Bachelor's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Halvor Bergstøl Birkeland
Ovidijus Cironka
Maja Austin Fauske
Wilhelm Merkesvik

Distributed Optimization Based Adaptive Underwater Communication Schemes

Bachelor's thesis in Electrical Engineering
Supervisor: Behdad Aminian

May 2023

NTNU
Norwegian University of
Science and Technology

Halvor Bergstøl Birkeland
Ovidijus Cironka
Maja Austin Fauske
Wilhelm Merkesvik

Distributed Optimization Based Adaptive Underwater Communication Schemes

Bachelor’s thesis in Electrical Engineering
Supervisor: Behdad Aminian
May 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Title: Distributed Optimization Based Adaptive Underwater Communication Schemes		
Candidates: Halvor Bergstøl Birkeland Ovidijus Cironka Maja Austin Fauske Wilhelm Merkesvik	**Projectnumber:**	E2308
	Submission date:	28.05.2023
Study:	Electrical Engineering - BIELEKTRO	
Field of study:	Automation and Robotics	
Supervisor:	Behdad Aminian	
Institute:	Department of Engineering Cybernetics (NTNU ITK)	
Client:	NTNU ITK v/Damiano Varagnolo	
Contact person:	damiano.varagnolo@ntnu.no	
Keywords EvoLogics, Subnero, Modem, Acoustic, Underwater Communication, Raspberry Pi, ROS 2, Convex Optimization, Consensus Algorithm, Interior Point Method, JANUS, ANEP-87, SDMSH, Multiprocessing, Cost Function, UnetStack, Python, C++, OFDM, Distributed System		

Preface

This final project represents the culmination of a three-year bachelor's study in Electrical Engineering at the Norwegian University of Science and Technology (NTNU). These three years have been an incredible journey of growth and learning for each member of our group, and we are glad to be ending these years with a fun and challenging project.

We wish to thank our supervisor, Behdad Aminian, for his support and knowledge, and continuous optimism. He has always been available for any question and has never said no whenever the group needed assistance.

Throughout this project, our client Damiano Varagnolo has shown understanding and continuous assistance with direction. We appreciate his time and dedication towards the success of this project.

By providing assistance and answering many of our questions, Emil Wengle has shown his interest and knowledge towards the progress and development of this project. His input in our process and thesis guided us past a bumpy road.

We want to also give thanks to Federico Iadarola for teaching us theory about optimization and showing us his own work which was an excellent head-start for this project.

Summary

This thesis presents the development of a communication mechanism for the EvoLogics modems using the JANUS protocol, an underwater communication standard, and the integration of an optimization algorithm with the Subnero modems using ROS 2 (Robot Operating System) as the framework.

The optimization algorithm, based on a robust and asynchronous Newton-Raphson consensus, ra-NRC, combined with underwater modems and ROS 2, aims to address the challenges faced in underwater communication schemes, by improving the efficiency and accuracy of data transmission by using convex optimization. The other key aspect lies within the development of software, meant to establish the possibility of mixed subsea communication with the EvoLogics and Subnero underwater modems. This communication is based on the ANEP-87 standard, the standard for underwater communication for NATO, an international standard for subsea communication using acoustic waves. Additionally, the use of SDMSH and JANUS for EvoLogics modems has been simplified.

Finally, these aspects are integrated together to achieve optimized mixed-vendor modem communication. The optimization of communication parameters to achieve efficient communication has yet to be applied. However, it has been tested through a UnetStack simulation that the convex optimization works between two virtual modems. Various tools are used to achieve a modular and comprehensive system, with the possibility to be built further for custom needs.

Sammendrag

Denne bacheloroppgaven presenterer utviklingen av et kommunikasjonssystem for EvoLogics modemer som bruker standardprotokollen JANUS for undervannskommunikasjon. Det blir også integrert en optimaliseringsalgoritme for Subnero modemer ved bruk av ROS 2 (Robot Operating System) som rammeverk.

Algoritmen for optimalisering er basert på en robust og asynkron Newton-Raphson konsensus (ra-NRC), som kombinert med undervannsmodemer og ROS 2 skal ta stilling til utfordringer ved undervannskommunikasjon. Dette ønskes å oppnås ved å forbedre effektiviteten og nøyaktigheten av sending av data ved bruk av konveks optimalisering. Et av hovedpunktene i oppgaven ligger i utviklingen av programvare som etablerer mulighet for kommunikasjon mellom undervannsmodem fra EvoLogics og Subnero. Denne kommunikasjonen er basert på ANEP-87 standard, og er en standard brukt av NATO for undervannskommunikasjon ved bruk av akustiske signaler. I tillegg er bruken av SDMSH og JANUS bibliotekene med EvoLogics modemene forenklet. Til slutt ønskes disse aspektene å bli integrert sammen for å oppnå optimalisert kommunikasjon mellom forskjellige typer undevannsmodem.

Optimaliseringen av parametre for effektiv undervannskommunikasjon har ikke blitt implementert enda. Imidlertid har det blitt testet at konveks optimalisering fungerer mellom virtuelle modem ved bruk av simulering i UnetStack. Videre har diverse løsninger blitt brukt for å oppnå et modulært og forståelig system, med en mulighet for en videreutvikling som kan tilpasses spesifikke behov.

Contents

List of Figures

List of Tables

List of Abbreviations

ASCII	American Standard Code for Information Interchange
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DC	Direct Current
Gen	Generation
IP	Internet Protocol
IPC	Inter-Process Communication
IPM	Interior Point Method
KKT	Karush–Kuhn–Tucker
MSB	Most Significant Bit
MTU	Maximum Transmission Unit
NR	Newton Raphson
OFDM	Orhtogonal Frequency Division Multiplexing
OS	Operating System
PI	Raspberry Pi
PID	Process identifier
PPID	Parent Process identifier
ROS	Robot Operating System
RX	Receive
SDM	Software Defined Modems
SDMSH	Shell for SDM
SSH	Secure Shell

TCP	Transmission Control Protocol
TX	Transmit
UTF	Unicode Transformation Format
VS Code	Visual Studio Code

Concepts and expressions

Bytecode	A code which an interpreter, like CPython, can read efficiently.
Cargo	Extra data that can be added to a packet.
Colcon	A program building tool made by ROS 2 development team.
Compilation	A computer program which translates human-written code into machine code.
Convolution coding	A type of error correction code, it is used to make a more reliable communication.
Corrupt	Damaged or altered data beyond repair.
CPython	The underlying compiler-interpreter which Python language uses.
Cyclic redundancy check	It is a type of error correction code, it is used to detect accidental changes in data transfer.
Daemon process	A process that continuously runs in the background, will wake up to do a specific task.
Datatype	A certain characteristic for values based on the machine and programming language.
Execution	An initialization of a program.
Float	A number which has a floating point.
Integer	A whole number.
JANUS frame	In this thesis it is used for JANUS baseline packet.
Machine code	Zeroes and ones which only a machine can read and understand.

Modulo
A way to calculate a number's reminder when an integer is divided with another integer.

Network switch
A device to connect several Ethernet devices together. When the switch receives a packet, it will forward it further to the correct device connected to it.

Output stream
Mechanism of sending data from one process to an external location.

Preamble
An introduction statement. In this thesis, a sound sample is used as a reference for JANUS.

Shell
A computer program which interprets commands.

1 Introduction

1.1 Background for Thesis

History

Transmission by underwater sound has been observed since ancient times, with Aristotle noting its existence almost 2000 years ago. Later, Leonardo da Vinci, in the 1400s, further observed the ability to hear ships through long tubes submerged in water, indicating that there was potential for long-distance underwater communication.

The development accelerated during the world wars and the following cold war. Extensive research at the time led to significant advancements in the underwater communication world. One notable invention was the underwater telephone by the United States in 1945 [43].

Motivation

Acoustic communication has experienced significant improvements in modern times as a result of the expansion of commercial interests. The field has had breakthroughs in high-quality video transmissions over large distances and allowed robots to preform challenging tasks. The improvements are continuing to come, and more fields are making a use of acoustic communication. Since underwater schemes have had somewhat new development, there is no telling the potential which may lie within [43].

The underwater realm is gathering increased attention in various industries and academic circles. The biggest challenge subsea devices face may be the use of underwater communication systems (optical and acoustic) with the change in channels over time (turbidity, temperature, etc.). Adapting the exchange of information based on current environmental conditions is an intriguing and crucial challenge for underwater communication [43].

1.2 Research Question

The task given to the team can be specified as follows:
Implement distributed optimization algorithms in ROS 2 based systems, and test them for the case of optimizing the acoustic channel usage in underwater multi-agent scenarios by means of the acoustic EvoLogics S2C and Subnero modems.

This project focuses on making a prototype adaptive communication mechanism to address the challenges in real-life scenarios in underwater communication. Conceptually, the project could be split into three phases:

The first phase of the project is about the understanding of the initially proposed real-time adaptive communication mechanisms. In this phase, distributed optimization algorithms will be used for the development of the optimization program.

The second phase is about the development and implementation of the algorithms into ROS 2, and the development of a communication mechanism for the EvoLogics. The ROS 2 system will be implemented with the Subnero & EvoLogics modems.

The third and last phase is the field test where the underwater devices will be tested in water.

1.3 Previous work

A part of the project is to further develop work that has been done in a previous bachelor's project. The project, "Development of Underwater Communication Rig", had the goal of developing a solution for underwater communication using acoustic modems. The system used ROS 2 (Robot Operating System) for data exchange and communicates using the JANUS protocol. Additionally, the project included setting up the system, implementing the communication protocol, and developing libraries and scripts in C++ for the communication mechanism [53].

At the beginning of this project, their bachelor's thesis, the developed code, a presentation, and other resources were handed over. In addition, a ROS 2 course was added. In the previous thesis, two main problems were mentioned about the handed-over code. The project had issues with transmissions over 100 JANUS packets with the EvoLogics modems and unwanted shell processes.

1.4 Project Plan

The project plan was to get the transmission and reception working on both the Subnero and the Evologics modems, in addition to implementing an optimization algorithm for parameter optimization. Because the Subnero modems have a more user-friendly interface due to the Unet framework, communication is easier to set up. While the communication is being worked out on the Subnero and EvoLogics modems, a parallel plan will go towards looking at the optimization and looking at setting up the ROS 2 structure.

If the EvoLogics communication is set up successfully, a communication test with both the

Subnero and EvoLogics modems will be set up. Further, the optimization algorithm will be tested between two Subnero modems, and possibly with the EvoLogics modem. Ideally, if everything works, a field test will be scheduled so the modems could be tested underwater.

1.5 Structure of the Thesis

This document is structured in the following way:

The theory section consists of explanations of basic concepts surrounding computer science, and necessary mathematical concepts for the optimization algorithm. This section is built from the basics, up to a more detailed theory that is relevant for this thesis.

The method section is split into four parts: **1.** The introduction to the methodology, **2.** the methodology of the optimization development, **3.** the methodology of the ROS 2, Subnero, and optimization code implementation, and lastly **4.** the methodology of the development of EvoLogics communication mechanism.

In the results, the findings from this project are shared and explained. The discussion section contemplates different aspects, personal discoveries, and suggestions for future development. Finally, the conclusion to the report.

The referenced literature and the attachments can be found at the end of this document. Additional attachments will also be provided in a separate zip folder delivered along with this thesis.

2 Theory

2.1 Python and C++

This project utilizes the programming language Python 3.11, C++23, and slightly C. The notable distinction between these languages is the way code is executed. Python is an interpreted language, which means that the code goes through a process that allows direct execution without the need for compilation directly. The Python code is read and executed from a source file with an interpreter/compiler called CPython, which compiles the written Python code into bytecode. Thereafter, it interprets the code when executing [34]. C++ code needs to be compiled into an executable directly, which means that the executable code needs to be turned directly into machine code in order to run [35]. The distinction in the compilation is that CPython compiles the code into a lower level code than the written Python code (bytecode), while C++ uses a compiler to compile to the lowest level possible (machine code). Due to the increased complexity of compilation and syntax in C++, python is more suited for experimentation and continuous development by different development teams. However, C++ is much faster and more efficient due to direct contact with a CPU. This project utilizes C++ where there needs to be C++ due to hardware requirements, while Python is used where it is easier to implement and experiment with.

2.2 Multithreading and Multiprocessing

Multithreading is a way the Central Processing Unit (CPU) can run several tasks concurrently. A CPU operates by calculating and executing machine code provided by the Operating System (OS) and programs; concurrent execution is achieved with multithreading. The CPU creates instances of dedicated (but shared) memory and gives each thread the necessary processing to finish its workload. Each dedicated instance for a program is called a thread [25] [26].

Multiprocessing is a more heavy-duty variant of multithreading. Instead of shared memory, in multiprocessing, there is a completely isolated instance. A feature with multithreading is efficient switching between the thread instances, whereas in multiprocessing intensive tasks are more efficient, but the switching is much slower. The benefit of multiprocessing, albeit more CPU intensive, is that a true parallel execution is possible.

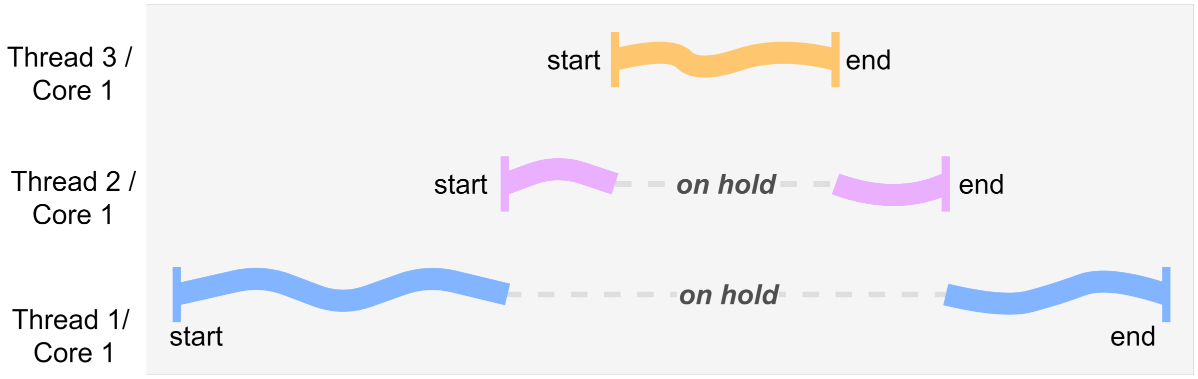


Figure 1: Multithreading; concurrent execution. Threads have issues due to interruption and need priority guidance.

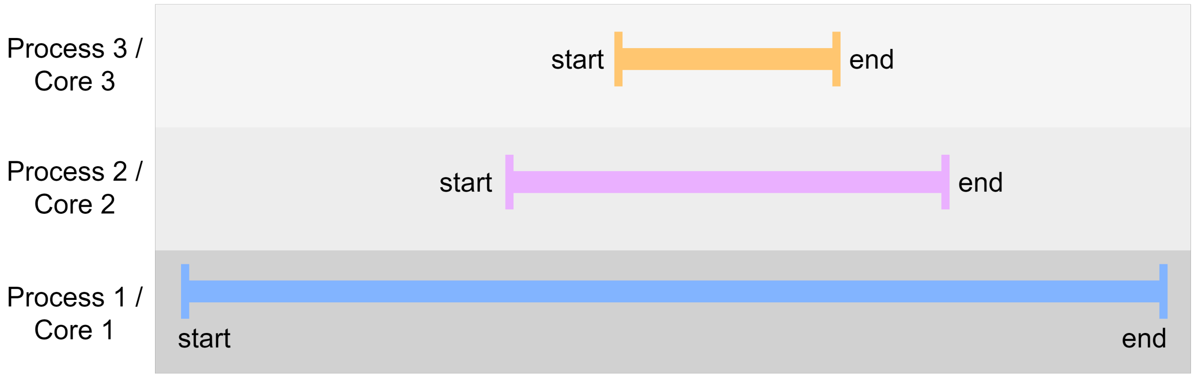


Figure 2: Multiprocessing; parallel processing. No interruption due to isolated processes.

2.2.1 Issues Related to Multithreading and Multiprocessing

2.2.1.1 Race Conditions

Race condition in software is the result of a timing-dependant algorithm that uses different processes or threads at the same time; if the expected sequence of an arbitrary algorithm has process A come first and then process B later, then a break in the consistency of timing, which lets process B come first, locks the algorithm. Process B races ahead of A. This is known as the race condition [32].

2.2.1.2 Deadlocks

Deadlock is a situation where process A is reserving a resource, which other processes like B and C need to continue and process A can not release the resource because it is waiting for B and C to complete their tasks [31].

2.3 Common Multiprocessing Concepts

In this section, specific scenarios related to C and C++ will be discussed. However, many concepts covered here are universal and can be applied to other topics and programming languages.

2.3.1 Inter-Process Communication (IPC)

Inter-Process Communication (IPC) are solutions that enable communication between different processes, ideally without any issues. There are two main types of processes: *independent* processes that do not share data with other processes, and *cooperating* processes that share data with other processes.

The advantages of IPC include information sharing, increased processing time (requires multiprocessing core), multithreading (optimization of resource utilization), and multitasking (handling multiple processes simultaneously).

However, the use of IPC solutions can lead to problems like the "Producer-Consumer Problem", the "Dining Philosophers Problem", the "Readers-Writers Problem", and the "Sleeping Barber Problem".

2.3.2 The Producer-Consumer Problem

When there are two processes, one of which wants to write to a pipe and the other wants to read from the same pipe, where the pipe has limited storage capacity, two rules will apply: If the pipe is empty, the process that wants to read from the pipe must wait. If the pipe is full, the process that wants to write must wait.

If proper handling of read/write operations is not implemented, it can lead to delays where processes must wait. The worst case scenario occurs when a process is stuck in an infinite wait [60].

2.3.3 The Readers Writers Problem

The Readers-Writers Problem is an unwanted process synchronization where multiple processes need to access shared data by utilizing read/write commands. In this scenario, several processes can read from the same data/buffer simultaneously, while only one process can write to that data/buffer at a time. Not handling this correctly can lead to delays, race conditions or corrupted data.

A race condition occurs when a system lacks proper sequence control, resulting in two or more processes attempting to access the same resources simultaneously [60].

2.3.4 pipe()

The `pipe()` function enables communication between processes. In some cases, processes are isolated and have their own private memory spaces, as mentioned in section 2.2. However, a connection can be established between two processes, allowing them to communicate by passing data to a buffer, called a pipe. Every process that has access to a pipe, receives both a read-end and a write-end of the pipe. This means that there is an equal number of read/write ends as the number of processes using the pipe. When a process writes to the pipe, it uses the write end, and when reading, it uses the read end. A notable part when using pipes is handling these read/write-ends correctly. This is done by closing the ends that are not being used or both when the process using it has finished [16].

If the handling of pipes is not implemented properly, it could lead to the problem discussed in section 2.3.2 *"The Produce-Consumer Problem"*.

2.3.5 fork(): Parent and Child Process

The `fork()` function is used to create a new process that is an identical copy of the existing process. This newly created process is referred to as the child process, while the original process becomes the parent process. The child process receives its own unique *PID* (Process ID), and it also has access to the *PPID* (Parent Process ID), which is the ID of its parent process. Another characteristic is that it inherits all the attributes of the parent process, including access to system resources such as `pipes()`. The parent process should always reap every child before terminating itself, which is done by using the commands `wait(0)`.

It is important to note that after the `fork()` operation, any subsequent changes made in the parent process do not affect the child process. The child process operates as an independent entity, maintaining a separate execution path from its parent and having its own memory [72] [67].

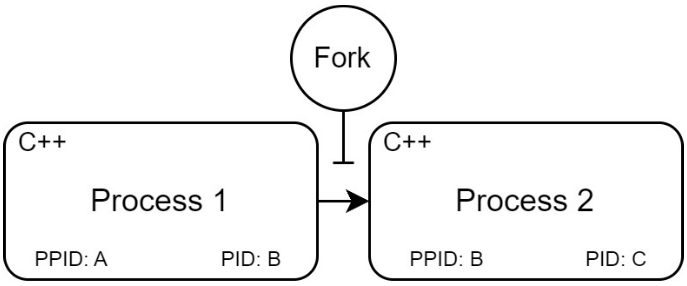


Figure 3: Illustration showcasing the the use of `fork`. Here will Process 1 use the fork and create Process 2. The new process will get a new PID and inherit the PPID equal to the PID of Process 1.

2.3.6 Zombies and Orphans

In multiprocessing, a zombie process is a state of a child process after it finishes its execution, and before it is reaped by the parent process. Until the parent reaps the zombie process, the zombie lingers in an indefinite waiting state. The reaping of the child process is a standard part of the `fork()` cycle.

In a scenario where a parent process terminates itself before a child process finishes execution, the child process becomes an orphan. Orphaned processes are adopted by the system's "init" process, which is responsible for managing the processes of a system. The "init" process assumes the role of the parent process to the orphan, it will be responsible for handling the reaping of the child process when it exits [78].

Problems can arise if the child process is not properly reaped, where it could become a zombie process and indefinitely occupy the system as a slot in the process window.

2.3.7 execvp()

The `exec()` family is a collection of functions that replace the existing process image with a new one. There are different versions of exec() functions and they differ in how they are used. `execvp()` is a variation of exec().

When `execvp()` is used, it replaces the existing process image with a new one. This means that the running process, such as a C++ code, can be changed to a shell process. Certain arguments are also passed to the `execvp()` function, enabling the new shell process to execute specific commands. Note that when adding an argument in the `execvp()` where the function is used to run a shell process, a completely new process might be initialized. Therefore, one could potentially with the use of `execvp()`, swap processing and create a new additional process.

Be aware that when swapping process image, for example, C++ to a shell, all C++ code that comes after the `execvp()` will never run. Another trait is that the shell automatically terminates itself when the commands placed in the `execvp()` finish running. However, if the new process initialized by the commands placed in the `execvp()` does not exist, the shell process will continue to live indefinitely.

It is of significance when distinguishing between `exec()` and `fork()`, and defining the relationship between a child process and a parent process. The new process image created by `execvp()` inherits various properties from its old process image, where the *Process ID* and *Parent process ID* are the most relevant [68].

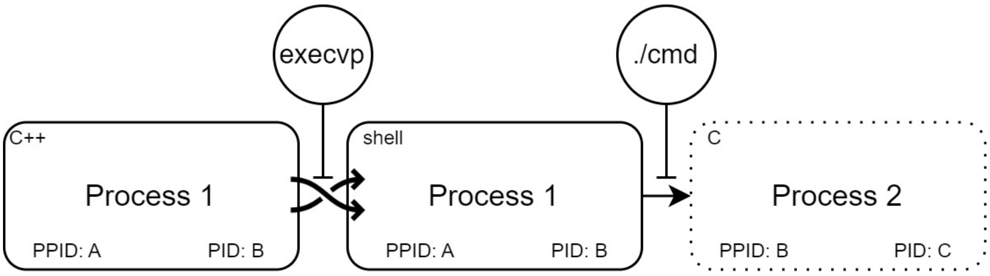


Figure 4: This illustration showcases the use of `execvp`*. When the command is used in "Process 1" it swaps the process image from C++ to a shell. The new process image will inherit both the PID and PPID. If an argument is placed in the* `execvp` *command an additional process, "Process 2", will open.*

2.3.8 popen()

Simplified, `popen()` is a function that uses pipe, fork and exec. It is a solution that is the same as creating a pipe, creating a fork, and then running an exec command. A more detailed description is that `popen()` is a function that can be used to read from or write to a terminal. It will create a stream with a connection to a pipe running the command given in the function parameter[69].

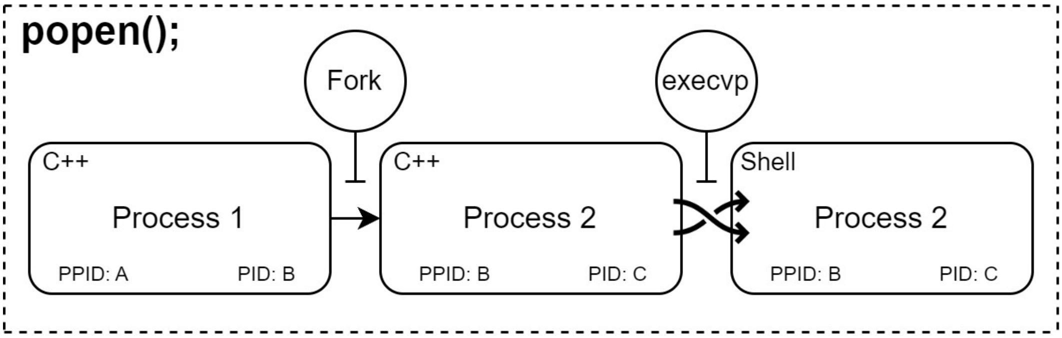


Figure 5: This illustration showcases the use of `pipe`*, where both* `fork` *and an* `exec` *command are used.*

2.3.9 Double Fork to Avoid Zombie Process

Double fork is a technique to disconnect the parent process from its child process. When doing so, the child processes will get a parent that will be an *INIT* process. The *INIT* process is a daemon process and is designed to wait for processes and reap them when they exit [66].

The advantage of using the *double fork* technique is to free the parent process from the responsibilities of reaping its children. Also, it is used to prevent zombies by letting the *INIT* process handle the reaping [71].

2.4 Datatypes

Datatypes are values in computers that have specialized representation, and utility based on the operations that can be performed on them. All written values in a programming language have a datatype, such as an *integer* or a *float*. This project treats specific datatypes with caution as they present issues when used incorrectly. On the other hand, this project has utilized the manipulation of datatypes to its advantage to solve some issues when it comes to the compression of data and their sizes in bits on a machine.

2.4.1 Integers, Unsigned Integers and int8

An *integer* is a discrete number stored in bits. This means that a whole number can be represented using binary code. Due to options for computer optimization, various sizes of integers exist. Although large decimal numbers can be represented by something like an int32 (-2^{31} to $2^{31} - 1$), the storage/memory space which an int32 occupies, has double the amount of bits stored on a machine compared to an int16. Int16 can have a value from −32768 up to 32767, however, one could change the range by using an *unsigned integer*, in this case, an int16 would be an uint16, and have its range be from 0 to 65,535.

The int8 is prevalent throughout this project, due to its tiny size (range of -128 to 127) and the prevalence of *bytes* in computer communication and storage. It is also known as a *char*.

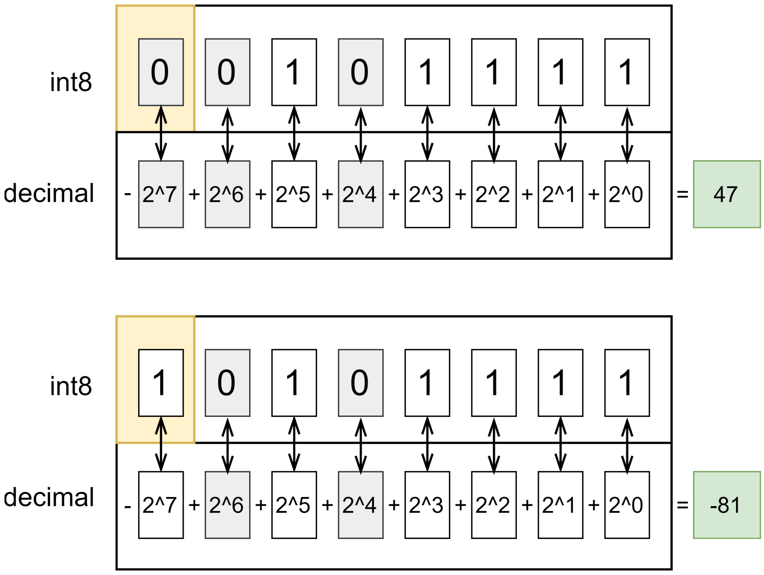


Figure 6: Top image demonstrates how the number 47 is represented by the binary code in int8 `00101111`*. When the MSB is high,* `10101111`*, the signed value of -2^7 is added to the calculation, resulting in* −81.

2.4.2 Floats, Float32 and Float16

Floating-point numbers, like float32 and float16 (half precision), are numbers that represent whole numbers with the addition of a decimal point, meaning numbers like 0.42 are possible to represent. Like int32 and int16, float32 takes up twice as much space as float16.

Although float16 uses half the memory space of float32, its precision is reduced, which leads to imprecise rounding if a desired number is entered. In figure 7, an illustration of how a 16-bit floating-point number of π is represented.

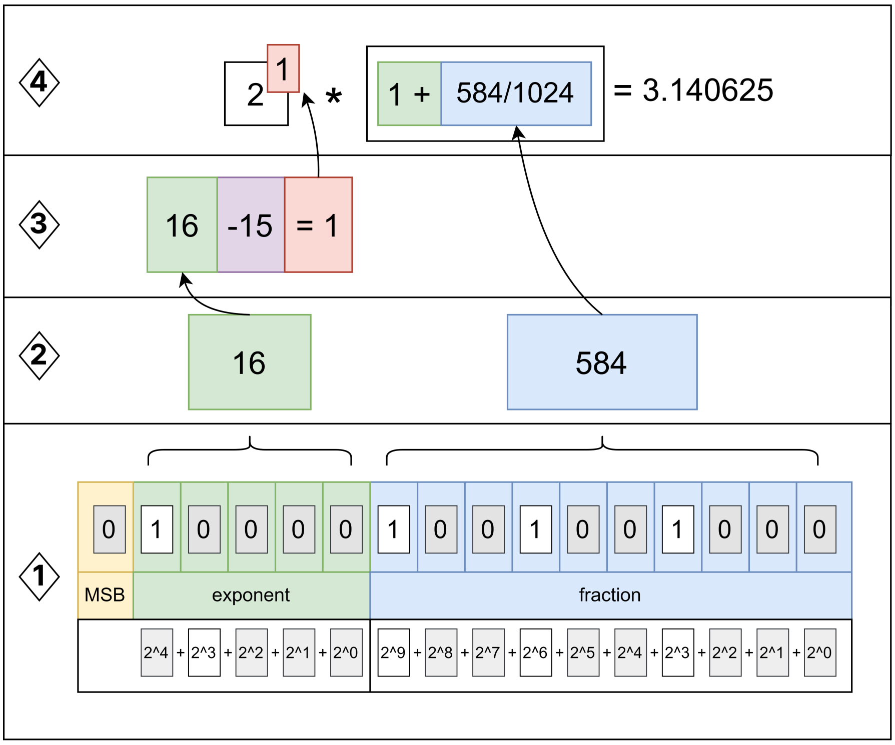


*Figure 7: **1**: float16 binary number split among MSB, the exponent, and the fraction. **2**: The decimal values from the exponent and fraction segments. **3**: Converting bits to decimal using excess − K, where the K for float16 is 15 [22]. **4**: Computing the exponent as 2 raised to the excess-K value, multiplying it with the decimal divided by 1024. Add 1 to the decimal joint, if the exponent bit-sequence has a high bit. In this figure, the number π is approximated through calculation with this formula by hand.*

However, if one were to compute π using float16 arithmetic through a programming language, the result would be rounded up to 3.14, as the precision of float16 is limited [4].

The range of the float16 data type spans from negative infinity (represented as $-\infty$) when the float16 value is set to 1111110000000000, to positive infinity (represented as ∞) when the float16 value is 0111110000000000. The smallest and largest possible numbers within this range are -65,504 and 65,504, respectively.

2.4.3 Characters and Strings

A *character* is a representation of a letter or a symbol, typically encoded using one or more bytes. The specific representation depends on the character encoding, which can vary based on factors such as localization and machine settings [24]. A string is a collection of these characters, which can be either human language or used for storage of data in character representations.

2.4.3.1 ASCII and UTF Encoding

There are many types of encoding, but the most prominent ones are in the American Standard Code for Information Interchange (ASCII) family and in the Unicode Transformation Format (UTF) family. Both ASCII and UTF have a byte encoding format, however, the basic ASCII on its own is limited to 128 different characters due to it only using seven bits. Although there is an extended version of ASCII; due to the way it has been implemented, it depends on the language and localization in order to work correctly, for example, a Nordic extended ASCII is different from an American extended ASCII. On the other hand, UTF is regarded as the universal encoding type; all of ASCII can be found within UTF, but not the other way around. Although the usage of ASCII was common before and can be found as the default encoding in older hardware, UTF is the relatively new standard that can reliably be expected to be consistent across software and hardware [21].

2.5 Networking

In both wired and wireless communication, bytes serve as the fundamental units for storing and transmitting information. Data is encoded by dividing the information into byte sections and subsequently transmitting them. For example, a simple message like `'hello'` consists of five characters, which can be represented using five bytes. It is worth noting that nearly all network communication relies on the use of bytes [23].

2.6 JANUS Communication Protocol

JANUS is an underwater communication protocol, which has become a NATO standard and a leading protocol in underwater communication throughout the globe. It is the first digital underwater communication protocol to be acknowledged at an international level, and it has opened up a standardized "Internet of Underwater Things". The JANUS protocol encodes information into simple acoustic waves, such that existing and new equipment can be used with the standard with ease [44].

2.6.1 ANEP-87

ANEP-87 is a NATO document for the standardization of underwater communication. It is primarily used for underwater communications. It serves roles including tactical operations, safety measures, and distress signal transmission. It also has other areas of use like networks of the nodes and node discovery. The protocol takes advantage of physical layer coding, with JANUS serving as the standard protocol. The ANEP-87 defines the different configurations for a JANUS packet.

A crucial note is that there exists a writing typo/error within ANEP-87 which affects the interleave and deinterleave process. UnetStack has implemented a variant of ANEP-87 where this specific error is present. This means that it will differ from the standard of the JANUS protocol, and any JANUS library will require modifications to communicate with systems using ANEP-87.

2.6.2 Janus-C Version 3.0.5

Janus-c-3.0.5 is a library written in C made to use the JANUS protocol. It is found on JANUS own wiki page. It uses the JANUS standard and not the ANEP-87 standard. The version found here will not work with the ANEP-87 without modifications. Commands for the library can be found in the document "README - JANUS Tool Kit 3.0.1" [56].

The library does not support the use of TCP from the get-go, but there are plugins that enable the use of TCP.

2.6.3 JANUS Packet

A JANUS packet is composed of several audio samples or acoustic waves. Figure 8 illustrates the different parts in a JANUS packet.

- The first part of the packet is a reference file, consisting of a fixed number of audio samples. It makes it possible to identify that it is a JANUS packet. If a receiver gets this reference, it will understand that what follows should be unpacked as a JANUS baseline packet.
- The second part is the baseline packet, which contains different information. Including information about a potential cargo, if there is one. For instance, a receiver can determine what comes after the baseline packet by reading the data in the baseline packet.
- The last part is the cargo. If there is cargo, the baseline packet will instruct how the cargo should be read. For example, *Reservation time* tells how long it will take to read the cargo.

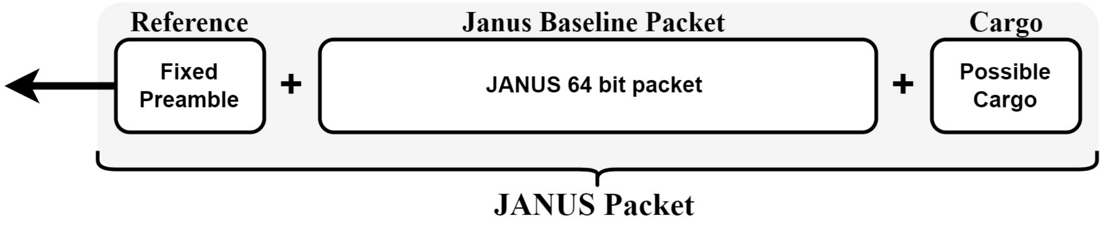


Figure 8: An illustration of a JANUS packet, it will consist of a fixed preamble used as a reference, then the JANUS baseline packet, and lastly optional cargo.

2.6.4 JANUS Baseline Packet

The JANUS baseline packet consists of 64-bit. These bits are used to store information. Each bit is defined from a bit allocation table, made for JANUS. Such a table can be found in ANEP-87 [57, pp. 2-3].

Some bits will be highlighted here:

- **Bit 6, Schedule Flag**: is used to determine how to read bits in the "Application Data Block".
- **Bit 9-16, Class User i.d**: An user ID is mostly used for the identification of nations, but also NATO has its own ID. A lookup table can be found in ANEP-87 [57, pp. A1-A3].
- **Bit 23-56, Application data Block (ADB)**: If *Schedule flag* is set to 1, meaning *ON*, the first bit will determine if *reservation time* or *repeat interval* is used. The time which is used for those two is defined in the next eight bits. Table for *reservation time* can be found in ANEP-87 [57, p. B-1].

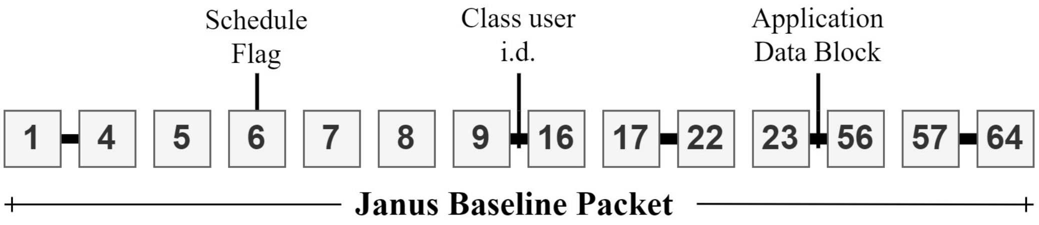


Figure 9: This illustrates the bits in the JANUS baseline packet, where some bits are highlighted.

2.6.5 Transmission Sequence Generation in JANUS

When the JANUS protocol manages information, it wraps the information in something called a *JANUS packet*. Before it can become a packet, the information needs to be handled in a specific order. For transmissions, this sequence is used: CRC, 2:1 convolution encoder and interleave encoding. For receiving, the opposite is used: deinterleave, 2:1 convolution decoder and CRC. In this thesis, it will put emphasis on interleave and deinterleave.

When the transmission sequence generation is done, it will be sent to a waveform generator where settings like bandwidth and sampling frequency are used to create sound samples. This process needs to be done for the JANUS frame and the cargo, separately.

Interleaving

Interleaving uses the convolution coding from the encoder, and intend to prevent a burst error; the error form when consecutive bits are corrupted. Interleaving prevents this by rearranging the bit sequence. By spreading out the corrupted bits, it will still be manageable to decode the packet as long as there are not too many corrupted bits [64].

If a JANUS baseline packet had a length of 144 symbols and the sequence were (b denotes bit):

$$\text{b0 b1 b2 ... b143} \tag{1}$$

Then this sequence interleaved in the JANUS library will have the order:

$$\text{b0 b13 b26 ... b143 b12 b25...b142 b11 b24 ...} \tag{2}$$

The order of the permutation table is created by using this code:

```
perm[i] = (perm[i-1] + prime) % packet length        (3)
```

The `perm` variable is the permutation table and decides the sequence the different bits are placed. It works as a table with a number of elements equal to the length of the packet. The element position in the permutation table is decided using a variable `i`. The `prime` variable from the code is the lowest prime number that is not a factor of the length of the packet. In addition, the `prime` must be greater than the square of the length of the packet. For the JANUS baseline packet in this example, with length 144, the `prime` would be equal to 13. When calculating the permutation element, the previous element in the permutation element will be added with the `prime`. The first element in the permutation table is defined as zero. The modulo of the sum and the length of the packet is then calculated. The reminder from the modulo calculation will be added as an element in the permutation table.

When the permutation table is built, which is when all the 144 elements in the table are assigned a number, the permutation will be performed. This is done by placing the bits in the sequence in the permutation table. See figure 10 for an example of the interleave process with the sequence: 0 1 2 ... 7.

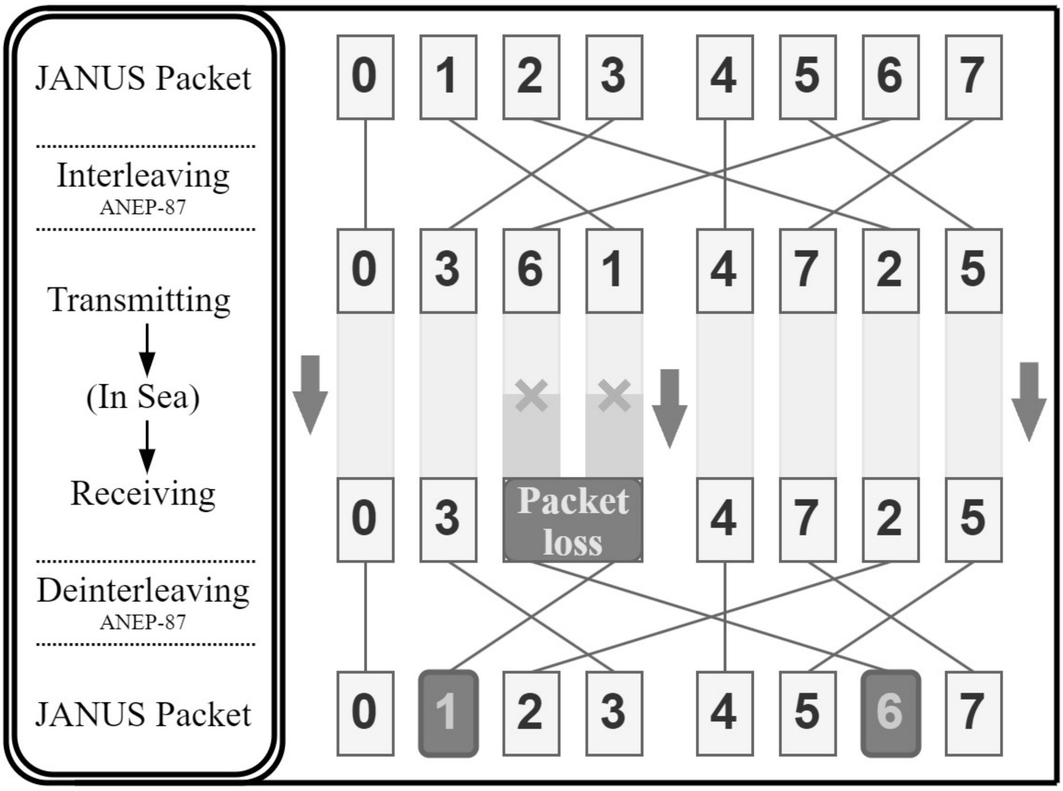


Figure 10: This illustration shows how interleaving deals with a burst error. The sequence is first interleaved. In the transmission, then a burst error occurs. When the packet is deinterleaved after transmission, the packet will go back to its original order. The corrupted bits are spread out and the packet can be decoded.

ANEP-87 contains an error that makes the interleave order different from the order mentioned in sequence 2. This is also the implementation used by UnetStack. If the sequence were:

$$\text{b0 b1 b2 ... b143} \tag{4}$$

Then the interleaved sequence from the ANEP-87 would be [57]:

$$\text{b0 b13 b26 ... b143 b1 b14 ... b131 b2 b15 ...} \tag{5}$$

The difference between the two interleaved sequences is what happens when the

$$\text{perm[i-1] + prime >= packet length} \tag{6}$$

In sequence 5, the modulo will not be calculated, but will instead start on the lowest element position available.

Interleaving will be done for the JANUS baseline packet and the cargo separately. This is so the baseline packet can first be decoded to know the number of bytes in the cargo [57]. When the packet is received, deinterleaving will be performed to get the same sequence that was before interleaving.

2.7 Transmission Control Protocol

TCP stands for *Transmission Control Protocol* and is one of the most used Internet Protocols. It determines how computers transmit packets of data. TCP uses acknowledges, which guarantees data sent from one computer to the other is received in the correct order and accurately. This means the TCP protocol essentially provides an error-free data transmission [58].

2.8 Underwater Modems

Wireless underwater communication is challenged by the properties of water; high attenuation and limited bandwidth [18]. To overcome these obstacles, acoustic waves are utilized as a reliable form of data communication, replacing unreliable radio technology underwater. Sending acoustic waves in the air, depending on pressure and humidity, may only reach about 331 meters/second. However, sending acoustic waves underwater is much faster at 1540 meters/second [19].

2.8.1 Optimization of The Underwater Modem Parameters

For optimal operation of modem communication, communication variables need to be changed regularly. Due to different ranges between modems, different water pressure and salinity, an adaptable modem can make sure that the best way to communicate is achieved. In this project, the variables being optimized and changed regularly are the subcarriers **N**, modulation order **M** and symbols per packet **m**.

2.9 Unet, UnetStack

Unet is an underwater communications framework developed by the National University of Singapore. The goal of Unet is to provide a good and simple service for underwater communication due to the challenges underwater communication has. The *UnetStack API* is available in several programming languages, including Python. UnetStack is the collection of technologies and software components that complement the Unet framework. It includes various modules, libraries, and tools that extend the capabilities of Unet for different projects and applications [9].

2.10 Subnero WNC-M25MRS3

Figure 11: The Subnero WNC-M25MRS3 modem [40].

Subnero WNC-M25MRS3 is an underwater modem designed for academic research and enthusiasts. The modem is integrated with the Unet framework, which allows for a user-friendly interaction with the modem by providing an interface on a web browser and direct access to the shell. Subnero modems have an operating range of 3-5km and a max depth of 300m [17].

2.11 EvoLogics S2C R 18/34 USBL

EvoLogics S2C R 18/34 is an underwater acoustic modem made for industry and science alike. It provides full-duplex digital communication, which makes it possible to have simultaneous transmission between modems. The modem has an operational range of 3500m and can reach a max depth of 2000m [14].

Figure 12: EvoLogics S2C R 18/34 USBL modem [39].

2.11.1 SDMSH

SDMSH or *SDM Shell* is a shell-based library developed by EvoLogics for their underwater modems. It provides an easy-to-use interface for accessing and configuring the modems. It enables the user to configure the API for the physical layer of the modems, also called PHY.

The SDMSH aims to simplify the process of sending and receiving data through the modems. This is done by providing simple commands such as `rx` for reception and `tx` for transmission. The `ref` command is used to set a reference on the modems, indicating when to initiate the data reception. The `config` command makes it possible to set the modem parameters before reception and transmission. It can be used for changing the threshold, gain, source level and pre-amplifier gain of the modem. Where the threshold is used to detect the reference or preamble and the source level sets the strength at which the message is transmitted. The content is displayed within listing 1:

```
config <threshold> <gain> <source level> [<preamp_gain>]
```

Listing 1: The content within the config command.

In table 1, the different source levels possible is listed, where "3" is the lowest source value and highly recommended when testing in air:

Value	Source level
0	max, 0 dB
1	75%, -6dB
2	50%, -12 dB
3	min, -20 dB

Table 1: This table shows the different source levels for Evologic modems S2C. In the SDMSH command: 'config 30 0 X 0' where X is the placement of the "value" [51].

Description for different commands used in the SDMSH library can be found on their wiki page [51].

2.11.2 EvoLogics AMA

The *AMA* software provided by EvoLogics has an interface that is easy to use for testing the EvoLogics S2C modems. The software is free to download and can be found on the EvoLogics website [59].

2.11.3 AT-Commands for Evologics Modem

AT-commands are commands that can be written to the EvoLogics modems. It makes it possible to get information, set parameters or change configurations on the modems. Netcat is used to connect to the modems to access the AT-commands. A command can be written by adding +++ before the command. These commands can be found in the modem's manual [55].

2.12 Raspberry Pi 4 Model B

The *Raspberry Pi* is a single-board computer which is developed by Raspberry Pi Foundation in association with Broadcom. The advantages are the small size, low cost and open design. The Raspberry Pi has GPIO (General-Purpose Input Output) pins, which makes it so that it can be connected to other circuits [46]. The Raspberry Pi computer has the goal of being the target hardware for most developments for this project.

Raspberry Pi 4 Model B features a quad-CPU, a larger memory, and two micro-HDMI ports port to connect to a screen. 4GB of RAM [79]

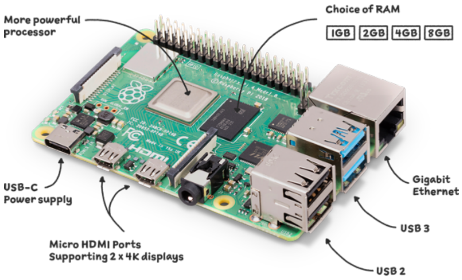


Figure 13: Raspberry Pi 4 Model B [41].

2.13 Visual Studio Code

Visual Studio Code (VS Code) is a lightweight source code editing and development program with a user friendly interface. It is available for Windows and Linux machines, and supports many different programming languages [20]. This project utilizes this editing software as its main coding tool.

2.14 ROS 2

Robot Operating System (ROS) is an open-source development kit for robotics applications. ROS provides an environment for rapid research and development of robotics software, and interchangeable understanding by those who are familiar with ROS and those who are learning about it. Rather than creating your own environments where you would need to handle threading issues, logic, and documentation, ROS 2 provides a shortcut for that; with an addition of already optimized packages and libraries for Python and C++. ROS 2 is the successor to ROS and is available for Python and C++ [1]. This project uses Humble Hawksbill version of ROS 2 distribution. It was released May 23rd 2022 and has an end-of-life date May 2027 [27].

2.14.1 ROS 2 and Multithreading

An issue with multithreading is the increased complexity of a program, which is prone to errors like race conditions and deadlocks 2.2.1. Maintaining or changing a complicated program can take a lot of resources. With ROS, the necessary work has already been done and can be used as a tool for the development of autonomous systems.

2.14.2 Nodes, Topics, Publishers and Subscribers

ROS 2 has several methods for sharing data between nodes. A node in ROS has a responsibility to handle and do something with data that it acquires through several methods: topics, services, actions and parameters. This project used topics as the solution to the data flow and processing.

- **ROS 2 Nodes** represent software modules; these nodes are modules that contain tasks that one wishes to execute. They hold the sub-modules mentioned underneath.
- **Topics** are shared memory spaces for nodes. They hold the information one wishes to grab/share and use in a node.
- **Publishers** are modules that ROS 2 uses to "publish" data to a topic. The datatype must be declared beforehand, and can only hold that datatype at all times.
- **Subscribers** get data from declared topics. Subscribers and publishers are similar in code, but they execute different tasks.

  Due to the multithreading capabilities of ROS 2, each node can work concurrently and create a modular system that can ease the work needed to achieve relatively complicated autonomous systems.

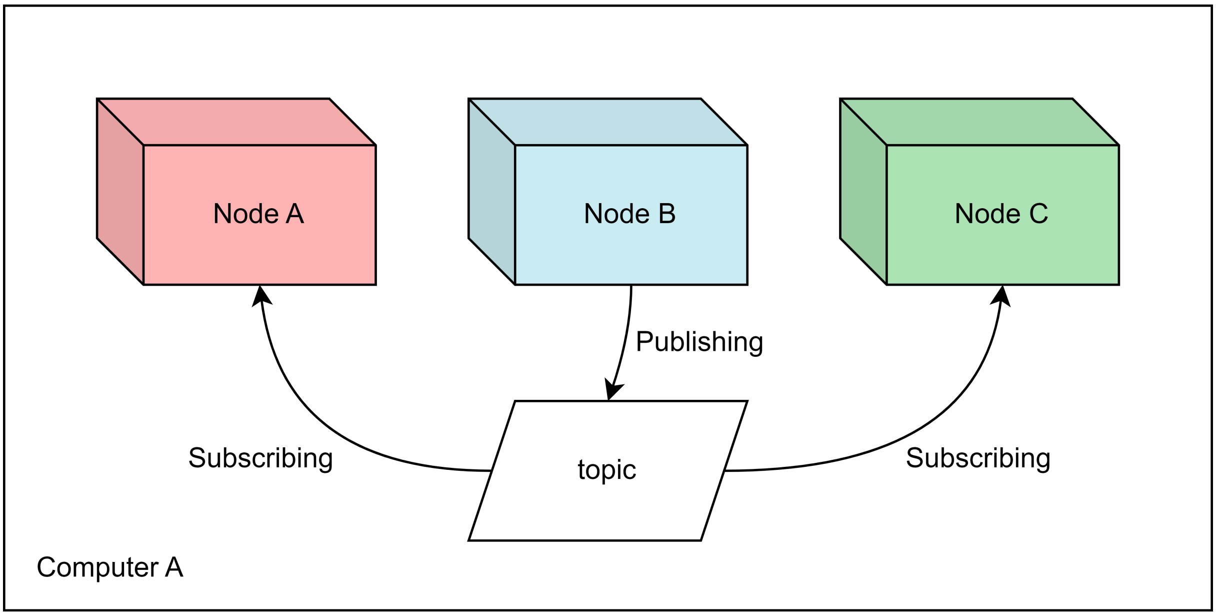


Figure 14: Three nodes sharing a 'topic' within a computer system. Node B is publishing to a topic, while node A and node C are subscribed to the same topic. Node A and C hold the subscriber module, while node B holds the publisher module; it is not an exclusive configuration and a mix of both is achievable, and there can be any amount of nodes as necessary.

2.15 Optimization Theory

The optimization problem comes down to one problem. Finding the x that minimizes the cost function, $f(x)$:

$$x^* := \underset{x}{\operatorname{argmin}} f(x) = \underset{x}{\operatorname{argmin}} \sum_{i=1}^{N} f_i(x) \tag{7}$$

The optimization algorithm can be split into three building blocks [36]. To understand these, some concepts are necessary to go through first. The main focus for the optimization in this project will be on the Subnero modems using Orhtogonal Frequency Division Multiplexing (OFDM), which is a particular method for transmitting data. However, the end goal is to include the optimization algorithm for both the Evologics and Subnero modems.

2.15.1 The Cost Function - Generalized

The *cost function* is a function one wants to minimize to get the best results. For our project, if the cost function is minimized, it will as a result maximize the data rate. A simpler example of a cost function is when you are doing linear regression analysis. In this instance, you have a function which is the measure of all the distances of your model compared to the data points. To get the best possible line, you want to minimize these distances, hence you want to minimize the cost function [30].

2.15.2 Convexity - Assumption I

There are two assumptions for the optimization algorithm to work. The first assumption is that the cost functions used are *convex*. A function is mathematically called convex if any pair of two distinct points $(x, f(x))$ and $(y, f(y))$, when making a connecting line, remains above the graph between these two points (see figure 15) [38]. A less mathematical explanation is: if a function is convex, it has a cup shape $\cup$. Further, the cost function has to be strongly convex, which means that its Hessian is bounded from below, $\nabla^2 f_i(x) > cI_n$ for all x, with $c > 0$. This makes it so the cost function has a unique minimum point [36].

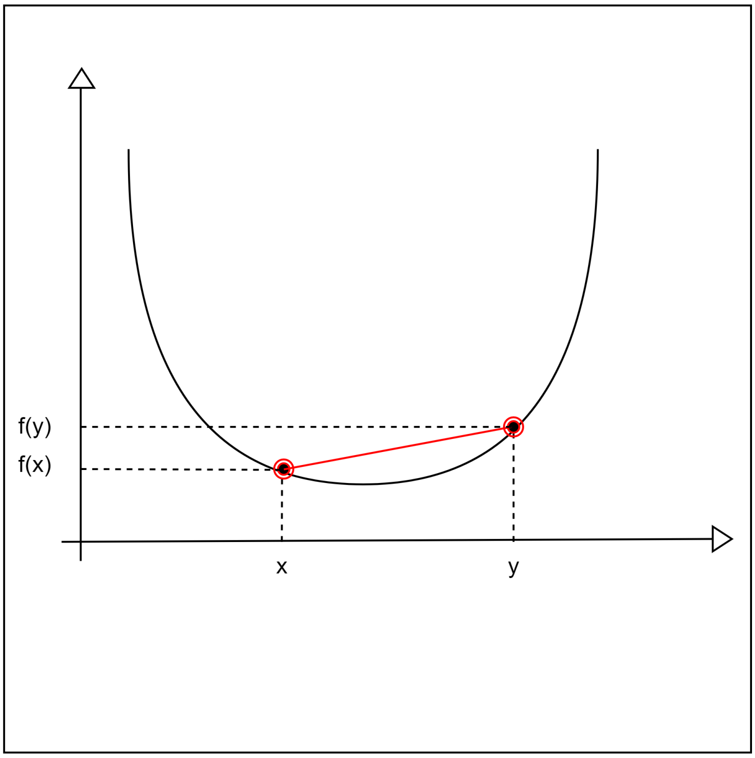


Figure 15: Example of a convex function. The two points $(x, f(x))$ and $(y, f(y))$ make a line above the graph.

2.15.3 The Cost Function - This Project

The derivation of the cost function and the different variables will not be covered in full detail because it is not this project's priority. A list of all the variable glossaries can be found in table 2. Full details on derivation can be found at *"Multi-agent algorithms for adaptation of underwater acoustic communication parameters"* by Iadarola, F. [33].

The objective of our optimization is to maximize the data rate over one OFDM packet. The function of the packet data rate, R_p, is a function one wants to maximize. By making use of that the maximizer of $\log R_p$ is equivalent to finding the maximizer of R_p, and that one can turn this into a minimizer problem by transforming $\log R_p$ into $(-\log R_p)$, the cost function for the packet data rate can be expressed as follows [33]:

$$\begin{aligned} f(x) &= -\log R_p \\ &= -(\log m + logR_c + \log B + \log R_n + \log N + \log(\log_2 M) - \log(m(1 + p_c)N + B(t_{oh} + t_d))) \end{aligned} \tag{8}$$

Variable	Meaning	Type
m	Symbols per packet	Optimizable
N	Subcarriers	Optimizable
M	Modulation order	Optimizable
p_c	Cyclic prefix fraction	Estimated
ν	Doppler spread	Estimated
r	Range	Estimated
t_{oh}	Overhead (preamble, propagation...)	Estimated
$\bar{\gamma}$	Average SNR at the receiver	Estimated, $0 < \bar{\gamma}_{dB} < 30$
R_n	Proportion of the carrier waves	Estimated, $0 < R_n < 1$
B	Bandwidth	Known
c	Speed of sound	Known
k	Relative doppler margin	Design
p_l^t	Target packet loss ratio	Design
n_x	Non-data subcarriers	Design
R_c	Coding rate	Design
t_d	Transmission delay	Calculated, $t_d = r/c$
R_p	Packet data rate	Calculated, $f(x) = -\log R_p$

Table 2: Glossary of variables in the protocol.

The values N, M and m ($x \in \mathbb{R}^3$) are the variables that are going to be optimized.

$$x = \begin{bmatrix} N \\ M \\ m \end{bmatrix}$$

The cost function also contains some constraints [33]:

$$f(x), \quad \text{subject to} \begin{cases} N \geq n_x + 1 \\ M \geq 2 \\ m \geq 1 \\ N \leq \frac{B}{k\nu} \\ \log m + \log R_n + \log N + \log(\log_2 M) + \frac{1}{R_c} \cdot (\log 0.2 - \frac{3\bar{\gamma}}{2(M-1)}) \leq \log p_l^t \end{cases}$$

Such that the algorithm stays within realistic values, these constraints are going to be applied:

$$f(x), \quad \text{subject to} \begin{cases} 400 < N < 2000 \\ 2 < M < 64 \\ 1 < m < 40 \\ \log m + \log R_n + \log N + \log(\log_2 M) + \frac{1}{R_c} \cdot (\log 0.2 - \frac{3\bar{\gamma}}{2(M-1)}) \leq \log p_l^t \end{cases}$$

2.15.4 Distributed and Centralized Systems

It is important to distinguish between a *distributed* and a *centralized* system. In a centralized system all the data is stored in a single computer. To access the information one has to access the main computer, often called "the server". However, this project works with a distributed network. In a distributed system the computations are distributed across multiple, separate computation nodes (mobile, computers, etc.) and the system no longer has one main computer. In this project, each node will figure out the best values for the cost function on its own and then share that information only with its nearby nodes. From the available information they have themselves and information coming from their neighbors, they will make appropriate changes using a distributed algorithm. With this distributed algorithm they will all come to a consensus (agreement), which means they will all eventually have the same values for the variables being optimized [28].

2.15.5 Network Connectivity - Assumption II

The other assumption for our optimizing algorithm has something to do with how the network is connected. Formally, the communication graph is represented as G = (V, E) where V = {1,...,N}. V is short for vertices, also known as nodes. E is short for edges. Edges are the connection between node i and node j written formally as E ⊆ V × V so that $(i, j) \in E$ if and only if node j can directly receive information from node i. Edges are represented as lines that connect nodes in a graph. A graph has arrows as indications if the connection is directional and only indicated as lines if the connection is bidirectional such as in figure 16.

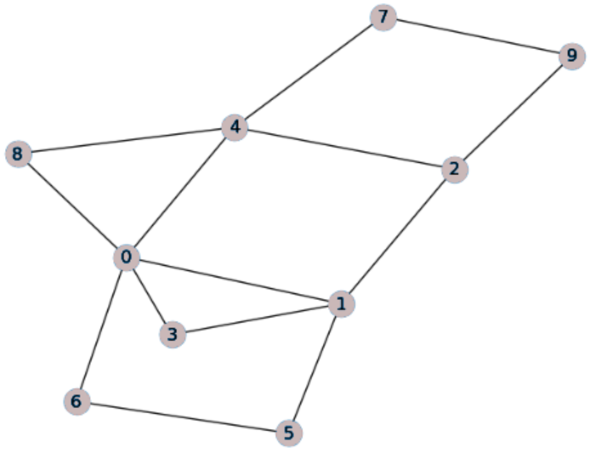


Figure 16: An example of a bidirectional graph with 10 nodes.

Neighbor nodes are nodes that are directly connected to each other. For example in figure 16, the neighbors of node 6 are node 0 and 5. More formally with the directions in mind, in-neighbors of node i are denoted as $\mathcal{N}_i^{in} := \{j \in V \mid \ (j, i) \in E, \quad i \neq j\}$, which are the set of nodes connected to node i that can send messages to node i. Out-neighbors of node i are denoted as $\mathcal{N}_i^{out} := \{j \in V \mid (i, j) \in E, \quad i \neq j\}$, which are the set of nodes connected to node i, receiving messages from node i. The number of neighbor nodes in the sets is indicated with $|\mathcal{N}_i^{out}|$ and $|\mathcal{N}_i^{in}|$, respectively [36].

2.15.6 Building Block A - NR Consensus

The *Newton-Raphson consensus* builds on Newton's method in optimization. Newton's method in optimization, also called the Newton-Raphson method, is an iterative method for finding the roots of a function. In other words, it finds the x that gives either the minimum or maximum of a function $f(x)$. In this project, its purpose is for finding the minimum point of the cost function. Newton's method in optimization is defined as follows [36]:

$$x_{k+1} = x_k + \varepsilon d_k = x_k - \varepsilon \frac{\nabla f(x_k)}{\nabla^2 f(x_k)} \tag{9}$$

For k = 0,1,2,... and where d_k acts as the direction, ε is the stepsize, $\nabla f(x_k)$ is the gradient and $\nabla^2 f(x_k)$ the hessian.

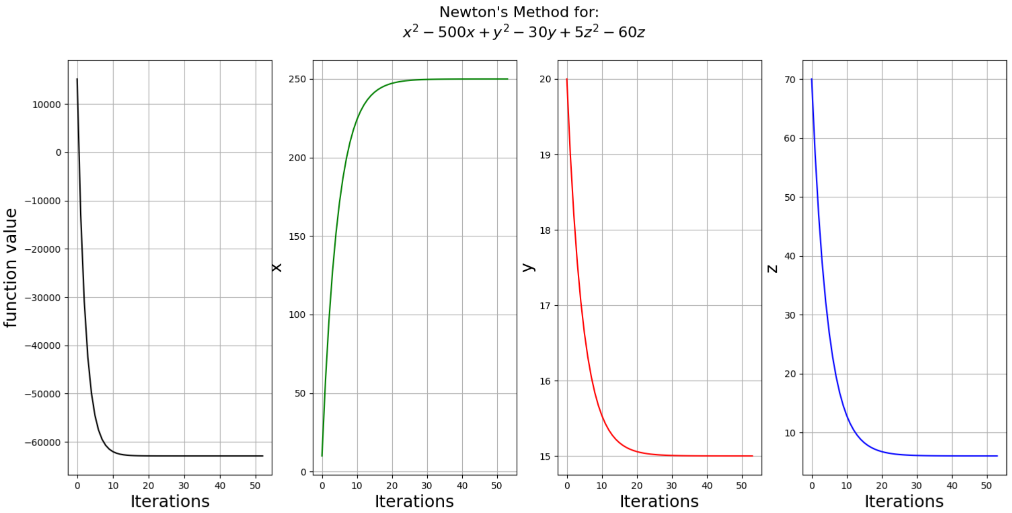


Figure 17: Newton's method used on a function with variables x, y and z. ε is set equal to 0.1 for smooth steps.

With some algebra, (9) can be rewritten as:

$$x_{k+1} = (1-\varepsilon)x_k + \varepsilon(\nabla^2 f(x_k))^{-1}(\nabla^2 f(x_k)x_k - \nabla f(x_k)) \tag{10}$$

Note that depending on the programming language, it can make a small difference when calculating (9) and (10). Since we want the x that minimizes some function with all the nodes in mind, the function is set equal to the sums of all the local cost functions. (10) can be rewritten as:

$$x_i^+ = (1-\varepsilon)x_i + \varepsilon(\sum_j \nabla^2 f_j(x_j))^{-1}(\sum_j(\nabla^2 f_j(x_j)x_j - \nabla f_j(x_j))) \tag{11}$$

With the notation $x_i^+ = x_{k+1}$ and $x_i = x_k$.

It is important to note this is for a standard centralized scenario and requires a main computer that knows all the cost functions of all the nodes. However, the project works with a distributed system and there is no way for each agent to compute the two sums instantaneously. The push-sum consensus will help to solve this problem.

2.15.7 Building Block B - Push-Sum Consensus

To use Newton's method in a distributed system, the *push-sum consensus* shall be used. The push-sum consensus eventually allows every node in the network to come to a consensus (agreement) for a variable. From the problems in (11), the push-sum consensus will be used twice on two variables so the following can be achieved [36]:

$$\begin{aligned} y_i &\longrightarrow \eta_i \sum_j \left(\nabla^2 f_j(x_j)x_j - \nabla f_j(x_j)\right) \\ z_i &\longrightarrow \eta_i \sum_j \left(\nabla^2 f_j(x_j)\right) \end{aligned} \tag{12}$$

Where η_i are possibly time-dependent nonzero scalars.

The push-sum consensus algorithm solves (12) in an asynchronous communication scenario. It is an algorithm that builds further on the classical consensus algorithm. Before understanding the consensus algorithm, one should know what a weight matrix is.

2.15.7.1 Weight Matrix

A *weight matrix*, also called *the weighted adjacency matrix*, is a matrix that represents the strength of the connectivity between the nodes of a system. A connection between node i to node j is represented as a number $w_{ij} > 0$ in the weight matrix. That means if one knows the weight matrix of the graph, one can draw the entire network and know how connected all the nodes are to each other. Further, a doubly-stochastic matrix is a matrix where all columns sum up to 1 and all the rows sum up to 1. This can be seen in figure 18 [42].

$$\begin{bmatrix} 0.632 & 0 & 0.368 & 0 \\ 0 & 0.632 & 0 & 0.368 \\ 0.368 & 0 & 0.356 & 0.276 \\ 0 & 0.368 & 0.276 & 0.356 \end{bmatrix}$$

Figure 18: Example of a doubly-stochastic weight matrix W with nodes {0...3}. Here w_{02} *is equal to 0.368 which represents the connection from node 0 to node 2.*

2.15.7.2 Classical Consensus Algorithm

The classical consensus algorithm is as follows:

$$x_i^{k+1} = \sum_{j=1}^{N} w_{ij} x_j^k \tag{13}$$

Where x_i^k is the variables for node $\{1, 2, \ldots, N\}$ that one wants to come to a consensus. For the consensus algorithm, the weight matrix should be doubly-stochastic in order to converge to the average of the local initial conditions. Otherwise, convergence is not guaranteed to the desired point [29]. This means that if every node in a network runs this classical consensus algorithm, and the weight matrix is doubly-stochastic, they will all eventually converge to the average of all their starting values of x_i.

2.15.7.3 Push-sum Consensus

Building on the classical consensus algorithm, we get the push-sum consensus. This algorithm works exactly like the classical consensus algorithm, but instead of the weight matrix needing to be doubly-stochastic, it only needs to be column-stochastic. This is more convenient but may require extra computation.

The push-sum consensus algorithm is as follows:

$$\begin{aligned} x_i^{k+1} &= \sum_{j=1}^{N} w_{ij} x_j^k \\ q_i^{k+1} &= \sum_{j=1}^{N} w_{ij} q_j^k \\ r_i^{k+1} &= \frac{x_i^{k+1}}{q_i^{k+1}} \end{aligned} \tag{14}$$

The first part is exactly the same as the consensus algorithm. However, there is another consensus on the value q_i^{k+1} which is, roughly speaking, a value on how much node i has communicated with the other nodes. If the initial values of q_i are 1 and we take the ratio of these two consensuses, we get the values r_i^{k+1} that acts as the average of the local initial conditions just like the normal consensus algorithm [29].

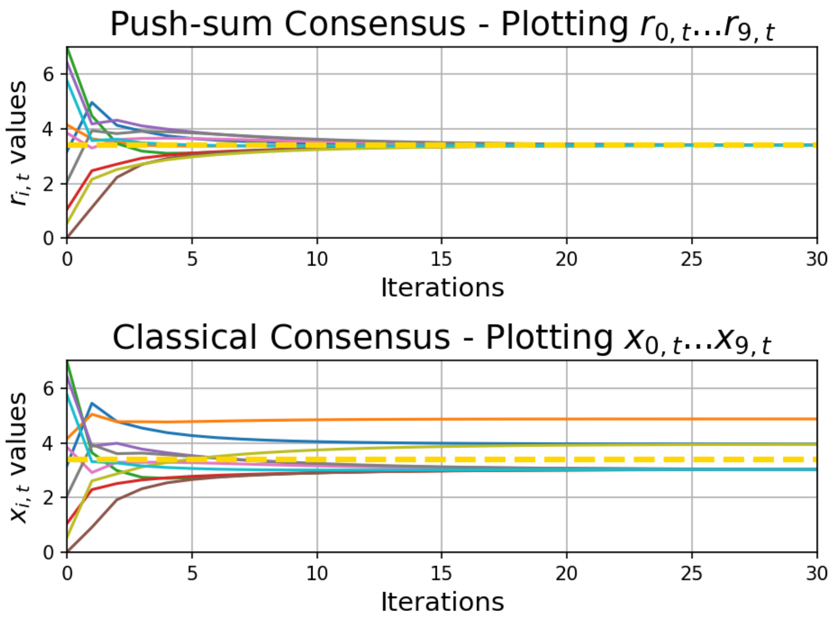


Figure 19: Push-sum consensus versus normal consensus used on a simulated ten node system with a column-stochastic weight matrix. Notice normal consensus breaks since the weight matrix is not doubly-stochastic.

In this project, the nodes send equal weights distributed to the neighboring nodes and also send the same weight to themselves. The algorithm uses the push-sum consensus on both the sum of the hessians (z_i) and what is essentially the sum of the gradients (y_i). Since Newton's method take the ratio between these two values (11) and equal weight is distributed, q_i^{k+1} can be simplified away and there is no need to take the ratio between x_i and q_i like in the push-sum consensus(14):

$$r_1^{-1} r_2 = \frac{\frac{x_2}{q}}{\frac{x_1}{q}} = \frac{x_2}{x_1} = x_1^{-1} x_2 \tag{15}$$

With this in mind, the local updates of the push-sum consensus under synchronous communication, with initialization $y_i(0)$, for each $i \in$ V are as follows [36]:

$$y_i^+ = \frac{1}{|\mathcal{N}_i^{out}| + 1} y_i + \sum_{j \in \mathcal{N}_i^{in}} \frac{1}{|\mathcal{N}_j^{out}| + 1} y_j \tag{16}$$

2.15.7.4 The Rule Update

The push-sum consensus can be extended further to an asynchronous implementation with the rule update (the structure is equal for z_i and z_j) [36]:

$$y_i^+ = \frac{1}{|\mathcal{N}_i^{out}| + 1} y_i \tag{17}$$

$$y_j^+ = y_j + \frac{1}{|\mathcal{N}_i^{out}| + 1} y_i = y_j + y_i^+, \quad \forall j \in \mathcal{N}_i^{out} \tag{18}$$

The rule update, (17) and (18), shows how the transmitter node i transmits and how the receiving node j updates its local variables.

2.15.8 Building Block C - Robust Ratio Consensus

Even though the push-sum consensus can be implemented for asynchronous communication, it loses its convergence properties in the case of lossy communication. For the consensus to go to the desired value, the masses have to be preserved, even in the presence of lost packets. Therefore, there is a need for another building block that handles packet loss in the push-sum consensus.

Robust ratio consensus adds variables $\sigma_{i,y}$ and $\rho_{i,y}^{(j)}$, which act as "mass counters" that keep track of the total mass sent and received. The mass counter variables accumulate for the mass lost, and hence, the mass is still preserved. The same structure is applied to z_i.

$\sigma_{i,y}$ is defined as follows [36]:

$$\sigma_{i,y}^+ = \sigma_{i,y} + y_i, \quad \forall i \in V \tag{19}$$

$\sigma_{i,y}(k)$ acts as a measurement on how much y-mass node i has sent to its neighbors. Moreover, $\rho_{i,y}^{(j)}(k)$ is a measure on the total y-mass node i has received from neighbor node j. They are both initialized to zero.

$\rho_{i,y}^{(j)}$ is defined as [36]:

$$\rho_{i,y}^{(j)+} = \begin{cases} \sigma_{j,y}, & \text{if } \sigma_{j,y} \text{ is received} \\ \rho_{i,y}^{(j)}, & \text{otherwise} \end{cases} \tag{20}$$

With the robust ratio consensus, the synchronous push-sum consensus (16) can be transformed into a robust consensus and rewritten as:

$$y_i^+ = \frac{1}{|\mathcal{N}_i^{out}| + 1} y_i + \sum_{j \in \mathcal{N}_j^{in}} (\rho_{i,y}^{(j)+} - \rho_{i,y}^{(j)}) \tag{21}$$

One can imagine there exists some fictitious virtual mass on the edge (j, i) ∈ E that is defined as

$v_{i,y}^{(j)} = \sigma_{j,y} - \rho_{i,y}^{(j)}$. Under a successful transmission, this value will be zero. However, if there is packet loss, this variable will accumulate for the additional mass node j wants to send to node i in the next successful transmission [36].

2.15.9 ra-NRC - Result of Building Blocks

Combining the robust ratio consensus with the update rule from the push-sum consensus, one gets a robust asynchronous push-sum consensus. Combining this again with the Newton-Raphson consensus, a robust and asynchronous Newton-Raphson consensus (ra-NRC) is achieved. See Attachment A8.

2.15.10 Constrained Optimization

Since this project's cost function (8) contains logarithmic elements which are unbounded above/-below, the cost function will diverge as an unconstrained problem. Therefore, constraints are needed and the original problem (7) is reformulated as:

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{s.t.} \quad & g_i(x) \leq 0, \quad i = 1, ..., m \end{aligned} \tag{22}$$

where $f(x)$ is the cost function and $g_i(x)$ are the constraints.

For unconstrained optimization problems, $\nabla f(x) = 0$ guarantees that x is an optimal solution. However, this implication is not quite true for problems that have constraints. Constrained optimization problems need some additional assumptions called the Karush–Kuhn–Tucker (KKT) conditions. The method used to solve inequalities in this project is derived from these KKT conditions. For simplicity's sake, these conditions will not be covered in detail. More details can be found at *"Convex Optimization"* by Boyd, S. P. & Vandenberghe, L. [30].

The general idea of how inequality constraints are handled is that the constraints are added directly to the cost function as their own part of the function. These constraint functions will increase vastly in value if a constraint is broken, such that the minimizing algorithm will know that this is not the way to the optimal solution. With this in mind, problem (22) can be rewritten as:

$$\text{minimize} \quad f(x) + \sum_{i=1}^{m} I\_(g_i(x)) \tag{23}$$

where

$$I\_(u) = \begin{cases} 0 & u \leq 0 \\ \infty & u > 0. \end{cases}$$

The problem with this approach is that $I\_(u)$ is discontinuous at $u = 0$. At this point, the derivative does not exist and this is a problem for the optimization algorithm. This is why an approximation of $I\_(u)$ is needed [30, pp. 562-563].

The term "objective function" will be used on the whole expression which is being minimized. For example, in equation (23), the objective function is the whole expression $f(x) + \sum_{i=1}^{m} I\_(g_i(x))$.

2.15.10.1 Logarithmic Barrier Function

$I\_(u)$ is approximated with a logarithmic barrier function (see figure 20). The approximation is as follows:

$$\hat{I}\_(u) = -\frac{1}{t}\log(-u) \tag{24}$$

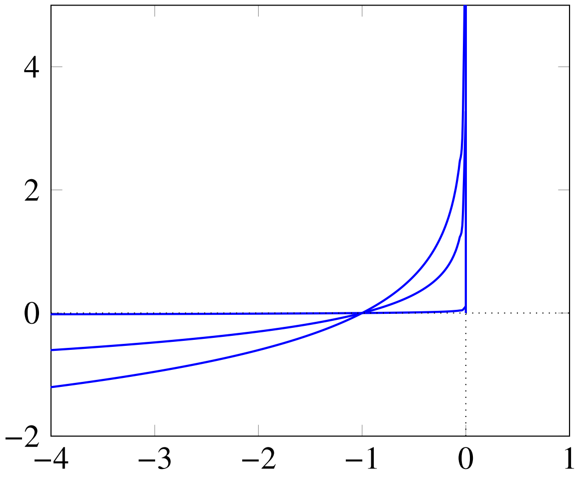


Figure 20: A plot of $\hat{I}\_(u)$ with $t = 0.5$, $t = 1$ and $t = 30$. The worst representation of $I\_(u)$ being $t = 0.5$ (the line that starts the lowest) and $t = 30$ (line that starts at 0) being what represents it the best.

The objective function becomes:

$$\text{minimize} \quad f(x) + \sum_{i=1}^{m} -\frac{1}{t} \log(-g_i(x)) = f(x) - \frac{1}{t}\phi(x) \tag{25}$$

Where $t > 0$ is a parameter that sets the accuracy of the approximation. Since $\hat{I}\_(u)$ is convex, the objective function stays convex. The more t increases the better of an approximation $\hat{I}\_(u)$ becomes. If the parameter t is too large, the hessian rapidly varies near the boundaries of the feasible set, and numerical difficulties occur.

2.15.10.2 The Interior Point Method

A solution to the t parameter problem is to start with a low value of t, then solve the minimization problem (25) for that given t with Newton's method until it converges. Then increase the value of t with a multiplier of μ and do Newton's method again, but this time using the last calculated value from the previous Newton's method as the starting point. These steps will repeat until a stopping criterion stops it. This method is called the barrier method or the Interior Point Method (IPM), and can be visualized in figure 21. A suggestion for a stopping criterion is when $\frac{m}{t} < \varepsilon$, where m is the number of constraints, t is the parameter increased in sequences and ε is the specified accuracy. It is important to note that this stopping criterion is not meant for a distributed system [30, pp. 563-569].

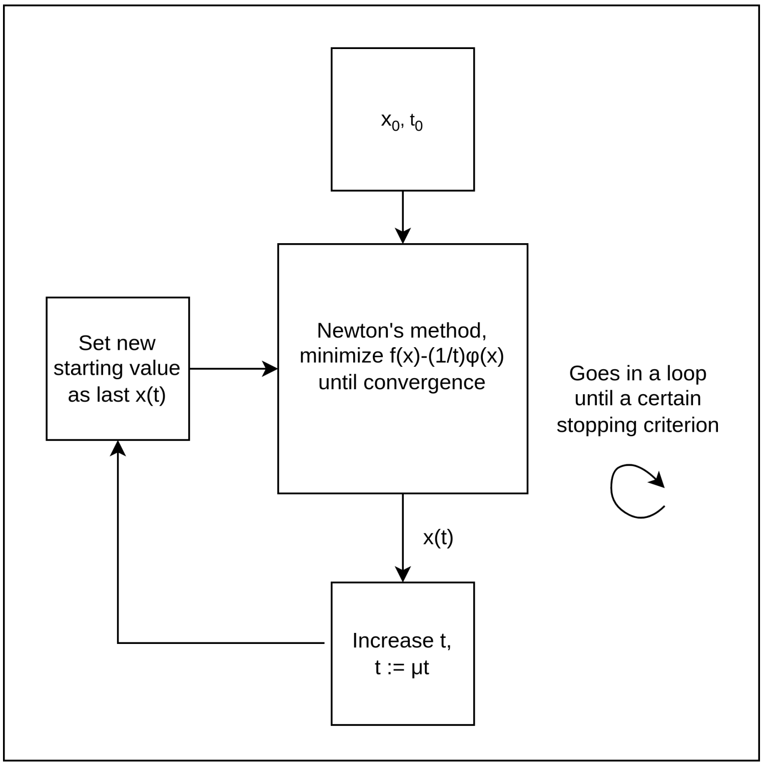

x0, t0
Newton's method, minimize f(x)-(1/t)φ(x) until convergence
Set new starting value as last x(t)
Goes in a loop until a certain stopping criterion
x(t)
Increase t, t := μt

Figure 21: The logic behind the interior point method.

Each point calculated with a certain t value ($t > 0$) is defined as $x(t)$, and the set of points $x(t)$ will all be in the interior of the feasible region. Hence, the name the interior point method.

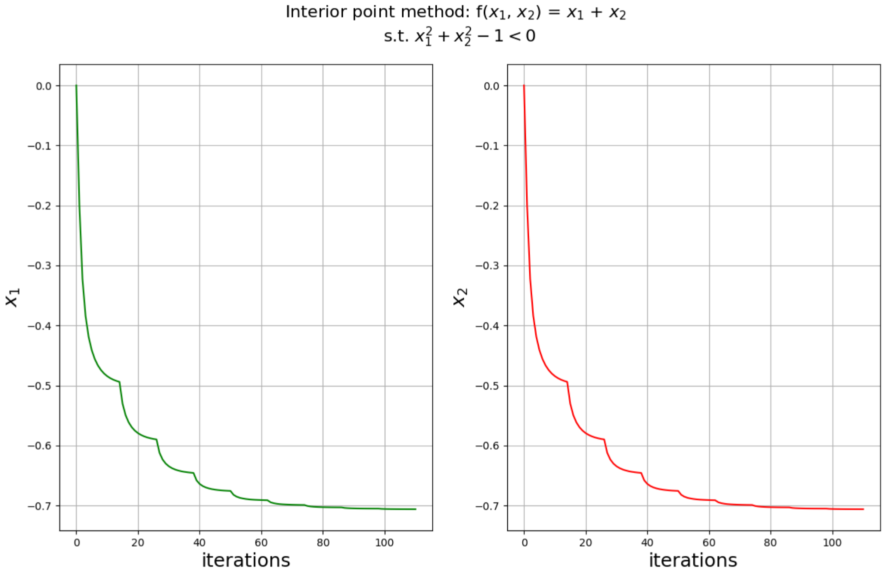


Figure 22: The interior point method on an arbitrary function $f(x_1, x_2) = x_1 + x_2$, with unit circle constraint $x_1^2 + x_2^2 - 1 \leq 0$. Note the function is unbounded above/below, but has an optimal value with the constraint.

2.15.11 Backtracking Line Search

The backtracking line search algorithm can be used to determine the step size ε of Newton's method that appropriately reduces the objective function. This line search is inexact but is quite effective and simple. It depends on two constants α and β with $0 < \alpha < 0.5$ and $0 < \beta < 1$. It is as follows [30, pp. 464-465]:

Algorithm 1 Backtracking line search for Newton's method

```
given Newton direction $d_k = \nabla^2 f(x)^{-1} \nabla f(x), \alpha \in (0, 0.5), \beta \in (0, 1)$.
$\varepsilon := 1$
while $f(x + \varepsilon d_k) > f(x) + \alpha\varepsilon\nabla f(x)^T d_k$
    $\varepsilon := \beta\varepsilon$
return $\varepsilon$
```

Where the step size starts with unit size and is reduced by a factor of β until the stopping condition $f(x + \varepsilon d_k) \leq f(x) + \alpha\varepsilon\nabla f(x)^T d_k$ is met.

3 Methodology: Introduction

This section focuses on the hardware, software and physical workspace setup utilized in this project. A general overview of what these methodologies are trying to accomplish is explained.

3.1 Equipment List

3.1.1 Hardware

Table 3: List of hardware utilities with description of the equipment.

Name	Description	Producer	Amount
RPi	Raspberry Pi 4, Model B, 8GB RAM	Raspberry Pi	1
SD-card	micro SD 32GB 100 MB/s	PNY	2
EvoLogics modem	S2C R 18/34 USBL Underwater Positioning and Communication system	EvoLogics	3
Subnero modem	M25MRS3	Subnero	2
Stationary PC's	Loaned PC's from NTNU	DELL	2
Monitors	For the school PCs and the RPi	DELL	3
DC-powersupply	Supply of power to the acoustic modems	GWINSTEK	1
Gigabit Switch		NETGEAR	2
Cat 5 cable			5
SubConn,Conductor inline cable, 6 pin	Waterproof cable	SubConn	5

3.1.2 The Physical Modem Setup

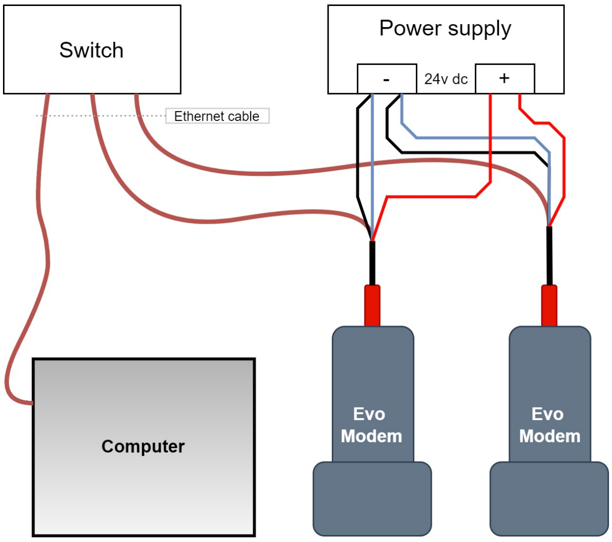


Figure 23: Diagram of how the EvoLogics modems were set up for testing in air. Modems were connected to a switch with an Ethernet cable. All of the modems were connected to a 24v DC-power supply. Note that this also applies to the Subnero modems.

The modems were connected to a switch with an Ethernet cable. All of the modems were connected to a 24v DC-power supply as shown in Figure 24.

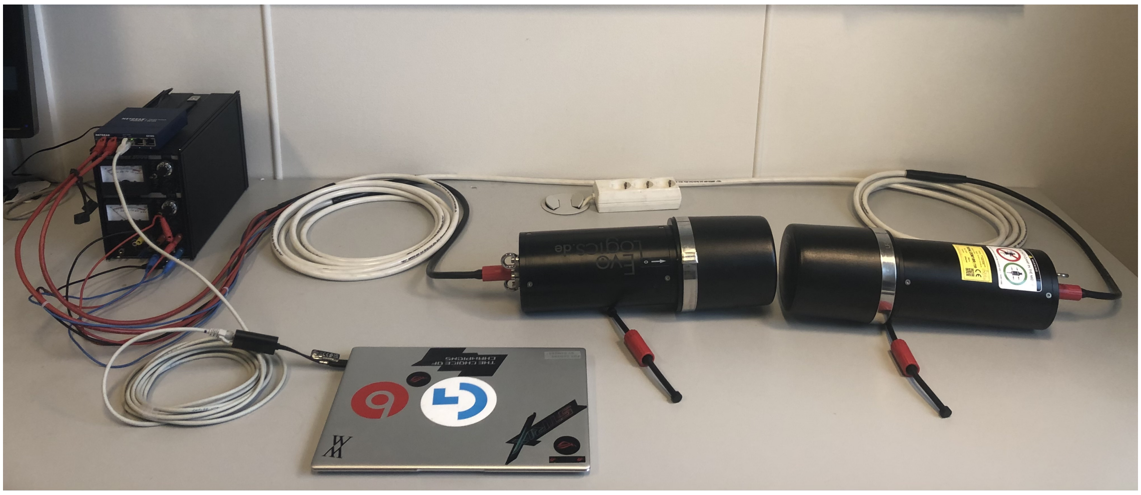


Figure 24: Image of the setup for the communication test in air with EvoLogics modem.

3.1.3 Software

Table 4 contains a list of software used during this project.

Table 4: List of software that has been used.

Software	**Description**	**Producer**	**Version**
EvoLogics AMA	Communication software for EvoLogics modems	EvoLogics	2.0.5
Raspberry Pi Imager	Used for setting up ubuntu server on RPi	Raspberry Pi	1.7.5 for Windows
Ubuntu Desktop	Environment for working on a Linux desktop	Canonical	Ubuntu Debian 22.04.1 LTS
Overleaf	Online Latex editor	Overleaf	2020 (legacy)
Google Drive	Project and file documentation	Google	
Microsoft Teams	Application for meetings and communication	Microsoft	
Visual Studio Code	Programming environment	Microsoft	Linux x64.deb
GitHub	Code documentation	GitHub, Inc.	
VirtualBox	Virtualization	Oracle	VirtualBox 7.0.6

3.2 Digital Workplace for Development

This section is about some of the setups the team has used as the virtual workplace. Although one could use alternative software, following this setup is the most reliable way of reproducing this project.

3.2.1 Ubuntu OS

The project used hardware and software that was the most compatible with the Linux operating system. The most recommended version of Linux was the Ubuntu OS, which is based on the Debian OS (Linux OS -> Debian OS -> Ubuntu OS). Ubuntu is a user-friendly OS that builds on top of Debian OS and has much of the needed ease of interaction and software available to use for free [6]. For Ubuntu OS installation, the instructions from the official documentation were used on desktop machines [7].

3.2.2 Virtualization of the Ubuntu OS with VirtualBox

Although two desktop PCs had Ubuntu OS installed as the main OS, some machines (both laptops and desktops) kept Windows 10 as the main OS, and virtualization to emulate the Ubuntu OS and work from there.

VirtualBox was used as software for virtualization, as it was available for free and met all the demands of this project. The installation was done by following the instructions in the official documentation for VirtualBox: [12, c. 1.5].

A new virtual workspace was created in combination with Ubuntu OS and VirtualBox by following the instructions from the official Ubuntu tutorials for installation of a virtual machine [13]. The Ubuntu image used was also found on this webpage. The group ticked off the `skip unattended installation` check box when the virtual machine was created. The setting creates a premade user profile without sudo privileges, which is unwanted due to the requirement of having sudo privileges throughout the project. If one wishes to install with the premade profile, it is possible to regain the sudo privileges by following many available guides found on search engines.

3.3 The General Idea of Communication and Data Processing Using Distributed Modems

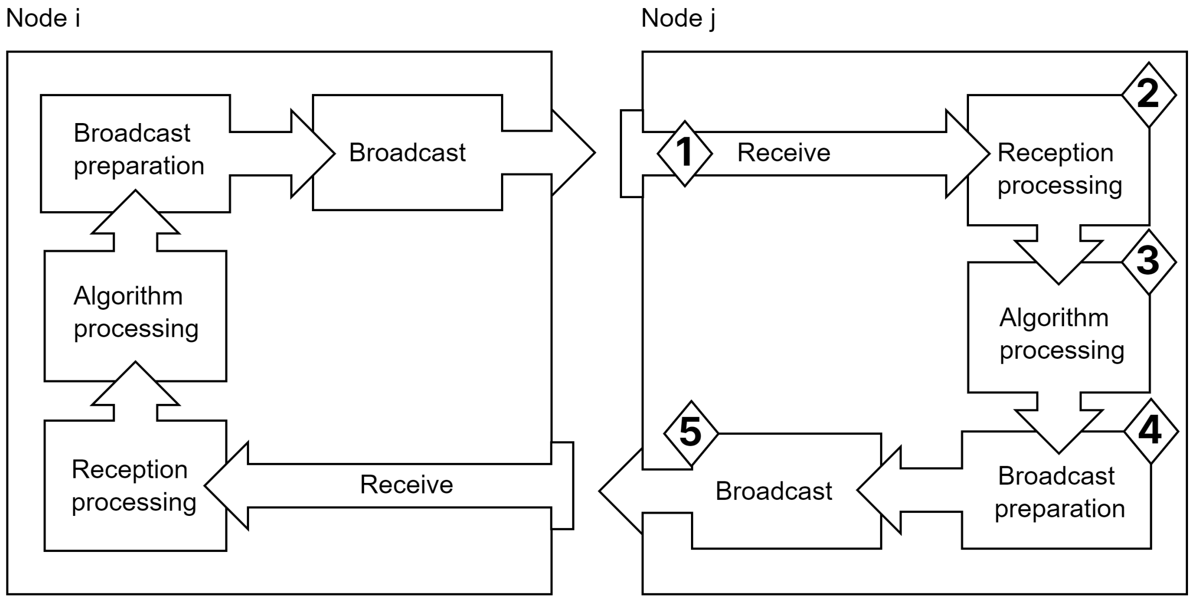


Figure 25: Basic overview of the data processing and communication between a node i and j

The project has many different parts working together to accomplish a distributed communication scheme. Our general plan for a solution is illustrated in figure 25.

1. **Receive** Each node has a method of reception. The method of reception depends on the modem in use, which in this project is either the EvoLogics or Subnero brand. While Subnero has a premade method of communication, which is the UnetStack API, and has its documentation available publicly; the EvoLogics modem has its reception module created by this projects team, and works differently than the Subnero modem.

2. **Reception Processing** Due to the way the transmission and reception work, the received data needs to be transformed into manageable data. This step is shared between this part and part 3.

3. **Algorithm Processing** In this step the data is transformed into manageable numbers so the algorithm module can work with the information. The algorithm updates the received data and prepares it for part 5.

4. **Broadcast Preparation** The same as part 2, the data needs to be processed so it can be transmitted.

5. **Broadcast** The same as part 1 except it is a method of transmission rather than reception.

4 Methodology: Optimization Algorithm Code

4.1 Pseudocode

The algorithm structure comes from the pseudocode found in Attachment A8. This algorithm merges the three building blocks together found in the theory section 2.15, into the ra-NRC algorithm found in section 2.15.9:

Each block is assumed to be executed atomically and sequentially. This means if a node is running the estimate update block and a new packet is on the way, the packet is either placed in a buffer or dropped until estimate update is done running. It is assumed that when a node is idle, it is ready for reception and that $flag_{recpetion,i}$ is set to one when a packet is received. The individual blocks are explained below.

ra-NRC for node i - Initialization block

```
procedure INITIALIZATION (ATOMIC)
    $x_i \leftarrow x^0$
    $y_i \leftarrow 0,\ g_i \leftarrow 0,\ g_i^{\text{old}} \leftarrow 0$
    $z_i \leftarrow I_n,\ h_i \leftarrow I_n,\ h_i^{\text{old}} \leftarrow I_n$
    $\sigma_{i,y} \leftarrow 0,\ \sigma_{i,z} \leftarrow 0$
    $\rho_{i,y}^{(j)} \leftarrow 0,\ \rho_{i,z}^{(j)} \leftarrow 0, \quad \forall j \in \mathcal{N}_i^{in}$
    $flag_{\text{reception},i} \leftarrow 0,\ flag_{\text{update},i} \leftarrow 0$
    $flag_{\text{transmission},i} \leftarrow 1$
end procedure
```

The initialization block initializes variables used by the algorithm. Here the initial guess x^0 is set for the global optimization and values are set to either zero or the identity matrix. This block will only run once at startup.

ra-NRC for node i - Data transmission block

```
procedure DATA TRANSMISSION (ATOMIC)
    if $flag_{\text{transmission},i} = 1$ then
        transmitterNodeID $\leftarrow i$
        $y_i \leftarrow \frac{1}{|\mathcal{N}_i^{\text{out}}|+1} y_i$
        $z_i \leftarrow \frac{1}{|\mathcal{N}_i^{\text{out}}|+1} z_i$
        $\sigma_{i,y} \leftarrow \sigma_{i,y} + y_i$
        $\sigma_{i,z} \leftarrow \sigma_{i,z} + z_i$
        Broadcast: transmitterNodeID, $\sigma_{i,y}$, $\sigma_{i,z}$
        $flag_{\text{transmission},i} \leftarrow 0$
    end if
end procedure
```

The data transmission block is where node i prepares the values it is going to transmit to its neighbors. The values it is sending are the weighted version of y_i and z_i, plus its node ID. The weighted versions stem from the first part of the update rule (equation 17 in the theory section 2.15.7.4). y_i is the sum of the expression $\nabla^2 f_j(x_j)x_j - \nabla f_j(x_j)$, but the value it sends can be thought of like a little slice of the gradient. h_i is the sum of the hessians $(\nabla^2 f_j(x_j))$ and the value it sends can be thought of as a little slice of the hessian. Before sending the slices of y_i and h_i, they get added to $\sigma_{i,y}$ and $\sigma_{i,z}$. The sigmas are counters on how much node i has sent of y_i and h_i. The two sigmas are what is being broadcast instead of y_i and h_i. Since the algorithm includes these sigmas, the receiving part can store the sigmas it receives and compare sigmas when the next packet comes. With this, the algorithm preserves the masses, and the algorithm will still work. The node returns to an idle state after transmitting and is then ready to receive a new packet.

ra-NRC for node i - Data reception block

```
21: procedure DATA RECEPTION (ATOMIC)
22:     if $flag_{\text{reception},i} = 1$ then
23:         $j \leftarrow$ transmitterNodeID, $(j \in \mathcal{N}_i^{\text{in}})$
24:         $y_i \leftarrow y_i + \sigma_{j,y} - \rho_{i,y}^{(j)}$
25:         $z_i \leftarrow z_i + \sigma_{j,z} - \rho_{i,z}^{(j)}$
26:         $\rho_{i,y}^{(j)} \leftarrow \sigma_{j,y}$
27:         $\rho_{i,z}^{(j)} \leftarrow \sigma_{j,z}$
28:         $flag_{\text{reception},i} \leftarrow 0$
29:         $flag_{\text{update},i} \leftarrow 1$ (optional)
30:     end if
31: end procedure
```

The data reception block is where node i receives the sigmas and the node ID from its neighbors. The receiving node is going to be noted as node j. Node j runs exactly the same algorithm as node i but could use a different cost function. One can imagine node j just ran the data transmission block and now node i has received the h_j and y_j slices from node j. Lines 24 and 25 in the pseudocode are where the second part of the rule update (equation 18 in the theory section 2.15.7.4) comes to play. It is written a little differently because of the addition of the sigmas in the robust ratio consensus, but when there is no packet loss it is exactly the same. When a packet is lost, the term $\sigma_{j,y} - \rho_{i,y}^{(j)}$ in line 24 will as a result be zero because $\rho_{i,y}^{(j)}$ was set equal to $\sigma_{j,y}$ in the last successful transmission. However, if there is no packet, $\sigma_{j,y}$ will be equal to what it was in the last successful transmission. This ensures that the algorithm will not add any false y-mass and that the average consensus will still converge to the desired value. The same logic for z-mass is applied to line 25 with $\sigma_{j,z}$ and $\rho_{i,z}^{(j)}$. Because this algorithm is robust to packet losses, multiple nodes can transmit at the same time, since collision is already handled.

ra-NRC for node *i* - Estimate Update block

```
32: procedure ESTIMATE UPDATE (ATOMIC)
33:     if $flag_{\text{update},i} = 1$ then
34:         $x_i \leftarrow (1-\varepsilon)x_i + \varepsilon z_i^{-1} y_i$
35:         $g_i^{\text{old}} \leftarrow g_i$
36:         $h_i^{\text{old}} \leftarrow h_i$
37:         $h_i \leftarrow \nabla^2 f_i(x_i)$
38:         $g_i \leftarrow h_i x_i - \nabla f_i(x_i)$
39:         $y_i \leftarrow y_i + g_i - g_i^{\text{old}}$
40:         $z_i \leftarrow z_i + h_i - h_i^{\text{old}}$
41:         $flag_{\text{update},i} \leftarrow 0$
42:         $flag_{\text{reception},i} \leftarrow 1$ (optional)
43:     end if
44: end procedure
```

The estimate update block is where node *i* uses the values it has received from its neighbors and updates its own variables accordingly. A Newton's method step is used in line 34 to update the local estimate x_i of the global optimizer. x_i will be more accurate the more nodes have communicated with each other because the push-sum consensus will make the network progressively agree on the values of y and g. Additionally, more of the Newton's steps will be executed. In line 35, g_i^{old} will be set equal to the last g_i value. Later, in line 38, the new value will be calculated and set to the variable g_i. In line 39 y_i get this difference added. If the old values and new values are not the same, y_i will change values accordingly, and the local estimate x_i will hopefully be sent in the right direction for the global optimal solution x^*. Same logic is applied to z_i, h_i^{old} and h_i.

4.2 Resources for Further Improvements

A more detailed explanation of the pseudocode can be found in the thesis "*Multi-agent Newton-Raphson Optimization Over Lossy Network*" by Bof, N. et al. [36]. Considerable use was made out of this thesis. Further, more intricacies about the interior point method, backtracking and convexity can be found in the book *"Convex Optimization"* by Boyd, S. P. [30].

Since there are many different parts at play in the ra-NRC, the algorithm has been split into different Python files for a more structured way of understanding the concepts. There will be some brief overviews of the content in the different Python files (more details are commented in the files). All files can be found in Attachment B as `Optimization-main-sims (code).zip` or in the Github [62].

`newtons_method.py`

In this Python code, Newton's method is used on an arbitrary three-variable function. To change the function one can just change the `function` variable. It has to be a three-variable function because of the gradient/hessian setup, but one can change this manually if wanted.

`backtracking.py`

Here Newton's method is used on a two-variable function. The backtracking line search algorithm is used in the `calculateHessianAndGradient()` function. To "turn off" backtracking and see the difference, one can manually set `epsilon` to 1. Backtracking did not get included in the final ra-NRC algorithm.

`interior_point_method.py`

Very similar in style as the `newtons_method.py`. When Newton's method makes `xi` converge, the interior point method parameter t is increased by a factor of μ. This runs until the stopping criterion $\frac{m}{t} < tolerance$ is met. See figure 21 for visualization. There are two example functions available. Note that the first function will require a lower starting t. When using the IPM it is important to be cautious when picking t values and μ.

`scipy_optimize.py`

Scipy is a Python library that can be used as a way of finding the optimal point in a centralized way. It can be useful to compare function values and see if your outputs make sense. Also possible to add constraints to the optimization problem. Notice the answers can be somewhat different because the code uses two different methods from the Newton's method. In the file, the cost function is used as an example function, with constraints.

`ra_nrc.py`

The structure of this code is very similar to the pseudocode. The pseudocode however does not directly include the interior point method. The code does not include packet losses and is an ideal situation with no float16 restrictions. It is set up such that two nodes, node i and node j, communicate with each other in a forced sequence. The sequence goes: node i transmits a packet and node j receives that packet and updates its values. Then node j transmits a packet itself and node i receives it and updates its own values. After this, the sequences go in a loop. Notice the interior point method parameter is called `bb` instead of `t`. The code is set up such that `bb` will increase in value for every `gamma` iteration. In the first iteration, the step size ε is initialized to a very small value because when doing the first Newton's method iteration, the y_i

is 0 so line 34 in the pseudocode would essentially be $x_i = (1 - \varepsilon)x_i + 0$. This means when initializing the ε with a low value in the first iteration, the first x_i value will not drop heavily in value. This code could be useful for simulating new ideas and trying new optimization values to see how the system reacts.

`ra_nrc_rl.py`

Very similar in structure to the `ra_nrc.py` with the two node forced sequence. However, this will represent the ra-NRC in real life, hence the `rl` suffix. One important difference is that the sigmas received are float16, instead of the original float32. This causes problems using the interior point method, so the code will now use a constant IPM parameter, $bb = 1$. Since this value is constant, the code will converge towards what is essentially an approximation of the objective function. If max iterations is modified larger, one will more clearly see the issues caused by the float16 restriction.

5 Methodology: Implementation of ROS 2 & Subnero Communication

5.1 Installation and Usage of ROS 2 on Ubuntu OS with Python 3.0

Before ROS 2 was ready for development, modification and execution, a preliminary setup of a ROS 2 workspace and package installation was required. The instructions were followed step-by-step, which are available online at the official ROS 2 webpage for Debian OS [5]. As per the documentation, repeated use of `sudo apt update` was used in order to make sure the system was up to date when installing new packages.

5.2 Creating the ROS 2 Workspace

5.2.1 Prerequisites

Before creating the workspace some additional features were added for an easier implementation and continuation of development. A tool developed by the ROS 2 team called "colcon" was installed for the ease of building each package using the following command in the Linux terminal from the root folder.

```
    sudo apt update
    sudo apt install python3-colcon-common-extensions
```

Listing 2: Colcon extension.

Afterward, automatic sourcing of the auto-complete function was installed by running (from the root folder)

```
    gedit ~/.bashrc
```

Listing 3: Sourcing directory.

and inserting this at the bottom of the bashrc file:

```
    '/usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash'
```

Listing 4: colcon argument auto-complete.

This made sure that every time a terminal in Linux was opened, that bash file was executed in every new terminal.

5.2.1.1 Prerequisites: Folder Structure

The main folder where all of the ROS 2 utilization was executed, was created in the root folder of the Ubuntu OS. The workspace folder was manually created through the Ubuntu folder menu. The structure, as per documentation [2], should be as in figure 26.

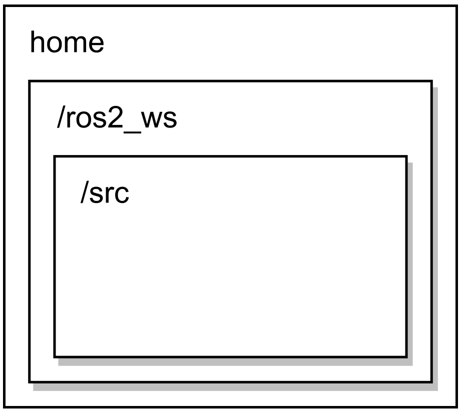


Figure 26: The basic folder structure of the ROS 2 workspace.

5.2.1.2 Prerequisites: Colcon Build Again

To finalize the workspace, the build command was used to add the necessary folders (/src, /build, /install, /log) automatically. The following command was used to finalize the prerequisites:

```
colcon build
```

Listing 5: The package build command.

5.2.2 Creating the ROS 2 package

After the necessary workspace was completed, a main package was created. The main package was to be used with Python 3.11 and had to be specified by using `ament_python`. The name of the package was chosen to be `nodecomx`. It is important that the terminal is located within the `/src` folder when creating a package:

```
ros2 pkg create --build-type ament_python nodecomx
```

Listing 6: Package creation.

After the package creation, ROS 2 modules were created by creating python files within the `/ros2_ws/src/nodecomx/nodecomx` folder, and writing the barebones ROS 2 code for the main set-up for the project.

5.3 Creating the Basic ROS 2 Nodes

After the workspace was complete it was ready for node and module implementation.

5.3.1 Receiver Node

The receiver node was meant to be used as a reception node for the UnetStack API and EvoLogics API. For all of the ROS 2 nodes, the necessary packages were included:

- `import rclpy` and `from rclpy.node import Node` are the imported modules for Python implementation for interacting with ROS 2 using Python 3.11 [8].
- `from std_msgsṁsg import String` allowed for the usage of string formatted messages when publishing and subscribing between topics. The choice for using strings was due to the changing of the datatype and formatting of the incoming messages through reception. Using strings made the flow of data easier to handle as the published message only needed to be turned into a string in order to work.

Due to how the entire system was to function, looking through a bird's eye perspective, the receiver node had the timing control of a chain reaction for data flow. This meant that the receiver node required a timer for cycling itself. This was achieved by using ROS 2 timer callbacks, however, it was later conceived that the cycling of this node was done automatically by the `receive` timeout from the UnetStack API.

The rest of the node followed the basic instructions for ROS 2 Humble for a publisher node, and the node was ready for further implementation of other modules like the UnetStack API for Python, UnetPy.

5.3.2 Processing Node

The processing node was implemented with the intention of it being the middleman in the ROS 2 system. This node structure was implemented similarly to the receiving node, however, it is without the addition of a timer callback and without any implementation of communication with UnetStack; and instead of just a publisher it also contains a subscriber. Due to the interest of this node to have the algorithm module for processing the incoming data, most of the development of the ROS 2 system was done in this node.

5.3.3 Transmitter Node

The last node in the system was the transmitter node and as its name implies, the node has the task of handling the broadcasting of any message that comes through the topic. The structure is similar to the receiver node with the exception of only using a subscriber.

The ROS 2 setup ended up having this logical software structure; the flow of data between three nodes as illustrated in figure 27:

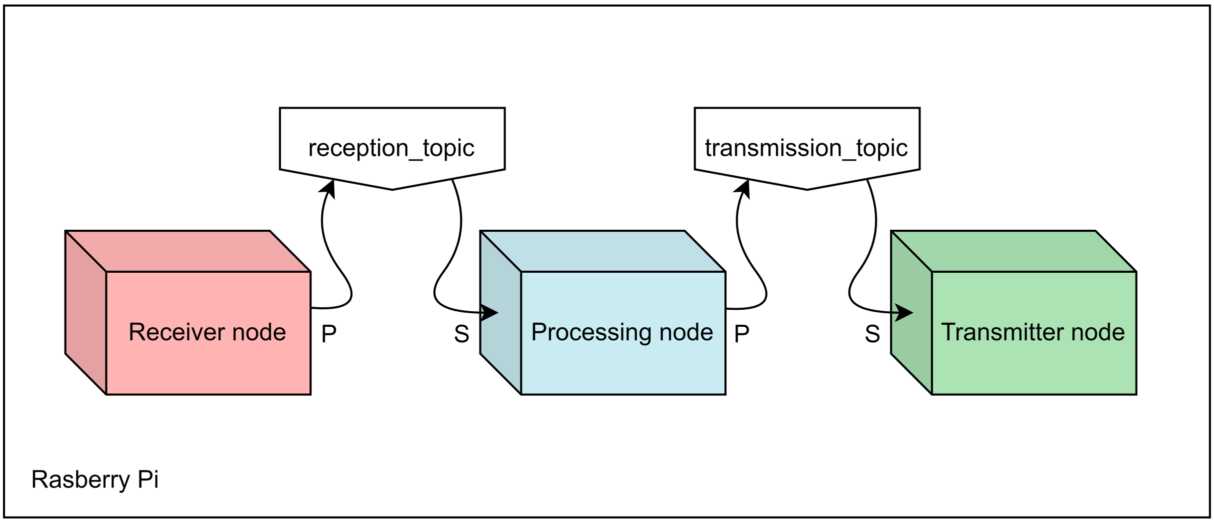


Figure 27: The logic flow of the ROS 2 system for the project. ***P*** *and* ***S*** *stand for Publisher and Subscriber.*

Before the deployment for testing of the ROS 2 system, it was required to build using `colcon build -symlink-install` in the Ubuntu terminal, from the root location of the workspace (/ROS2_ws). The `-symlink-install` command to the `colcon_build` allowed modification of each node's code, without building each time after a minor change. The skeleton was ready for further development and implementation of communication and algorithm processing.

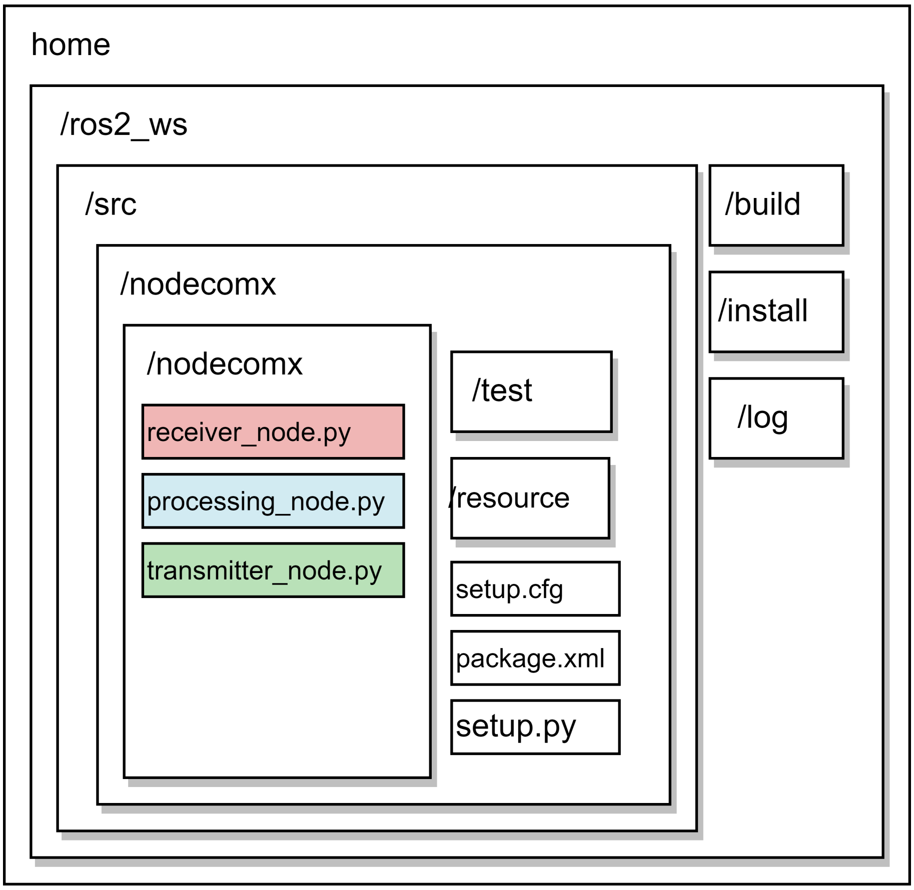
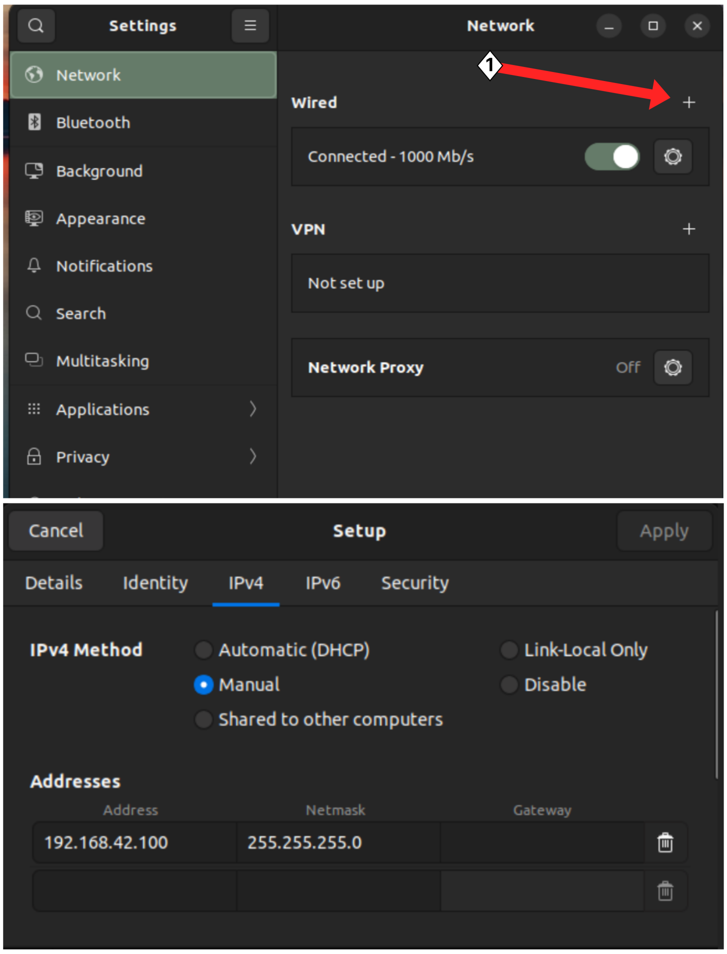


Figure 28: The final structure of the skeleton of the ROS 2 system, is ready to be expanded on.

5.4 UnetStack & Subnero Communication Setup with ROS 2

Before utilizing the UnetStack interface by accessing the web interface, one has to be in the same network. Two of the Subnero modems that were available to the project had the local static IP addresses `192.168.42.195` and `192.168.42.86`. By changing the address of the interface machine (an external machine), one can access the modem's information, settings and storage through the Unet web interface. The modems and the interface machine were connected to a network switch, and a new network profile was made in the Ubuntu machine so that the third segment was the same. The new static local IP address on the machine is `192.168.42.100`. The host ID, which is the last element in the IP address, can be anything between 0 and 255 except `195` and `86`, since they are reserved by the Subnero modems (but can be changed through the use of commands).



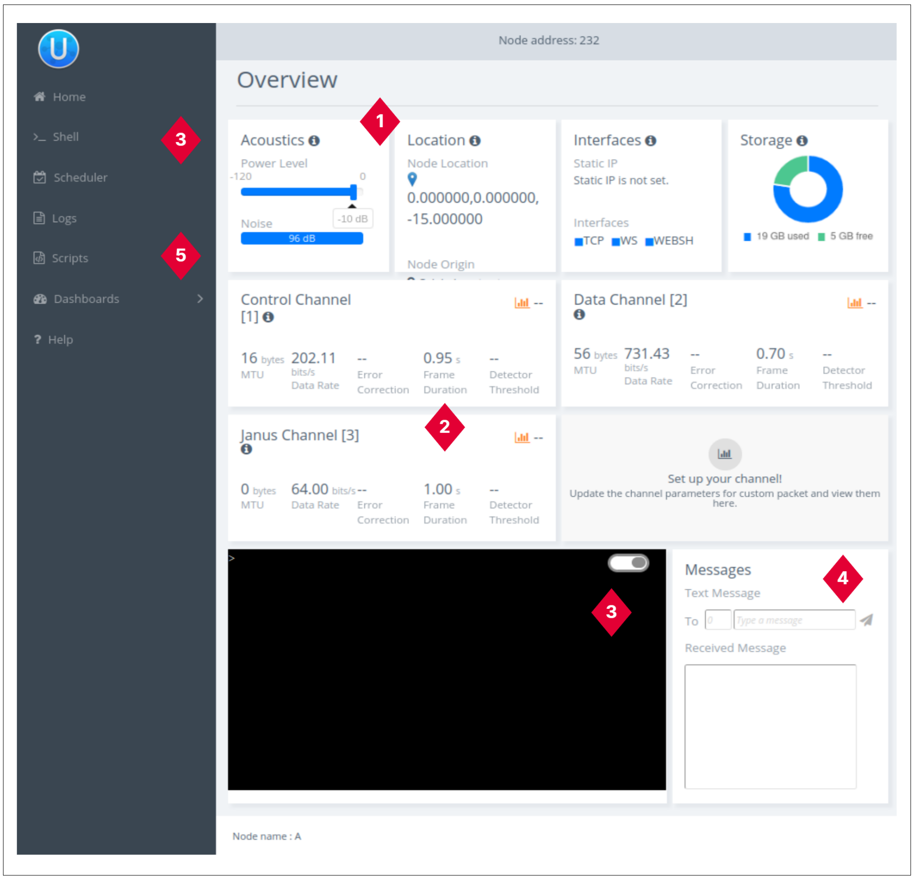
IPv4 Method
○ Automatic (DHCP)
● Manual
○ Shared to other computers
○ Link-Local Only
○ Disable

Addresses

Address	Netmask	Gateway
192.168.42.100	255.255.255.0	

Figure 29: On an Ubuntu Debian machine, by going to the Network settings one can change the address to access the Subnero modem web interfaces. ***1****: Creating a new profile.* ***2****: through IPv4 tab a new address is added.*

5.4.1 The UnetStack Interface



Node address: 232

Power Level -120 0 -10 dB

Noise 96 dB

Node Location 0.000000,0.000000, -15.000000

Node Origin

Static IP: Static IP is not set.

Interfaces: TCP WS WEBSH

19 GB used 5 GB free

Control Channel [1]: 16 bytes MTU; 202.11 bits/s Data Rate; -- Error Correction; 0.95 s Frame Duration; -- Detector Threshold

Data Channel [2]: 56 bytes MTU; 731.43 bits/s Data Rate; -- Error Correction; 0.70 s Frame Duration; -- Detector Threshold

Janus Channel [3]: 0 bytes MTU; 64.00 bits/s Data Rate; -- Error Correction; 1.00 s Frame Duration; -- Detector Threshold

Set up your channel! Update the channel parameters for custom packet and view them here.

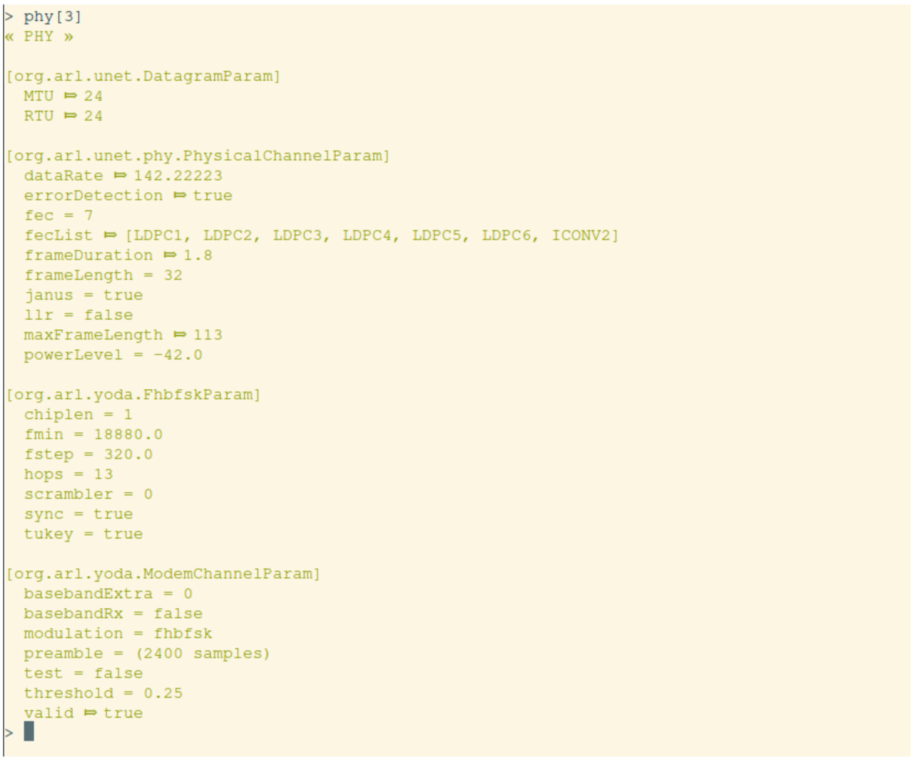
Text Message: To 0 Type a message

Received Message

Node name : A

Figure 30: The UnetStack web interface, which can be accessed by entering the respective modem IP in a web browser. ***1****: The power level of the transducer. When testing in air, the level should be at -40dB.* ***2****: The physical channel layer, JANUS channel.* ***3****: The shell command window. The interfacing happens through here with the use of commands.* ***4****: A simple message transmission, it was not used as much, however, it is useful to test whether the modems can communicate.* ***5****: The scripts section allows for logging of data and uploading of scripts, including startup scripts which may be written to change the settings automatically.*

For the project, a setup was required before possible communication between the two Subnero modems. Through the shell, writing the command `phy[3]` would display the current settings of the physical layer 3, JANUS; which is the number **2** in figure 30.



```
> phy[3]
« PHY »

[org.arl.unet.DatagramParam]
  MTU ⤇ 24
  RTU ⤇ 24

[org.arl.unet.phy.PhysicalChannelParam]
  dataRate ⤇ 142.22223
  errorDetection ⤇ true
  fec = 7
  fecList ⤇ [LDPC1, LDPC2, LDPC3, LDPC4, LDPC5, LDPC6, ICONV2]
  frameDuration ⤇ 1.8
  frameLength = 32
  janus = true
  llr = false
  maxFrameLength ⤇ 113
  powerLevel = -42.0

[org.arl.yoda.FhbfskParam]
  chiplen = 1
  fmin = 18880.0
  fstep = 320.0
  hops = 13
  scrambler = 0
  sync = true
  tukey = true

[org.arl.yoda.ModemChannelParam]
  basebandExtra = 0
  basebandRx = false
  modulation = fhbfsk
  preamble = (2400 samples)
  test = false
  threshold = 0.25
  valid ⤇ true
> █
```

Figure 31: This is a printout of the `phy[3]` *parameters in the shell. These are the available parameters of the JANUS channel.*

5.4.2 Physical Subnero Modems

Physical Connection

Before interfacing with the Subnero modems, a physical setup as shown in 23 is required.

Depending on the use case, some settings need to be configured for proper communication. Reliable communication in air was defined by setting the threshold value to 0.25. In order not to damage the modems, the powerlevel was set to -42dB. These settings were set by using the shell within the UnetStack web interface. These settings are not persistent through restart of the modem, and need to be re-initiated by manual setting or using a startup script:

```
phy[3].threshold = 0.25
phy[3].powerLevel = -42
```

Listing 7: "Settings for the layer 3 channel JANUS".

5.4.2.1 Setting Up the Broadcasting/Reception Settings

Using the JANUS protocol, the message length (amount of symbols) that could be sent is restricted to the `frameLength` setting. A `frameLength` of 32 would mean that the entire transmission was 32 bytes long, however only 24 bytes were available for transmission (Maximum Transmission Unit (MTU)); the JANUS protocol reserves eight of the bytes for its frame. A `frameLength` of 8 would mean that there are no available bytes for a cargo inside the frame.

5.4.2.2 For Testing in Air

The messages were broadcast using different methods: as a shell command; through the simplified message interface; Python with UnetStack API, UnetPy. Only the shell and UnetPy were the focus.

Using the shell, a message was transmitted using the following command:

```
phy[3] << new TxJanusFrameReq(data: "This is 21 bytes long" as bytes[])
```

Listing 8: "The transmission command for JANUS".

However, if not specified, the default settings for the frame settings will be applied; `appData`, `classUserID`, and `appType` will all be `0`. In order to transmit correctly, a correct `appData` setting must be used in conjunction with the requested frameLength and a `classUserID` of `16`. The available combinations of `appData` and `frameLength` can be found in the attachment A2.

Thereafter, a successful message can be transmitted by using these settings for a 21 byte long message:

```
phy << new TxJanusFrameReq(classUserID:16, appData:6, data: "This is 21
    bytes long" as byte[])
```

Listing 9: Command for transmission from Subner.

For the message to be received, a command had to be ran to enable reception mode. However, it was also required to enable `subscribe` mode, by using the command `phy subscribe`. For the Subnero modems that is enough to communicate between them.

However, if receiving from the EvoLogics modems, the following command was used to enable reception for the Evologics:

```
phy << new RxJanusFrameNtf(classUserID:16, appData:6)
```

Listing 10: Command for reception.

The modem was able to receive a message transmitted by the other modem. In order to see the message, one had to unpack it by using the following command right after the complete message was received (and not the "reception incoming" message):

```
new String(ntf)
```

Listing 11: "Unpacking the notification message".

The message then would be returned in decoded strings.

For a total list of available commands, one can write `help phy` in the shell.

5.4.3 Simulated UnetStack Modems

It was possible to simulate two modems using UnetStack simulation. This has been done by downloading the UnetStack 3.4.0 Community Edition folder from the official website [10], and opening a terminal on a Linux machine with a directory within the downloaded folder. Using the following command would simulate two UnetStack modems (not specifically Subnero):

```
bin/unet samples/2-node-network.groov
```

Listing 12: The groovy files are run from the Unet download folder

For more information regarding simulated modems see the Unet documentation [11, c. II, 4].

5.5 Using UnetPy to Establish Broadcasting and Reception in Python

The usage of UnetPy was mandatory for the usage of the Subnero modems with Python code. By using the documentation [11, c. III, 9], and with trial and error, a usable code was written which allowed for basic communication. The transmission code is:

```python
from unetpy import *

sock = UnetSocket(modem_ip, portnumber)
rx = sock.getGateway().receive(RxJanusFrameNtf, timeout=5000)
print(rx.data)
```

Listing 13: unetpy reception.

And the reception code is:

```python
from unetpy import *

sock = UnetSocket(modem_ip, portnumber)
phy = sock.agentForService(Services.PHYSICAL)

phy << TxJanusFrameReq(classUserID = 16, appData = 6, data = "15 byte
    message")
```

Listing 14: unetpy transmission.

This setup was all that was needed to establish a JANUS frame communication between the Subnero modems using Python. However, the `appData` is interdependent with the `frameLength`, which is shown in attachment A2.

5.5.0.1 The Float16 Converter

It was essential that the transmitted and received message adhered to a byte format for wireless communication. To fulfill that requirement a float-to-byte and byte-to-float converter module was developed in Python.

The converter addressed the need for slicing and assembling a float16 with bytes in an efficient manner. The module was designed with object-oriented programming principles, enabling accessibility for both internal and external groups. Within this module, a numerical value was passed into the `numpy.float16` module, subsequently quantized and then manipulated through the process of slicing the 16 bits into two sets of 8-bit segments. These segments were treated as int8 datatypes and utilized for transmission purposes. The recombination of the bytes followed an opposite procedure. A simplified depiction of the data flow within the float16 converter is presented in Figure 32.

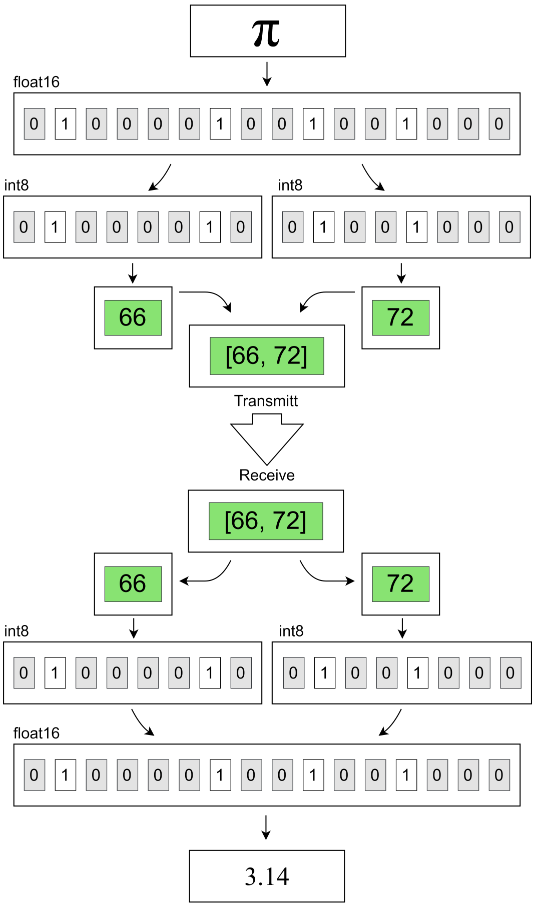


Figure 32: The float16 to bytes converter, and bytes to float16 merger flowchart. Notice the quantization of the irrational number π; entering π into a float16 quantize its value, and due to the half-precision of float16 the result ends up being 3.14 on the receiving end.

5.5.0.2 The Final System with ROS 2, UnetPy, Float16 Converter and the Optimization Algorithm

The implementation of the ROS 2 modules, the UnetPy, the float16 converter and the algorithm continued after each module was ready. These modules were combined into each of the ROS 2 nodes (receiver, processing, and transmitter). The complete system was then tested and adjusted accordingly until non-interrupted communication was achieved, and the processing of information, was successful.

6 Methodology: Communication Mechanism for EvoLogics

6.1 Prerequisites for EvoLogics

This section covers how the setup was done to run code for the EvoLogics modems and how JANUS was utilized. It includes the necessary installations and preparations before any execution of the code.

Communication Verification

Before connectivity, a physical setup was required as shown in the introduction of the methodology, see section 3.1.2.

The AMA communication software was used the first time to check the connection to the EvoLogics modems with the ethernet cable and to test transmission between the modems.

6.1.1 Installing Programs and Libraries for EvoLogics Communication Development

The commands used for the necessary installations are described below. They were executed in the terminal on the virtual machines. The following steps can also be found on the GitHub [48] and in the previous bachelor group's thesis [53, p. 13].

Make and Cmake

Make and Cmake were installed to be able to compile the code. The version used by the group was: GNU Make 4.3.

```
make -version
sudo apt install make
sudo apt install build-essential
sudo snap install cmake --classic
```

Listing 15: Installing Make, Cmake and some essential packages for compiling. Build-essential also includes Make [61].

FFTW3

The FFTW3 library is required to run JANUS and is described in the document "README -JANUS Tool Kit 3.0.1" [56]. There was created a new folder in the "lib" folder called "fftw" and the `FFTW3` was installed in this folder.

```
1 cd project/lib/fftw
2 wget https://fftw.org/fftw-3.3.10.tar.gz
3 tar -xzf fftw-3.3.10.tar.gz
4 cd fftw-3.3.10
5 ./configure
6 make
7 sudo make install
8 make check
```

Listing 16: Terminal commands for FFTW3 configurations: The "project" in line 1 must be replaced with the current projects name.

Libreadline

`libreadline` needs to be installed to be able to run the SDMSH library [50].

```
sudo apt-get install libreadline-dev
```

Listing 17: Terminal command for installing libreadline.

6.1.2 Compilation

Compiling for SDMSH and JANUS

Compiling the SDMSH library:

```
cd project/lib/sdmsh
make
```

Listing 18: In the SDMSH library folder, run `make` to compile.

Setup and compile the Janus-c-3.0.5 library:

```
cd project/lib/janus-c-3.0.5/
cmake -S . -B bin/
cd bin
make
sudo make install
```

Listing 19: In the janus-c.3.0.5 library folder, create a bin folder using `cmake`. This bin folder must be compiled using `sudo make install`.

The command `make` was used when modifications were made to either the `SDMSH` or the JANUS library. Sometimes the group got an error after trying `make` in the *janus-c-3.0.5*. To fix this, the bin was deleted and made again following the steps above.

6.2 Setting up the EvoLogics Modems

In this section, the wiring and settings needed for both the Receive (RX) and Transmit (TX) modems will be explained. The "Getting Started Guide" for the S2C EvoLogics modems was read and reviewed before conducting any testing. This guide provides a detailed description of what to be thoughtful about when using the modems. The guide is listed as a source; however, it is not accessible on the internet [54].

6.2.1 Setting the PHY-mode

PHY mode is required to be able to use the SDMSH library [50]. The mode was set in two different ways.

One way was using the **AT command:** Connect to the modem using Netcat, and use the AT commands. This was done with the command:

```
nc MODEM_IP MODEM_SOCKET #Default socket is 9200
+++ATP
```

Listing 20: Setting the modem in PHY mode. To check if the modem is already in PHY mode use the AT?S command instead of ATP.

When the modem was set in "PHY" using this method, a message was printed to the terminal when successful:

```
+++ATP:14:INITIATION PHY
```

Listing 21: The modem is in PHY mode when receiving the message "INITIATION PHY".

Using `stop;` in the SDMSH commands is another way of setting the modem in PHY mode.

```
./sdmsh mIP -e 'stop;config ...'
```

Listing 22: This is an example of how to use `stop;` in the config command to set PHY mode.

The AT command for the EvoLogics modems was used in early testing, later `stop;` was used instead. It was implemented as a default setting for many "Evo_janusXsdm"-commands and is a part of the final code.

6.2.2 Setting the Modem Configurations

The configurations for the modems were set before starting a transmission. Choosing the wrong configuration can damage the modems. In air it is important to choose the lowest source setting. Therefore the source value 3 was chosen [54, p. 22].

The code created by the previous bachelor group had two functions: one for configuring settings and another for determining the source level. The group used both of the functions when testing and they were implemented in the new library "Evo_janusXsdm".

`SdmConfig()` was used for manual source level control. It gave an option for choosing a value between zero and three in the terminal, where "3" was the lowest setting. This function used the following command to set the source level:

```
./sdmsh mIP -e 'stop;config 30 0 X 0'
```

Listing 23: The X is the placement of the source value.

`SdmConfigAir()` was used to set the source level automatically to the lowest level, which is best suited for testing in air. This is done by the function running the command and changing the third number representing the source level to three:

```
./sdmsh mIP -e 'stop;config 30 0 3 0'
```

Listing 24: The source value is set to 3.

6.2.3 Setting the Reference Signal

Preamble is a reference file that helps the receiving modem understand and recognize that it is detecting a JANUS packet. This enables the modem to pack out the message as a JANUS packet and make sense out of it. It is important to note that other modems also need to use the same preamble as reference.

Preamble is a file that was provided to the previous group by Emil Wengle, and it should resemble the JANUS reference signal.

`SetPreamble()` was used to set the preamble by using the `preamble.raw` file in a SDMSH command:

```
./sdmsh mIP -e 'stop;ref preamble.raw'
```

Listing 25: Setting preamble.

6.3 Preliminary Work for Evo_janusXsdm Implementation

This section is about different problems the group has encountered while working on this project, and the solutions it resulted in. Not every little problem will be mentioned; the focus will be on the decisions which lead to the final product.

Two main problems described by the previous group were the "zombie problem" and the "100 Packet problem". It was decided to try to fix the zombie problem first.

6.3.1 Transmission Process - a Minor Zombie Problem

In the previous bachelor group's presentation, it was explained that they were not able to close some processes the right way [65, t. 20:15]. The first thing our group did was to run the previously developed code. To check if there were zombies, the command `ps -ef` was used in the terminal. It was discovered that almost every time the commands for TX were executed, it would create zombies.

The next step was to look at what caused the zombies and how the different processes were handled. In the library, `fork()` and `execvp()` were used to create new processes which were not done correctly. To fix this, the group had to implement the right amount of wait(0), and place them in the correct place in the code. The `ps -ef` command was used to check, all zombies were gone.

6.3.2 Receiving Process - a Major Zombie Problem

Zombie problems happened as well when executing the RX command. Here, the zombies were generated at a significantly higher speed compared to the TX command. After reviewing the code and conducting many tests there were uncovered two main problems. One, for the code to work properly the parent process needed to be able to work individually while the child processes did work on their own. Two, it was not implemented a way of closing the processes for JANUS and SDMSH. In the code to the previous group, it was attempting to use kill() commands.

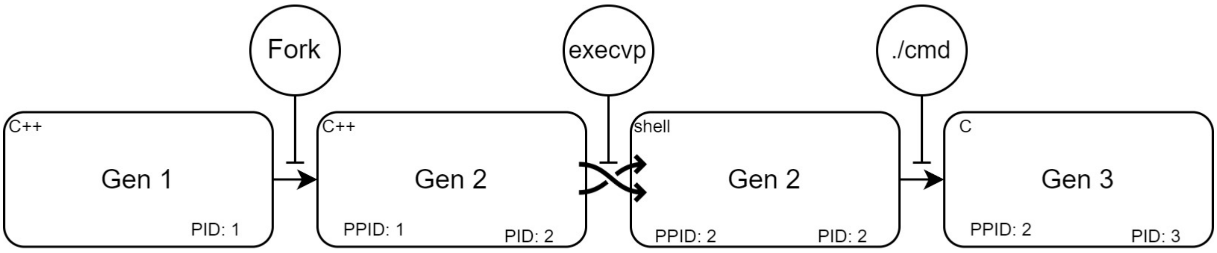


Figure 33: This illustration provides an example of the generation of processes. First Gen 2 was created using fork(), then the process image is swapped to a shell. Finally, the Gen3 process gets created through an argument executed from the shell. Either ./janus or ./sdmsh.

The first thing done was to map out the processes, figuring out the parent-child relations. It was revealed that there were three generations of processes and not two, and it was the last generations that was not reaped properly. This was due to using `fork()` once and using an `execvp()` command to launch a new process. This excluded the use of `kill()` where there was no access to the process identifier (PID) of the Gen 3 process, see figure 33.

Following this, a technique called "double forking" was implemented which would cut the connection between the parent and the child processes. Now the parent and the children could work individually.

6.3.3 Transmission Process - Stops, But Does Not Terminate

A new problem was identified when conducting more tests. When running the code for transmitting JANUS packets it would stop without any warning and it did not exist. Before the fault occurred, a short sound could be heard from the modem and a DROP message was printed from the janus-c-3.0.5 library. The next packet was not sent and a TCP error was printed to the terminal.

Since there was no way of knowing where and why the fault occurred in the code, there were added printouts to files for different places in the code. After a while, it was possible to pinpoint where the fault was. Incorrect use of `read()` from a `pipe()` leads to a problem called *Producer-Consumer problem*. The reason for this problem was the way the `read()` was implemented to read from `pipe`. The code required a specific string in order to exit; the string would be received from the pipe. In the case where the pipe did not receive this string and the pipe was emptied, the process would go into a "wait to read state" that could last indefinitely if no new data was written to the pipe. This would happen when the JANUS and SDMSH processes had terminated.

Based on this, `Poll()` was implemented before reading from the pipe. It adds a timeout so that the reading does not wait indefinitely. The code was now able to handle errors in a good way without getting stuck.

6.3.4 Transmission Process - TCP DROP Error

Stemming from what was discovered in section 6.3.3, the underlying problem was still there; where two packets got lost and a printout was printed to the terminal:

> » rx cmd REPORT: DROP 2544
>
> » ERROR: failed to open output stream ’tcp’ with argument ’connect:127.0.0.1:9914’: connecting socket: Connection refused
>
> » ERROR: failed to initialize transmitter.

The first step to solving this problem was to run the `ps -ef` command and track the PID. This made it possible to map in which order the JANUS and SDMSH processes were running. It became clear that some processes did not exit properly, and this led to multiple processes trying to use the same TCP port simultaneously. This made a chain of lost transmissions. The error would occur approximately every 10th transmission between 2 to 4 packets in a row would be lost.

A `stop;` command was added to the SDMSH command line [52]. This significantly improved the TCP problem. When running the SDMSH and JANUS processes for transmission, the problem would now only happen in approximately 1 of 100 transmissions. Also, only 2 packets were lost now compared to the previous 2 to 4 packets in a row.

6.3.5 The 100th Packet Problem

The 100th packet problem was mentioned by the previous group. It was not documented how the transmissions were done when it occurred. An email was sent to the previous group and it was explained that they had transmitted a number starting from 0 and increased with +1 for every transmission. Once the number reached 100 or higher, every other message became corrupted.

The group decided to redo some of the previous calculations of the number of samples. The number of samples in the JANUS packet is needed to use the TCP connection between JANUS and SDMSH. Having a sample value that is too low can lead to parts of the JANUS packet being "cut short" and one would experience corrupt packets. It was uncovered that how the number of samples were calculated in the previously developed code had this problem. The packet would be corrupted if the cargo used 3 bytes or more. That meant the packet was corrupt when the cargo was for example "`100`". Also, the calculation method did not consider that certain characters used 2 bytes instead of 1 byte.

Therefore, a new method was developed for calculating the number of samples. After implementing the new method no corruption would occur. The new method can be found in section 6.4.2.1.

6.3.6 Receiving Process - Cannot Detect Every Other Packet

The next problem was that almost every other packet received was lost when receiving. At this point in the project, the RX commands would establish contact, setting up a new pipe and closing it again when one packet was received. The underlying problem was that multiple processes were trying to run at the same time.

A new solution was made where the SDMSH and JANUS processes were not closed and would operate over multiple receptions. This was done by not closing the SDMSH and JANUS processes and continuing to read from the same pipe over time. Now it was possible to listen over a longer period of time without faults. This new method was an early version of `listenRX()` and can be found in section 6.4.3.

6.3.7 Finding a Way to Stop JANUS and SDMSH

There was a need for closing the JANUS and SDMSH processes for the receiving modem. This was to be able to switch between receiving and transmitting on the same modem. The main problem here was the use of the `execvp()` command to open JANUS and SDMSH. When using the `execvp()` command, the process-image gets switched from C++ to a shell. The consequence of this is that the control over the code will be "lost". It is no longer possible to call an exit command within the shell process. However, the shell will close automatically if the process initialized by the shell closes, this would be the JANUS and SDMSH process.

Knowing this, an attempt was done to find a way of closing both the SDMSH and the JANUS process. In the documentation for the SDMSH and JANUS library, it was not found a closing command. It is however mentioned a stop command in a TODO.txt in the SDMSH code. The group concluded that this had not been implemented in the SDMSH library yet. There was conducted a test to better understand how the JANUS and SDMSH processes worked together. This was done by manually starting the JANUS and SDMSH processes in terminals, as described in section 6.8. There was found a new method for closing both the JANUS and SDMSH by crashing the processes, it was done by running an additional SDMSH process. This method of stopping the processes is implemented in the final code.

6.3.8 Issues With Receiving the First JANUS Packet

After solving the problem "Cannot Detect Every Other Packet" in section 6.3.6, there was still an issue with not always receiving the first packet after a new SDMSH and JANUS process was created. The fault would then occur almost half of the time when receiving. It was also observed that a sound sample was received by the modem, but not decoded by JANUS; meaning that the preamble was not detected.

1. A test with three EvoLogics modems was done to figure out if it was a problem with the receiving or transmitting part of the communication. One of the modems would transmit a JANUS packet and the other two modems would receive it. By doing this test it was discovered that the transmission was not the issue. This was concluded when two modems were set to receiving, where one would receive a packet and one would not receive the same packet.

2. Thereafter, a test where done to rule out if the use of the `stop;` command in SDMSH was causing an issue since this also sets PHY-mode. By testing with and without it, without any difference, it was ruled out.

3. Next test was to redirect the data from SDMSH to a .raw file, where the data stored in the file should contain the error. Then, the .raw file with the error could be sent into the JANUS library and be decoded multiple times. The goal was to exclude the JANUS library. This was done by checking if the error would still occur the same way as before, when using the .raw file instead of TCP. Since the error did not disappear, JANUS was excluded as the problem.

4. The problem had been narrowed down to either SDMSH process or the modem itself. The hypothesis was that the issue had something to do with the initialization of the receiving process or an ACK-process that starts when receiving the first JANUS packet.

5. The last test was changing the threshold level on the modems. Changing the threshold was done by changing the first of four variables in the SDMSH config command. The threshold value was changed from 30 to 8. A change in the threshold value resulted in solving the problem of not always receiving the first packet.

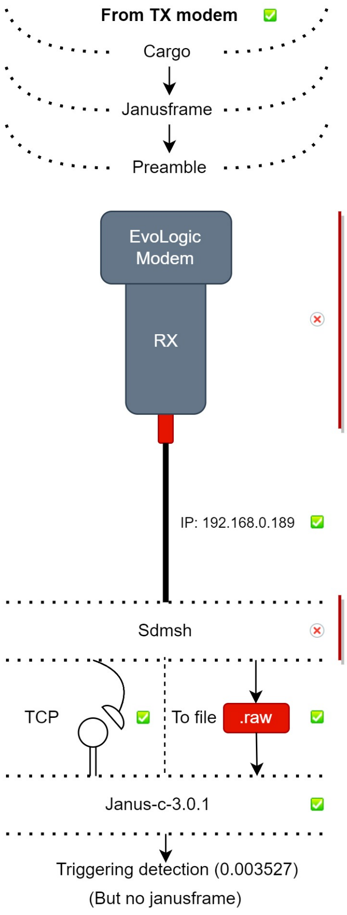


Figure 34: The debugging process method by elimination. Functional parts are identified with a green check mark, while areas with potential issues are marked with a red "X"

6.4 Building the Evo_janusXsdm Library

Adjustments were made to the code developed by the previous group; some parts have been redesigned to work differently, and some parts are new. This section is about how the code was built.

6.4.1 Establishing a Framework for the Library

A new library was built in the existing framework from the previous bachelor group's library. Three functions were kept as they were, but given new names: `sdmConfigAir`, `sdmConfig` and `setPreamble()`. Also the use of a "Constructor" was kept and expanded upon.

6.4.2 Explanation of the Transmission Process

In order for a modem to send a message, two processes must be executed. First, an SDMSH process, where a connection to the modem is established; then, the creation of a TCP socket. The second process is a JANUS process that takes a message from the user as a string and packs it into a JANUS packet as cargo. While the packet is being made, the process connects to the TCP socket of the SDMSH process. The modem can now start a transmission and send the packet.

6.4.2.1 Calculate the Number of Samples

A JANUS packet consists of a certain number of samples. This amount is needed for the TCP connection between the JANUS and SDMSH. The reason behind creating this function can be found in section 6.3.5.

The `getNumberOfSamples` function was developed to calculate the samples needed depending on how many bytes the cargo consisted of. It packs the message string into a JANUS packet and stores the data in a .wav file. It will then read the .wav file. The first 44 bits of this file are skipped, since this is the header in a .wav file. Next, a loop will read the file and count the samples by the size of 16-bit. The function will return the number of samples counted.

6.4.2.2 Calculation of the Reservation Time

The document ANEP-87 describes what the different bits in a particular JANUS frame mean. When attaching a cargo to a JANUS packet, some bits need to be changed so that the receiving modem knows how to read the contents of the cargo. In janus-c-3.0.5, when adding reservation time, it also flips the bit for "Schedule Flag". Reservation time tells the receiver the duration of the cargo; only the duration of the cargo, not the whole JANUS packet. A table for the reservation time can be found in ANEP-87 [57, p. B-1]. The command for adding reservation time in janus-tx is as follows:

```
--packet-reserv-time <reserv_time>
```

`getPacketReservTime()` is a private function in the `Evo_janusXsdm` library. It takes in the number of samples from the function `getNumberOfSamples()`, see section 6.4.2.1. It then calculates the time it would take to read the cargo and returns this value as a float.

The reservation time is calculated by using this formula:

$$\text{samples in cargo} = \text{samples total packet} - \text{samples no cargo} \tag{26}$$

$$\text{Reservation time} = \frac{\text{samples in cargo}}{\text{samples total packet}} \tag{27}$$

Samples in cargo: samples only in the cargo of the packet.
Samples total packet: samples in whole packet.
Samples no cargo: samples in a whole packet without a cargo.

6.4.2.3 Starting the Transmission Process

`startTX()` is a public function that is used to start the transmission process and send a message. A new function was created instead of building on top of the code from the previous bachelor's project. A decision was made to redesign the structure of the multiprocessing such that every process would be reaped properly. The image below illustrates the new structure, indicating the order of the processes through the "PID" label (Figure 35):

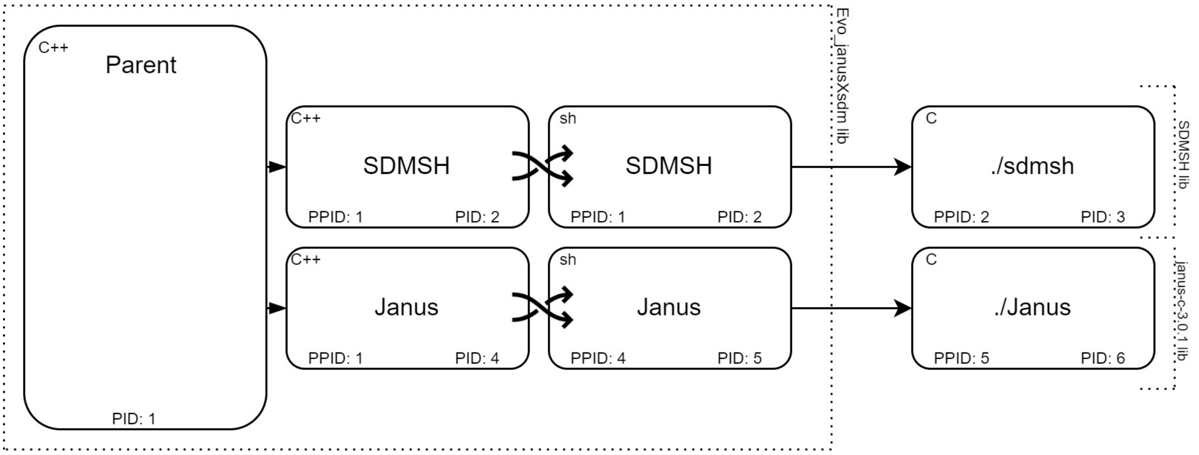


Figure 35: Here are the different processes that are initialized by `startTX` *visualized. It illustrates in what order one could expect the processes to run by looking at the PID. First, the parent will fork twice and create SDMSH and JANUS processes. In the upper left corner one can see what kind of process image is used. In the bottom is the PPID and PID for the given process.*

It starts by running the two functions: `getNumberOfSamples()` describe in section 6.4.2.1 and `getPacketReservTime()` in section 6.4.2.2, and stores the return values for later use. Then `fork()` is used to create two new processes, one for SDMSH and one for JANUS.

The new **SDMSH process** will use the `execvp()` function which will swap out the current process image, which is of the type C++, to the new process image type shell. The new shell process will take in arguments and run those much like one would run an argument in the terminal. The first part of the argument is the path to the folder where the `./sdmsh` i placed. Then an argument is used to run the `./sdmsh`. Here a TCP socket is created which is used for transferring data to the modem by utilizing the SDMSH library.

```
cd  project/lib/sdmsh
./sdmsh  mIP  -e 'stop; tx  NumberOfSamples tcp:listen:127.0.0.1:TX_PORT;
```

Listing 26: In the SDMSH library, run the command to create a TCP socket. This is an example from the developed code, and will not work copied in the terminal without changing Number Of Samples and TX_PORT.

The **JANUS process** is delayed by 500 ms. This is because the SDMSH process needs time to set up the TCP socket. The JANUS process uses the `execvp()` function the same way SDMSH did.

```
cd project/lib/janus-c-3.0.5/bin
./janus-tx --pset-file ../etc/parameter_sets.csv --pset-id 2 --stream-
    driver tcp
--stream-driver-args connect:127.0.0.1: TX_PORT)  --stream-fs stream_fs
--stream-format S16 --packet-reserv-time reserv_time
--verbose 9 --packet-cargo "message";
```

Listing 27: In the janus-c-3.0.5/bin folder, run the command to create a TCP connection to the socket created in Listing 23. This command will send "message" as cargo in this JANUS packet.

- `--pset-file` refers to the file `parameter_sets.csv` where the acoustic frequency specifications is stored and `--pset-id` decides which of these settings to use. The path is from the folder of `./sdmsh`, and not from where the main code is running.
- `--stream-driver` and `--stream-driver-args` sets up the TCP connection from JANUS to SDMSH.
- `--stream-fs stream_fs` sets the sampling frequency.
- `--packet-reserv-time` adds the reservation time for JANUS. Use the time that has been calculated in `getPacketReservTime()` as the argument.
- `--verbose 9` is the information we get in the terminal when creating a JANUS packet. It can be arbitrary, but the "Evo_janusXsdm" library uses "9" to get the right information.
- `--packet-cargo` are the messages written to the JANUS and needs to be a string.

When the **Parent process** has made both forks, it will wait for the JANUS process to exit, with PID 2 (Figure 35). This way, one knows when the transmissions has ended. The Parent uses the `wait(0)` function to wait for the child processes to give an exit signal. When the Parent gets the signal, the Child get reaped. When the Parent has closed both the JANUS and SDMSH process, it exits and returns to where the `startTX()` was called from.

A flowchart has been made to demonstrate this process, see attachment A6.

6.4.3 Explanation of the Reception Possess

In order for a modem to receive a message, two processes must be executed. First, a JANUS process where a TCP socket is created. It then waits for a data input to the socket. The second process is an SDMSH which connects to the TCP socket created by JANUS and redirects its output to it.

The structure of how these processes are executed has been redesigned. It was implemented as a "double fork", mentioned in section 2.3.9. "Double fork" is a technique that helps to cut the connection between a parent process and a child process. The thought behind this was to let the parent process have the freedom to work without interruption. The children will now be reaped by an init process and will not become zombies.

This is illustrated in figure 36 below, where the JANUS and the SDMSH processes are disconnected from the Parent by killing the Dummy process. When this happens, an init process will become the new parent process of SDMSH and JANUS. Init is designed to reap processes and therefore there will not be any zombies.

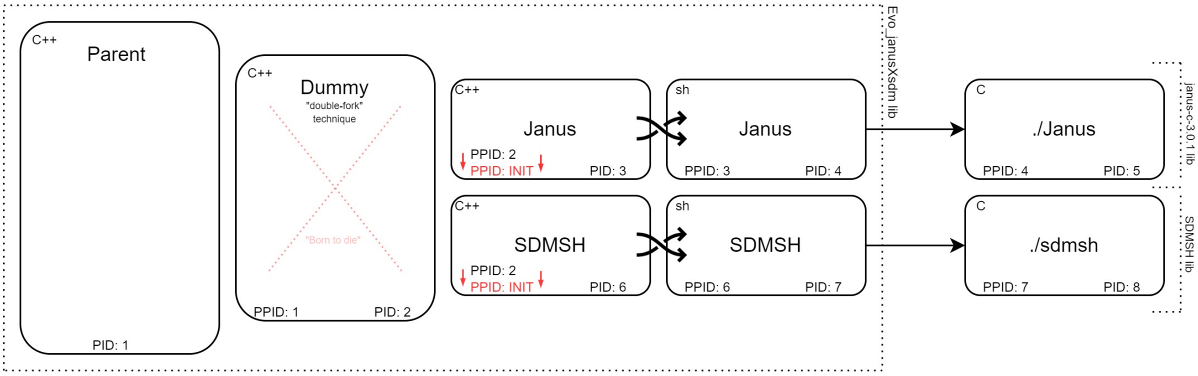


Figure 36: Illustration showcasing the redesigned process structure using the "double fork" technique. The parent process achieves freedom to work by disconnecting itself from the JANUS and SDMSH processes through the death of the Dummy process. The init process assumes the role of the new parent process for SDMSH and JANUS. It will then wait to reap them, and by that, preventing the creation of zombies.

In the library `Evo_janusXsdm`, four functions have been created that are categorized as part of "RX_FAM". The `startRX()`, `listenRX()`, `stopRX()` and `closePipeRX()` functions all fall into this category. These functions are designed to be modular pieces, where each piece initializes a part of the receiving process.

In the `startRX()` function, a similar approach is used to execute JANUS and SDMSH as in the case of the `startTX()` function by using `execvp()`, with a small difference. The output stream of the JANUS process is connected to a pipe, allowing the parent process to read the output generated by JANUS. When the execution of the `startRX()` function is ending, this pipe is returned. This enables the main program to have access to JANUS by handling the pipe. `startRX()` will end as soon as the Dummy process dies.

The arguments in section 28, for the `startTX()` function are the same for the `startRX()`. The big difference from TX is that the `--verbose` is set to "1". This is to gain access to the right amount of information from the JANUS process.

```
./janus-rx --pset-file ../etc/parameter_sets.csv --pset-id 2
--stream-driver tcp --stream-driver-args listen:127.0.0.1:RX_PORT
--stream-fs 250000 --verbose 1
```

Listing 28: In the janus-c-3.0.5/bin folder, run the command to create a TCP socket.

```
./sdmsh  mIP  -e 'stop;rx 0 tcp:connect:127.0.0.1: std::RX_PORT'
```

Listing 29: In the SDMSH library, run the command to create a TCP connection to the socket created in Listing 25. The modem will be ready to receive JANUS packets when this command has been ran.

To read the content of the pipe created, `listenRX()` is used. It takes the pipe and a message string as an argument. It uses pointers for both the pipe and the message string. This enables the `listenRX()` to write to variables outside of the function itself. To read the pipe, the `pull()` function is used to prevent the costumer-producer problem. If there is something in the pipe, it will read from it. When a cargo is found in the JANUS packet, the content of the cargo will be added to the shared message string. If the pipe never gets data, the timeout in `poll()` will end the function and return. In addition to the cargo, it is possible to retrieve more information from the JANUS frame. This is done by returning an array.

Due to the nature of how the `execvp()` function works, it was no good way of closing the JANUS and SDMSH processes, see section 6.3.2. An alternative method was developed, it involved crashing the two processes, the SDMSH and JANUS process, by running a new SDMSH process. A function was made for this called `stopRX()`. The command for this is:

```
./sdmsh  mIP -e 'stop;rx 0 tcp:connect:127.0.0.1:std::RX_PORT'
```

Listing 30: This is a command ran in the SDMSH library to create a TCP connection. If two of these commands is ran right after each other, all receiving processes will be stopped.

The last function in the collection of receiving (RX) commands is `closePipeRX()`. It takes the pipe connected to the JANUS process from earlier, as an argument and closes it.

The `startRX`, `listenRX`, `stopRX` and `closePipeRX` functions makes it possible to build many different types of receiving processes. For instance, there was made a user-friendly function called listenOnceRXsimple, designed specifically to simplify the process for users where they do not need to handle pipes directly. This is how it was built:

```
startRX();
listenRX(fd_pipe,message);
closePipeRX(fd_pipe);
stopRX();
return message
```

Listing 31: Shows how the different functions for the receiving process is used together to create the listenOnceRXsimple() function.

A flowchart that describes the different uses of startRX, listenRX, stopRX and closePipeRX can be found in Attachment A4. An additional flowchart has been made for the listenOnceRXsimple, see attachment A6.

6.5 Communication Setup Between Evologics and Subnero

To make the EvoLogics and Subnero modems able to encode and decode messages from each other, the EvoLogics needed to be edited to the standard of ANEP-87 [57]. This included steps like altering `interleave.c` and `deinterleave.c`, changing sampling frequency, bandwidth and other settings so they could correspond to ANEP-87. Some settings need to be checked on the Subnero modem as well.

A document with different settings between EvoLogics and Subnero was made and can be found in Attachment A2.

6.5.1 Edit Janus-c-3.0.5: Interleave

The files `interleave.c` and `deinterleave.c` from the janus-c-3.0.5 library were altered. The main function `janus_interleave()` in "`janus-c-3.0.5/src/c/janus/interleave.c`" builds a permutation table, see Theory section 2.6.5 for more details. In order for the Subnero modems to recognize the packet, the interleaved order in the JANUS library, used by EvoLogics, needed to follow the ANEP-87 standard. This is due to an error in the ANEP-87 documentation as explained in Theory section 2.6.1.

The interleaved sequence from ANEP-87 was achieved by adding an if-statement in the building of the permutation table. The if-statement will check if the previous number in the permutation table added with the “prime” is greater or equal to the length of the packet. When the if-statement is true, it will give this position in the permutation table the number n. The variable n starts on one and is increased by one every time the if-statement is true.

6.5.2 Edit janus-c-3.0.5: Deinterleave

The `deinterleave.c` code was altered similarly, however the permutation will also be performed in the if-statement and not after. The `janus_deinterleave` function in the `deinterleave.c` file will do the opposite of the interleave. It is coded such that it takes the interleave order from the ANEP-87 and makes it go back to the order the sequence was before the interleaving.

The deinterleaved sequence is achieved by using an if-statement that works almost exactly the same way as the one added to the interleave function. However, it is the index that will be changed, which means that it will be decided by the placement in the permutation table.

6.5.3 Setting Sampling Frequency

The sampling frequency should be set to 250 kHz for the JANUS packet. On EvoLogics, this is done by using the command `--stream-fs` in the command line for `./janus`. The same should be done on Subnero.

For EvoLogics, when altering the sampling frequency, the sampling count for the TCP connection between JANUS and SDMSH will change. If this is not taken into account, it could lead to corrupt packets. A sampling frequency of 250 kHz is implemented in the new library `Evo_janusXsdm` as a default setting.

Sampling frequency needs to satisfy the Nyquist sampling theorem. A f_s of $250kHz$ fulfills this requirement by a good enough margin [56, p. 52].

6.5.4 Acoustic Frequency Specifications

Some of the settings used in this thesis might differ from the ANEP-87 document. It is discussed in section 8.7.1. These settings should match the EvoLogics and Subnero modems.

The acoustic frequency band specifications which are used:

$$F_c = 23040 \text{ [Hz] or } F_{min} = 18880 \text{ [Hz]}$$
$$Bw = 8329 \text{ [Hz]}$$

Frequency slot, width and chip duration:

$$FS_w = 320 \text{ [Hz]}$$
$$Cd = 0.003125 \text{ [s]}$$

On the Evologics, the settings were set in the file `parameter_sets.csv`. The FS_w and Cd will be calculated automatically by the JANUS library.

On the Subnero modem, this setting needs to be set to $f_{min} = 18880$, $hops = 13$ and $f_{step} = 320$. The Bw will be calculated automatically.

6.5.5 Sending a Message/Cargo from EvoLogics to Subnero Using JANUS

On EvoLogics, one needs to set up the configuration and start the transmission process, see section 6.2.2 and 6.4.2. Then write a message as a string and send it.

On Subnero one needs to use the same command as mentioned with Subnero communication in section 5.4.2.2, command 10.

```
phy[3] << new RxJanusFrameNtf(classUserID:16, appData:6)
```

Listing 32: Command for reception.

In order to unpack and read the message, the same actions are required as mentioned in section 5.4.2.2, command 11.

```
new String(ntf)
```

Listing 33: Unpacking the notification message.

6.5.6 Sending a Message/Cargo from Subnero to EvoLogics Using JANUS

On EvoLogics, one needs to set up the config, preamble and start the receiving process. See section 6.2.2 for configurations and section 6.2.3 for setting the reference signal.

On Subnero, one needs to use the same command as with how transmission was done between the two Subnero modems using the JANUS channel. From the section 5.4.2.2, listing: 9.

```
phy << new TxJanusFrameReq(classUserID:16, appData:6, data: "This is 21
    bytes long" as bytes[])
```

Listing 34: Command for transmission from Subnero.

When using TX on Subnero, the reservation time and application data need to be chosen. Reservation time will say how long the duration of the cargo will be. The "appData" tells how long it is. These settings are parameters that belong in the JANUS frame and tell the RX how to read the cargo [57]. If one has the wrong settings, the receiving modem will get no cargo or a corrupt cargo. A guide/table for choosing these parameters has been made, see Attachment A2.

The size of the cargo in a JANUS packet is not linear. This is due to the use of bits in "appData" in the JANUS frame. Those bits are used to inform the receiver about the expected cargo size and therefore it increase in steps.

On the EvoLogics, the "appData" is automatically selected based on the cargo size (the message sent as a string), but the reservation time needs to be calculated. This is implemented in the new library `Evo_janusXsdm`, see section 6.4.2.2.

6.6 Field Test in a Water Tank

The purpose of this test was to assess the performance of the modems in underwater conditions. By conducting a series of tests, where different variables were shifted. The goal was to gain insights and better understand the EvoLogics modems.

The tests were done in a tank at Gløshaugen with the same dimensions as illustrated in figure 37 to the right. The modems were connected and attached to a metal rod as shown on figure 38 below. The setup for cables and the PC was done like in section 3.1.2, but without the switch. There were used two computers where each computer was directly connected to one modem. The code for the receiving modem used an early version of the function `listenRX()`. This means that the code for RX did not reset during the test. The code for TX was the same as `startTX()`.

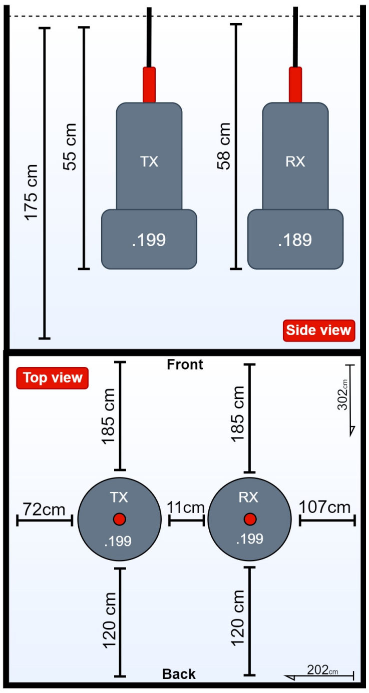


Figure 37: This illustration shows the different measurements used during the main tests. The tank itself is depth: 175 cm, length: 302 cm, width: 202 cm.

There were three main tests where the source level would be altered. For each test, printouts to .txt files were used. This was a way of documenting what was sent and received. For every test run, the TX modem would send a number, starting at zero and increased by one for every transmission.

First, a test with source level 3 was conducted. The group waited for the modem to successfully receive over 100 packets. The same was done with level 2 and 1. However, the sound when conducting the test with source level 1 was unpleasant to hear, enough that the test was canceled after ten transmissions.

There were done some tests where the modems were closer to the side of the tank. No measurements were taken, but it was about 10 to 15 cm from the side of the tank.

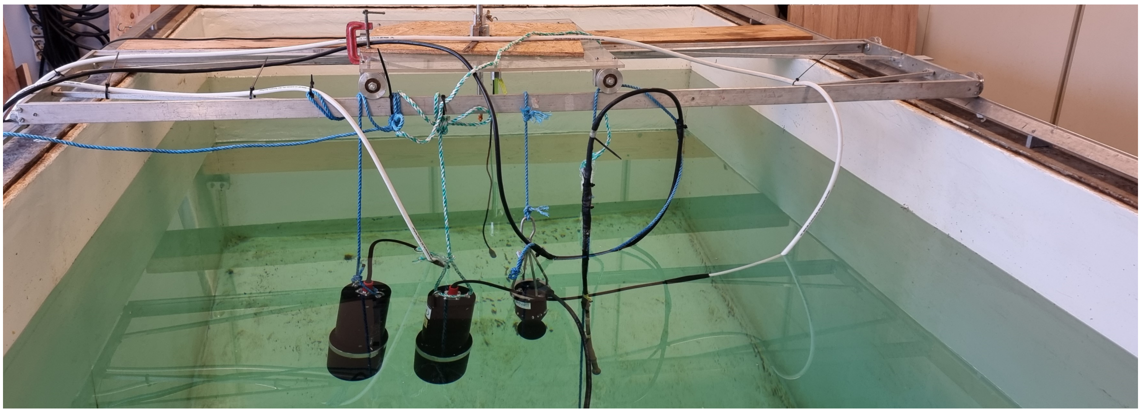


Figure 38: Here is an image of the actual setup for the three main tests conducted.

6.7 Testing In Air - Subnero and EvoLogics, With JANUS

The purpose of these tests was to assess the possibility to communicate with Subnero modems and EvoLogics modems using JANUS. When communication was achieved, the team would continue testing and documenting different settings that needed to be altered in order to send different types of data. By conducting a series of tests where different variables were shifted, we aimed to gain insights and gain a better understanding of the EvoLogics and Subnero modems.

The tests were done in air where the EvoLogics modem was placed between 40 cm to 100 cm from the Subnero modems. The setup for the EvoLogics modem was done as in section 3.1.2. The setup for Subnero modems was done as in section 5.4. Also, the EvoLogics modems were lying on the table while the Subnero modems were standing as shown in figure 40 below.

There were done two tests. First, a much smaller test where the goal was to get one successful transmission between the Subnero and EvoLogics modem. Since it was not done before, the right settings needed to be figured out. The setup for a successful transmission is described in section 6.5.

The second test was the main test. Here the team would test many different settings for the two modems with a focus on JANUS. Many settings described in ANEP-87 were tested and written down. A table was made where settings that worked between the EvoLogics modem and the Subnero modem were noted. The code used for the EvoLogics modems for receiving was an early version of `litenRX()`. This meant that the RX code did not reset during tests. The code for TX was the same as `startTX()`.

For the Subnero modem, commands were done with the browser interface. Commands for JANUS were used as described in section 6.5.5 and 6.5.6.

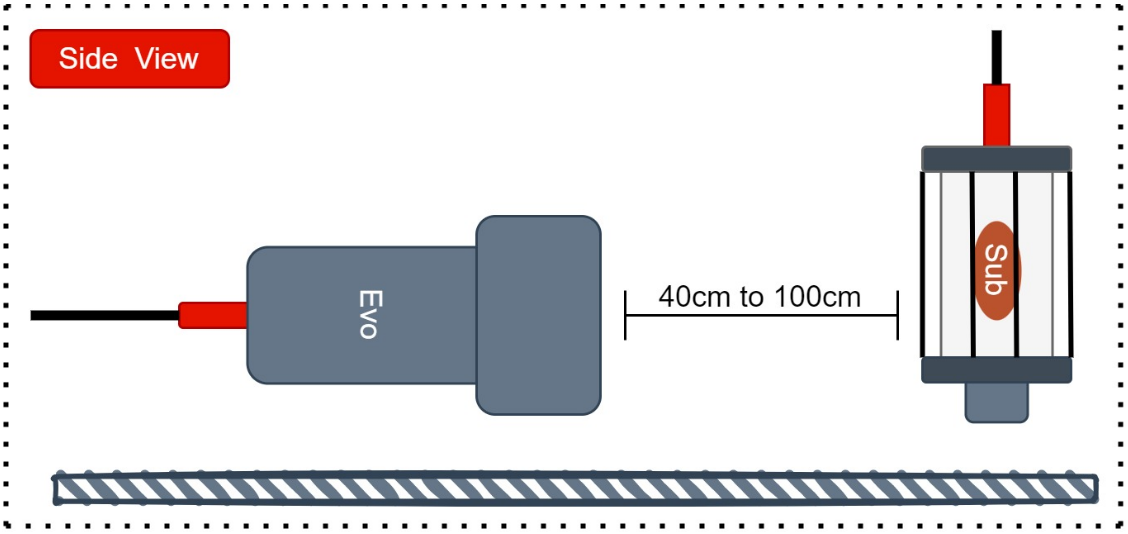


Figure 39: This illustration depicts the setup of the two modems during the testing. The Subnero modem is shown in a vertical orientation, while the EvoLogics modem is shown in a horizontal orientation.

6.8 Test to Find Reliable Termination of JANUS and SDMSH Processes

This test was conducted to find a new way of closing the JANUS and SDMSH processes. It was observed that during the use of the EvoLogics modems, some processes would terminate as a result of running multiple commands in a row. This realization was the reason for conducting the test. The results were used to solve the problem outlined in section 6.3.7.

The tests were conducted in air with two EvoLogics modems, which were placed 10-20 cm apart from each other. The setup for the modems was done as described in section 3.1.2.

Under the test, different commands were written directly to the terminal for both JANUS and SDMSH. These commands are written at the top of this document, see the attachment A3. The commands were for starting new processes for JANUS and SDMSH, where many different combinations of such commands were tested to see if a reliable method of closing could be found.

All results were documented. But, this document only contains the final sequence see attachment A3.

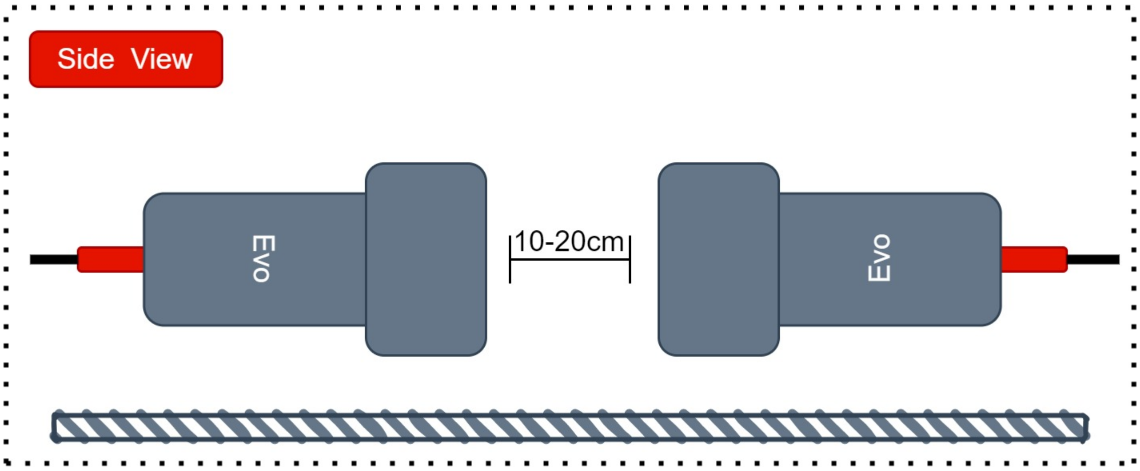


Figure 40: This illustration shows the setup of the two modems during the testing. Both are shown in a horizontal orientation.

6.9 Setup for Raspberry Pi

The group got a Raspberry Pi 4 with the SD-card that the previous group used. The group was not able to get the Pi working and was afraid of ruining the settings and existing code. Therefore, a new SD-card was used and set up from scratch. This was done by following parts of the *3. Implementation* from the bachelor's thesis from the previous project [53, pp. 13-17].

6.9.1 Setup SD-card for Raspberry Pi

When the group tried turning on the Pi and logging in to their user, the group was not able to connect the Pi to internet and was not able to ping it from the laptops. Therefore, the group got a new SD-card and flashed it with Ubuntu Server 22.04 LTS (64-bit) on Raspberry Pi (Headless) with the use of the Raspberry Pi Imager. The SD-card was flashed to get Internet connection from a phone.

The steps explained in the *3. Implementation* section from the previous bachelor's thesis was followed for the network configurations [53, p. 14].

6.9.2 Get Pi Working with The Developed Code

ROS was installed on the Pi by following the same steps as for the virtual machines. Make, Cmake and libreadline was also installed the same way as for the virtual machines, see section 6.1.1. After ROS was installed, the code from GitHub [63] got cloned to the Pi by using the command *git clone LINK*. To make the code run successfully, some alterations had to be done.

FFTW3 Pi Configuration

Since the code cloned from GitHub already had the FFTW3 library installed, it just had to be configured on the Pi.

```
cd lib/fftw/fftw-3.3.10
./configure
make
sudo make install
make check
```

Listing 35: Configuration of the FFTW3 library.

Compiling for SDMSH and JANUS

Before compiling the SDMSH library on the Pi, the SDMSH folder in the `project/lib/` had to be deleted and cloned again.

```
cd ../project/lib/
rm -r sdmsh/
git clone https://github.com/evologics/sdmsh.git
rm -r .git .gitignore .gitmodules
cd sdmsh/
make
```

Listing 36: Reinstalls and compiles the SDMSH library.

Then the Janus-c-3.0.5 library was set up and compiled:

```
cd ../project/lib/janus-c-3.0.5/
cmake -S . -B bin/
cd bin
make
sudo make install
```

Listing 37: In the janus-c.3.0.5 library folder, created a bin folder using Cmake. This bin folder was compiled using sudo make install.

7 Results

7.1 Optimization

7.1.1 Simulation

The gradient and hessian were calculated with the help of the Python library Sympy. The actual derivatives were done automatically with Sympy and the different derivatives got placed in the respected Numpy matrix. This was done for a 3x1 gradient matrix and a 3x3 hessian matrix since three variables are being optimized. Packet loss will not be included in the simulations.

In the simulation with a two-node system, with no float16 restriction and a forced sequence, the interior point method works fine. One forced sequence is as follows: node *i* transmits a packet and node *j* receives that packet and updates its values; then node *j* transmits a packet itself and node *i* receives it and updates its own values. By looking at figure 41, one can see the two nodes first come to an agreement, then later converge together to an optimal point with the help of Newton's method. The jumps in the picture are when the interior point method parameter is increased.

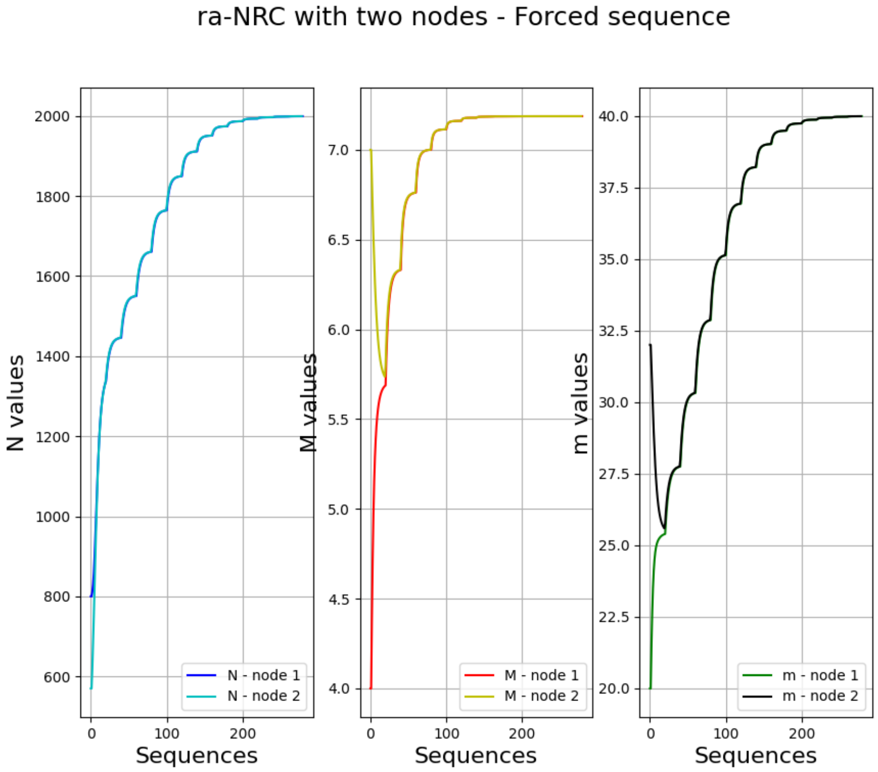


Figure 41: The ideal result of the ra-NRC without any float16 restrictions. The IPM parameter increases in value every 20th forced sequence with an initial IPM parameter value set to 5. The result can be found in python file `ra_nrc.py` *in Attachment B, "Optimization-main-sims (code).zip".*

With the same starting position as in figure 41, one can see the results of the restricted version of the ra-NRC in figure 42. Notice since this has a constant interior point method parameter, the values will converge towards an approximated version of the objective function and will not converge towards the same values as in figure 41. From figure 42 one can see the algorithm works fine for the first 100 sequences or so, then it will start to collapse. The higher the sequence number is, the worse the result. Eventually, the algorithm will break because the σ values in the robust ratio consensus (see section 2.15.8) will get so large, it will break the boundaries of float16. It is also a problem that float16 makes rougher approximations of the σ values when they are nearing the boundaries of float16. It is due to the quantization drawback of float16, the nodes in the restricted ra-NRC never quite seem to come to a consensus. The σ values will increase even more rapidly with a higher IPM parameter and that is why a constant value ($IPMparameter = 1$) is used in the restricted ra-NRC version.

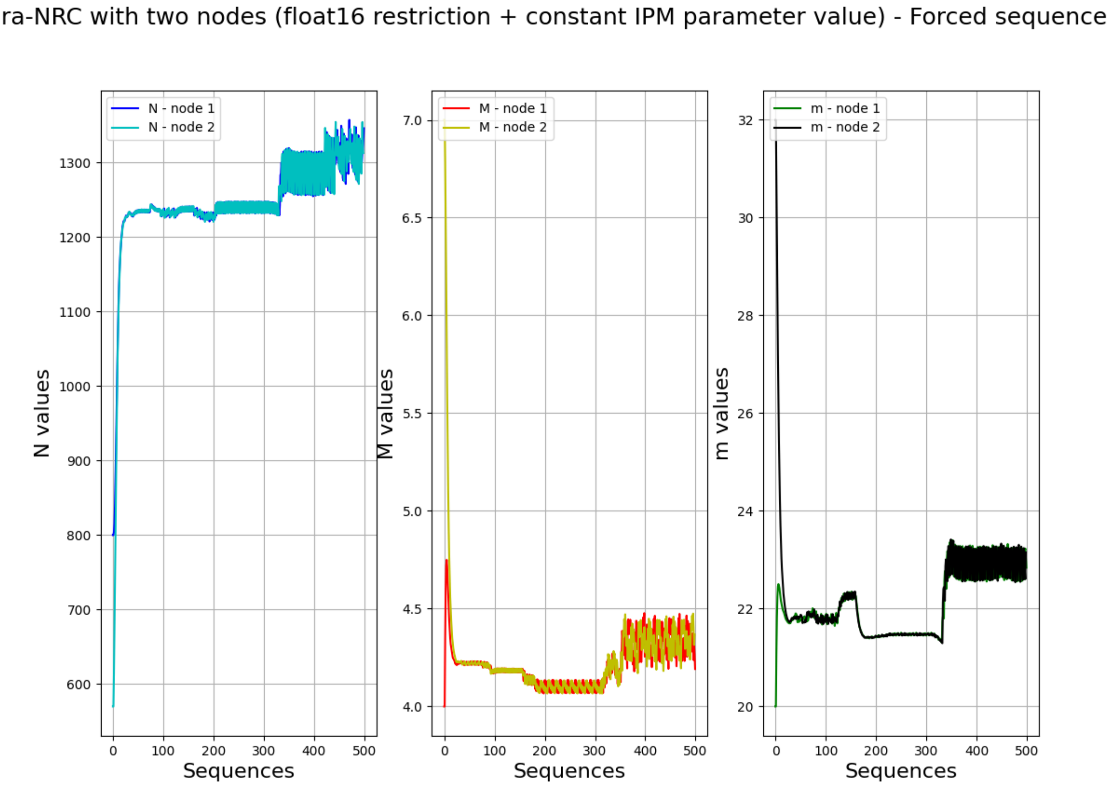


Figure 42: The restricted result of the ra-NRC with float16 restrictions and a constant IPM parameter = 1, using a forced sequence. The result can be found in python file `ra_nrc_r1.py` *in Attachment B, "Optimization-main-sims (code).zip".*

7.2 The Finished ROS 2 Package 'nodecomx'

The resulting ROS 2 package, 'nodecomx', has successfully tested an Unet simulation without any issues. However, any real-world testing with the physical Subnero modems and EvoLogics modems has not been conducted.

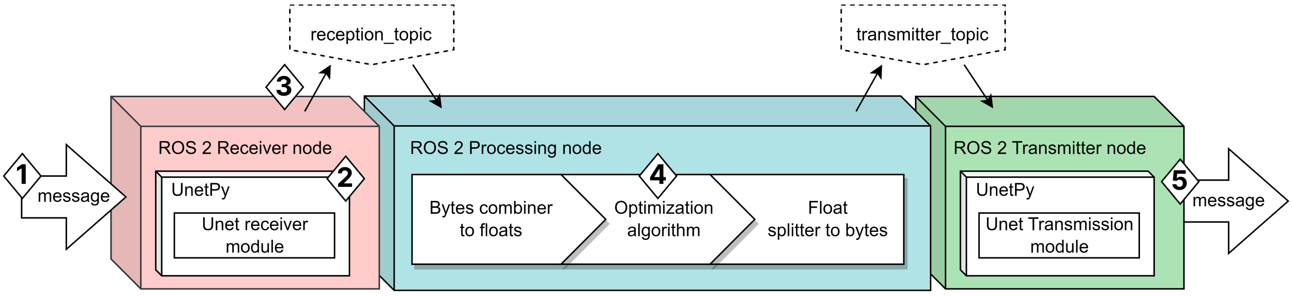


Figure 43: The final setup of the ROS 2 system with UnetPy, optimization algorithm and the float16 converter: Nodecomx. ***1:*** *A message from another modem is incoming.* ***2:*** *The UnetPy module acknowledges and receives the message.* ***3:*** *The message is published to the reception topic.* ***4:*** *The subscribed data from the reception topic is processed before being published to the transmitter topic.* ***5:*** *The subscribed message from the transmitter topic is broadcast.*

For instance, after a message of bytes was received, it is converted to a string before being published to the reception topic. In the processing node (which subscribes to the reception topic), processes the message with the help of the byte merger into float16 datatype. The list of floats is then used in the optimization algorithm where they are updated into new values. These values are then processed by the float splitter into bytes, which are published as a string list to the transmitter node. Within the transmitter node, the data is then broadcast, ready to be received by another modem to repeat this process.

7.2.1 Nodecomx Composition

The finished product *nodecomx* is composed of the following python modules:

- **receiver_node.py**: A ROS 2 node with the UnetPy module, responsible for receiving messages. The module can be found in:
`Attachment B/ROS 2 Project/raw_files/receiver_node.py`
- **transmitter_node.py**: A ROS 2 node with the UnetPy module, responsible for transmitting messages. The module can be found in:
`Attachment B/ROS 2 Project/raw_files/transmitter_node.py`
- **processing_node.py**: A ROS 2 node that processes incoming messages. The module can be found in:
`Attachment B/ROS 2 Project/raw_files/processing_node.py`

- **FP16_converter.py**: A module created to facilitate conversion between float16 and bytes. The module can be found in:
  `Attachment B/ROS 2 Project/raw_files/FP16_converter_node.py`
- **modem_info.py**: A Python file containing information about the used modem. The contents of this file need to match the modem's IP address and port number, as the information is imported by **transmitter_node.py** and **processing_node.py**. The file can be found in:
  `Attachment B/ROS 2 Project/raw_files/modem_info.py`
- **algo.py**: A modified version of the `ra_nrc.py` file, implemented using an object-oriented structure. Some nonessential functions have been disabled to focus on consensus. The module can be found in:
  `Attachment B/ROS 2 Project/raw_files/algo.py`
- **gradient_hessian_calculator.py**: A module for calculating hessian and gradient, extracted from the `ra_nrc.py` file. This module is imported into **algo.py**. The module can be found in:
  `Attachment B/ROS 2 Project/raw_files/gradient_hessian_calculator.py`

7.2.1.1 FP16_converter.py

This module is designed to create bytes from float32 and float16 values and can merge them back into their respective floats. It also allows for potential further development by adding features such as the mixed conversion of float16 and float32 from the same list. The module provides the following callable methods:

- **Converter.FP16_to_int8**: Returns two int8 integers from a float16.
- **Converter.int8_to_FP16**: Returns a float16 from two int8 integers.
- **Converter.FP32_to_int8**: Returns four int8 integers from a float32.
- **Converter.int8_to_FP32**: Returns a float32 from four int8 integers.
- **Converter.FP16_list_to_int8**: Returns a list of int8 integers from a list of float16 floats. The order of integers corresponds to pairs that can be combined into a single float16 value.
- **Converter.int8_list_to_FP16**: Returns a list of float16 floats from a list of int8 integers.
- **Converter.FP32_list_to_int8**: Same as float16 to int8, except there are four bytes for each float element.
- **Converter.int8_list_to_FP32**: Same as int8 to float16, except there needs to be four bytes for each float element.

  The input integers or floats can be provided either as a string or a list.

Although the individual files can be accessed, they will not function on their own outside the established system inside ROS 2. The attachment "`Attachment B/ROS 2 Project/ros2_ws.zip`" contains the ROS 2 workspace and needs to be prepared on a Linux machine before initializing. A quick startup has been implemented for initializing all three nodes at the same time, and can be done by running the following command:

```
ros2 launch nodecomx launch.nodecomx.py
```

Listing 38: Launching of all of the nodes in the nodecomx package.

If the `modem_info.py` file has the correct information; various packages have been installed as mentioned in the methods section; and have been rebuilt in case of modification, then the package should be working.

7.2.1.2 Consensus Results Using UnetStack Simulation in Conjunction With ROS 2 System

A 6-hour test has been conducted using the UnetStack simulation. The system, consisting of the two simulated modems, demonstrated a high degree of similarity in the progression of consensus, with the same initial starting conditions as the single-machine simulation in Figure 44.

It is observed that the quantization of data was only dependent on numpy's float16 datatype, and not the other implemented modules.

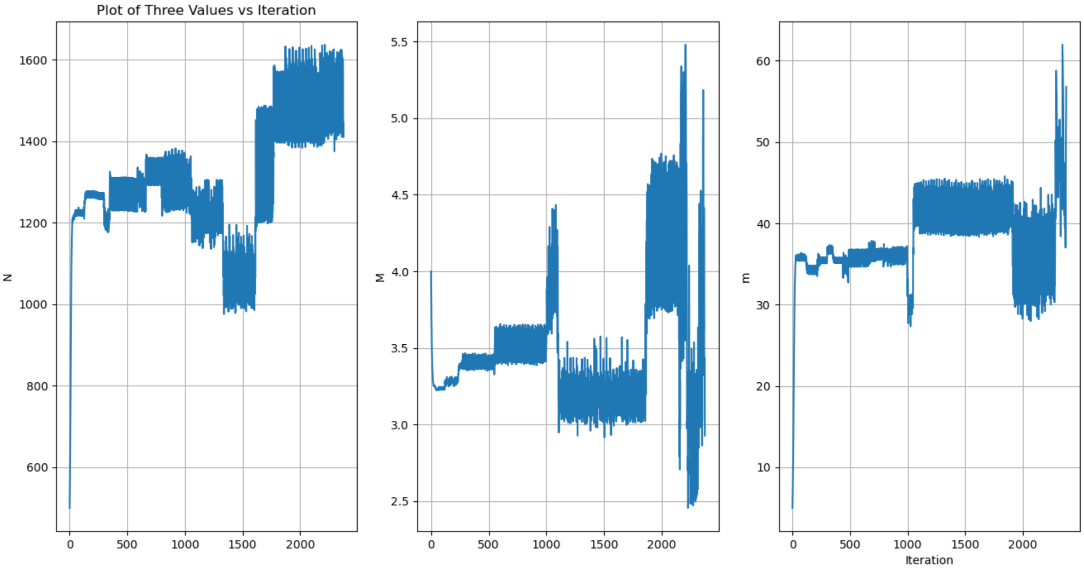


Figure 44: The progression of N, M and m are similar to a simulation absent of UnetStack, ROS 2, and float16 converter. After a 6-hour run, the algorithm crashes due to value overflow as will be mentioned in section 8.1.2.

7.3 The Finished "Evo_janusXsdm" Library

One of the major results of this project is the library Evo_janusXsdm. The library contains files and code for enabling an ease of use of the EvoLogics modems with a JANUS implementation. One of the features is that it allows for an automatic transmission of data between multiple modems, by simplifying the usage of SDMSH and JANUS libraries. Additionally, proper handling of processes has been implemented, and there are no longer any *zombies* present in the operation.

The program is flexible enough to be implemented into ROS 2 without any issues. In addition, the program has also been tested on a Raspberry Pi.

The library class is defined with the name "connection". It has six arguments where fs-stream is the newest:

1. **modemIP**: Contains the acoustic modem IPv4 address.
2. **JANUSPATH**: The path to the JANUS executable code location relative to where the code using the library is placed.
3. **SDMPATH**: The path to the SDMSH executable code location relative to where the code using the library is placed.
4. **rxPort**: The TCP port number for janus-rx and the SMDSH command.
5. **txPort**: The TCP port number for janus-tx and the SMDSH command.
6. **stream_fs**: The sampling frequency for the acoustic modem.

Here is an example on how the arguments can be declared in C++:

```
#include "../lib/janusxsdm/Evo_janusXsdm.cpp"
std::string JANUSPATH = "../lib/janus-c-3.0.5/bin/";
std::string SDMPATH = "../lib/sdmsh/";
int JANUS_RX_PORT = 9938;
int JANUS_TX_PORT = 9955;
string IP = "192.168.0.189";
float fsSTREAM = 250000.0;

Evo_janusXsdm::connection modem(IP, JANUSPATH, SDMPATH, JANUS_RX_PORT,
                                JANUS_TX_PORT, fsSTREAM);
```

Listing 39: Declaring the "connection" class as the object "modem" with different arguments.

7.3.1 Available Functions

Public Functions for Evo_janusXsdm

- **sdmConfigAir()**: Sets the threshold to 8, gain to 0, source level to value 3 (-20 dB) and preamplifier gain to 0 for the acoustic modem.
- **sdmConfig()**: Sets the threshold to 8, gain to 0, source level to the value written in the terminal and preamplifier gain to 0 for the acoustic modem.
- **setPreamble()**: Sets the preamble for JANUS.
- **startTX(message)**: It starts the SDMSH and JANUS processes that belong to TX. The message parameter will be the cargo added to the JANUS packet.
- **startRX()**: Starts the SDMSH and JANUS processes that belong to RX and creates a pipe connected to the JANUS output stream. The function returns the read end of this pipe.
- **listenRX(readJanusPipe, &message)**: Reads from the pipe returned by the `startRX()` function when the pipe contains data. The function will return the data as an array of 4 string elements; cargo, cyclic redundancy check (CRC), cargo size and reservation time. If no data is found in X amount of time, the function will return all elements as "NaN". The cargo will be set in the function and is also available without a return from the message string.
- **stopRX()**: Stops the JANUS and SDMSH processes that belong to RX. The modem will stop listening for JANUS packets if it is placed after `startRX()` and `listenRX(readpipe, &message)`.
- **closePipeRX(&pipe)**: Closes the pipe placed as the parameter of the function.
- **listenOnceRXsimple(&message)**: Listens for one JANUS packet. When it receives a JANUS packet, all reception processes stop running. The function will return the data as an array of 4 string elements; cargo, cyclic redundancy check (CRC), cargo size and reservation time. Cargo will be set in the function and is also available without a return from the function.
- **dummyFlushJanusTX()**: Used for debugging. It will create a TX JANUS process that makes an internal connection to an existing TCP socket, where the socket needs to be created by running `startRX()`. It will then create a JANUS packet and send it.
- **sdmshToRawFile()**: Used for debugging. Starts a receiving process and redirects the data stream to a .raw file.
- **JanusFromRawFile()**: Used for debugging. Reads from a .raw file and unpacks it as a JANUS packet. Will read from the file generated by sdmshToRawFile.

Private Functions for Evo_janusXsdm

- **getNumberOfSamples(message)**: Take in a message as a string. Calculate the number of samples of a JANUS packet with the message as cargo and return it.
- **ToRawFileTX(message)**: Used for debugging. Possible to use inside startTX. It will make a copy of the JANUS packet that is sent and store it in a .raw file.
- **getPacketReservTime(samplesCount)**: Takes in the number of samples from a JANUS packet and returns the reservation time for the packet's cargo.
- **findInJanusFrame(idStr, janusFrame)**: Takes in a string ID and the JANUS frame and returns specific content from the JANUS frame.

The Evo_janusXsdm library can be found in attachment B Evo_JANUSxSDM(code).zip, in file lib/Evo_janusXsdm/Evo_janusXsdm.cpp.

7.3.2 An Implementation of The "Evo_janusXsdm" Library

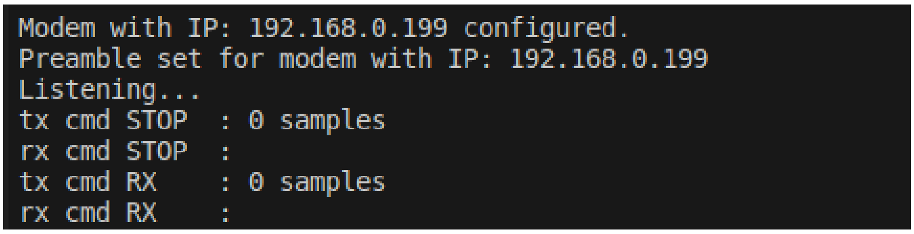
There are included example codes to how the `Evo_janusXsdm` library can be used. This code can be found in the `src` folder in the project, found on the "Evo_janusXsdm" GitHub [63].

The `RXandTX_example.cpp` code shows an example on how the modem can switch between receiving and transmitting the JANUS packets.

```
int main()
{
    //Constructing a connection object
    Evo_janusXsdm::connection modem(IP, JANUSPATH, SDMPATH, JANUS_RX_PORT, JANUS_TX_PORT, STREAMFS);

    //Configures modem for air test and sets preamble
    modem.sdmConfigAir();
    std::this_thread::sleep_for(500ms);          //TODO:Test if sleep is needed
    modem.setPreamble();
    std::this_thread::sleep_for(500ms);          //TODO:Test if sleep is needed

    while(true)
    {
        //listens for a JANUS packet for 30 seconds
        std::array<std::string,4> responsFromFrame = modem.listenOnceRXsimple(responsOnce);
        std::cout << "\n\nMessage: " << responsFromFrame[0] <<" \n" << "CRC (8 bits): " <<responsFromFrame[1]
        <<" \n" "Cargo size: " <<responsFromFrame[2] <<" \n" "Reservation Time: " <<responsFromFrame[3]
        <<"\n"<< std::endl;
        std::this_thread::sleep_for(500ms);       //500ms break between listening and sending

        //sending a JANUS packet with myString as cargo
        std::cout <<"Write a message: " <<std::endl;
        std::getline(std::cin,myString);
        modem.startTX(myString);
        std::this_thread::sleep_for(500ms);    //500ms break between sending and listening

    }
}
```

Listing 40: This code shows an example of how the library "Evo_janusXsdm" can be used for switching between transmitting and receiving. It is done by using the listenOnceRXsimple() and the startTX() function in a loop.

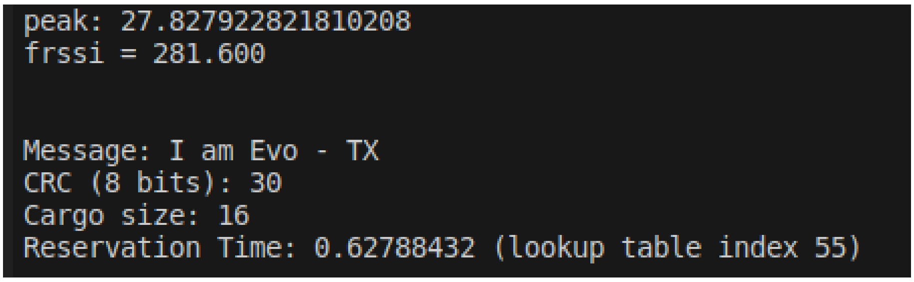
First, a connection object *modem* is created with the constructor parameters defined, as shown in section 7.3. Then the modem gets configured for air test and the preamble will be set. The terminal will print out this message:



```
Modem with IP: 192.168.0.199 configured.
Preamble set for modem with IP: 192.168.0.199
Listening...
tx cmd STOP  : 0 samples
rx cmd STOP  :
tx cmd RX    : 0 samples
rx cmd RX    :
```

Figure 45: The image shows what the terminal looks like when the modem is configured, the preamble is set and ready for receiving.

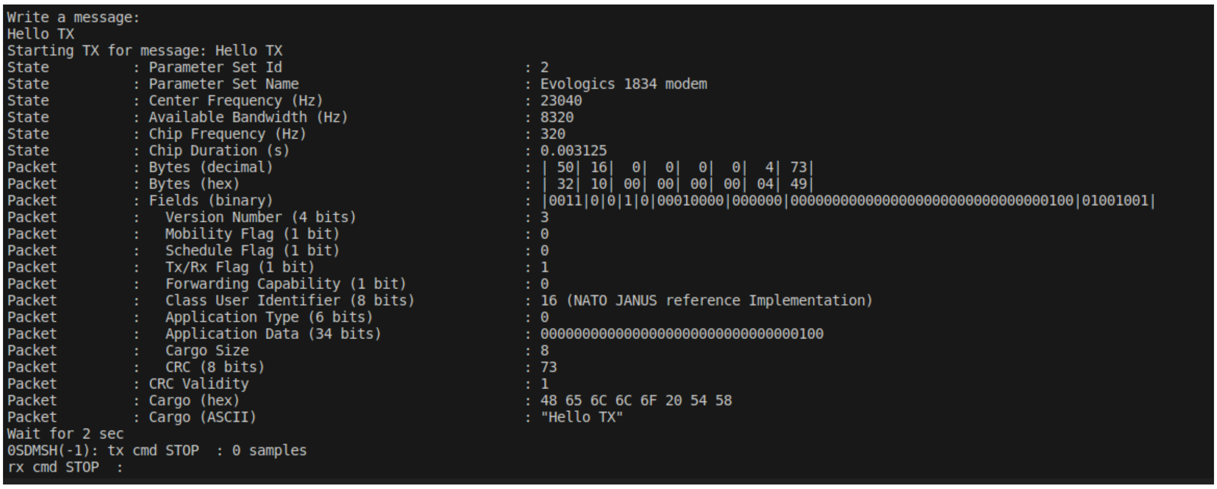
After this, the modem waits for a JANUS packet for 30 seconds. When one is received, parts of the packet will be printed to the terminal. The terminal receive the message "*I am EVO - TX*":



```
peak: 27.827922821810208
frssi = 281.600


Message: I am Evo - TX
CRC (8 bits): 30
Cargo size: 16
Reservation Time: 0.62788432 (lookup table index 55)
```

Figure 46: This is the terminal print from parts of the JANUS frame. It shows the quality of the transmission, by printing `peak` and `frssi` to the terminal. In addition, it shows the `cargo`, CRC, `cargo size` and the `reservation time`.

The terminal will wait for a message to be written before sending it. In this test "*Hello TX*" was sent back.



```
Write a message:
Hello TX
Starting TX for message: Hello TX
State          : Parameter Set Id                             : 2
State          : Parameter Set Name                           : Evologics 1834 modem
State          : Center Frequency (Hz)                        : 23040
State          : Available Bandwidth (Hz)                     : 8320
State          : Chip Frequency (Hz)                          : 320
State          : Chip Duration (s)                            : 0.003125
Packet         : Bytes (decimal)                              : | 50| 16|  0|  0|  0|  0|  4| 73|
Packet         : Bytes (hex)                                  : | 32| 10| 00| 00| 00| 00| 04| 49|
Packet         : Fields (binary)                              : |0011|0|0|1|0|00010000|000000|0000000000000000000000000000000100|01001001|
Packet         :   Version Number (4 bits)                    : 3
Packet         :   Mobility Flag (1 bit)                      : 0
Packet         :   Schedule Flag (1 bit)                      : 0
Packet         :   Tx/Rx Flag (1 bit)                         : 1
Packet         :   Forwarding Capability (1 bit)              : 0
Packet         :   Class User Identifier (8 bits)             : 16 (NATO JANUS reference Implementation)
Packet         :   Application Type (6 bits)                  : 0
Packet         :   Application Data (34 bits)                 : 0000000000000000000000000000000100
Packet         :   Cargo Size                                 : 8
Packet         :   CRC (8 bits)                               : 73
Packet         : CRC Validity                                 : 1
Packet         : Cargo (hex)                                  : 48 65 6C 6C 6F 20 54 58
Packet         : Cargo (ASCII)                                : "Hello TX"
Wait for 2 sec
0SDMSH(-1): tx cmd STOP  : 0 samples
rx cmd STOP  :
```

Figure 47: This is the printout from the JANUS library for a TX process. It shows the information added to the JANUS frame

Then, the modem will begin listening for a JANUS packet again and continue this sequence until the terminal is stopped.

7.4 Result/Finding for EvoLogics Modem

7.4.1 Highlighted Problems Presented by the Previous Bachelor's Group

When the group got the project these problems were highlighted: 100 packets problem, lost packets, and an excessive number of shell processes that run in the background.

The problem where an excessive number of shell processes is run in the background is solved. Now there are no more zombie processes and only the right amount of shell processes running in the background.

The problem where packets would get lost is solved. Now every packet is sent and received without any issues. However, there is a minor issue with the transmitting process: 1 out of 100+ transmissions gets an error where two packets in a row are lost. The problem is a TCP Connection Error, this will be further discussed in section 8.6.5.

The problem with corrupt packets after the 100th transmission is solved. Now every packet is successfully sent and received without any corrupt data. Both when testing in water and in air.

7.4.2 Additional Results for EvoLogics

In section 6.3 a lot of problems related to the EvoLogics have been addressed. All of these have now been fixed.

New methods for closing JANUS and SDMSH have been developed and implemented as parts of the new library `Evo_janusXsdm`. A series of tests were conducted and it appears to work well. See the section 6.8 for the test and attachment for the final results A3.

The `Evo_janusXsdm` library was tested on Raspberry Pi. There was no problem when doing so.

7.4.3 Results From Field Test Conducted In A Water Tank

In the previous group's presentation, it was presented a diagram showing data lost and corrupted packets while doing a field test in a water tank. This test was recreated as written in section 6.6, "Field Test in a Water Tank". The results from the test are:

For the test with source level 1. The test was done from 0 up to 11 transmissions. It was registered 14 received packets, where every number up to 11 was delivered. In addition, duplicates for numbers "0" and "2" were received twice.

For the test with source level 2. The test was done from 0 up to 109 transmissions. It registered 225 received packets, where every number up to 109 was delivered except "34" "35", "91" and "92". These faults were registered as a TCP Connection Error that will be further discussed in section 8.6.5. In addition, many duplicates were received.

For the test with source level 3. The test was done from 0 up to 179 transmissions. It was registered 206 received packets, where every number up to 121 was delivered except "8", "71" and "121". In addition, a lot of duplicates were received.

There were no corrupt packets during these three main tests. The entire text file with these results can be found in attachment B.

7.4.4 Results From Communication Between Subnero and EvoLogics Modems

Communication between Subnero and Evologic using JANUS has been achieved. Both receiving and transmitting have been accomplished in both directions.

Tests were conducted while altering various settings such as `appdata`, `reservationtime` and others. Settings that corresponded between EvoLogics and Subnero were documented. These results can be found in the attached document A2.

Communication between Subnero and EvoLogics using JANUS is now possible without any issues. JANUS packets can be sent with and without cargo, and tests have been carried out up to "appData 7".

8 Discussion

8.1 Optimization

There were some problems getting started because many different elements were unfamiliar to us. Graph theory and optimization in a distributed system were difficult to comprehend. The cost function we worked with was difficult to grasp since there was a considerable amount of theory behind OFDM, and we were not sure what kind of results we should have expected. However, with some help, we managed to get some results and insights for further research.

8.1.1 Constrained Optimization

Before this project, we did not have much knowledge about optimization and we did not know that constraints were something we had to consider. In the beginning, a big challenge was finding out why the optimization algorithm would make the values in the cost function diverge. As we found out, we needed to add constraints to our cost function. We also found out this was a topic with a lot of theory behind it and a much-talked-about topic in optimization. It proved to be a challenging task getting the constraints implemented in the ra-NRC algorithm, due to the float16 restriction.

8.1.2 The Float16 Restriction

We were aware early on in the project that a decrease in the packet size was going to be necessary, but not quite aware of all of the problems this would result in. We thought that the quantization of float16 would make the algorithm a little slower, or that the algorithm maybe would have a small error because of the approximations of the σ values (σ_y and σ_z).

As we found out, because the sigmas increase in size with every transmission, the float16 restriction will make rougher and rougher approximations of the sigmas. This problem is magnified with the interior point method, because of how rapidly the sigmas will increase in size the larger the IPM parameter gets. If a value is broadcast and goes over the float16 boundary, the value will as a result be infinity and cause difficulties.

The inaccurate approximations could easily cause problems if the optimal point of the optimization lay on the boundaries of the constraints of the function. Now, if an approximation is a little inaccurate, the algorithm could break the constraint if the approximation goes slightly in the wrong direction. The algorithm would essentially break since the algorithm would try to calculate the logarithm of a negative number, which is not mathematically possible.

Because of the limited time frame, we have yet to test out how consistent and slow a packet using 32 bit float would be. The reason for choosing float16 is that we think this would be much faster and fewer packet losses would occur. If it was possible to reset the sigmas to fix the problem of them getting too large, maybe it would be possible to use float16 in the transmission.

However, with what we have for now, it would look like the transmission variables would have to be sent as float32.

Originally, the plan for increasing the IPM parameter on all of the nodes was with the built-in clocks of the nodes (either the modems themselves or the Raspberry pies). When a certain time had gone, it was assumed the nodes had come to a consensus, and the IPM parameter would then be increased. More details can be found at *"A Randomized Linear Algorithm for Clock Synchronization in Multi-Agent Systems"* by Bolognani, S. [37].

Because of the problems with float16, we found out we could not get good results using the interior point method. We believe this would work with float32 since it did work to increase the IPM parameter in the simulations every 20th sequence.

8.1.3 Backtracking Line Search

With a limited time frame for testing and other problems that took more priority, the backtracking line search algorithm did not get included in the ra-NRC algorithm. We do not think it is a necessity for the algorithm, but that maybe it would make it better, depending on the cost functions. It was originally implemented for the IPM because we thought this would be a fix for a problem we were having at the time, but as we found out, the method for calculating the hessian/gradient was the cause of the problem. The backtracking line search is something to consider for future work.

8.1.4 Optimal Values of the Different Optimization Parameters

Because of limited testing, no parameter values were set in stone. We do not know the optimal values of ε, the IPM parameter and the IPM parameter multiplier μ. The values used can be considered for future work, but more testing will probably give different results. ε will probably have different optimal values used for different local cost functions, and therefore the backtracking line search could be considered.

A suggestion in the source *"Convex Optimization"* by Boyd, S. P [30, pp. 570-571], is the value $10-20$ for μ, but it seems like it should be smaller in a distributed environment. The suggested starting value for t is not so straightforward and requires some deeper understanding. We have chosen the t values based on trial and error in simulations.

8.1.5 Calculating the Gradient & Hessian

At the start of the project, we used the Python library *numdifftools* to calculate the gradient and hessian. When trying to implement the IPM, we found out the code kept crashing at the boundaries of the constraints. It would look like it converged towards the value as expected, but at the end of the algorithm, it would barely break a constraint. When testing with an arbitrary function, we calculated the gradient and hessian by hand to check if this was the cause of the problem. As we discovered, this made the IPM work. This is why our method for calculating the gradient and hessian is done by calculating the derivatives with the python library *Sympy*, and then manually added in the `Numpy` matrices afterward. It has not entirely been concluded why the other library did not work, since it seems to work until the very end. One of the reasons for this could be that the library did not cooperate well with the `Numpy` library, which was used to calculate Newton's method. Another explanation could be that `numdifftools` did some small approximations which sent the algorithm over the boundaries.

8.1.6 Stopping Criteria

We have not implemented a certain stopping criterion in the ra-NRC algorithm because of other priorities. In practice, one can stop the algorithm after it has run a certain amount of time or after a certain amount of transmissions. Another possible solution could be to check the difference between the local sigmas and the sigmas received from the neighbors. If the IPM is implemented in the ra-NRC algorithm, the stopping criterion probably has to take the IPM parameter into account and stop it after a certain t value. These ideas have not been tested but could be taken into consideration for future work.

8.1.7 Cost Function

The constraints used with the cost function are not final. They were simplified down because the theory behind the cost function was not our priority. For further research, more details can be found at *"Multi-agent algorithms for adaptation of underwater acoustic communication parameters"* by Iadarola, F. [33]. Values used in the cost function were starting suggestions from Emil Wengle, but they were not decisive since we did not get to test the Subnero modems with the ra-NRC algorithm.

8.2 ROS 2

8.2.1 ROS 2 From the Previous Project

The team decided not to continue using the existing ROS 2 environment set up by the previous group. The reason behind this was that a new environment needed to be built to accomplish the implementation and use of the algorithms and different modems. Therefore, it was easier to build a new environment from the ground up.

8.2.2 ROS 2 in the Current Project

Initially, it was difficult to grasp the purpose and usage of ROS 2 for this project. However, its benefits became clear after the scope of the project was more defined. The initial resistance to using ROS 2 stemmed from a lack of understanding regarding its capabilities as a tool. Could the project team achieve the same result without using ROS 2 and rather create their own multi-threaded system for managing communication and algorithm implementation? The answer to this question can only be known if such an alternative approach was attempted. However, it is reasonable to assume that the development time and complexity would have significantly increased, leading to a higher amount of bugs and inefficiencies in the system.

Once there was a solid understanding of ROS 2, working on other tasks became much easier, which is one of the features of ROS 2. However, it is important to acknowledge that there is a learning curve that must be overcome (which can be said about most tools). It is recommended that any future developers invest in a solid understanding of ROS 2 development in order to utilize it to a further extent if required. The recommended approach to learning ROS is by reading the documentation and experimenting/playing with it. While utilizing secondary sources can be helpful, relying solely on them may lead to challenges that are difficult to overcome due to a lack of thorough understanding or inadequate documentation for specific solutions.

8.2.3 The Modularity of Nodecomx

The composition of the *nodecomx* was developed after some initial foresight for flexibility. In this project, the usage of Python is not universal, and the use of C++ language is necessary for the final result with EvoLogics. Even though the complete development and implementation of the optimization algorithm was successful with the UnetStack simulation (and potentially the Subnero modems), the implementation with the EvoLogics was and still is a huge part of this project. Due to the EvoLogics library being written in C++, compatibility was required with the Python-developed optimization algorithm and ROS 2 code. Although there are several solutions for implementing a C++ library into Python, it is of most interest to do that through the use of different ROS 2 nodes; a C++ ROS 2 node.

The benefit of creating a C++ ROS 2 node and using it in combination with Python ROS 2 nodes, is that one can skip using modules like `Ctypes` [76] or `Boost.Python` [77] (although, one may if wished so). The benefits are for usability and readability of code: the ROS 2 package becomes easier to handle due to the interchangeability of nodes. Being able to externally develop the C++ libraries and test them in a C++ ROS 2 node directly, is also a benefit. Using the mentioned modules for C++ implementation to Python may increase the complexity of the package, and that can be avoided by using mixed-language nodes.

Such a setup is achievable because ROS 2 is agnostic in its language when it comes to the intercommunication of nodes [73]. A topic is itself agnostic, and not a "C++ topic" or "Python topic". There are many resources available through user-made contributions to such development online [74] [75]. It is recommended to continue development through these techniques if C++ is to be used for a particular reason.

8.3 Python 3.11

The project has given a fruitful outcome in terms of a deeper understanding of Python and object-oriented programming. The adoption of an object-oriented approach in Python has proven to be enhancing the project's manageability and efficiency. This approach offered several benefits, such as facilitating the creation of isolated modules that maintain a clean code workspace, and providing a straightforward way of structuring different tasks into separate components.

For instance, "objectification" of the developed optimization algorithm allowed for easier navigation and method calling in the `processing_node.py` file. By structuring each method into a class, and moving the initialization variables inside a `__init__` method, allowed the project team to think more clearly about how the program worked; the reduction of clutter allowed for better readability. If something was wrong, a single line of code could be commented out for debugging, if the printed error was not helpful.

In addition to a friendlier development workplace, a crucial benefit is that the handling of these modules by future developers would be much more fluent. It is expected that an understanding of the system will be relatively easier to understand due to the isolation of tasks in their own modules; one module can be developed at a time while maintaining the functionality of the rest of the system, giving the developers an opportunity to grasp the wholeness of the project easier while working.

For any future development, learning of object-oriented programming is crucial for further development of this project. Learning how to design proper classes in Python would leave future development more comfortable in extending or managing the current work.

8.4 Transmitting Floating-point Numbers

One of the goals of the ROS 2 'nodecomx' package was the transmission of a list containing data from and for the optimization algorithm. Transmission of such data was at first attempted with strings, as in a float "42.1984" with `UTF8` encoding, however, it was soon found out that this was not an appropriate formatting due to how many elements were required for the algorithm. The transmitted message, 42.1984, contained in total of seven bytes, even though much longer floats were expected. This was not feasible as there are hard limitations on how much a modem can transmit, since the contents of a message can get very easily corrupted due to noise and interference from other nodes while traversing water.

For this project, the optimization algorithm required a hessian, a gradient and an ID number. In total that is 13 elements, but with the symmetry of the hessian it was reduced to 11; 10 if the ID number is treated as an int8. Due to the optimization algorithm using a float32, a transmission by treating each number as a string would lead to a very large data transfer (as each digit in each number, in each element in a list, would add a byte). This was not a desired preference and a different approach was required, as the expected amount of bytes for a transmission was under 30.

8.4.1 Splitting The Float

In order to address the challenge of transmitting floats with the cargo limitations, the solution involved breaking down the binary representation of a float into four sections and transmitting each section as an 8-bit character. The concept behind this approach is explained in detail in section 5.5.0.1, figure 32.

For example, consider the number 32.42145 represented as a float32, which requires 32 bits of data to be handled by a computer. Instead of attempting to transmit 8 bytes (the `.` is a byte) of data by treating the float as a string and converting each element into a character (1 byte), the float is instead divided into four bytes.

By splitting the float32 into four sections and transmitting them as 8-bit characters, the overall data size required for transmission is reduced. This approach allows for the successful transmission of the float data within the constraints of around a 30 byte packet, without compromising the accuracy or integrity of the original float value.

8.4.2 The Issue With Large Datatypes

The algorithm required the use of a 32 bit datatype, float32, for the precision. The implementation of the algorithm as it was, made the transmission messages larger than what was expected. As mentioned earlier, there has to be a transmission of 10 elements. Even with the float splitting, the resulting size of the cargo would still end up too large, $11 \cdot 4 = 44$ bytes; or rather $10 \cdot 4 + 1 = 41$ bytes if the ID is treated as a single byte. Even though an increase in the `frameLength` on the modems was possible, it was most unwanted due to the increased transmission time that followed along. There was an interest in a further reduction of the transmitted data size.

8.4.3 A Solution To Transmitting Fractions Using Fewer Bytes: Float16

As a continued solution to the size of the cargo, the datatype float16 was introduced into the project. Float16 allowed to halve the cargo of 41 bytes to 21 bytes (the ID as a byte). This reduction of 50% had a drastic effect on the transmission length, which our aim was to reduce. However, float16 did not come without drawbacks, and other issues followed. The discussion on consequences of using float16 has been discussed in the section 8.1.2 "Float16 restriction".

8.4.4 Encoding of he Transmitted and Received Messages

The usage of different computers and modems could have issues regarding encoding. If a modem is not configured to the same encoding as the other modems, then a received or transmitted message could be encoded/decoded improperly by another modem or machine. The same applies to different projects with developed libraries and modules. Inconsistencies in encoding type could be problematic.

In this project the interaction between different modems, different computers, different libraries and different modules is widespread. Due to a late discovery of a potential issue, a manual choice of encoding in the project files has been ignored, due to not having any issues so far. The reasoning for why it might cause issues is due to unknown hardware and software which may interact with the modems and/or the software. As mentioned in the theory section 2.4.3.1 "ASCII and UTF encoding", if the encoding is not specified within a code, then the compiler or programming language will use the default encoding on the machine.

If the hardware is old or not from the same region as the project code was created (Norway), then using the work from this project with such hardware might cause incompatibilities in communication. That is due to ASCII/ANSI encoding. Certain symbols would either have different placements, or some symbols may not exist at all [3]. It is generally an untested area of this project. However, it is mostly a safe assumption that the hardware and software which will be used in conjunction with this project uses the UTF family. It is recommended to make sure that the machine does in fact use the UTF family as its default encoding by checking what the OS is using.

In general, all systems should have UTF-8 encoding installed and available when dealing with this project.

8.5 Subnero Modems

The Subnero modems were available from the beginning, however, their usage was limited by the group's understanding of hardware interaction from an engineering point of view. A lot of assistance has been provided for the utilization of the Subnero modems, and the documentation provided is most useful.

However, the UnetStack API, UnetPy, was difficult to comprehend due to the use of Groovy programming language rather than Python. Almost all of the examples were demonstrated in Groovy, which the project team had no experience or interest in learning in a short amount of time, since the main work was planned to be in Python for both ease of use for the current team and any future teams.

8.5.1 The Subnero Modem Availability

The Subnero modems were crucial for live-testing of the optimization algorithm and ROS 2 'nodecomx' package. However, due to real-life circumstances, these modems were unavailable towards the end of the project when the modules were ready for end-of-development testing, resulting in unverifiable results due to only being able to use the simulated modems.

Although the UnetStack simulation can represent real-life modems well enough for communication testing, it does not provide the necessary JANUS channel in the simulation. In order to implement the optimization algorithm with the possibility of changing the parameters, we needed real modems. However if a future development were to continue with the work, one should look into changing these parameters within the 'nodecomx' package for each iteration of the receive/process/transmit cycle. A method of importing the variables after each iteration would be the first step toward implementation.

```
phy[x].nc = N
phy[x].psk = M
phy[x].blks = m // blks is read-only*
```

Listing 41: "Setting the parameters in python".

Due to the unavailability of testing, it cannot be verified how often or how effective it is to change these parameters. However, it is known that these parameters shall not be setting the JANUS channel, `phy[3]`, but rather either the control channel `phy[1]`, or data channel `phy[2]`. Although the changing of `N` and `M` can be done directly, the changing of `m` is only changeable by setting the `frameLength` of the JANUS channel to a correct value.

8.6 EvoLogics Modems

8.6.1 Recreation of Previous Results

At the start of the project, the group received a task to fix the problems of the previous project's code. After that was done, the next step was to further develop the communication mechanism. However, the group was not able to get the previously developed code to work well. The group experienced problems that were not well documented or not documented at all. This led the group unsure if the problems our team experienced were problems that had happened in the previous bachelor's project also.

It could have been a possibility that we did not implement the code the same way the previous group had, but this is not likely. The code environment was set up and tested as explained in the "3. Implementation" section and the user guide from the previous group [53, pp. 13-17]. The setup guide from the previous project proved to be confusing, as the explanations for all the steps were not sufficiently clear, and there were typos in some of the command lines. However, we were informed that the previous group did not have much time to write the thesis and that they had limited access to the modems during the project period. This could explain why some problems were poorly documented, and several others were not mentioned. As a consequence, the group had to spend time troubleshooting a lot of different problems that were not mentioned in the previous project's report.

It was a challenge to read the developed code from the previous project since the code was not commented as thoroughly as our group would have liked. The use of some key aspects was not talked about in the report at all and was not correctly implemented. Additionally, the results, such as those from the tank test, were not provided to us, except for being discussed in their presentation.

Considering all this, it was difficult to recreate the previous bachelor's project, and what the end product was supposed to be. However, an email was sent to the previous team with some questions, and their reply helped us to understand some of their work better.

8.6.2 Evo_janusXsdm

We creating a new library, as described in section 6.4. We decided to continue using multiprocessing instead of multithreading in the library. This decision was based on our gained understanding of multiprocessing, acquired through the time we spent troubleshooting the previous library. Additionally, since SDMSH is designed to utilize commands like `popen()` for command execution [51], it made sense to stick with multiprocessing. Furthermore, we had ideas on how this approach could help resolve some of the existing issues.

Since the code was developed on top of the existing folders from the previous project, there might still be some fragments of unused code in the new project. Given the extensive troubleshooting process, multiple #include files have been added at the beginning of the library. Some of these might no longer be in use and should be deleted.

8.6.3 Handling New Processes

A big part of the task that needed to be done for the EvoLogics was to manage multiple processes and pipes well and finding a way of closing them. It was implemented proper code and techniques for handling pipes and forks, and it seems to work well. It was also implemented a new way of closing JANUS and SDMSH, this also seems to work well. However, there should be done more testing since there might be bugs that need to be sorted out. We suggest checking the process window after every test using this command in the terminal: `ps -ef`. Observe how many processes there are running at once, and if there are any zombie processes. There has not been a lot of testing specifically on the Raspberry Pi.

A suggestion for future work is to try to "stress test" the new library. This could be done by opening and closing different processes over a time period. Start by running code on the computer and later, when potential bugs have been sorted out, try the same on Raspberry Pi.

8.6.4 Receiving Process - Lowering the Threshold in Configurations

A potential problem was observed one week before the deadline. After lowering the threshold in the modem configurations, which was the solution to the problem described in section 6.3.8, the receiving modem started behaving strangely and stopped receiving. Several tests were conducted where the threshold was set between "0" and "30". It appeared that setting the threshold between "8" and "12" worked well in the air. However, at a threshold of "8" there was one fault after 100+ transmissions.

It was observed that the problem with the first packet reception did not occur when testing in the tank. The threshold was then set to 30. In all of the three main tests, the first packet was always received. This can be because the modems could be better calibrated for transmission in water and the threshold used was one suited for those conditions. Also, the use of a lower source level could also make a difference.

To summarize, for future testing it is recommended to test if there is a difference in water and in air. Also if the source level can play a factor. If more tests were to be done in air, it is recommended to find the balance point between the problems in section 6.3.8 and in this section, where the threshold should be placed between 8 and 30. Use the `listenOnceRXsimple()` to conduct these tests, because it will restart the RX processes every time it receives something.

This is not a major problem though. The threshold should be set to a higher value when testing in water, and a different source level should be used. Also, the problem should rarely occur when testing in air with a threshold below 8.

The `listenOnceRXsimple()` function was made later in the project. This function was affected a lot by the problem talked about in section 6.3.8, "Issues with Receiving the First JANUS Packet", where the threshold was too high. The group was able to get it to work without faults a week before the deadline. Therefore, this function has not been tested much. But it will most likely work fine.

8.6.5 Transmission Process - TCP Connection Error

The *TCP connection error* is a problem on the transmitting modem and was easy to detect because when this error occurred, it was always two packets in a row that were lost. In addition, these errors were printed to the terminal; a `"DROP"` error on the first packet and a `"TCP"` error on the second packet. The *TCP connection error* would rarely occur; we believe that the probability of encountering this issues is approximately 1 in 100 or more transmissions. The reason for this issue was not completely solved.

We have a couple of leads to what could be wrong for the *TCP connection error*:

1. A similar issue is the *TCP DROP error* from section 6.3.4, which had the same symptoms and printouts. It was eliminated by implementing the `stop` command in the SDMSH command line. This was done due to a suspicion that there were still running processes or some buffer that was not emptied on the modem. When the first message got lost, the `"DROP"` error was printed. Then, the second message got lost, and the `"TCP"` error was printed. It is worth checking if there are any processes that fail to exit when they are supposed to.

2. It is possible to trigger the same `"TCP"` error (not the `"DROP"` error) by running commands manually in the terminal for JANUS and SDMSH. If we do not open a socket for the TCP (for TX it is the SDMSH that creates the socket), and only run the command to connect (the JANUS), the same fault would be printed to the terminal.

3. It might have something to do with the PC used, being too slow. It was observed that different computers would have a different frequency of the *TCP connection error*. This could mean that the creation of the TCP socket takes too long to setup.

We recommended checking if the socket has been created for the second TX process when the *TCP connection error* occur. If the first TX processes lets one of its processes live until the next transmission, this could lead to crashing the second process, which is the SDMSH (the socket). A big test has been done to map out how the different processes act in different sequences, found in section 6.8. It is also worth doing the same test again and having a focus on documenting the different errors.

8.6.6 100 Packet And Number of Samples

The previous bachelor's project calculated the samples with a function that increased linearly with the number of characters. The function was developed to work only with a sampling frequency of 96kHz [53, p. 18], and did not take into account that some characters are counted as 2 bytes. It was discovered that the number of samples did not increase linearly by the increase of characters, but rather in steps by the size of cargo (1 byte per cargo). The solution is a new method for calculating the number of samples, as explained in the section "*Calculate the Number of Samples*" 6.4.2.1. The new way of calculating the number of samples increases in steps, works with every character and is not hard-coded, and therefore works with multiple sampling frequencies.

A comparison is added in table 48. When using the old method, data would be lost when it exceeded 3 characters or 3 bytes. Therefore, the cause of the "100 packet problem", see section 6.3.5 for the problem description.

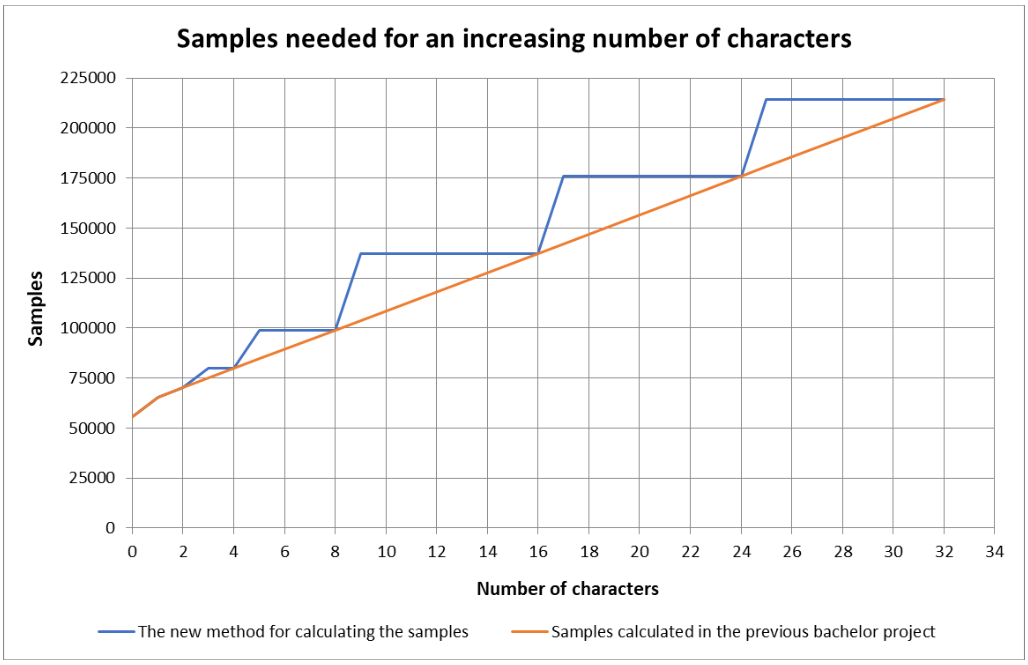


Figure 48: This graph shows the difference between how the number of samples was calculated in the previous Bachelor project (the orange line) and the new method developed by us (the blue line). It is possible to see the reason behind the "100 packet problem", where packets got corrupted when exceeded 3 characters. You can think about it as if some of the samples in a JANUS packet were "cut off".

8.6.7 Field Testing Reflection

When doing the field test it was discovered that if the modems were too close to the walls of the tank, all the packets would be corrupt or lost. There were also too many trigger warnings, due to reflections of the sound from the wall of the tank. When moving the modems to the center of the tank it was possible to do some testing.

When conducting the tests, there were a lot of reflections where a packet could be received multiple times. This is most likely because of the size of the tank. In EvoLogics "Getting Started Guide" [54, p. 20], it is recommended to place the modems in a pool that is 3 x 2 x 2 meters and 1 meter from the surface and the walls. It was not possible to achieve this in the test tank we had access to for executing the tests.

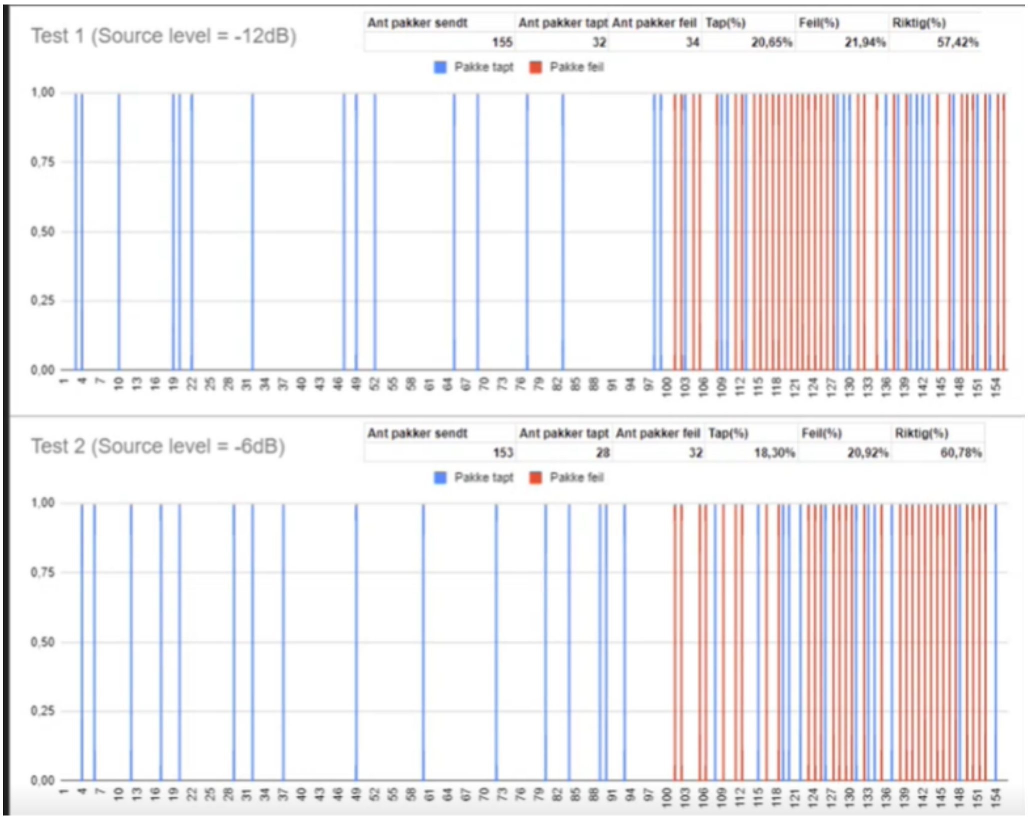
- For the test with source level 1, it was not registered one packet lost or corrupted. It was decided to end it after only 11 transitions because the sound could be heard outside the tank and it was not a pleasant sound.
- For the test with source level 2, there were only two pairs of packets that were not received. This fault was registered as a known TCP error on the transmissions modem, see section 8.6.5. It stops two transmissions before sending, therefore the message was never

received on the RX modem. To summarize, all packets that were sent were also received. There were no packets lost during testing. Also, there were a lot of duplicates.

- For the test with source level 3, three packets were lost out of 179 transmissions. The reason for this occurrence is unknown, but when comparing it to level 2, where no losses were observed, it suggests that setting it to level 3 is on the border of being too weak for transmissions in water. It is also plausible that if two packets were read at the same time it could lead to only one being registered lost.

It is recommended to test in a larger tank in the future.

The diagram below is taken from last year's presentation and illustrates the results of the tank test.



Test 1 (Source level = -12dB)

Ant pakker sendt	Ant pakker tapt	Ant pakker feil	Tap(%)	Feil(%)	Riktig(%)
155	32	34	20,65%	21,94%	57,42%

Test 2 (Source level = -6dB)

Ant pakker sendt	Ant pakker tapt	Ant pakker feil	Tap(%)	Feil(%)	Riktig(%)
153	28	32	18,30%	20,92%	60,78%

Figure 49: This graph shows the last year's results. Blue lines are packets lost and red lines are corrupted packets. Both tests are up to 154 transmissions. The upper one is source level 2 and the one on the bottom is source level 1. The horizontal axes are the number of packets sent and the vertical axes stops at 1 and is just for visualization [65, t. 22:44].

When comparing the last bachelor's project results with our results it is a big improvement. This is because there are no longer corrupt packets and almost no packets are lost. For source level 2, no packets are lost on the receiving modem.

8.7 Transmission between Subnero and EvoLogic

The group managed to achieve successful communication between the Subnero and EvoLogics modems using JANUS. This communication setup was figured out and tested only one day prior to the Subnero modems being sent away, making them inaccessible for future testing. Therefore, the communication is not tested with the newest version of the code.

While conducting the test with Subnero and EvoLogics, we were not able to send a payload with an `appData` larger than 7. Since the test, improvements have been made to the code for EvoLogics, which enables them to use a larger `appData`. This means that a bigger cargo can be added to the JANUS packet. We have not been able to test this between EvoLogics and Subnero, because not having the Subnero modem back before the deadline of this thesis. However, some tests were done between two EvoLogics modems and it worked.

8.7.1 Non-standard Frequency Band

For Subnero and EvoLogics, it was used a bandwidth that differ from the document of ANEP-87, where the bandwidth was set to a higher value. We are unsure if it is defined as a non-standard frequency band or not. It is stated in the standard documentation for ANEP-87 that additional bands may be added to this standard [57, pp. 2-10]. Therefore the bandwidth used in this project might fall under the ANEP-87 standard. It is worth noting that the bandwidth will vary based on the choice of the center frequency, meaning that doubling the center frequency would result in a doubling of the bandwidth as well. If the bandwidth used in this project is divided by 2, it would be the same as in the ANEP-87 standard.

8.8 General

8.8.1 Time Lost Due To Starting with Raspberry Pi

The group spent the beginning part of the project figuring out how to set up the Pi, since it was implied that the setup on the Pi should be ready as it was. A major issue with the Pi was that the internet connection did not work. Given that the team did not have much prior experience with Pi, it was a challenging task.

One of the first steps described in the previous bachelor's projects was the implementation guide to set up the Pi [53, pp. 13-17]. Following these instructions, the group thought it would be easier than what it ended up being setting up the Pi, given the SD card had already the necessary implementations from the previous project. The main issue the group encountered in getting the Raspberry Pi to function was that the code on the Pi was not working. There were several issues, including problems with paths to different libraries.

In the end, the group was able to get a Pi working with a new SD card, and with the new code developed during this bachelor project. It was easier to set up a new one from scratch because the group had a better overview of what was done. That way the group was able to follow most of the same steps as the previous group had written in their thesis to set up the Pi.

Instead of starting out with the Pi, it could have been better to focus mainly on the code itself and making it run on the laptops. This could have spared us a lot of time.

8.8.2 Documentation

There was a focus on writing a detailed description of what was done in this project's thesis. The assumption was that it would be easier to continue working on this project for future developers. Also, make it easier to solve potential problems by using what is written in the section

"Preliminary Work for Evo_janusXsdm Implementation" 6.3 as a history for the EvoLogics modems. Our group recognized the significance of understanding the library, SDMSH and the JANUS processes for the project's progress.

There have also been many tests conducted, and they have all been documented. Some of the test results have been added as attachments. Additionally, throughout the project, a folder called "PA" has been filled with a lot of information, tests, debugging, and other documents that could be useful for future work. This folder will be delivered after the deadline to our supervisor.

8.9 Addressing Research Question

We have implemented a distributed optimization algorithm in a simulation and it should be realizable with the physical modems. The algorithm is only implemented with the Subnero modems. However, Emil Wengle has proposed that it should be possible to use EvoLogics as well.

The algorithm has been successfully tested and implemented on ROS 2. The communication mechanism for EvoLogic has been developed and successfully tested. ROS 2 has been successfully tested in a simulation with Subnero. The Subnero modem has also undergone partial physical testing and should function correctly. ROS 2 has yet to be tested with EvoLogics. However, a solution to solve the implementation of ROS 2 with EvoLogics is in development.

Field testing with EvoLogics has been successfully conducted in a water tank. The entire project in its entirety has yet to be implemented together and tested.

9 Conclusion

We have implemented a distributed optimization algorithm in an Unet simulation, however, it should be realizable with the physical Subnero modems and has been proposed that it should be possible to use the EvoLogics modems as well. The algorithm is based on previous work of the provided pseudocode, and modified for constrained optimization.

A new environment in ROS 2 has been developed for underwater nodes for automated data processing and communication tasks. The ROS 2 package has been created using Python and called *nodecomx*. The optimization algorithm has been implemented with the ROS 2 environment and has been proven to be working with the UnetStack communication mechanism and ROS 2. However, problems were discovered when using the float16 datatype due to quantization.

A notable amount of issues were found in a previous communication mechanism for the EvoLogics modems, and a new solution was developed. The *Evo_janusXsdms* library provides an efficient integration between JANUS and the EvoLogics modems, by a collection of convenient functions for transmitting and receiving. Field testing with EvoLogics has been successfully conducted in a water tank. The communication proved to be effective in water, it was not observed packet loss with a source level stronger than *3*.

Communication between the EvoLogics modems and the Subnero modems has been accomplished by using the ANEP-87 standard. A document has been developed that outlines the specific settings for communication between these modems.

The entire project in its entirety has yet to be implemented together and tested thoroughly. It is suggested to adapt the ROS 2 environment for the use of the *Evo_janusXsdms* library. A further look into the use of float16 is a strong recommendation for the optimization algorithm, due to the reduction of cargo sizes which reduces the transmission time.

Overall, the project has proved to give us valuable insights into the domain of underwater communication, something we have not been exposed to before. We are confident this work will be a stepping stone for the further development of underwater communication and multi-agent systems.

References

[1] Robot Operating System. (n.d.) *ROS 2 Documentation*, docs.ros.org Retrieved: April 14, 2023, from https://docs.ros.org/en/humble/Releases.html

[2] Robot Operating System. (n.d.) *Creating a package*, docs.ros.org Retrieved: May 20, 2023, from https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Creating-Your-First-ROS2-Package.html

[3] Parr, Kealan (March 1, 2021) *everything you need to know about encoding*. freecodecamp https://www.freecodecamp.org/news/everything-you-need-to-know-about-encoding/

[4] The MathWorks, Inc. (n.d.) *What is Half Precision?*. mathworks, Retrieved: May 15, 2023, from https://www.mathworks.com/help/coder/ug/what-is-half-precision.html

[5] Open Robotics. (n.d.) *Ubuntu (Debian)*. docs.ros.org, Retrieved: May 12, 2023, from https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debians.html

[6] Canonical Ltd. (n.d.) *Debian: Debian is the rock on which Ubuntu is built*, https://ubuntu.com/community/governance/debian

[7] Canonical Ltd. (2023) *Install Ubuntu desktop* Retrieved: February 4, 2023, from https://ubuntu.com/tutorials/install-ubuntu-desktop#1-overview

[8] Open Robotics. (n.d.) *The rclpy Package* Retrieved: May 20, 2023, from https://docs.ros.org/en/humble/Concepts/About-ROS-2-Client-Libraries.html#the-rclpy-package

[9] Underwater Networks Handbook. (08.04.2021) *What is a Unet?* https://unetstack.net/handbook/unet-handbook_introduction.html

[10] National University of Singapore (2022). *Downloads: Download UnetStack community edition* Retrieved: April 22, 2023, from https://unetstack.net/#downloads

[11] Chitre, Mandar. (n.d.) *Underwater Networks Handbook* Retrieved: April 20, 2023, from https://unetstack.net/handbook/

[12] VirtualBox, (n.d.) *Chapter 1. First Steps* Retrieved: April 20, 2023, from https://www.virtualbox.org/manual/ch01.html#intro-installing

[13] Canonical, (n.d.) *Chapter 1. First Steps* Retrieved: April 20, 2023, from https://ubuntu.com/tutorials/how-to-run-ubuntu-desktop-on-a-virtual-machine-using-virtualbox#1-overview

[14] EvoLogics GmbH (2020) *UNDERWATER ACOUSTIC MODEMS* https://evologics.de/product/attachments/s2c-modems-brochure-49

[15] Sercan, Sari. (04.11.2022), *The Sleeping Barber Problem* https://www.baeldung.com/cs/sleeping-barber-problem

[16] Shovon, Shahriar, (2019). *Pipe System Call in C* https://linuxhint.com/pipe_system_call_c/

[17] SUBNERO UNDERWATER MODEMS.(July 08, 2022)*Appendix A: Technical Specifications* https://subnero.com/brochures/modem-manual.pdf

[18] Butler, Lloyd. (n.d.) *Underwater radio communication* Retrieved: April 2, 2023, from https://www.robkalmeijer.nl/techniek/electronica/radiotechniek/hambladen/ar/1987/04/page05/index.html

[19] Libretexts, (n.d.) *17.3: Speed of Sound* Retrieved: April 17, 2023, from https://phys.libretexts.org/Bookshelves/University_Physics/Book%3A_University_Physics_(OpenStax)/Book%3A_University_Physics_I_-_Mechanics_Sound_Oscillations_and_Waves_(OpenStax)/17%3A_Sound/17.03%3A_Speed_of_Sound

[20] Heller, Martin (08.07.2022). (n.d.) *What is Visual Studio Code? Microsoft's extensible code editor* https://www.infoworld.com/article/3666488/what-is-visual-studio-code-microsofts-extensible-code-editor.html

[21] Unicode.org (n.d.) *UT8, UTF16, UTF32 & BOM: General questions, relating to UTF or Encoding Form* Retrieved: April 12, 2023, from https://unicode.org/faq/utf_bom.html#gen7

[22] Delmas, Patrice (03.07.2006) *Offset Binary representation (ExcessK)* https://www.cs.auckland.ac.nz/~patrice/210-2006/210%20LN04_2.pdf

[23] broadbandsearch.net (n.d.) *Role of Bytes in Network Data Transmission* Retrieved: April 19, 2023, from https://www.broadbandsearch.net/definitions/byte

[24] Character Encodnig. (n.d.) *ROS 2 Documentation* Obtained: Retrieved: April 12, 2023, from https://studyrocket.co.uk/revision/gcse-computer-science-aqa/written-assessment/character-encoding

[25] Intel, (n.d.) *What Is Hyper-Threading?* Retrieved: April 20, 2023, from https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html

[26] Wong, Kay Jan. (Mar 24, 2022) *Multithreading and Multiprocessing in 10 Minutes* https://towardsdatascience.com/multithreading-and-multiprocessing-in-10-minutes-20d9b3c6a867

[27] Open Robotics. (n.d.) *Distributions.* Retrieved: April 15, 2023, from https://docs.ros.org/en/humble/Releases.html

[28] Steen, M. v. & Tanenbaum, A. S. (2018). *Distributed systems* http://www.dgma.donetsk.ua/docs/kafedry/avp/metod/van%20Steen%20-%20Distributed%20Systems.pdf

[29] Farina, F., Camisa, A., Testa A., Notarnicola I. & Notarstefano G. (2020) *Consensus Algorithms*, Disropt documentation. https://disropt.readthedocs.io/en/latest/api_documentation/disropt.algorithms.consensus.html#

[30] Boyd, S. P. & Vandenberghe, L. (2004) *Convex Optimization*. Cambridge University Press. https://web.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf

[31] JavaTPoint, (n.d.) *What is Deadlock in Operating System (OS)?*. Retrieved: April 10, 2023, from https://www.javatpoint.com/os-deadlocks-introduction

[32] JavaTPoint, (n.d.) *Race Condition in Operating Systems (OS)*. Retrieved: April 10, 2023, from https://www.javatpoint.com/race-condition-in-operating-system

[33] Iadarola, F. (2023) *Multi-agent algorithms for adaptation of underwater acoustic communication parameters*, University of Bologna. https://amslaurea.unibo.it/28412/

[34] Shaw, Anthony, *Your Guide to the CPython Source Code* https://realpython.com/cpython-source-code-guide/#whats-in-the-source-code

[35] freeCodeCamp, (FEBRUARY 10, 2020) *C++ Compiler Explained: What is the Compiler and How Do You Use it?* https://www.freecodecamp.org/news/c-compiler-explained-what-is-the-compiler-and-how-do-you-use-it/

[36] Bof N., Carli R., Notarstefano G., Schenato L. & Varagnolo D. (2019) “Multiagent Newton-Raphson Optimization Over Lossy Network”, *IEEE Trans. Autom. Control*, vol. 64, no. 7, pp. 2983-2990. https://ieeexplore.ieee.org/document/8485728

[37] Bolognani S., Carli R., Lovisari E. & Zampieri S., (July 2016) "A Randomized Linear Algorithm for Clock Synchronization in Multi-Agent Systems", in *IEEE Transactions on Automatic Control*, vol. 61, no. 7, pp. 1711-1726, doi: 10.1109/TAC.2015.2479136. Obtained: 14.05.2023 https://ieeexplore.ieee.org/abstract/document/7270258?casa_token=C44btocuvOEAAAAA:Fsth_bsF0Y8PFhA3tt1vU_daSrI2voaGl--pTDOPMJh6_IfK7sFKkdcOMmaKdOURHJPS2uJpFFg

[38] Grasmair, M. (2016) *Basic properties of convex functions*, NTNU. https://wiki.math.ntnu.no/_media/tma4180/2016v/note2.pdf

[39] HS Devices | EvoLogics [Image]. (n.d). Retrieved: 10.04.2023, from https://evologics.de/acoustic-modem/hs

[40] Underwater Modems - ROMOR [Image]. (n.d). Retrieved: 10.04.2023, from https://romor.ca/underwater-modems/

[41] Raspberry Pi 4 [Image]. (n.d). Retrieved: 10.05.202, from https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

[42] Ermagun, A. & Levinson, D. (2017) *An Introduction to the Network Weight Matrix: Introduction to the Network Weight Matrix*, The University of Sydney. https://www.researchgate.net/publication/318231876_An_Introduction_to_the_Network_Weight_Matrix_Introduction_to_the_Network_Weight_Matrix

[43] University of Rhode Island & Inner Space Center, *History of Underwater Acoustics* Retrieved: 20.05.202, from https://dosits.org/people-and-sound/history-of-underwater-acoustics/

[44] CMRE PAO, (2017) *JANUS, the CMRE underwater communication procol becomes a NATO Standard*, NATO https://www.cmre.nato.int/rockstories-blog-display/398-janus-the-cmre-underwater-communication-protocol-becomes-a-nato-standard

[45] P. Loshin & M. Cobb, (2021) *Secure Shell (SSH)* https://www.techtarget.com/searchsecurity/definition/Secure-Shell

[46] J. Kalasniemi, (2019) *Introduction to the Raspberry Pi*, CERN IdeaSquare https://indico.cern.ch/event/788273/attachments/1777934/3174908/RaspberryPi_Workshop_EPSTIG.pdf

[47] K. Jamieson, (2020) *COS 461 Computer Networks Lecture 4: Hubs, Switches, and Routers*, Princeton University https://www.cs.princeton.edu/courses/archive/fall20/cos461/lectures/lec04-routers.pdf

[48] Ervik et al. (2022, June 3). *NTNU_ROV_COM*. GitHub. Retrieved: 20.05.2023, from https://github.com/markerv/NTNU_ROV_COM.

[49] EvoLogics GmbH. (2021, April 22). *sdmsh*. GitHub. Retrieved: 20.05.2023, from https://github.com/EvoLogics/sdmsh.

[50] EvoLogics Gmbh. (2020, January 9). *sdmsh Wiki, SDM: SDMShell Compile and Run*. GitHub. Retrieved: 20.05.2023, from https://github.com/EvoLogics/sdmsh/wiki/SDM-:-SDMShell---Compile-and-Run.

[51] EvoLogics Gmbh. (2021, May 9). *sdmsh Wiki, sdmsh: Commands and Parameters*. GitHub. Retrieved: 20.05.2023, from https://github.com/EvoLogics/sdmsh/wiki/sdmsh-:-Commands-and-Parameters.

[52] EvoLogics Gmbh. (2020, Jan 9). *sdmsh Wiki, SDM: Protocol Description*. GitHub. Retrieved: 20.05.2023, from https://github.com/EvoLogics/sdmsh/wiki/SDM-:-Protocol-Description.

[53] Eriksen et al. (2022). *Development of Underwater Communication Rig* [Bachelor thesis, Norwegian University of Science and Technology]. https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3002917

[54] EvoLogics. (2021, February). *Getting Started Guide: S2C R 18/34 USBL Underwater Positioning and Communication system*. Obtained from: Supervisor.

[55] EvoLogics. (2020, March). *S2C Reference Manual* (Edition: Standard). Obtained from: Supervisor.

[56] Zappa, G. (2013, 25. March). *README - JANUS Tool Kit 3.0.1*. Retrieved: 20.05.2023, from https://www.januswiki.com/tiki-index.php after making an account on the website.

[57] NATO SANDARDIZATION OFFICE (NSO). (2017). *DIGITAL UNDERWATER SIGNALLING STANDARD FOR NETWORK NODE DISCOVERY & INTEROPERABILITY* (Edition A Version 1). Obtained from: https://nso.nato.int/nso/nsdd/main/standards?search=ANEP-87

[58] Zarki, M. E, (2002) *Introduction to TCP/IP*, Donald Bren School of Information and Computer Sciences. https://www.ics.uci.edu/~magda/ics_x33/ch0.pdf

[59] EvoLogics. (n.d) *EvoLogics AMA*. Retrieved: 21.05.2023, from https://evologics.de/software/ama.

[60] Kendall. G. (2002, 13. January). *Inter-Process Communication Problems* https://www.cs.nott.ac.uk/~pszgxk/courses/g53ops/Processes/proc13-processcommunication.html

[61] Prakash, A. (n.d.) *What is Build Essential Package in Ubuntu? How to Install it?* Retrieved: 22.05.2023, from https://itsfoss.com/build-essential-ubuntu/.

[62] Birkeland, H. B. (n.d.) *Github repository*. Retrieved: 23.05.2023, from https://github.com/HaIvor/Optimization

[63] Fauske, M. F. & Merkesvik, W. B. *NTNU_COM_JANUSxSDM*. Retrieved: 23.05.2023, from https://github.com/Majaafa/NTNU_COM_JANUSxSDM.git.

[64] Shi et al. (2004). Interleaving for Combating Bursts of Errors.*IEEE Circuits and Systems Magazine.* https://web.njit.edu/anl/papers/04CASMag.pdf

[65] NTNU. (2022). Final meeting_20220602_104053.mp4 (sharepoint.com)*IEEE Circuits and Systems Magazine.*[Video]. Teams. https://studntnu-my.sharepoint.com/personal/behdada_ntnu_no/_layouts/15/stream.aspx?id=%2Fpersonal%2Fbehdada%5Fntnu%5Fno%2FDocuments%2FRecordings%2FFinal%20meeting%5F20220602%5F104053%2Emp4&ga=1. (accessed: 24.05.2023)

[66] Chakraborty Arnab (2019, Oct 11). *Init process on UNIX and Linux systems*. tutorialspoint. https://www.tutorialspoint.com/init-process-on-unix-and-linux-systems

[67] Academic tutorials. (2008). *Process Creation*. http://www.academictutorials.com/ipc/ipc-process-creation.asp

[68] ibm. (2021-03-03). *exec functions*. https://www.ibm.com/docs/en/zos/2.3.0?topic=functions-exec#rtexe

[69] The Regents of the University of California. (2021-03-22). *popen(3) Linux manual page*. https://man7.org/linux/man-pages/man3/popen.3.html

[70] The Regents of the University of California. (2021-03-22). *pipe(2) — Linux manual page*. https://man7.org/linux/man-pages/man2/pipe.2.html.

[71] Chen, Miles MH Chen (2009). *double fork to avoid zombie process*. http://thinkiii.blogspot.com/2009/12/double-fork-to-avoid-zombie-process.html.

[72] JavaTPoint, (n.d.) *Difference between fork() and exec()*. Retrieved: 05.19.2023 , from https://www.javatpoint.com/fork-vs-exec

[73] The Robotics Back-End, (n.d.), *When to use Python vs Cpp with ROS*. Retrieved: 05.26.2023, from https://roboticsbackend.com/python-vs-cpp-with-ros/

[74] The Robotics Back-End, (n.d.) *How to Add a Python ROS2 Node to a C++ ROS 2 Package*. Retrieved: 05.26.2023, from https://automaticaddison.com/how-to-add-a-python-ros2-node-to-a-c-ros-2-package/

[75] JavaTPoint, (2015) *Boost.Python*. Retrieved: 05.26.2023, from https://roboticsbackend.com/ros2-package-for-both-python-and-cpp-nodes/

[76] Kravets, Alexey. (Nov 28, 2021) *Calling C++ code from Python with ctypes module*. Retrieved: 05.26.2023, from https://towardsdatascience.com/calling-c-code-from-python-with-ctypes-module-58404b9b3929

[77] JavaTPoint, (2015) *Boost.Python*. Retrieved: 05.26.2023, from https://www.boost.org/doc/libs/1_80_0/libs/python/doc/html/index.html

[78] JavaTPoint, (n.d.) *What is zombie process* Retrieved: 05.19.2023. From https://www.javatpoint.com/what-is-zombie-process

[79] Raspberry Pi Trading Ltd, (June 2019) *Boost.Python*. Retrieved: 05.26.2023, from https://static.raspberrypi.org/files/product-briefs/Raspberry-Pi-4-Product-Brief.pdf

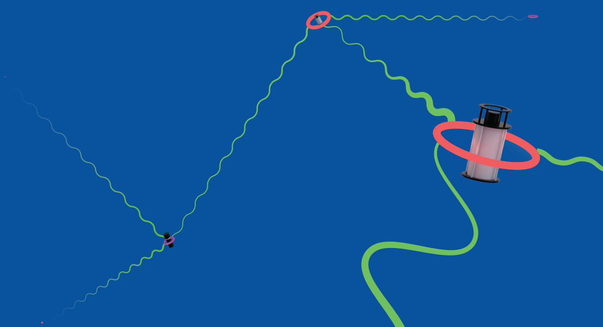
Attachment

A Attachments

NTNU
Faculty of Information Technology
and Electrical Engineering
Department of Engineering
Cybernetics

Halvor Bergstøl Birkeland
Ovidijus Cironka
Maja Austin Fauske
Wilhelm Merkesvik

Underwater communication is about sending and receiving information through water. However, underwater communication faces challenges due to water's properties. With its high density compared to air, underwater communication demands smart solutions. Enter acoustic communication, a reliable method of using sound waves to send and receive messages under water.



This project focuses on making a prototype of an adaptive communication mechanism to address the challenges in real-life scenarios. The optimization of the adaption of underwater modems consists of several key-aspects.

The ra-NRC optimization algorithm has been realized for real-world usage. This algorithm calculates incoming values based on the knowledge of multiple, distant, nodes. The constant talking between these distant nodes leads them to agree on the best values that fits for every node.

Evo_janusXsdm is a program for managing communication with EvoLogics modems. The program integrates with JANUS, the standard for underwater communication defined by NATO, enabling communication between various systems, including the Subnero modems.

Nodecomx, a program which aims to combine the algorithm and the communication programs, allowing a smooth transition of information coming in and out of each node.

These two underwater communication devices talk to eachother using the JANUS protocol. This protocol can be thought of as a languge they use to communicate.

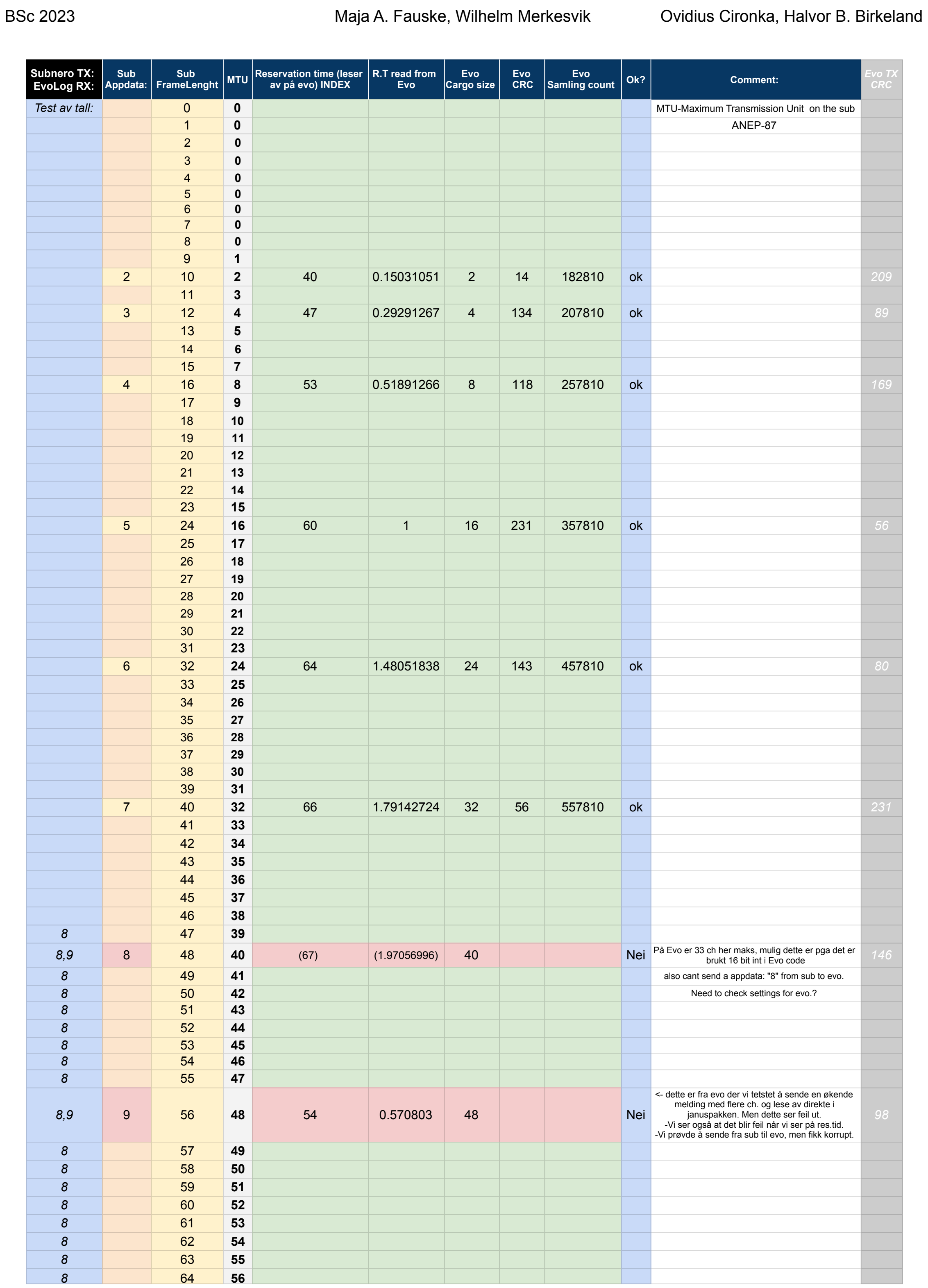
This project is part of a larger project, which can be viewed through this QR code.

Our efforts have resulted in the implementation of a significant amount of useful software tools that can greatly benefit further development in the field of underwater communication.

Overall, this project has provided us with valuable insights and exposure to new learning topics. We are confident that it will serve as a stepping stone for the development of real-life underwater communication.

Distributed Optimization-Based Adaptive Underwater Communication Schemes

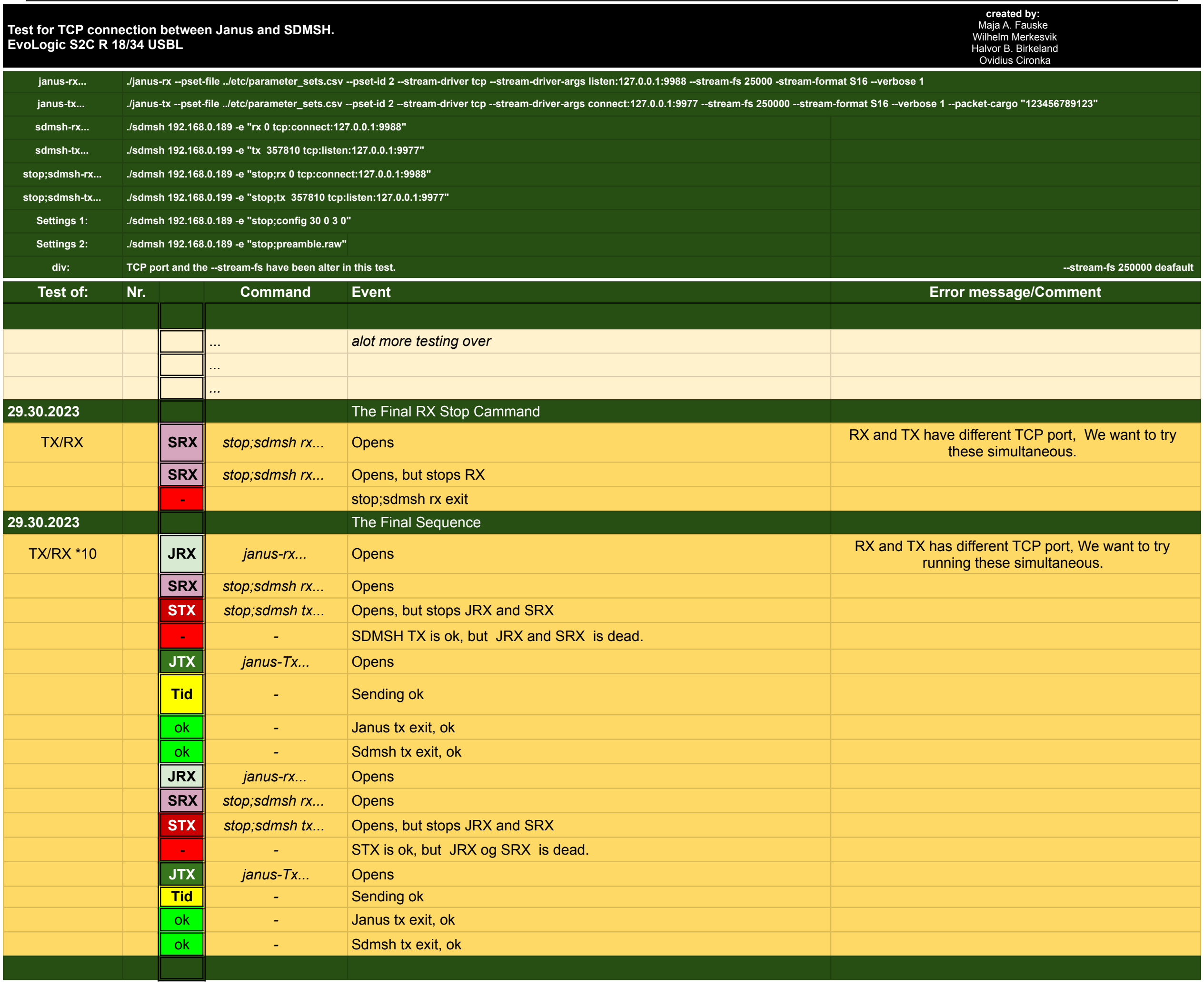
25.05.2023



BSc 2023 Maja A. Fauske, Wilhelm Merkesvik Ovidius Cironka, Halvor B. Birkeland

Subnero TX: EvoLog RX:	Sub Appdata:	Sub FrameLenght	MTU	Reservation time (leser av på evo) INDEX	R.T read from Evo	Evo Cargo size	Evo CRC	Evo Samling count	Ok?	Comment:	Evo TX CRC
Test av tall:		0	**0**							MTU-Maximum Transmission Unit on the sub	
		1	**0**							ANEP-87	
		2	**0**								
		3	**0**								
		4	**0**								
		5	**0**								
		6	**0**								
		7	**0**								
		8	**0**								
		9	**1**								
	2	10	**2**	40	0.15031051	2	14	182810	ok		*209*
		11	**3**								
	3	12	**4**	47	0.29291267	4	134	207810	ok		*89*
		13	**5**								
		14	**6**								
		15	**7**								
	4	16	**8**	53	0.51891266	8	118	257810	ok		*169*
		17	**9**								
		18	**10**								
		19	**11**								
		20	**12**								
		21	**13**								
		22	**14**								
		23	**15**								
	5	24	**16**	60	1	16	231	357810	ok		*56*
		25	**17**								
		26	**18**								
		27	**19**								
		28	**20**								
		29	**21**								
		30	**22**								
		31	**23**								
	6	32	**24**	64	1.48051838	24	143	457810	ok		*80*
		33	**25**								
		34	**26**								
		35	**27**								
		36	**28**								
		37	**29**								
		38	**30**								
		39	**31**								
	7	40	**32**	66	1.79142724	32	56	557810	ok		*231*
		41	**33**								
		42	**34**								
		43	**35**								
		44	**36**								
		45	**37**								
		46	**38**								
8		47	**39**								
8,9	8	48	**40**	(67)	(1.97056996)	40			Nei	På Evo er 33 ch her maks, mulig dette er pga det er brukt 16 bit int i Evo code	*146*
8		49	**41**							also cant send a appdata: "8" from sub to evo.	
8		50	**42**							Need to check settings for evo.?	
8		51	**43**								
8		52	**44**								
8		53	**45**								
8		54	**46**								
8		55	**47**								
8,9	9	56	**48**	54	0.570803	48			Nei	<- dette er fra evo der vi tetstet å sende en økende melding med flere ch. og lese av direkte i januspakken. Men dette ser feil ut. -Vi ser også at det blir feil når vi ser på res.tid. -Vi prøvde å sende fra sub til evo, men fikk korrupt.	*98*
8		57	**49**								
8		58	**50**								
8		59	**51**								
8		60	**52**								
8		61	**53**								
8		62	**54**								
8		63	**55**								
8		64	**56**								

A3 Attachments: Final sequence reliable termination of JANUS & SDMSH

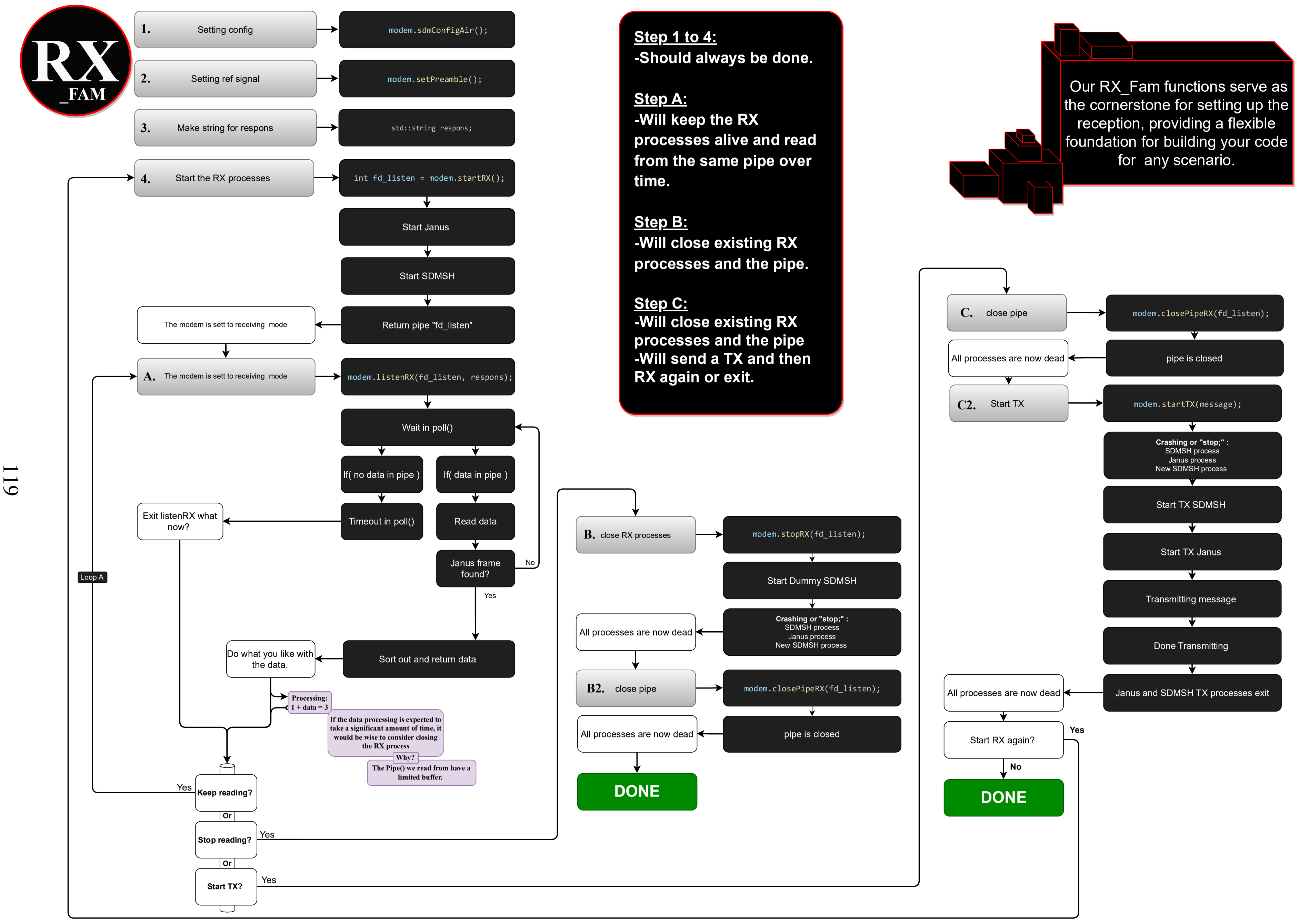


Test for TCP connection between Janus and SDMSH.
EvoLogic S2C R 18/34 USBL

created by:
Maja A. Fauske
Wilhelm Merkesvik
Halvor B. Birkeland
Ovidius Cironka

janus-rx...	./janus-rx --pset-file ../etc/parameter_sets.csv --pset-id 2 --stream-driver tcp --stream-driver-args listen:127.0.0.1:9988 --stream-fs 25000 -stream-format S16 --verbose 1	
janus-tx...	./janus-tx --pset-file ../etc/parameter_sets.csv --pset-id 2 --stream-driver tcp --stream-driver-args connect:127.0.0.1:9977 --stream-fs 250000 --stream-format S16 --verbose 1 --packet-cargo "123456789123"	
sdmsh-rx...	./sdmsh 192.168.0.189 -e "rx 0 tcp:connect:127.0.0.1:9988"	
sdmsh-tx...	./sdmsh 192.168.0.199 -e "tx 357810 tcp:listen:127.0.0.1:9977"	
stop;sdmsh-rx...	./sdmsh 192.168.0.189 -e "stop;rx 0 tcp:connect:127.0.0.1:9988"	
stop;sdmsh-tx...	./sdmsh 192.168.0.199 -e "stop;tx 357810 tcp:listen:127.0.0.1:9977"	
Settings 1:	./sdmsh 192.168.0.189 -e "stop;config 30 0 3 0"	
Settings 2:	./sdmsh 192.168.0.189 -e "stop;preamble.raw"	
div:	TCP port and the --stream-fs have been alter in this test.	--stream-fs 250000 deafault

Test of:	Nr.		Command	Event	Error message/Comment
			...	*alot more testing over*	
			...		
			...		
29.30.2023				The Final RX Stop Cammand	
TX/RX		SRX	*stop;sdmsh rx...*	Opens	RX and TX have different TCP port, We want to try these simultaneous.
		SRX	*stop;sdmsh rx...*	Opens, but stops RX	
		-		stop;sdmsh rx exit	
29.30.2023				The Final Sequence	
TX/RX *10		JRX	*janus-rx...*	Opens	RX and TX has different TCP port, We want to try running these simultaneous.
		SRX	*stop;sdmsh rx...*	Opens	
		STX	*stop;sdmsh tx...*	Opens, but stops JRX and SRX	
		-	-	SDMSH TX is ok, but JRX and SRX is dead.	
		JTX	*janus-Tx...*	Opens	
		Tid	-	Sending ok	
		ok	-	Janus tx exit, ok	
		ok	-	Sdmsh tx exit, ok	
		JRX	*janus-rx...*	Opens	
		SRX	*stop;sdmsh rx...*	Opens	
		STX	*stop;sdmsh tx...*	Opens, but stops JRX and SRX	
		-	-	STX is ok, but JRX og SRX is dead.	
		JTX	*janus-Tx...*	Opens	
		Tid	-	Sending ok	
		ok	-	Janus tx exit, ok	
		ok	-	Sdmsh tx exit, ok	



RX _FAM

Our RX_Fam functions serve as the cornerstone for setting up the reception, providing a flexible foundation for building your code for any scenario.

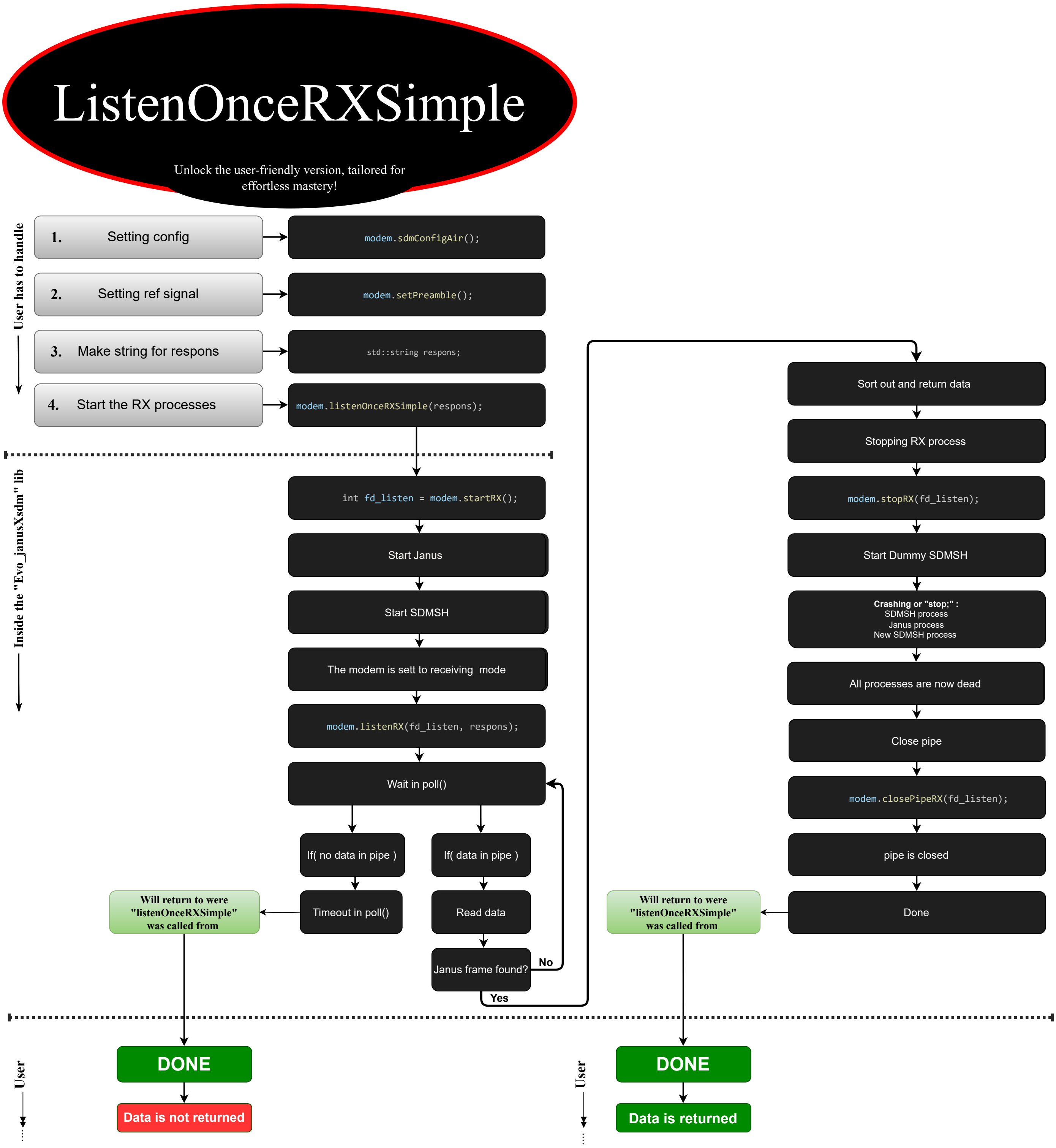
Step 1 to 4:
-Should always be done.

Step A:
-Will keep the RX processes alive and read from the same pipe over time.

Step B:
-Will close existing RX processes and the pipe.

Step C:
-Will close existing RX processed and the pipe
-Will send a TX and then RX again or exit.

1. Setting config
modem.sdmConfigAir();
2. Setting ref signal
modem.setPreamble();
3. Make string for respons
std::string respons;
4. Start the RX processes
int fd_listen = modem.startRX();
Start Janus
Start SDMSH
Return pipe "fd_listen"
The modem is sett to receiving mode
A. The modem is sett to receiving mode
modem.listenRX(fd_listen, respons);
Wait in poll()
If(no data in pipe)
If(data in pipe)
Timeout in poll()
Read data
Exit listenRX what now?
Janus frame found?
No
Yes
Sort out and return data
Do what you like with the data.
Processing: 1 + data = 3
If the data processing is expected to take a significant amount of time, it would be wise to consider closing the RX process
Why?
The Pipe() we read from have a limited buffer.
Loop A
Yes
Keep reading?
Or
Stop reading?
Yes
Or
Start TX?
Yes
B. close RX processes
modem.stopRX(fd_listen);
Start Dummy SDMSH
Crashing or "stop;" : SDMSH process Janus process New SDMSH process
All processes are now dead
B2. close pipe
modem.closePipeRX(fd_listen);
pipe is closed
All processes are now dead
DONE
C. close pipe
modem.closePipeRX(fd_listen);
pipe is closed
All processes are now dead
C2. Start TX
modem.startTX(message);
Crashing or "stop;" : SDMSH process Janus process New SDMSH process
Start TX SDMSH
Start TX Janus
Transmitting message
Done Transmitting
Janus and SDMSH TX processes exit
All processes are now dead
Start RX again?
Yes
No
DONE



ListenOnceRXSimple

Unlock the user-friendly version, tailored for effortless mastery!

User has to handle

1. Setting config
```
modem.sdmConfigAir();
```
2. Setting ref signal
```
modem.setPreamble();
```
3. Make string for respons
```
std::string respons;
```
4. Start the RX processes
```
modem.listenOnceRXSimple(respons);
```

Inside the "Evo_janusXsdm" lib

```
int fd_listen = modem.startRX();
```
Start Janus

Start SDMSH

The modem is sett to receiving mode
```
modem.listenRX(fd_listen, respons);
```
Wait in poll()

If(no data in pipe)

Timeout in poll()

Will return to were "listenOnceRXSimple" was called from

If(data in pipe)

Read data

Janus frame found?

No

Yes

Sort out and return data

Stopping RX process
```
modem.stopRX(fd_listen);
```
Start Dummy SDMSH

Crashing or "stop;" :
SDMSH process
Janus process
New SDMSH process

All processes are now dead

Close pipe
```
modem.closePipeRX(fd_listen);
```
pipe is closed

Done

Will return to were "listenOnceRXSimple" was called from

User

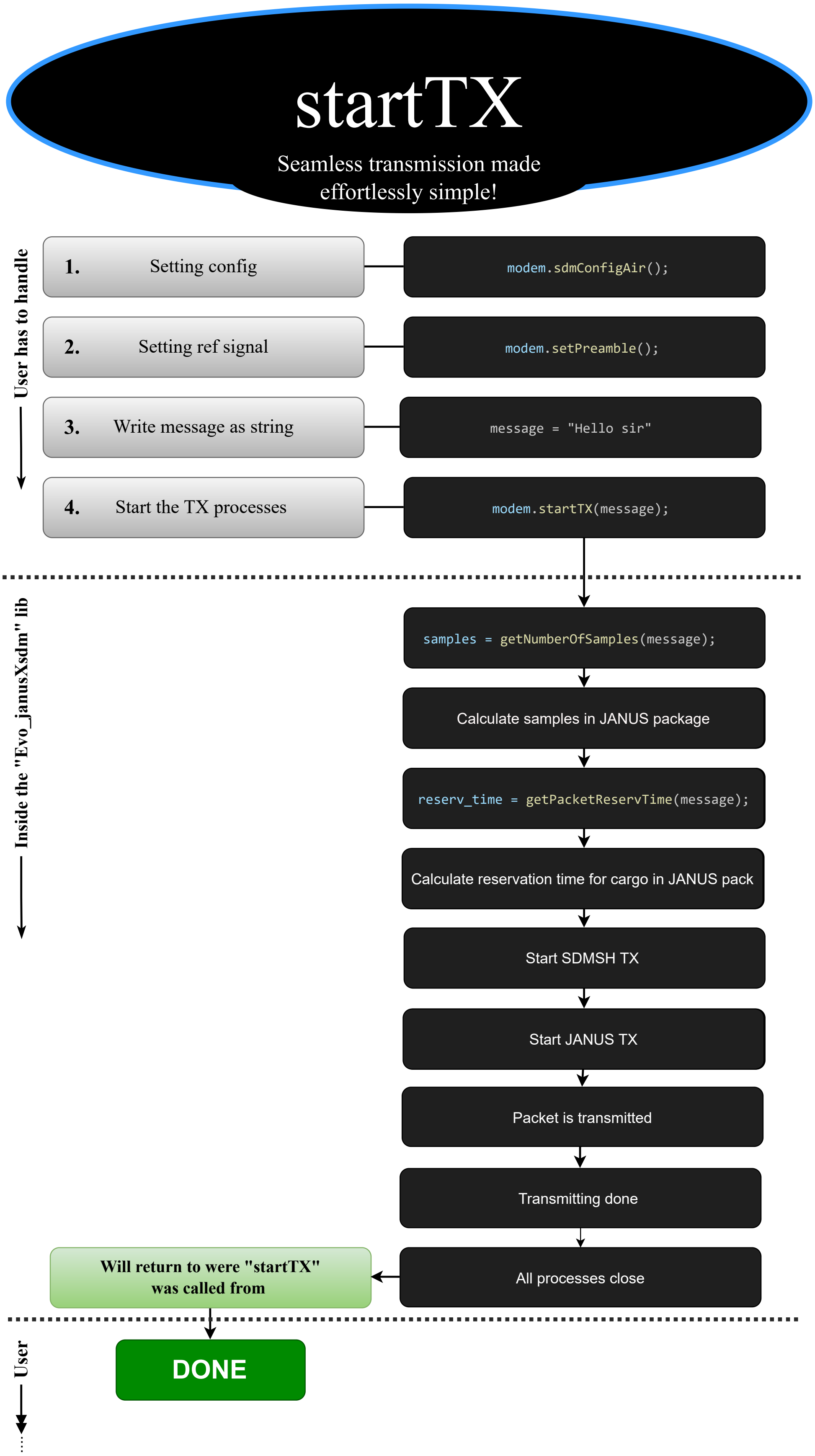
DONE

Data is not returned

User

DONE

Data is returned



startTX

Seamless transmission made effortlessly simple!

User has to handle

1. Setting config

```
modem.sdmConfigAir();
```

2. Setting ref signal

```
modem.setPreamble();
```

3. Write message as string

```
message = "Hello sir"
```

4. Start the TX processes

```
modem.startTX(message);
```

Inside the "Evo_janusXsdm" lib

```
samples = getNumberOfSamples(message);
```

Calculate samples in JANUS package

```
reserv_time = getPacketReservTime(message);
```

Calculate reservation time for cargo in JANUS pack

Start SDMSH TX

Start JANUS TX

Packet is transmitted

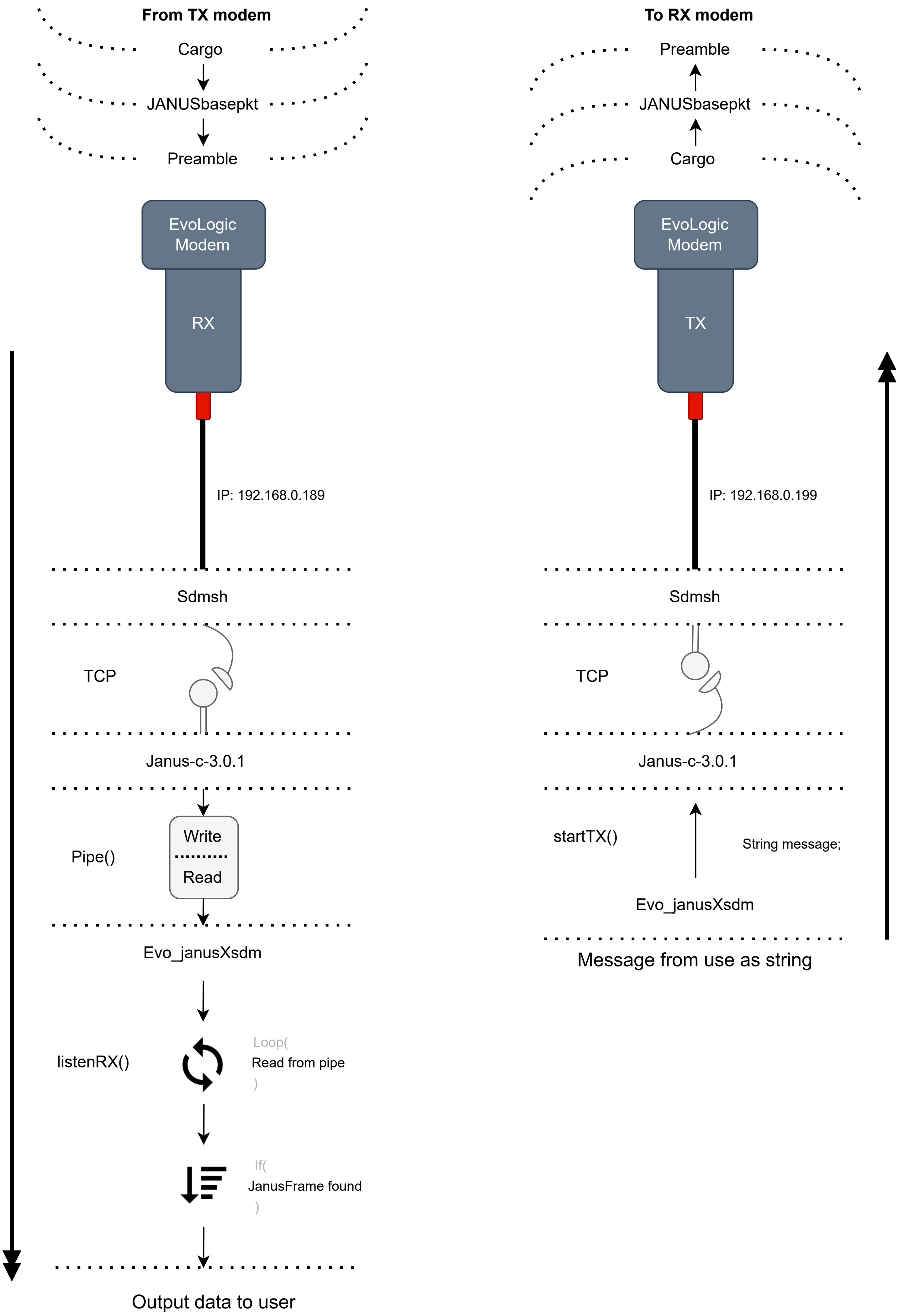
Transmitting done

All processes close

Will return to were "startTX" was called from

User

DONE


From TX modem
Cargo
JANUSbasepkt
Preamble
EvoLogic Modem
RX
IP: 192.168.0.189
Sdmsh
TCP
Janus-c-3.0.1
Pipe()
Write
Read
Evo_janusXsdm
listenRX()
Loop(
Read from pipe
)
If(
JanusFrame found
)
Output data to user
To RX modem
Preamble
JANUSbasepkt
Cargo
EvoLogic Modem
TX
IP: 192.168.0.199
Sdmsh
TCP
Janus-c-3.0.1
startTX()
String message;
Evo_janusXsdm
Message from use as string

ra-NRC for node i

```
procedure INITIALIZATION (ATOMIC)
    $x_i \leftarrow x^0$
    $y_i \leftarrow 0,\ g_i \leftarrow 0,\ g_i^{\text{old}} \leftarrow 0$
    $z_i \leftarrow I_n,\ h_i \leftarrow I_n,\ h_i^{\text{old}} \leftarrow I_n$
    $\sigma_{i,y} \leftarrow 0,\ \sigma_{i,z} \leftarrow 0$
    $\rho_{i,y}^{(j)} \leftarrow 0,\ \rho_{i,z}^{(j)} \leftarrow 0, \quad \forall j \in \mathcal{N}_i^{in}$
    $flag_{\text{reception},i} \leftarrow 0,\ flag_{\text{update},i} \leftarrow 0$
    $flag_{\text{transmission},i} \leftarrow 1$
end procedure
procedure DATA TRANSMISSION (ATOMIC)
    if $flag_{\text{transmission},i} = 1$ then
        transmitterNodeID $\leftarrow i$
        $y_i \leftarrow \frac{1}{|\mathcal{N}_i^{\text{out}}|+1} y_i$
        $z_i \leftarrow \frac{1}{|\mathcal{N}_i^{\text{out}}|+1} z_i$
        $\sigma_{i,y} \leftarrow \sigma_{i,y} + y_i$
        $\sigma_{i,z} \leftarrow \sigma_{i,z} + z_i$
        Broadcast: transmitterNodeID, $\sigma_{i,y}$, $\sigma_{i,z}$
        $flag_{\text{transmission},i} \leftarrow 0$
    end if
end procedure
procedure DATA RECEPTION (ATOMIC)
    if $flag_{\text{reception},i} = 1$ then
        $j \leftarrow$ transmitterNodeID, $(j \in \mathcal{N}_i^{\text{in}})$
        $y_i \leftarrow y_i + \sigma_{j,y} - \rho_{i,y}^{(j)}$
        $z_i \leftarrow z_i + \sigma_{j,z} - \rho_{i,z}^{(j)}$
        $\rho_{i,y}^{(j)} \leftarrow \sigma_{j,y}$
        $\rho_{i,z}^{(j)} \leftarrow \sigma_{j,z}$
        $flag_{\text{reception},i} \leftarrow 0$
        $flag_{\text{update},i} \leftarrow 1$ (optional)
    end if
end procedure
procedure ESTIMATE UPDATE (ATOMIC)
    if $flag_{\text{update},i} = 1$ then
        $x_i \leftarrow (1-\varepsilon)x_i + \varepsilon z_i^{-1} y_i$
        $g_i^{\text{old}} \leftarrow g_i$
        $h_i^{\text{old}} \leftarrow h_i$
        $h_i \leftarrow \nabla^2 f_i(x_i)$
        $g_i \leftarrow h_i x_i - \nabla f_i(x_i)$
        $y_i \leftarrow y_i + g_i - g_i^{\text{old}}$
        $z_i \leftarrow z_i + h_i - h_i^{\text{old}}$
        $flag_{\text{update},i} \leftarrow 0$
        $flag_{\text{reception},i} \leftarrow 1$ (optional)
    end if
end procedure
```

B Attachments

1. ROS 2 Project (code).zip
2. Optimization-main-sims (code).zip
3. Evo_JANUSxSDM(code).zip
4. Assignment text.pdf
5. poster_w_QR.pdf
6. tank_testRX.txt

NTNU

Norwegian University of
Science and Technology