Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for ingeniørvitenskap
Institutt for maskinteknikk og produksjon

Morten Melby Dahl

Expanding the ROS2 communication architecture.

Data bridging by utilization of network sockets.

Masteroppgave i Produktutvikling og Produksjon
Veileder: Lars Tingelstad
Juni 2021

NTNU
Norwegian University of
Science and Technology

Morten Melby Dahl

Expanding the ROS2 communication architecture.

Data bridging by utilization of network sockets.

Masteroppgave i Produktutvikling og Produksjon
Veileder: Lars Tingelstad
Juni 2021

Norges teknisk-naturvitenskapelige universitet
Fakultet for ingeniørvitenskap
Institutt for maskinteknikk og produksjon

Preface

This thesis completes my master's degree in Mechanical Engineering at the Norwegian University of Science and Technology.

Having completed a pre-project that involved implementing a mobile KUKA KMR iiwa robot with the Robotic Operating System (ROS2), the theme of multi-robot control became a subject of interest. During research on implementing robotic fleets in ROS2, a discussion on this exact topic was found with no solution. Doing further research, it appeared that solving the issues that allowed for robotic fleet control in ROS2 would also open up for layering communication, enabling the development of robotic fleet control and control system architectures that correspond to commonly used Distributed Control Systems.

The writing in this thesis assumes the reader possesses basic knowledge of robotics, programming, and computer networking. It is also advantageous to have knowledge about the robotic operating system ROS2.

Summary

The thesis investigates how the robotic operating system, ROS2, can be used as a factory-wide communication architecture and development platform by including the ability to separate different communication channels utilizing built-in features and socket programming. To do this, a stack named *ros2_socket_bridge* is developed using the combination of socket and ROS2 programming.

It is desired to have the program be scenario-independent and work for any task that the user desires. This includes the ability to process real-time data with minimal delay. In addition, the ability to connect devices using Bluetooth adds to the overall usability.

After a working prototype was developed, improvements were implemented to make the stack usable for large-scale information layering and robot control. Practical and performance testing was done to uncover flaws in the software, and finally, ros2_socket_bridge is compared to a different project with similar goals.

Sammendrag

Denne masteroppaven undersøker hvordan et operativsystem for roboter, ROS2, kan bli brukt til å styre hovedkommunikasjonen i moderne fabrikker. Dette gjøres ved å implementere mulighet for å lage forskjellige kommunikasjonsnivåer ved kobinert bruk av innebyggde ROS2 funksjoner og socket programmering. Programvaren blir navngitt *ros2_socket_bridge*.

Det er ønskelig at programvaren kan brukes i mange ulike oppsett, og derved er fleksibelt. Det må ha muligheten til å prossesere sanntidsinformasjon uten tillegg av betydelig forsinkelse forbundet med prosseseringstid.

En prototype ble utviklet og testet hvorav flere funksjoner ble forbedret og implementert. Eksempelvis ble Bluetooth implementert som et mulig kommunikasjonsmedium. Testing ble gjort i både praktisk- og ytelsessammenheng, og til slutt ble ros2_socket_bridge sammenliknet med et prosjekt som har liknende mål.

Contents

Preface i

Summary iii

Sammendrag v

1. Introduction 1
- 1.1. Control systems in Industry 4.0 . . . 1
- 1.2. Robotics vendors and factory connectivity . . . 2
- 1.3. ROS2 . . . 2
  - 1.3.1. Data Distribution Service . . . 3
  - 1.3.2. RTPS protocol . . . 4
  - 1.3.3. ROS2 implementation of DDS . . . 4
- 1.4. Problem introduction . . . 5
- 1.5. Related work . . . 5
  - 1.5.1. A ROS2 based communication architecture for control in collaborative and intelligent automation systems . . . 5
  - 1.5.2. Robotics Middleware Framework . . . 6
  - 1.5.3. Free_fleet . . . 6
  - 1.5.4. ros2/domain_bridge . . . 7
- 1.6. Problem definition . . . 7

2. Method 9
- 2.1. Design considerations . . . 9
  - 2.1.1. Topic visibility restrictions . . . 9
  - 2.1.2. Network security . . . 12
  - 2.1.3. Adaptability . . . 13
- 2.2. Development . . . 14
  - 2.2.1. Version 1 - Primitive robot-specific server-client program . . . 14
  - 2.2.2. Version 2 - Over-advanced topic streaming setup . . . 15
  - 2.2.3. Version 3 - Generalized version with serializer and cryptography . . . 15
  - 2.2.4. Version 4 - Unproven working program . . . 16

2.2.5. Version 5 - Proven working program 16
2.2.6. Version 6 - Improved program with shutdown handling and Bluetooth 16
2.3. Software architecture 17
2.3.1. User setup 18
2.3.2. Server-client relationship 22
2.3.3. Connection modes 22
2.3.4. Encryption key generation 25
2.3.5. Bluetooth channel checker 26
2.4. Performance experiments 26
2.4.1. Setup 27
2.4.2. Performance considerations 30
2.4.3. Result interpretation 30

3. Results 33
3.1. Robot simulation dataset 33
3.2. Message processing time 33
3.2.1. Serialization and encryption 34
3.2.2. Decryption and deserialization 37
3.3. Processing time versus message size 39
3.3.1. Simulated robot 40
3.3.2. Controlled publisher 41
3.4. Maximum publishing rate versus message size 42
3.4.1. Internal communication tests 42
3.4.2. External communication tests 45
3.4.3. Summary 48

4. Discussion 49
4.1. Benchmarking 49
4.1.1. Measuring method 49
4.1.2. Maximum publishing rates 50
4.2. Usability 51
4.2.1. Internal and local network transmissions 51
4.2.2. Bluetooth transmission 51
4.2.3. Software setup 51
4.3. Further work 51
4.3.1. Optimization of initialization message and callback function 51
4.3.2. Integration of services and actions 52
4.3.3. Remove the need to set specific topic sockets 53
4.3.4. Test the viability of low-powered devices 53

5. Conclusion **55**
5.1. Comparison to 'ros/domain_bridge' 55
5.1.1. Communication method 55
5.1.2. Pros and cons 56
5.1.3. Significance in practice 56

A. ReadTheDocs **59**
A.1. Front Page 60
A.2. Quick-start guide 61
A.3. Internal workings 63
A.4. Testing and benchmaking 65
A.5. License 69

List of Figures

1.1. Hierarchical layout of a DCS. 2
1.2. DDS communication architecture 3
1.3. How ROS2 implements the DDS communication architecture. . . . 4

2.1. Comparison of UDP and TCP headers 11
2.2. Three different use cases using the same software. 14
2.3. How the software is initialized. 18
2.4. Server and client booting information 20
2.5. Hierarchical DCS setup using ros2_socket_bridge. 22
2.6. Server-client local connection modes 23
2.7. External server-client connection across the internet. 24
2.8. Bluetooth connection mode. 25
2.9. Message processing bottlenecks 26
2.10. Performance experiment setup using a simulated robot 28
2.11. Performance experiment setup using a custom publisher with set data size and rate 29
2.12. Performance experiment setup using Bluetooth as a communication medium. 29

3.1. Results gathered form the time it took to serialize and encrypt different messages. 35
3.2. Results gathered form the time it took to decrypt and deserialize different messages 38
3.3. Packing time versus message size for messages used by the simulated robot 40
3.4. Packing time versus message size using a custom publisher at a publishing rate of 50Hz 41
3.5. Internal transmission of messages using UDP 43
3.6. Internal transmission of messages using TCP 44
3.7. Local network transmission of messages using UDP 45
3.8. Local network transmission of messages using TCP 46
3.9. Wireless transmission of messages using Bluetooth 47

4.1. Two nodes connected to the same service node. 52
4.2. Two nodes communicating using an action. 53

List of Tables

2.1. Variables available to the user. 21
2.2. Hardware information of the computers used for experiments . . . 30

3.1. Sample information gathered from transferring messages. 33
3.2. Results gathered form the time it took to serialize and encrypt different messages. 36
3.3. Results gathered form the time it took to decrypt and deserialize different messages. 39
3.4. A summary of peak publishing speeds for all message sizes and protocols. 48

Chapter 1.

Introduction

This chapter intends to introduce the reader to Industry 4.0, control system architecture, the Robot Operating System 2, and its advantages and disadvantages. Once a general understanding is achieved, the problems which this thesis intends to solve are introduced. Related work is presented, and a solution is proposed.

1.1. Control systems in Industry 4.0

The ongoing change in industry, known as the fourth industrial revolution, changes how factories are operated. Cyber-physical production systems are becoming the norm of modern manufacturing [1], where digitization and the Internet of Things (IoT) are at the center of how factories are constructed and controlled. Labor-intensive production line jobs are being equipped with more efficient robots, which cooperate by communicating over the IoT to continuously report the state of different processes to a centralized hub [2]. The system can be connected to other factories over the internet, bringing logistics directly into the manufacturing process.

The control of factories using IoT is usually done using Distributed Control Systems (DCS). A DCS is made up of digital field busses and decentralized control computers, assisted by a central control computer for monitoring and controlling [3]. The DCS is commonly split into five levels [4], as shown in Figure 1.1. Using this architecture allows for simple implementation and removal of robots, microcontrollers (μC), and other hardware in the control system by simply modifying behavior trees based on available units.

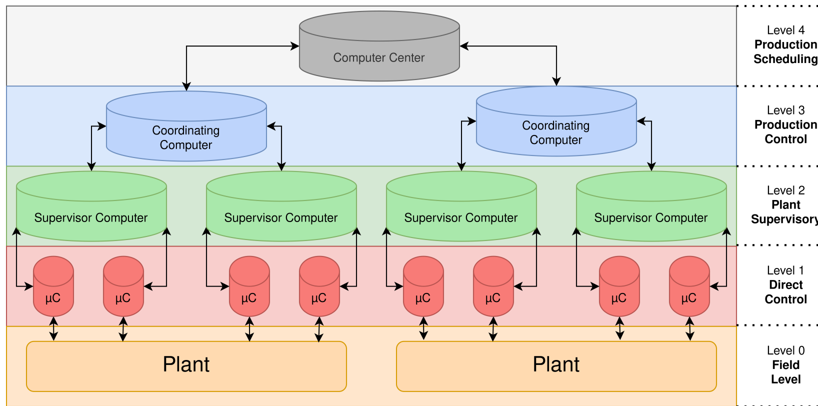


Figure 1.1.: Hierarchical layout of a DCS.

1.2. Robotics vendors and factory connectivity

To fulfill the requirements of an industry 4.0 factory, a communication system that connects different robotic and control systems are required. Common robotics vendors such as KUKA, ABB, Univeral Robots, and others often come with their own operating systems and different ways of connecting to the IoT. Buying robots from a single vendor brings advantages as systems interface with each other [5], but it leaves the factory with the disadvantage of a vendor lock-in. A way of leaving this vendor lock is to utilize the open-source and free robotics middleware, Robotic Operating System 2 (ROS2).

1.3. ROS2

As ROS2 is open-source, there is no vendor lock-in, and the applications are multi-domain and multi-platform. It also features multiple community-made stacks for interfacing with robots of the aforementioned vendors. This provides a sturdy development platform for many industrial applications, although the development of ROS2 is still in its infancy. The predecessor of ROS2 is ROS(1), which had the same intentions for being an open-source robotics software, but did not use the same internal communication architecture [6]. Programs running on ROS1 and ROS2 cannot cross-communicate with each other unless a package named ‘ROS1_bridge’ is utilized. ROS2 is updated in distributions with names where the first letter is correlated to the release version. The latest released version of ROS2 is ‘Foxy Fitzroy’, with ‘Galactic Geochelone’ being next in line.

Using ROS2 allows for the use of open-source stacks such as Navigation2 which

allow for robot navigation [7], while MoveIt2 provides advanced kinematics, motion planning, and collision checking to manipulator robots [8]. Additional stacks allow for Simultaneous localization and mapping (SLAM), integration of machine learning algorithms, and much more.

The communication architecture of ROS2 is built on the Data Distribution Service (DDS) standard.

1.3.1. Data Distribution Service

DDS is an industry-standard developed by the Object Management Group for use as a Data-Centric Publish-Subscribe model [9]. DDS builds upon complex network communication, allowing the users to take advantage of the advanced architecture in a simple, user-friendly way. Multiple vendors have built software to comply with the DDS standard, such as RTI, ADLINK Technologies, Twin Oaks Software, and eProxima. The user decides the spesific DDS software to be used by ROS2.

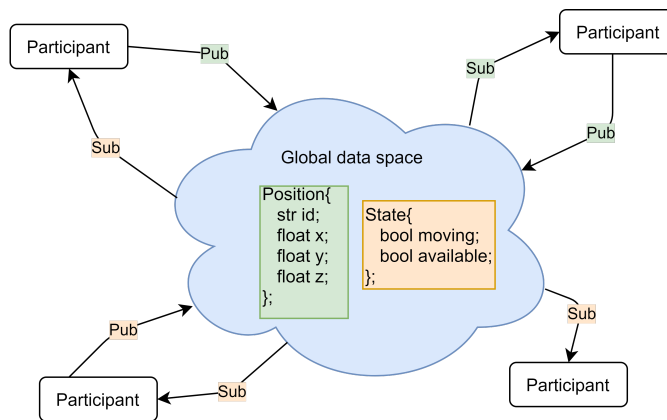


Figure 1.2.: DDS communication architecture

Figure 1.2 shows a simplified model of the publish-subscribe communication architecture used by DDS. Multiple participants can communicate with each other over topics, which are denoted by the arrows. The arrows lead to the global data space, which is an area of the network where the topics are hosted. Each topic can only have a single message type, as exemplified by the `Position` and `State` messages.

1.3.2. RTPS protocol

DDS utilizes a Real-Time Public-Subscribe (RTPS) protocol. This protocol uses the UDP (User Datagram Protocol) as a backbone, which is one of the core members of the internet protocol suite. UDP is a simple protocol where messages are sent without any confirmation algorithm to ensure that the receiver has received the data, leading to low reliability and high speed. By adding what DDS calls Quality of Service (QoS), the RTPS protocol allows the user to adjust reliability settings for their specific program [10]. For faster transmission with low latency and reliability requirements, QoS is set to simulate UDP. If reliability is a requirement above speed, the protocol can be adjusted to simulate TCP (Transmission Control Protocol), which features a triple handshake to ensure data integrity. QoS essentially allows the user to decide what level of reliability to speed is wanted for the current information stream.

1.3.3. ROS2 implementation of DDS

The ROS2 implementation of DDS changes some of the names previously used. In Figure 1.2, 'participant' has been renamed 'node'. The 'global data space' will be known as the 'ROS domain'. The domain is set by using an assigned ID, `ROS_DOMAIN_ID`. This ID is set in the terminal before launching or running ROS2 programs. Nodes running in terminals with the same ID can discover each other, even though the programs run on different computers. The only requirement is that they are connected to the same network that has UDP multicasting enabled. An example of how ROS2 implements the DDS model is shown in Figure 1.3.

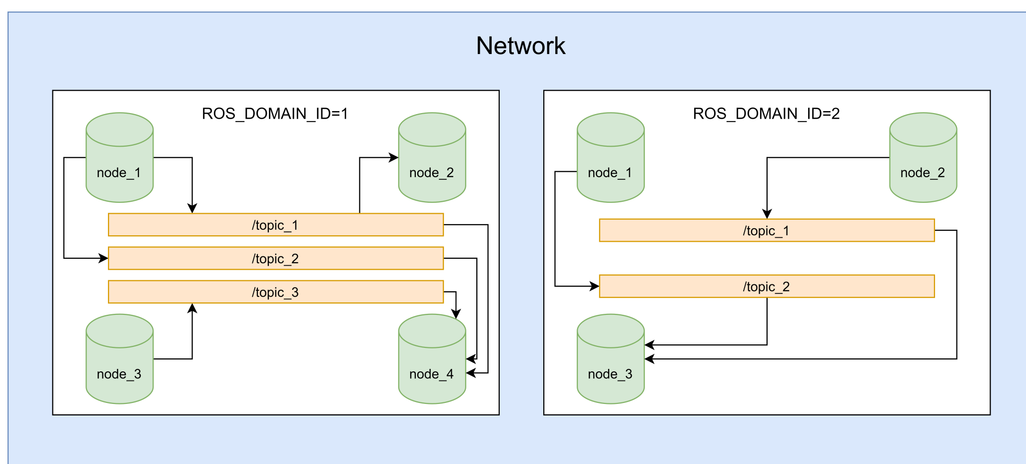


Figure 1.3.: How ROS2 implements the DDS communication architecture.

1.4. Problem introduction

The DDS method of communicating is excellent when there is a single robot being controlled, but problems arise when multiple robots or sensors of similar nature are present on the same ROS domain. By default, the namespace of topics that contain odometry, laser-scanner data, joint positions, and frame transforms are the same. Changing the namespace of topics manually could cause confusion and clutter. This is why the subject of robotic fleet control has been a theme of discussion in the ROS2 community for years [11]. This also limits the use of ROS2 in a DCS factory setup, as there is no way of layering information while simultaneously transferring specific data between the layers.

The information clutter problem is a restriction when implementing multiple robots or sensors in a factory setup. However, this problem is not sourced in the use of DDS. One of the features of DDS is the ability to assign partitions as a part of the QoS settings [12]. Partitions are simply a second layer added on top of the global data space (which is shown in Figure 1.2). Partitions allow participants to subscribe to topics with an additional layer setting, solving namespace clutter problems. The problem with using partitions comes from the fact that it is in use by lower levels of ROS2 DDS, and is not available to users.

Making direct changes to how ROS2 implements DDS is complex and comes with a multitude of possible issues. The compatibility of software between different DDS vendors is already a restriction in ROS2. Making these changes would induce multiple problems, not only with DDS compatibility but also with already existing packages.

Instead of making internal software changes, a new ROS-based program could add to the communication layer and provide ways to sort the network. This would solve the main challenge that ROS2 has to overcome to be used as a DCS.

1.5. Related work

1.5.1. A ROS2 based communication architecture for control in collaborative and intelligent automation systems

Erős, Dahl, Bengtsson, *et al.* [13] proposed a factory implementation of what they called ROS hubs. Each hub is a singular robot running ROS1 or ROS2 with its accompanying communication node, which communicates robot states to a centralized controller. Their design mainly focused on implementing specific robots in a network running both ROS1 and ROS2 to interface with Sequence Planner [14]. The Sequence Planner software is made to model operations and sequences, controlling the planned sequences with cooperation from an operator.

The article does not mention the use of different domains and how the nodes are set up. The only description of overall system architecture is the fact that additional sorting nodes are created. These nodes take information from each 'hub', and sort it according to state. This state information is subscribed to by the sequence planner software, which then creates a command message. This command message is parsed by a node running on the 'hub', which is then converted to robot movement or state. There are no set communication boundaries except for the use of ROS1 bridges.

The idea of making robot hubs that connect to a central node is optimal in the form of reducing clutter and having only vital information streams to the global planner. Each 'hub' can be complex in its internal workings, but the information is communicated in a readable way for all systems involved. This article focuses on the integration of specific software, which makes the general application of their system harder.

1.5.2. Robotics Middleware Framework

The Robotics Middleware Framework (RMF) is an application developed to manage mobile robot fleets in ROS2. It does so by creating paths on a map where robots can traverse, integrating door and elevator control. A simple controller which converts paths to velocity movements of individual robots provides basic navigation. This navigation is very basic and lacks features such as SLAM and real-time path planning.

This is a great software to control robot fleets for navigating large spaces and executing pick-and-place tasks. This software currently has no way of implementing manipulator robots and does not currently allow for implementing custom pathfinding algorithms or costmaps. It is still under heavy development and has currently not been applied outside simulations.

1.5.3. Free_fleet

The free_fleet software is developed to control multiple robots using the old ROS1 'Navigation' path planning. Robot states are communicated over their own topics using special messages. These messages are transferred to a server running ROS2 and RViz. Goal poses are sent from the ROS2 server to the individual clients which handle them using 'Navigation'.

This solves the cluttered namespace problem by having robots running on ROS1 and only transferring the customized robot state messages. The downside is that additional robots have to be implemented on ROS1, as the client software has not been implemented for ROS2. This is also only a solution for the specific

goal of navigating multiple mobile robots, and does not provide any flexibility for different processes. In addition, having to use the older ROS1 software also has limitations in future-proofing and development.

1.5.4. ros2/domain_bridge

The domain_bridge stack saw its release on GitHub in late February, at the same time as the work on this thesis started. As the name implies, it is developed with the main intention of transmitting messages across domains within the network. Internally, the program takes in a ”from” and ”to” domain. In each domain, a node is created. In the domain where messages are transmitted to, a publisher is created. In the ”from” domain, a subscriber is created within the node. The callback function of this subscriber is to publish the same message on the publisher in the ”to” domain.

This is a seemingly smart and convenient way of transmitting topics across domains, which allows for the separation of data, as this thesis also intends. This package is created for the ‘Galactic’ version of ROS2 and later. As it is newly created and is for a yet-to-be-released version of ROS2, it has not seen much use. The ”bridge” is limited to topics on the local network.

1.6. Problem definition

In order to make ROS2 applicable to being a large part of a factory operating system, a better communication architecture has to be established. By providing developers with the ability to create multiple communication layers, developing more advanced ROS2-based control systems becomes practical.

The goal of this thesis will be;

- Make ROS2 more applicable as a factory operating system by creating a stack which allows users to order data in a simple, flexible and ordered manner.
- Make the stack based on ROS2 programming features such as publishers and subscribers, supported by features enabled by socket programming.

Chapter 2.

Method

This chapter will explain the process and considerations taken when implementing a communication architecture on top of the already existing systems in ROS2. To complement topics of discussion, theory is weaved in to give the reader insight into why certain choices are made.

2.1. Design considerations

Designing a communication architecture to interface any type of robot or software comes with multiple design features that should be considered.

One of the most important factors when implementing a system that allows every possible ROS2 program to be plugged in is simplicity and flexibility. The system should stay as close to the ROS2 architecture as possible by utilizing domains, messages, subscribers, and publishers. As the software is created in the form of ROS2 nodes, developers can implement the nodes in their launch files to automatically start transferring topics.

2.1.1. Topic visibility restrictions

Having multiple robots and control systems implemented in a factory without restricting the number of visible topics would increase the amount of work required when implementing new systems. As an example - simulating a single Turtlebot3 robot and making it navigate using Navigation2 will create a total of 58 topics. Adding additional mobile robots on the same domain would require renaming all these topics not to confuse the navigation software.

The simplest way to avoid this problem would be to utilize the domain assignation using ROS_DOMAIN_ID to restrict robots and their control computer to the

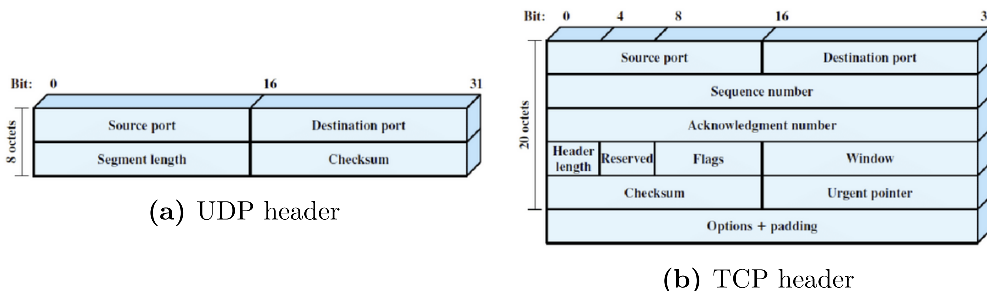
same domain. This brings with it its own issue - how does one allow specific topics to exist outside the domain without using partitions? A simple solution would be to utilize the same method that DDS uses to send and receive data - sockets.

Socket programming

Socket programming opens methods for computers to establish communication without the use of physical connections. As the name implies, a socket is a point of connection. Programs running on the same or different computers can communicate by sending information to a specific port and IP address, often called just an address when put together. Ports are identified by a port number, which can be anything between 0 and 65535. Picking which port to use for communication should have some considerations taken into account. Ports 0 to 1023 are used for known services, 1024 to 49151, known as registered ports, are registered for use through applications signed by the Internet Assigned Numbers Authority (IANA). Ports 49152-65535 are assigned as Dynamic/Private, having no assigned use case [15]. Keeping this in mind, using dynamic ports for self-programmed software is the least likely to already be bound by running processes. Assigned ports can be used as long as the computer in question is not running any of the applications assigned. A list of which applications are registered to specific ports can be easily found on Wikipedia.

Knowing what a port is, determining what protocol to use is also an important consideration. Of all existing transport-layer protocols, TCP and UDP are the most commonly used [15]. Message headers always contain an address, which is a combination of the receiving computers IP address and the port where the message is sent to. Message headers using UDP contain the address, in combination with segment length and checksum information (as shown in Figure 2.1a). There is no way to ensure that the message sent arrived at its intended destination or arrived intact. If reliability is a concern, TCP provides a way of ensuring that the receiver has received the whole and intact message. This is done by a three-way handshake. The message to be sent has a sequence number which is a number used to track the data contents of the message. Once the receiver gets the message, it generates a sequence number and sends it back. If the sequence number matches, the original sender confirms that the message is correct [16]. Of course, the disadvantage is that this takes more time than using a simple one-way UDP message.

Selecting protocol is dependent on the nature of the data which is to be sent. For cases where speed is prioritized over reliability, UDP is preferred. If reliability is essential, TCP should be used. Reliability comes at the cost of speed and header size, as shown in Figure 2.1.



(a) UDP header

(b) TCP header

Figure 2.1.: Comparison of UDP and TCP headers. Made by Saha [17], used with permission.

The protocol act as a wrapper around the message, ensuring that it gets sent to the correct destination. The message itself is a byte object. Bytes are what computers use to store information about characters, and they need to be decoded to be readable. That also means that data that is to be sent needs to be encoded into bytes. To encode objects which are more complex than strings, we need to perform serialization.

Serialization

In order to send items using sockets, the data has to be sent as a *byte object*. Converting data to bytes is known as serialization. Using Python, the conversion from string to byte object is simple;

```
>>> msg = 'Hello, world!'.encode('utf-8')
>>> msg
b'Hello, world!'
```

The encoding language is data-specific. MP3 is commonly used for music, MP4 for video, and UTF-8 for standard text. For encoding more uncommon letters, for example Arabic, UTF-16 would be the most space-saving encoding system.

Serializing and encoding strings is one thing, but objects and other data types require more intricate methods. Two common packages used for serialization in Python is ‘JSON’ and ‘pickle’. JSON, or ‘JavaScript Object Notation’, converts standard objects into a human-readable text format while pickle converts the object to a byte string. Here is an example where a list is encoded;

```
>>> import json, pickle
>>> data = [1,2,3]
>>> json_example = json.dumps(data)
>>> pickle_example = pickle.dumps(data)
>>> json_example
'[1, 2, 3]'
>>> pickle_example
b'\x80\x04\x95\x0b\x00\x00\x00\x00\x00\x00\x00]\x94(K\x01K\x02K\x03e.'
```

Using JSON is faster, gives human-readable data, and has no security loopholes, but misses one vital feature. It is limited to standard Python objects, such as lists, strings, libraries, etc. Pickle has no limitation and can serialize any Python object [18]. Unfortunately, pickle has a security flaw where a malicious actor could run Python code during de-serialization [19]. This means that any messages received over the internet to be de-serialized have to come from a trusted source.

2.1.2. Network security

As the software will be developed to be usable across networks and in factories, network security is vital. As serialization opens up the possibility of a malicious actor passing in code, we need to ensure that a trusted system sends the data we are receiving. To achieve this, encryption will be implemented by using the ‘cryptography’ Python package.

Cryptography

Cryptography is essentially the distortion of data using a key that only the user knows about. If you do not have the key, it is virtually impossible to read or create data that is encrypted in the same manner. The Python package ‘Cryptography’ has many tools to manage encryption and decryption, allowing for safe transportation of messages.

More specifically, the Fernet function is used. It utilizes the Advanced Encryption Standard (AES), which is known for its speed and security. AES is used in Cipher Block Chaining mode, which is essentially a way of converting blocks of plaintext into ciphertext using a key [20]. Each block consists of 128, 192 or 256 bytes of plaintext, sourced from the byte object we want to encrypt. The bytes are placed in a matrix format, so that we can move around values and perform matrix operations in the encryption process.

To simply explain the process, each byte is replaced by values in a lookup table. Once this is done, all the column values are mixed around. Next, all the rows are converted to vectors and multiplied by a set matrix. Now that the text is sorted in a seemingly random way, a ‘round-key’ is added. This is done for all of

the plaintext blocks, and we end up with a seemingly random ciphertext. All of these randomization actions are sourced in an encryption key. These operations provide non-linearity to the ciphertext, making it practically impossible to invent a mathematical function that decrypts the message. The only way to reverse the encryption process is to acquire the key used and precisely follow the encryption process backward. Here is an example of what an encrypted object looks like;

```
>>> from cryptography.fernet import Fernet
>>> key = Fernet.generate_key()
>>> f = Fernet(key)
>>> msg = b'Super secret message'
>>> msg_encrypted = f.encrypt(msg)
>>> msg_encrypted
b'gAAAAABgUKlooXlptV1hIovyl2wct4JwDRMI-rNyj58jg89inHTDdWfnUXJG_-bGFoDAcOAVBgzfAs
↪  2BKte4KoMW_JKi2qeOCnu19-AvADQ5v-2l42mcLnY='
>>> msg_decrypted = f.decrypt(msg_encrypted)
>>> msg_decrypted
b'Super secret message'
```

Not only does encryption prevent anyone without the key from reading messages, but it also prevents messages generated using the wrong key from getting through the decryption function. Here is an example where 'malicious code' is encrypted using the wrong key;

```
>>> from cryptography.fernet import Fernet
>>> real_key = Fernet.generate_key()
>>> fake_key = Fernet.generate_key()
>>> f_real = Fernet(real_key)
>>> f_fake = Fernet(fake_key)
>>> malicious_code_encrypted = f_fake.encrypt(b'Malicious code')
>>> received_msg = malicious_code_encrypted
>>> received_msg_decrypted = f_real.decrypt(received_msg)
raise Exception(cryptography.fernet.InvalidToken)
```

If a malicious actor sends a message which is either encrypted using the wrong key or not encrypted at all, an error is raised, and the malicious message does not get through to the next steps in the code. This will keep any malicious code from being deserialized by pickle.

2.1.3. Adaptability

To be used as a basis for a large-scale control system of robot fleets amongst other uses, the software must be adaptable. This would include the ability to create a big network between different domains in any way the user desires. Changing a few settings would provide the user with a range of use-cases. Three different examples of use cases are shown in Figure 2.2. The only difference between the three cases is the amount of server-node pairs connected and topics transferred.

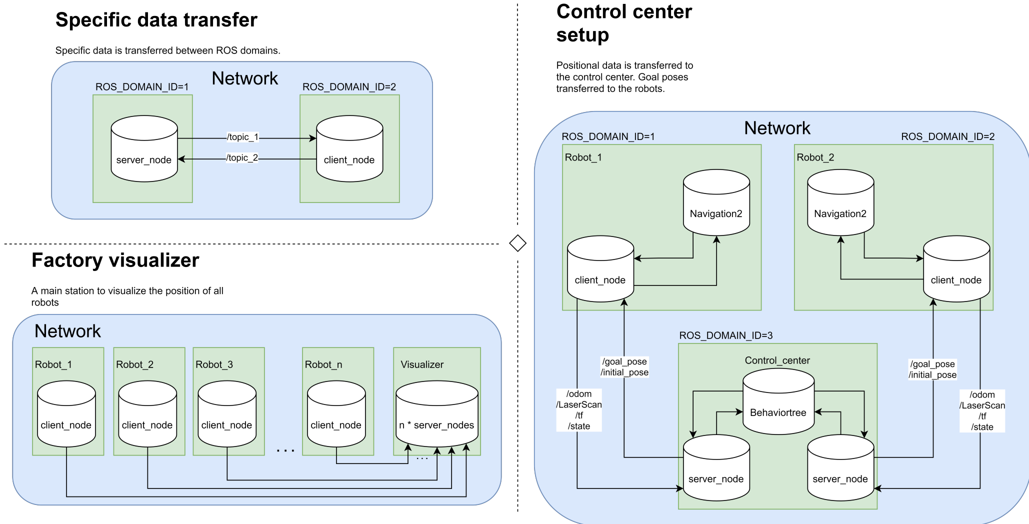


Figure 2.2.: Three different use cases using the same software.

2.2. Development

The stack was developed as an open-source project using continuous integration. During development, the code was updated on a GitHub[1] repository. A fictional version number is applied to each major iteration to provide the reader with some insight into development stages.

2.2.1. Version 1 - Primitive robot-specific server-client program

The first version was a simple and primitive server-client relationship that tested the viability of use in ROS2. The program was created to transfer robot-specific messages across domains, such as LaserScan and Odometry, based on the type of robot set in the configuration file. The robot type was set on the client, and a message was created and sent to the server to enable it to do its corresponding setup and start the communication. This avoids the need to configure both server and client, as the server setup comes automatically from setting up the client. Communication was one-way, as messages were sent from client to server. This version had a very specified use-case and required scenario-specific code, which proved unsustainable for more advanced setups.

The first version served as an excellent proof-of-concept for further development. Several concepts were kept in the next version, such as the server-client setup and an initialization message.

[1] https://github.com/MortenMDahl/ros2_socket_bridge

2.2.2. Version 2 - Over-advanced topic streaming setup

Stepping into the development phase at version 2, many modifications have been made to how the program is set up. The user can specify topics to be transferred, adjust QoS settings and choose transmission protocol. This was done using a 'str_to_class'-function which can convert strings that are received into objects defined in the code. The server-client relationship now allows for two-way communication, enabling streaming from server to client and vice versa, but only using UDP. A specialized serialization script was written for outgoing and incoming messages. It would parse messages into strings, send them, and de-serialize on arrival. This would require manual creation of a serialization script for every message type, and for users who add their own custom message types to also create a serializer and de-serializer function. Topics that were received on the server side were prefixed with the robot name. The idea is that the use case would mainly be the 'control center setup' shown in Figure 2.2. All topics to be received or transmitted are assigned as their own object, a 'BridgeObject'. The BrigeObject contains all the settings for each topic set in the configuration file, in addition to its belonging publisher or subscriber.

This version had many useful traits, but it was still too specified for general use. Creating a custom serialization and deserialization function for each message to be transmitted is cumbersome and requires work for each custom message to be implemented.

2.2.3. Version 3 - Generalized version with serializer and cryptography

This version implements pickle, which is a Python object serialization library. It generalizes the serialization of message objects, removing the need for a custom serializer script having to be made by the user. Due to security risks using pickle, the cryptography Python package was also introduced. In order to be able to communicate between server and client, both sides need to have the same cryptographic key.

Version 3 closes in on being usable as a general communication architecture. There are still some features missing, such as the ability to set the process name as a prefix for published topics and avoiding setup errors when one-way communication is required.

2.2.4. Version 4 - Unproven working program

Changes between version 3 and 4 is mainly code cleanup and generalization, making it more straightforward for a new user to read and use the program. Practical settings such as topic namespace prefixes, a script to generate a random cryptographic key, and a readme file were added. Only a single scenario was tested, where a mobile robot transmits its laser scan and odometry topics to a server. The server transmits the goal pose back to the robot.

A thorough testing phase is needed to uncover any flaws in the software.

2.2.5. Version 5 - Proven working program

Between version 4 and 5, testing of different scenarios and bug fixing was done. The testing unveiled multiple flaws, such as errors when the list of topics to be transmitted is empty. A buffer was needed when transferring large messages with TCP. Clearer error messages are given to the user if an exception occurs. If a topic is stale for a set amount of time, the user is warned. Multiple different scenarios were tested, with messages going both ways between server and client.

2.2.6. Version 6 - Improved program with shutdown handling and Bluetooth

Shutdown handling was added, which closes the sockets and threads used on the server. This allows a new initialization message to be passed in from the client without resetting the server. In addition, the ability to use Bluetooth as a communication method was added through additional socket programming. This allows devices that are not connected to the network but still possess a Bluetooth adapter to receive and send messages. To accompany the introduction of Bluetooth and the challenges that come with it, a script that shows the user the MAC address of the Bluetooth adapter and which ports are busy was made. This was very useful to acquire the MAC address and choose ports quickly as was discovered when experimenting. Setting the server_ip to this MAC address automatically sets the server and client to 'Bluetooth mode'.

At this stage, the newly created stack was named *ros2_socket_bridge*. This will be the reference used when implying the software stack created as part of this thesis. The project was organized using the Black code formatting method to provide users with a consistent and organized experience when exploring the source code.

2.3. Software architecture

The architecture and setup process is shown in Figure 2.3. The server is started and waits for a message from a client on a set port. When the client is started, it reads a setup file from which it creates an initialization message to send to the server. The initialization message is sent from the client to the server, and both parties create BridgeObjects for each topic to be transmitted or received. Each BridgeObject contains information about the topic to be transmitted, whether it is to be received or sent, and a callback function that sends the message based on protocol, and either a subscriber or publisher.

All the BridgeObjects are then connected to each other, and transmission of topics starts. When a message is received on a topic that is set to be transmitted, the subscriber calls the callback function belonging to the BridgeObject. This callback function serializes, encrypts, and sends the message to the receiving socket, which is running on a thread. The message is received, decrypted, deserialized, and published to its belonging topic.

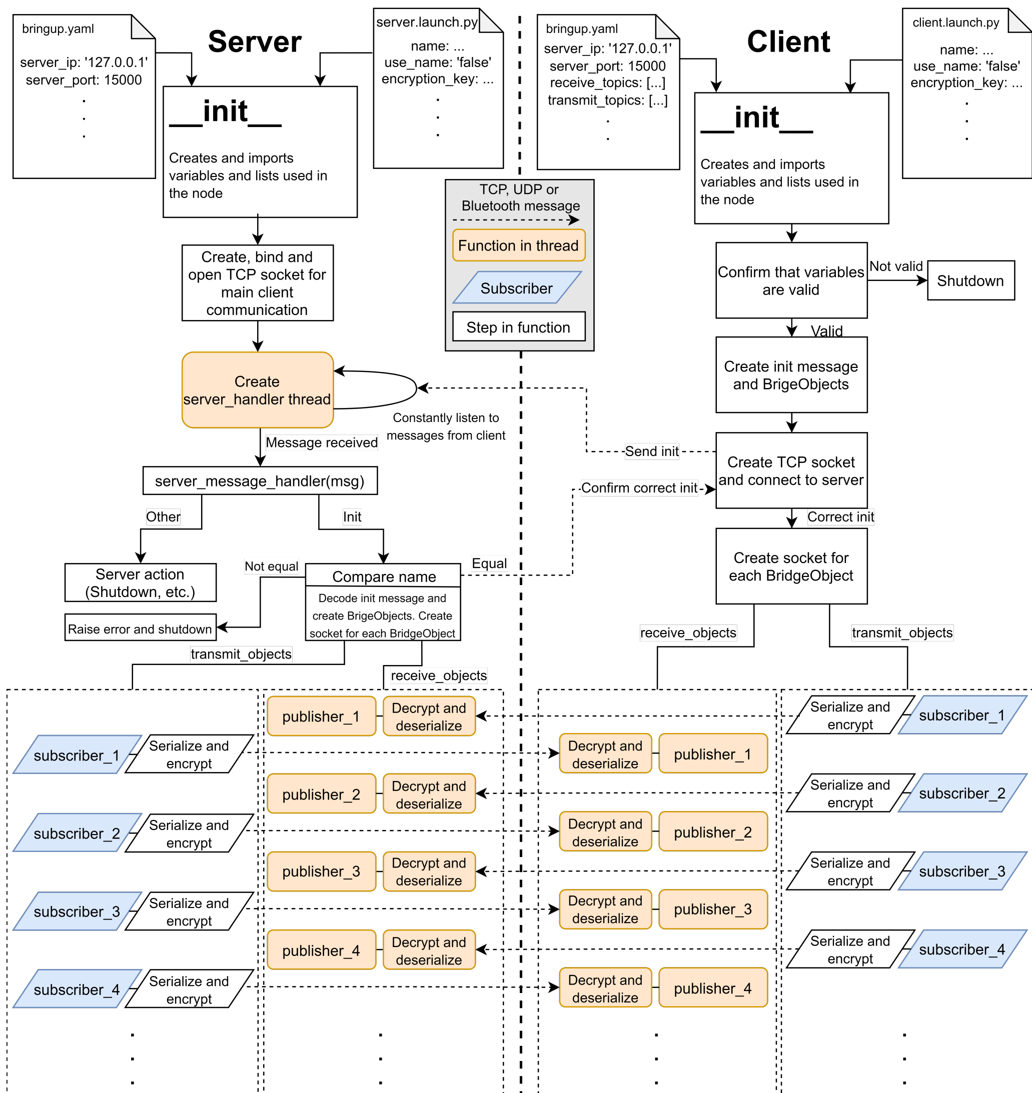


Figure 2.3.: How the software is initialized.

2.3.1. User setup

The way a user learns to use the program is explained in a readme-file located in the GitHub[2] repository, with more detailed documentation hosted on ReadTheDocs[3]. It includes a detailed installation guide, setup guide and example scenarios. The user has a few settings to tweak in order to use the software. The settings,

[2] https://github.com/MortenMDahl/ros2_socket_bridge
[3] https://ros2-socket-bridge.readthedocs.io/

their location, and a short description is listed in Table 2.1.

To set up the client-server relationship, the IP of the computer running the server has to be set in the launch file of both server and client. The server needs to know which port to use as a main communication line with the client. The user also has to set whether or not to use a namespace in front of the incoming topics for both server and client. The last thing required in the launch files is the encryption key to be used. This has to match with both the server and client in order to be able to encrypt and decrypt messages.

The server is only dependent on its IP and the port it should listen to. All the other settings, such as topics to transmit, the message type of these topics, the QoS it uses, the port used for transfer, and the protocol of that port, are all client-side settings. This brings with it the advantage that only the client needs to be rebuilt and recompiled when tweaks to the settings are done.

Booting the server and client displays information about the connection process, gives the user what address is selected, and displays a warning if topics are stale. An example of the booting process is shown in Figure 2.4.

```
kmriiwa@kmriiwa-NUC8i7HNKQC:~/Skrivebord/ros2_socket_bridge$ ros2 launch rsb_server server.launch.py
[INFO] [launch]: All log files can be found below /home/kmriiwa/.ros/log/2021-05-13-13-09-15-015857-kmriiwa-NUC8i7HNK
QC-25348
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [server-1]: process started with pid [25350]
[server-1] [INFO] [1620904155.568869439] [server_node]: fleet1/robot1_server_node waiting for connection at port 3000
...
[server-1] [INFO] [1620904159.300440603] [server_node]: fleet1/robot1_server_node: data received from 127.0.0.1:60640
[server-1] [INFO] [1620904159.495219084] [server_node]: Received initialization message.
[server-1] [INFO] [1620904159.751095382] [server_node]: Matching initialization settings. Confirming with client.
[server-1] [INFO] [1620904160.017198574] [server_node]: Receiving connections established!
[server-1] [INFO] [1620904163.017619548] [server_node]: topic subscription started!
```

(a) Server booting information.

```
kmriiwa@kmriiwa-NUC8i7HNKQC:~/Skrivebord/ros2_socket_bridge$ ros2 launch rsb_client client.launch.py
[INFO] [launch]: All log files can be found below /home/kmriiwa/.ros/log/2021-05-13-13-09-18-551144-kmriiwa-NUC
8i7HNKQC-25364
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [client-1]: process started with pid [25366]
[client-1] [INFO] [1620904159.182466029] [client_node]: fleet1/robot1 connecting to server (127.0.0.1:3000)...
[client-1] [INFO] [1620904159.182944876] [client_node]: Connected! Sending initialization message...
[client-1] [INFO] [1620904159.751890446] [client_node]: Matching initialization message confirmed.
[client-1] [INFO] [1620904161.754382459] [client_node]: Establishing connections...
[client-1] [INFO] [1620904161.754758336] [client_node]: Transmission channels established!
[client-1] [INFO] [1620904163.259431531] [client_node]: topic connected!
[client-1] [INFO] [1620904163.260138182] [client_node]: Receiving connections established!
[client-1] [WARN] [1620904178.275056790] [client_node]: No data received from topic | Warning #1
[client-1] [WARN] [1620904193.291450516] [client_node]: No data received from topic | Warning #2
[client-1] [WARN] [1620904208.308240607] [client_node]: No data received from topic | Warning #3
[client-1] [WARN] [1620904223.325046959] [client_node]: No data received from topic | Warning #4
[client-1] [WARN] [1620904223.326641979] [client_node]: ==============================
[client-1] [WARN] [1620904223.328030320] [client_node]: Stopping warning for topic
[client-1] [WARN] [1620904223.328876014] [client_node]: ==============================
```

(b) Client booting information.

Figure 2.4.: Server and client boot information. A single stale topic named '/topic' is being transmitted from server to client. As the topic is stale, warnings are triggered client-side.

Table 2.1.: Variables available to the user.

File	Variable	Description
Server		
'src/rdb_server/config/bringup.yaml'	server_ip	IP of the computer where the server node is running.
'src/rdb_server/launch/server.launch.py'	name	Name of the process or robot to be communicated.
	server_port	Port used for the main communication between server and client.
	use_name	Whether or not to use name as a topic prefix.
	use_encryption	Whether or not to encrypt and decrypt messages
	encryption_key	Key used for encryption and decryption. Must match with the client.
Client		
'src/rdb_client/config/bringup.yaml'	server_ip	IP of the computer where the server node is running.
	server_port	Port used for the main communication between server and client.
	*_topics	Topic which are to be received or transmitted.
	*_msg_types	The message type of topics. Example: LaserScan
	*_ports	Ports used for communication. Bound on the server.
	*_protocols	Which protocol is to be used for communicating the topic. TCP or UDP.
	*_qos	QoS which is to be used for the publisher and subscriber.
'src/rdb_client/launch/client.launch.py'	name	Name of the process or robot to be communicated.
	use_name	Whether or not to use name as a topic prefix.
	use_encryption	Whether or not to encrypt and decrypt messages
	encryption_key	Key used for encryption and decryption. Must match with the server.

2.3.2. Server-client relationship

The software works by enabling a server-client relationship. The only meaningful difference between a server and a client is that ports are bound on the computer running the server. This also requires some under-the-hood changes to the code used for communicating, but it makes no difference to the user. It is the placement difference of the client and server which allows for cross-domain or even cross-network communication. This allows for the creation of custom communication architectures, much like the five layers of a DCS. Figure 2.5 shows an example of how this can be done.

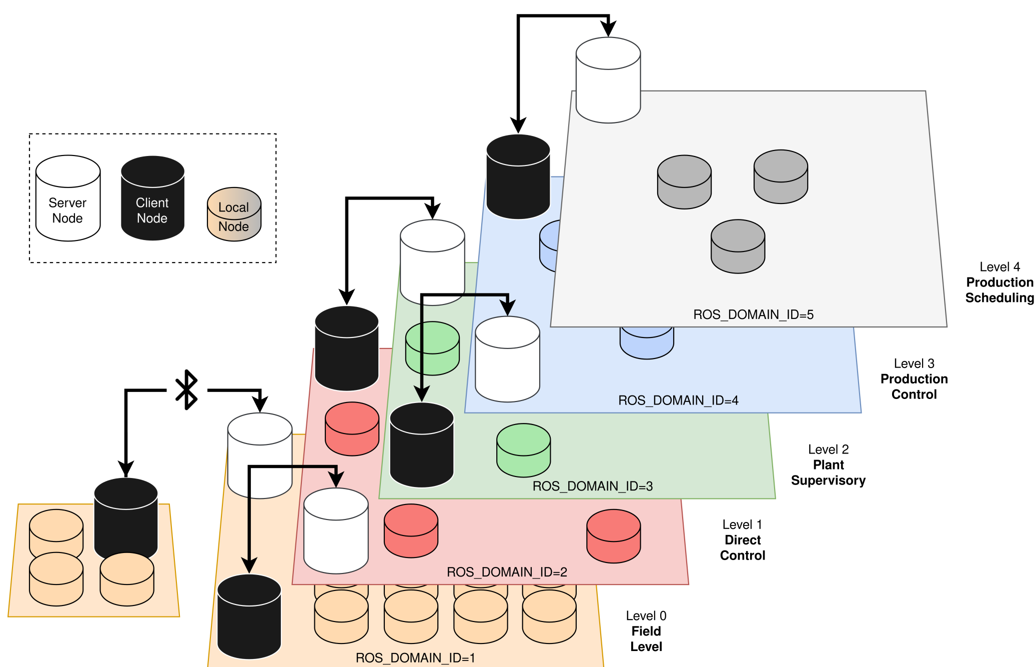


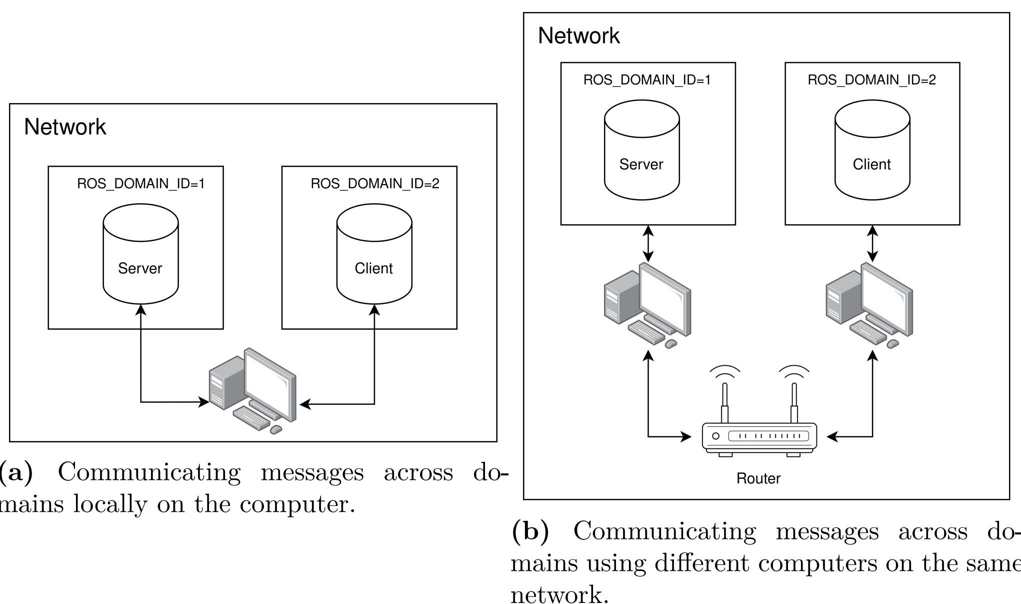
Figure 2.5.: Hierarchical DCS setup using ros2_socket_bridge.

2.3.3. Connection modes

It is possible to transfer topics between local domains existent on the network and transfer to different networks. Currently, the IPv4 protocol is set, but users could easily enable IPv6 by changing a single variable in the socket creation code of both server and client. It is also possible to connect two devices using Bluetooth.

Local network

On the local network, IPv4 addresses which typically starts with 192.168.x.x is used (Figure 2.6b). If the server and client are located on the same computer, the IP address can be passed in as 127.0.0.1 (Figure 2.6a). The server and client nodes should be located in different domains, as the domains are network-wide. It is also possible to transfer topics between computers on a network where UDP multicasting is disabled, allowing for communication between ROS2 nodes that natively cannot discover each other. This is due to the direct communication (unicasting) nature of socket communication which does not require multicasting to be enabled, unlike DDS.



(a) Communicating messages across domains locally on the computer.

(b) Communicating messages across domains using different computers on the same network.

Figure 2.6.: Server-client local connection modes.

External network through port forwarding

One way of connecting two computers over the internet is to use port forwarding. The external IP of the computers has to be found, and port forwarding has to be enabled through the Internet Service Provider [21]. Once this is done, the external IP with the set external port will redirect to a local IP and port where messages are received. Additionally, exceptions have to be made in the firewall on both ends to prevent the blocking of messages.

As the distance between the two connected network increase, so does the latency. A common problem with communicating across the open internet is that messages

are forwarded using intermediary routers. An effect often called the “trombone effect” could occur [22]. This is when the data messages are taken on detours through different directions, making the path that data has to make excessively far.

External network through a VPN tunnel

One of the more secure and easy to set up methods of connecting two networks is through a VPN tunnel. This tunnel handles all the advanced low-level network interfacing and grants the user simple access to which IP and port to use. Many vendors offer specific VPN routers, such as Linksys and D-link. Certain local IP addresses are automatically forwarded to computers on the connected network.

Unlike port forwarding, VPN vendors often have their own servers where messages are passed through. This could help negate the trombone effect, depending on the location of the servers relative to the two networks.

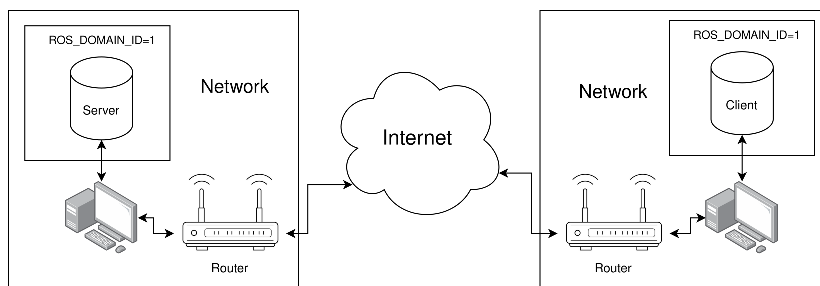


Figure 2.7.: External server-client connection across the internet. This can be done with both port forwarding and a VPN tunnel.

Bluetooth connection

As Bluetooth programming is part of the standard socket library of Python, communicating messages across devices using Bluetooth was enabled. To set the server and client to communicate using Bluetooth, the ‘server_ip’ has to be set to the MAC address of the Bluetooth device attached to the server computer. Additionally, any topics to be transmitted should have their protocol set to ‘BLUETOOTH’ instead of ‘TCP’ or ‘UDP’. The server and client devices first have to be manually connected using Bluetooth, commonly available in system settings.

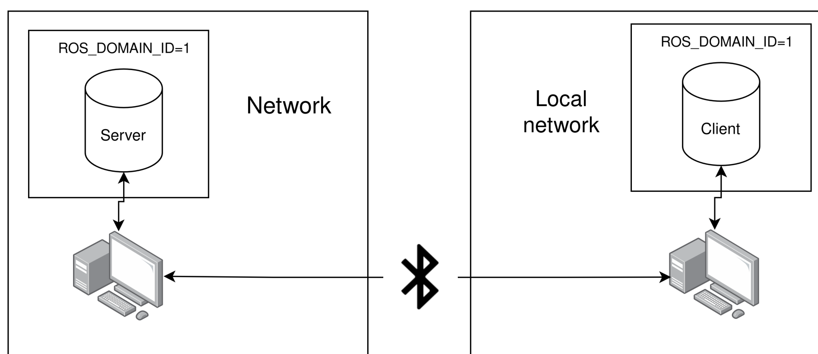


Figure 2.8.: Bluetooth connection mode.

Latency considerations

When dealing with systems dependent on information in real-time, it is important to consider what effect the method of transferring messages has in slowing the system down. In the case of steering and control processes, the combination of input delays with regulators causes problems. Therefore, it is essential to consider where controlling stacks are located and for this thesis to find out how the transfer of topics causes delays. Dependent on the transfer method and the amount of delay it causes, the need to have the controlling computer of a robot locally should be considered. Independent of delays, commands such as goal positions could be transferred from an external source. Any emergency stop mechanisms should be independent of the external source and be locally placed on either the control computer or the hardware.

2.3.4. Encryption key generation

A script to generate a random 32 url-safe base64-encoded bytes object is included when downloading the package. The script generates a key and prints it together with the time and date to a separate ‘key.txt’-file. This is done by using the `fernet.generate_key()` function from the cryptography package. An example of the generated ‘key.txt’-file is;

```
Key generated at 16/04/2021 14:54:53:

VdzT2kwMacThZWkBigjbtte9iRjW8djEJ10JiemVwLM=
```

2.3.5. Bluetooth channel checker

A custom script to show which Bluetooth channels are busy for communication was implemented. Running the script returns the MAC address of the Bluetooth adapter, which needs to be set as the server's IP address and a list of busy channels on this adapter. This helps the user select appropriate channels if Bluetooth is to be used. An example of what the script returns is;

```
----------------------------------------
Your Bluetooth adapters MAC address is:
 80:32:53:E0:12:C0
----------------------------------------
Busy Bluetooth channels [1-30]:
 [3, 9, 10, 12, 14, 15, 16, 17]
----------------------------------------
```

2.4. Performance experiments

There are two main limiting factors when transmitting sensor data from a robotic system. The first is the real-time requirement of the system - that is the total time delay created when communicating a message from origin to destination. The second is potential bottlenecks created by the rate of incoming data. If the system which readies messages to be transmitted is slower than the rate that messages arrive, we get a bottleneck. The total time is shown in Figure 2.9, where a new message can start to get processed once the data reaches the end of transmission from outgoing to incoming.

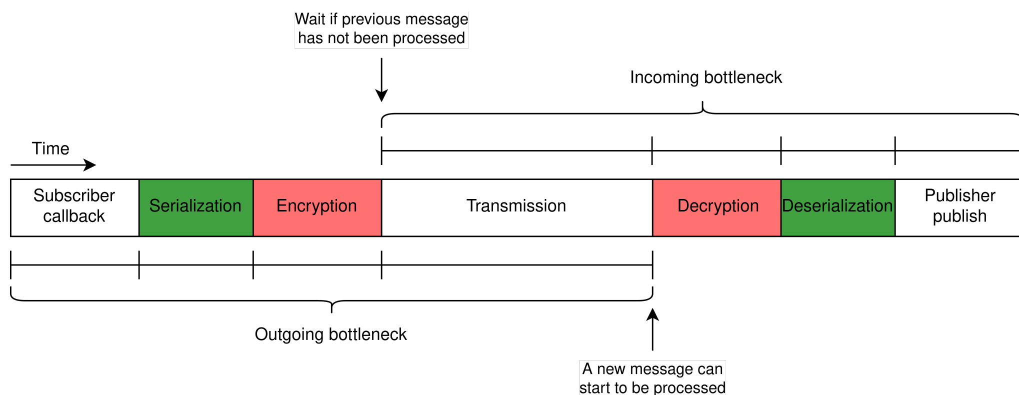


Figure 2.9.: Message processing bottlenecks. Once a message reaches the end of transmission, a new message can be taken in by the subscriber.

According to Weyuker and Vokolos [23], both functional and performance testing is essential to the results when developing system architectures. The functional testing was done throughout development, and different setups were created to test performance and find bottleneck-prone areas of the software.

2.4.1. Setup

Simulated robot

An experiment setup is created as shown in Figure 2.10. The Bridge computer is responsible for running both server and client, taking data from topics that are run to control a mobile robot using Navigation2. The Navigation2 stack and Gazebo simulator is run on a separate simulation computer but share a domain with the bridge computer. A TurtleBot3 is simulated in Gazebo using the 'nav2_bringup' stack. Multiple topics which are vital to visualize and control the robot is transmitted from one domain to another. At the same time, the time requirement of serialization, encryption, decryption and deserialization is recorded on the bridge computer by timing and writing to a file during transmission. The robot is controlled using keyboard controls through the teleop_twist_keyboard stack running on the bridge computer. The topics transferred will be referenced to as 'Laser', 'Transformation', 'Costmap' and 'Footprint'. Laser is the laser-scanner sensor data, transformation is transformation matrices between different coordinate systems on the robot, costmap is the weighted map used for path planning, and footprint is the volume of which the robot takes up in space.

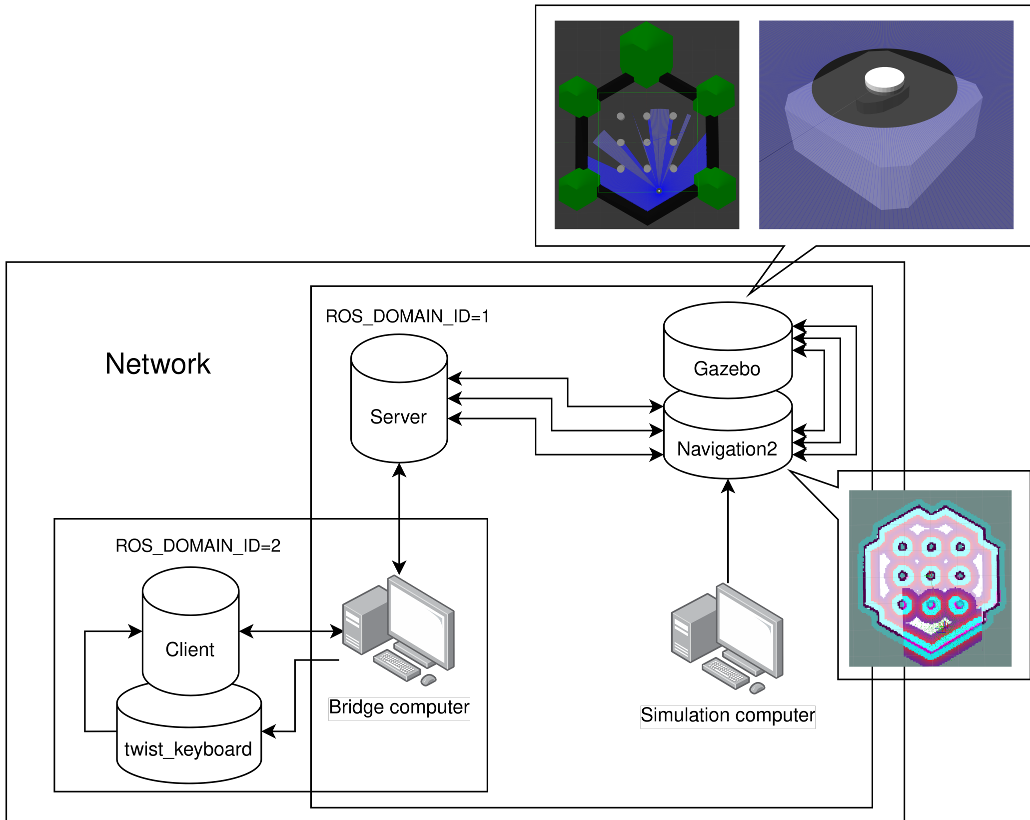


Figure 2.10.: Performance experiment setup using a simulated robot. The bridge computer has two different terminals in separate ROS domains, and is running both server and client. The simulation computer is running Navigation2 and Gazebo to simulate messages coming from a mobile robot.

Controlled publishing

A second experiment setup is created as shown in Figure 2.11. Two different tests are done in this setup - one which tests the time it takes to pack and unpack messages based on the size of the message, and another to test the maximum output rate at the destination domain based on message size. The messages published by the simple_publisher is a ROS-style 'String' message. The size of this message is set by adding a certain amount of letters equivalent to the number of bytes requested, minus size required by the message object and serialization wrapper as shown;

```
package = b'' + b'0'*(msg_size - (ros_msg_wrapper_size + pickle_wrapper_size))
msg = String(data=package)
```

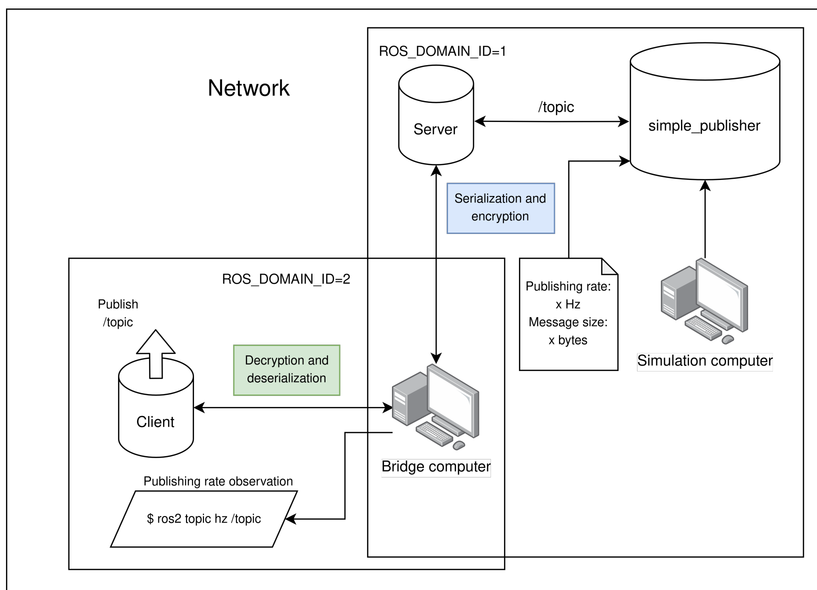


Figure 2.11.: Performance experiment setup using a custom publisher with set data size and rate. The bridge computer has two different terminals in separate ROS domains and runs both server and client. The simulation computer is running the custom publisher from which the rate and size of messages are set.

This experiment was tested on all internal and external communication protocols, including Bluetooth, as shown in Figure 2.12. Publishing rates at the origin and destination (target) were noted as the publishing frequency increased. An upper limit was discovered for all message sizes and protocols when the target publishing rate was lower than the origin.

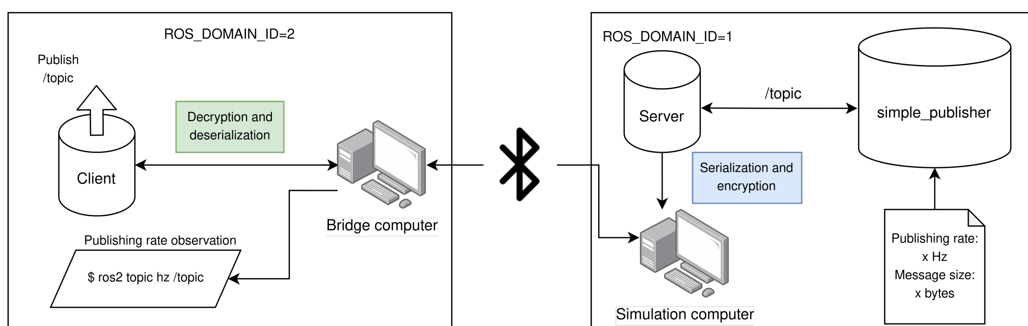


Figure 2.12.: Performance experiment setup using Bluetooth as a communication medium.

2.4.2. Performance considerations

To eliminate performance influence by other programs running on the computer, the CPU was set to performance mode by running `sudo cpufreq-set -r -g performance` from the cpufrequtils package. Python automatically clears memory leaks at random times, which costs processing power. To remove any impact of this, automatic garbage collection in Python was disabled for the experiment by running `gc.disable()` from the gc package which is available in the standard Python library. The hardware of each computer is listed in Table 2.2.

Table 2.2.: Hardware information of the computers used for experiments

Computer	Name	Processor	Clock speed	Cores	Threads
Bridge Computer	Intel NUC	Intel® Core™ i7-8705G	3.10-4.10 GHz	4	8
Simulation Computer	Surface Book 2-in-1	Intel® Core™ i5-6300U	2.40-3.00 GHz	2	4

2.4.3. Result interpretation

Looking at the average time it takes for a message to go through the two bottleneck-prone areas of transmission and adding the time of three standard deviations gives us the maximum time it takes for 99.73% of the messages to be processed (Equation 2.1) [24]. This is known as the *Empirical Rule*, and will then be used as an approximate for the maximum message rate that can be transmitted. Three standard deviations are chosen due to the results being realistic at a high percentage of incidence but not too high, bringing unrealistic handicaps to expected outcomes in real-life scenarios. The results are hardware and message dependent but should still give a pointer to expected performance on publicly available hardware.

$$P(\mu - 3\sigma \leq X \leq \mu + 3\sigma) \approx 99.73\% \tag{2.1}$$

The average was calculated using Equation 2.2, while the standard deviation was calculated using Equation 2.3.

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2.2}$$

$$\sigma = \sqrt{\sum_{i=1}^{n} (x_i - \mu)^2} \tag{2.3}$$

These results should then be tested by having an experiment that record the output and input publication rate on origin and goal domains to see at which rate the bottleneck occurs.

All the experiments are hardware-dependent, as they all depend on processing- and transmission time of messages to test their limits. Variability is also found in the router's signal strength, which links the two computers in experiments where server and client are not on the same computer. The experiment testing Bluetooth transmission is dependent on both the Bluetooth version of both devices and signal strength.

Chapter 3.

Results

3.1. Robot simulation dataset

The dataset is composed of multiple messages from different common topics of varying size, listed in Table 3.1. The samples were gathered by running a simulated Turtlebot3 robot in circles over the course of 12 hours at a simulation speed of $\sim 0.3\times$ real-time.

Table 3.1.: Sample information gathered from transferring messages.

Message	Sample size	Serialized message size [bytes]
LaserScan	140,862	3416
TF	1,389,059	752 ± 106
Costmap	24,852	4358
Published footprint	101,968	1132

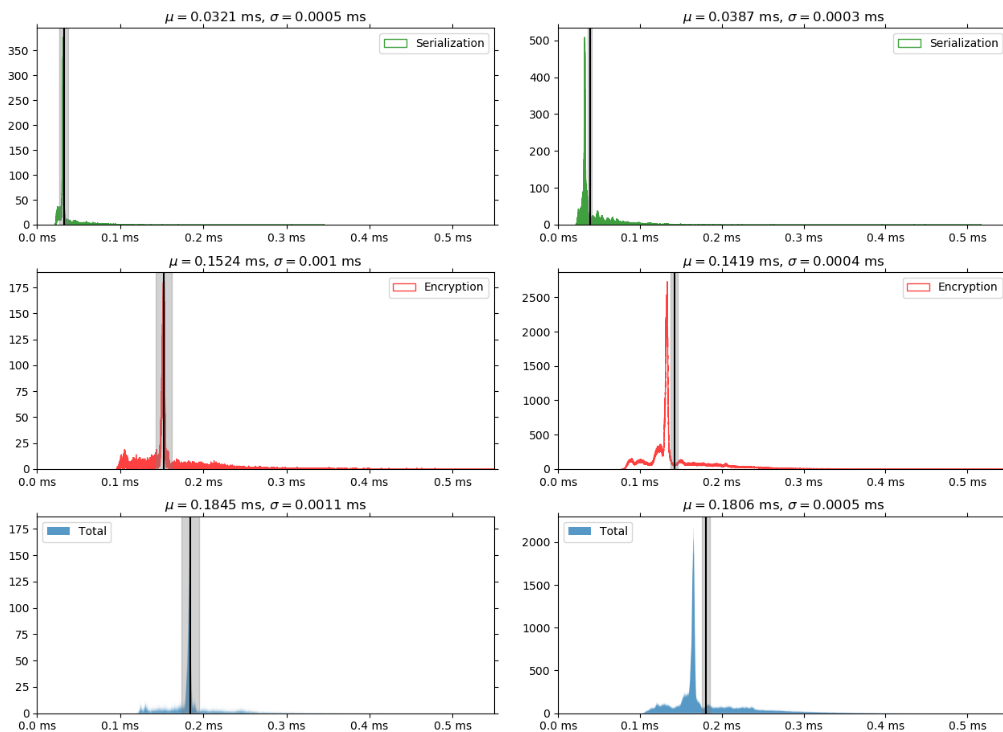
3.2. Message processing time

The robot simulation dataset was processed, and the time requirement for serialization, encryption, deserialization and decryption was transferred to graphs and tables to give a better understanding of which process takes up the most processing time. The setup used in this experiment is explained in Section 2.4.1 on page 27.

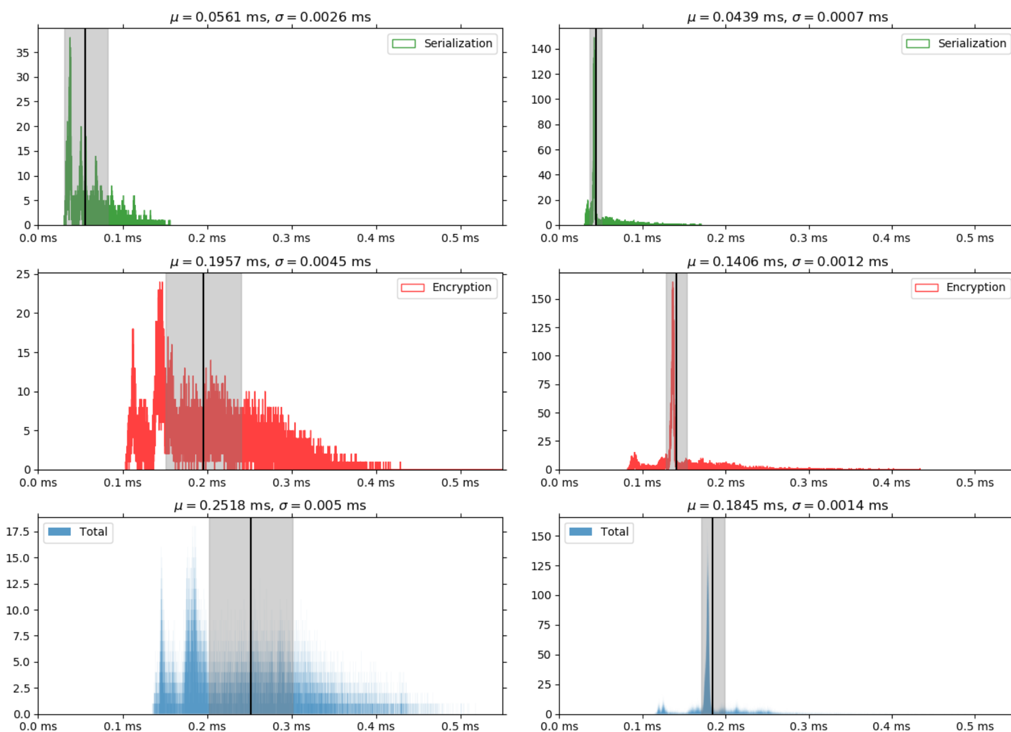
3.2.1. Serialization and encryption

The results gathered from serializing and encrypting messages are shown in Figure 3.1 and listed in Table 3.2. It is apparent that encryption is what contributes to the processing time of outgoing messages. Despite this, the total processing time is around 0.2 ms for all messages. Using the Empirical rule, the theoretical publishing rate limit stays above 2 kHz for all outgoing topics.

Figure 3.1 shows time of the process on the x-axis, while the y-axis shows the amount of messages at the same time duration at an accuracy of three decimal places. The black vertical line shows the average, while the grey area shows the standard deviation.



(a) Laser messages **(b)** Transformation messages



(c) Costmap messages **(d)** Robot footprint messages

Figure 3.1.: Results gathered form the time it took to serialize and encrypt different messages.

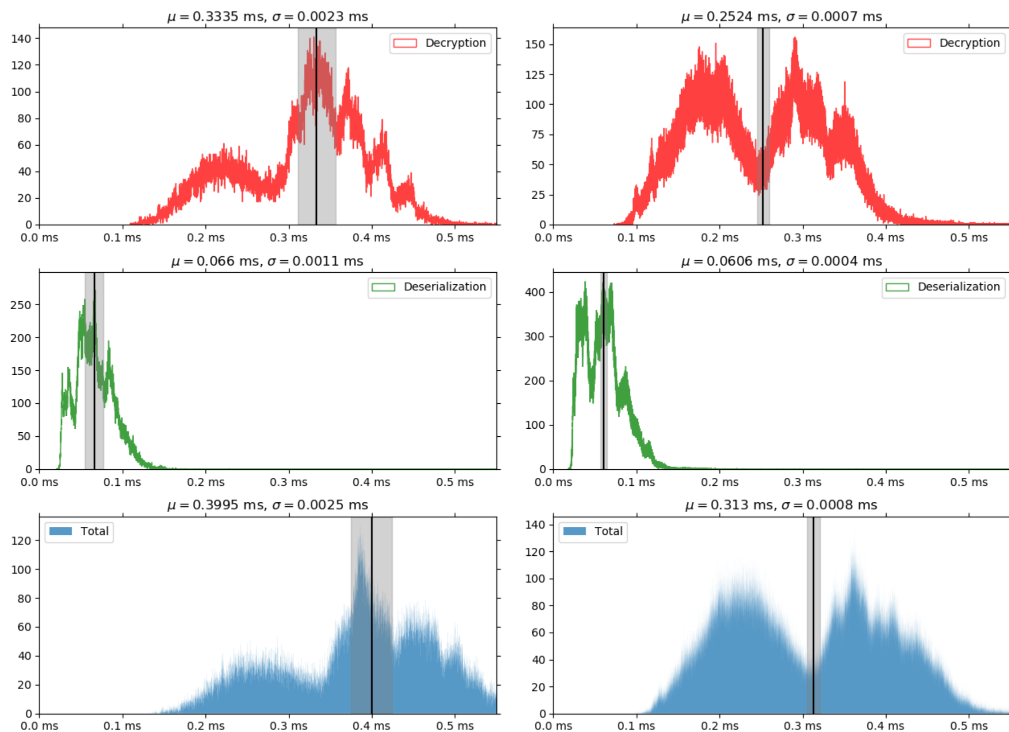
Table 3.2.: Results gathered form the time it took to serialize and encrypt different messages.

Variable	Message type			
	Laser	TF	Costmap	Footprint
Serialization				
Mean (μ) [ms]	0.032	0.039	0.056	0.044
Standard deviation (σ) [ms]	0.005	0.003	0.026	0.007
Encryption				
Mean (μ) [ms]	0.152	0.142	0.196	0.141
Standard deviation (σ) [ms]	0.01	0.004	0.045	0.012
Total				
Mean (μ) [ms]	0.184	0.181	0.252	0.185
Standard deviation (σ) [ms]	0.011	0.005	0.05	0.014
$\mu + 3\sigma$ [ms]	0.217	0.196	0.402	0.227
Theoretical subscription rate limit [Hz]	4608	5102	2487	4405

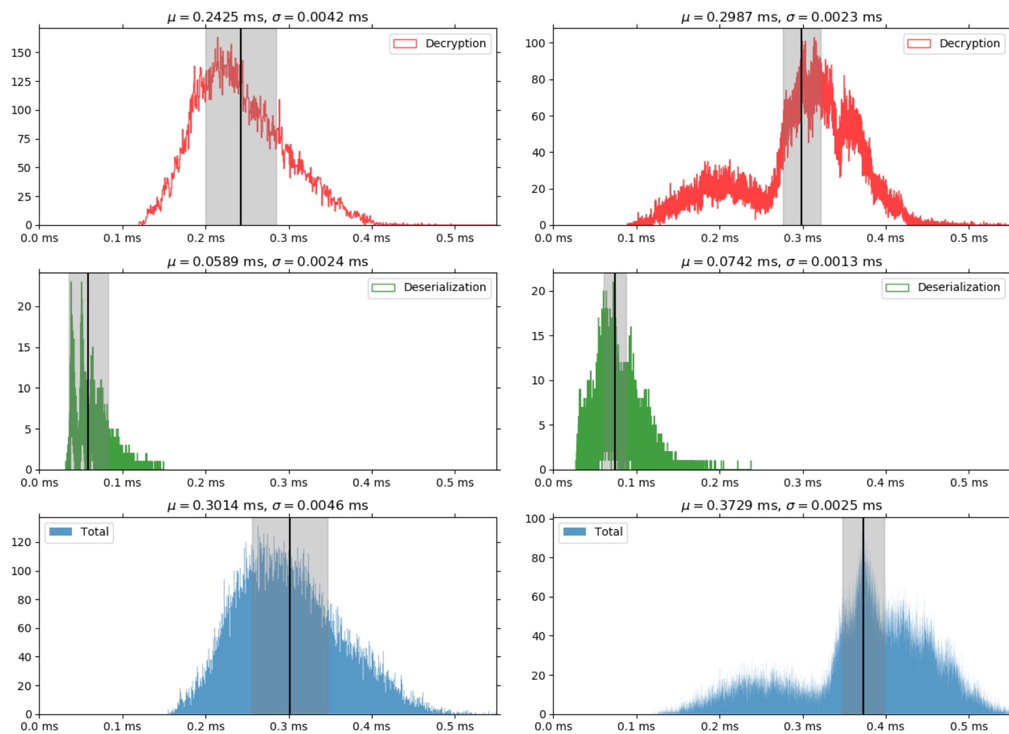
3.2.2. Decryption and deserialization

The results gathered from decrypting and deserializing messages are shown in Figure 3.2 and listed in Table 3.3. As with serialization and encryption, the encryption part contributes to the highest increase in processing time. On the receiving part of the program, the theoretical publishing rate limit still stays above 2 kHz for all message types.

As with serialization and encryption, the x-axis of Figure 3.2 shows the processing time. The y-axis shows the number of messages recorded at this time step at an accuracy of three decimal places. The black vertical line shows the average, while the grey box shows the standard deviation.



(a) Laser messages **(b)** Transformation messages



(c) Costmap messages **(d)** Robot footprint messages

Figure 3.2.: Results gathered form the time it took to decrypt and deserialize different messages

Table 3.3.: Results gathered form the time it took to decrypt and deserialize different messages.

Variable	Message type			
	Laser	TF	Costmap	Footprint
Decryption				
Mean (μ) [ms]	0.333	0.252	0.242	0.299
Standard deviation (σ) [ms]	0.023	0.007	0.042	0.023
Deserialization				
Mean (μ) [ms]	0.066	0.061	0.059	0.074
Standard deviation (σ) [ms]	0.011	0.004	0.024	0.013
Total				
Mean (μ) [ms]	0.40	0.313	0.301	0.373
Standard deviation (σ) [ms]	0.025	0.008	0.046	0.025
$\mu + 3\sigma$ [ms]	0.475	0.337	0.439	0.448
Theoretical publishing rate limit [Hz]	2105	2967	2277	2232

3.3. Processing time versus message size

To compare message processing time to the size of the message, both message size (x-axis) and processing time (y-axis) was plotted against each other. The semi-transparent areas are the extrapolated standard deviation between data points. This was done in two experiments - one where the dataset from the simulated robot is used (Section 2.4.1, page 27), and one where the publisher publishes messages at a controlled rate and size (Section 2.4.1, page 28).

3.3.1. Simulated robot

The results shown in Figure 3.3 is generated from data taken when simulating a Turtlebot3 and transferring messages. These results lack the expected linearity between message size and processing time.

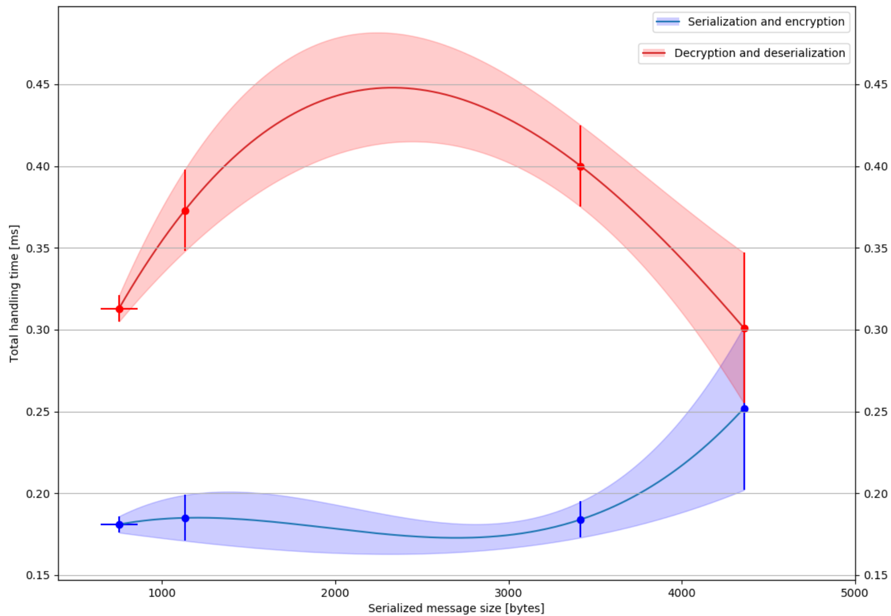


Figure 3.3.: Packing time versus message size for messages used by the simulated robot. The x-axis values corresponds to the message sizes in Table 3.1

3.3.2. Controlled publisher

To get data under a more controlled environment results shown in Figure 3.4 were gathered using a custom publisher publishing messages of varying size at 50Hz. 50Hz was chosen due to its regularity in normal operation and the fact that this rate is very low compared to the maximum rates of the software.

In this result, we can see that the handling time increases with message size, and that decryption and deserialization have a big standard deviation relative to serialization and encryption. The time required to decrypt and deserialize also increases faster than serialization and encryption, making it apparent that the bottleneck is on the receiving end of the communication.

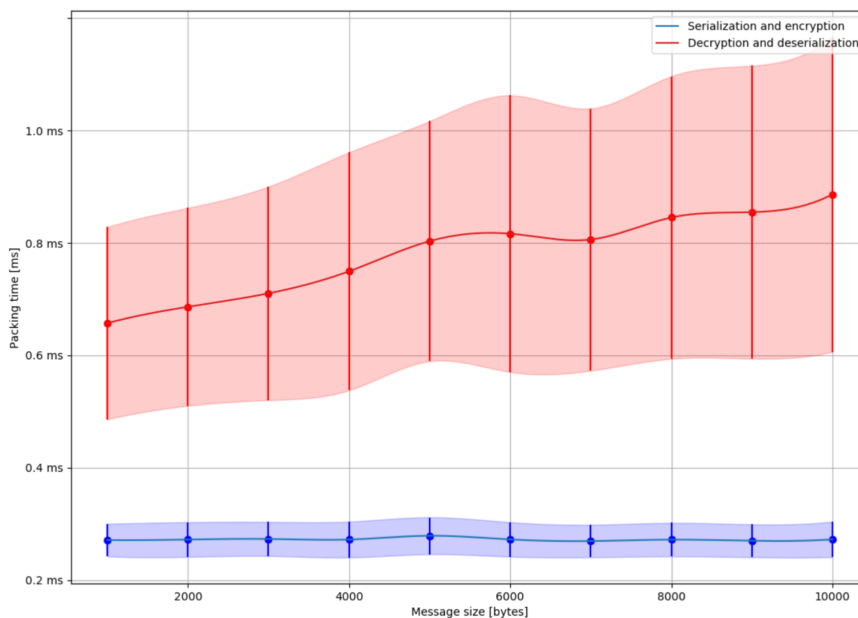


Figure 3.4.: Packing time versus message size at a publishing rate of 50Hz. Custom messages were published at a rate of 50 Hz with a size varying from 1 to 10 kB at 1 kB intervals.

3.4. Maximum publishing rate versus message size

Adjusting the message size and publishing rate allowed for publishing rate benchmarking between two domains. This was done by increasing the publishing rate at the original domain by set intervals until the publishing rate at the goal domain reached an apex which was lower than the original domain. The point of this apex varies by message size, as shown in this experiment.

The setup during this experiment is of the 'Controlled publisher' type.

For the graphs in this section, the x-axis is the publishing rate at the original domain where messages are taken. The y-axis is the publishing rate at the target domain, i.e. where messages are transferred to. There would be no loss in an ideal situation, and all the protocols and message sizes would follow a linear line.

3.4.1. Internal communication tests

The internal communication tests were conducted while transmitting messages internally on the Bridge computer using an IP-address of '127.0.0.1'. Both server and client was running on the same computer.

UDP

As shown in Figure 3.5, the maximum publishing frequency reaches its highest rate at around 2300 Hz for 10 kB, 2950 Hz for 5 kB and 3300 Hz for 2.5 kB.

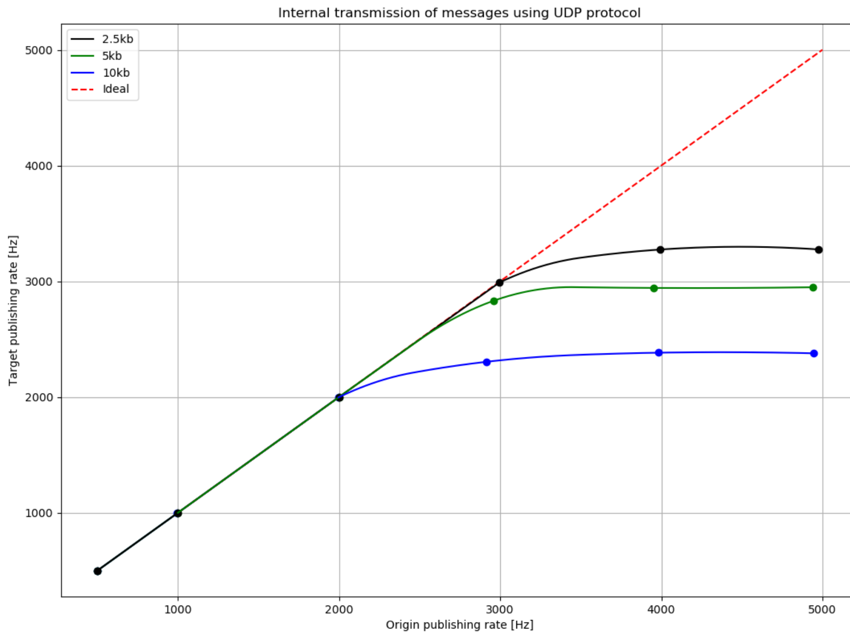


Figure 3.5.: Internal transmission of messages using UDP

TCP

As shown in Figure 3.6, the maximum publishing frequencies are slightly increased at around 2800 Hz for 10 kB, 3400 Hz for 5 kB and 3600 Hz for 2.5 kB.

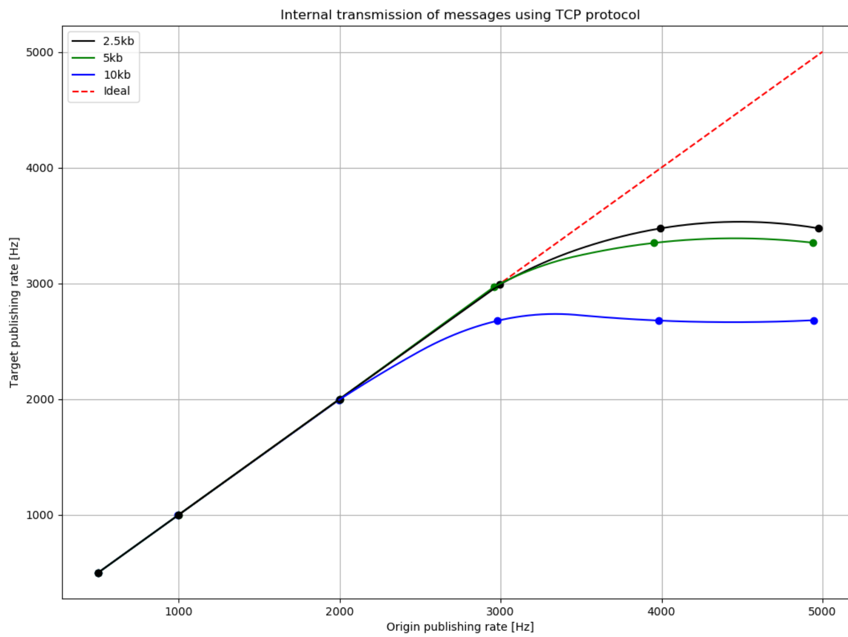


Figure 3.6.: Internal transmission of messages using TCP

3.4.2. External communication tests

The external communication tests were conducted while transmitting messages using a router on the local network. The IP address of the server was set to its static IP address on the network. In the case of Bluetooth, both computers were disconnected from the local network and connected using Bluetooth. The server and client were running on different computers.

UDP

In Figure 3.7 we see publishing rates capped at around 900 Hz for 10 kB, 1700 Hz for 5 kB and 1950 Hz for 2.5 kB.

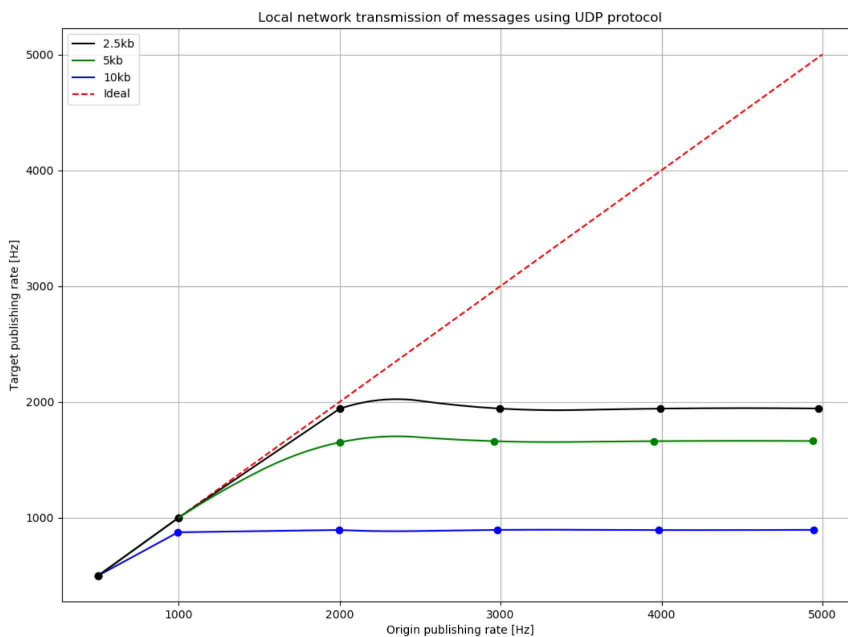


Figure 3.7.: Local network transmission of messages using UDP

TCP

In Figure 3.8 we again see an increase in the publishing cap at around 900 Hz for 10 kB, 1800 Hz for 5 kB and 2050 Hz for 2.5 kB.

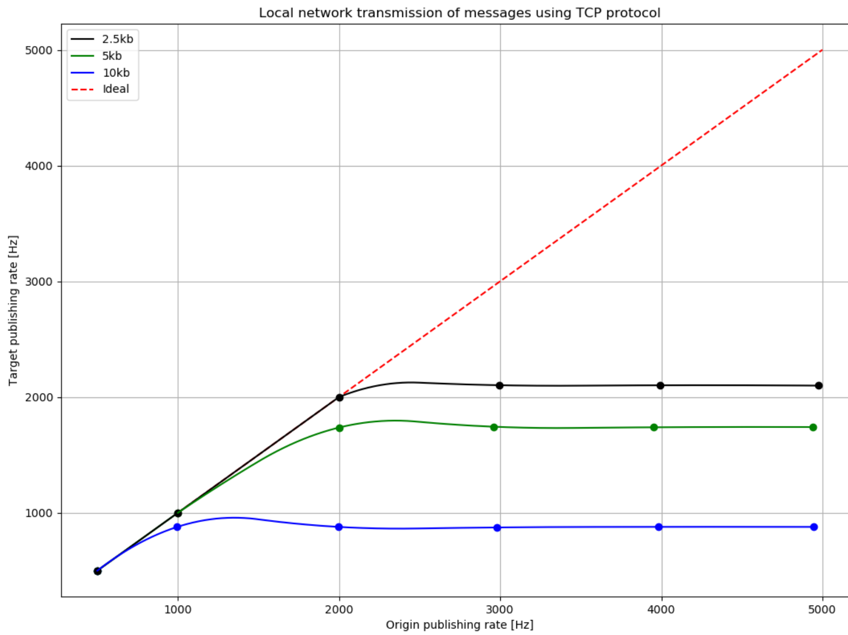


Figure 3.8.: Local network transmission of messages using TCP

Bluetooth

In Figure 3.9, we see that relative to using the local network, Bluetooth is very slow. Rates peaks at around 5 Hz for 10 kB, 11 Hz for 5 kB and 22 Hz for 2.5 kB.

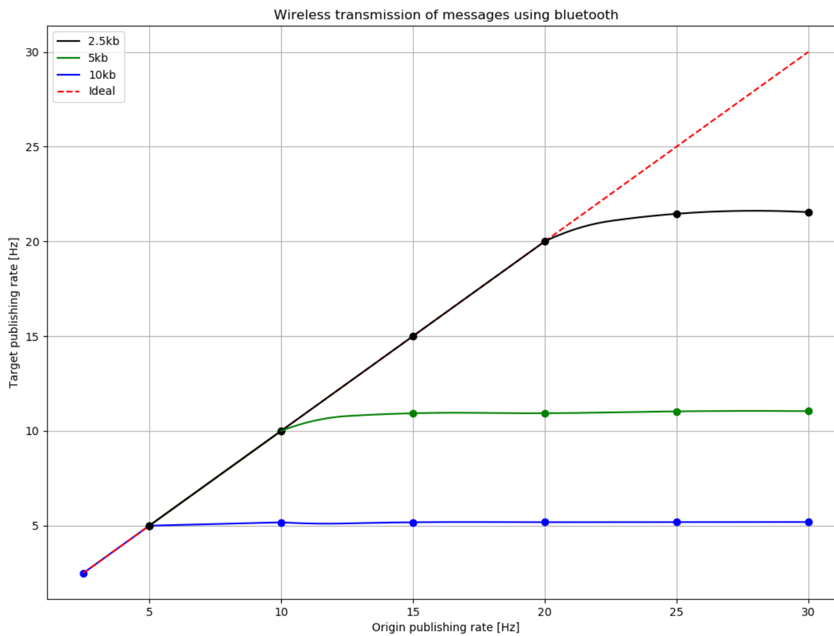


Figure 3.9.: Wireless transmission of messages using Bluetooth

3.4.3. Summary

A summary of all the maximum publishing rates is listed in Table 3.4. Transferring messages internally on the computer gives the highest publishing rate cap, while local network transmissions are slightly behind. Surprisingly, TCP is marginally better in both scenarios. Transferring using Bluetooth is slow and not suited for real-time commands but is still useful for transmitting sensor data with a low measuring rate.

Table 3.4.: A summary of peak publishing speeds for all message sizes and protocols.

(a) A summary of speeds with a 10kB message.

Message size: 10 kB		
Protocol	Peak publishing rate [Hz]	
	Internal	External
UDP	2300	900
TCP	2800	900
Bluetooth	-	5

(b) A summary of speeds with a 5kB message.

Message size: 5 kB		
Protocol	Peak publishing rate [Hz]	
	Internal	External
UDP	2950	1800
TCP	3400	1800
Bluetooth	-	11

(c) A summary of speeds with a 2.5kB message.

Message size: 2.5 kB		
Protocol	Peak publishing rate [Hz]	
	Internal	External
UDP	3300	1950
TCP	3600	2050
Bluetooth	-	22

Chapter 4.

Discussion

4.1. Benchmarking

4.1.1. Measuring method

As all the areas of the code that affect the performance of the software are dependent on receiving an information stream, the timing measurements had to be taken in real-time whilst running the code. This prevented the use of function testing, and an in-code timer had to be implemented.

```
import timeit
import random
# Function we want to measure
def foobar(x):
    randomlist = []
    n = random.randint(0,30)
    if len(randomlist) < x:
        randomlist.append(n)
    else:
        return randomlist.sort()

# Example of function testing:
%timeit foobar(100)

# Example of a manual in-code timer:
start = timeit.default_timer()
foobar(100)
print(timeit.default_timer()-start)
```

This slightly affects the results, as creating a timer and writing the results to a file while running requires some processing power. Timeit commonly runs the function multiple times and generates both an average and standard deviation, which in this case is done manually from the data file.

Despite this slight source of decreased performance, the expected publishing rate

from experiments was close to the actual measured rates.

4.1.2. Maximum publishing rates

Surprisingly, TCP was the fastest protocol to use for both internal and external tests. This was an unexpected result, as messages using sent using TCP has an additional handshake and buffer procedures included in its process;

TCP

```
data_stream =
↪ obj.connection.recv(1024)
buf += data_stream
if b"_split_" not in buf:
    continue
else:
    buf_decoded = buf.decode()
    split =
    ↪ buf_decoded.split("_split_")
    data = split[0].encode("utf-8")
    buf = split[1].encode("utf-8")

    try:
        if self.encrypt:
            data = self.fernet.decr
            ↪ ypt(data)
        msg = pickle.loads(data)
        if msg != None:
            obj.publisher.publish(m
            ↪ sg)
            warn = 1
        else:
            continue
```

UDP

```
data, addr = obj.soc.recvfrom(self
↪ .BUFFER_SIZE)
if self.encrypt:
    data = self.fernet.decrypt(data)
msg = pickle.loads(data)
if msg != None:
    obj.publisher.publish(msg)
    warn = 1
else:
    continue
```

The handshake process is automatic, while the buffer is included in the code. As explained earlier and shown in Figure 2.9, the extra time required for buffer processing may occur while the subscriber is processing a new message. This would then affect the results less, as the receiving end would be waiting for a new message instead of processing at this time if there had been no buffer.

4.2. Usability

The usability is rooted in how well the software can complete its task. This includes both performance and the ability for a new user to set up the software.

4.2.1. Internal and local network transmissions

Looking at the benchmarking experiments which were done on commercially available hardware, large messages had a limit of ∼900 Hz. Messages published when simulating a TurtleBot3 had a publishing rate of anywhere from 5 to 50 Hz, which is well within the performance of this software.

4.2.2. Bluetooth transmission

The publishing rate could be a problem when using Bluetooth, as larger messages had a limit of ∼5 Hz. Using Bluetooth would be most suited to low-power devices which monitor at a low rate and is not an option for real-time control or when a high publishing rate is required.

4.2.3. Software setup

Using the provided documentation and examples, setting up the software should be trivial to any novice ROS2 user. ReadTheDocs documentation is included in Appendix A.

4.3. Further work

4.3.1. Optimization of initialization message and callback function

As the ability to serialize and encrypt messages has been implemented, the act of creating and sending BridgeObjects from the client to the server during initialization should be possible. This would save time during server initialization, but this is not dependent on the total performance. Multiple BridgeObjects could be defined, custom to either sending or receiving data. This would reduce their size and customize the callback function based on protocol, making this a possible performance improvement.

4.3.2. Integration of services and actions

ROS2 is not limited to using topics as the only means of communication. Two different communication forms based on topics, services and actions, are also available. Typically, topics are continual information streams, while services only publish when requested. The request and answer are node-specific, meaning that multiple nodes have access to the same service, but only the node that sent the request gets an answer. Figure 4.1 shows two nodes connected to the same service node.

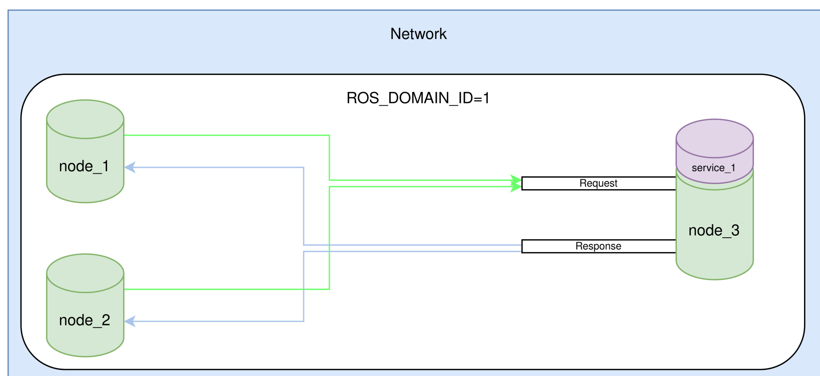


Figure 4.1.: Two nodes connected to the same service node.

Transmitting service requests is not as simple as transmitting a message. In order to even send a request, a service server has to be started. This server processes the request by putting the request through a function, which returns the answer. To transmit these messages across domains, the server or client of this stack would have to create a service that imitates incoming requests, calls the request in the local domain and returns the answer to the original domain.

One of the main issues to overcome is the fact that service servers and clients are created with set '.srv' messages in mind. Having a service that has no set request or answers messages is uncommon, if at all possible. The previously explored 'ros/domain_bridge' package, which is similar to this stack in many ways, has not found a solution to this issue.

Actions are built on both topics and services and therefore also pose a challenge to transmit across domains. An action is built on one or more services with the ability to provide constant feedback while a request is processed. Figure 4.2 shows two nodes connected by an action.

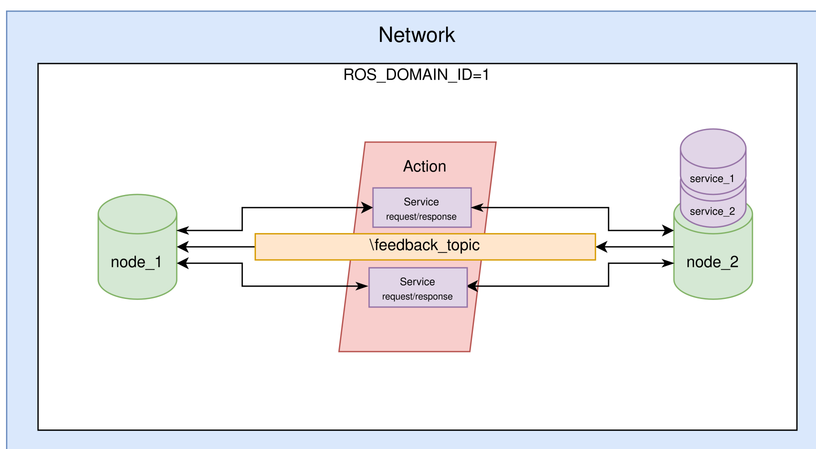


Figure 4.2.: Two nodes communicating using an action.

Implementing actions should be trivial once a method to implement services has been found.

This should be one of the main focus areas for the future of this software, as services and actions are widely used in multiple packages and are practical in many applications. The current workaround would be to establish two topics where one would be dedicated to requests, and a subscriber to the requested topic would send an answer on a response topic that is subscribed to by the original sender.

4.3.3. Remove the need to set specific topic sockets

To establish an initial connection between the server and client, a port needs to be specified. Without doing this, the client has no way of knowing where to send data. However, when establishing the remaining connections, it is possible to bind each BridgeObject to port 0. This causes an automatic port assignment to an available port, and removes the need for the user to set the ports manually. If this is to be done, the ports each BridgeObject binds to will have to be sent to the client over the main server-client connection.

This is possible and brings practicality to users but requires a rework of how the initialization procedure is performed.

4.3.4. Test the viability of low-powered devices

Utilizing low-powered computers such as Raspberry Pi could be viable, as the transfer speeds on relatively high-performance publicly available hardware is way

past requirements for most topics. This should be tested further.

Chapter 5.

Conclusion

The thesis set out to create a ROS2-based stack which further improves the DDS-based communication architecture. This was achieved by utilizing socket programming combined with built-in features of ROS2 such as publishers and subscribers. The major achievements can be itemized as follows;

- Created an open-source stack for ROS2 named 'ros2_socket_bridge' which transmit topics across domains or networks. This acts as an information bridge, enabling the creation of communication layers.
- Tested both practical and performance use of the stack, with benchmarks testing different configurations and real-life scenarios.
- Enabled Bluetooth transmission of ROS2-style messages by additional use of socket programming.

5.1. Comparison to 'ros/domain_bridge'

The most comparable competitor to 'ros2_socket_bridge' would be 'ros/domain_bridge'. The stacks are created with many similar goals in mind, but have a different approach.

5.1.1. Communication method

Domain_bridge (DB) creates nodes in both domains, adds a subscriber and a publisher to the 'from' and 'to' domain respectively, and makes the callback on the 'from' domain subscriber to publish the incoming message on the 'to' domain. Ros2_socket_bridge (SB) also uses nodes in different domains, but instead of using internal ROS2-communication to transfer the messages, socket programming is used.

5.1.2. Pros and cons

ros/domain_bridge

DB uses a straightforward way of transmitting messages, utilizing the simplicity of subscribers and publishers already existing in ROS2. With this comes a simple setup that only requires the topic name with 'from' and 'to' domain. This is optimal if simple internal topic transmission is the goal.

The developers of DB do not mention any tests done to explore the performance and reliability of the stack. In addition, it is developed for the 'Galactic' version of ROS2, and therefore not usable with earlier distributions. As this stack only uses subscribers and publishers, it is a requirement that the network that is running the bridge has multicasting enabled. This means that nodes can discover each other on the network, not just locally on the computer.

ros2_socket_bridge

SB uses sockets as the communication driver for its transmission of topics across domains. With this comes the ability to transmit on the local network and across the internet and use radio waves in the form of Bluetooth. As communication between nodes happens using sockets, having multicasting enabled is not a requirement. The stack has been thoroughly tested in both practical and performance use. It has been developed for the 'Foxy' version of ROS2, with little constraining it from being forward and backward compatible, as no significant changes are being done to the way topics are being communicated.

Setting up SB is more extensive than DB as IP for the server is required, ports for both the main server-client communication and each topic to be transmitted have to be set manually, together with the message type and QoS.

5.1.3. Significance in practice

DB is simpler to set up and achieves its purpose to the user if the goal is to transmit topics across domains on the local network. SB has more flexibility in its practical use-cases at the cost of setup time, such as Bluetooth integration and not being restricted to the local network.

References

[1] T. H.-J. Uhlemann, C. Lehmann, and R. Steinhilper, "The digital twin: Realizing the cyber-physical production system for industry 4.0," *Procedia Cirp*, vol. 61, pp. 335–340, 2017.

[2] A. Gilchrist, "Introducing Industry 4.0," in *Industry 4.0*, Springer, 2016, pp. 195–215.

[3] H. Smajic and N. Wessel, "Remote Control of Large Manufacturing Plants Using Core Elements of Industry 4.0," in. Jan. 2018, pp. 546–551, ISBN: 978-3-319-64351-9. DOI: 10.1007/978-3-319-64352-6_51.

[4] "An Overview Of Distributed Control Systems (DCS)," [Online]. Available: https://www.plantautomation-technology.com/articles/an-overview-of-distributed-control-systems-dcs (visited on 05/19/2021).

[5] *Hello Industrie 4.0*. 2019. [Online]. Available: https://www.kuka.com/en-us/future-production/industrie-4-0/industrie-4-0-introduction (visited on 05/10/2021).

[6] M. Quigley, "ROS: an open-source Robot Operating System," in *ICRA 2009*, 2009.

[7] S. Macenski, F. Martín, R. White, and J. Ginés Clavero, "The Marathon 2: A Navigation System," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020. [Online]. Available: https://github.com/ros-planning/navigation2.

[8] "The MoveIt Motion Planning Framework for ROS 2," [Online]. Available: https://github.com/ros-planning/moveit2 (visited on 05/31/2021).

[9] A. Corsaro and D. Schmidt, "The Data Distribution Service | The Communication Middleware Fabric for Scalable and Extensible Systems-of-Systems," in. Mar. 2012, ISBN: 978-953-51-0101-7. DOI: 10.5772/30322.

[10] W. Woodall, *ROS on DDS*, 2019. [Online]. Available: https://design.ros2.org/articles/ros_on_dds.html (visited on 03/05/2021).

[11] "Restricting communication between robots," [Online]. Available: https://discourse.ros.org/t/restricting-communication-between-robots/2931 (visited on 02/12/2021).

[12] G. Pardo-Castellote, "OMG Data-Distribution Service: architectural overview," in *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, 2003, pp. 200–206. DOI: 10.1109/ICDCSW.2003.1203555.

[13] E. Erős, M. Dahl, K. Bengtsson, A. Hanna, and P. Falkman, "A ros2 based communication architecture for control in collaborative and intelligent automation systems," *Procedia Manufacturing*, vol. 38, pp. 349–357, 2019.

[14] M. Dahl, K. Bengtsson, P. Bergagard, M. Fabian, and P. Falkman, "sequence planner: Supporting integrated virtual preparation and commissioning," pp. 5818–5823, Jul. 2017.

[15] K. E. Foltz and W. R. Simpson, "Enterprise considerations for ports and protocols," JSTOR, Tech. Rep., 2016.

[16] L. Chappell, "Inside the TCP handshake," *NetWare Connection*, 2000.

[17] R. Saha, *TELECOMMUNICATIONS ENGINEERING - Technical Note.* Jul. 2016.

[18] *Programming Python: Powerful object-oriented programming.*

[19] E. Sangaline. "Dangerous Pickles — Malicious Python Serialization," [Online]. Available: https://www.networkworld.com/article/2224213/why-the--trombone--effect-is-problematic-for-enterprise-internet-access.html (visited on 05/25/2021).

[20] S. Bray, *Implementing Cryptography Using Python*, 1st ed. Wiley, 2020, ISBN: 1119612209,9781119612209.

[21] N. Verma, M. kashyap, and A. Jha, "Extending Port Forwarding Concept to IoT," in *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, 2018, pp. 37–42. DOI: 10.1109/ICACCCN.2018.8748430.

[22] A. Gottlieb. "Why the 'trombone' effect is problematic for Enterprise Internet access," [Online]. Available: https://www.networkworld.com/article/2224213/why-the--trombone--effect-is-problematic-for-enterprise-internet-access.html (visited on 05/24/2021).

[23] E. J. Weyuker and F. I. Vokolos, "Experience with performance testing of software systems: issues, an approach, and case study," *IEEE transactions on software engineering*, vol. 26, no. 12, pp. 1147–1156, 2000.

[24] E. W. Grafarend, *Linear and nonlinear models: fixed effects, random effects, and mixed models.* de Gruyter, 2006.

Appendix A.

ReadTheDocs

A.1. Front Page

Welcome to the documentation of ros2_socket_bridge!

This is a package developed for ROS2 'Foxy' which allows the utilization of Python sockets in ROS2, with the main goal being transmission of topics between domains. It can be used for multiple purposes, and could add an additional communication layer on top of ROS-implemented DDS.

An example use case being a control domain and multiple robot domains. The domains important to monitor the robots are transmitted from the robot domain to the control domain, and tasks are sent to each robot individually. This would allow for easier control of robot fleets with less namespace clutter and setup.

Contents:

- Quick-start guide
  - Server setup
  - Client setup
- Internal workings
  - Initialization
  - UDP sockets
  - TCP sockets
  - Bluetooth sockets
- Testing and benchmarking
  - Serialization and encryption
  - Decryption and deserialization
  - Maximum publishing rate
- License

v: latest

A.2. Quick-start guide

Quick-start guide

This guide will let you quickly start using ros2_domain_bridge. To start off, we need to select our configuration based on the intention of using this package.

If you are intending on using this as a domain bridge, the most proficient way would be to transfer topics locally on your computer. That would base both the client and server running on the same computer, but on different ROS_DOMAIN_ID's. In this case, the server_ip setting would be set to '127.0.0.1'.

If you are intending on transferring between computers, set the IP of the server computer as something memorable and static. This will later be the setting for server_ip. If you are using a VPN tunnel or other means of transferring data across the internet, make sure the server computer's IP is static.

If you are planning on transferring messages using Bluetooth, you first need to connect the server and client devices. Once this is done, find the MAC address of the Bluetooth attached to your server device. This is done in Linux by running 'hciconfig' in the terminal. Alternatively, just run the 'check_bluetooth_MAC_and_ports.py' script located inside the project folder. This will give you the MAC address and busy ports on the device. The MAC address will later be used as the server_ip variable.

Server setup

To configure the server, simply head into 'src/rsb_server'.

First off, we will configure the 'bringup.yaml' file, located inside the config folder. Simply set the server_ip to be either an IP address or MAC address, based on your intention.

Navigate back into 'rsb_server' and enter 'launch'. Locate the 'server.launch.py' file. First, we need to set the name of the process. This name has to match with the client, and can be used as a prefix to the topics received. Secondly, set the server_port variable. This variable has to be an open port, best suited to any number between 49152 and 65535. If you are using Bluetooth, the ports to be used should not be listed when running the 'check_bluetooth_MAC_and_ports.py' script. Next, we select whether or not to use the name as a prefix for incoming topics. As an example, if we set the name to be 'robot1', and we are to receive a topic '/goal_pose', the name of the incoming topic would be '/robot1/goal_pose'. We also have to select if we want to use encryption or not. The encryption key could be any 32 url-safe base64-encoded bytes object, which can easily be generated using the 'generate_key.py' script in the main folder. This has to match with the client.

That's it for server setup. The topics to be transferred and their settings are all defined client-side.

Client setup

Head to 'src/rsb_client'

First, we will configure the 'bringup.yaml' file inside the 'config' folder. As with the server, we need to set the server_ip and server_port variables. We then need to select which topics to transmit and receive. Every setting exept for *_ports should be inside a string. The receive_* settings are for topics which are received on the client, and transmit_* are for topics which are sent from the client to the server. Each topic has its belonging message type, added to the *_msg_types variable. The port to be used is in the *_ports list, the protocol for transmission is in the *_protocols list, and finally the QoS to be used by the publisher is in the *_qos list. The first inquiry in each list is related. As an example, here is a configuration which sends '/scan', '/odom' and '/shutdown' topics, and receives '/initialpose', '/goal_pose' and '/shutdown'.

```
receive_topics: ['scan', 'odom', 'shutdown']
receive_msg_types: ['LaserScan', 'Odometry', 'String']
receive_ports: [12004, 12005, 12006]
receive_protocols: ['UDP', 'UDP', 'TCP']
receive_qos: ['qos_profile_sensor_data', 'qos_profile_sensor_data', '10']
```

```
transmit_topics: ['initialpose', 'goal_pose', 'shutdown']
transmit_msg_types: ['PoseWithCovarianceStamped', 'PoseStamped', 'String']
transmit_ports: [12002, 12012]
transmit_protocols: ['UDP', 'UDP', 'TCP']
transmit_qos: ['qos_profile_system_default', 'qos_profile_system_default', '10']
```

If you are going to use Bluetooth to communicate the topics, you need to set all *_protocols to 'BLUETOOTH'.

Once the configuration file is finalized, we need to set up the launch file. Head to 'src/rsb_client/launch/client.launch.py'. We first need to set the name which must match with the name used in the server node. Additionally, we need to set whether or not to use name as a prefix, to use encryption, and the encryption key which must match with the server.

Once this is done, both server and client should be ready to go.

v: latest

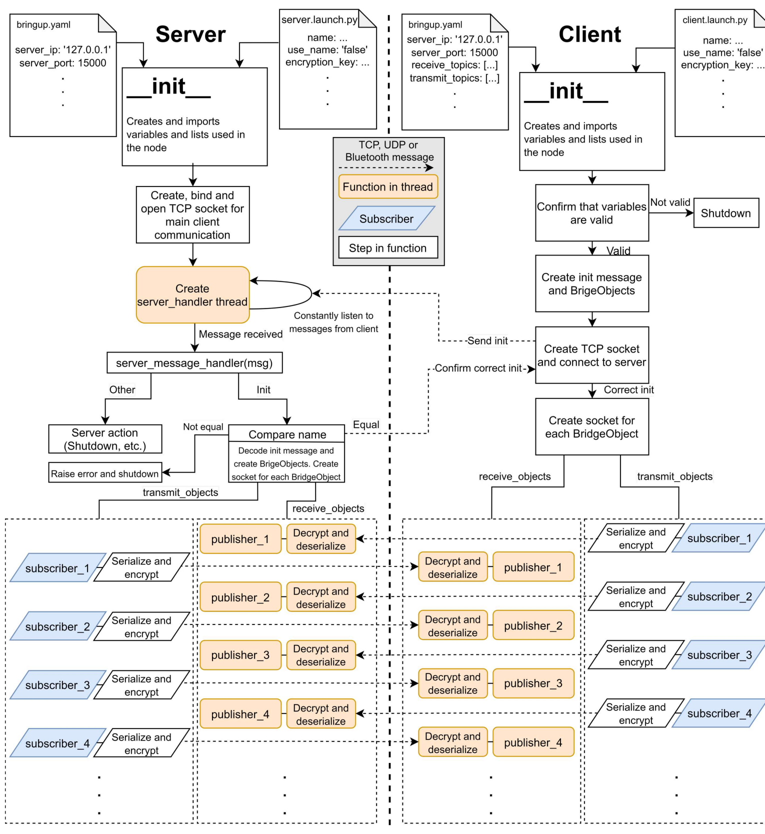
A.3. Internal workings

Internal workings

The only meaningful difference between the server and client is that the ports are bound server-side, and that the server-client relationship is configured on the client. Both server and client reads from a configuration file named 'bringup.yaml', and additional settings such as the encryption key is set in the launch files.

Initialization

The configuration file of the client contains all settings for transmitting and receiving messages from the server. Topics that are set to be received or transmitted are added to a string, and sent as a byte object to the server using the main communication line. From here, both server and client creates BridgeObjects for each topic to be sent or received. These BridgeObjects contain information about the connection such as direction, encryption key, topic name, message type, port, protocol, qos and encryption setting. When starting connections, socket and connection objects are also attached to the BridgeObject. In addition, a callback function exists so that the BridgeObject is referenced when creating subscribers for outgoing topics.



Topics that are to be received are running on threads which contain a while-loop, receiving function, deserializer, decryptor and publisher. Once a serialized and encrypted message is received, it is first decrypted, deserialized and then published to its correct topic.

Topics that are to be sent is only started as a subscriber with the callback function being inside the BridgeObject belonging to said topic. The callback function serializes, encrypts and sends based on protocol.

v: latest

UDP sockets

When using UDP as transmission protocol, sockets are set up with the following settings;

```
obj.soc = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
obj.soc.settimeout(15)
obj.soc.setsockopt(socket.SOL_SOCKET, socket.SO_RCVBUF, self.BUFFER_SIZE)
```

It is an UDP (SOCK_DGRAM) socket using IPv4 (AF_INET), with a buffer-size of self.BUFFER_SIZE. The timeout is set to 15 seconds. Every 15 seconds of no data, the user receives a warning that the topic is stale. Once the warning has been given 5 times, the warnings stop.

TCP sockets

The TCP sockets use the following settings;

```
obj.soc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
obj.soc.settimeout(15)
obj.soc.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

SOCK_STREAM indicates TCP, and it is set to SO_REUSEADDR. This allows reconnection to said socket, which is useful in cases where the socket needs to be rebooted.

TCP sockets are streaming sockets, meaning that they continually sends data. To ensure that the data we receive is split in the correct manner, a b'_split_' is added to the end of each message when sent. On the receiving end, a buffer continually checks to see if '_split_' is in the buffer. If it is, the buffer is split into two parts. The first part is out message, and the second part is the start of a new message. The first part is decrypted, deserialized and published, while the second part is re-added to the now empty buffer.

Bluetooth sockets

As with TCP, SOCK_STREAM is used. This is due to the low reliability of Bluetooth, ensuring that the messages we receive are intact. In addition, BTPROTO_RFCOMM ensure that the address we pass in is (bdaddr, channel), where as it normally is (ip, port).

```
obj.soc = socket.socket(socket.AF_BLUETOOTH,socket.SOCK_STREAM,socket.BTPROTO_RFCOMM)
obj.soc.settimeout(15)
obj.soc.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

Bluetooth sockets also use a buffer, just like TCP sockets do.

A.4. Testing and benchmaking

Testing and benchmarking

As this software was made in relation to a masters thesis, testing and benchmarking was done. Time it took to serialize and encrypt, deserialize and decrypt, and maximum topic publishing rate using different protocols and connection modes was tested.

Details about the two computers used during testing can be seen here;

Name	Processor	Clock speed	Cores	Threads
Intel NUC	Intel® Core™ i7-8705G	3.10-4.10 GHz	4	8
Surface Book 2-in-1	Intel® Core™ i5-6300U	2.40-3.00 GHz	2	4

Serialization and encryption ¶

Variable	Message type			
	Laser	TF	Costmap	Footprint
Serialization				
Mean (μ) [ms]	0.032	0.039	0.056	0.044
Standard deviation (σ) [ms]	0.005	0.003	0.026	0.007
Encryption				
Mean (μ) [ms]	0.152	0.142	0.196	0.141
Standard deviation (σ) [ms]	0.01	0.004	0.045	0.012
Total				
Mean (μ) [ms]	0.184	0.181	0.252	0.185
Standard deviation (σ) [ms]	0.011	0.005	0.05	0.014

Decryption and deserialization

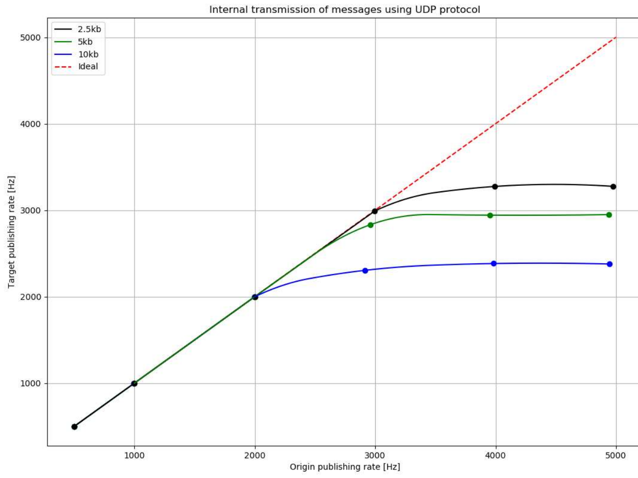
Variable	Message type			
	Laser	TF	Costmap	Footprint
Decryption				
Mean (μ) [ms]	0.333	0.252	0.242	0.299
Standard deviation (σ) [ms]	0.023	0.007	0.042	0.023
Deserialization				
Mean (μ) [ms]	0.066	0.061	0.059	0.074
Standard deviation (σ) [ms]	0.011	0.004	0.024	0.013
Total				
Mean (μ) [ms]	0.40	0.313	0.301	0.373
Standard deviation (σ) [ms]	0.025	0.008	0.046	0.025

Maximum publishing rate

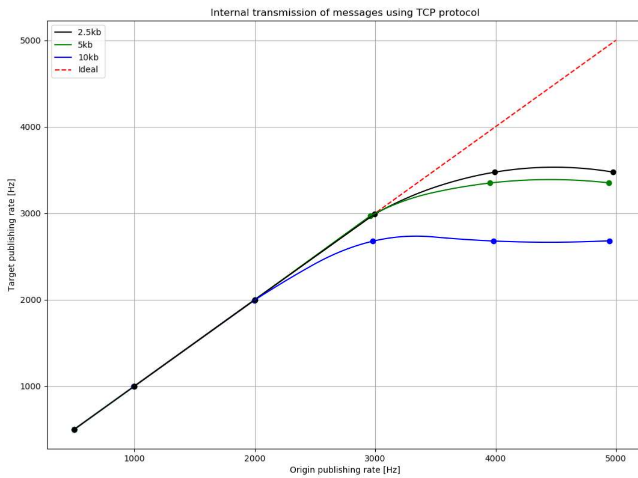
The maximum publishing rate was tested in relation to message size.

Internal transmission using UDP:

v: latest

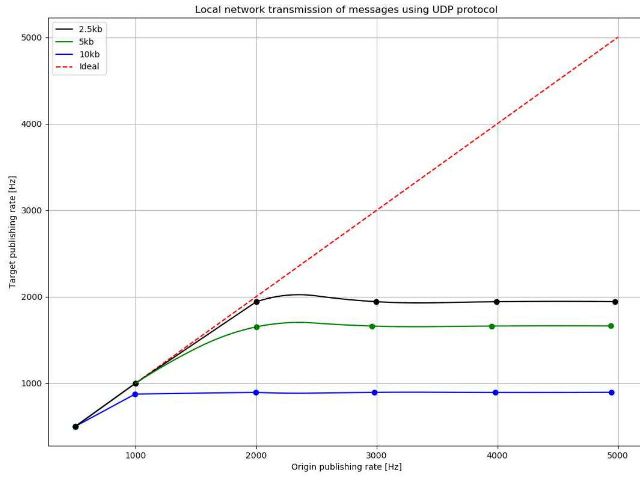


Internal transmission using TCP:

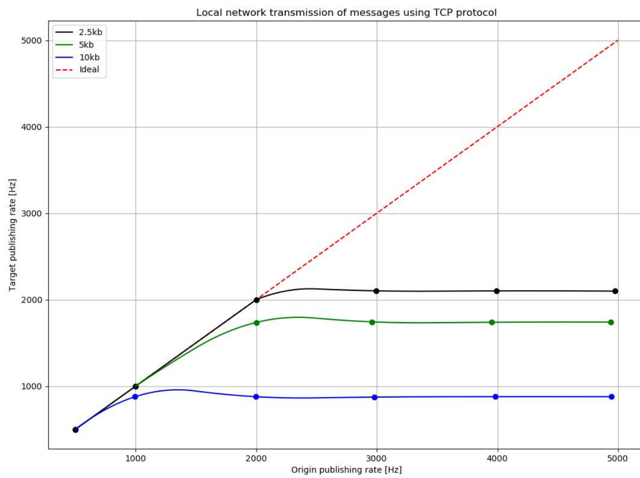


External (between computers on local network) transmission using UDP:

v: latest

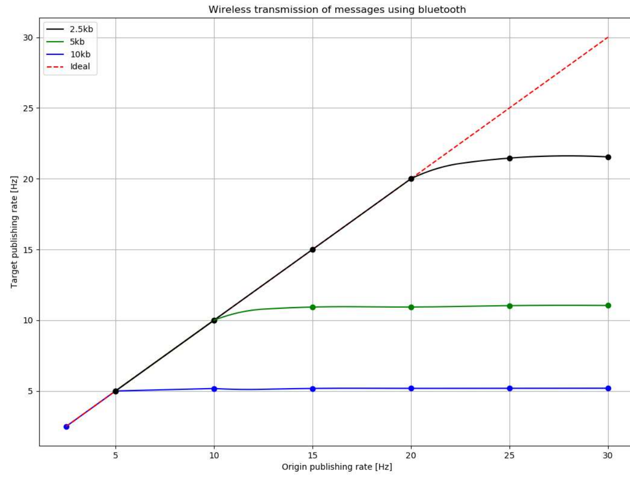


External transmission using TCP:



External transmission using Bluetooth:

v: latest



v: latest

A.5. License

License

Apache License Version 2.0, January 2004 http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   a. You must give any other recipients of the Work or Derivative Works a copy of this License; and
   b. You must cause any modified files to carry prominent notices stating that You changed the files; and
   c. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
   d. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, includ-

v: latest

ing any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

> To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

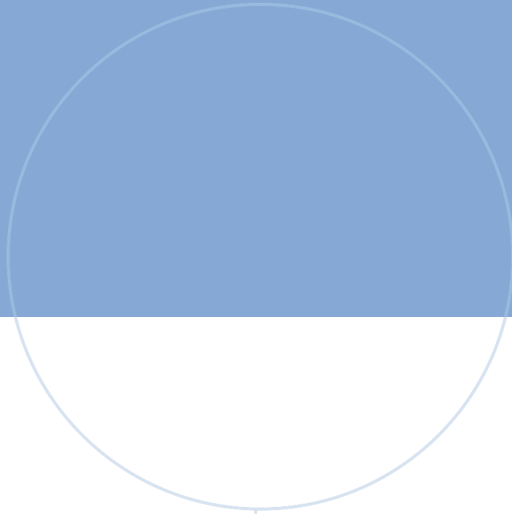
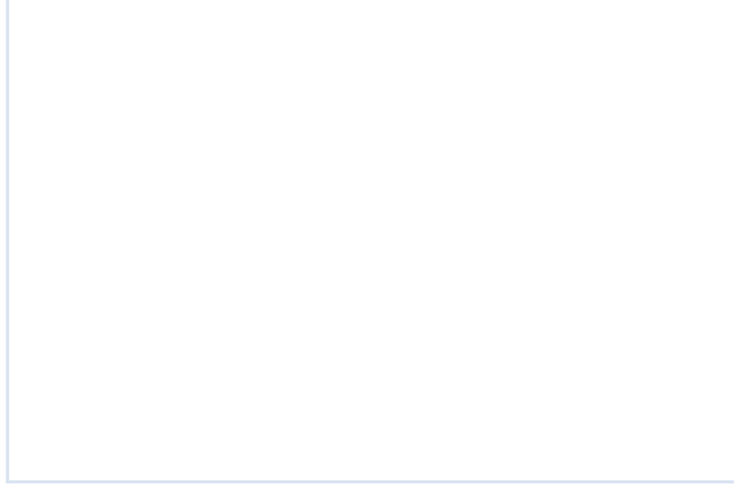
Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

> http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

v: latest

NTNU

Norwegian University of
Science and Technology