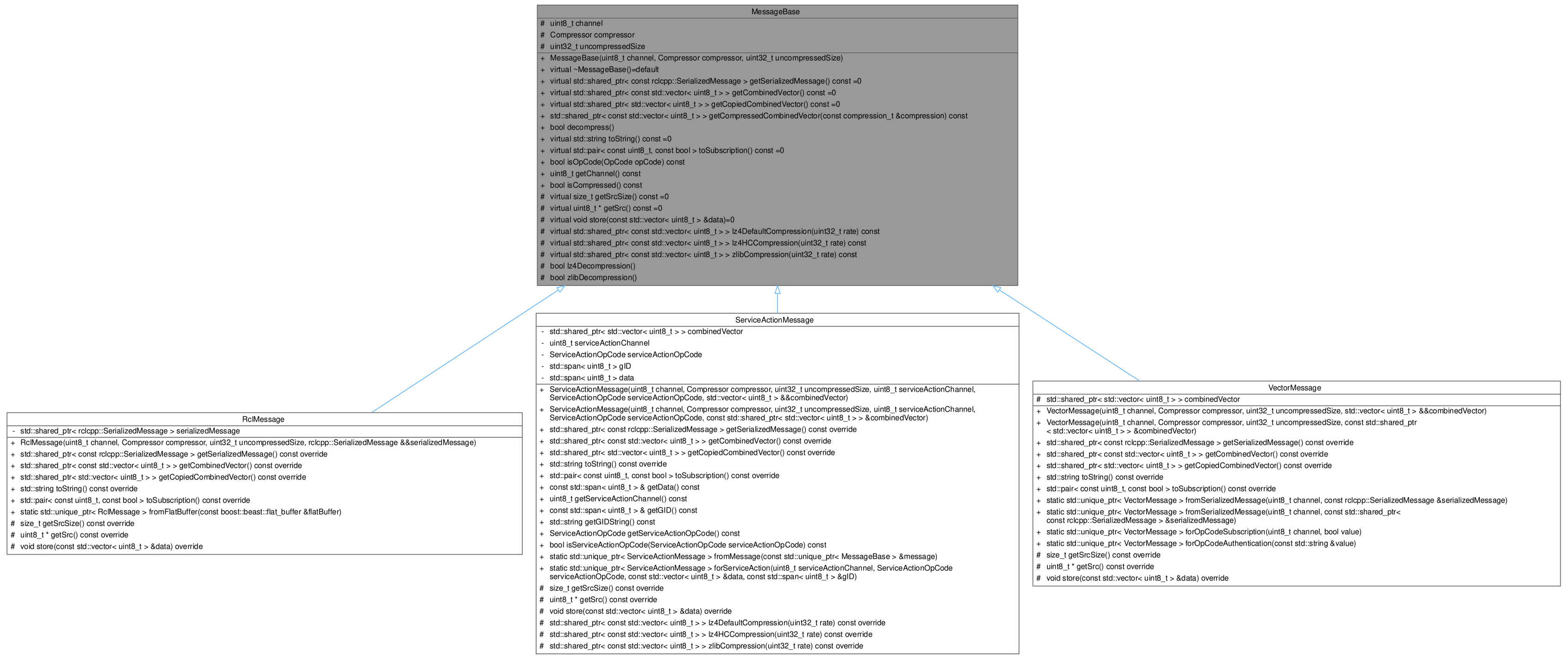

rclcpp::Node
ConnectBase
+ ConnectBase(const std::string &node_name)
+ int evalNrOfMultiThreadedExecutorThreads()
+ bool init()
+ virtual void spin(rclcpp::executors::MultiThreadedExecutor &executor)=0
+ virtual void halt()=0
- void resolveDomain()
- bool initializeParameters()
ConnectServer
- std::shared_ptr< Server > webSocketConnection
+ ConnectServer()
+ ~ConnectServer() override
+ void spin(rclcpp::executors::MultiThreadedExecutor &executor) override
+ void halt() override
ConnectClient
- std::shared_ptr< Client > webSocketConnection
+ ConnectClient()
+ ~ConnectClient() override
+ void spin(rclcpp::executors::MultiThreadedExecutor &executor) override
+ void halt() override

Fig. B.2: Inheritance graph of `ConnectBase`



```
MessageBase
#  uint8_t channel
#  Compressor compressor
#  uint32_t uncompressedSize
+  MessageBase(uint8_t channel, Compressor compressor, uint32_t uncompressedSize)
+  virtual ~MessageBase()=default
+  virtual std::shared_ptr< const rclcpp::SerializedMessage > getSerializedMessage() const =0
+  virtual std::shared_ptr< const std::vector< uint8_t > > getCombinedVector() const =0
+  virtual std::shared_ptr< std::vector< uint8_t > > getCopiedCombinedVector() const =0
+  std::shared_ptr< const std::vector< uint8_t > > getCompressedCombinedVector(const compression_t &compression) const
+  bool decompress()
+  virtual std::string toString() const =0
+  virtual std::pair< const uint8_t, const bool > toSubscription() const =0
+  bool isOpCode(OpCode opCode) const
+  uint8_t getChannel() const
+  bool isCompressed() const
#  virtual size_t getSrcSize() const =0
#  virtual uint8_t * getSrc() const =0
#  virtual void store(const std::vector< uint8_t > &data)=0
#  virtual std::shared_ptr< const std::vector< uint8_t > > lz4DefaultCompression(uint32_t rate) const
#  virtual std::shared_ptr< const std::vector< uint8_t > > lz4HCCompression(uint32_t rate) const
#  virtual std::shared_ptr< const std::vector< uint8_t > > zlibCompression(uint32_t rate) const
#  bool lz4Decompression()
#  bool zlibDecompression()
```

```
RclMessage
-  std::shared_ptr< rclcpp::SerializedMessage > serializedMessage
+  RclMessage(uint8_t channel, Compressor compressor, uint32_t uncompressedSize, rclcpp::SerializedMessage &&serializedMessage)
+  std::shared_ptr< const rclcpp::SerializedMessage > getSerializedMessage() const override
+  std::shared_ptr< const std::vector< uint8_t > > getCombinedVector() const override
+  std::shared_ptr< std::vector< uint8_t > > getCopiedCombinedVector() const override
+  std::string toString() const override
+  std::pair< const uint8_t, const bool > toSubscription() const override
+  static std::unique_ptr< RclMessage > fromFlatBuffer(const boost::beast::flat_buffer &flatBuffer)
#  size_t getSrcSize() const override
#  uint8_t * getSrc() const override
#  void store(const std::vector< uint8_t > &data) override
```

```
ServiceActionMessage
-  std::shared_ptr< std::vector< uint8_t > > combinedVector
-  uint8_t serviceActionChannel
-  ServiceActionOpCode serviceActionOpCode
-  std::span< uint8_t > gID
-  std::span< uint8_t > data
+  ServiceActionMessage(uint8_t channel, Compressor compressor, uint32_t uncompressedSize, uint8_t serviceActionChannel,
   ServiceActionOpCode serviceActionOpCode, std::vector< uint8_t > &&combinedVector)
+  ServiceActionMessage(uint8_t channel, Compressor compressor, uint32_t uncompressedSize, uint8_t serviceActionChannel,
   ServiceActionOpCode serviceActionOpCode, const std::shared_ptr< std::vector< uint8_t > > &combinedVector)
+  std::shared_ptr< const rclcpp::SerializedMessage > getSerializedMessage() const override
+  std::shared_ptr< const std::vector< uint8_t > > getCombinedVector() const override
+  std::shared_ptr< std::vector< uint8_t > > getCopiedCombinedVector() const override
+  std::string toString() const override
+  std::pair< const uint8_t, const bool > toSubscription() const override
+  const std::span< uint8_t > & getData() const
+  uint8_t getServiceActionChannel() const
+  const std::span< uint8_t > & getGID() const
+  std::string getGIDString() const
+  ServiceActionOpCode getServiceActionOpCode() const
+  bool isServiceActionOpCode(ServiceActionOpCode serviceActionOpCode) const
+  static std::unique_ptr< ServiceActionMessage > fromMessage(const std::unique_ptr< MessageBase > &message)
+  static std::unique_ptr< ServiceActionMessage > forServiceAction(uint8_t serviceActionChannel, ServiceActionOpCode
   serviceActionOpCode, const std::vector< uint8_t > &data, const std::span< uint8_t > &gID)
#  size_t getSrcSize() const override
#  uint8_t * getSrc() const override
#  void store(const std::vector< uint8_t > &data) override
#  std::shared_ptr< const std::vector< uint8_t > > lz4DefaultCompression(uint32_t rate) const override
#  std::shared_ptr< const std::vector< uint8_t > > lz4HCCompression(uint32_t rate) const override
#  std::shared_ptr< const std::vector< uint8_t > > zlibCompression(uint32_t rate) const override
```

```
VectorMessage
#  std::shared_ptr< std::vector< uint8_t > > combinedVector
+  VectorMessage(uint8_t channel, Compressor compressor, uint32_t uncompressedSize, std::vector< uint8_t > &&combinedVector)
+  VectorMessage(uint8_t channel, Compressor compressor, uint32_t uncompressedSize, const std::shared_ptr
   < std::vector< uint8_t > > &combinedVector)
+  std::shared_ptr< const rclcpp::SerializedMessage > getSerializedMessage() const override
+  std::shared_ptr< const std::vector< uint8_t > > getCombinedVector() const override
+  std::shared_ptr< std::vector< uint8_t > > getCopiedCombinedVector() const override
+  std::string toString() const override
+  std::pair< const uint8_t, const bool > toSubscription() const override
+  static std::unique_ptr< VectorMessage > fromSerializedMessage(uint8_t channel, const rclcpp::SerializedMessage &serializedMessage)
+  static std::unique_ptr< VectorMessage > fromSerializedMessage(uint8_t channel, const std::shared_ptr<
   const rclcpp::SerializedMessage > &serializedMessage)
+  static std::unique_ptr< VectorMessage > forOpCodeSubscription(uint8_t channel, bool value)
+  static std::unique_ptr< VectorMessage > forOpCodeAuthentication(const std::string &value)
#  size_t getSrcSize() const override
#  uint8_t * getSrc() const override
#  void store(const std::vector< uint8_t > &data) override
```

Fig. B.3: Inheritance graph of `MessageBase`

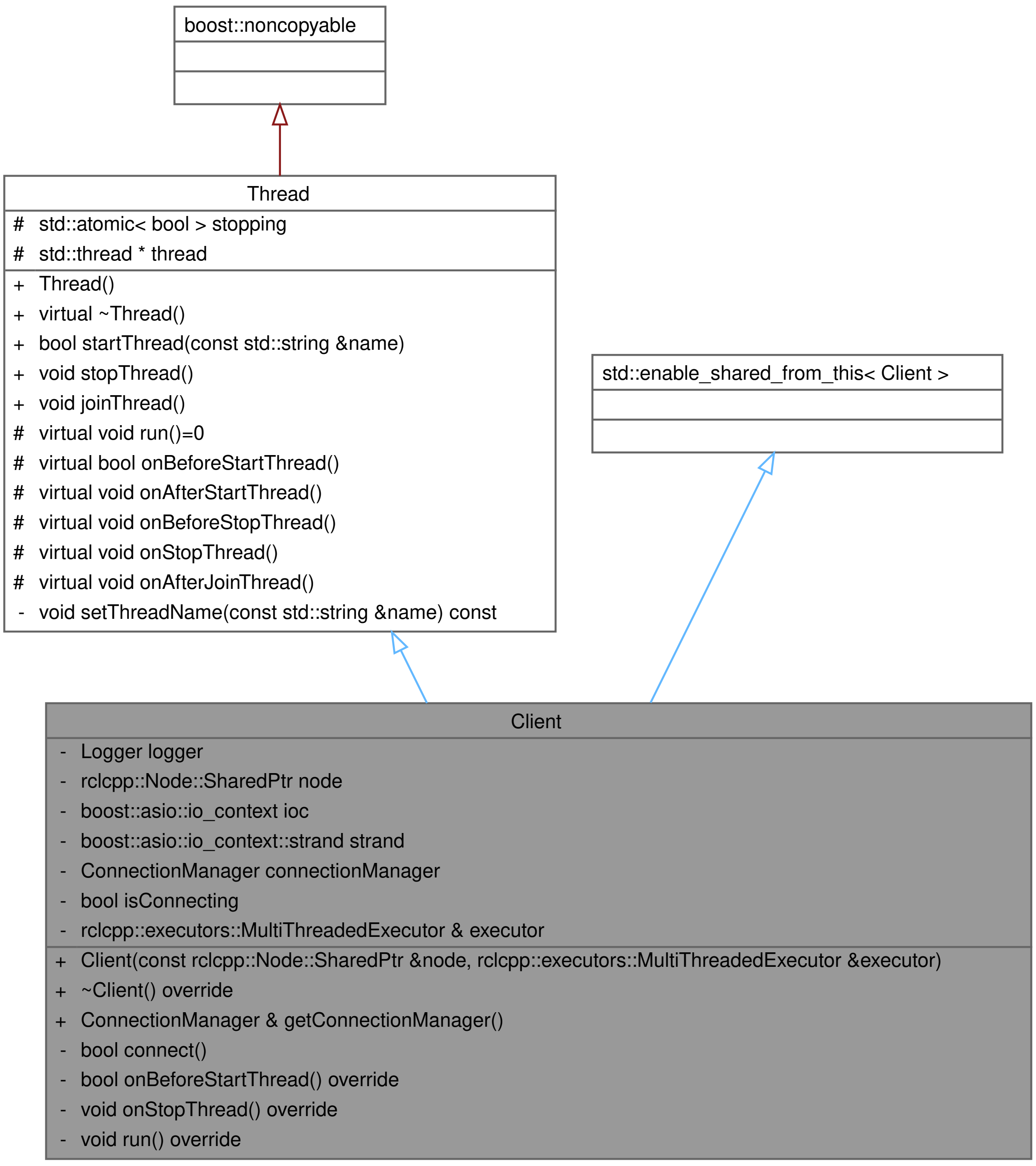

boost::noncopyable
Thread
# std::atomic< bool > stopping
# std::thread * thread
+ Thread()
+ virtual ~Thread()
+ bool startThread(const std::string &name)
+ void stopThread()
+ void joinThread()
# virtual void run()=0
# virtual bool onBeforeStartThread()
# virtual void onAfterStartThread()
# virtual void onBeforeStopThread()
# virtual void onStopThread()
# virtual void onAfterJoinThread()
- void setThreadName(const std::string &name) const
std::enable_shared_from_this< Client >
Client
- Logger logger
- rclcpp::Node::SharedPtr node
- boost::asio::io_context ioc
- boost::asio::io_context::strand strand
- ConnectionManager connectionManager
- bool isConnecting
- rclcpp::executors::MultiThreadedExecutor & executor
+ Client(const rclcpp::Node::SharedPtr &node, rclcpp::executors::MultiThreadedExecutor &executor)
+ ~Client() override
+ ConnectionManager & getConnectionManager()
- bool connect()
- bool onBeforeStartThread() override
- void onStopThread() override
- void run() override

Fig. B.4: Inheritance graph of `Client`

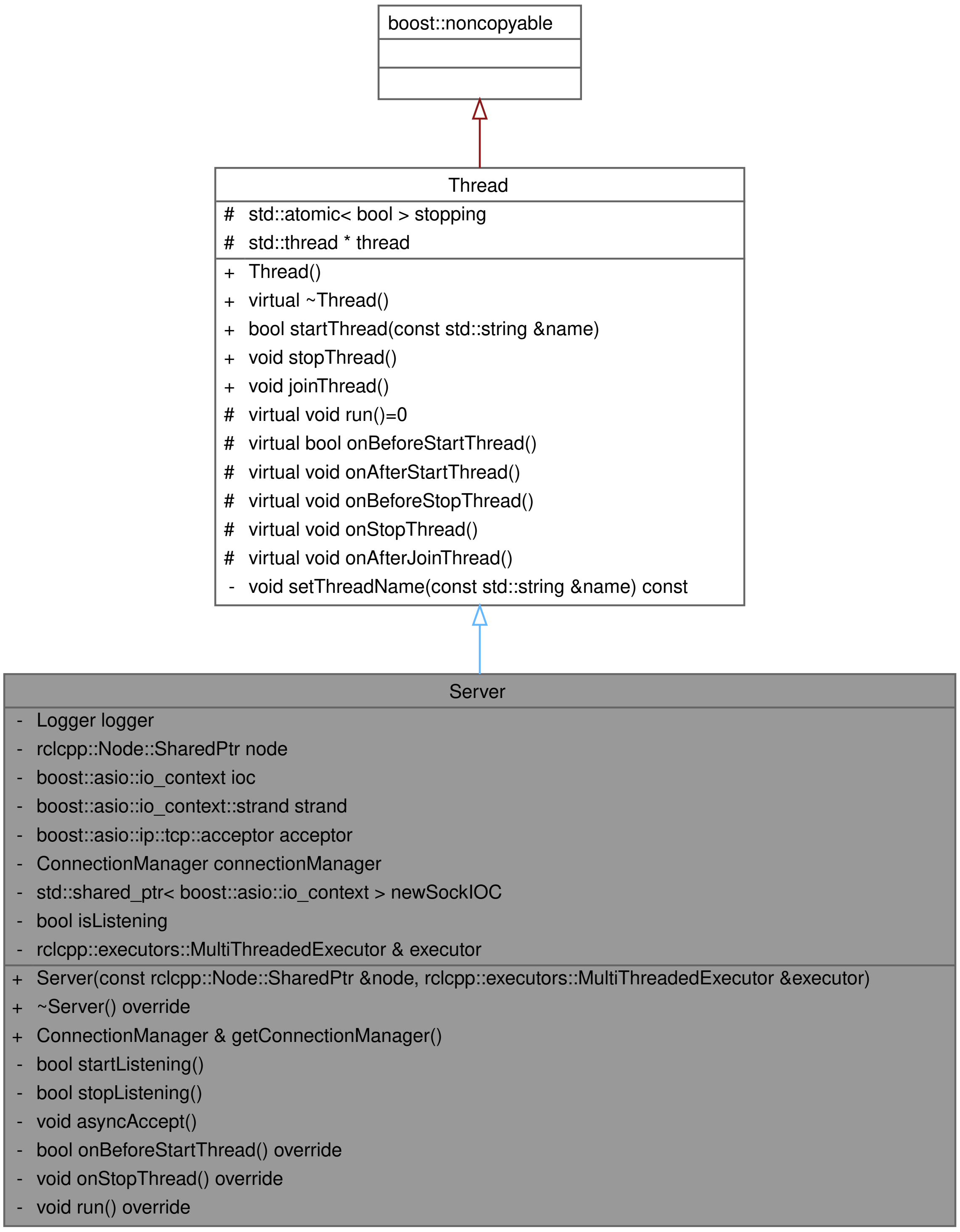

boost::noncopyable
Thread
# std::atomic< bool > stopping
# std::thread * thread
+ Thread()
+ virtual ~Thread()
+ bool startThread(const std::string &name)
+ void stopThread()
+ void joinThread()
# virtual void run()=0
# virtual bool onBeforeStartThread()
# virtual void onAfterStartThread()
# virtual void onBeforeStopThread()
# virtual void onStopThread()
# virtual void onAfterJoinThread()
- void setThreadName(const std::string &name) const
Server
- Logger logger
- rclcpp::Node::SharedPtr node
- boost::asio::io_context ioc
- boost::asio::io_context::strand strand
- boost::asio::ip::tcp::acceptor acceptor
- ConnectionManager connectionManager
- std::shared_ptr< boost::asio::io_context > newSockIOC
- bool isListening
- rclcpp::executors::MultiThreadedExecutor & executor
+ Server(const rclcpp::Node::SharedPtr &node, rclcpp::executors::MultiThreadedExecutor &executor)
+ ~Server() override
+ ConnectionManager & getConnectionManager()
- bool startListening()
- bool stopListening()
- void asyncAccept()
- bool onBeforeStartThread() override
- void onStopThread() override
- void run() override

Fig. B.5: Inheritance graph of `Server`

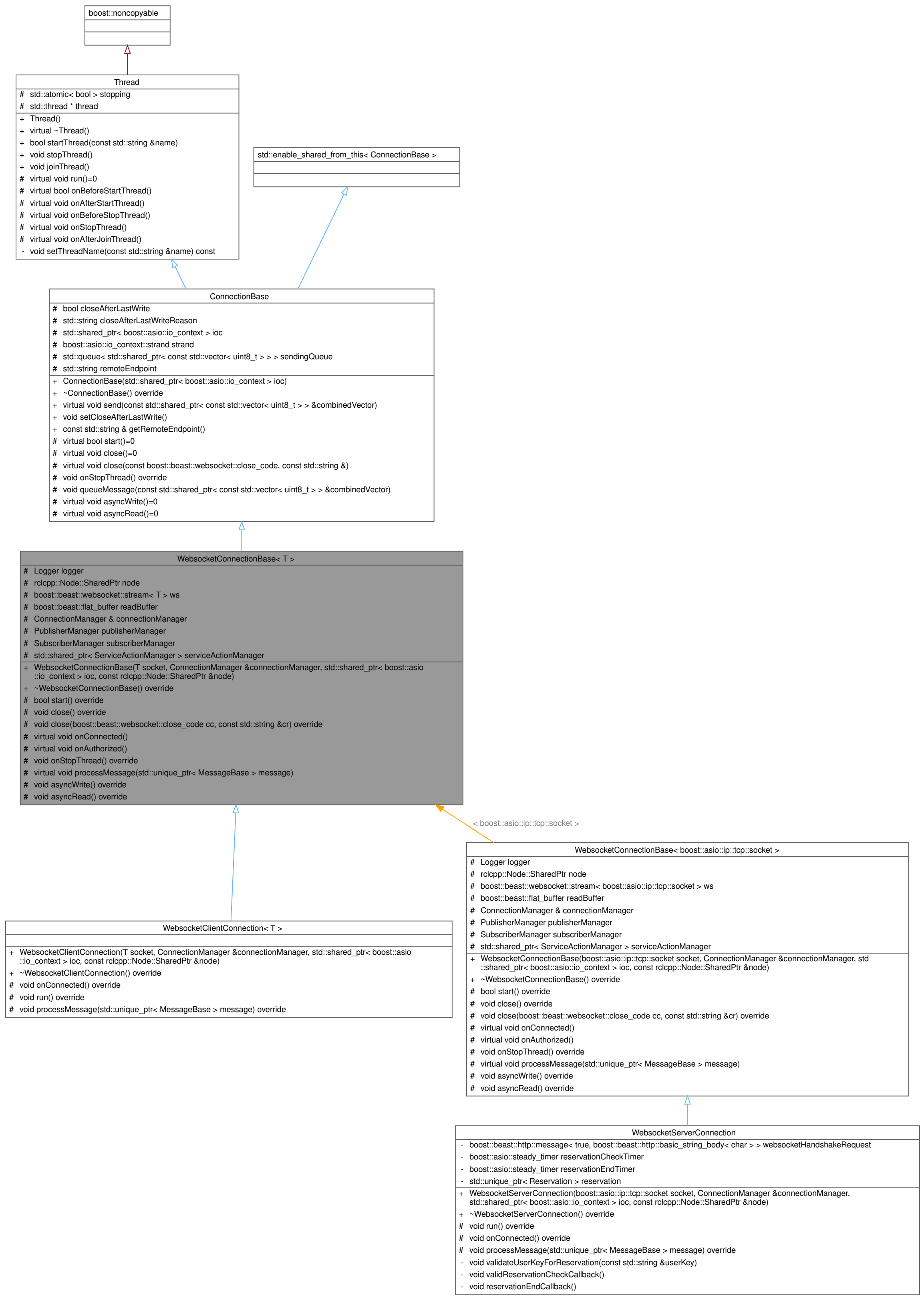

boost::noncopyable
Thread
# std::atomic< bool > stopping
# std::thread * thread
+ Thread()
+ virtual ~Thread()
+ bool startThread(const std::string &name)
+ void stopThread()
+ void joinThread()
# virtual void run()=0
# virtual bool onBeforeStartThread()
# virtual void onAfterStartThread()
# virtual void onBeforeStopThread()
# virtual void onStopThread()
# virtual void onAfterJoinThread()
- void setThreadName(const std::string &name) const
std::enable_shared_from_this< ConnectionBase >
ConnectionBase
# bool closeAfterLastWrite
# std::string closeAfterLastWriteReason
# std::shared_ptr< boost::asio::io_context > ioc
# boost::asio::io_context::strand strand
# std::queue< std::shared_ptr< const std::vector< uint8_t > > > sendingQueue
# std::string remoteEndpoint
+ ConnectionBase(std::shared_ptr< boost::asio::io_context > ioc)
+ ~ConnectionBase() override
+ virtual void send(const std::shared_ptr< const std::vector< uint8_t > > &combinedVector)
+ void setCloseAfterLastWrite()
+ const std::string & getRemoteEndpoint()
# virtual bool start()=0
# virtual void close()=0
# virtual void close(const boost::beast::websocket::close_code, const std::string &)
# void onStopThread() override
# void queueMessage(const std::shared_ptr< const std::vector< uint8_t > > &combinedVector)
# virtual void asyncWrite()=0
# virtual void asyncRead()=0
WebsocketConnectionBase< T >
# Logger logger
# rclcpp::Node::SharedPtr node
# boost::beast::websocket::stream< T > ws
# boost::beast::flat_buffer readBuffer
# ConnectionManager & connectionManager
# PublisherManager publisherManager
# SubscriberManager subscriberManager
# std::shared_ptr< ServiceActionManager > serviceActionManager
+ WebsocketConnectionBase(T socket, ConnectionManager &connectionManager, std::shared_ptr< boost::asio ::io_context > ioc, const rclcpp::Node::SharedPtr &node)
+ ~WebsocketConnectionBase() override
# bool start() override
# void close() override
# void close(boost::beast::websocket::close_code cc, const std::string &cr) override
# virtual void onConnected()
# virtual void onAuthorized()
# void onStopThread() override
# virtual void processMessage(std::unique_ptr< MessageBase > message)
# void asyncWrite() override
# void asyncRead() override
< boost::asio::ip::tcp::socket >
WebsocketClientConnection< T >
+ WebsocketClientConnection(T socket, ConnectionManager &connectionManager, std::shared_ptr< boost::asio ::io_context > ioc, const rclcpp::Node::SharedPtr &node)
+ ~WebsocketClientConnection() override
# void onConnected() override
# void run() override
# void processMessage(std::unique_ptr< MessageBase > message) override
WebsocketConnectionBase< boost::asio::ip::tcp::socket >
# Logger logger
# rclcpp::Node::SharedPtr node
# boost::beast::websocket::stream< boost::asio::ip::tcp::socket > ws
# boost::beast::flat_buffer readBuffer
# ConnectionManager & connectionManager
# PublisherManager publisherManager
# SubscriberManager subscriberManager
# std::shared_ptr< ServiceActionManager > serviceActionManager
+ WebsocketConnectionBase(boost::asio::ip::tcp::socket socket, ConnectionManager &connectionManager, std ::shared_ptr< boost::asio::io_context > ioc, const rclcpp::Node::SharedPtr &node)
+ ~WebsocketConnectionBase() override
# bool start() override
# void close() override
# void close(boost::beast::websocket::close_code cc, const std::string &cr) override
# virtual void onConnected()
# virtual void onAuthorized()
# void onStopThread() override
# virtual void processMessage(std::unique_ptr< MessageBase > message)
# void asyncWrite() override
# void asyncRead() override
WebsocketServerConnection
- boost::beast::http::message< true, boost::beast::http::basic_string_body< char > > websocketHandshakeRequest
- boost::asio::steady_timer reservationCheckTimer
- boost::asio::steady_timer reservationEndTimer
- std::unique_ptr< Reservation > reservation
+ WebsocketServerConnection(boost::asio::ip::tcp::socket socket, ConnectionManager &connectionManager, std::shared_ptr< boost::asio::io_context > ioc, const rclcpp::Node::SharedPtr &node)
+ ~WebsocketServerConnection() override
# void run() override
# void onConnected() override
# void processMessage(std::unique_ptr< MessageBase > message) override
- void validateUserKeyForReservation(const std::string &userKey)
- void validReservationCheckCallback()
- void reservationEndCallback()

Fig. B.6: Inheritance graph of `WebsocketConnectionBase`

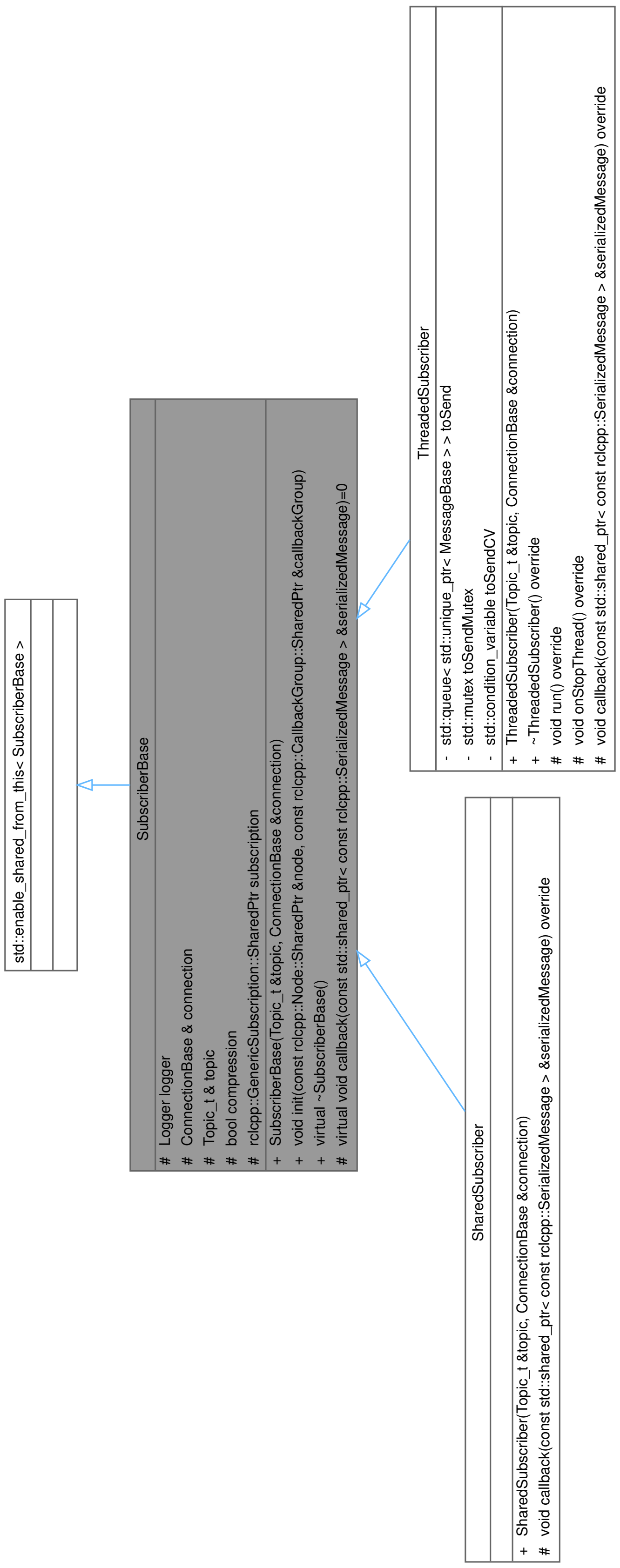

std::enable_shared_from_this< SubscriberBase >
SubscriberBase
# Logger logger
# ConnectionBase & connection
# Topic_t & topic
# bool compression
# rclcpp::GenericSubscription::SharedPtr subscription
+ SubscriberBase(Topic_t &topic, ConnectionBase &connection)
+ void init(const rclcpp::Node::SharedPtr &node, const rclcpp::CallbackGroup::SharedPtr &callbackGroup)
+ virtual ~SubscriberBase()
# virtual void callback(const std::shared_ptr< const rclcpp::SerializedMessage > &serializedMessage)=0
ThreadedSubscriber
- std::queue< std::unique_ptr< MessageBase > > toSend
- std::mutex toSendMutex
- std::condition_variable toSendCV
+ ThreadedSubscriber(Topic_t &topic, ConnectionBase &connection)
+ ~ThreadedSubscriber() override
# void run() override
# void onStopThread() override
# void callback(const std::shared_ptr< const rclcpp::SerializedMessage > &serializedMessage) override
SharedSubscriber
+ SharedSubscriber(Topic_t &topic, ConnectionBase &connection)
# void callback(const std::shared_ptr< const rclcpp::SerializedMessage > &serializedMessage) override

Fig. B.7: Inheritance graph of `SubscriberBase`

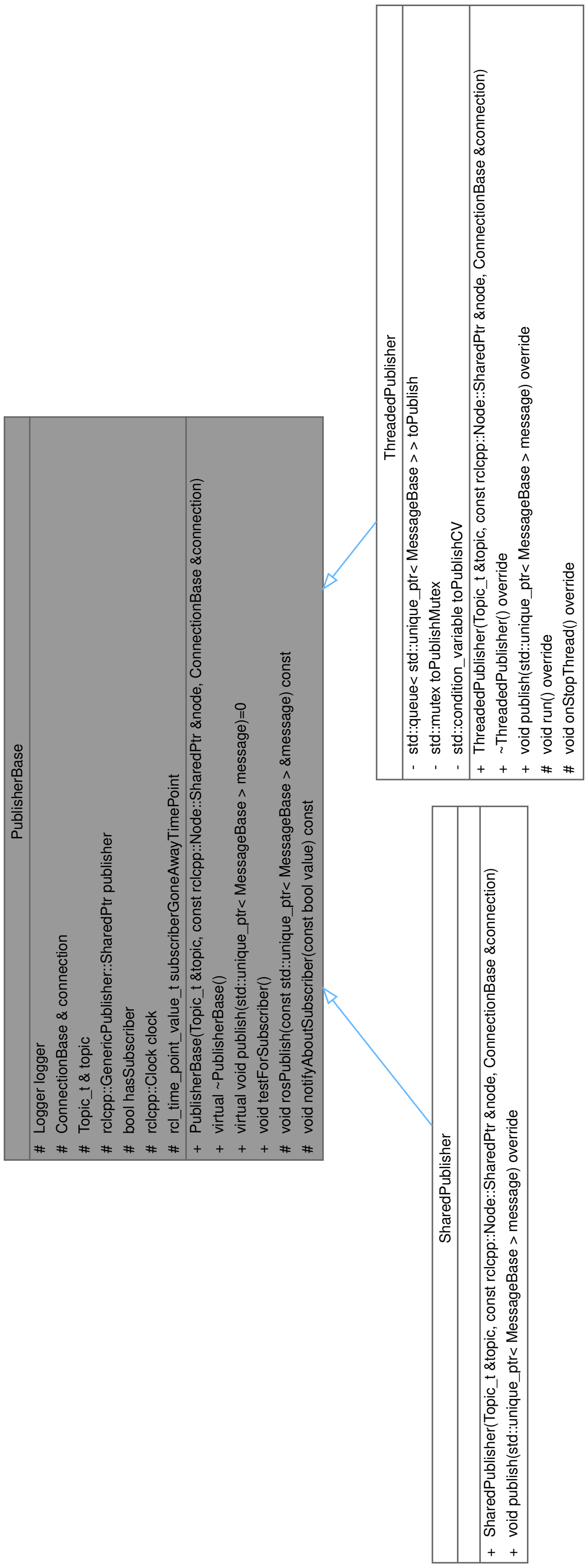

PublisherBase
# Logger logger
# ConnectionBase & connection
# Topic_t & topic
# rclcpp::GenericPublisher::SharedPtr publisher
# bool hasSubscriber
# rclcpp::Clock clock
# rcl_time_point_value_t subscriberGoneAwayTimePoint
+ PublisherBase(Topic_t &topic, const rclcpp::Node::SharedPtr &node, ConnectionBase &connection)
+ virtual ~PublisherBase()
+ virtual void publish(std::unique_ptr< MessageBase > message)=0
+ void testForSubscriber()
# void rosPublish(const std::unique_ptr< MessageBase > &message) const
# void notifyAboutSubscriber(const bool value) const
ThreadedPublisher
- std::queue< std::unique_ptr< MessageBase > > toPublish
- std::mutex toPublishMutex
- std::condition_variable toPublishCV
+ ThreadedPublisher(Topic_t &topic, const rclcpp::Node::SharedPtr &node, ConnectionBase &connection)
+ ~ThreadedPublisher() override
+ void publish(std::unique_ptr< MessageBase > message) override
# void run() override
# void onStopThread() override
SharedPublisher
+ SharedPublisher(Topic_t &topic, const rclcpp::Node::SharedPtr &node, ConnectionBase &connection)
+ void publish(std::unique_ptr< MessageBase > message) override

Fig. B.8: Inheritance graph of `PublisherBase`

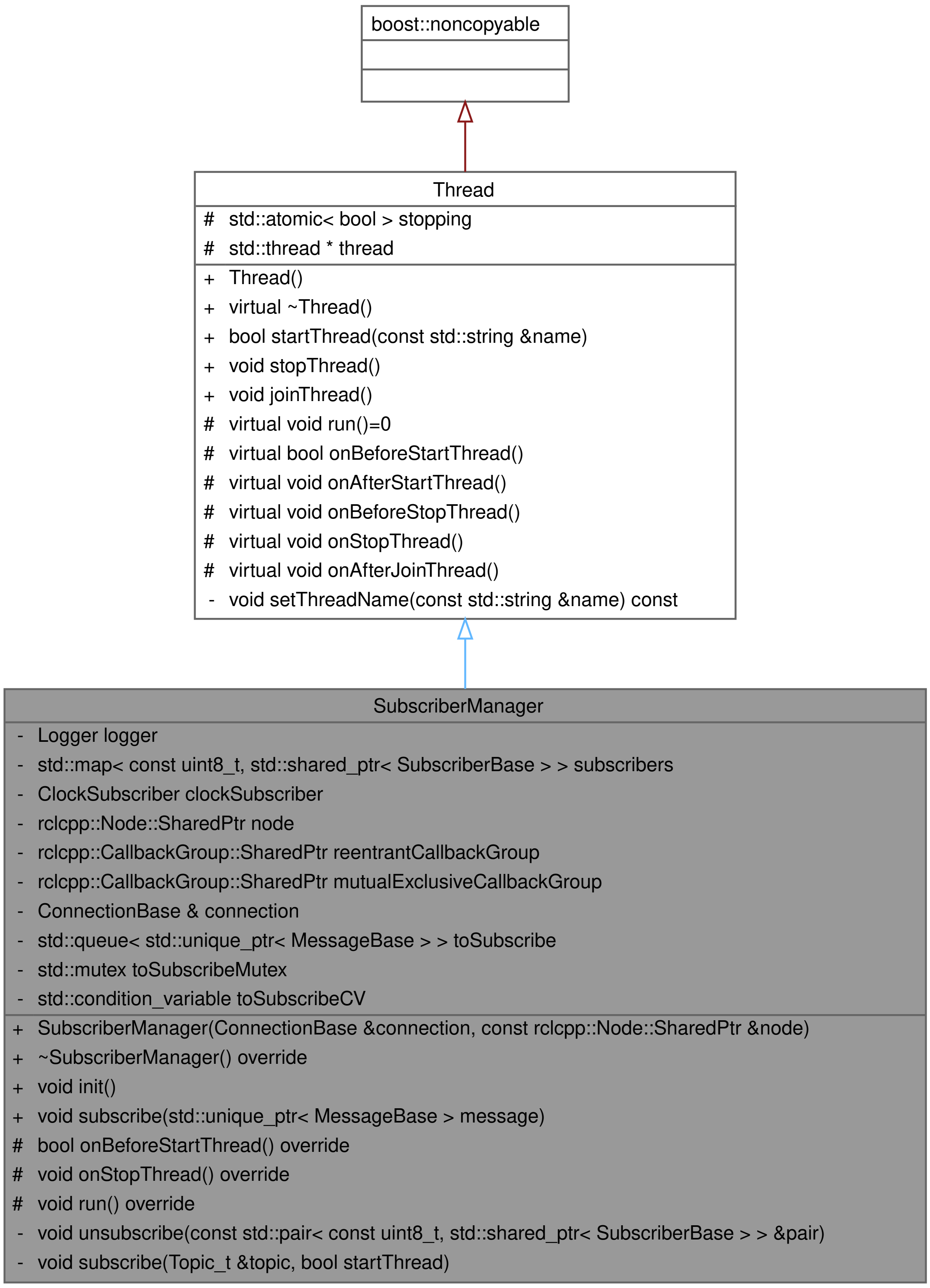

boost::noncopyable
Thread
# std::atomic< bool > stopping
# std::thread * thread
+ Thread()
+ virtual ~Thread()
+ bool startThread(const std::string &name)
+ void stopThread()
+ void joinThread()
# virtual void run()=0
# virtual bool onBeforeStartThread()
# virtual void onAfterStartThread()
# virtual void onBeforeStopThread()
# virtual void onStopThread()
# virtual void onAfterJoinThread()
- void setThreadName(const std::string &name) const
SubscriberManager
- Logger logger
- std::map< const uint8_t, std::shared_ptr< SubscriberBase > > subscribers
- ClockSubscriber clockSubscriber
- rclcpp::Node::SharedPtr node
- rclcpp::CallbackGroup::SharedPtr reentrantCallbackGroup
- rclcpp::CallbackGroup::SharedPtr mutualExclusiveCallbackGroup
- ConnectionBase & connection
- std::queue< std::unique_ptr< MessageBase > > toSubscribe
- std::mutex toSubscribeMutex
- std::condition_variable toSubscribeCV
+ SubscriberManager(ConnectionBase &connection, const rclcpp::Node::SharedPtr &node)
+ ~SubscriberManager() override
+ void init()
+ void subscribe(std::unique_ptr< MessageBase > message)
# bool onBeforeStartThread() override
# void onStopThread() override
# void run() override
- void unsubscribe(const std::pair< const uint8_t, std::shared_ptr< SubscriberBase > > &pair)
- void subscribe(Topic_t &topic, bool startThread)

Fig. B.9: Inheritance graph of `SubscriberManager`

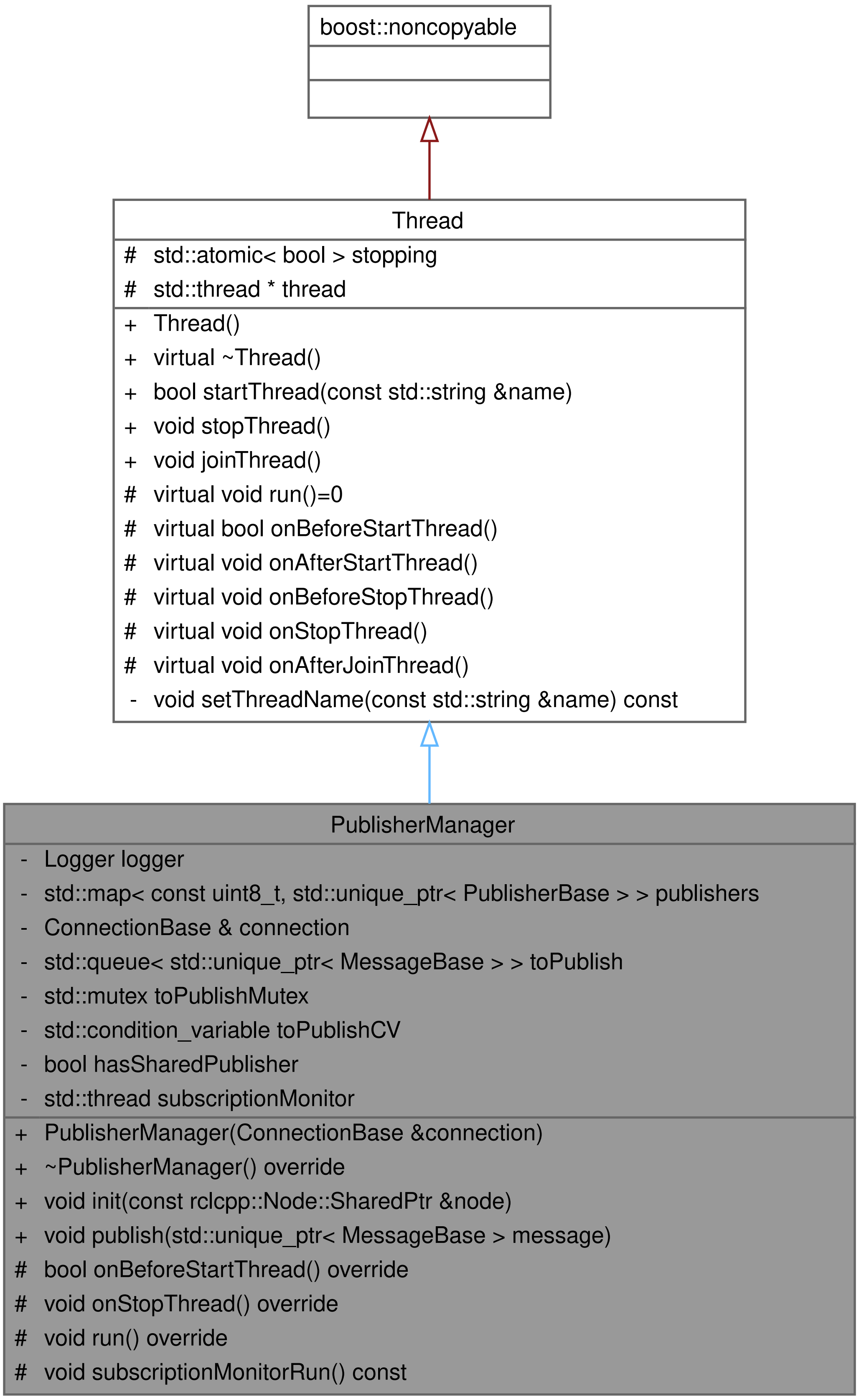

boost::noncopyable
Thread
# std::atomic< bool > stopping
# std::thread * thread
+ Thread()
+ virtual ~Thread()
+ bool startThread(const std::string &name)
+ void stopThread()
+ void joinThread()
# virtual void run()=0
# virtual bool onBeforeStartThread()
# virtual void onAfterStartThread()
# virtual void onBeforeStopThread()
# virtual void onStopThread()
# virtual void onAfterJoinThread()
- void setThreadName(const std::string &name) const
PublisherManager
- Logger logger
- std::map< const uint8_t, std::unique_ptr< PublisherBase > > publishers
- ConnectionBase & connection
- std::queue< std::unique_ptr< MessageBase > > toPublish
- std::mutex toPublishMutex
- std::condition_variable toPublishCV
- bool hasSharedPublisher
- std::thread subscriptionMonitor
+ PublisherManager(ConnectionBase &connection)
+ ~PublisherManager() override
+ void init(const rclcpp::Node::SharedPtr &node)
+ void publish(std::unique_ptr< MessageBase > message)
# bool onBeforeStartThread() override
# void onStopThread() override
# void run() override
# void subscriptionMonitorRun() const

Fig. B.10: Inheritance graph of `PublisherManager`

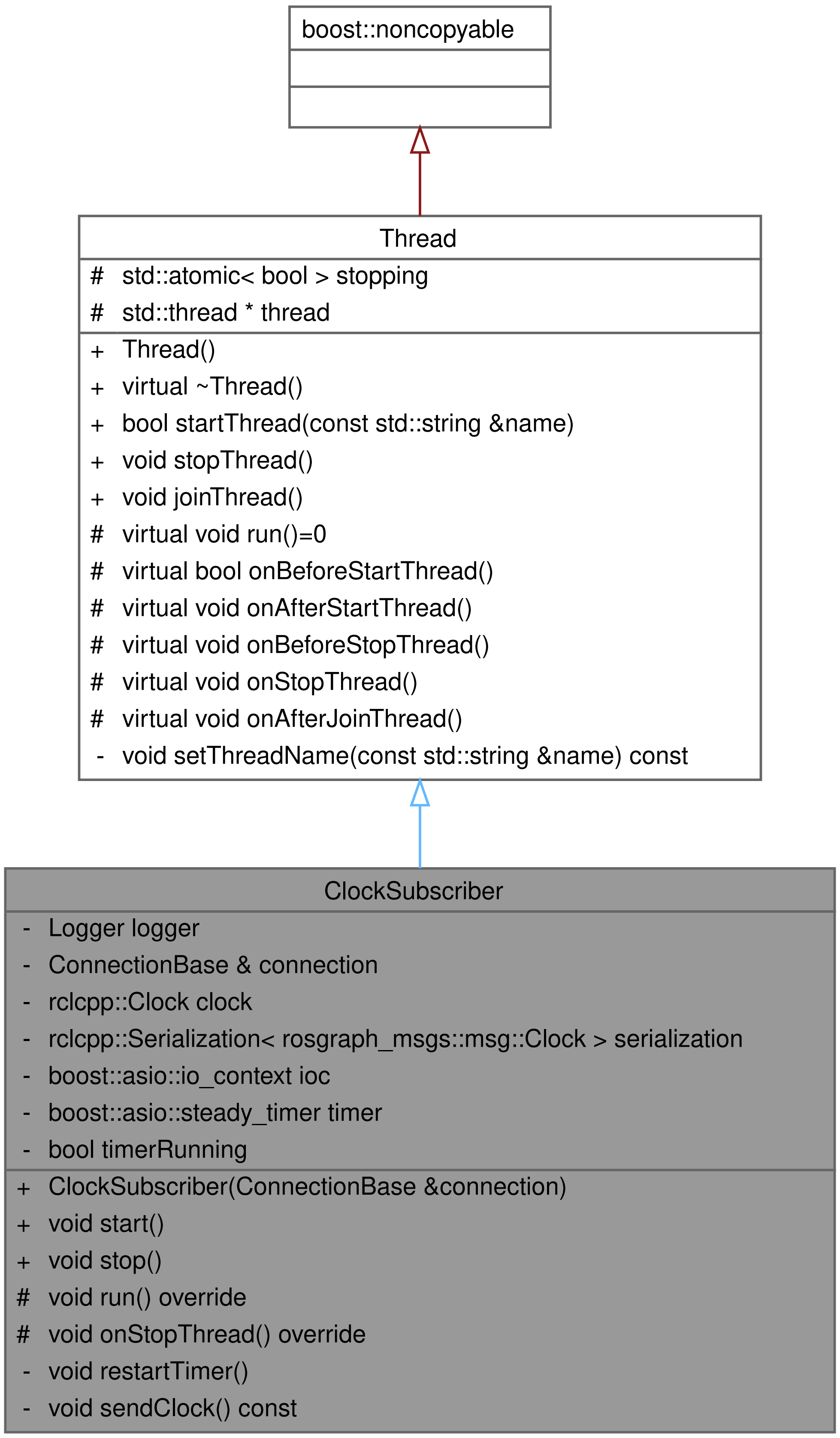
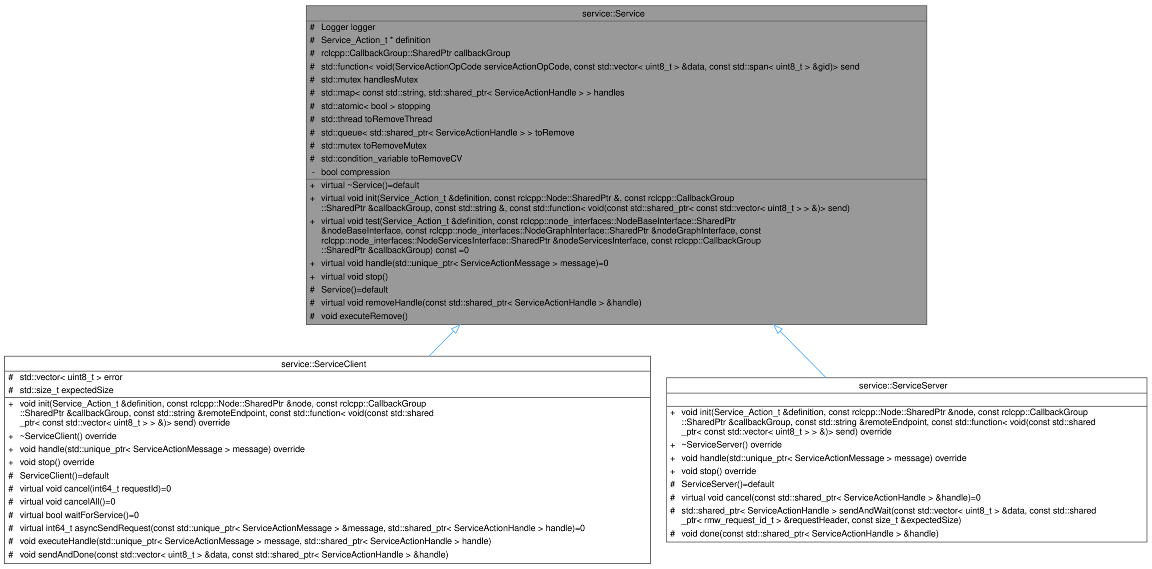

boost::noncopyable
Thread
# std::atomic< bool > stopping
# std::thread * thread
+ Thread()
+ virtual ~Thread()
+ bool startThread(const std::string &name)
+ void stopThread()
+ void joinThread()
# virtual void run()=0
# virtual bool onBeforeStartThread()
# virtual void onAfterStartThread()
# virtual void onBeforeStopThread()
# virtual void onStopThread()
# virtual void onAfterJoinThread()
- void setThreadName(const std::string &name) const
ClockSubscriber
- Logger logger
- ConnectionBase & connection
- rclcpp::Clock clock
- rclcpp::Serialization< rosgraph_msgs::msg::Clock > serialization
- boost::asio::io_context ioc
- boost::asio::steady_timer timer
- bool timerRunning
+ ClockSubscriber(ConnectionBase &connection)
+ void start()
+ void stop()
# void run() override
# void onStopThread() override
- void restartTimer()
- void sendClock() const

Fig. B.11: Inheritance graph of `ClockSubscriber`

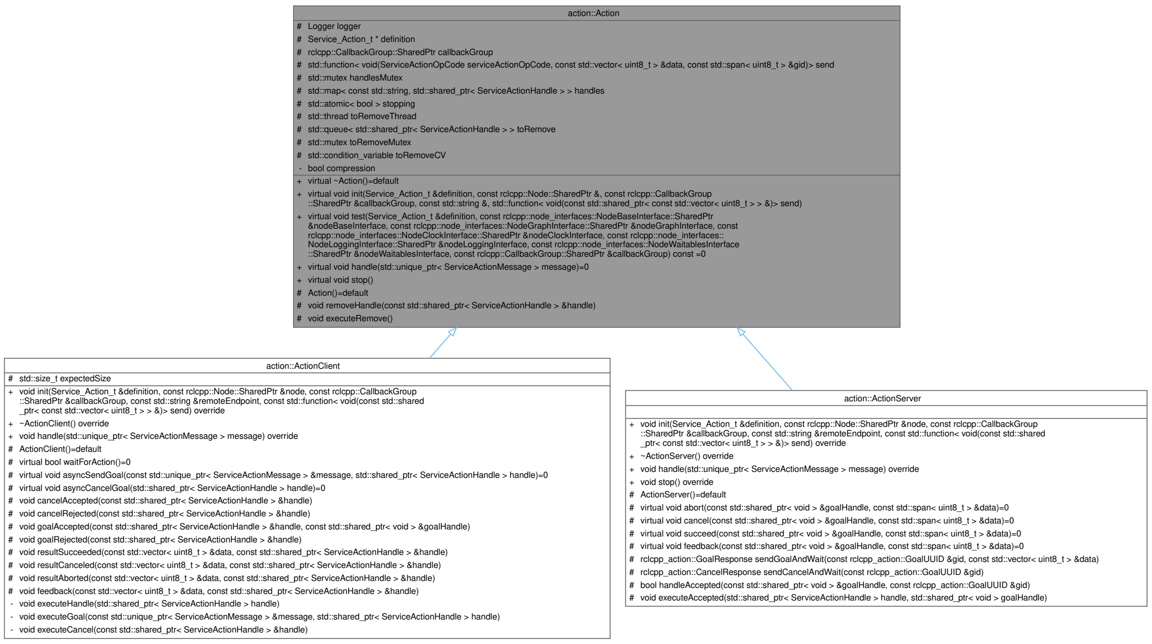


```
service::Service
# Logger logger
# Service_Action_t * definition
# rclcpp::CallbackGroup::SharedPtr callbackGroup
# std::function< void(ServiceActionOpCode serviceActionOpCode, const std::vector< uint8_t > &data, const std::span< uint8_t > &gid)> send
# std::mutex handlesMutex
# std::map< const std::string, std::shared_ptr< ServiceActionHandle > > handles
# std::atomic< bool > stopping
# std::thread toRemoveThread
# std::queue< std::shared_ptr< ServiceActionHandle > > toRemove
# std::mutex toRemoveMutex
# std::condition_variable toRemoveCV
- bool compression
+ virtual ~Service()=default
+ virtual void init(Service_Action_t &definition, const rclcpp::Node::SharedPtr &, const rclcpp::CallbackGroup
  ::SharedPtr &callbackGroup, const std::string &, const std::function< void(const std::shared_ptr< const std::vector< uint8_t > > &)> send)
+ virtual void test(Service_Action_t &definition, const rclcpp::node_interfaces::NodeBaseInterface::SharedPtr
  &nodeBaseInterface, const rclcpp::node_interfaces::NodeGraphInterface::SharedPtr &nodeGraphInterface, const
  rclcpp::node_interfaces::NodeServicesInterface::SharedPtr &nodeServicesInterface, const rclcpp::CallbackGroup
  ::SharedPtr &callbackGroup) const =0
+ virtual void handle(std::unique_ptr< ServiceActionMessage > message)=0
+ virtual void stop()
# Service()=default
# virtual void removeHandle(const std::shared_ptr< ServiceActionHandle > &handle)
# void executeRemove()
```

```
service::ServiceClient
# std::vector< uint8_t > error
# std::size_t expectedSize
+ void init(Service_Action_t &definition, const rclcpp::Node::SharedPtr &node, const rclcpp::CallbackGroup
  ::SharedPtr &callbackGroup, const std::string &remoteEndpoint, const std::function< void(const std::shared
  _ptr< const std::vector< uint8_t > > &)> send) override
+ ~ServiceClient() override
+ void handle(std::unique_ptr< ServiceActionMessage > message) override
+ void stop() override
# ServiceClient()=default
# virtual void cancel(int64_t requestId)=0
# virtual void cancelAll()=0
# virtual bool waitForService()=0
# virtual int64_t asyncSendRequest(const std::unique_ptr< ServiceActionMessage > &message, std::shared_ptr< ServiceActionHandle > handle)=0
# void executeHandle(std::unique_ptr< ServiceActionMessage > message, std::shared_ptr< ServiceActionHandle > handle)
# void sendAndDone(const std::vector< uint8_t > &data, const std::shared_ptr< ServiceActionHandle > &handle)
```

```
service::ServiceServer
+ void init(Service_Action_t &definition, const rclcpp::Node::SharedPtr &node, const rclcpp::CallbackGroup
  ::SharedPtr &callbackGroup, const std::string &remoteEndpoint, const std::function< void(const std::shared
  _ptr< const std::vector< uint8_t > > &)> send) override
+ ~ServiceServer() override
+ void handle(std::unique_ptr< ServiceActionMessage > message) override
+ void stop() override
# ServiceServer()=default
# virtual void cancel(const std::shared_ptr< ServiceActionHandle > &handle)=0
# std::shared_ptr< ServiceActionHandle > sendAndWait(const std::vector< uint8_t > &data, const std::shared
  _ptr< rmw_request_id_t > &requestHeader, const size_t &expectedSize)
# void done(const std::shared_ptr< ServiceActionHandle > &handle)
```

Fig. B.12: Inheritance graph of `Service`



```
action::Action
# Logger logger
# Service_Action_t * definition
# rclcpp::CallbackGroup::SharedPtr callbackGroup
# std::function< void(ServiceActionOpCode serviceActionOpCode, const std::vector< uint8_t > &data, const std::span< uint8_t > &gid)> send
# std::mutex handlesMutex
# std::map< const std::string, std::shared_ptr< ServiceActionHandle > > handles
# std::atomic< bool > stopping
# std::thread toRemoveThread
# std::queue< std::shared_ptr< ServiceActionHandle > > toRemove
# std::mutex toRemoveMutex
# std::condition_variable toRemoveCV
- bool compression
+ virtual ~Action()=default
+ virtual void init(Service_Action_t &definition, const rclcpp::Node::SharedPtr &, const rclcpp::CallbackGroup
  ::SharedPtr &callbackGroup, const std::string &, std::function< void(const std::shared_ptr< const std::vector< uint8_t > > &)> send)
+ virtual void test(Service_Action_t &definition, const rclcpp::node_interfaces::NodeBaseInterface::SharedPtr
  &nodeBaseInterface, const rclcpp::node_interfaces::NodeGraphInterface::SharedPtr &nodeGraphInterface, const
  rclcpp::node_interfaces::NodeClockInterface::SharedPtr &nodeClockInterface, const rclcpp::node_interfaces::
  NodeLoggingInterface::SharedPtr &nodeLoggingInterface, const rclcpp::node_interfaces::NodeWaitablesInterface
  ::SharedPtr &nodeWaitablesInterface, const rclcpp::CallbackGroup::SharedPtr &callbackGroup) const =0
+ virtual void handle(std::unique_ptr< ServiceActionMessage > message)=0
+ virtual void stop()
# Action()=default
# void removeHandle(const std::shared_ptr< ServiceActionHandle > &handle)
# void executeRemove()
```

```
action::ActionClient
# std::size_t expectedSize
+ void init(Service_Action_t &definition, const rclcpp::Node::SharedPtr &node, const rclcpp::CallbackGroup
  ::SharedPtr &callbackGroup, const std::string &remoteEndpoint, const std::function< void(const std::shared
  _ptr< const std::vector< uint8_t > > &)> send) override
+ ~ActionClient() override
+ void handle(std::unique_ptr< ServiceActionMessage > message) override
# ActionClient()=default
# virtual bool waitForAction()=0
# virtual void asyncSendGoal(const std::unique_ptr< ServiceActionMessage > &message, std::shared_ptr< ServiceActionHandle > handle)=0
# virtual void asyncCancelGoal(std::shared_ptr< ServiceActionHandle > handle)=0
# void cancelAccepted(const std::shared_ptr< ServiceActionHandle > &handle)
# void cancelRejected(const std::shared_ptr< ServiceActionHandle > &handle)
# void goalAccepted(const std::shared_ptr< ServiceActionHandle > &handle, const std::shared_ptr< void > &goalHandle)
# void goalRejected(const std::shared_ptr< ServiceActionHandle > &handle)
# void resultSucceeded(const std::vector< uint8_t > &data, const std::shared_ptr< ServiceActionHandle > &handle)
# void resultCanceled(const std::vector< uint8_t > &data, const std::shared_ptr< ServiceActionHandle > &handle)
# void resultAborted(const std::vector< uint8_t > &data, const std::shared_ptr< ServiceActionHandle > &handle)
# void feedback(const std::vector< uint8_t > &data, const std::shared_ptr< ServiceActionHandle > &handle)
- void executeHandle(std::shared_ptr< ServiceActionHandle > handle)
- void executeGoal(const std::unique_ptr< ServiceActionMessage > &message, std::shared_ptr< ServiceActionHandle > handle)
- void executeCancel(const std::shared_ptr< ServiceActionHandle > &handle)
```

```
action::ActionServer
+ void init(Service_Action_t &definition, const rclcpp::Node::SharedPtr &node, const rclcpp::CallbackGroup
  ::SharedPtr &callbackGroup, const std::string &remoteEndpoint, const std::function< void(const std::shared
  _ptr< const std::vector< uint8_t > > &)> send) override
+ ~ActionServer() override
+ void handle(std::unique_ptr< ServiceActionMessage > message) override
+ void stop() override
# ActionServer()=default
# virtual void abort(const std::shared_ptr< void > &goalHandle, const std::span< uint8_t > &data)=0
# virtual void cancel(const std::shared_ptr< void > &goalHandle, const std::span< uint8_t > &data)=0
# virtual void succeed(const std::shared_ptr< void > &goalHandle, const std::span< uint8_t > &data)=0
# virtual void feedback(const std::shared_ptr< void > &goalHandle, const std::span< uint8_t > &data)=0
# rclcpp_action::GoalResponse sendGoalAndWait(const rclcpp_action::GoalUUID &gid, const std::vector< uint8_t > &data)
# rclcpp_action::CancelResponse sendCancelAndWait(const rclcpp_action::GoalUUID &gid)
# bool handleAccepted(const std::shared_ptr< void > &goalHandle, const rclcpp_action::GoalUUID &gid)
# void executeAccepted(std::shared_ptr< ServiceActionHandle > handle, std::shared_ptr< void > goalHandle)
```

Fig. B.13: Inheritance graph of `Action`

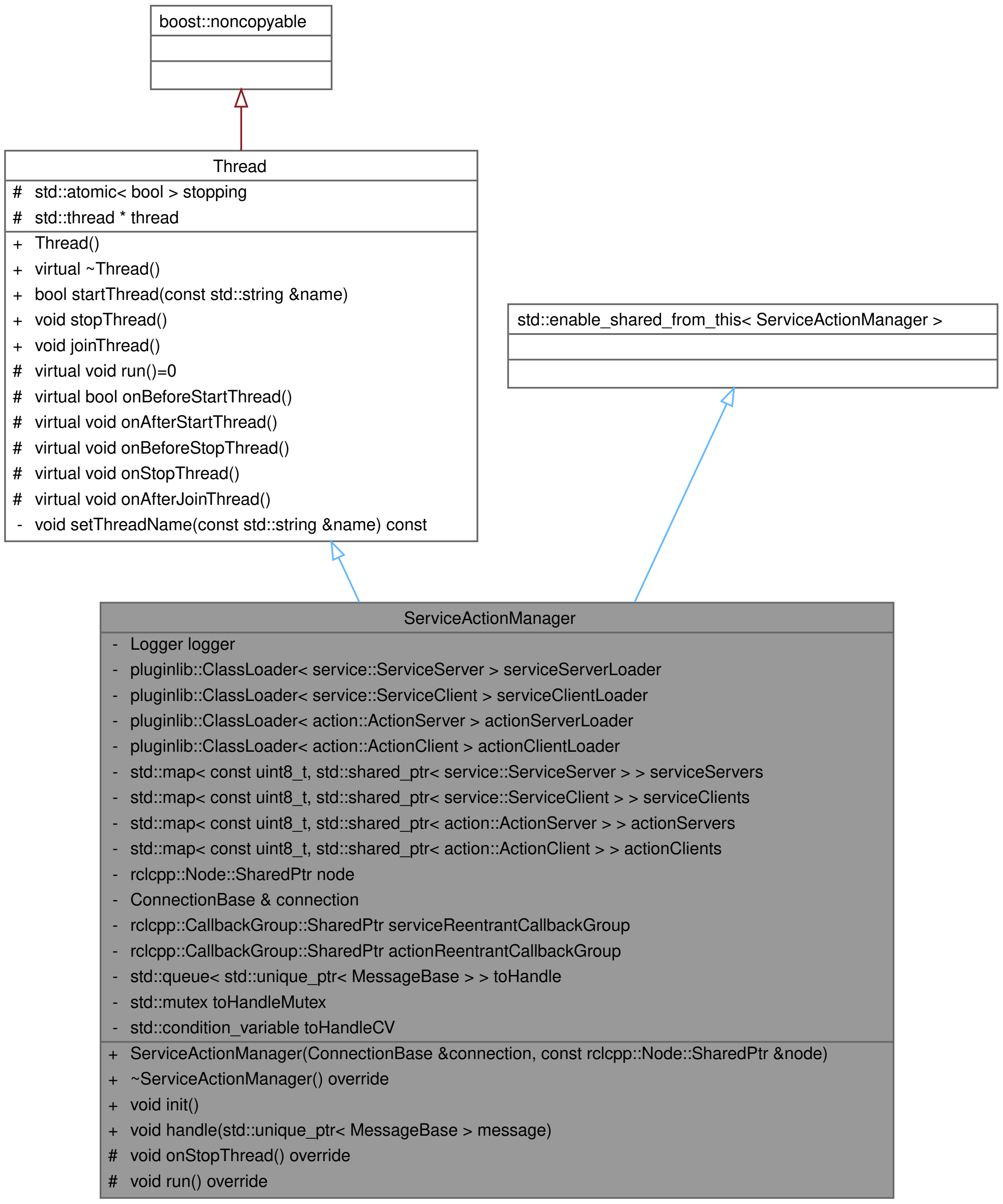

boost::noncopyable
Thread
# std::atomic< bool > stopping
# std::thread * thread
+ Thread()
+ virtual ~Thread()
+ bool startThread(const std::string &name)
+ void stopThread()
+ void joinThread()
# virtual void run()=0
# virtual bool onBeforeStartThread()
# virtual void onAfterStartThread()
# virtual void onBeforeStopThread()
# virtual void onStopThread()
# virtual void onAfterJoinThread()
- void setThreadName(const std::string &name) const
std::enable_shared_from_this< ServiceActionManager >
ServiceActionManager
- Logger logger
- pluginlib::ClassLoader< service::ServiceServer > serviceServerLoader
- pluginlib::ClassLoader< service::ServiceClient > serviceClientLoader
- pluginlib::ClassLoader< action::ActionServer > actionServerLoader
- pluginlib::ClassLoader< action::ActionClient > actionClientLoader
- std::map< const uint8_t, std::shared_ptr< service::ServiceServer > > serviceServers
- std::map< const uint8_t, std::shared_ptr< service::ServiceClient > > serviceClients
- std::map< const uint8_t, std::shared_ptr< action::ActionServer > > actionServers
- std::map< const uint8_t, std::shared_ptr< action::ActionClient > > actionClients
- rclcpp::Node::SharedPtr node
- ConnectionBase & connection
- rclcpp::CallbackGroup::SharedPtr serviceReentrantCallbackGroup
- rclcpp::CallbackGroup::SharedPtr actionReentrantCallbackGroup
- std::queue< std::unique_ptr< MessageBase > > toHandle
- std::mutex toHandleMutex
- std::condition_variable toHandleCV
+ ServiceActionManager(ConnectionBase &connection, const rclcpp::Node::SharedPtr &node)
+ ~ServiceActionManager() override
+ void init()
+ void handle(std::unique_ptr< MessageBase > message)
# void onStopThread() override
# void run() override

Fig. B.14: Inheritance graph of `ServiceActionManager`

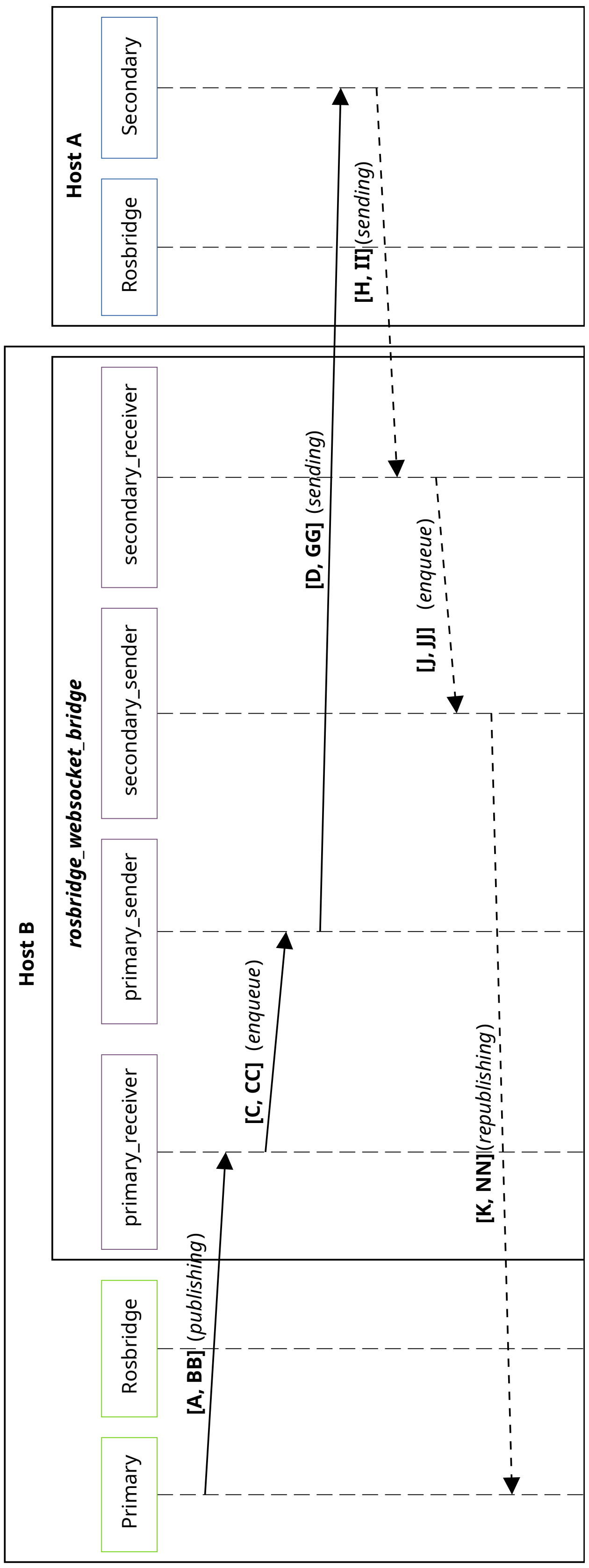


Fig. B.15: Rosbridge Time Measurement Sequence

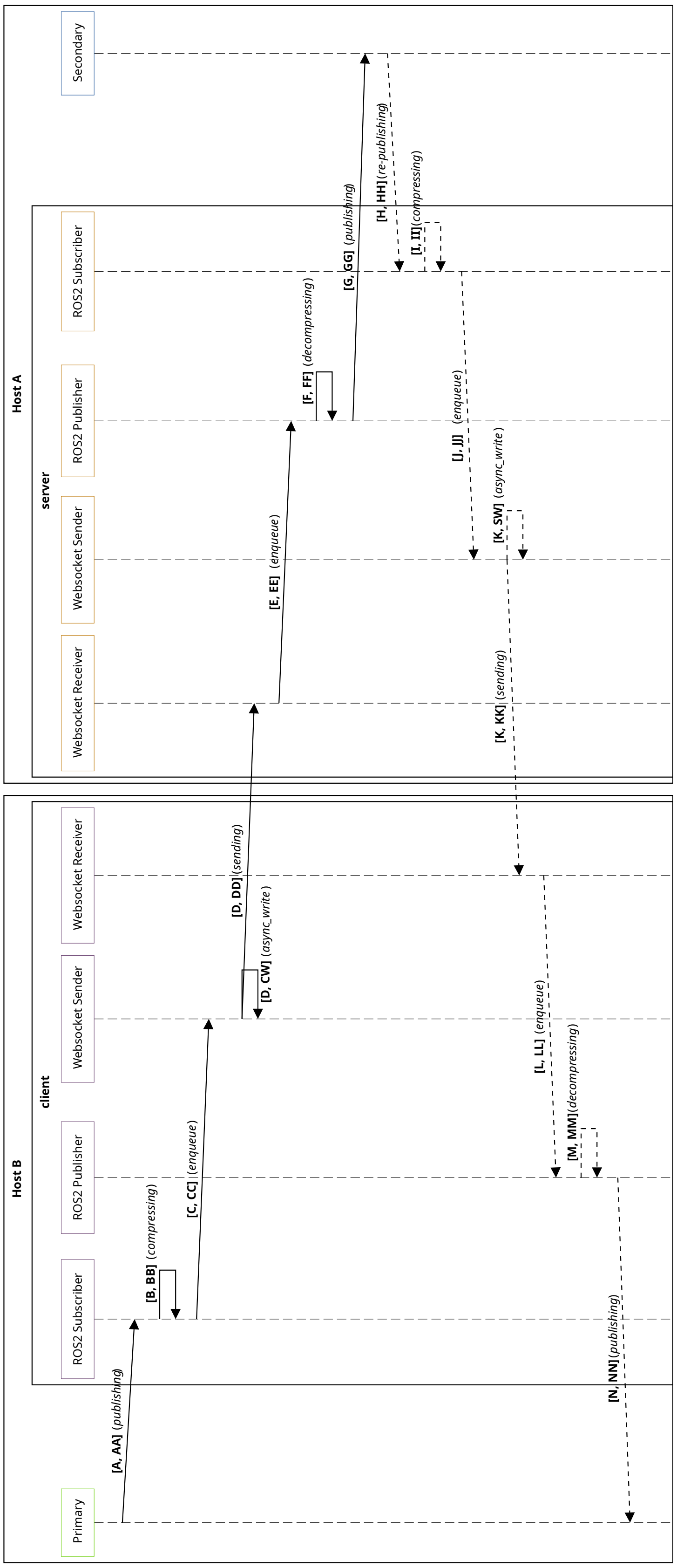


Fig. B.16: ROS2 Connect Time Measurement Sequence

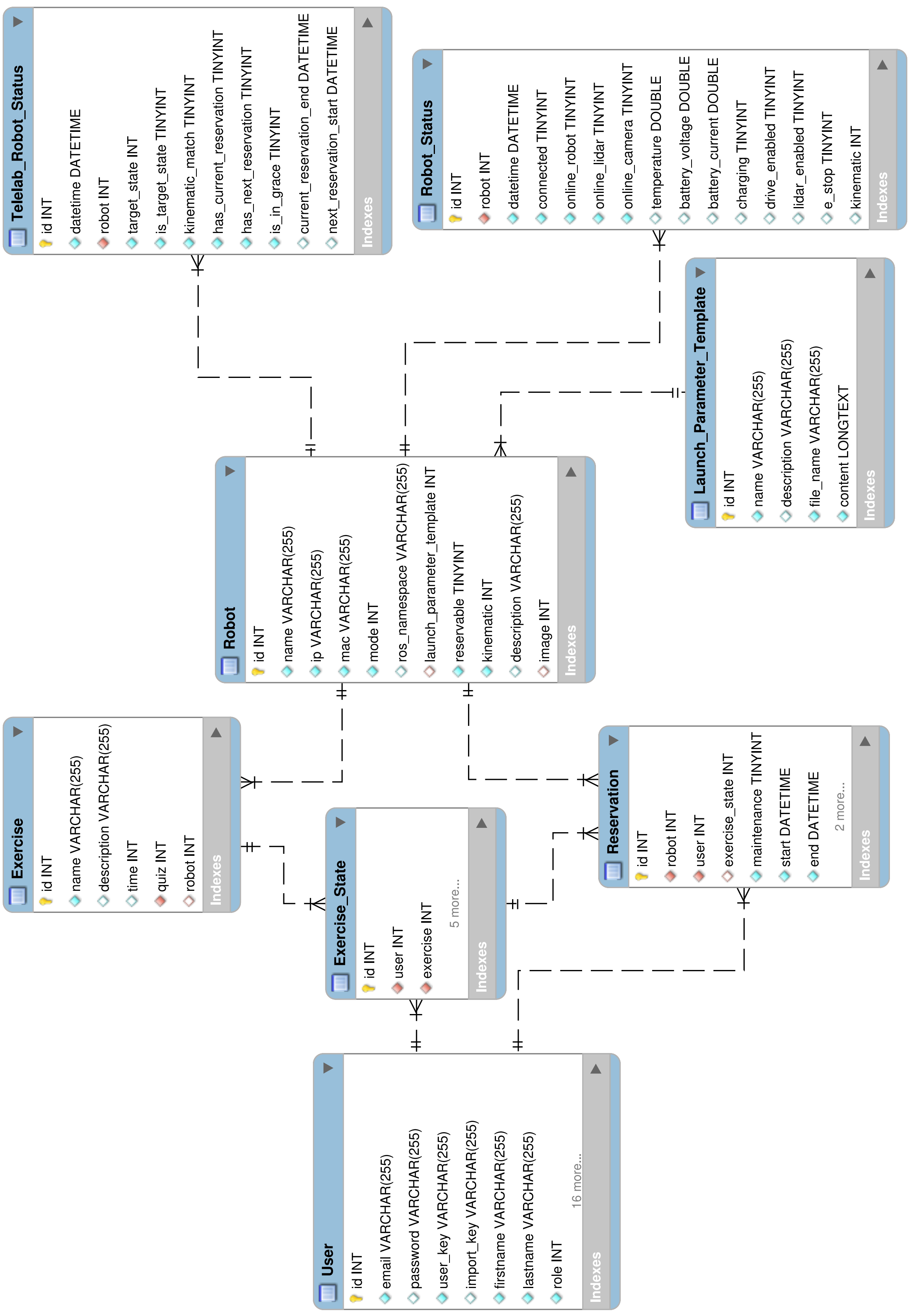


Fig. B.17: Excerpt of the database model concerning “Robot”

Appendix C

Content of Data Medium

This appendix lists the directory structure and content of this work. For the sake of clarity, not all files and directories are listed here.

```
├── Additional_Media
│   ├── Sequence_Diagrams ................Source code of sequence diagrams
│   ├── Photographs ...........................Full-resolution photographs
│   ├── ros2_connect_classes.svg ...........Class diagram of ROS2 Connect
│   ├── ros2_connect_classes_high-res.png .Class diagram of ROS2 Connect
│   └── ros2_connect_classes_low-res.png ...Class diagram of ROS2 Connect
└── Code
    ├── Index_Frontend ......JavaScript project of Telelab index frontend
    ├── PHP_Backend ........................PHP project of Telelab backend
    ├── ROS2_Additional
    │   ├── common ......Common source files of projects in this directory
    │   ├── rosbridge_websocket_bridge ....ROS2 rosbridge_websocket_bridge
    │   │                                                       source code
    │   └── time_measurement ............ROS2 time measurement source code
    ├── ROS2_Eduard
    │   ├── connect ................................ROS2 Connect source code
    │   ├── connect_plugins ..............ROS2 Connect-Plugins source code
    │   ├── edu ...........ROS2 meta-package bundling Telelab launch files
    │   ├── edu_robot .....................ROS2 edu_robot package by EduArt
    │   ├── edu_robot_control ....ROS2 edu_robot_control package by EduArt
    │   ├── sick ............................SICK TiM571 ROS2 device driver
    │   ├── telelab ................................ROS2 Telelab source code
    │   ├── telelab_companion ...........ROS2 Telelab-Companion source code
    │   ├── telelab_interfaces ........ROS2 Telelab-Interfaces definitions
    │   ├── teleop ..................................ROS2 Teleop source code
    │   └── usb_cam ....................USB camera (UVC) ROS2 device driver
    ├── Time_Measurement_Scripts .....Time Measurement Evaluation Scripts
    ├── Websocket_Comparison ........WebSocket implementation comparison
    │                                                           source code
    ├── Model.mwb .................MySQL Workbench Telelab Database Model
    └── Model.sql ...................................Telelab Database Model
```

```
├── Design ................................... Computer-Aided Design Files
├── Literature ............................... Excerpt of used literature
├── Thesis ......................... Source files of this master thesis
│   ├── data ..................................................LaTeX styles
│   ├── media ...................... Media depicted in this master thesis
│   └── plot ......................... Data plotted in this master thesis
├── Time_Measurement ..... Raw and evaluated data from time measurements
└── MasterThesis.pdf ................................. This master thesis
```

List of Abbreviations

ROS2	Robot Operating System 2	1
DDS	Data Distribution Service	1
RTPS	Real-time Publish-Subscribe Protocol	1
LAN	Local-Area-Network	1
WAN	Wide-Area-Network	1
DCPS	Data-Centric Publish-Subscribe	3
IDL	Interface Definition Language	5
QoS	Quality of Service	4
VPN	Virtual Private Network	5
SPDP	Simple Participant Discovery Protocol	7
SEDP	Simple Endpoint Discovery Protocol	7
LTS	Long Term Support	8
rmw	ROS Middleware	8
rcl	ROS Client Libraries	8

rclcpp	ROS Client Libraries for C++	8
rclpy	ROS Client Libraries for Python	8
SIGINT	Signal Interrupt	14
SIGTERM	Signal Terminate	14
IETF	Internet Engineering Task Force	15
RFC	Request for Comments	15
CV	Coefficient of Variation	23
opcode	Operation Code	15
NAT	Network Address Translation	16
AAA	Authentication, Authorization and Accounting	17
TURN	Traversal Using Relays around NAT	19
NTP	Network Time Protocol	21
PTB	Physikalisch-Technische Bundesanstalt	21
JSON	JavaScript Object Notation	29
API	Application Programming Interface	29
PPA	Personal Package Archive	38
BBR	Bottleneck Bandwidth and Round-trip propagation time	62

TOF	Time of Flight	70
CAN	Controller Area Network	70
NIC	Network Interface Controller	71
LiDAR	Light Detection and Ranging	71
UVC	USB Video Class	72
HDDM	High Definition Distance Measurement	72
NIR	near-infrared	72
RSSI	Received Signal Strength Indicator	73
ICP	Iterative Closest Point	73
CC	Creative Commons	75
BY-SA	Attribution-ShareAlike	75
URDF	Unified Robot Description Format	76
V4L	Video4Linux	77
DHCP	Dynamic Host Configuration Protocol	80
DMA	Direct Memory Access	81
WDS	Wireless Distribution System	81
ICMP	Internet Control Message Protocol	86

SLAM	Simultaneous Localization and Mapping 91

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Object Management Group. **Data Distribution Service (DDS), Version 1.4**. Technical report, Object Management Group, 9C Medway Road, PMB 274 Milford, MA 01757 USA, April 2015. URL: `https://www.omg.org/spec/DDS/1.4/PDF`.

[2] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. **Robot Operating System 2: Design, architecture, and uses in the wild**. *Science Robotics*, 7(66):eabm6074, 2022. URL: `https://www.science.org/doi/abs/10.1126/scirobotics.abm6074`.

[3] Object Management Group. **Interface Definition Language(TM), Version 4.2**. Technical report, Object Management Group, 9C Medway Road, PMB 274 Milford, MA 01757 USA, March 2018. URL: `https://www.omg.org/spec/IDL/4.2/PDF`.

[4] Object Management Group. **The Real-time Publish-Subscribe Protocol (RTPS) DDS Interoperability Wire Protocol Specification, Version 2.2**. Technical report, Object Management Group, 9C Medway Road, PMB 274 Milford, MA 01757 USA, September 2014. URL: `https://www.omg.org/spec/DDSI-RTPS/2.2/PDF`.

[5] eProsima. **Fast DDS, Version 3.2.0**. Revision from March 27, 2025. URL: `https://fast-dds.docs.eprosima.com/en/v3.2.0/index.html`.

[6] Keenan Wyrobek. **The Origin Story of ROS, the Linux of Robotics**. Revision from October 31, 2017. URL: `https://spectrum.ieee.org/the-origin-story-of-ros-the-linux-of-robotics`.

[7] Evan Ackerman and Erico Guizzo. **Wizards of ROS: Willow Garage and the Making of the Robot Operating System**. Revision from November 7, 2017. URL: `https://spectrum.ieee.org/wizards-of-ros-willow-garage-and-the-making-of-the-robot-operating-system`.

[8] kwc. **ROS 0.4 Release**. Revision from Februrary 10, 2009. URL: `https://www.ros.org/news/2009/02/ros-04-release.html`.

[9] kwc. **ROS 1.0**. Revision from January 22, 2010. URL: `https://www.ros.org/news/2010/01/ros-10.html`.

[10] Willow Garage. **Open Source Robotics Foundation**. Revision from April 16, 2012. URL: `https://web.archive.org/web/20120421060102/http://www.willowgarage.com/blog/2012/04/16/open-source-robotics-foundation`.

[11] Brian Gerkey. **ROS @ OSRF**. Revision from February 11, 2013. URL: `https://web.archive.org/web/20130217032327/https://www.osrfoundation.org/blog/ros-at-osrf.html`.

[12] Open Source Robotics Foundation, Inc. **ROS 2 Documentation: Jazzy**. Revision from March 25, 2025. URL: `https://docs.ros.org/en/jazzy/index.html`.

[13] Steven Macenski, Alberto Soragna, Michael Carroll, and Zhenpeng Ge. **Impact of ROS 2 Node Composition in Robotic Systems**. *IEEE Robotics and Autonomous Letters (RA-L)*, 2023. URL: `https://arxiv.org/abs/2305.09933`.

[14] Open Source Robotics Foundation, Inc. **ROS 2 Design**. Revision from March 25, 2025. URL: `https://design.ros2.org`.

[15] Open Source Robotics Foundation, Inc. **rclcpp: Jazzy, The ROS client library in C+++**. Revision from March 25, 2025. URL: `https://docs.ros.org/en/jazzy/p/rclcpp/index.html`.

[16] Open Source Robotics Foundation, Inc. **launch's documentation**. Revision from April 28, 2025. URL: `https://docs.ros.org/en/jazzy/p/launch/index.html`.

[17] Tully Foote. **tf: The transform library**. In *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*, pages 1–6, April 2013.

[18] Open Source Robotics Foundation, Inc. **ros2/geometry2**. Git commit 8ab7963. URL: `https://github.com/ros2/geometry2/tree/jazzy`.

[19] Ian Fette and Alexey Melnikov. **The WebSocket Protocol**. RFC 6455, December 2011. URL: `https://www.rfc-editor.org/rfc/rfc6455`.

[20] Michelle Cotton, Leo Vegoda, and David Meyer. **IANA Guidelines for IPv4 Multicast Address Assignments**. RFC 5771, March 2010. URL: `https://www.rfc-editor.org/rfc/rfc5771`.

[21] Steve Deering. **Host Extensions for IP Multicasting**. RFC 1112, August 1989. URL: `https://www.rfc-editor.org/rfc/rfc1112.html`.

[22] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Prentice Hall, Upper Saddle River, NJ, USA, 5 edition, 2010.

[23] Eclipse Foundation. **Cyclone DDS, Version 0.10.5**. Revision from August 28, 2025. URL: `https://cyclonedds.io/docs/cyclonedds/0.10.5/`.

[24] Jonathan Postel. **INTERNET PROTOCOL**. RFC 791, September 1981. URL: `https://www.rfc-editor.org/rfc/rfc791.html`.

[25] eProsima. **DDS Router, Version 3.2.0**. Revision from May 05, 2025. URL: `https://eprosima-dds-router.readthedocs.io/en/v3.2.0/`.

[26] Google LLC. **Material Symbols Rounded (icon set)**. Revision from June 10, 2025; Apache License 2.0. URL: `https://fonts.google.com/icons`.

[27] Andrew N. Novick and Michael A. Lombardi. **Practical Limitations of NTP Time Transfer**. In *2015 Joint Conference of the IEEE International Frequency Control Symposium & the European Frequency and Time Forum*, pages 570–574, 2015.

[28] David L. Mills. **Measured Performance of the Network Time Protocol in the Internet System**. RFC 1128, October 1989. URL: `https://www.rfc-editor.org/rfc/rfc1128.pdf`.

[29] Hervé Abdi. **Coefficient of Variation**. In Neil J. Salkind, editor, *Encyclopedia of Research Design*, pages 169–171. SAGE, 2010.

[30] Nicolas Malm, Kalle Ruttik, and Olav Tirkkonen. **Midhaul Performance Modelling Using Commodity Hardware C-RAN Testbed**. *Wireless Personal Communications*, 131(2):1339–1363, 2023.

[31] Srikanth Sundaresan, Yan Grunenberger, Nick Feamster, Dina Papagiannaki, Dave Levin, and Renata Teixeira. **WTF? Locating Performance Problems in Home Networks**. Technical Report GT-CS-13-03, Georgia Institute of Technology, Atlanta, GA, USA, 2013. URL: `http://hdl.handle.net/1853/46991`.

[32] Youngki Lee, Sharad Agarwal, Chris Butcher, and Jitu Padhye. **Measurement and Estimation of Network QoS among Peer Xbox 360 Game Players**. In Mark Claypool and Steve Uhlig, editors, *Passive and Active Network Measurement*, Lecture Notes in Computer Science, pages 41–50. Springer Berlin Heidelberg, 2008.

[33] Robot Web Tools. **rosbridge_suite**. Revision from May 26, 2025. URL: `https://github.com/RobotWebTools/rosbridge_suite`.

[34] Lakshminarasimhan Srinivasan, Julian Scharnagl, and Klaus Schilling. **Analysis of WebSockets as the New Age Protocol for Remote Robot Teleoperation**. *IFAC Proceedings Volumes*, 46(29):83–88, 2013. 3rd IFAC Symposium on Telematics Applications.

[35] Lakshminarasimhan Srinivasan, Julian Scharnagl, Zhihao Xu, Nicolas Faerber, Dinesh K. Babu, and Klaus Schilling. **Design and Development of a Robotic Teleoperation System using Duplex WebSockets suitable for Variable Bandwidth Networks**. *IFAC Proceedings Volumes*, 46(29):57–61, 2013. 3rd IFAC Symposium on Telematics Applications.

[36] Daniel Schott. **Encoder, Kamerabild Augmentierung und eingebettete Spriktsprache für einen teleoperierten mobilen Roboter**. Bachelor Thesis, Julius-Maximilians-Universität Würzburg, 2020.

[37] Vinnie Falco. **Boost.Beast: HTTP and WebSocket built on Boost.Asio in C++11**. Revision from June 17, 2025. URL: `https://github.com/boostorg/beast`.

[38] Peter Thorson. **WebSocket++**. Revision from June 20, 2025. URL: `https://github.com/zaphoyd/websocketpp`.

[39] Ole Christian Eidheim. **Simple-WebSocket-Server**. Revision from June 20, 2025. URL: `https://gitlab.com/eidheim/Simple-WebSocket-Server`.

[40] Open Source Robotics Foundation, Inc. **rclcpp: Kilted, The ROS client library in C+++**. Revision from July 31, 2025. URL: `https://docs.ros.org/en/kilted/p/rclcpp/index.html`.

[41] Stephen Hemminger and David Miller. **[TCP]: make cubic the default**, September 2006. Git commit 597811ec167fa01c926a0957a91d9e39baa30e64 in the *torvalds/linux* repository. URL: `https://github.com/torvalds/linux/commit/597811ec167fa01c926a0957a91d9e39baa30e64`.

[42] Lisong Xu, Sangtae Ha, Injong Rhee, Vidhi Goel, and Lars Eggert. **CUBIC for Fast and Long-Distance Networks**. RFC 9438, August 2023. URL: `https://www.rfc-editor.org/rfc/rfc9438.html`.

[43] Juniper Networks, Inc., 1133 Innovation Way, Sunnyvale, California 94089, USA. *Junos® OS Routing Policies, Firewall Filters, and Traffic Policers User Guide*, July 2025.

[44] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. **BBR: Congestion-Based Congestion Control: Measuring bottleneck bandwidth and round-trip propagation time**. *Queue*, 14(5):20–53, 2016.

[45] Wansu Pan, Xiaofeng Li, Haibo Tan, Jinlin Xu, and Xiru Li. **Improvement of RTT Fairness Problem in BBR Congestion Control Algorithm by Gamma Correction**. *Sensors*, 21(12), 2021.

[46] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. **Modeling BBR's Interactions with Loss-Based Congestion Control**. In *Proceedings of the Internet Measurement Conference*, IMC '19, pages 137–143, New York, NY, USA, 2019. Association for Computing Machinery.

[47] EduArt Robotik GmbH. **R&D platform "Eduard"**. Revision from September 02, 2025. URL: `https://eduart-robotik.de/products/rd-platform-eduard/`.

[48] EduArt Robotik GmbH. **edu_robot - Control Software for IoT Shield, Ethernet Gateway and RaspberryPi Stack**. Revision from September 02, 2025. URL: `https://github.com/EduArt-Robotik/edu_robot`.

[49] Jingang Yi, Hongpeng Wang, Junjie Zhang, Dezhen Song, Suhada Jayasuriya, and Jingtai Liu. **Kinematic Modeling and Analysis of Skid-Steered Mobile Robots With Applications to Low-Cost Inertial-Measurement-Unit-Based Motion Estimation**. *IEEE Transactions on Robotics*, 25(5):1087–1097, 2009.

[50] Dr. Christian Herrmann and Prof. Dr. Klaus Schilling. **Robotics I Mobile Robots - Kinemtics**. Lecture script of Robotics I, 2022.

[51] Nurallah Ghaeminezhad Hamid Taheri, Bing Qiao. **Kinematic Model of a Four Mecanum Wheeled Mobile Robot**. *International Journal of Computer Applications*, 113(3):6–9, 2015.

[52] EduArt Robotik GmbH. **edu_drive_ros2**. Revision from September 02, 2025. URL: `https://github.com/EduArt-Robotik/edu_drive_ros2`.

[53] e-con Systems India Pvt Ltd. *See3CAM_24CUG Datasheet*, November 2024. Revision 1.6.

[54] e-con Systems India Pvt Ltd. *See3CAM_24CUG Lens Datasheet*, November 2023. Revision 1.5.

[55] SICK AG. *TiM55x/56x/57x/58x OPERATING INSTRUCTIONS*, September 2024. URL: `https://www.sick.com/media/docs/2/02/602/operating_instructions_tim55x_56x_57x_58x_2d_lidar_sensors_en_im0097602.pdf`.

[56] SICK AG. *TiM571-2050101 Produktdatenblatt*, August 2025. URL: `https://www.sick.com/media/pdf/4/44/444/dataSheet_TiM571-2050101_1075091_de.pdf`.

[57] TP-Link Technologies Co., Ltd. *TL-WR802N V4 Datasheet*, October 2020. URL: `https://static.tp-link.com/2020/202010/20201019/TL-WR802N(EU&US)%204.0-datasheet.pdf`.

[58] GL Technologies (Hong Kong) Limited. *GL-MT3000 (Beryl AX) Datasheet*, March 2025. URL: `https://static.gl-inet.com/www/images/products/datasheet/mt3000_datasheet_20250320.pdf`.

[59] dunkndonuts. **GL-INET-BERYL-AX-HOLSTER**. `https://www.thingiverse.com/thing:6579293`, April 2024. Licensed under Creative Commons Attribution-ShareAlike 3.0 (CC BY-SA 3.0) `https://creativecommons.org/licenses/by-sa/3.0/`.

[60] EduArt Robotik GmbH. **EduArt Robotik**. Revision from September 9, 2025. URL: `https://github.com/EduArt-Robotik`.

[61] Open Source Robotics Foundation, Inc. **Robot State Publisher**. Revision from September 10, 2025. URL: `https://github.com/ros/robot_state_publisher`.

[62] Open Source Robotics Foundation, Inc. **image_transport**. Revision from September 10, 2025. URL: `https://wiki.ros.org/image_transport`.

[63] SICK AG. **sick_scan_xd**. Revision from September 11, 2025. URL: `https://github.com/SICKAG/sick_scan_xd`.

[64] SICK AG. *Telegram Listing Ranging sensors LMS1xx, LMS5xx, TiM2xx, TiM5xx, TiM7xx, LMS1000, MRS1000, MRS6000, NAV310, LD-OEM15xx, LD-LRS36xx, LMS4000, LRS4000, multiScan*, February 2023. URL: `https://www.sick.com/media/docs/7/27/927/telegram_listing_telegram_listing_ranging_sensors_lms1xx_lms5xx_tim2xx_tim5xx_tim7xx_lms1000_mrs1000_mrs6000_nav310_ld_oem15xx_ld_lrs36xx_lms4000_lrs4000_multiscan100_picoscan100_en_im0045927.pdf`.

[65] USB Implementers Forum, Inc. **Universal serial bus specification revision 2.0**. Technical report, USB Implementers Forum, Inc., April 2000. URL: `https://www.usb.org/document-library/usb-20-specification`.

[66] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 3 edition, 2005.

[67] IEEE SA. **IEEE Standard for Information Technology - Telecommunications and information exchange between systems - Local and Metropolitan Area networks - Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications**. *ANSI/IEEE Std 802.11, 1999 Edition (R2003)*, November 1998.

[68] Darwin Engwer. **"WDS" Clarifications, IEEE P802.11 Wireless LANs**. Technical Report IEEE 802.11-05/0710r0, IEEE 802.11 TGm Group, July 2005.

[69] OpenWrt Project Community. **Wi-Fi Extender/Repeater with WDS**. Revision from September 17, 2025. URL: `https://openwrt.org/docs/guide-user/network/wifi/wifiextenders/wds`.

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Versicherung zur selbständigen Leistungserbringung

Titel der Masterarbeit:
ROS2 Over Wide-Area-Networks: Teleoperation of Mobile Robots in an E-Learning Environment

Thema bereitgestellt von (Titel, Vorname, Nachname, Lehrstuhl):
Prof. Dr. Andreas Nüchter, Informatik XVII

Eingereicht durch (Vorname, Nachname, Matrikel):
Daniel Schott, 2172200

Ich versichere, dass ich die vorstehende schriftliche Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die benutzte Literatur sowie sonstige Hilfsquellen sind vollständig angegeben. Wörtlich oder dem Sinne nach dem Schrifttum oder dem Internet entnommene Stellen sind unter Angabe der Quelle kenntlich gemacht.

Weitere Personen waren an der geistigen Leistung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich nicht die Hilfe eines Ghostwriters oder einer Ghostwriting-Agentur in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar Geld oder geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Arbeit stehen.

☐ Mit dem Prüfungsleiter bzw. der Prüfungsleiterin wurde abgestimmt, dass für die Erstellung der vorgelegten schriftlichen Arbeit Chatbots (insbesondere ChatGPT) bzw. allgemein solche Programme, die anstelle meiner Person die Aufgabenstellung der Prüfung bzw. Teile derselben bearbeiten könnten, entsprechend den Vorgaben der Prüfungsleiterin bzw. des Prüfungsleiters eingesetzt wurden. Die mittels Chatbots erstellten Passagen sind als solche gekennzeichnet.

Der Durchführung einer elektronischen Plagiatsprüfung stimme ich hiermit zu. Die eingereichte elektronische Fassung der Arbeit ist vollständig. Mir ist bewusst, dass nachträgliche Ergänzungen ausgeschlossen sind.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Ich bin mir bewusst, dass eine unwahre Erklärung zur Versicherung der selbstständigen Leistungserbringung rechtliche Folgen haben kann.

Würzburg, 30.09.2025, D. Schott
Ort, Datum, Unterschrift