Politecnico di Torino

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Optimizing Inter-Process Communication for Robotics Applications

Relatori
Prof. Giorgio Guglieri
Prof. Fulvio Risso

Candidato
Andrea Fasano

Ottobre 2025

Abstract

Robots have become increasingly complex and computationally capable, resulting in equally complex software architectures composed of numerous modules that must exchange large volumes of data. ROS and its successor, ROS 2, have become the de facto standards for managing interprocess communication (IPC) in robotics. However, in its default configuration, ROS 2 suffers from significant performance degradation when communicating over unreliable networks, negatively impacting both local and remote communication. Additionally, its overhead grows substantially with the number of nodes and communication interfaces. This thesis proposes an alternative IPC solution that is fast, reliable, deterministic, efficient, and easy to use, with a strong focus on robust local communication. The solution comprises a library for local IPC and a gateway application for extending communication across networks. Experimental results demonstrate that the proposed approach effectively mitigates the performance issues of ROS 2 under unreliable network conditions. Compared to alternative ROS 2 middleware configurations addressing similar challenges, the solution consistently delivers lower latency, reduced CPU and memory usage, and improved scalability.

Contents

Chapter 1

Introduction

1.1 IPC in Robotics

Over the years robots became increasingly complex machines. In particular as technology progressively allowed to fit more computational power inside robots, the size and complexity of software controlling their functions grew along side it.

An autonomous robot can need many different capabilities, ranging from the acquisition of data from external sensors, to all kinds of processing of said data, in order to achieve the desired behavior. The data streams between different software components of a robot can be extremely significant. It is common for robots to have multiple high resolution cameras, as well as other kinds of information-dense sensors such as LIDAR.

Data coming from sensors can be used for a variety of things, some vital to the functioning of the robot (safety-critical) some less so. The many different components can have varying degrees of real-time requirements. For example an obstacle avoidance algorithm based on information coming from a camera can have different real-time requirements based on the speed of the robot and the ability to change direction [1].

A very common way of structuring software for robotics prototyping is to have separate processes take care of different tasks. This on one hand follows the UNIX maxim "Make each program do one thing well" [2], on the other decouples separate components that may have varying degrees of importance for the safety of the robot. Decoupling here simply means that if one component unexpectedly crashes, the other can keep running, which would not be the case if they belonged in the same process.

While operating systems provide ways to exchange data between processes (section 4.2.1), chief among them the file-system, setting up such Inter Process Communication (IPC)in an efficient and safe way is not nearly as straightforward as it would be to communicate data within a process. Implementing a custom communication solution for each specific use-case may yield the best performance but is simply unfeasible when project size grows past a certain point. It becomes therefore necessary to standardize communication among software components.

In the field of robotics the by far most successful and very widely used framework to facilitate Inter Process Communication is the Robot Operating System (section 2.1) and in more recent years its successor ROS 2 (section 2.2).

1.2 DRAFT Team

The DRAFT(DRone Autonomous Flight Team) team is a student team at Politecnico di Torino [3]. It focuses on the development of autonomous drones, from the design and construction to

software development. The team concurrently develops several projects, but its main focus since foundation has been the Leonardo Drone Contest competition (see section 1.2.1). The team is subdivided in four technical divisions:

- Robot System Engineering (RSE)
- Simultaneous Localization And Mapping (SLAM)
- Deep Learning and Computer Vision (DLCV)
- Obstacle Avoidance and Motion Planning (OAMP)

Since the focus of the team is autonomous flight, the majority of members are tasked with developing software that enables the many features required. This means that the combined software running on the robots is the result of the work of many different people, with varying degrees of ability and different backgrounds.

1.2.1 Leonardo Drone Contest

The Drone Contest (LDC) organized by Leonardo is a competition that involves seven Italian universities [4]. The main challenge is to navigate an unknown environment with a drone in the absence of GNSS signal.

The competition evolved over the years to require increasingly complex solutions, with the latest editions featuring a drone-rover pair cooperating to solve a number of tasks within the environment. The challenge requires robots to perform a number of tasks such as:

- Landing in specific positions.
- Tracking a moving object on the ground.
- Identifying ArUco markers and estimating their position.
- Identifying a number of objects present in an object database and estimating their position.
- Mapping the environment.
- Coordinating different robots to perform cooperative tasks.

The separate tasks often require specific technological solutions but for them to work the vital requirement is the ability to navigate the obstacle filled environment safely. This is especially complicated due to the absence of GNSS signal and the regulation forbidding the use of certain sensors such as LIDAR on the drone. As a consequence the principal mean of localization used by all participants in the competition is Visual Inertial Odometry.

The nature of the competition environment and the different tasks also make the navigation particularly challenging. The computer vision and object detection tasks are especially computationally intensive, and Visual Inertial Odometry, while sometimes performed directly by the sensor, can also be a very significant computational load. This means that competitors have to carry a fairly expensive payload, both in terms of weight and power consumption. These requirements effectively limit the minimum size of the drone. Given the design and positions of the obstacles within the environment, the drone has to maneuver through gaps that are only marginally wider than the robot itself. As a consequence, a very precise and stable localization is required, both to avoid unwanted drifts and to minimize noise when mapping the obstacles.

Due to the many different capabilities required by this challenge to perform the tasks effectively, many separate software modules have to be running and communicating at all times. Some of these are especially safety-critical and have stringent real-time requirements, such as Visual Inertial Odometry, obstacle avoidance logic and communication with the flight controller.

1.2.2 Software IPC Requirements

As mentioned in previous sections, the combined software running on the robots developed by DRAFT, especially for the LDC competition, consists of many semi-independently developed software modules. In general, they perform operations on data originating from sensors. There are specialized modules dealing with the sensors themselves and extracting data. The sensor data is then processed by one or multiple modules, with data dependencies between modules being quite complex at times.

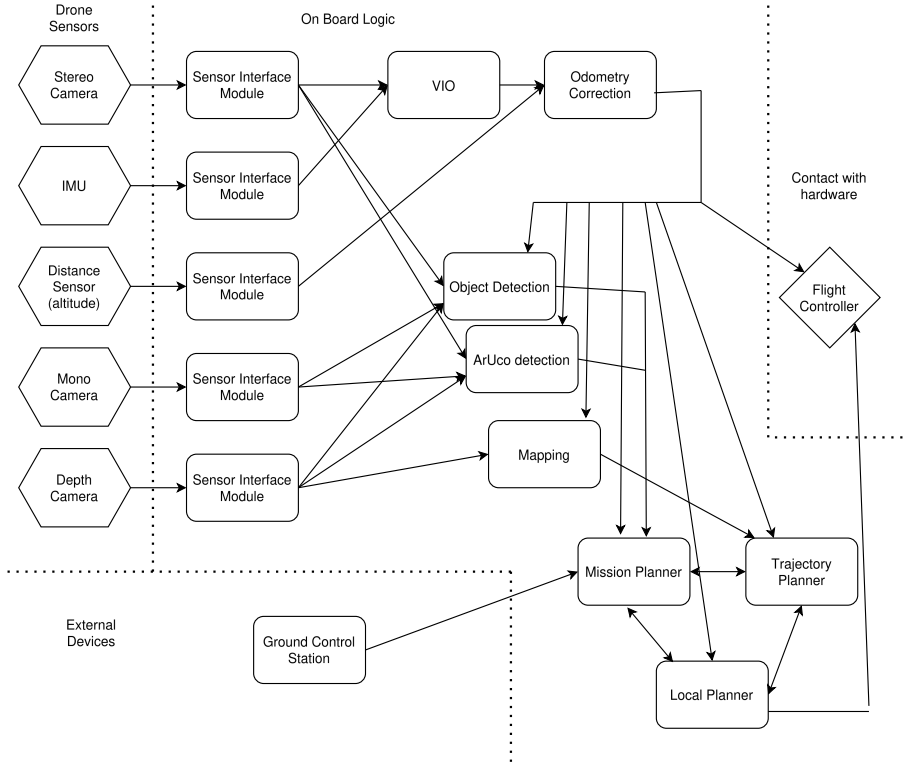

Figure 1.1. Simplified structure of software modules and data paths on an autonomous drone for the LDC competition.

Figure 1.1 shows a simplified example of the software running on a drone during the LDC competition. Due to the nature of development, many of these modules (especially newer ones) can be unstable. It is therefore mandatory to separate them in different processes to make sure that one does not go down with the other. This unfortunately complicates the communication between them.

The data exchanged between processes can be anything from sensor data, status information, commands and more. As shown in table 1.1 the messages exchanged can vary significantly in terms of size and frequency.

At DRAFT we have been developing software for autonomous drones for several years. When starting out, the obvious solution to IPC problems was ROS (see section 2.1). It already had effectively become the standard to organize and facilitate robotics software development. In addition, thanks to its popularity came many community packages that it was possible to use to fill knowledge gaps.

Message Type	Frequency (Hz)	Size (Bytes)	Source	Usage
IMU	100–1000	48–100	IMU Sensor	Orientation, acceleration, control loops
Odometry	10–200	100–200	Localization	Robot pose estimation and navigation
Camera Image	10–120	100k–5M	RGB/D Camera	Perception, vision-based navigation
Laser Scan	5–30	10k–800k	LiDAR	Obstacle detection, mapping
Point Cloud	1–30	1M–10M	Depth Camera, LiDAR	3D mapping, perception
Fused/ Synchronized Sensor Data	10-60	1M-20M	Sensors/Synchronizer	Processing, object detection, obstacle avoidance
Diagnostics	0.1–10	100–1k	System Nodes	Debugging and health status
Control	1-100	10-1k	Control nodes	Hardware

Table 1.1. Non exhaustive list of common message types in robotics

1.2.3 Issues with ROS communication

ROS undoubtedly proved vital in allowing the first prototypes to work and especially in enabling integration between different modules and sensors. With passing time however, the team was able to progressively erode the mountain of external code it depended on. More and more modules were being substituted with custom made ones. Additionally, with the evolution of the LDC competition and the shift in focus to multi-agent exploration, the communication requirements started to shift from pure local IPC, to a mix of network and local communication.

The competition organizers, starting from the 4th edition, also mandated a switch from ROS to ROS2 (see section 2.2). This made a lot of sense, especially for robots like the ones in the LDC competition, since two of the main improvements ROS2 was supposed to bring were fault tolerance and better real-time performance. Unfortunately however the combination of ROS 2 and network communication (especially unreliable and disturbed connections) proved disastrous for the overall performance of the communication system (Section 6.2).

The amount of time dedicated to understanding the communication issues and trying to find workarounds quickly overtook that dedicated to actual development. The general unreliability of communication (both local and network) caused non-deterministic testing and a remarkable amount of wasted time and resources. The problems experienced by DRAFT were shared by all other universities participating in the competition.

First attempts to mitigate the problem were directed not a ROS 2 itself but rather at improving the network conditions, reducing traffic and limiting interaction between agents over the network. Overall however, ROS was becoming more of a liability than an asset. This together with the tendency of ROS to be an all-conquering library (effectively forcing a pre-determined project structure, package manager, build system and so on) became a significant hinder to development.

Furthermore ROS 2 in its default configuration suffers from a significant overhead when increasing the amount of nodes and interfaces communicating, both locally and over the network. This can be mitigated by leveraging manual or dynamic node composition [5], but this solution has the fundamental disadvantage of running multiple nodes within the same process, which

effectively removes the safety net of separating software modules.

It was possible to consider switching to an alternative to ROS also thanks to the team's reduced reliance on ROS community packages. A less mature project would have struggled a lot more to make this kind of jump.

1.3 Thesis Goal

The goal of this thesis is to develop an alternative solution for Inter Process Communication, both on the local and network level.

The found solution should allow to set up IPC that is:

- Fast
- Reliable
- Deterministic
- Efficient
- Easy to use
- Flexible

The found solution should be compared to the current standard in robotics (i.e. ROS 2) and prove itself to be comparable in capabilities while solving the problems in previous and following sections.

Chapter 2

Related Work

2.1 ROS

The Robot Operating System (ROS) [6] is a framework designed to facilitate the development of software for Robotics. Despite its name ROS is not an operating system. It provides a communication layer on top of the underlying operating system, that allows separate software components (generally called nodes) to exchange data and coordinate behavior.

ROS communication architecture at its core supports:

- Topics: based on the publish-subscribe model, they allow asynchronous exchange of messages. Their are typically employed to make streams of sensor or processed data available to other software modules.
- Services: based on the remote procedure call (RPC) model, they allow nodes to make requests to other nodes, trigger a procedure on the node providing the service and receive a response.
- Actions: built on top of topics, they provide a structured protocol for long-running tasks.

ROS deals with node discovery in a centralized manner, with a master node acting as a intermediary in registration and communication setup between nodes [7]. The master node can also serve as a parameter server.

Over time ROS has grown to include not just the core communication component, but also a number of tools for monitoring, visualization, recording and playback, orchestration. With its growing popularity [8] ROS has also acquired its biggest strength: a very large quantity of community-developed packages, that implement a wide range of key features in robotics.

2.2 ROS 2

While ROS has significantly simplified research and development in many fields of robotics, it presented a number of shortcomings especially when research work transitioned to production. These range from failing to match real-time requirements to lack of built-in security support and limited fault tolerance. [9, 10]

ROS 2 was developed from scratch to address these limitations and it represents a significant departure from ROS implementation-wise. A major design change was to base the architecture on the Data Distribution Service (DDS) standard, meaning communication is peer-to-peer and decentralized. This gives ROS 2 access to features such as Quality of Service policies, native

security support and removes the single point of failure represented by the ROS master node. ROS 2 also introduces support features for real-time execution and leverages DDS to improve fault tolerance through liveliness detection and recovery mechanisms. Thanks to the promising feature it brings to the table and the legacy of ROS, ROS 2 has quickly risen to prominence. [11]

2.2.1 ROS Middleware

Importantly, ROS 2 does not directly implement the DDS communication, but rather builds its own messaging layer on top of an abstraction. This allows developers to choose among multiple DDS implementations (ROS middleware or RMW). The most widely used DDS implementations are FastDDS [12] and CycloneDDS [13]. These implement DDS and follow its philosophy of decentralized, peer-to-peer communication. DDS implementations use RTPS over multicast UDP, both communication and discovery are peer-to-peer, although they may need discovery servers to traverse NATs.

Very recently, a new RMW implementation, Zenoh RMW [14], has been released. Zenoh RMW builds on Zenoh (Zero Overhead Network Protocol), and while it allows for a substantial amount of configuration options, its most meaningful change is allowing the ROS 2 network communication to be handled by a single "router" node per machine.

It has been repeatedly shown that ROS 2 communication over DDS RMW suffers particularly over unreliable networks and that Zenoh used as a bridge or Zenoh RMW show significantly better performance in such conditions [15, 16].

In general most ROS 2 middlewares make use of loopback UDP sockets to implement local Inter Process Communication, this gives them the flexibility to be able to use the same mechanism to communicate with both local and remote peers. However it is debatable whether this strategy is optimal for local IPC.

2.2.2 FastDDS

The FastDDS middleware is the most widely adopted middleware in the ROS 2 landscape as of now. This is mostly due to it being the default option with most ROS 2 versions.

FastDDS uses the DDS protocol over UDP. It implements all the necessary systems such as discovery and message exchange according to the DDS protocol. It uses SPDP (Simple Participant Discovery Protocol) and SEDP (Simple Endpoint Discovery Protocol) to perform discovery between nodes and endpoints(publishers/subscribers). User data is then exchanged between nodes using RTPS messages.

FastDDS uses UDP sockets as its primary and only mean of communication in its default configuration. It does allow to explicitly enable shared memory communication between nodes on the same machine but this is hard to use in practice due to issues both in terms of reliability (the current implementation does not protect the shared memory against erroneous access) and in terms of complexity of configuration (for example maximum message size needs to be explicitly configured).

Communication over UDP socket naturally requires an extra copy to go from a user space memory buffer to a kernel space one. This is pretty much unavoidable for network communication, but is entirely unnecessary when communicating locally.

The DDS protocol lacks native support for flow control, which is a major drawback when transmitting over unreliable or bandwidth constrained networks. With all communication, including discovery and control messages, taking place over UDP and no control over bandwidth utilization, a saturation of resources caused by one feature can easily and significantly impact the others.

2.2.3 Zenoh RMW

Zenoh RMW is a relatively new addition to the ROS 2 middleware ecosystem, offering a fundamentally different approach compared to traditional DDS-based solutions.

Zenoh does not implement the DDS protocol. Instead, it provides its own minimal, efficient protocol optimized for scalability and low-latency communication.

Unlike DDS, Zenoh includes built-in support for flow control, congestion control, and reliability, and can operate efficiently in both reliable and unreliable network environments. It employs a publish-subscribe model similar to DDS, but is more flexible in terms of routing and resource utilization. The system can route messages through brokers, peer-to-peer connections, or a combination thereof, depending on the deployment needs.

Discovery in Zenoh is also different from DDS. Rather than relying on multicast-based protocols like SPDP and SEDP, Zenoh employs a more lightweight and extensible discovery mechanism based on resource identifiers and name-based addressing.

Zenoh RMW also supports optional shared memory communication that has to be explicitly enabled by the user. However its implementation partly suffers from the same issues that plague the FastDDS implementation. Synchronization primitives are stored directly in shared memory and are as such not protected from malign or erroneous access.

While Zenoh RMW is not yet the default in ROS 2, it is gaining traction due to its performance advantages and flexibility, particularly in unreliable network scenarios. Being a very recent addition, the ecosystem around Zenoh is still maturing, and some ROS 2 features that tightly integrate with DDS-specific behavior may not yet be fully supported.

2.3 ROS messages

One of the original design pillars of ROS was for it to be Multi-lingual [6]. This naturally lead to having to abstract away message type definition from any specific language. In particular ROS uses a interface definition language (IDL) to describe the message types sent between modules. The IDL is based on a few base types and can then be composed to define more complex data structures. ROS 2 has not steered away from this choice, as such all considerations regarding message definition and serialization apply to it as well.

The decision to abstract message types away from the used programming language has several advantages: standardization, no reliance on language characteristics, compatibility.

Below in Listing 2.1 is an example of what a ROS message definition looks like. [17]

Listing 2.1. ROS Image Message Definition

```
std_msgs/Header header
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

2.3.1 Message Serialization

When messages are transmitted between nodes it is necessary to serialize them. This is imperative because:

- If communication happens between different languages or between different ABIs of the same language the serialization works as a mutually intelligible version of the data.

- It is imperative for a modern library to support dynamically allocated data structures. These hold data that can be dispersed in many separate allocations. The serialization thus takes care of putting all data necessary to interpret the structure in a single chunk of memory.

Messages are thus transmitted in their serialized form. This effectively requires two full copies of the message content, one at the sender side and one at the receiver.

2.3.2 Limitations in the Approach of ROS Messages

While there are advantages to having a language-independent message definition, there is one main drawback. The language specific types that are generated from the IDL representation serve effectively only the purpose of transmission, and are inconvenient if not impossible to use inside application logic.

To go back to the Listing 2.1 example, the raw image type used for message transmission in ROS is generated in the C++ binding as shown in Listing 2.2.

Listing 2.2. ROS Image C++ generated message

```
struct Image
{
        std_msgs::msg::Header_<ContainerAllocator> header;
        uint32_t height;
        uint32_t width;
        std::basic_string<char> encoding;
        uint8_t is_bigendian;
        uint32_t step;
        std::vector<uint8_t> data;
}
```

As evidenced in Listing 2.2 the most relevant field of the image data structure (the data field, which was a variable size array of uint8 in the IDL definition), is implemented in the C++ generated message type as a std::vector, making it type safe and easy to deal with in language. It is however clear that this layout of the image data structure is not viable for most purposes other than very simple manipulation. Because of this the vast majority of ROS packages, that deal with data structures of any significant complexity, perform a further copy, to go from the application layout of the data to the ROS message layout. This brings the effective total number of copies from one side to the other to four.

ROS actually provides ways to customize generated messages and make user-defined data structures transmissible.[18] However this is impractical at best, as the requirements to fulfill are many and they change over time.

As such most developers choose the overhead of two further copies over the increased complexity and potential incompatibility of developing custom language-specific types. A relevant example of a widely used package that made this choice would be the Realsense ROS package, used to interface with the Realsense Camera sensors. [19]

Given that one of ROS's main strenths is the number of community developed packages available, and the vast majority of them accepts this overhead to simplify development, the extra copies effectively become a negative feature of the ROS framework.

Chapter 3

SM2 Architecture

The library developed to fulfill the thesis goal is called SM2. It is a IPC library written in C++.

3.1 Target Scenario

The fields of robotics is too vast for one solution to be the best fit to all scenarios. For this reason SM2 is designed with the intent of working well in a limited range of operating conditions, and its design is heavily influenced by the specific necessities of such scenarios:

- A robot has several software modules (nodes) running concurrently, each specializing in one or more specific tasks. Nodes have data and possibly timing dependencies on each other.
- A robot has several sensors generating data, the data from these sensors is required by nodes for processing.
- Multiple robots may be working together in an environment connected by some kind of network.
- The rate of data produced by each robot far exceeds the capability of each robot to transmit over the network.
- There can be real-time dependencies between nodes on the same robot, meaning that no external factor should impact the speed or reliability of the local communication.
- Given the limited computing capacity, the overhead of message exchange between nodes should be minimized.

To complete the scenario description it is worth listing end user (developer) requirements for the API that the SM2 library was built upon:

- Knowledge of the implementation details of the SM2 library should not be needed to use it optimally.
- Library behavior should be as deterministic and predictable as possible.
- Expert users may want to alter library behavior.

The SM2 library itself does not provide network communication. It is designed with the primary objective of ensuring reliable, deterministic, fast and efficient local IPC by exposing high-level communication interfaces such as the aforementioned Publishers, Subscribers, Service Clients and Providers. Network communication is provided by an ad-hoc application. The SM2 library makes sure to provide the necessary tools and was specifically designed with the efficient interaction with such an application in mind.

3.2 High-Level Communication Primitives

SM2 adheres to the communication model brought forward by ROS. As such it currently allows to communicate sequential data on a Topic via Publisher and Subscriber interfaces and to perform Remote Procedure Call (RPC) on a Service by using the Service Client and Service Provider interfaces.

SM2 does not implement a communication system similar to ROS actions, as they are fairly easy to replicate by using the already provided topic and service interfaces.

Topics and Services are identified by names and by the types that are exchanged. A Topic is therefore identified by a name and a type; while a Service is identified by a name, a request type and a response type.

3.3 Low-Level Communication Primitives

To perform data exchange and behavior coordination SM2 relies on system IPC primitives.

The choice of IPC primitives for the SM2 library is discussed in Subsection 4.2.1.

However for the purpose of understanding the library architecture it is sufficient to know that SM2 builds a abstraction on top of these, that essentially consists of:

- Socket. Control messages are sent via Socket. In SM2 a Socket is an abstraction largely based on the Unix Domain Socket.
- SharedMemory. Data is shared via SharedMemory. In SM2 SharedMemory is an abstraction largely based on POSIX shared memory.

Through a Socket message it is possible to send a fixed-structure message header. Additionally a Socket message can carry another Socket reference and/or a SharedMemory reference, allowing to share said resources with the process on the other side of the Socket.

3.4 Proposed Local Architecture

Focusing on the communication on a single device, which is the core purpose of the SM2 library, the proposed structure is one where node registration and communication setup is centralized, while data exchange and feedback is transmitted in a peer-to-peer way (Figure 3.1).

3.4.1 Manager

The centralized operations are conducted by a manager node. For any node (called Client) to set up communication with other clients, it must first register itself with the manager. The client is then allowed to request a number of operations to the manager, including but not limited to:

- Creating a Publisher (Figure 3.2)
- Creating a Subscriber

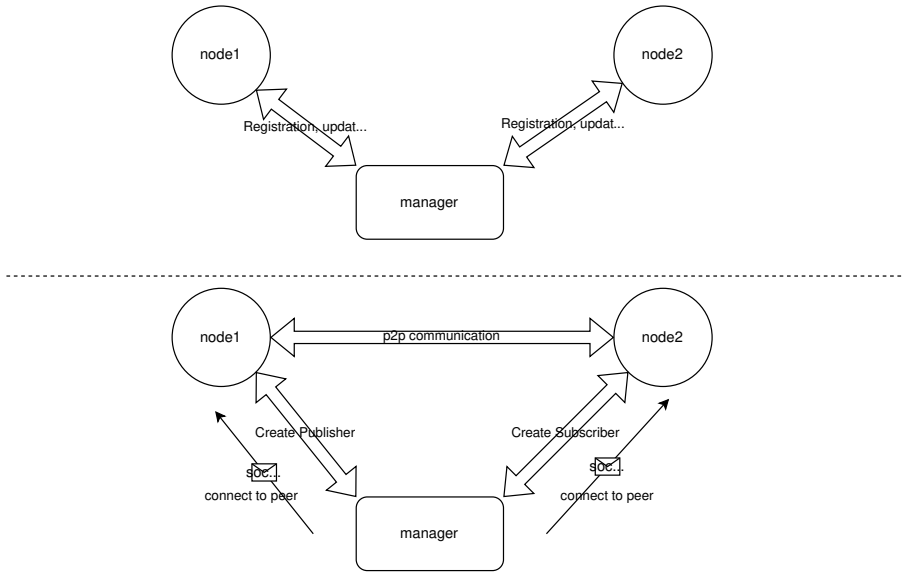

Figure 3.1. Simple example of SM2 architecture.

- Creating a Service Client
- Creating a Service Provider

Together, the manager, all the clients connected to it and their interfaces form what is called the local domain.

The manager is authoritative when it comes to the current state of the local domain, and all its decisions are based on it. In general, the manager holds the current coherent state of the local domain, and informs registered clients on any update that might concern them, such as the creation of a new publisher on a topic they are subscribed to.

3.4.2 Client

The Client is the interface used by the developer to interact with the manager from a process.

Before any other operation the Client should register itself with the manager. Each Client is identified by a name that is unique within the local domain.

When registration is complete the Client can create Communication interfaces. This operation consist of allocating the necessary resources on the Client side, and making the proper request to the manager. The manager will then process the request. If there are no errors, the manager will update its internal local domain state, and send updates to all interested Clients.

Clients can receive a message from the manager telling them to connect to another Client on one of their interfaces. This message will carry a socket, one of two in a socket pair created by the manager and sent to the Clients that need direct communication. From then on, the client can use the received socket to perform communication in a peer-to-peer way using the protocol that the type of interface requires (Figure 3.3).

This means that two Clients can never autonomously initiate direct communication, just the manager can set it up.

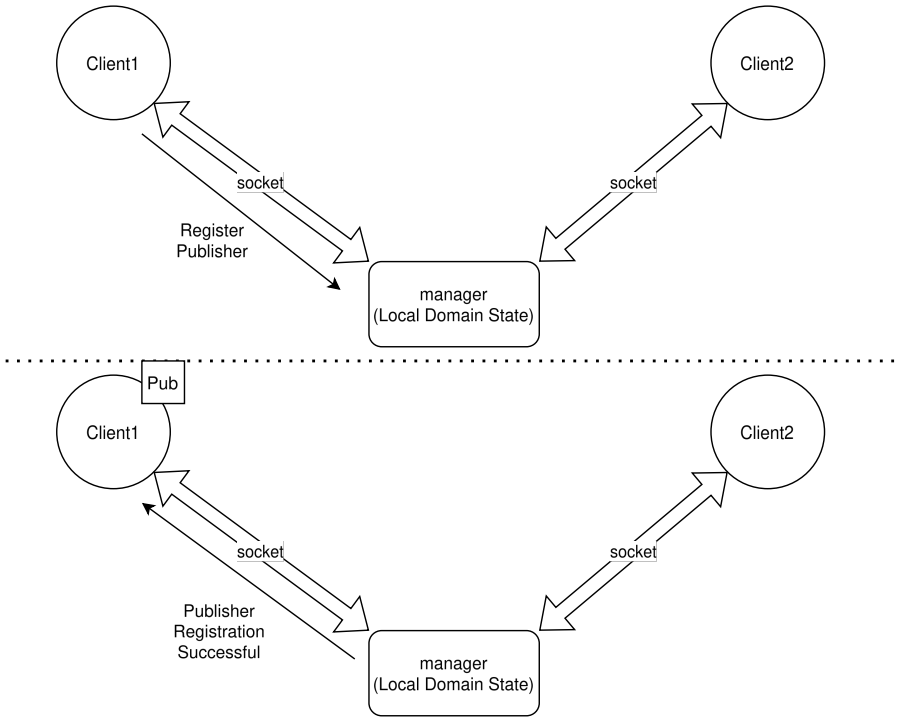

Figure 3.2. Publisher Registration example (same is valid for all interfaces).

3.4.3 Topic

Once two Client interfaces, a Publisher and a Subscriber, are connected via Socket, they can begin exchanging messages.

The Publisher-Subscriber interaction consists of:

- The Subscriber informing the Publisher it is ready to receive new messages, if possible with information about what the last received message was.
- If or once a message is published on the Publisher side, sharing the SharedMemory containing the serialized message with the Subscriber.

On Publishers, it is possible to set a size for the Publisher queue length. This essentially allows the publisher to keep a number of messages queued, so that if the subscriber is not fast enough to consume them right away they can be transmitted later.

Subscribers are created with a callback that is invoked once a message is received.

While message publication is a synchronous action, triggered by the application, Subscriber callbacks are invoked asynchronously. To simplify the synchronization and coordination of many Subscribers SM2 provides the Execution Thread abstraction. Each Subscriber is created with an Execution Thread id, the library manages creation and destruction of these threads. All Subscriber callbacks assigned to the same Execution Thread will not be executed concurrently with one another. As such it is simple to configure groups of callbacks that can or cannot run concurrently.

On the manager a topic is created once a Client first creates a publisher or subscriber with an unregistered topic name. The topic will keep existing until the last interface to it has been deleted.

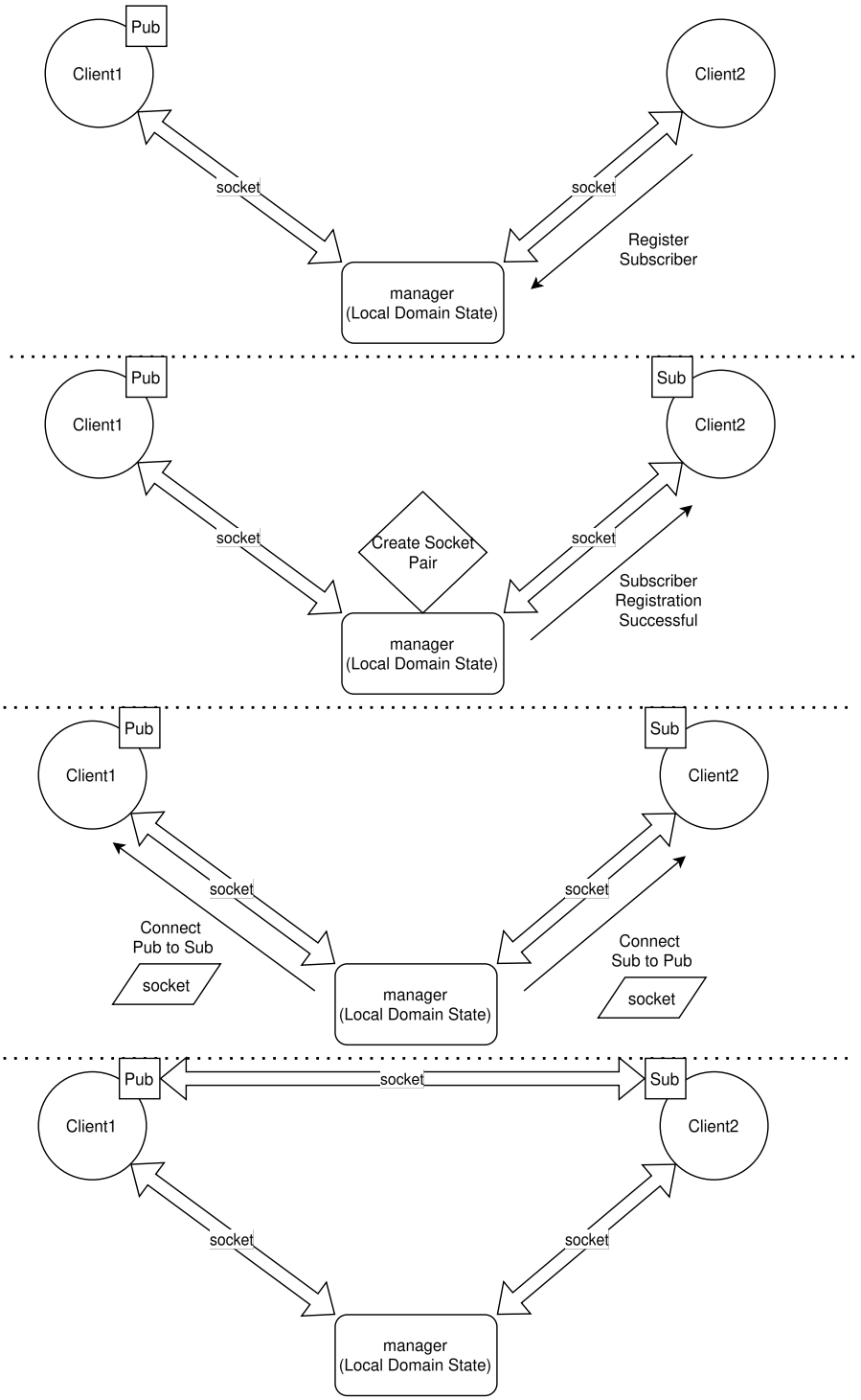

Figure 3.3. Subscriber Registration and P2P communication setup between Publisher and Subscriber (the same applies to Service interfaces).

3.4.4 Service

Once a Service Client interface and a Service Provider interface are connected by the manager, they can begin exchanging messages.

A Service Client and a Service Provider behave as follows:

- A Service Client performs a Service Request, triggered by the application. This Request carries a SharedMemory reference, where the serialized request data is stored.
- When it receives a request, the Service Provider invokes the callback it was created with, which creates the response data. A Service Response message is then sent to the Service Client carrying the serialized service response data.

Service Providers can be configured to either queue their callbacks on an Execution Thread like Subscribers or to have all requests be executed asynchronously from one another.

On the manager a service is created once a Client first creates a Service Client or a Service Provider with an unregistered service name. The service will keep existing until the last interface to it has been deleted.

3.4.5 Discussion on the Centralized Approach

Having a centralized system for registration and communication setup has both advantages and disadvantages.

The obvious critique to a design like this is that the manager is a single point of failure, and in case of many connected Clients, it could become a bottleneck in the local domain. There are however, multiple considerations that mitigate these potential problems:

- The operations that the manager needs to conduct are fairly simple both computationally and logically, with appropriate data structures ensuring they can be completed in constant or logarithmic time.
- The manager follows a strict policy to validate messages from Clients. A deviation from the protocol causes immediate disconnection of the Client.
- Manager operations are restricted to changes in the local domain topology (e.g. creation/deletion of interfaces), meaning the manager is in no way involved in the data path for messages exchanged by the Clients.
- Even on manager death, the already established peer-to-peer connections between Clients will keep operating, meaning that a previously set-up connection can continue to transmit data regardless of manager status.

With that said, having a centralized management of local domain topology brings a number of advantages to the table:

- With all local domain state information in one place, it is easy at all times to determine the exact state of the local domain, if necessary to query it.
- Having a central manager taking responsibility of informing other Clients of local domain changes significantly simplifies the protocol.
- No unnecessary communication needs to take place between nodes, since they are only connected through the interfaces by the manager.
- No discovery system is necessary since nodes do not need to autonomously find each other.
- Potentially, multiple local domains can be created on the same machine, simply by deciding what manager clients talk to.

3.5 Local Data Path

The basic practical end goal of the SM2 library is to transfer a in-language object from one process to another.

It is a precise goal of the library to not force the user to treat the object intended for transfer differently from any other object. This means that in no case should it be possible to directly share the memory holding the object (process memory) with another process.

Furthermore, for the reasons enumerated in Subsection 2.3.1, the data representation that is shared with another process should be the serialized form of the data, to ensure mutual intelligibility.

If it is necessary to serialize the data, it is also necessary to deserialize it. With these conditions in place, in a one-to-one communication scenario then, the number of copy operations (copy here and from now on can refer to both in-language copies and serialization/deserialization operations) cannot be lower than two.

In a one-to-n communication scenario the number of copies cannot be lower than $n + 1$. One copy for serialization and n copies for the respective deserialization operations of each receiver (Figure 3.4).

The library achieves this by serializing the data on a SharedMemory buffer, which is then shared to all interested processes through the relevant interfaces. The SharedMemory buffer is written to before being shared, and must not be modified after sharing. To prevent memory from being accidentally written to by the sender or a receiver process, it is sealed. For details about how this is implemented refer to Subsection 4.2.3.

Since the sender node only has to copy the data to be transferred once, the additional overhead due to a greater-than-one number of receivers is minimal. This is important as it is very common for multiple nodes to receive data from the same source.

3.6 Concurrent Execution

As mentioned previously (Subsections 3.4.3, 3.4.4) the reception of messages is an asynchronous event. SM2, like many other libraries, allows the user to determine the behavior of the application by providing callbacks that are to be executed asynchronously from the rest of the application.

In addition, the majority of library operation must happen in the background, without the user explicitly running them in a application thread. The operations of different library objects also need to happen concurrently.

For this reason, Clients, Publishers, Subscribers, Service Clients and Providers all operate concurrently on their own control threads. Naturally most of the time these threads will be idle waiting for Socket messages, but running the logic of each interface on a separate thread allows for better modularization and more effective distribution of host resources.

3.6.1 Execution Threads

While it is desirable for internal interface logic to run concurrently, user callbacks may have some synchronization requirements. Obviously the burden of performing the necessary synchronization operations could be left to the user to allow for maximum flexibility. However, SM2 opts for a middle ground solution, that allows the user to define groups of callbacks that cannot execute concurrently with one another.

This is made possible by the Execution Thread abstraction (Figure 3.5). Execution Threads are allocated and held by the Client as requested by the user. Each Execution Thread is identified by a positive integer. Whenever the user is creating an interface that requires an asynchronous callback, it is possible to specify an id of the Execution Thread that should run the callback. The

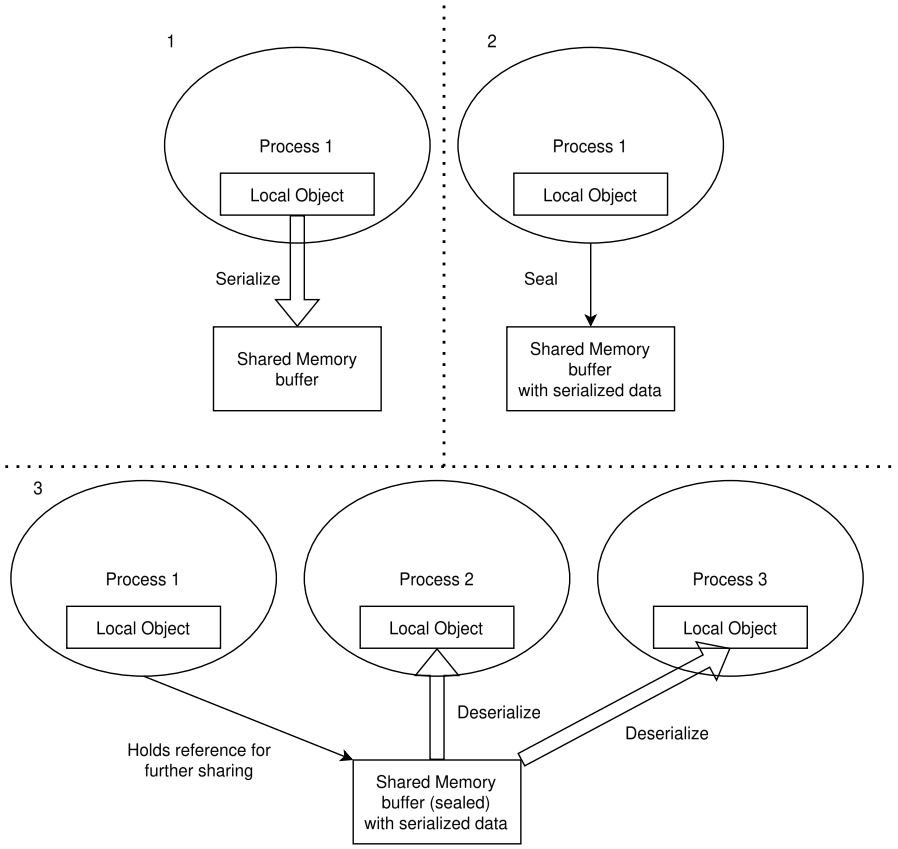

Figure 3.4. How data is transferred from a sender process to a number of receivers.

Client then ensures that the necessary threads are created and interfaces can queue a task on them.

Simply put, callbacks assigned to the same Execution Thread id will not run parallel with one another. However since they need to be executed serially, their execution may face additional delay. It is up to the user to decide how to best distribute callbacks according to application requirements.

3.7 Network Communication

As mentioned before the SM2 library itself does not provide communication over the network. It is a local IPC library and strives to allow efficient and easy communication between processes on the same machine.

However it is a vital feature of modern robotics for multiple machines to be able to communicate with each other.

To achieve this result a tool specifically designed to bridge SM2 local domains across the network was developed together with the library.

This tool is called the SM2 Network Gateway (SNG for short).

The SNG must provides a number of features:

- It can connect to multiple peers on the network.

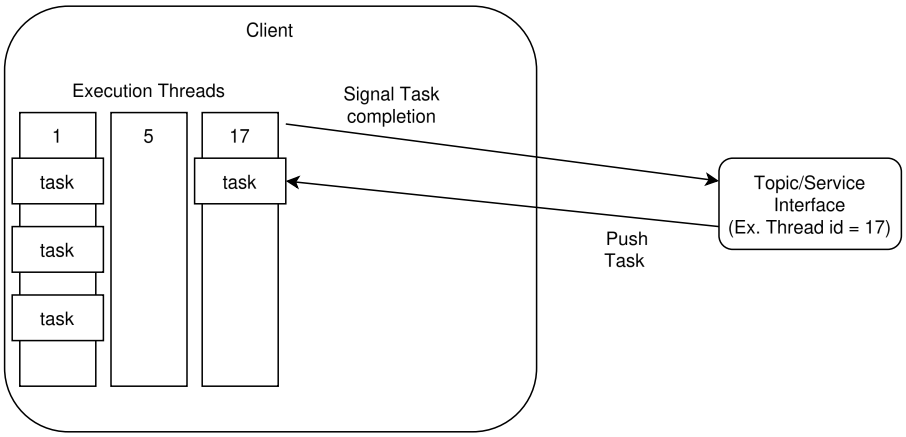

Figure 3.5. Execution Threads.

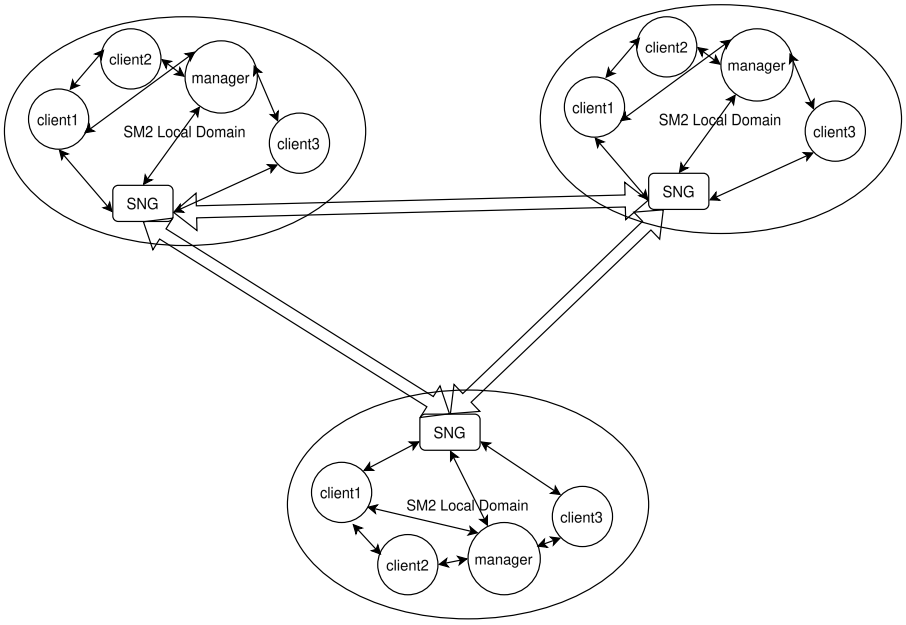

Figure 3.6. Architecture of communication between local domains.

- It can authenticate itself and its peers.
- It can automatically discover peers in the same subnet.
- It provides secure communication.

Once connected to a peer, the SNG can, depending on configuration, allow the transmission of topic messages and service requests/responses.

3.7.1 SNG Configuration

While designing the SNG, emphasis was placed on the configurability of what is allowed to be transmitted and how.

While on one hand, during development for example, it may be desirable to have simple access to all remote topics and services, this is very much undesirable in deployment. This can be true for several reasons:

- Security concerns: only a very limited number of topics or services should be made visible to limit access from outside.
- Organizational concerns: having access to all topics and services of a remote SM2 domain can clutter the local domain with unnecessary messages.
- Efficiency concerns: allowing unchecked transmission of topics and services can lead to wasteful use of often limited network resources.

Furthermore it is desirable to have control, not just on whether a topic or service is allowed to be transmitted, but also on what direction it is allowed to be transmitted in. The SNG has four possible configuration rules:

- Allow all: topic messages can go in both directions, service requests can come from either side.
- Allow incoming: topic messages can only come from the remote domain, service requests can only come from the local domain to the remote (the idea is that the remote is providing the service, therefore the service is incoming).
- Allow outgoing: specular to allow incoming.
- Deny: no communication is allowed.

These rules can be used to specify a rule-set that has a default rule and can have a number of exceptions based on topic name. The SNG allows exceptions to be based both on the exact name of the channel(topic/service) but also allows to specify characteristics of the name such as what the name starts with, or a string it contains.

Exceptions are applied in order, so if a topic or service matches multiple ones the first one is picked.

3.7.2 SNG within the Local Domain

Naturally in order to work, the SNG needs to be able to communicate with the other Clients in the local domain. This could be easily achieved by simply leveraging the SM2 library top-level abstractions, but such an approach would incur into several problems:

- To determine the topics and services to interact with the SNG would first need to know which are available. This could be fixed by forcing the user to compile a "white-list" but it goes against the configuration design.
- Within the local domain topics and services can change over time, the same topic could technically first be associated with a certain type, then with another. Although this is an unlikely occurrence in a normal scenario, it should be covered by the SNG.
- For efficiency's sake, the SNG should only open interfaces to topics and services when there is mutual interest between the local and remote domains (i.e. when actual transmission can take place based on local/remote configuration and domain state). This is impossible to do without having more information about the local domain.

3.7.3 SM2 Support Features for SNG

To allow for more efficient and automatic interaction with the local domain, the SM2 library provides a number of additional features that are heavily used by the SNG. It is important to emphasize that these features, while developed mostly to assist the SNG, can also be very useful for a number of different uses (a record/play tool similar to ROSBAG comes to mind).

There are two main support features that were added to the SM2 library:

- Ability to query the manager for current local domain state or receive updates about the local domain state.
- An extension to interface creation that allows to create "Ghost" interfaces. The main idea behind ghost interfaces is that they do not count to keep their respective channel (topic/service) alive, meaning that they allow a topic or service to be deleted or change type(s) even if they are not closed.

3.7.4 Local Domain State Querying and Updates

To allow any registered Client to know the current state of the local domain, the SM2 library allows two different approaches:

- Local Domain State Query: this is a interaction initiated by the Client, which requests the local domain state. The manager will reply with a data structure containing all the necessary information.
- Local Domain State Updates: this interaction is meant to avoid the periodic polling of the manager by a Client to keep itself updated. The Client can activate or deactivate Local Domain State Updates at any time. On activation the manager will send a data structure containing information about the current state, and will then follow up with further updates every time the local domain topology changes. This will continue until the Client that activated the behavior disconnects or or deactivates it.

The SNG make use of the Local Domain State Updates feature to dynamically manage its interfaces with the SM2 local domain.

3.7.5 Ghost Extension

The ghost extension builds upon the already existing protocol to confer a specific behavior to topic or service interfaces.

In particular an interface marked as ghost:

- Is not allowed to be created on a channel (topic/service) that does not already exist.
- Does not contribute to the interface count keeping the topic or service alive.
- Once the topic or service it was created on are deleted, all related ghost interfaces are detached from it and the Clients owning them are appropriately informed.
- Cannot communicate (the manager does not connect them) with another ghost interface.

The SNG combines the domain updates and the use of ghost interfaces to make informed decisions based on the current state of the local domain, while avoiding resource waste.

3.7.6 Centralized Approach applied to Network Communication

The choice of centralizing the communication across the network as opposed to ROS 2 is driven by several factors. Some are similar to those discussed in section 3.4.5 so they will not be repeated now.

The key reason for this choice however is that the scenario SM2 aims to tackle is one where network communication happens over an unreliable, constrained medium, meaning that the ultimate bottleneck will always be the network communication speed. In such a situation, all the critiques about the lacking scalability of a centralized approach become far less convincing.

Furthermore centralizing network communication brings the additional advantage of being able to control and monitor network resource usage from one place. It allows, for example, to easily limit the used bandwidth based on readily available (because centralized) statistics.

3.8 Security

SM2 was created for use in the field of robotics. This means that it can and will transport vital information for the proper functioning of a robot. Therefore its main concern is to protect the proper functioning of robot processes, as these processes could be safety-critical and their malfunction may pose a real danger to people and resources. The other asset to be protected is the exchanged data itself, especially from potential outside interest.

3.8.1 SM2 Library Threat Model

Since the SM2 library only directly provides local communication, its threat model does not focus on data protection outside the machine. The assets to defend in the threat model for the SM2 library are:

- Process Stability
- Message Data
- System Consistency/Availability (manager)

The SM2 library generally follows the ROS model that makes all topics and services available to everyone in the conversation. This means that it does not directly take steps to prevent a local node from having access to local domain channels. The general idea is that if a process has access to the manager Socket (that can be managed via file-system), it is treated as equal to all other processes in the SM2 local domain.

While it is unlikely that a process able to join the local domain would have malign intentions (since the user should have control over it), it cannot be excluded. Furthermore threats to the mentioned assets could come not just from intentional attempts, but also from implementation mistakes.

Given that SM2's main job it to setup and manage Inter Process Communication, the threat surfaces are the points of contact between processes:

- Shared Memory.
- Sockets.

In particular these entry points could be used to create the following threats:

- Process stability could be undermined by sending improperly formatted messages/fuzzing over Socket communication.

- Shared Memory could be tampered with or its content could be improperly serialized, leading to wrong data transferred o stability issues in case of failed deserialization.

The threat of improper messages over Socket are mitigated by SM2's strict enforcement of its protocol (section 4.5). In particular a violation of the protocol leads to immediate disconnection. This ensures that, even with fuzzing attacks, the result on the local domain is coherent and consistent with the rules enforced by the manager.

The threat to stability coming from improperly serialized data can be effectively removed by performing checks before memory accesses during deserialization. This allows to always be able to either deserialize successfully or catch the exception before it impacts process stability. When caught such an exception can be treated as a protocol violation leading to the disconnection of the offending Interface.

The threat of Shared Memory tampering is mitigated thanks to the memory sealing measure (section 4.2.3). This makes it impossible for any process to write to the Shared Memory after sharing, ensuring its validity. The task of sealing the memory is left to the process allocating it (the sender).

Finally, it is important to note that SM2 does not know nor does it take decisions based on the content of the memory shared in messages. As long as protocol is respected and deserialization is successful, the message is delivered as a proper in-language object. As for the contents of this object the application should implement necessary checks based on the context.

3.8.2 Network Communication Threats

Once SM2 messages are transmitted over the network, they are subject to all threats common to Internet communication. For this reason the SNG must provide appropriate security measures, ensuring the reliable, confidential and secure transmission of messages from itself to a peer.

The threat model of the SNG therefore does not differ particularly from that of any other application having to perform secure data exchange on public networks.

The job of protecting the network communication from attacks falls on the protocol chosen by SNG (section 5.1.1).

Once secure network communication is ensured, SNG only needs to carefully enforce its own overarching protocol, mostly for the sake of the SNG's own stability. The SM2 library already considers and mitigates threats coming from malformed messages and as such the SNG can effectively transfer them blindly, which is a boon for efficiency and flexibility.

Chapter 4

SM2 Implementation

4.1 Overall Structure

The SM2 library is designed with ease-of-use in mind, but also wants to remain relatively easily customizable. For this reason the code is split in several sections:

- The back-end: deals with Inter-Process Communication primitives, handles system errors, prevents dangerous use of resources.
- The front-end: presents an API similar to that of ROS to create communication interfaces such as Publishers, Subscribers and Service Clients and Providers.
- The type-access module: allows to identify, serialize and deserialize types.from multiple

It is therefore possible to change the behavior of the library by acting on front-end logic for example, with the other modules staying as a good foundation.

4.2 Back-End

At the core of the SM2 library is the concept of Inter-Process Communication (IPC).

In all most used operating systems, processes are isolated from one another, this prevents one process from mistakenly corrupt another's memory, and makes sure that the failure (crash) of one process does not affect the others.

Operating systems provide a number of IPC primitives that allow processes to break this isolation. The most famous and widely used are files stored in permanent memory, however the communication requirements of robotics software push for a faster and more efficient solution.

Unfortunately different systems offer very different solutions when it comes to efficient IPC. The SM2 library was originally developed for use on Linux systems and as such it draws its back-end abstractions from Linux IPC primitives.

4.2.1 IPC Primitives

A modern Linux operating system offers these IPC primitives [20]:

- Pipes: allow exchange of a FIFO stream of data between processes.
  - Anonymous pipes: can only be created between related processes.

  - Named pipes: are tied to a file-system path and can be shared between unrelated processes.

- Message Queues: allow exchange of fixed size messages with different priorities.
  - System V message queues: are identified by an integer key.
  - POSIX message queues: are identified by a name and generally mounted on the file-system.

- Shared Memory: a segment of memory that can be concurrently mapped and used by multiple processes.
  - System V shared memory: is identified by an integer key and must be explicitely released.
  - POSIX shared memory: is identified by a file descriptor, depending on configuration it can be automatically released by the system once no processes hold open file descriptors to it.

- Sockets: allow FIFO transfer of data between processes.
  - INET (TCP/UDP) Sockets: network based IPC, can be used for local communication via loopback interface.
  - Unix Domain Sockets: Local sockets that can be identified by a file path or be abstract(only referenced by file descriptors). Additionally they allow to transfer other file descriptors within socket messages. [21]

This list purposely leaves out the slower IPC primitives such as permanent memory files, since they do not match the required speed criteria.

The Linux implementation of SM2 utilizes Abstract Unix Domain Sockets for the exchange of control messages. As stated above, they are able to transfer file descriptors within a message, meaning it is possible to exchange any system resource that is represented by a file descriptor.

The reason to use Abstract sockets instead of named ones, is to avoid cluttering the filesystem with too many named resources and to prevent processes uninvolved with the SM2 resources to accidentally gain access over them.

The only named socket in SM2 is the manager socket, which needs to be reached by clients without previous knowledge. All other sockets are created as a abstract socket pair within the manager and shared to the Clients via manager socket.

To transfer variable sized data between processes SM2 on Linux uses POSIX Shared Memory. In particular it uses said IPC primitive by creating it with the memfd_create Linux system call [22], which allows to create anonymous POSIX Shared Memory. This again with the intention of not cluttering the filesystem and preventing erroneous access to SM2 resources.

Naturally with SharedMemory being a system resource that is identified by a file descriptor, ownership can be shared through Socket.

4.2.2 Sockets

Sockets are the foundation of SM2 communication. The Socket abstraction, exposed by the backend, while heavily influenced by the Linux Unix Domain Socket in its design, purposely restricts what the user can do with it, to ensure safe and reliable message transfer.

The fundamental unit at the base of all SM2 communication is the SM2 message (Listing 4.1). The message is composed of a 128bit header, and an optional payload of either or both a shared memory buffer and a socket.

In the Linux implementation of SM2 this is achieved by leveraging the control messages that can be sent alongside data on Unix Domain Sockets. Thanks to them almost any system resource handled by a file descriptor can be sent to another process. However it is neither the aim nor the desire of the SM2 library to allow such a flexible use of the Socket abstraction.

Because of this the exposed abstraction performs all necessary checks and ensures that the received messages follow the expected format. It deals with platform-specific problems, to make sure that the user of the back-end only has to deal with a opaque, type-safe, RAII wrapper of the underlying resources.

Listing 4.1. SM2 message

```
class SharedMemory;
class Socket;

struct SM2MsgHdr
{
        std::uint32_t type;
        std::uint32_t ext;
        std::uint64_t u1;
};

struct SM2Msg
{
        SM2MsgHdr hdr;

        std::shared_ptr<SharedMemory> mem;
        std::unique_ptr<Socket> sock;
};
```

4.2.3 Memory Sealing

The SharedMemory that is used as buffer for a serialized message is first allocated and held only by one process. This process is going to write to the memory and after that it will be shared among multiple processes to transfer the data. Once the memory has been shared, it can, and one has to assume that it will, be used concurrently by any subset of the processes it was shared with.

It is therefore a vital requirement when handling SharedMemory, to prevent any process from writing to the SharedMemory buffer once it has been shared.

How to do this heavily depends on the operating system.

In its Linux implementation, the SM2 library relies on the file sealing feature introduced alongside memfd_create [22]. This feature allows to prevent certain actions from being taken on the memory. In particular the memory is prevented from being resized, and from having write access mappings.

Interestingly some of the earlier Linux kernel versions that support memfd_create (such as 5.15) pose some stringent conditions on the sealing operations. In particular when using the F_SEAL_WRITE seal, no write access mappings can later be created from a file descriptor open on the resource with write permissions. For this reason to perform the seal it is actually necessary to open a new file descriptor to the same SharedMemory resource with read-only access, and close the other.

More modern Linux kernels (such as 6.11) have relaxed restrictions, allowing read-only mappings to be created from write-access file descriptors after sealing [23].

4.2.4 Idle Time Optimization

As mentioned in Subsection 3.6 many library operations need to occur concurrently, however most of these concurrent threads will be idle at any give time, waiting for resources. These resources are, the grand majority of times, Sockets.

Ideally, threads would only ever wake up when a message is received, and stay idle at other times. In addition, it is quite common for a single thread to necessitate waiting on multiple Sockets at the same time. Finally it may also be necessary to wake up the thread from inside the application, without necessarily sending a Socket message.

Linux offers the poll and select system calls to perform waiting on several resources at the same time. SM2 uses the poll system call in its Linux implementation, and exposes a Poll abstraction from the back-end.

To address the necessity of signaling the thread to wake up from the same application, without having to resort to inelegant workarounds such as sending socket messages among threads of the same process, the SM2 back-end exposes a Event abstraction that can be waited upon with Poll and can be signaled and cleared. Event is based on Linux's eventfd [24] but also closely matches similar primitives of other operating systems such as Windows Events.

The combination of the Socket, Poll and Event abstractions allow to efficiently wake up threads only at times of necessity, without having to resort to periodic polling.

4.2.5 Error Handling

The back-end has to deal with a lot of possible error situations. Some of these can be solved on the spot but most are beyond the scope of what the back-end is designed to handle.

Errors can stem very different causes such as:

- System resources unavailable.
- Process resource limits reached.
- Protocol violations in message transmission.
- Improper use of abstractions.

Since many of these errors have to be dealt with immediately, the back-end embraces the C++ error model, by forcing the modules using the back-end to immediately acknowledge and take care of possible errors.

This design choice is common to the other modules as well.

4.2.6 Platform Support

While the front-end of the SM2 library is completely platform-independent, the back-end is inextricably linked to the operating system it runs onto.

SM2 was developed primarily for Linux machines, and as such it draws most of its abstractions from Linux System primitives, and its Linux implementation is by far the most tested.

Despite this, a non-negligible amount of effort was put into developing a working back-end for Windows to demonstrate capability of the overarching modules to work with different operating systems.

To complete the Windows implementation, a number of workarounds had to be conceived to adapt the different Windows System objects to the SM2 abstractions:

- SharedMemory is implemented using anonymous file mapped shared memory.

- Sockets are a custom made data structure that leverages a number of system features such as Events, process handles and shared memory to obtain a comparable behavior to unix domain sockets.

- Events are easily implemented with Windows events.

- The Poll abstraction was implemented by leveraging the WaitForMultipleObjects system call, but would require some adjustment for a production build.

The Windows implementation was built for demonstrative purposes and in its current state it is not well tested enough to be considered reliable.

4.2.7 System Resource Release

Thanks to the chosen IPC primitives and their characteristics the SM2 library makes sure that processes always release the resources, even in cases of unplanned exit (crash).

For example within the Linux implementation, both the Unix Domain Socket and memfd created memory are handled via file descriptors, meaning their reference count will be decreased by the operating system even if the process closes unexpectedly, and will be released once the reference count reaches zero.

Even in the unfortunate case of a process dying during the transfer of the shared memory handle to another process via socket, the system takes care of making sure that it is delivered to the other process, as long as the sendmsg system call is completed successfully.

This kind of clean, predictable behavior would be impossible to achieve with other IPC primitives such as System V shared memory.

4.3 Type Access Module

One of the key features of the SM2 library is the ability to transfer C++ native types and data structures recursively based on them directly. It is a precise design choice to have no IDL like ROS. This decision is based on the drawbacks of the ROS approach described in Subsection 2.3.2.

This means that the library has to provide a way to identify, serialize and deserialize types directly defined by the user.

4.3.1 Challenges posed by C++

The C++ language was not designed to support runtime type reflection. While the language may eventually evolve to support some kind of type reflection that will make it easier to implement generic serialization, current realistic proposals are being considered for the C++26 standard [25]. This means that a long time is still needed until such features are stably available on most compilers (Some of the most popular compilers still have not fully implemented all features of C++20 [26]).

C++ over time has come to support RTTI (Run-Time Type Information) which allows to get a runtime type name and hash. This would solve at least the matter of identification, if not for the fact that there are no guarantees these will be consistent across different program executions, let alone separate programs altogether [27].

Not having access to type reflection means that there is no simple way to query a type for the number and nature of its fields. Nor any type-safe way to iterate over them in a generic way.

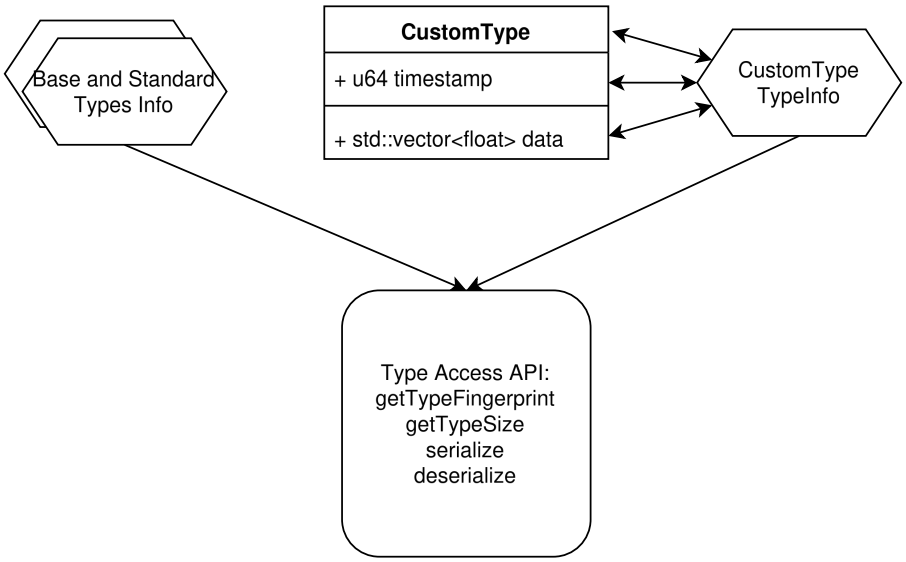

Figure 4.1. General structure of the Type Access Module with an example type.

4.3.2 Type Description

The way the SM2 library tackles the type access problem, is essentially by requiring the user to provide a description of the type he intends to make transportable by the library. This description must let the library know the number, name and type of each member of the custom type. It goes without saying that it would go against one of the main goals of SM2's design to force the user to do this operation by hand, which is why the process is mostly automatic, requiring minimal effort by the user (see Subsection 4.3.4).

The type information is "stored" by the TypeInfo template class. For each type that is made eligible for transport via SM2 a template specialization of TypeInfo must be defined. The reason why this uses a template struct and not directly a template function has to do with the C++ standard not allowing partial template specializations for functions but only for classes/structs [28].

```
template<typename T>
struct TypeInfo
{
 static constexpr auto get()
 {
  static_assert(!std::is_same_v<T, T>, "Type T has not been made inspectable");

  return std::make_tuple();
 }
};

template<typename T>
constexpr auto getTypeInfo()
{
 return TypeInfo<T>::get();
}
```

The code above shows the base template for the TypeInfo struct. It makes sure that if in any part of the code there is an attempt to access SM2 type information that was not provided by specializing the TypeInfo struct, a meaningful compilation error will be thrown.

A user can then define a specialization either for the TypeInfo struct or for the of the getTypeInfo function that would look something like this:

```
struct Data
{
 std::uint64_t timestamp;
 std::uint64_t id;
};

template<>
constexpr auto sm2::type_access::getTypeInfo<Data>()
{
 return std::make_tuple(
  sm2::type_access::TypeInfoHeader
   <std::extent_v<std::remove_reference_t<decltype("Data")>, 0>>
   {"Data", std::is_trivially_copyable_v<Data>},
  sm2::type_access::makeTypeInfoMemberAccess(
   "timestamp",
   [](Data& type_name_instance_ref)->decltype(auto)
   {
    return std::ref(type_name_instance_ref.timestamp);
   },
   [](const Data& type_name_instance_ref)->decltype(auto)
   {
    return std::cref(type_name_instance_ref.timestamp);
   }
  ),
  sm2::type_access::makeTypeInfoMemberAccess(
   "id",
   [](Data& type_name_instance_ref)->decltype(auto)
   {
    return std::ref(type_name_instance_ref.id);
   },
   [](const Data& type_name_instance_ref)->decltype(auto)
   {
    return std::cref(type_name_instance_ref.id);
   }
  )
}
```

Essentially, the getTypeInfo function returns a tuple that contains a list of other tuples, each providing necessary details about the type. The first one contains the name and some general characteristics of the type. The following tuples respectively hold the information about the fields of the type. In particular each provides the name of the field and two accessor lambda functions that allow type-safe access to the field when the type is either const-qualified or not.

This tuple structure holds enough information to allow writing a automatic identification and serialization system, provided that all types of the fields also have a working getTypeInfo specialization.

4.3.3 Base and Standard Types

Since the general requirement to make a type transportable by SM2 is that all field types must also be transportable, SM2 needs to provide the basic building blocks that the user needs to use when defining data structures.

SM2 therefore natively provides type info definitions for all base types, and many of the stdint types. It also provides support for some of the standard library types such as std::vector, std::string and std::array. SM2 support of standard library types could be extended in the future, but what it currently support is sufficient to satisfy most IPC requirements.

In C/C++ base types can also match to different kinds of data depending on compiler. SM2 by defining the type information for each, can define different ones based on compiler or system, ensuring that programs running on different data models [29] can understand each other.

4.3.4 Type Description Generation

As shown by Subsection 4.3.2 writing a type description is complicated and definitely not something that the SM2 library wants to force the user to do. At the same time, SM2 does not wish to need any tool external to normal compilation aiding this process.

For these reasons SM2 defines macros that allow the user to generate a type description by only providing the essential details.

As an example the type description shown in Subsection 4.3.2 can be generated with this one line:

```
SM2_CORE_MACRO_MAKE_TYPE_INSPECTABLE(Data, timestamp, id);
```

This is achieved by leveraging a fairly complicated set of macros. In particular the biggest obstacle is the fact that C/C++ macros cannot be called recursively, meaning that it is almost impossible to scan through a list of items (like the list of field names) via macro. To get around this SM2 makes use of a trick that imitates macro recursion up to a pre-determined depth [30].

4.3.5 Type Identification

To determine the identity of a type, SM2 makes use of a name, generated by recursively making use of the type info. For the Data data structure shown before as an example the identification name would be similar to this:

```
Data{u64, u64}
```

For more complex data structures, the name can become quite long, since it recursively specifies the details of each data structure contained in the main one.

4.3.6 Compile-Time Name Generation

Since all details of SM2 transportable types are known at compile time, thanks to the definition of type info, the type name generated by SM2 could in theory be pieced together at compile time, to achieve better performance.

This is possible because all information coming from the getTypeInfo function, is constexpr [31].

There is however one obstacle in the fact that C++ does not directly provide tools to work at compile-time with dynamically allocated types such as std::string, making the job of piecing together text before runtime quite challenging.

SM2 uses a custom made CompileTimeString type, that is effectively a fixed size string, with a number of useful adjustments to work with it as a constexpr. This custom data structure allows

for compile time concatenation, which is effectively the only kind of manipulation necessary to build the identifier name for a SM2 transportable type.

The generated names are therefore calculated by the compiler itself, and stored directly in the executable, achieving an almost zero-overhead type identification system, that works across separate programs, compilers, operating systems.

4.3.7 Serialization

With type information available and separate processes being able to make sure they are talking about the same type with type identification, it is possible to serialize data for IPC in a way that is mutually understandable.

Achieving this effectively consist of recursively serializing the member fields of a data structure one after the other by either following a general model, or special behaviors (defined for example for dynamically allocated standard library data structure such as std::vector).

Serializing to a memory buffer also requires knowing how large the memory to serialize to needs to be. This requires traversing the data structure similarly to how one would in order to serialize it, but without actually copying the data.

Deserialization is then trivially implemented, by reversing the serialization process.

4.4 Front-End

The front-end for the SM2 library is perhaps the least interesting module in terms of challenges and implementation details. It is a more or less trivial formalization of what is described in section 3.4.

Since the front-end is what determines the API that average users will interact with the vast majority of times, it attempted to:

- Be as clean and expressive as possible. Each interaction with the API should need no unnecessary details, and the call syntax should be fairly self-explanatory.
- Allow for enough flexibility. Users should have some control over all the important parameters of the underlying logic. The library should try to limit what the user can do with API objects as little as possible.
- Not force any code structure (such as the ROS 2 node structure) on the user. In general leave the user as free as possible when it comes to organizing the code.

4.4.1 API

The SM2 front-end exposes five classes to the user to access the features provided by the library:

- `class Client`

  The Client class represents a client node in the SM2 local domain (see 3.4.2). A Client object provides the necessary functions to interact with the manager to create interfaces.

- `template<typename T> class Publisher`

  The Publisher class is the interface that allows to send messages on a topic. The template type is the type of message sent. It can be configured with a custom queue size.

- `template<typename T> class Subscriber`

The Subscriber class is the interface that allows to receive messages sent on a topic. The template type is the type of message received. It is created with a callback to be called on message reception, associated with an execution thread.

- `template<typename ReqType, typename ResType> class ServiceClient`

  The ServiceClient class is the interface that allows to call a Service. The template types determine the request and response types.

- `template<typename ReqType, typename ResType> class ServiceProvider`

  The ServiceProvider class is the interface that allows to advertise and provide a Service. The template types determine the request and response types. It is created with a callback that is the procedure in the RPC paradigm. It can be configured to execute calls serially on a execution thread, or to execute each asynchronously.

4.4.2 RAII and Object Lifetime

SM2 objects are RAII (Resource Allocation Is Initialization) wrappers around several resources that manage the logic underneath the front-end abstraction. The intent of this design is to make it easy, if not encouraged, for the user to create specific interfaces with a limited scope, such as a Subscriber that exists only within a certain function.

Since the SM2 objects manage a number of resources, many of which are tied to asynchronous tasks, destroying a SM2 object requires synchronizing and waiting a number of events, such as resource release.

To allow for maximum flexibility SM2 explicitly allows destroying an interface such as a Subscriber inside the callback itself, to achieve behaviors such as a waitForMsg function.

The only restriction is to not destroy the Client object inside an interface callback, since the client owns the threads executing all callbacks and it would lead to a deadlock.

In general the library poses no limits to how many API objects can be created by one process, so it is possible to have several clients within the same process, and several interfaces to the same channel (topic/service) within one Client.

In practice this could be limited by some operating system constraints such as Linux's limit on the number of open file descriptors a process can hold. While this should never be a problem for normal use, it is advisable to look into it for production. Many of these limits can be manually raised without too much effort, but the SM2 library does not automatically take care of it.

One last thing to keep in mind is that each interface (publishers, subscribers etc...) keeps a counted reference to the Client they were created with. This means that the user should be aware that if said reference is the last one to the Client, destroying the interface inside a callback can lead to the aforementioned deadlock. It is therefore good practice to keep a explicit reference to the Client until all interfaces are destroyed.

4.4.3 Ease of Use

Since the purpose of the library is to facilitate prototyping, it is vital for it to be easy to use.

One key feature that simplifies the work of the user and prevents unnecessary headaches is that all library calls are thread-safe, there is no need to manually synchronize any of the library objects.

Furthermore, all library objects are managed via std::shared_ptr, adhering to a well known C++ pattern and allowing the safe managing of object lifetimes while avoiding unnecessary (and unwanted) copies.

This is an example use of the library, a Client with a publisher that publishes a increasing integer every second.

```
#include <sm2/sm2.hpp>

int main()
{
 auto client = sm2::Client::create("client_name");

 auto pub = client->createPublisher<uint32_t>("topic_name");

 unsigned int i = 0;
 while(true)
 {
  pub->publish(i++);
  sleep(1);
 }
}
```

4.4.4 Publisher and Subscriber Implementation

The communication between publisher and subscriber follows relatively simple rules:

- A subscriber informs the publisher about its readiness to receive a message.
- A publisher sends the message only when requested by the subscriber.

The SM2 protocol is actually fairly permissive when it comes to Publisher/Subscriber interactions. The way it works is that every published message (also called frame) has a associated increasing 64bit id. When a subscriber informs a publisher that it is ready to receive a new message, it also sends the id of the last frame it received. This behavior could potentially be easily changed to limit the amount of messages a subscriber receives. For example by not sending the last frame id but the last frame id + n, it would effectively limit the amount of frames received by the Subscriber to 1 every n. This is not currently a configurable option via the front-end, but it is easily implemented.

As mentioned in section 3.4.3 the publisher can be configured to have a maximum message queue size. The purpose of a message queue is to keep a certain number of published messages saved, so that if a Subscriber cannot temporarily keep up with the publication frequency it still would not lose messages. The implemented behavior upon being informed that a connected Subscriber is waiting for a message is to send the earliest message in the queue with a frame id greater than the Subscriber's last frame id.

Figure 4.2 shows an example of Publisher/Subscriber interaction.

It is important to note that topic communication cannot guarantee that all sent messages are delivered. Since the Publisher drops messages from the queue when it is at maximum size, if the Subscriber has not received them in time, messages could be dropped. This can be mitigated and in some cases effectively avoided by selecting an appropriate queue size.

4.4.5 Service Client and Provider Interaction

The interaction between Service Client and Provider is very straightforward:

- When a Service request is created from the front-end, it is added to a list of the pending requests for that Service Client.
- If a Service Provider is available the request is sent to that service provider.

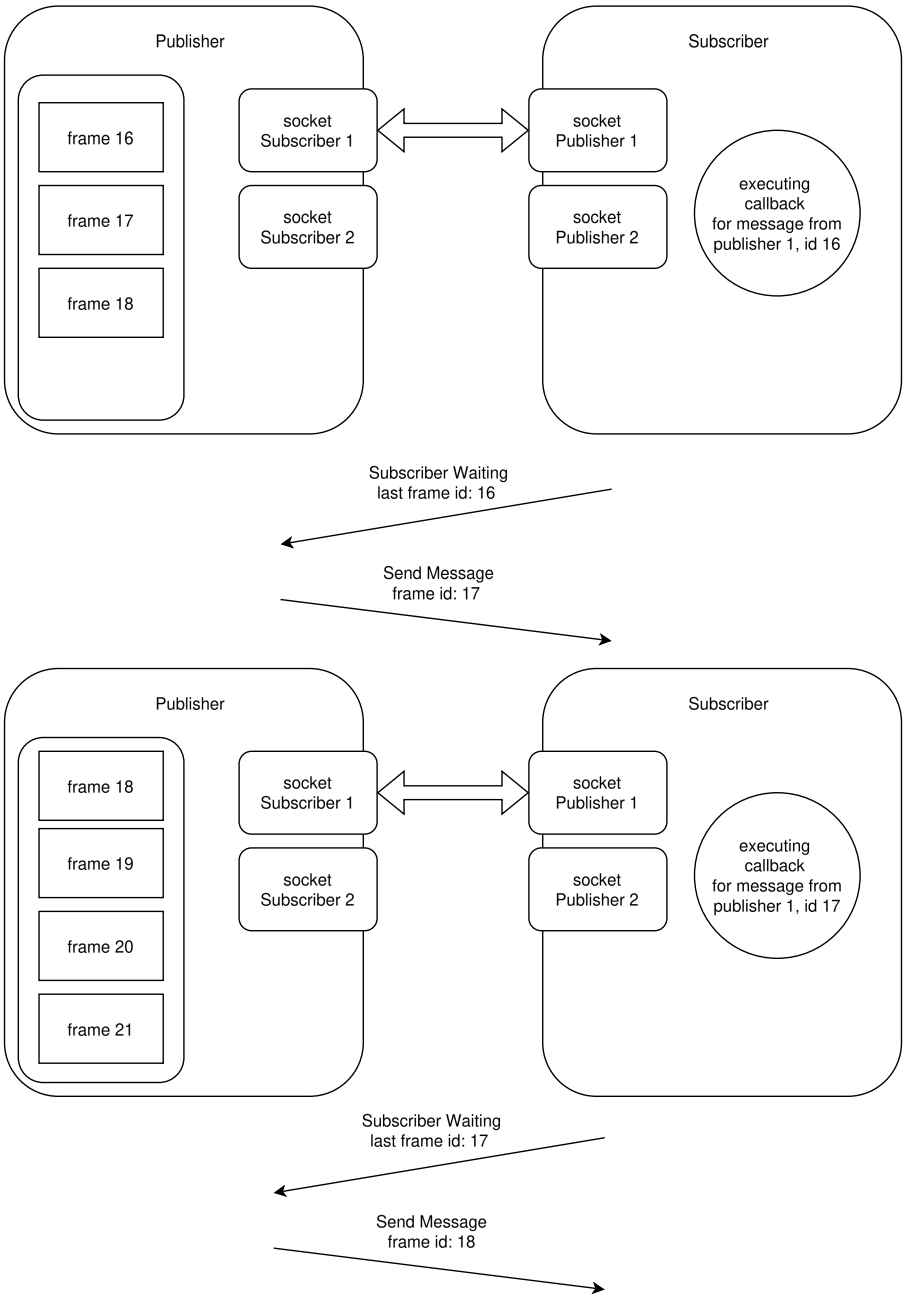

Figure 4.2. Example interaction between a Publisher and a Subscriber. In this example, 3 messages are published on the Publisher side in between subscriber requests.

- When a new Service Provider is connected, all pending requests are sent.
- When a Service Provider receives a request, it queues the execution of the procedure based on the scheduling configuration it was created with.
- When the procedure associated with a request has been completed, the reply is sent to the

Service Client.

- Upon reception of the Service reply, the Client removes the request from the pending list and notifies the user of Service completion.

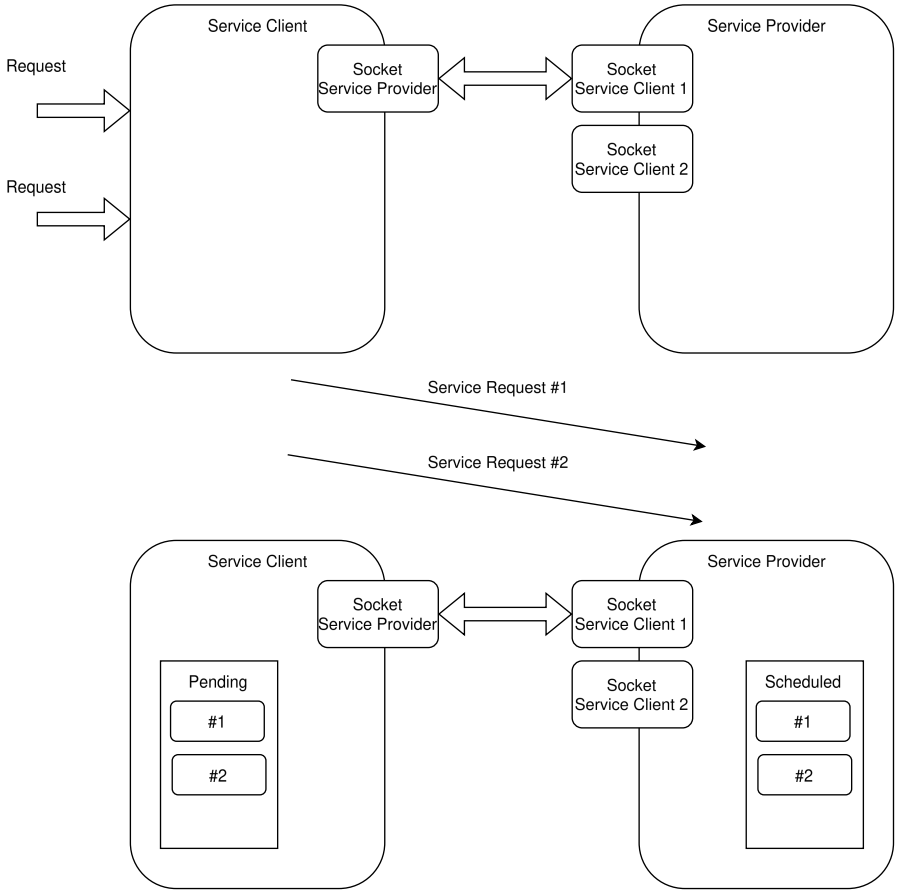

Figure 4.3. Example interaction between a Service Client and a Service Provider. In this example, 2 requests are made in quick succession. The requests are then both sent to the provider.

Communication over Service in SM2 is inherently reliable, meaning that unless the Service Client is destroyed ahead of time, all requests are delivered, and eventually fulfilled (not necessarily in order).

4.5 SM2 Protocol

Table 4.1 shows a list of all messages of the sm2 protocol and their function.

The general rule followed by the implementations of the manager, Client and SNG is that if a connected Client fails to adhere to the protocol it is immediately disconnected.

The SM2 message shown in listing 4.1 is the unit exchanged in all SM2 communications, the fields generally have these meanings:

- **type**: is the type of the message, clarifies the intent of the message and determines how all other message fields are interpreted.

- **ext**: is generally used to extend or specify the meaning of the message.
- **u1**: as the only 64 bit field, it is used to transfer long ids.
- **sock**: is the socket that the message can transport, it is used to provide the receiver with a connection to someone else.
- **mem**: is the shared memory handle that the message can transport, it is used to transfer data of various kind depending on message type.

Message Type	**Sender**	**Receiver**	**Fields / Notes**
ERR	Any	Any	Error condition.
REGISTER_CLIENT	Client	Manager	`ext = PROTOCOL_VERSION`, `mem = client name`
CLIENT_REGISTRATION-_SUCCESS	Manager	Client	-
CLIENT_REGISTRATION-_FAILURE	Manager	Client	`mem = error text`
REGISTER_PUBLISHER	Client	Manager	`u1 = pub id`, `mem = topic name + type id`, `ext = EXT`
REGISTER_SUBSCRIBER	Client	Manager	`u1 = sub id`, `mem = topic name + type id`, `ext = EXT`
PUBLISHER_REGISTRATION-_SUCCESS	Manager	Client	`u1 = unique id`
PUBLISHER_REGISTRATION-_FAILURE	Manager	Client	`u1 = id`, `mem = error text`
SUBSCRIBER_REGISTRATION-_SUCCESS	Manager	Client	`u1 = unique id`
SUBSCRIBER_REGISTRATION-_FAILURE	Manager	Client	`u1 = id`, `mem = error text`
CONNECT_SUBSCRIBER	Manager	Client (Sub Owner)	`u1 = sub id`, `sock = connection`, `mem = pub client name`
CONNECT_PUBLISHER	Manager	Client (Pub Owner)	`u1 = pub id`, `sock = connection`, `mem = sub client name`
DELETE_PUBLISHER	Client/ Manager	Manager/ Client	`u1 = id (if to manager)`, `ext = EXT`
DELETE_SUBSCRIBER	Client/ Manager	Manager/ Client	`u1 = id (if to manager)`, `ext = EXT`
SUBSCRIBER_WAITING	Client (Sub)	Client (Pub)	`u1 = last frame id`
PUBLISHER_MSG_SEND	Client (Pub)	Client (Sub)	`u1 = frame id`, `mem = data`
SERVICE_ADVERTISE	Client	Manager	`u1 = id`, `ext = EXT`, `mem = name + req/res type ids`
SERVICE_ADVERTISE_SUCCESS	Manager	Client	`u1 = id`
SERVICE_ADVERTISE_FAILURE	Manager	Client	`u1 = id`, `mem = error text`

Message	Sender	Receiver	Fields / Notes
SERVICE_CONNECT_PROVIDER	Manager	Client (Provider)	`u1 = prov id, mem = client name, sock = connection`
SERVICE_CONNECT_CLIENT	Manager	Client (Client)	`u1 = client id, mem = provider name, sock = connection`
SERVICE_REGISTER_CLIENT	Client	Manager	`u1 = id, ext = EXT, mem = service name + req/res types`
SERVICE_CLIENT-_REGISTRATION_SUCCESS	Manager	Client	`u1 = client id`
SERVICE_CLIENT-_REGISTRATION_FAILURE	Manager	Client	`u1 = client id, mem = error text`
SERVICE_REQUEST	Client	Client	`u1 = request id, mem = request data`
SERVICE_RESPONSE	Client	Client	`u1 = request id, mem = response data`
SERVICE_DELETE_PROVIDER	Client/ Manager	Manager/ Client	`u1 = id, ext = EXT,`
SERVICE_DELETE_CLIENT	Client/ Manager	Manager/ Client	`u1 = id, ext = EXT,`
DOMAIN_STATE_REQUEST	Client	Manager	`mem = serialized state request`
DOMAIN_STATE	Manager	Client	`mem = serialized state`
DOMAIN_UPDATE_REQUEST	Client	Manager	`ext = 0 (stop) / 1 (start)`
DOMAIN_UPDATE	Manager	Client	`mem = update data`

Table 4.1: SM2 Protocol messages.

The EXT mentioned in Table 4.1 is simply a list of possible values that the ext field can take, that modify the meaning of the message. Currently only the ghost extension (section 3.7.5) is supported meaning that EXT allows only two values (no extension and ghost).

Chapter 5

SNG Implementation

The SM2 Network Gateway is a fairly complex application having to manage authentication, connection setup, secure communication and interaction with the SM2 local domain. Because of this it is implemented by separating the modules taking care of each individual feature and exposing only very limited abstractions to the main SNG logic.

The main modules are:

- Network communication.
- Multicast discovery.
- Configuration validation and loading.
- SM2 local domain interaction.
- Logging.

5.1 SNG Network Communication

The SNG's ability to communicate over the network must satisfy the following requirements:

- It must communicate with multiple peers.
- Communication must be authenticated and secure.
- It should be able to transmit messages that belong to different channels (topic/service) independently from each other (no head-of-line blocking between them).

5.1.1 QUIC

To satisfy the communication requirements the chosen protocol was QUIC [32].

QUIC is a UDP-based protocol that enables reliable, secure and multiplexed transport. It has progressively gotten more popular and has been adopted by large internet corporations as the preferred transport protocol [33].

The choice of QUIC stems primarily from its ability to enable several independent streams on a single connection. This means that it natively supports parallel transmission of data with not mutual head-of-line blocking. Achieving this with a protocol such as TCP would require to effectively establish multiple separate TCP connections.

5.1.2 Network Communication Module Implementation

Among several QUIC implementations available, MsQuic [34] was chosen for the SNG. MsQuic is an efficient QUIC implementation developed by Microsoft in C.

The SNG Network Communication module manages the resources needed to work with MsQuic and handles all the asynchronous logic that MsQuic requires. It exposes a limited abstraction to higher logic, mostly event queues.

In MsQuic, as required by the QUIC specification, Transport Layer Security(TLS) is an integral part of the communication protocol. The SNG implementation leverages this and allows authentication using X.509 certificates, optionally with a custom certification authority.

MsQuic is mostly oriented to a client-server communication model, but with limited effort it is possible to use it effectively for peer-to-peer communication.

5.1.3 Authentication

As mentioned before, the SNG uses X.509 certificates for authentication. The user can configure whether authentication is required or not, based on his requirements. Disabling authentication is, of course, discouraged for any real-world application that has to communicate over the network.

MsQuic, following the client-server model, forces the user to always provide credentials on the server side of the connection. To work around this, when the user wants to use the SNG without providing authentication, a self-signed certificate is generated on the fly.

Currently the suggested way of managing authentication between SNG peers is to let each have a certificate signed by a custom certification authority, and then provide the configuration with a private key file, a certificate file and a certification authority certificate file.

5.2 Multicast Discovery

While it is obviously possible to connect using MsQuic by knowing the IP address of the peer, it is quite common for robotics prototyping to have several machines connected to the same subnet, and to want automatic discovery between them.

A very common approach to implement this is to have the peers send out periodic multicast messages, that can be picked up by other peers to initiate the connection. ROS 2 for example utilizes multicast discovery between nodes. One advantage the SNG design has over ROS 2 in utilizing multicast discovery is that since network communication is centralized for each local domain, the amount of traffic generated by discovery is minimal, while it can become quite significant when each node (in ROS 2 DDS decentralized approach) has to independently discover the others.

SNG strictly uses IPv6 multicast, which offers several advantages over IPv4 multicast, such as scoped addressing [35].

5.3 SNG Configuration

The SNG was designed to be flexible and fit many possible use-case scenarios. For this reason its configuration is fairly extensive. In particular it is necessary to configure user preferences regarding authentication and the specifics with regard to the SM2 local domain.

The configuration is provided in the form of a json file.

5.3.1 General Configuration

Here is an example configuration, missing the SM2 configuration which is expanded upon in the next subsection.

```
 "sng_config" :
{
 "provide_auth" : true,
 "require_auth" : true,
 "key_file" : "/path/to/certs/key.pem",
 "cert_file" : "/path/to/certs/cert.pem",
 "ca_cert_file" : "/path/to/certs/ca-cert.pem",
 "tag" : "local_tag"
},
```

Most parameters are related to authentication. The **tag** parameter is a name that connected SNGs can see and based on which can make decisions. In particular as evidenced by the next section, it can be used to decide which of several configurations of for SM2 should be used when communicating with the peer.

5.3.2 SM2 Rulesets

As explained in section 3.7.1, the SNG allows for flexible configurations about what topics or services can be transmitted and how. Additionally it also allows to specify different behaviors for specific tags. This is especially useful when dealing with several kinds of robots, that need to be treated differently to optimize traffic.

Here is a example SM2 configuration for the SNG:

```
"sm2_config" :
[
{
 "topics" :
 {
  "base_rule" : "x"
 }
 "services" :
 {
  "base_rule" : "x"
 }
},

{
 "tag" : "allow_all_tag",
 "topics" :
 {
  "base_rule" : "="
 },
 "services" :
 {
  "base_rule" : "="
 }
},
```

```
{
 "tag" : "example_tag",
 "topics" :
 {
  "base_rule" : "=",
  "exceptions" :
  [
  {
   "name" : "allowed_topic",
   "rule" : "="
  },
  {
   "starts_with" : "allowed_out",
   "rule" : ">"
  }
  ]
 },
 "services" :
 {
  "base_rule" : "=",
  "exceptions" :
  [
  {
   "name" : "important_service",
   "rule" : "<"
  }
  ]
 }
}
]
```

As shown here, it is possible to define several configurations, with a default one applying to all tags, while tag-specific configurations only apply to matching tags. Each of these sub-configurations is called a ruleset. Every ruleset has a base rule, and exceptions based on the name of the topic/service.

Rule symbols are explained in tables 5.1 and 5.2.

Symbol	Meaning
x	Transmission is not allowed (regardless of peer rule)
=	Transmission allowed in both directions (peer can restrict with its own rule)
>	Transmission allowed from local domain to peer (for services this means that only the local can provide the service)
<	Transmission allowed from peer to local domain (for services this means that only the remote can provide the service)

Table 5.1. Transmission Rule Symbols and Their Meanings

Local Rule \ Peer Rule	x	$=$	$<$	$>$
x	x	x	x	x
$=$	x	$=$	$>$	$<$
$<$	x	$<$	x	$<$
$>$	x	$>$	$>$	x

Table 5.2. Combination of local and peer rules.

5.4 SNG SM2 Interface

The SNG naturally needs to be able to communicate with other Clients on the SM2 local domain. As mentioned in section 3.7.4 the SNG receives timely updates about changes in the local domain, and based on the updated local domain state and the combined configurations of the connected peers determines what on what topics and services it is necessary to have an interface.

This means that the SNG never keeps open an interface on the local SM2 side if it is unnecessary. This minimizes the overhead caused by the SNG within the local domain.

The SNG makes use of the ghost extension (see section 3.7.5) to avoid preventing the deletion of topics and services once every Client but the SNG has deleted their interfaces to them.

Furthermore, thanks to a custom SM2 front-end implementation it directly manages the SharedMemory buffer, without performing unnecessary copies or serializations.

Chapter 6

Experimental Results

To evaluate the SM2 architecture and implementation a number of tests were conducted, putting it up against ROS 2. The different tests were designed to put a strain on different aspects of the communication, and always tried to remain realistic and relevant to real-world use cases.

In general, tests were conducted with 3 configurations:

- ROS 2 Humble with DDS (FastDDS)
- ROS 2 Humble with Zenoh RMW
- SM2 with SNG

The different tests covered local and network communication. They tried to stress the different communication stacks in ways that mirrored typical development patterns in robotics prototyping.

The libraries and middlewares used for testing were left in their default configurations.

6.1 Expectations

ROS 2 with DDS has shown several times in the past to suffer from an extreme performance degradation when used over unreliable networks. As such it is expected for it to fare far worse in network communication scenarios.

The new RMW for ROS 2 based on Zenoh on the other hand, follows a similar philosophy to SM2 in that communication over the network is conducted in a centralized way via router nodes. For this reason Zenoh RMW and SM2 should show similar behaviors in a unreliable network scenario.

6.2 Network Communication Results

Network communication was tested by having two machines communicate over a wi-fi network.

The wi-fi network in question was purposely isolated, no traffic aside from what resulted from testing was generated.

The main test for communication over the network consisted of:

- On machine 1, a publisher node publishing at a fixed frequency of 30Hz a 1MB message.
- On machine 1, a subscriber node receiving the messages.
- On machine 2, a subscriber node receiving the messages.

This test was designed to both determine the transmission performance of each setup, but also to identify possible impacts of the network transmission on local communication.

The message size and frequency were chosen to be realistic and at the same time to put saturate network bandwidth, in order to see setup performance in non-ideal conditions.

Figure 6.1. Network test conducted with SM2.

The results of this test match the expectations. SM2 (Figure 6.1) and ROS 2 with Zenoh (Figure 6.2) perform very similarly. When constrained by insufficient bandwidth they obviously cannot transmit all messages, therefore they drop a more or less constant percentage of messages. They both exhibit a few latency spikes but latency remains fairly constant around 250ms for Zenoh and 300ms for SM2.

However the most important result from this test is the extreme performance degradation suffered by the ROS 2 with DDS setup. As shown in Figure 6.3, the startup of the remote node and the consequent establishment of network communication leads to an extreme reduction in frequency on the publisher side. This behavior is extremely dangerous for safety critical applications, as the data flow from this publisher could be used in real-time operations such as Visual Inertial Odometry or other vital robot systems. This behavior alone is enough to disqualify ROS2 with DDS for use in a scenario similar to the one of these tests. It is important to note that this test for DDS was repeated with multiple QoS configurations, such as reliable and best-effort settings. All of them gave very similar results.

This problem affects communication regardless of whether other transport means are used for local communication, for example shared memory between local nodes. The overhead on the publish call itself is so great that it prevents the calling thread to match the desired frequency.

Overall ROS 2 with Zenoh and SM2 show similar results, with latency and throughput slightly

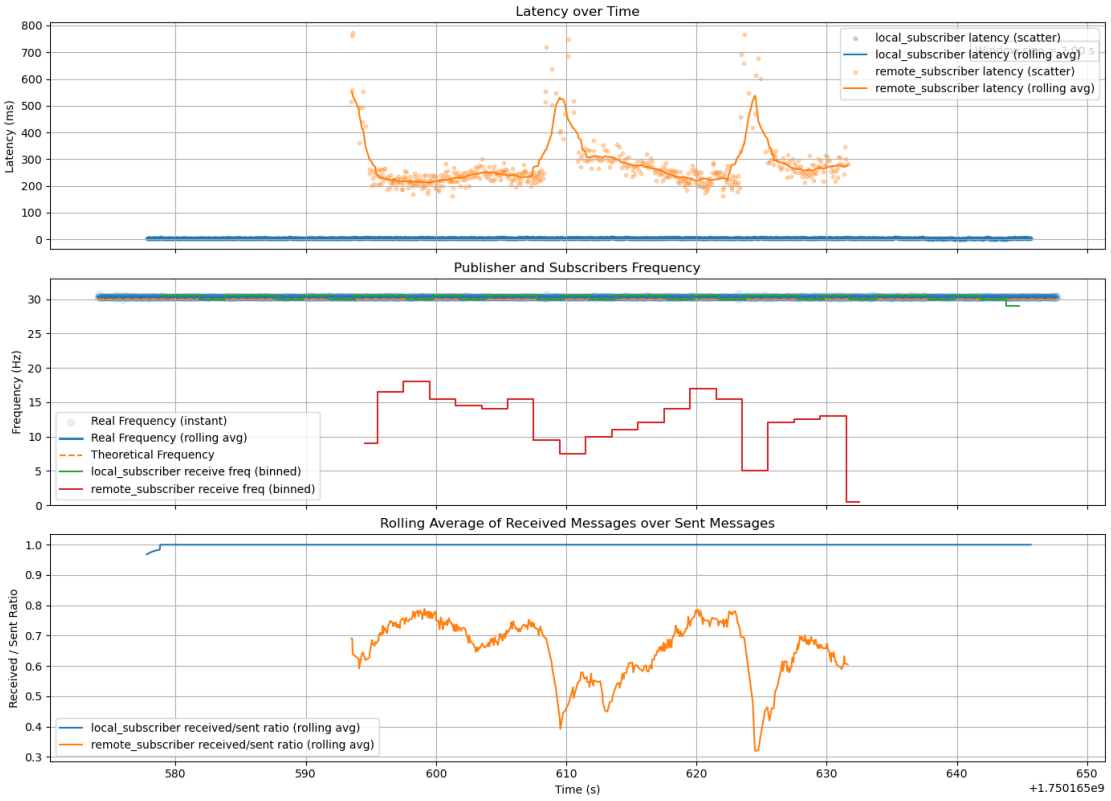

Figure 6.2. Network test conducted with ROS 2, Zenoh RMW.

favoring Zenoh. The FastDDS setup performs far worse than the other two, with throughput being lower than half of both SM2 and Zenoh, and latencies easily double.

Figure 6.3. Network test conducted with ROS 2, FastDDS.

6.3 Local Communication Results

The main goal of the SM2 library is to provide efficient and reliable local IPC. The robotics scenarios it plans to enable are ones where local communication is far more relevant, and its performance directly impacts the behavior of safety-critical real-time applications.

For this reason these tests put an emphasis on the performance of local communication.

The results from the network tests leave Zenoh RMW as the only real option for ROS 2 in unreliable network conditions. For this reason most of the following tests were only conducted on the SM2 and Zenoh RMW setups.

6.3.1 Frequency Test

This test is designed to observe the changing behavior of local IPC when increasing message frequency, and consequently throughput.

The test is conducted as follows:

- One publisher publishes a fixed size message at an increasing frequency.
- Frequency starts at 10Hz.
- Frequency increases by 1Hz every 10 published messages.
- One subscriber node records all received messages.

This test is conducted with a large message ($\sim$ 5.5 MB). While this message size may seem too large for a realistic test, it is actually a fairly common message size, especially when working with images. I corresponds roughly to one raw color full-hd image or (as it is quite common) a message comprised of several synchronized images.

As shown by figures 6.4 and 6.5, again SM2 and Zenoh show similar behavior.

It is important to interpret the graphs in figures 6.4 and 6.5 correctly. The bottom graph shows the actual frequency achieved by the publisher node. Both SM2 and Zenoh have a maximum achievable frequency past which it is impossible to go due to the delays introduced by publication. From now on this will be called limit frequency.

The top graph shows the latency at different frequencies. The behavior of latency over frequency should be evaluated by taking into account the fact that real frequency does not got past the limit frequency, meaning that all data past the limit frequency (dotted red line) is to be considered behavior at the limit frequency.

The middle graph shows the amount of dropped messages. In general both SM2 and Zenoh, in almost all conditions, drop very few messages. This suggests that for both of them the bottleneck is likely to be the publisher node and not the subscriber.

Both Zenoh and SM2 show more or less constant latency at frequencies below the limit frequency. They both show an uptick (slight for SM2, pronounced for Zenoh) in latency leading up to the limit frequency and a (expected) constant latency at the limit frequency.

Zenoh shows a similar albeit slight worse limit frequency compared to SM2 ($\sim$ 250 vs $\sim$ 300 Hz). As shown in figure 6.6 Zenoh suffers from a significantly higher latency at the limit frequency, about 100% slower than SM2.

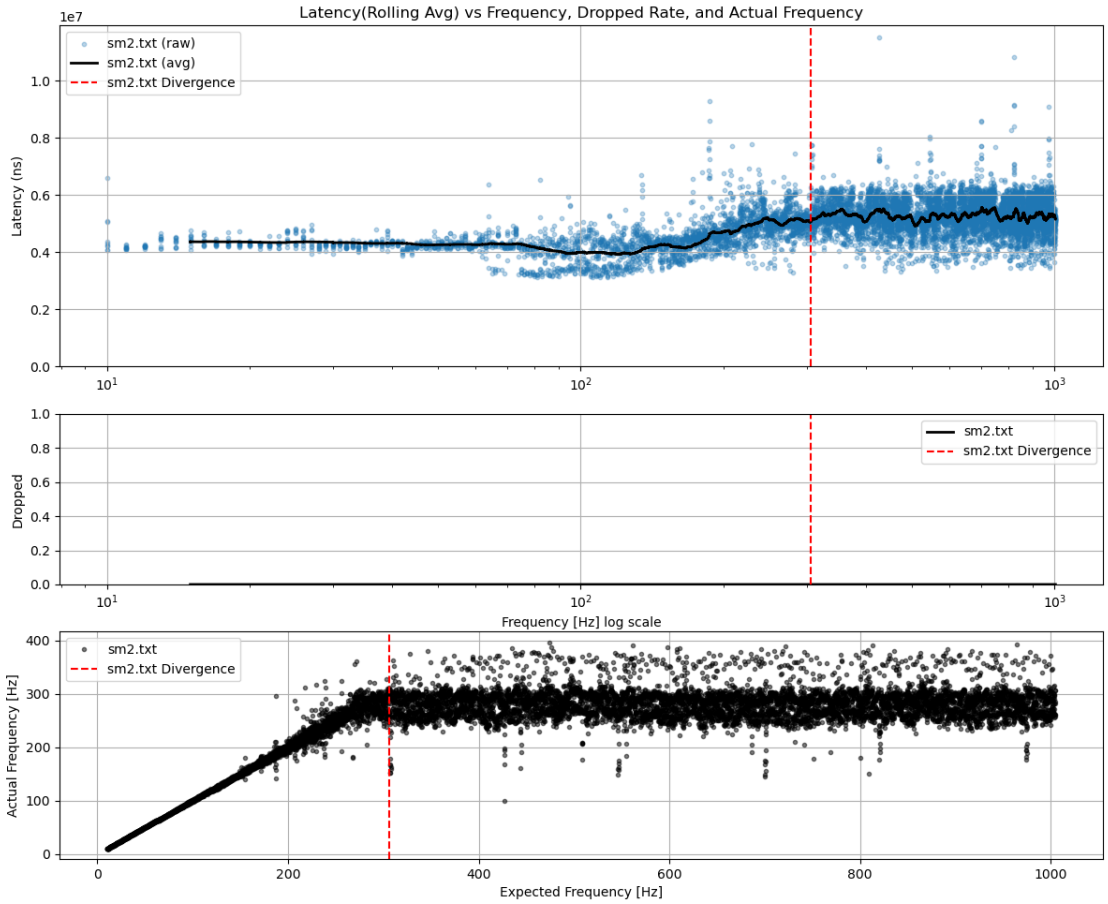

Figure 6.4. Frequency test conducted with SM2.

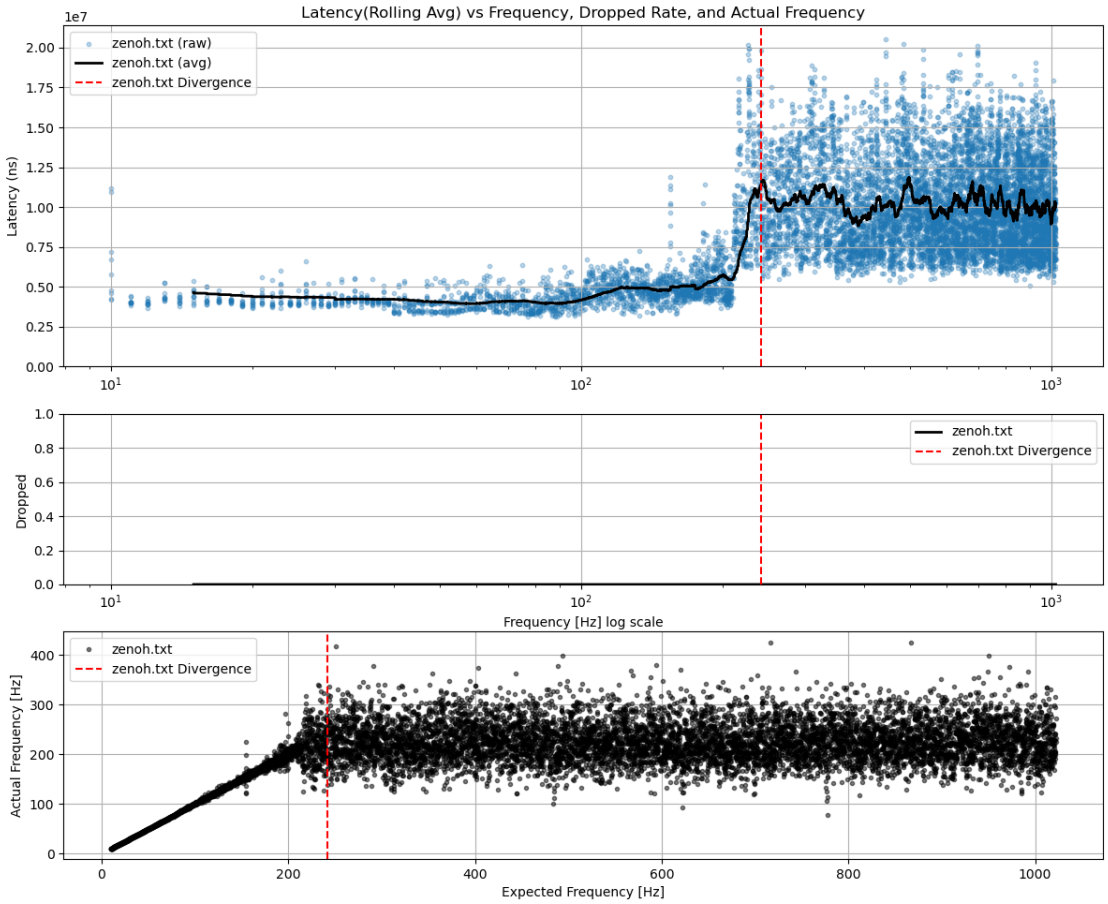

Figure 6.5. Frequency test conducted with ROS 2 and Zenoh RMW.

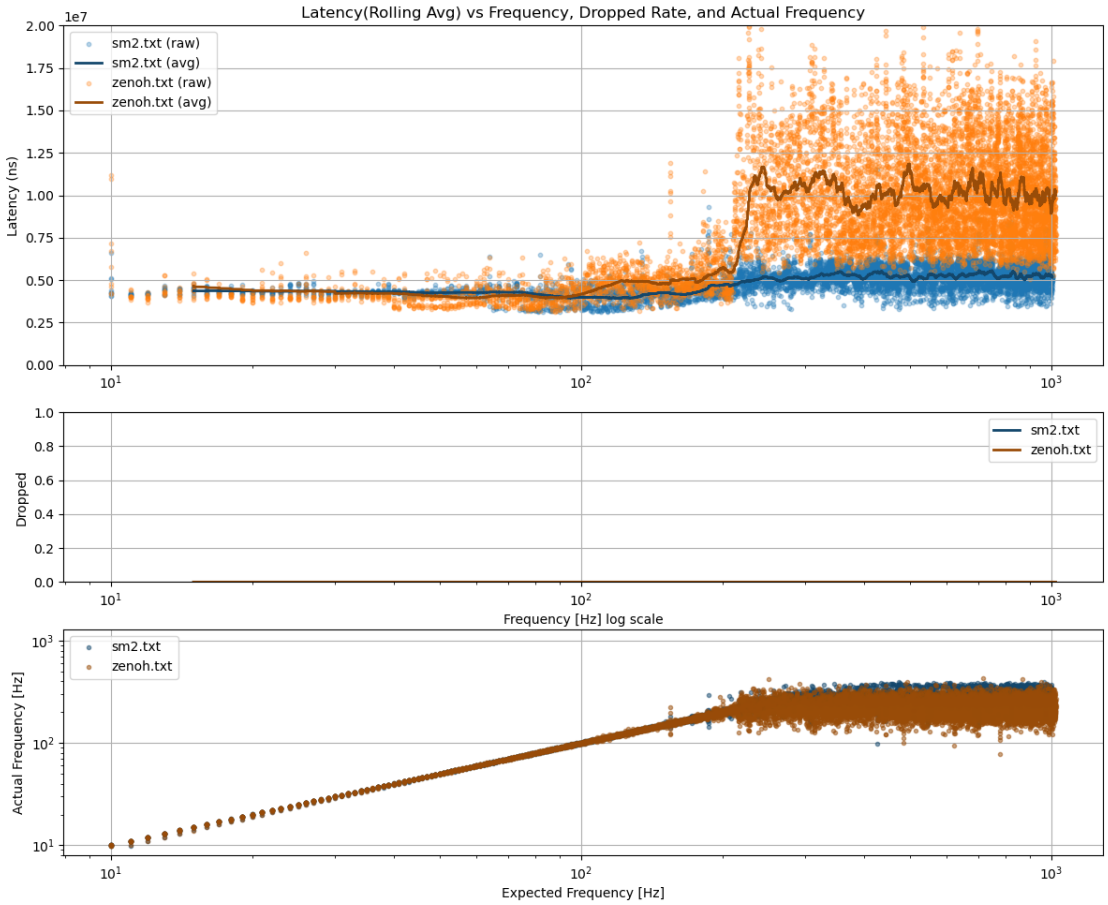

Figure 6.6. Results of tests in Figures 6.4 and 6.5 compared.

6.3.2 Frequency Test with Small Messages

The same test with increasing frequency conducted with a smaller message ($\sim$ 100B) shows an interesting behavior (Figure 6.7).

Importantly no limit frequency is reached up to the maximum tested frequency of around 4500Hz. The spread in achieved frequency is due to lacking precision of sub-millisecond sleeps.

Latency decreases with frequency. One possible explanation for this is increased cache retention due to shorter waiting times.

Some messages are lost at higher frequencies, showing that in this scenario the subscriber actually becomes the bottleneck.

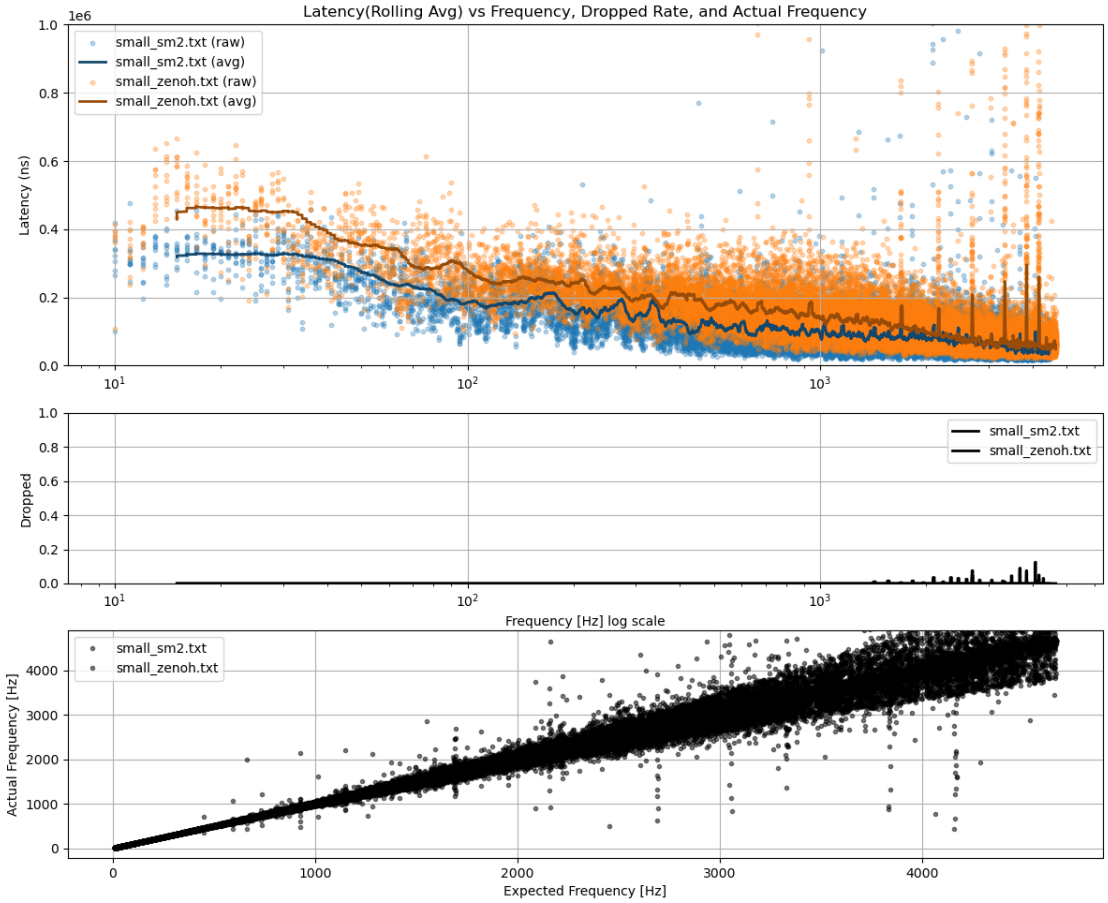

Figure 6.7. Results of frequency tests for SM2 and Zenoh compared.

6.3.3 Frequency Test with Multiple Subscribers

It is very common in robotics to have one sensor data stream feed into multiple separate processes that utilize it for different things. For example there could be algorithms performing Visual Inertial Odometry, object detection, obstacle avoidance and mapping. It is therefore relevant to test the same scenario with more than one subscriber.

This test is conducting as the one in section 6.3.1, but with five nodes subscribing instead of one.

Figures 6.8 and 6.9 show both SM2 and Zenoh suffer a worsening in performance when more subscribers are added.

Interestingly, when adding multiple subscribers Zenoh shows consistently different latencies for each, potentially indicating a correlation with subscribe order.

The other important result is that both SM2 and Zenoh have a lower limit frequency when the subscriber count is increased. This is coherent with the higher load on the publisher, having to serve more subscribers.

Comparing the results (Figure 6.10), SM2 delivers messages at a significantly lower latency with Zenoh being up to more than 100% slower. This is true both at low and limit frequencies. The difference in latency for low frequencies is particularly relevant, as many use cases require streaming raw images to multiple nodes at the same time.

Maybe more importantly Zenoh shows a massive collapse of the limit frequency (from ~ 250 to ~ 70 Hz, compared to SM2's drop from ~ 300 to ~ 250 Hz), meaning that Zenoh introduces far more substantial overhead to the publishing thread when multiple subscribers are receiving messages.

The frequency range from ~ 70 to ~ 250 Hz shows therefore critically worse performance for Zenoh. This frequency range together with this message size is comparable to that of some high frequency cameras used in robotics, and therefore a relevant scenario.

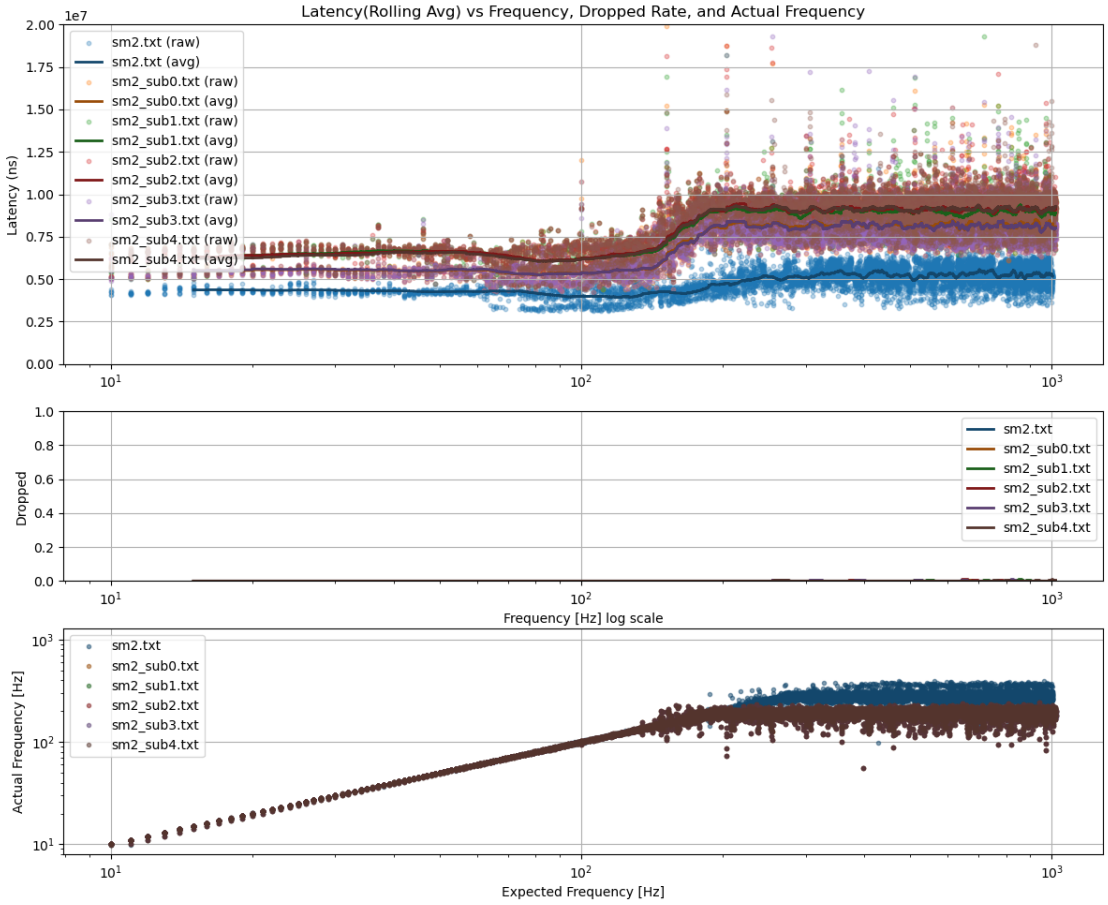

Figure 6.8. Results of frequency tests with one subscriber and with 5 subscribers using SM2 compared.In blue the results of Figure 6.4.

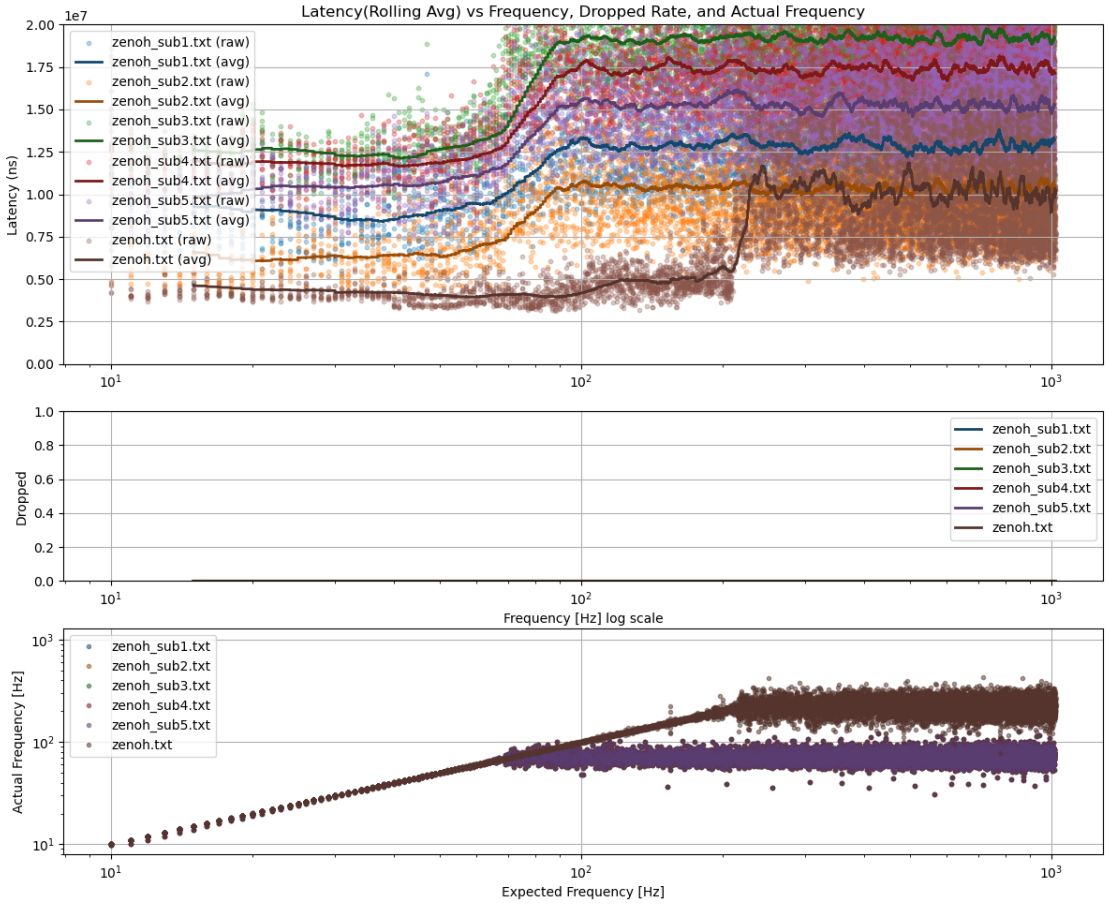

Figure 6.9. Results of frequency tests with one subscriber and with 5 subscribers using ROS 2 with Zenoh RMW compared. In brown the results of Figure 6.5.

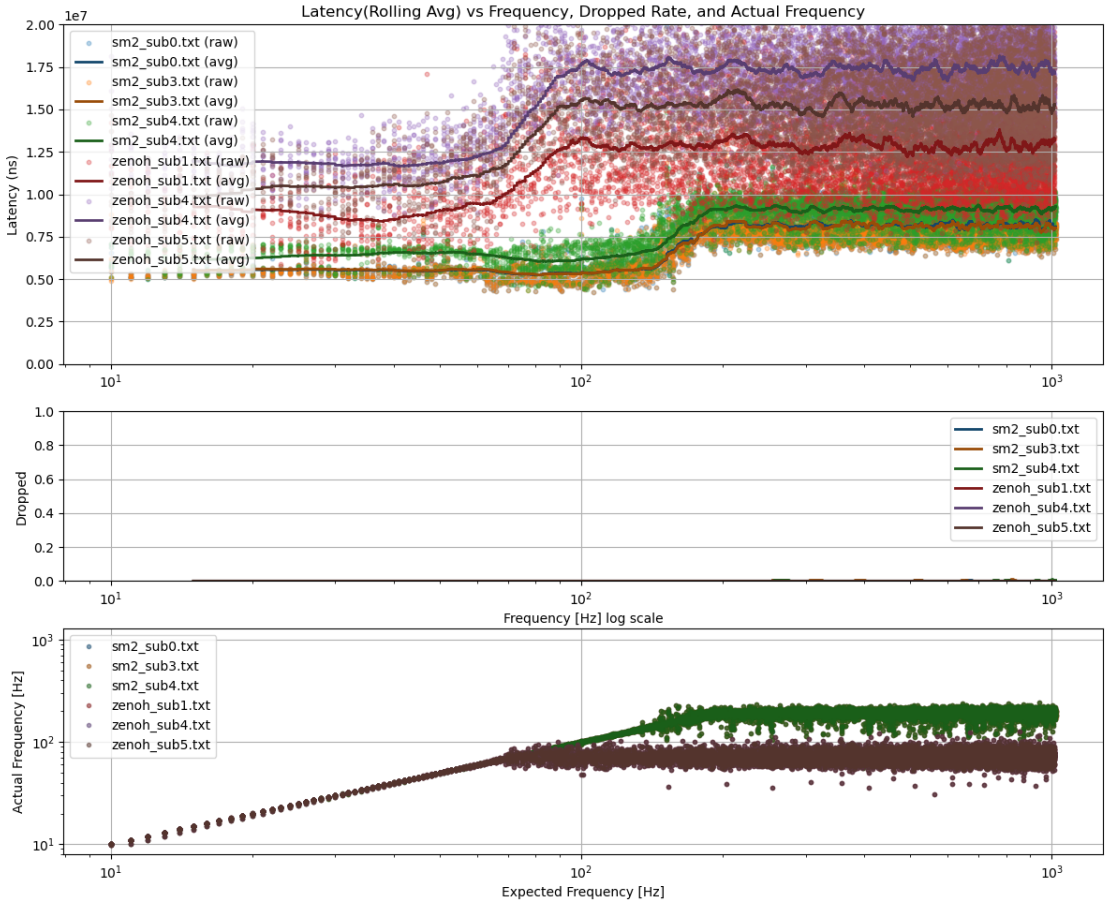

Figure 6.10. Results of frequency tests (Figures 6.8, 6.9) with 5 subscribers with SM2 and Zenoh compared (only the middle 3 subscribers are shown for each). In the latency graph the 3 datasets with higher latencies are Zenoh's, the others are SM2.

6.3.4 Impact of multiple Subscribers and Publishers

Previous tests suggest that the impact of a more complex local domain topology (i.e. multiple subscribers or multiple publishers on the same topic) can be significant. To investigate this, a test with the following characteristics was conducted:

- A large message ($\sim$ 5.5MB) is published at a fixed frequency of 30 Hz.
- The test is executed multiple times with different numbers of publishing and subscribing nodes.
- Message latency is measured.

As shown by figures 6.11 and 6.12, in 1-to-1 communication SM2 and Zenoh show very similar latency (in line with results in Figure 6.6).

As the number of subscribers and publishers increases, both Zenoh and SM2 exhibit higher latency. Notably, Zenoh's average latency increases approximately three times more than SM2's with additional subscribers, and about twice as much with additional publishers. Zenoh also demonstrates significantly greater jitter when multiple subscribers are present.

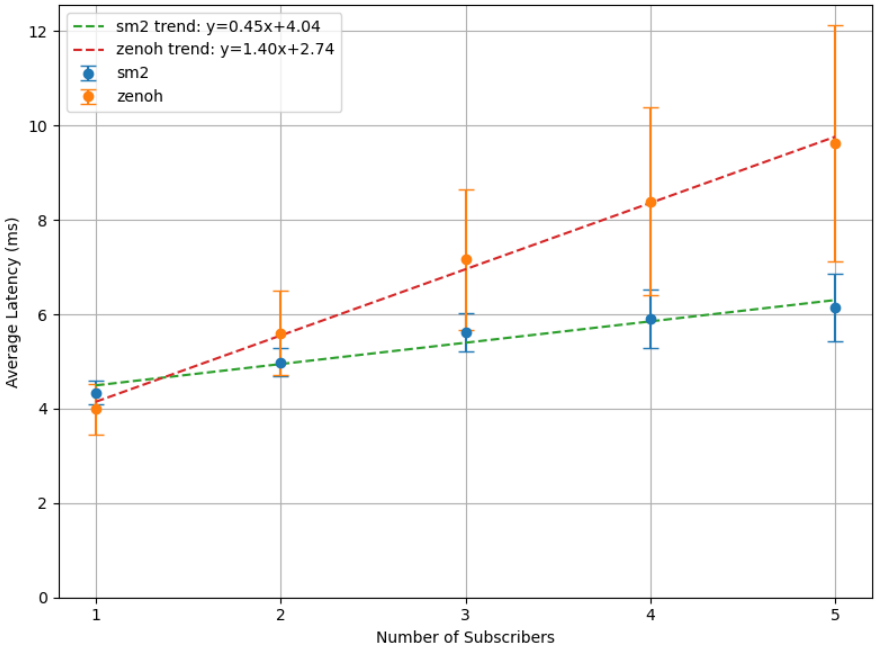

Figure 6.11. Compared average message latency with one publisher and different numbers of subscribers. Message size is 5.5MB, frequency 30Hz. Error bars are standard deviation.

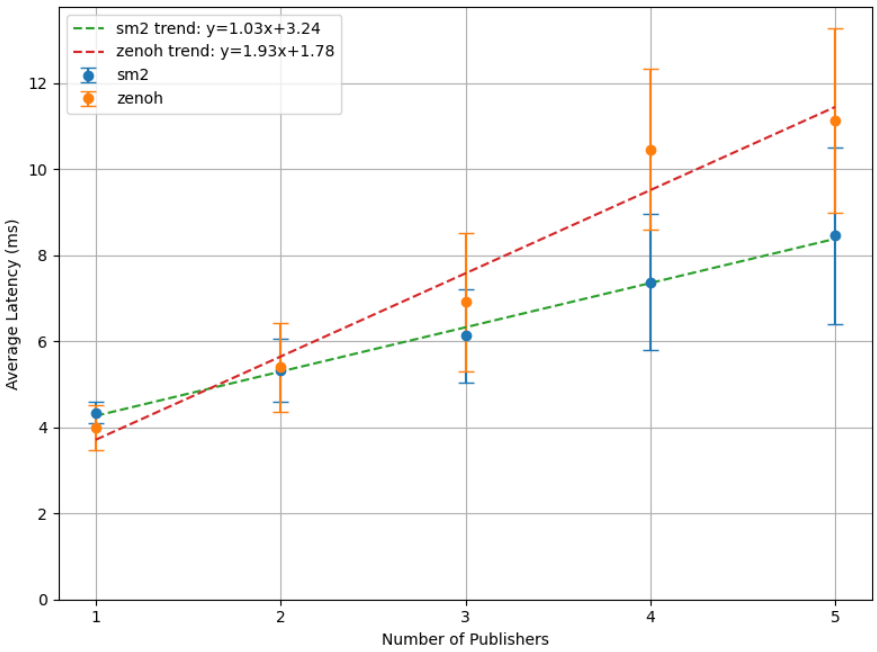

Figure 6.12. Compared average message latency with one subscriber and different numbers of publishers. Message size is 5.5MB, frequency 30Hz. Error bars are standard deviation.

6.3.5 Investigating the Impact of Message Size

The test results shown in sections 6.3.1 and 6.3.2 demonstrate that different message sizes lead to significant variation both in terms of latency and introduced overhead.

Figures 6.13, 6.14 show results for tests conducted with increasing message size. In particular:

- One publisher publishes messages at a fixed frequency of 30Hz.
- One subscriber receives the messages and logs.
- The message size starts from 10 bytes, and progressively increases every 10 published messages by $max(1, current\_size/100)$ bytes.

As evidenced by Figure 6.13 again Zenoh and SM2 exhibit similar behavior. The latency data remains constant with small message sizes, up to about 100KB messages. It then grows linearly with message size for larger messages. This behavior is most likely due to message copies (serialization/deserialization) becoming the main source of delay for large messages.

SM2 shows consistently lower latency compared to Zenoh RMW. As evidenced by Figure 6.14 the difference in latency becomes especially pronounced with large messages, with Zenoh showing latencies above 10ms, with SM2 around 6ms.

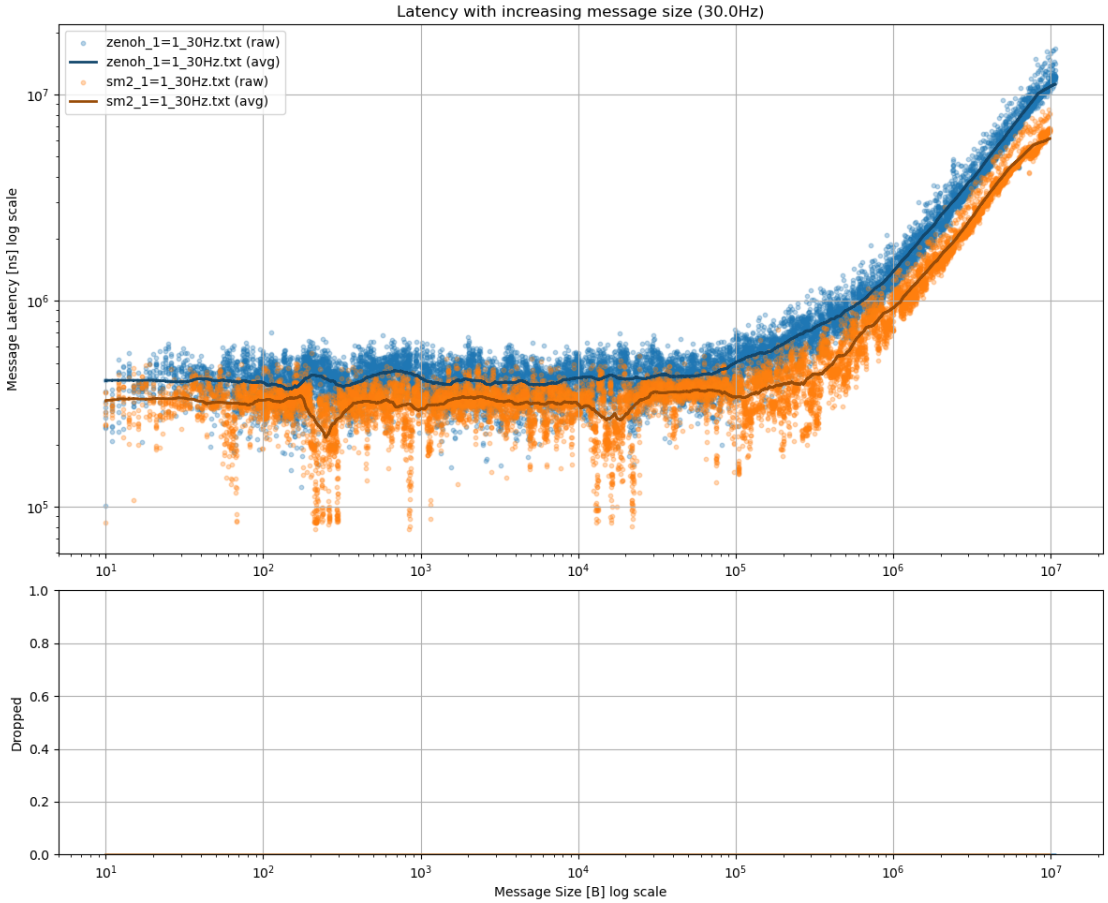

Figure 6.13. Message latency over varying message sizes. Results for ROS 2 + Zenoh RMW and SM2 compared.

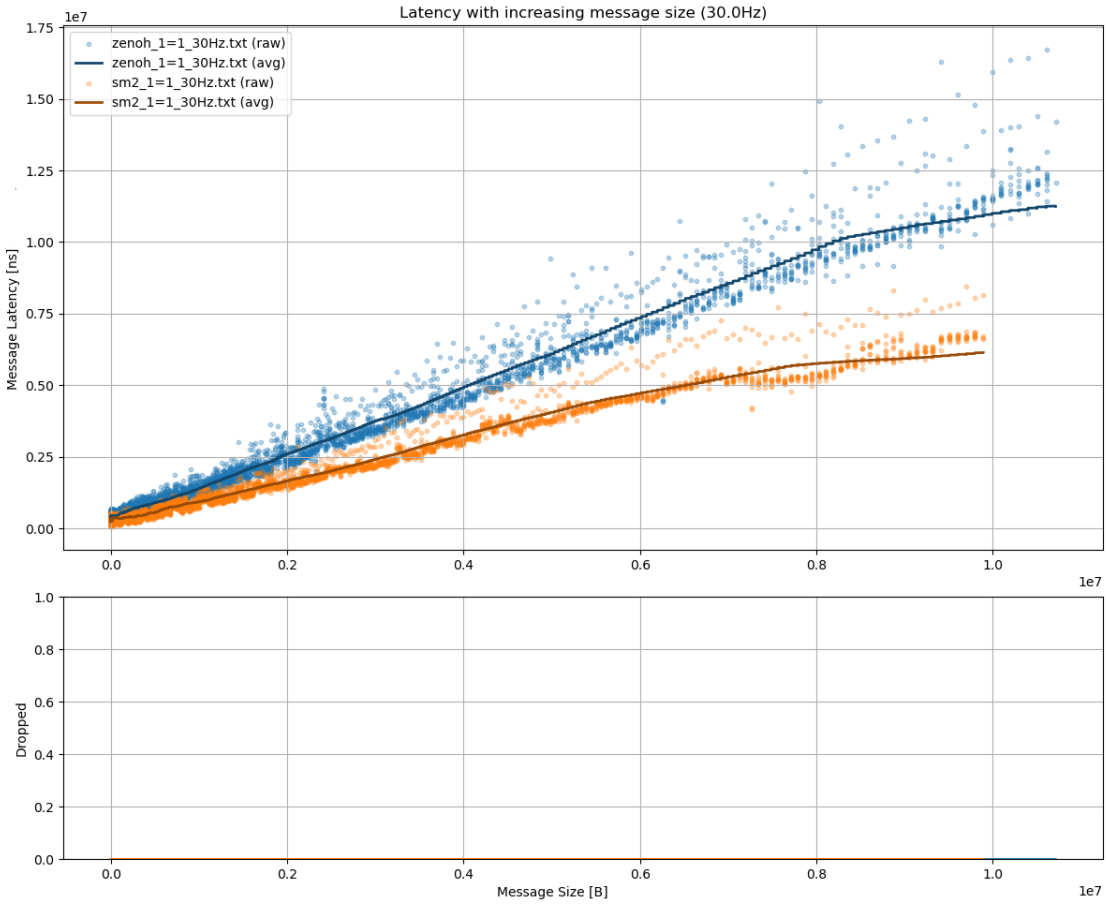

Figure 6.14. Same results as Figure 6.13 but in linear scale.

6.3.6 System Resource Utilization

Lastly, it is important to investigate the amount of system resources used by the different communication libraries. In particular this section focuses on average CPU and RAM usage. To conduct this test multiple scenarios where created with node publishing and subscribing similarly to previous tests. While the communication was ongoing the system statistics regarding process memory usage and CPU time were periodically logged by an external process.

Figure 6.15 clearly shows that communication over ROS2 using Zenoh RMW takes a significantly higher toll on system resources both in the scenario with 1 subscriber and the one with 5. In particular, compared to SM2, Zenoh utilizes almost twice as much CPU in the 1v1 scenario, and three times as much in the 1v5 scenario.

Memory utilization is also heavily in favor of SM2, with Zenoh consistently occupying four times as much.

Such an impact on system resources can be very limiting in real-world applications that require complex local domain topologies and have limited computational and memory capacity.

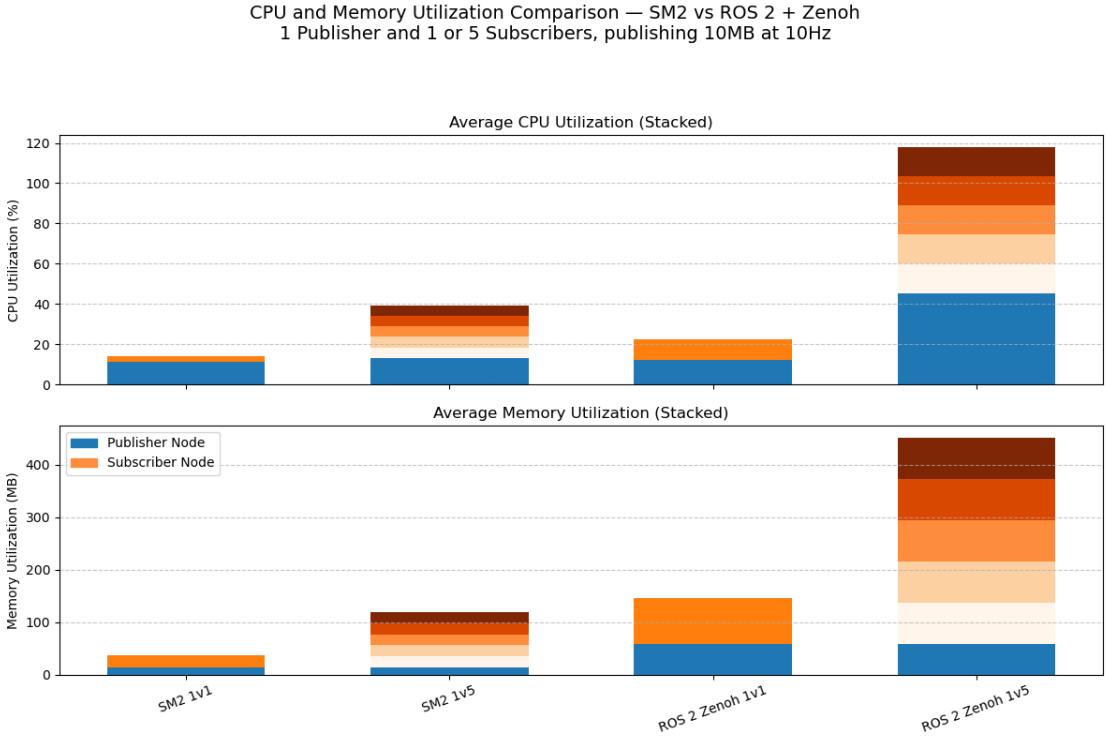

Figure 6.15. System resource utilization with SM2 and Zenoh.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This thesis had the goal of developing a library for Inter Process Communication for robotics applications alternative to ROS 2. The need for such a library was rooted in the deficiencies shown by ROS 2 and its middlewares (see for example sections 1.2.3, 2.3.2). The library design and implementation was driven by the requirements of software development for robotics prototyping, specifically within the DRAFT team (section 1.2).

The main problems that ROS 2 presented and the developed library (SM2) aimed to address were:

- Reliability of local communication regardless of network conditions.
- Respect of real-time constraints and more generally communication performance.
- Utilization of system resources, which directly translates to ability to scale within a single machine.
- Ease of use and limiting API design choices, which directly lead to suboptimal decisions during development.

SM2 achieves these goals both thanks to design and implementation decisions. The library structure mirrors the necessity of ironclad local communication, by splitting it from the network side. With reliability being a major interest, particular attention was put in preventing malicious or unintentional actions of any process from impacting the stability or communication of others. Major emphasis was put on efficiency, both in terms of performance and in terms of resource utilization. The parallel execution of library tasks and well coordinated synchronization between events on potentially different processes ensure that API calls and internal library routines cause minimal overhead. The practical choices in terms of Inter Process Communication primitives and the way to abstract them ensure proper cleanup of all resources even in the event of process instability. All the while ensuring that no unnecessary system resources are requested.

The experimental results (section 6) show that SM2 performs significantly better than ROS2 used with its default and most widely adopted middleware (FastDDS) in constrained network conditions, while being on par with the more recent Zenoh-based middleware for ROS2.

Local communication results (section 6.3) place SM2 consistently on top compared to the best available contender (ROS2 with Zenoh RMW). In particular SM2 showed lower latency and significantly more limited overhead, especially in situations with multiple subscribers and publishers. In general SM2 performed better in all situations that involved some kind of stress

or increase in scale, such as increased message size, frequency or increased number of nodes. Particular attention should be put on the comparison in terms of system resource utilization, with ROS 2 requiring four times as much memory and up to three times as much CPU time compared to SM2.

These gains in performance and efficiency directly translate to the developer's ability to deploy more complex software on less powerful machines. They allow to perform the same tasks at higher frequency or to save CPU time for additional processes. All these advantages are critical to many robotics projects that deal with deploying complex software (often with real time constraints) on limited hardware.

7.2 Future Work

The development of a Inter Process Communication library that can provide comparable features to ROS2 while improving in several aspects was a challenging task. Given the limited time, a number of areas are left that could benefit from further work.

Within the local communication library itself extended platform support is needed to make it available on all major systems. Additional effort should be put into generalizing the serialization to be compatible on more machines, for example ones with different endiannes.

The network communication especially requires some further extension in terms of features. Additional parallelization of message transmission would allow to fully exploit QUIC's parallel streams. It is worth considering having the SNG create separate interfaces for each open connection, to better exploit the disconnect-on-error behavior of SM2 clients and single out offending connected peers.

From an evaluation point of view, it would be worth comparing SM2's performance against other, less used, IPC libraries. Serialization performance could be specifically evaluated both against ROS2 middleware and specialized serialization libraries. It would be worth exploring potential integration of a serialization/reflection library in place of part of the custom SM2 type access module, to support a wider array of standard library types.

Bibliography

[1] Davide Falanga, Suseong Kim, and Davide Scaramuzza. How fast is too fast? the role of perception latency in high-speed sense and avoid. *IEEE Robotics and Automation Letters*, 4(2):1884–1891, 2019.

[2] M McIlroy, EN Pinson, and BA Tague. Unix time-sharing system. *The Bell system technical journal*, 57(6):1899–1904, 1978.

[3] Draft polito. `https://www.draftpolito.it/`. Accessed: 2025-06-26.

[4] Leonardo drone contest. `https://www.leonardo.com/en/innovation-technology/open-innovation/drone-contest`. Accessed: 2025-06-26.

[5] Steve Macenski, Alberto Soragna, Michael Carroll, and Zhenpeng Ge. Impact of ros 2 node composition in robotic systems. *IEEE Robotics and Automation Letters*, 8:3996–4003, 2023.

[6] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, 2009.

[7] Ros master. `https://wiki.ros.org/Master`. Accessed: 2025-07-22.

[8] Steve Cousins. Exponential growth of ros. *IEEE Robotics & Automation Magazine*, 18(1):19–20, 2011.

[9] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science robotics*, 7(66):eabm6074, 2022.

[10] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ros2. In *Proceedings of the 13th international conference on embedded software*, pages 1–10, 2016.

[11] Tully Foote Katherine Scott. Ros metrics report 2023, 2023. Accessed: 2025-06-18.

[12] Fastdds. `https://fast-dds.docs.eprosima.com/en/latest/index.html`. Accessed: 2025-06-18.

[13] Cyclonedds. `https://cyclonedds.io/`. Accessed: 2025-06-18.

[14] Zenoh rmw. `https://github.com/ros2/rmw_zenoh`. Accessed: 2025-06-19.

[15] Loïck Chovet, Gabriel Garcia, Abhishek Bera, Antoine Richard, Kazuya Yoshida, and Miguel Olivares-Mendez. Performance comparison of ros2 middlewares for multi-robot mesh networks in planetary exploration. 07 2024.

[16] Jiaqiang Zhang, Xianjia Yu, Sier Ha, Jorge Peña Queralta, and Tomi Westerlund. Comparison of middlewares in edge-to-edge and edge-to-cloud communication for distributed ros 2 systems. *Journal of Intelligent & Robotic Systems*, 110(4):162, November 2024.

[17] sensor_msgs/image. `https://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/Image.html`. Accessed: 2025-06-19.

[18] Messages serialization and adapting types. `https://wiki.ros.org/roscpp/Overview/MessagesSerializationAndAdaptingTypes`. Accessed: 2025-06-19.

[19] Realsense ros implementation. `https://github.com/IntelRealSense/realsense-ros/blob/ros2-master/realsense2_camera/src/base_realsense_node.cpp\#L916`. Accessed: 2025-06-19.

[20] Unix manual. https://man7.org/linux/man-pages/man2/. Accessed: 2025-06-19.
[21] Unix domain socket. https://man7.org/linux/man-pages/man7/unix.7.html. Accessed: 2025-06-19.
[22] memfd_create. https://man7.org/linux/man-pages/man2/memfd_create.2.html. Accessed: 2025-06-19.
[23] [patch v4 0/3] permit write-sealed memfd read-only shared mappings. https://patchew.org/linux/cover.1697116581.git.lstoakes@gmail.com/. Accessed: 2025-06-20.
[24] eventfd. https://man7.org/linux/man-pages/man2/eventfd.2.html. Accessed: 2025-06-20.
[25] Reflection for c++26. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p2996r12.html. Accessed: 2025-06-23.
[26] Compiler support for c++20. https://en.cppreference.com/w/cpp/compiler_support/20. Accessed: 2025-06-23.
[27] Type info hash reference page. https://en.cppreference.com/w/cpp/types/type_info/hash_code.html. Accessed: 2025-06-23.
[28] C++ template partial specialization. https://eel.is/c++draft/temp.spec.partial. Accessed: 2025-06-23.
[29] C++ base types data models. https://en.cppreference.com/w/cpp/language/types.html. Accessed: 2025-06-23.
[30] C preprocessor tricks, tips, and idioms. https://github.com/pfultz2/Cloak/wiki/C-Preprocessor-tricks,-tips,-and-idioms. Accessed: 2025-06-23.
[31] C++ constexpr. https://en.cppreference.com/w/cpp/language/constexpr.html. Accessed: 2025-06-23.
[32] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021. https://www.rfc-editor.org/rfc/rfc9000.html.
[33] How facebook is bringing quic to billions. https://engineering.fb.com/2020/10/21/networking-traffic/how-facebook-is-bringing-quic-to-billions/. Accessed: 2025-06-24.
[34] Msquic. https://github.com/microsoft/msquic. Accessed: 2025-06-24.
[35] Ipv6 multicast address space registry. https://www.iana.org/assignments/ipv6-multicast-addresses/ipv6-multicast-addresses.xhtml. Accessed: 2025-06-24.