Finn Christian Eriksen
Markus Næss Ervik
Pål Ivar Delphin Kværnø Fosmo
Aleksander Misje Furnes

Development of Underwater Communication Rig

Bachelor's thesis in Electrical Engineering
Supervisor: Torleif Anstensrud
May 2022

Bachelor's thesis

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

NTNU
Norwegian University of
Science and Technology

Finn Christian Eriksen
Markus Næss Ervik
Pål Ivar Delphin Kværnø Fosmo
Aleksander Misje Furnes

Development of Underwater Communication Rig

Bachelor's thesis in Electrical Engineering
Supervisor: Torleif Anstensrud
May 2022

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

NTNU
Norwegian University of
Science and Technology

Preface

This thesis marks our last endeavor as Bachelor's students at NTNU. It has been three years filled with excitement, stress and fun. All four of us met the first week and have been friends ever since. A Friendship that made the whole experience special to us all. We want to thank all our peers and lecturers who took part in our daily lives over the past three years and, of course, our institute, the Department of Engineering Cybernetics.

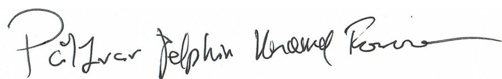
A special thanks to Damiano Varagnolo for the exciting task. We also appreciate him taking time out of his busy schedule these past months to contribute with his insight. A big thanks to Behdad Aminian for all his efforts to procure the necessary equipment for the group and his important guidance throughout the project, as well as his positive attitude. The help he has provided is greatly appreciated. Additionally, we would like to thank Emil Wengle, who contributed with his knowledge of the JANUS protocol and his insight into communication systems, and for helping the group set up the modems in the initial phase of the project.

The group would like to express its gratitude to our supervisor, Torleif Anstensrud, for his helpful guidance and continued availability whenever the group had any questions.

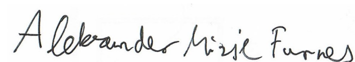
We would also like to thank Ida Heggen Trosdahl. Her help was essential for structuring and developing parts of the project. Lastly, we would like to offer special thanks to Finn Eriksen and Øyvind Delphin Fosmo for proofreading this paper.

Finn Christian Eriksen

Markus Næss Ervik



Pål Ivar Delphin Kværnø Fosmo



Aleksander Misje Furnes

Summary

This thesis aims to provide a standardized solution for an underwater network using underwater acoustic modems. All types of data should also be able to be exchanged between multiple ROS 2 based systems. The task is part of a larger project where the final product will be integrated on a BlueROV2, a high-performance underwater ROV. In addition to the acoustic modems, some sensory equipment is mounted on the ROVs. The ROS 2 system publishes the sensor data to the connected modem. The modem uses the JANUS protocol, an underwater communication standard for acoustic modems, to transmit the data in question.

Through the preliminary project work the group has planned the entire project using work packages and milestones. A plan for deadlines, working hours and prioritization of the milestones is visualized through a GANTT chart, all of which are also described in the preliminary project report. All group members attended a crash course in ROS 2 held by the client, as well as individual beginner courses in C++ through Codecademy. Introductory demonstrations of JANUS and the modems are also provided by the client in the water tank to get the group started with the modems.

ROS 2 is a set of open-source software libraries and helpful tools for developing robot applications used throughout the project. Raspberry PI 3 Model B+ is used for controlling the modems and handling information and data. With its ROS 2 capabilities it also offers an optimal platform for a future ROS 2 implementation. All code is written in C++ as it features faster runtimes than Python, which is preferred when developing systems where low latency is essential.

SDMSH, a shell for SDM-mode for the underwater modems, is used to access the modem's configuration settings and simplify the sending and receiving of data. As this is a big project that can develop over several years, the idea is to use JANUS, which is developed to be the communication standard for modems with different manufacturers. This way, the project will not be prolonged if the modems are changed to different ones. EvoLogics AMA and netcat are used as learning tools in order to get a better understanding of how the modems operate.

The first step is setting the whole system up and manually transmitting data through the modems by using EvoLogics AMA and netcat. The modems are connected to a 24V power supply and through Ethernet cables to one of the switches. The group uses OpenSSH with password authentication enabled to access the Pis remotely. All devices on the Ethernet are configured with a static IPv4 address. JANUS is patched and installed by using files available when logged in on januswiki.com and a set of patch files developed by EvoLogics.

By utilizing the executables janus-rx and janus-tx in parallel with SDMSH the modems are able to send and receive JANUS-packets. An automatic implementation is realized by a C++ library called janusxsdm that handles all processes related to the transmission and reception of data. The packet length in samples is calculated using a linear function derived from seven different packets of differing lengths. The library forms the basis for communication pertaining to a protocol developed by the group. The protocol handles communication in a network of three or more nodes through a master-slave structure. The protocol is implemented through C++ scripts and all the nodes communicate through standardized commands.

A ROS 2-implementation is achieved by including ROS 2 subscribers and publishers in the scripts for the master and the slaves. The slaves have subscriber nodes embedded into their code to continually listen for new data from the sensors on a topic. The data is then forwarded to the

master during transmission, who in turn publishes the data to another topic through a publisher-node integrated into the master script.

The group has developed a library to handle the automatic data transmission using JANUS and SDMSH, containing a class that takes five arguments when declared. This class is called in the master and slave scripts upon transmission or reception. The class has eight different methods which help automate the processes linked with JANUS and SDMSH, some of which havehave not been adequately implemented. The protocol uses a master-slave structure where the master determines which slave node can transmit and when it can transmit. The master also keeps records of which nodes are active in the network. The ROS 2 implementation is included in the code but not properly tested.

The final product involves a standardized solution for underwater communication based on the janusxsdm-library with the promised attributes. The solution is further comprised of a communication protocol with a master-slave structure, with the possibility of ROS 2 implementation with some minor tweaks to the code. In the end, the group is satisfied with the effort put into the work and the collaboration, together with the experience and knowledge gained. At the same time, the group can't help but feel disappointed with the final product compared to the work put in and the anticipated result.

The group recommends that further work on this project is centered around properly implementing the SDM and JANUS libraries directly into the master and slave scripts and using executors to handle the ROS 2 implementation. Therefore the group also recommends that further development is done by developers who have more knowledge of C++ and ROS 2 than the group itself.

Sammendrag

Målet med denne oppgaven er å produsere en standarisert løsning for et undervannsnettverk ved bruk av akustiske undervannsmodem. Alle typer data skal også kunne utveksles mellom flere ROS 2 baserte systemer. Oppgaven er en del av et større prosjekt, og sluttproduktet monteres på en BlueROV2, som er en "high-performance" undervannsdrone. I tillegg til de akustiske modemene er det diverse sensorutstyr montert på undervannsdronen. ROS 2-systemet publiserer sensordataen til det tilkoblede modemet. Modemet bruker JANUS, en undervanns kommunikasjonsstandard for aukustiske modem, til å sende nevnte data.

Gjennom arbeidet i forprosjektet har gruppen planlagt hele prosjektet ved å bruke arbeidspakker og milepæler. En plan for frister, arbeidstimer og prioritering av milepæler er visualisert gjennom bruken av et GANTT-diagram, og alt dette står beskrevet i rapporten til forprosjektet. Alle gruppemedlemmene gjennomfører et krasjkurs i ROS 2 som er avholdt av oppdragsgiver, i tillegg til nybegynner kurs i C++ gjennom Codecademy. Introduksjonsdemonstasjoner av JANUS og modemene i vanntanken er også gjennomført av oppdragsgiveren, for at gruppen skal kunne starte arbeidet.

ROS 2 er et sett med åpen-kilde programvare bibliotek og andre hjelpsomme verktøy for utvikling av robotapplikasjoner som skal benyttes i prosjektet. Raspberry Pi 3 Model B+ brukes til å styre modemene og håndterer all informasjonsflyt. Piens muligheter for å kjøre ROS 2 gjør det til den optimale plattformen med tanke på en fremtidig implementasjon av ROS 2. All kode er skrevet i C++ siden språket har raskere kjøretid enn Python, noe som er foretrukket i systemer der liten ventetid er viktig.

SDMSH, som er en shell for SDM-modus for undervannsmodemene brukes for å få tilgang til modemets konfigurasjonsinnstillinger, og for å forenkle sending og mottak av data. Siden dette er et stort prosjekt som kan utvikle seg over flere år, er ideen å bruke JANUS, som er utviklet for å være kommunikasjonsstandarden for modemer fra forskjellige produsenter. På denne måten vil ikke prosjektet bli forlenget dersom modemene byttes ut med andre modemer.

Det første steget er å sette opp hele systemet og sende data gjennom modemene manuelt ved å bruke EvoLogics AMA og netcat. Modemene er koblet til en 24V spenningskilde og gjennom Ethernet kabler til en av nettverksswitchene. Gruppen bruker OpenSSH med password autentisering aktivert for å koble til Raspberry PI eksternt. Alle enheter på Ethernettet er konfigurert med en statisk IPv4-adresse. JANUS er patchet og innstallert ved å bruke filer som er tilgjengelig etter å ha logget inn på januswiki.com og med et sett patchfiler som er utviklet av EvoLogics.

Ved å ta i bruk de kjørbare filene janus-rx og janus-tx i parallell med SDMSH er modemene i stand til å sende og motta JANUS-pakker. En automatisk implementasjon er realisert ved bruk av et C++-bibliotek kalt janusxsdm som tar for seg alle prosesser relatert til sending og mottak av data. Lengden på en pakke i form av antall tastinger er kalkulert ved å bruke en lineær funksjon som er utledet fra syv forskjellige pakker med forskjellige lengder. Biblioteket danner grunnlaget for kommunikasjon i forbindelse med en protokoll som gruppen har utviklet. Protokollen håndterer kommunikasjonen i et nettverk av tre eller flere noder gjennom en master-slave-struktur. Protokollen er implementert gjennom flere C++-skript og alle nodene kommuniserer ved bruk av standardiserte kommandoer.

En ROS 2-implementasjon er realisert ved å inkludere ROS 2 publiserere og abonnenter i skriptene

til masteren og slavene. Slavene har abonnent-noder som en del av koden sin for å kontinuerlig lytte etter ny data fra sensorene på et emne. Dataen blir så videresendt til masteren ved overføring, som igjen publiserer dataen til et nytt emne gjennom en publiserende node som er integrert i master skriptet.

Gruppen har utviklet et bibliotek som håndterer den automatiske dataoverføringen ved bruk av JANUS og SDMSH. Dette biblioteket inneholder en klasse som tar fem argumenter ved deklarasjon. Klassen kalles i master- og slave-skriptene ved sending eller mottagelse av data. Klassen har åtte forskjellige metoder som automatiserer forskjellige prosesser forbundet JANUS og SDMSH, derav noen ikke helt ferdigutviklet enda. Protokollen bruker en master-slave-struktur der masteren bestemmer hvilken node som kan sende, og når denne noden kan sende. Masteren har også oversikt over hvilke noder som er aktive i nettverket. ROS 2-implementasjonen er inkludert i koden men ikke testet skikkelig.

Det ferdige produktet inneholder en standardisert løsning for undervannskommunikasjon basert på janusxsdm-biblioteket med de lovede funksjonene avtalt med oppdragsgiver. Løsningen består også av en kommunikasjonsprotokoll med master-slave-struktur med mulighet for en ROS 2-implementasjon med noen få endringer i koden. Til slutt er gruppen fornøyd med arbeidet som er nedlagt og samarbeidet innad i gruppen, sammen med erfaringen og kunnskapen hver og en har tilegnet seg. Samtidig er ikke gruppen helt fornøyd med det ferdige produktet sammenlignet med hvor mye arbeid som er lagt ned og resultatet man hadde sett for seg tidligere i prosjektet.

Gruppen anbefaler at videre utvikling av produktet i all hovedsak dreier seg om å implementere SDM- og JANUS-bibliotekene direkte inn i master- og slave-skriptene på en bedre måte, samt å bruke executors for å håndtere ROS 2-aspektene ved produktet. Gruppen anbefaler derfor at videre utvikling gjøres av personer med bredere kunnskap om C++ og ROS 2 enn gruppen selv.

Table of Contents

Preface i

Summary ii

Sammendrag iv

Table of Contents vi

List of Figures ix

List of Tables x

Glossary and Acronyms xii

1 Introduction 1

1.1 History 1

1.2 Research question 1

1.3 Context 2

1.4 Procedure 2

1.5 Challenges 3

1.6 Structure 3

2 Theory 4

2.1 Hardware 4

2.1.1 BlueROV2 4

2.1.2 EvoLogics S2C R 18/34 USBL 5

2.1.3 Raspberry PI 3 Model B+ 6

2.1.4 Network switches 7

2.1.5 DC Power supply 7

2.2 Software 8

2.2.1 ROS 8

2.2.2 SDMSH 8

2.2.3 JANUS 9

2.2.4 EvoLogics AMA 11
2.2.5 Netcat 12
2.2.6 Visual Studio Code 12
2.2.7 C++ 12

3 Implementation **13**
3.1 Raspberry PI 13
3.2 Wiring 13
3.3 Networking 14
3.3.1 Static IP configuration 14
3.3.2 SSH setup 15
3.4 Manual transmission using AMA and NetCat 15
3.5 JANUS 15
3.5.1 Installing Dependencies 16
3.5.2 Installing SDMSH 16
3.5.3 Installing JANUS 16
3.5.4 Preparation 17
3.5.5 Manual transmission 19
3.5.6 Automatic transmission 22
3.6 The Protocol 23
3.6.1 Handling of Commands 23
3.6.2 The Master 25
3.6.3 The Slave 28
3.6.4 ROS 2 Implementation 31

4 Results **36**
4.1 Library description 36
4.2 Protocol 37

5 Discussion **40**
5.1 Raspberry PI 40
5.2 Networking 40

5.3 Brand specific implementation 40
5.4 Implementation difficulties 41
5.5 Limited access 41
5.6 The Protocol 42
5.6.1 TDMA 42
5.6.2 Testing 42
5.7 ROS 2, experience and future solutions 42

6 Conclusion **44**

References **45**

Appendix **47**

A JANUS BIT ALLOCATION TABLE 47
B JANUS README 48
C JANUS-EVOLOGICS WORKSHOP PRESENTATION 60
D PROJECT POSTER 74

List of Figures

1 Early underwater acoustics [26]. 1

2 Underwater USBL Positioning Systems [28]. 2

3 JANUS wiki 3

4 BlueROV2 [2]. 4

5 Newton Gripper [5]. 5

6 EvoLogics S2C R 18/34 USBL [8]. 5

7 Modem Directivity Pattern. 6

8 Raspberry PI 3 Model B+ [24]. 7

9 List of SDMSH commands[10] 9

10 JANUS baseline bit allocation table. (Appendix p. 47) 9

11 The 13 tone pairs for the JANUS initial frequency band [19]. 10

12 Communication window of Evologics AMA 12

13 Proposed wiring diagram of the networking module when implemented on the ROV 13

14 Picture of the test setup in air 14

15 Graph for samples per character 18

16 Structure of the four processes. Modified picture from a Evologics-JANUS workshop presentation. (Appendix p. 74) 20

17 The test setup in the water tank 22

18 Complete protocol logic for the master side 25

19 Complete protocol logic for the slave nodes 29

20 Ideal implementation of ROS 2 in the network 32

21 Basic idea of the two threads 33

22 Result of a transmission using the library 37

23 Chart showing normal initialization of master and slave 38

24 Chart showing initialization of master and a slave without a registered master . . . 38

25 Chart showing an example of the flow of communication in the network during network transmission 38

26 Executor schedule [16]. 43

List of Tables

2 Configuration flags for both executables. (Appendix, p. 60) 11

3 Configuration flags for *janus-tx*. (Appendix, p. 60) 11

4 parameter_sets.csv with frequency settings for the 18/34 modems 17

5 Source Level configuration table 19

6 List of commands sent from the master 28

7 List of commands sent from the slave nodes 31

Glossary and Acronyms

AC	Alternating Current.
BlueROV2	A small high-performance ROV manufactured by BlueRobotics.
CCITT	Consultative Committee for International Telephony and Telegraphy, now ITU-T.
CIDR	Classless Inter-Domain Routing.
DC	Direct Current.
EvoLogics S2C R 18/34 USBL	Acoustic underwater modem manufactured by EvoLogics.
FIFO	First in, first out.
GSM	Global System for Mobile Communication.
IP	Internet Protocol.
IPv4	Internet Protocol version 4.
LAN	Local Area Network.
method	A method is a function that belongs to a class.
network level	One of seven layers in the ISO OSI reference model. The network level is responsible for moving data from one host to another[14]..
OpenSSH	Connectivity tool for remote login using the SSH protocol.
PI	Short name for the Raspberry PI product family.
PI3	Short name for the Raspberry PI 3.
Raspberry PI 3	A small single-board computer.
ROS	Robot Operating System.
Round-robin	CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way. It is basically the preemptive version of First come First Serve CPU Scheduling algorithm.
ROV	Remote Operated Vehicle.

SDM	Software Defined Modems.
SDMSH	Shell for SDM.
SSB	Single-sideband.
SSH	Secure Shell.
stream-driver	The medium through which the communication flows.
TDMA	Time-Division Multiple Access.
Transceiver	A device that can both transmit and receive communications, in particular a combined radio transmitter and receiver.
Transponder	A device for receiving a signal and automatically transmitting a different signal..
USBL	Ultrashort baseline.
WLAN	Wireless LAN.

1 Introduction

1.1 History

The fact that sound can be heard underwater as well as in the air was noted by Aristotle nearly 2000 years ago. In the 1400's Leonardo Da Vinci observed that you could hear ships through the ocean over great distances, through a long tube submerged in the water[27]. Underwater acoustics developed quickly during the world wars and in the cold war that followed. The wartime developments led to large-scale research, and the modern sense of the word "underwater communication" started to develop during the second world war. The underwater telephone, developed in 1945 by the US, is regarded as one of the first underwater communication systems. It was used to communicate with submarines through a SSB suppressed carrier amplitude modulation in the frequency range of 8-11kHz.[20]

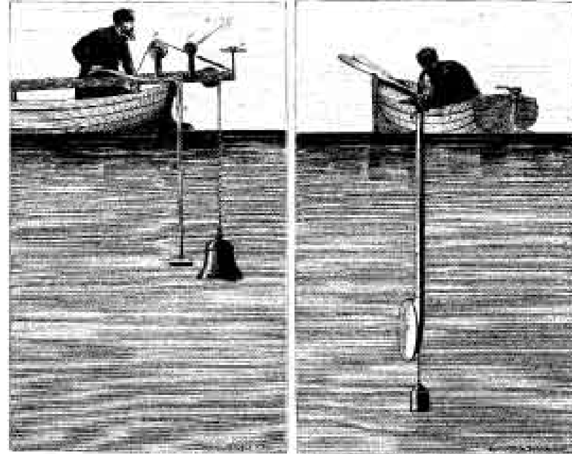


Figure 1: Early underwater acoustics [26].

Coupled with other technological advancements, underwater acoustics has evolved rapidly in modern times. The rapid growth can also be attributed to the fact that the field extended from being exclusively military to entering the commercial sector after the cold war. This extension leadsled to acoustic technology making considerable advancements in range and data throughput. High-quality video transmission over 6000 km and acoustically controlled robots performing maintenance work on submerged platforms are some of the latest breakthroughs in the field[20]. This bachelor's thesis will "dive deeper" into underwater acoustic technology and data transmission.

1.2 Research question

The project is divided into two different parts. Firstly the physical part required building a communications platform for a network of underwater modems. This platform is meant to operate while the modems are mounted on BlueROV underwater drones. The platform will be responsible for all data transmission between the modems. Therefore the problem this report sought to solve is:

Develop a standardized solution for underwater wireless networking using Evologics S2C Underwater Acoustic Modems as transceiver modules to exchange all types of data between multiple ROS 2 based systems.

This wording is quite vague, but the client's main target was to have a foundation for future development. Therefore the goal has been to create a functioning platform without any special requirements other than transferring data. However, the scope may be extended if the goal is completed earlier than expected. The possible additions that have been discussed include bit error correction, multiple transmission protocols and GSM interfacing.

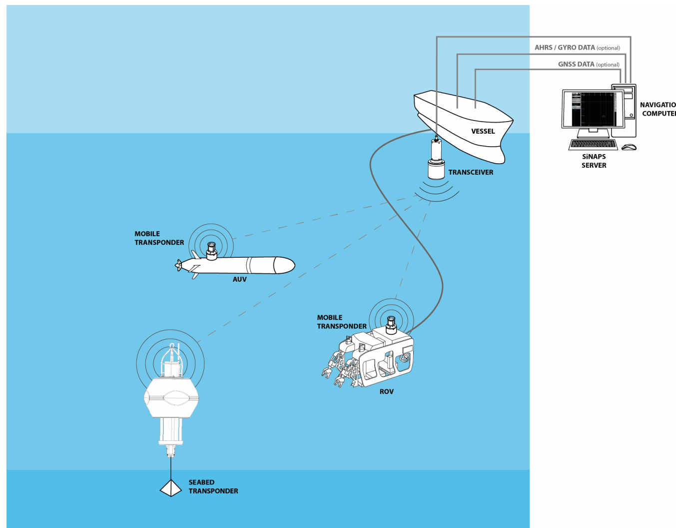


Figure 2: Underwater USBL Positioning Systems [28].

1.3 Context

The project's purpose is to deliver a platform on which the BlueROV underwater robots can communicate. The client will expect a fully working and documented product, which in this case will be the finished communication unit with documented protocols, reports and work packages.

The project consists of three groups working on their bachelor's degrees and the clients working on their doctorates. The three underlying tasks given to the bachelor students were Remote operation of ROV's, Communication protocols, and collection of sensory data. The platform will serve as one of the main components in an already developed system consisting of multiple robots. Therefore the client will expect an integrated product regarding the already existing modules. The client can use the platform to conduct underwater studies using different types of underwater robots with modems and sensors attached to them. The platform should therefore be able to operate with modems mounted on a Bluerobotics BlueROV and handle the exchange of data between them.

1.4 Procedure

To start off the semester, the group met with the clients to discuss the project. The clients presented their thoughts and ideas to the group and did a quick demonstration, briefly showing transmission of a "string" between two modems. The agreement was for the group to carry out the transmission of data and, to the best of the group's ability, build a protocol for the communication.

As time was limited, the project scope was restricted so that the goals would be obtainable. Then, if possible, the scope could be broadened towards the end of the project. As there were insubstantial amounts of documentation, much time was spent troubleshooting and figuring out how JANUS behaved.

Biweekly meetings were held with the client to ensure the project was on track and to discuss any challenges. Possible solutions were written down and further examined after the meeting.

1.5 Challenges

The group encountered many challenges. The modems are quite expensive. Consequently, they were locked in a room the group did not have access too. This meant the group depended on a third party opening every morning and closing every evening. Furthermore, it was discovered that the modems picked up a lot of interference when used in air. Talking while transmitting would ruin the data output. Moreover the "water tank room" was occupied for three weeks due to laboratory training. Additionally, the water tank in and of itself was smaller than recommended for the modems utilized in this project.

The project relied on the use of Raspberry PI's to transmit data via ROS 2. The client had placed an order early in the semester, but unfortunately, it was impossible to get a hold of. One of the group members had two, and these were deployed for the entirety of this project. Unfortunately, ROS 2 ceased functioning on the RPIs approximately two weeks before the deadline.

Figure 3: JANUS wiki

Smaller difficulties that led to a slow start originated from a lack of experience in C++ and the inadequate documentation relating to JANUS.

1.6 Structure

The second part of the bachelor thesis is this technical report. It will shed light on the process and issues that have been encountered and discuss the most important choices that were made. Furthermore, it will contain documentation on the code that has been written and utilized.

As this project might run over multiple years, the outline is intended to make it easy for future collaborators to quickly lookup certain parts of the project. At the same time, it should be easy to read through. Thus the academic chapters are split into hardware, software, implementation, results and discussion.

This report methodically reviews the process in which the project unfolded and the final product was developed. It starts with a short introduction to the hardware and the related specifications, and the usage of every piece of equipment is also presented. The software section contains information about the programs and related firmware utilized with the hardware.

The specifics of the implementation are presented in the implementation section, where the method and explanations are laid out and described. The resulting product is presented in the result section, culminating all the steps and assessments from the implementation section. The result is discussed in detail in the following discussion section, where alternative solutions are also presented. Finally, the group brings the whole report to an end in the conclusion section, which summarizes the final product in light of the task described beforehand.

2 Theory

2.1 Hardware

2.1.1 BlueROV2

BlueROV2 is a high-performance underwater ROV with six- and eight-thruster configurations, many available accessories and open-source software. At the front of the ROV there is a 1080p wide-angle low-light camera that is mounted to a tilt mechanism so that the pilot can control the camera tilt to look up and down. The ROV uses the t200 thrusters in a vectored configuration, giving it a high thrust-to-weight ratio and the ability to move precisely in any direction. The depth rating of the BlueROV2 is 100m or 330ft. It is controlled by a drone flight controller running the open-source ArduSub subsea vehicle control firmware. The pilot controls the ROV using a laptop or gamepad controller from the surface. [1] [2]

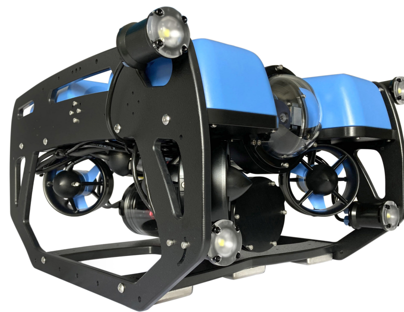


Figure 4: BlueROV2 [2].

The ROV used in this project is expanded with the heavy configuration retrofit kit, adding two more thrusters and more buoyancy for additional stability and payload carrying capacity. The ROV has a light accessory and a Newton Gripper to move and manipulate objects underwater. [3] [4] [5]

Figure 5: Newton Gripper [5].

2.1.2 EvoLogics S2C R 18/34 USBL

The project utilizes EvoLogics S2C R 18/34 USBL modems for the underwater positioning and communication system. These modems calculate the positioning data simultaneously with the acoustic transmission, and the communication is full-duplex with a range up to 3500 meters [8]. These qualities are the main reasons why the modems were chosen for the project. The idea is for one modem to be connected via a cable to a computer on a boat or land to "float" close to the surface and receive data from the two other modems. These other two modems will be connected to each of their BlueROV2 underwater drones. The drones will collect sensory data as well. All data transmission will be done through the three underwater modems.

Figure 6: EvoLogics S2C R 18/34 USBL [8].

The system integrated into the modems is called a USBL-system, or Ultrashort baseline. It is a versatile system that often is used to calculate a position relative to a Transponder. The modem "floating" closest to the surface in this project is not a transponder but a Transceiver. Which means it can receive and transmit signals. For the sake of simplicity, the "floating" modem will be referred to as the transponder.

To calculate the positions, the transceiver sends an acoustic pulse to the transponder. The time it takes for the pulse to reach the transponder is measured and used to calculate the range. There are also an array of transducers, and these use phase differencing to calculate the angle between the modems in question. The range is then returned as delta x-, delta y- and delta z-coordinates. [13]

While most sensors radiate sound in a conical directivity pattern, these modems have horizontally omnidirectional patterns. This pattern is not as accurate over greater distances, but the frequency band only returns a range up to 3500 meters. Within this range, the directivity pattern is good enough to give an acceptable quality signal. Additionally, it is better suited for use in shallower water.

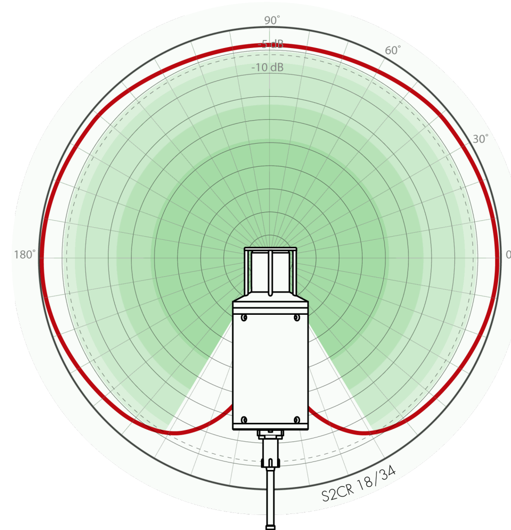


Figure 7: Modem Directivity Pattern.

The modems are powered by 24 volts DC and draw 2.5mW in standby and up to 65 Watts when transmitting a message[8].

2.1.3 Raspberry PI 3 Model B+

The Raspberry PI 3 is a small form factor computer. It is based on the Broadcom BCM2837B0 microprocessor, which has a 64-bit architecture. With 1GB of LPDDR2 SDRAM. A 5V DC supply powers the Pi through a USB Micro-B connector. The card is also equipped with multiple interfacing options, including an SD-card slot, HDMI, USB, Ethernet, WLAN (WiFi) and Bluetooth. In addition to this, it has a 40-pin GPIO-header that supports multiple forms of digital signaling.[24] Due to the PI3 costing approximately 300 NOK per unit [7] and featuring many interfacing options. It can serve as a low-cost processing unit for various applications requiring low- to moderate processing power.

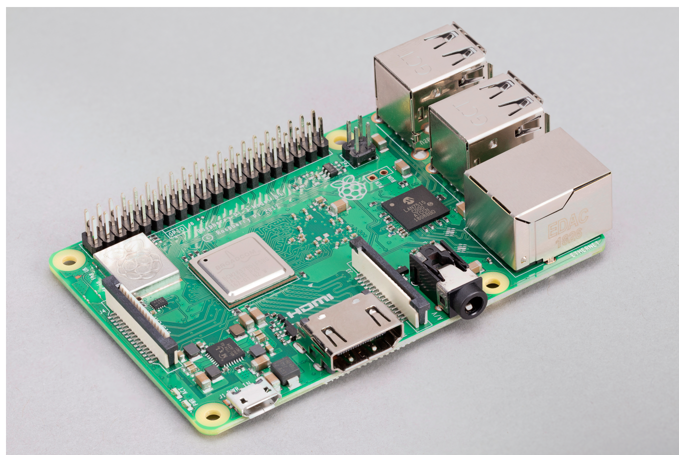


Figure 8: Raspberry PI 3 Model B+ [24].

In the project, the PI3 is used to manage data between the acoustic modem and the local Ethernet. It also serves as a *network level* controller, which is used to configure the acoustic modem and keep a table of the available acoustic connections.

2.1.4 Network switches

Netgear network switches are used to connect devices by ethernet cable. The specific type used in this project is equipped with RJ-45 sockets.

2.1.5 DC Power supply

A DC power supply converts an AC voltage source to a DC voltage. In this project, it is used to power the modems during testing.

2.2 Software

The following is a list of software and tools utilized by the group throughout the project.

2.2.1 ROS

ROS stands for Robot Operating System, a set of open-source software libraries and other helpful tools for developing robot applications. Even though operating system is in the name, ROS is not actually an operating system. ROS-based processes take place in nodes that can receive, post, and multiplex sensor data, control, state, planning, actuator and other messages. ROS is not a real-time operating system, despite the importance of low latency and reactivity in robot control. The lack of support for real-time systems was one of the reasons for the development of ROS 2. [30]

ROS 2 is an environment developed with some specific interests in mind, including but not limited to teams of multiple robots, real-time systems, non-ideal networks, and production environments. Another reason for building ROS 2 was to take advantage of the opportunity to improve the user-facing API (application programming interface). A lot of the ROS code that exists now is compatible with client libraries from as far back as 2009. This is good for stability, but it also means that API decisions that were made a long time ago are still in effect, and not all of those decisions are considered to be best at the current time. One could argue that just enhancing ROS would be better than developing ROS 2, and in principle, that was possible. However, the intrusive nature of the changes that would have to be made to achieve the functionality that the developers wanted would be too risky for the system that many people rely upon. [6]

2.2.2 SDMSH

SDMSH is a shell for *SDM*-mode for the underwater modems, developed by EvoLogics for easy access to the modems' configuration settings. It offers the possibility of configuring the **SDM PHY**, which is an API for the physical layer of the modems. This includes changing the reference signal to a custom signal and also the ability to transmit a custom signal by passing 16-bit pulse-code modulated data to the modem.

SDMSH also simplifies the sending and reception of data through the modems by using simple commands such as *rx* for reception and *tx* for transmission. Both are followed by a set of parameters relating to the general nature of the sent data. [10]

value	description	usage
config	Config SDM command	config <threshold> <gain> <source level> [<preamp_gain>]
usbl_config	Config SDM USBL command	usbl_config <delay> <samples> <gain> <sample_rate>
stop	Stop SDM command	stop
ref	Update reference signal	ref [<n samples>] [<driver>:]<params>
tx	Send signal	tx [<n samples>] [<driver>:]<params>
rx	Receive signal [0 is inf]	rx <n samples> [<driver>:]<params>
rx_janus	Receive signal [0 is inf]	rx_janus <n samples> [<driver>:]<params>
usbl_rx	Receive signal from USBL ch.	usbl_rx <channel> <n samples> [<driver>:]<params>
systime	Request systime	systime
waitsyncin	Wait SYNCIN message	waitsyncin
usleep	Delay in usec	usleep <usec>
source	Run commands from file	source <source-file>
help	This help	help
history	Display history	history [number-lines]

Figure 9: List of SDMSH commands[10]

2.2.3 JANUS

The JANUS communication protocol is an open-source signaling method developed as an underwater acoustic communication standard for modems from differing manufacturers[19]. This protocol utilizes frequency shifting in order to transmit data through sound waves. The JANUS packet consists of a 64-bit header which is further divided into 11 data fields describing the nature of the sender and the packet. Appended to the header is the optional cargo field holding the raw data the modems wish to communicate, whose length is specified in the Application Data Blocks.

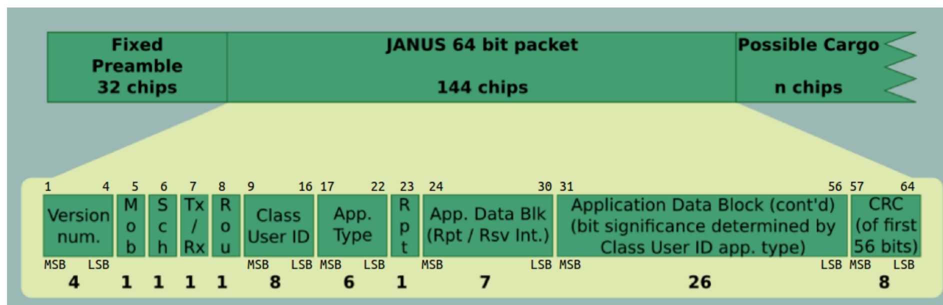


Figure 10: JANUS baseline bit allocation table. (Appendix p. 47)

For the transmission with acoustic modems, the protocol uses *Frequency-Hopped Binary Frequency Shift Keying*. Frequency Hopping is when the wave traveling between the modems hops between different frequencies within the frequency band in a predetermined order known both by the sender and receiver. This is done in order to minimize the chance of successful interception and to counter

interference-related issues. Binary Frequency Shift Keying is the method of encoding used for the data itself, where the data is transformed into a stream of bits and transmitted using sound waves with two different frequencies representing the two different binary values.

The frequency hopping is specified as using 13 evenly-spaced tone pairs within the frequency band, and the predetermined hopping between these 13 tone pairs is derived from Galois Field arithmetic. Figure 11 shows the 13 tone pairs defined for the initial JANUS frequency band defined with center frequency

$$F_c = 11520Hz$$

and therefore, a bandwidth of

$$Bw = 4160Hz$$

FH	bit	freq. (Hz)	FH	bit	freq. (Hz)
0	0	9440	7	0	11680
	1	9600		1	11840
1	0	9760	8	0	12000
	1	9920		1	12160
2	0	10080	9	0	12320
	1	10240		1	12480
3	0	10400	10	0	12640
	1	10560		1	12800
4	0	10720	11	0	12960
	1	10880		1	13120
5	0	11040	12	0	13280
	1	11200		1	13440
6	0	11360			
	1	11520			

Figure 11: The 13 tone pairs for the JANUS initial frequency band [19].

FH denotes the sequence number of the hop (tone pair) and is divided into two separate frequencies for the two binary values.

The integrity and validity of the packet are checked using an 8-bit *Cyclic Redundancy Check* (CRC), the value of which is represented by the last 8 bits of the 64-bit packet. The CRC is generated by the preceding 56 bits using the CRC-CCITT 8-bit polynomial:

$$p(x) = x^8 + x^2 + x^1 + 1$$

JANUS allows for the optional use of so-called wake-up tones, which are described as three tones of differing frequencies. The wake-up tones signal to an idle modem in a low-power mode that it should wake up and prepare to receive packets. The frequencies of the three tones range from the lowest frequency in the band through the center frequency to the highest frequency in the band, or more specifically

$$F_c - \frac{Bw}{2}; F_c; F_c + \frac{Bw}{2}.$$

Usage

The "C" version of JANUS supplies two executables, *janus-tx* and *janus-rx*. When running the executables, there are several configuration flags that need to be applied for encoding/decoding to take place. (Appendix, p. 60)

"C" option	**Function**	**default**
−−verbose	Verbose level	0
−−stream-driver	Stream Drv (nul/alsa/pulse/raw/wav/wmm)	wav
−−stream-driver-args	Stream Driver Arguments	janus.wav
−−stream-fs	Stream Sampling Frequency (Hz)	44100
−−stream-format	Stream Format	S16
−−stream-channels	Stream Channels configuration	1
−−stream-channel	Stream Active Channel	0
−−stream-passband	Stream Passband signal	1
−−config-file	Configuration file	

Table 2: Configuration flags for both executables. (Appendix, p. 60)

Further there are a number of specific flags for encoding of JANUS messages with *janus-tx*:

"C" option	**Function**	**default**
−−pad	Enable/Disable Padding of output	1
−−wut	Enable/Disable Wake Up Tones	0
−−stream-amp	Stream Amplitude Factor (0.0 - 1.0]	0.95
−−stream-mul	Stream samples multiple of given number	1
−−packet-mobility	Mobility	0
−−packet-tx-rx	Tx/Rx capability	1
−−packet-forward	Forwarding capability	0
−−packet-class-id	Class User Identifier	16
−−packet-app-type	Application Type	0
−−packet-reserv-time	Reservation Time	0.0
−−packet-repeat-int	Repeat Interval	0.0
−−packet-app-data	Application Data	0x0000000
−−packet-app-data-fields	Application Data Fields	
−−packet-cargo	Packet Optional Cargo	

Table 3: Configuration flags for *janus-tx*. (Appendix, p. 60)

2.2.4 EvoLogics AMA

Evologics AMA is a user-friendly interface for function testing of Evologics Acoustic Modems developed by the manufacturer of the modems. It is easy to set up, only requiring the IPv4 address of the Modems to test. [9]

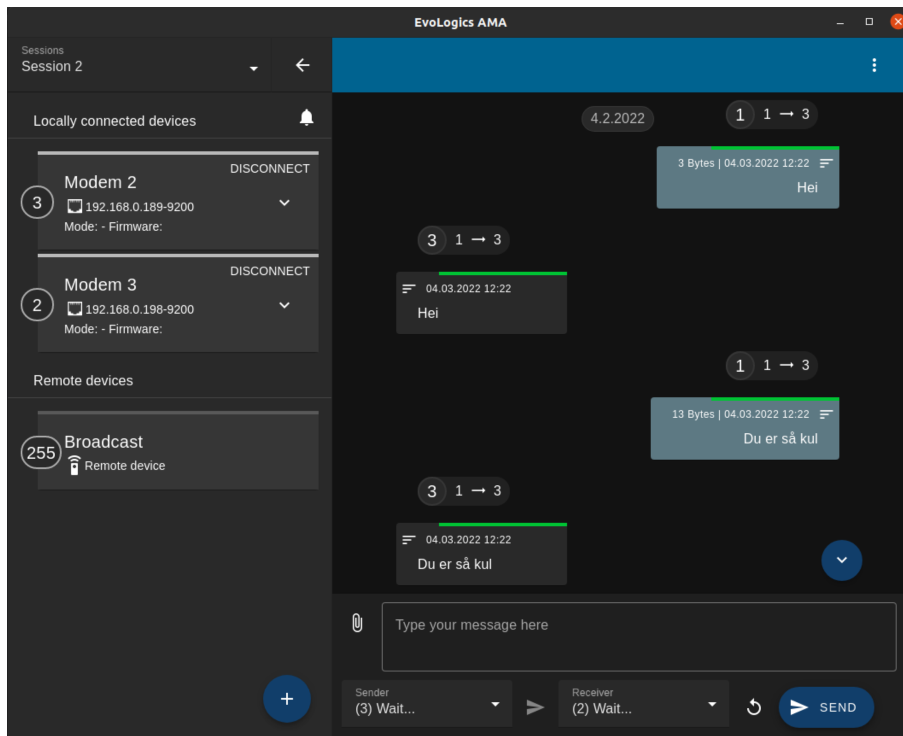


Figure 12: Communication window of Evologics AMA

2.2.5 Netcat

Netcat is a computer networking utility used for writing to and reading from network connections. It uses TCP or UDP. Commands can be given directly or by other programs and scripts. Netcat's list of features includes file transferring, port scanning and port listening. Like any server, netcat can also be used as a backdoor(bypassing authentication or encryption). [29]

2.2.6 Visual Studio Code

Visual Studio Code is a code editor made by *Microsoft*. The code editor is supported by *Windows*, *Linux* and *macOS*. The program includes features like debugging, syntax highlighting, intelligent code completion, and embedded Git [31]. The program also lets the user install extensions that add additional functionality. VS code supports several different languages, including C++.

2.2.7 C++

C++ is a programming language created in the 80's. The original purpose was to extract the strengths in the programming language C and combine them with Simula features. Today the language has two main components, a direct mapping of hardware features provided primarily by the C subset and zero-overhead abstractions based on those mappings. The founder, Bjarne Stroustrup, has summarized it as "a light-weight abstraction programming language [designed] for building and using efficient and elegant abstractions.". [21]

3 Implementation

The first part of the project mainly involved setting up the various subsystems and gathering information. As acoustic communication was foreign to all project participants, having access to a "getting started" guide and the reference manual was essential.

3.1 Raspberry PI

Firstly, the group installed the necessary software. The SD-card of the PI3 was flashed with Ubuntu server 22.04 LTS (64-bit) as the PI3 lacked the necessary memory to run a desktop version. This was done using the Raspberry PI Imager[23]. The hostname used on the PI3 was "amodemN" where the N denotes the modem number as there were two of them. While the username was set to "Janus" with password "passord1".

The project members also installed Ubuntu Desktop in a dual-boot configuration on their computers to facilitate easier development.

3.2 Wiring

The system was connected as seen in figure 13. Here the modems are wired to a power supply that supplies 24V DC. And the modem "POWER_ON" input is connected directly to the same 24V DC source in order to have the modems power on when the voltage source is activated. Both the PI3 and the modem is connected to a network switch.

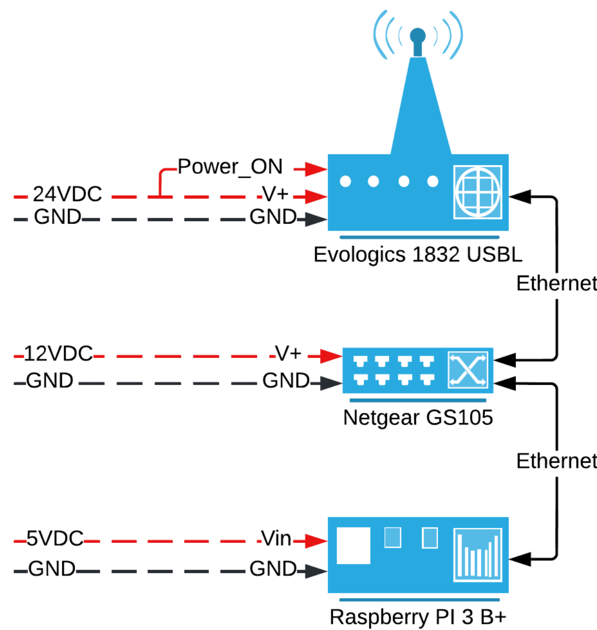


Figure 13: Proposed wiring diagram of the networking module when implemented on the ROV

During testing, the setup was duplicated and the two switches were connected in order to be able

to access both modems from the same computer.

Figure 14: Picture of the test setup in air

3.3 Networking

To install packages on the PI while it was connected to the modem via ethernet, the group installed *nmcli* in order to configure the WLAN interface.

The acoustic modems were configured by the manufacturer to have a static IPv4 address in the 192.168.0.xxx/24 range, where 24 denotes the subnet mask *255.255.255.0* in CIDR format[11]. In order to communicate with the modem, the connecting device would also have to be in that IP range. This was realized by setting static IP addresses on the PI3 Ethernet adapter and the Ubuntu computers used in testing.

3.3.1 Static IP configuration

As *nmcli* did not have permissions to change the properties of the Ethernet connection, this had to be configured manually. First, a service named "cloud" preinstalled on the operating system had to be disabled.

In "/etc/cloud/cloud.cfg.d/99-disable-cloudinit-networking.cfg":

```
network: {config: disabled}
```

Further the file "/etc/netplan/50-cloud-init.yaml" was copied and renamed to "/etc/netplan/99-config.yaml". Then "99-config.yaml" was edited to specify the static IP-address "192.168.0.18"

and subnet "/24". [32]

In "/etc/netplan/99-config.yaml":

```
network:
    ethernets:
        eth0:
            addresses: [192.168.0.18/24]
            optional: true
    version: 2
```

The IP address can be configured to any number in the 2-255 range excluding addresses already present in the network.

The Ubuntu laptops were configured using the GUI.

3.3.2 SSH setup

In order to access the PI3 without screen and keyboard *OpenSSH* was installed.

```
sudo apt install openssh-server
```

This enables the user to access the Raspberry Pi's OS remotely through the use of the standard SSH-protocol in the terminal. This basic standard only allows for connection establishment using SSH-keys, which is not very user-friendly. In order for connection establishment using username and password, password authentication had to be enabled in the file "/etc/ssh/sshd_config":

```
PasswordAutentication yes
```

The PI3 was then accessible through SSH using the terminal command:

```
ssh username@IP
```

where IP is the IPV4-address of the PI and username is a user configured on the PI.

3.4 Manual transmission using AMA and NetCat

At the very start of the project, to make sure that the modems worked properly, the group used *EvoLogics'* own application called *EvoLogics AMA* to send data. The application is very intuitive, and the modems were in good condition, so the test was done quickly and successfully.

The group also performed tests by using *netcat* in the early stages of the project. The tests revolved around issuing configuration commands and sending simple messages underwater. This was done partly to make sure that the group could send data by using software other than *EvoLogics AMA*, which is the built-in *EvoLogics* software. It was also a way for the group to gain experience with underwater transmission while developing the knowledge to be able to transmit using *JANUS*.

3.5 JANUS

An important part of the project was to use a communication standard that did not rely on a specific hardware platform. With guidance from the client, JANUS was selected as the preferred

protocol since there already existed a guide for interfacing JANUS with the Evologics SDMSH platform.

A project folder was established. Code-sharing and co-developing code were done by utilizing GitHub. By using this tool, the group could develop code in parallel, and all members could easily fetch the latest code at any time. GitHub was also used to keep track of all the libraries that were needed for the system to work as intended. For code development and testing, the group opted to use *Visual Studio Code* as it features an easy-to-use GitHub integration.

3.5.1 Installing Dependencies

In order to use JANUS and SDMSH, some software/libraries had to be installed.

Install make

```
make -version #Check if make is installed
sudo apt install make
sudo apt install build-essential
```

Install cmake

```
sudo snap install cmake --classic
```

Install FFTW3

```
wget http://fftw.org/fftw-3.3.10.tar.gz
tar -xzf fftw-3.3.10.tar.gz
cd fftw-3.3.10
./configure
make
sudo make install
make check
```

Install libreadline

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

3.5.2 Installing SDMSH

Further sdmsh was installed in "*$project/lib/*" from the EvoLogics github repository:

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

3.5.3 Installing JANUS

The JANUS code was retrieved from januswiki.com where one has to have an account to download the files. The files were then unpacked to the project folder: "*$project/lib/*".

To support the use of TCP streams, EvoLogics had developed a patch for the JANUS library that had to be applied. This patch was provided in a .mbox format which reads like a series of e-mails.

To apply the patch, git was initialized in the "*janus-c-3.0.5/*" folder with the .mbox file located one level above.

```
git init .
git add .
```

the patch was then applied using *git am*[12]

```
git am ../janus-c-3.0.5.evologics-patchset.mbox
```

With the patch applied, JANUS was prepared and built to the subfolder "bin/" using CMake and make

```
cmake -S . -B bin/
cd bin
make .
```

The baseline JANUS frequency band is not within the operating frequency band of the modems. To make JANUS encode and decode messages in a frequency spectrum the modems supported the file "parameter_sets.csv" in "$project/lib/janus-c-3.0.5/etc/" was altered.

# Id	Center Frequency (Hz)	Bandwidth (Hz)	Name
1	11520	4160	Initial JANUS band (9-14 kHz)
2	23040	8320	Evologics 1834 modem

Table 4: parameter_sets.csv with frequency settings for the 18/34 modems

The new set is based around a center frequency of 23040Hz and has a bandwidth of 8308Hz. This is in accordance with the operating frequency band of the modems and within the JANUS protocol standard of $Bw = \frac{F_c}{3} \pm 10\%$.

3.5.4 Preparation

JANUS sample size

In order to transfer data using SDMSH and have the function call terminate properly, one would need to know how many samples are being passed. The number of samples varies depending on the size of the packet being sent. Therefore one has to calculate the number of samples. To calculate the number of samples, there has to be an understanding of how many bytes and thereby samples there are for different sized packets. For this to work, one has to have an automatic variation in the number of samples. This variation has to correlate with the size of the packet being sent. To make sure the number of samples is correct every time. A test was established. First multiple JANUS messages with varying cargo sizes were generated and output to .raw files. This makes it possible to see how many bits the JANUS message consists of.

```
./janus-rx  --pset-file ../etc/parameter_sets.csv --pset-id 2 --stream-driver raw
    --stream-driver-args ../data/janusMessage.raw --stream-fs 96000 --stream-format
     S16 --verbose 1
```

Now that the size of the raw file is available. There were sent seven different packets. The first was empty, and then the following packets incremented the number of characters in the application data by a factor of two. Starting at one and ending with 32. This method results in seven different

byte sizes. Since a byte is eight bits and a sample is two bytes, the group calculated the number of samples required for the different amounts of characters. This resulted in the following graph:

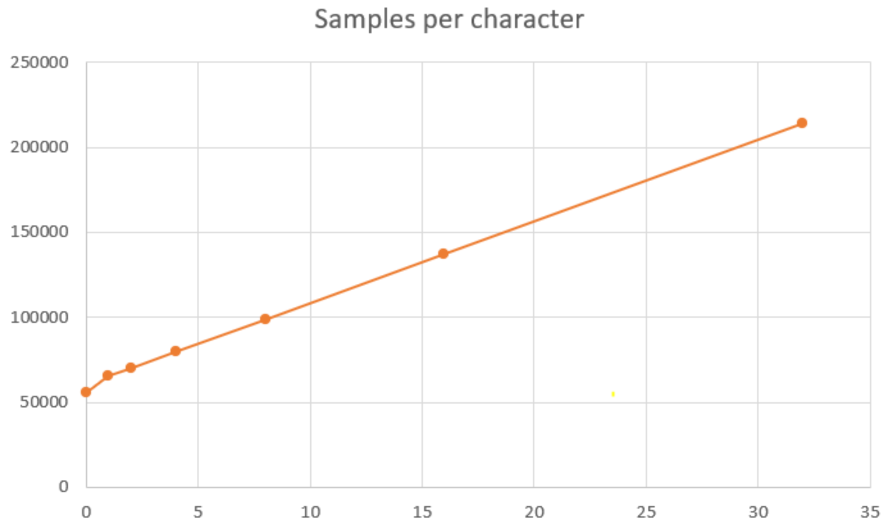


Figure 15: Graph for samples per character

There is a clear linear correlation between the number of characters in the packets. The exception is when the packet is empty. The empty packets had a sample size of 55800, and the packet with one character had 65400. This is a difference of 9600. From one character onward, the rate of change remained constant at 4800. This difference in rate is attributed to a payload descriptor that is added to the header, informing the packet recipient that the packet will be followed by an appended cargo with a given size.
Since the function is linear from the first sent character, some conditional logic was implemented. If the packet is empty, the sample size is fixed at 55800, and every other amount of characters follows a simple linear function:

$$n_{samples} = 4800 * n_{characters} + 60600$$

This relation only applies to a specific sampling frequency of 96kHz. For other sampling frequencies, new measurements are required.

```
if(message == "")
{
    samples = 55800;
}
else
{
    samples = (message.length() * 4800 + 60600);
}
```

Listing 1: Code implementation of the JANUS sample size.

Modem Setup

To be able to use the modems with SDMSH and JANUS, they first had to be put in *PHY-mode*. This was done by connecting to the modem on port 9200 and sending the command "+++ATP".

```
nc ModemIP 9200
+++ATP
```

As most of the testing was done in air the source level of the modems had to be set to its lowest setting.

Value	Source Level
0	0dB (max)
1	-6db
2	-12dB
3	-20dB (min)

Table 5: Source Level configuration table

This was realized by running the command:

```
./sdmsh ModemIP -e "stop;config 30 0 3 0"
```

Where the third number represents the source level setting.

In order for the receiver to recognize a JANUS-packet a preamble had to be set on the receiving modem. This preamble was acquired from the client and placed in "$project/lib/sdmsh/" and then uploaded to the modem.

```
./sdmsh ModemIP -e "stop;ref preamble.raw"
```

3.5.5 Manual transmission

Changes to the default configuration of the JANUS-encoder/decoder are done through the use of execution flags.

To set the modem specific frequencies, which can be found in the file from Table 4, the file, first line, and its entry, second line, in the table must be specified with the flags below.

```
--pset-file ../etc/parameter_sets.csv
--pset-id 2
```

By default, the "stream format" is set to "S16", but in case of future changes to the library, this flag is specified as well. This ensures that the JANUS packet is encoded in the signed 16-bit pulse-code modulated waveform that SDMSH requires.

Further, the verbosity is set to 1 to allow for reading of the cargo from *stdout*.

The standard sampling rate for the EvoLogics implementation of JANUS is 62,5kHz, but due to the change in center frequency, and therefore f_{max}, for the new parameter set, this sampling rate does not satisfy the *Nyquist sampling theorem*. The theorem states that

$$f_s > 2 * f_{max}$$

in order for the sampled signal to accurately represent the actual signal. The group therefore opted to use a sampling frequency of 96kHz, which satisfies the sampling theorem since

$$96000Hz > 2 * (23040Hz + 8320Hz)$$

$$96000Hz > 62720Hz.$$

Lastly the stream-driver was configured to be a tcp-stream. The group ended up using the following configuration throughout the project period, with the exception of the port number.

```
--pset-file ../etc/parameter_sets.csv
--pset-id 2
--stream-driver tcp
--stream-driver-args listen:127.0.0.1:9988
--stream-fs 96000
--stream-format S16
--verbose 1
```

Listing 2: The JANUS configuration for reception.

```
--pset-file ../etc/parameter_sets.csv
--pset-id 2
--stream-driver tcp
--stream-driver-args listen:127.0.0.1:9977
--stream-fs 96000
--stream-format S16
--verbose 1
```

Listing 3: The JANUS configuration for transmission

To perform the manual transmission, four processes had to be run in parallel as shown in Figure 16.

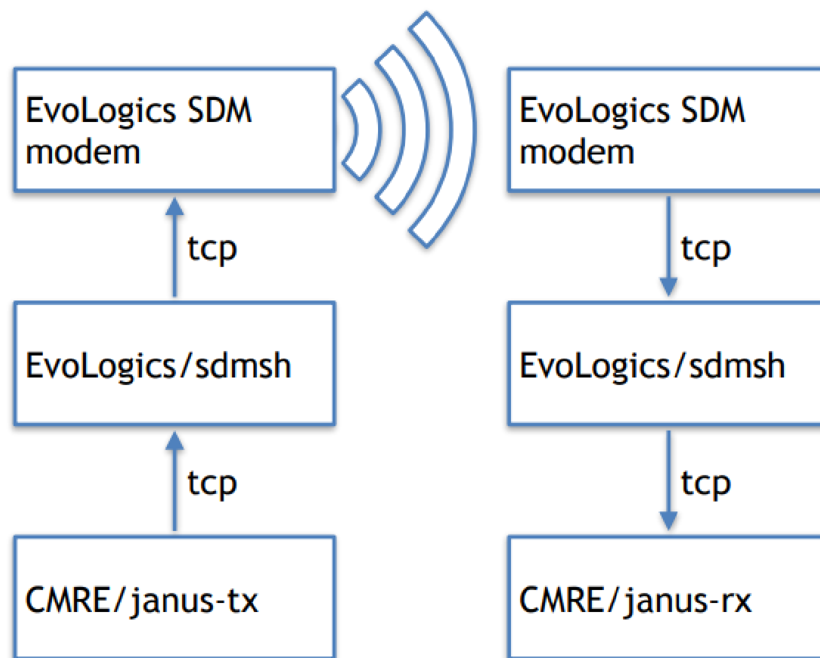


Figure 16: Structure of the four processes. Modified picture from a Evologics-JANUS workshop presentation. (Appendix p. 74)

Firstly for the receiving modem, a *janus-rx* instance was launched to decode the incoming signal.

In $project/lib/janus-c-3.0.5/bin/

```
./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 96000 --stream-format S16 --verbose 1
```

Then an instance of SDMSH was launched to relay the data from the receiving modem to the JANUS receiver.

In $project/lib/sdmsh/

```
./sdmsh ModemIP -e "rx 0 tcp:connect:127.0.0.1:9988"
```

The receiver was then actively listening for a message and the next step was to transmit a message.

Further a SDMSH server was set up to relay incoming messages to the transmitting modem. The number of samples to transmit was set according to the equation in Section 3.5.4

$$4800 * n_{characters} + 60600 = n_{samples}$$

For the test, the group sent the string "Hello World". which consists of 11 characters, including the space:

$$4800 * 11 + 60600 = 113400$$

In $project/lib/sdmsh/

```
./sdmsh ModemIP -e "tx 113400 tcp:listen:127.0.0.1:9977"
```

Lastly, a message was encoded using JANUS with the "packet-cargo" flag and passed to the SDMSH transmitter.

In $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:127.0.0.1:9977 --stream-fs 96000 --stream-format S16 --verbose 1 --packet-cargo "Hello World"
```

A sound from the modems could now be heard and shortly after the decoded message appeared in the terminal window of the *janus-rx* instance.

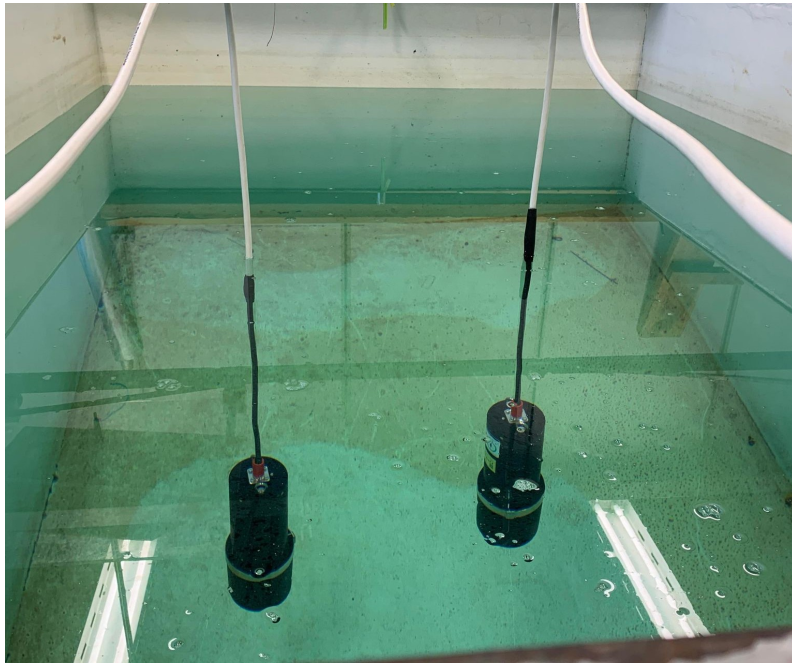


Figure 17: The test setup in the water tank

The transmission was repeated in water as seen in figure 17.

3.5.6 Automatic transmission

For the implementation of a system for automatic transmission, the group had the option to choose between coding in Python or C++. The latter was chosen as its processing speed compared to Python was important in a situation where low latency is valued.

During testing of the manual transmission, it soon became clear that a library for sending, receiving and configuring the modems would improve future development time as the library would handle the complexity of encoding/decoding JANUS and passing the data to the modems.

The development of the functions for transmission and reception proved difficult as large parts of the code had to work in parallel. First, multithreading was attempted but proved to not work in cases where the main thread had to terminate a child thread due to a blocking system call. The group instead opted to use forking due to the ease of parallelization and termination by process ID.

The library began with functions intended to take a modem-IP and a message. But was later improved to a class that contained the variables of a specific modem connection. This allowed for the declaration of multiple modems as objects and simplified the individual function calls to transmit/receive as the connection properties were stored in the object.

The library was named *janusxsdm* as it is a library designed to interface SDM with JANUS-encoded messages. See Section 4.1 for usage and a link to the complete library code.

3.6 The Protocol

By setting up and patching the different parts of the system, the group gained basic and vital knowledge of the different subsystems, which allowed for further development of the communication system. First of all, the group set out to outline the basic architecture of the protocol and mapped out the desired protocol behavior, trying to fulfill some fundamental criteria defined in advance.

- The protocol should handle communication both ways, from master to slave and vice versa.
- The protocol should be able to handle communication between three or more nodes.
- The protocol should be based on a TDMA-model, where each node is assigned a time slot for communication.

Through these bullet points, the group had laid a foundation upon which a protocol could be developed. A draft for the basic protocol logic was produced as a simple flow chart for visualization purposes. It included the initialization of the master and slaves together with the standard procedure for network transmission and how the master should respond to the nodes in different phases of communication.

3.6.1 Handling of Commands

Sending and reception of data through commands was facilitated through the *janusxsdm*-library developed by the group. Two functions called *janus_tx* and *janus_rx* handled sending and reception respectively in both the master and the slave scripts.

```
1  int janus_tx(string command){
2      cout << "Sending " << command << endl;       //For debugging purposes
3      //std::this_thread::sleep_for(1000ms);       //Redundant
4      modem.sendSimple(command);
5      return 1;
6  }
7
8  string janus_rx(int timeOut_interval){
9      string response;
10     std::chrono::duration<double> t;
11     t = std::chrono::duration<double> {timeOut_interval};   //Timer for the
       interruption of the listener, the time slot
12     if(modem.listen(response, t)){
13         cout << "Cargo: " << response << endl;       //For debugging purposes
14     }
15     return response;
16 }
```

Listing 4: The two functions handling sending and reception in both scripts

These functions use the *listen* and *sendSimple* functions from the library, with *modem* being the object declared earlier in the code.

For the protocol to function as intended, there has to be a standardized form of communication between them. The group solved this by introducing a standard format for the commands sent by both the master and the slave nodes.

//SUP-CMD//data;data>>.

The group opted to use this standardized format in order for the script to easily be able to split the string into different parts to determine what information has been received, as well as determine the end of the command. The first of the two data fields often consists of a MAC address, either of the intended recipient or the master, depending on what kind of command it is. *SUP* denotes the name of the protocol while the *CMD*-field is reserved for the actual command, and is a command-string composed of two letters. The first of these letters indicates whether the command is sent from the master or from a node through the use of *M* or *N*.

The reading and checking of the command strings are done in the respective scripts through a function called *response_check()*, which takes one or two arguments, depending on if it is the master or the slave. The function separates the received command string from *CMD* to >>, and uses if-sentences to differentiate between the commands. The optional data-fields are processed within the relevant if-sentences, and all further actions are determined here as well.

```
string response_check(string response,string mac){
    size_t spos = response.find("-");       //Separate the "SUP" from the actual command
    size_t epos;
    string command = response.substr(spos+1, 2);    //Extract command

    if(command == "NR"){
        spos = 10;          //Standard for every command received
        epos = response.find(">>");     //Find the end of the command
        string node_mac = response.substr(spos,epos-spos);  //Extract the data, here a MAC-address
        return node_mac;
    }
    else if(command=="NI"){
        .....
    }
    ...
}
```

Listing 5: Example of a response check for the "NR"-command in the master

Above is an example of the handling of the *NR* command, where the master wants to extract the MAC address of the new node for registration. The first three lines of the function are basic separation between the protocol name and the command line and data fields. Each command has its own if-sentence in the same manner, and the amount of actions performed by each command varies.

3.6.2 The Master

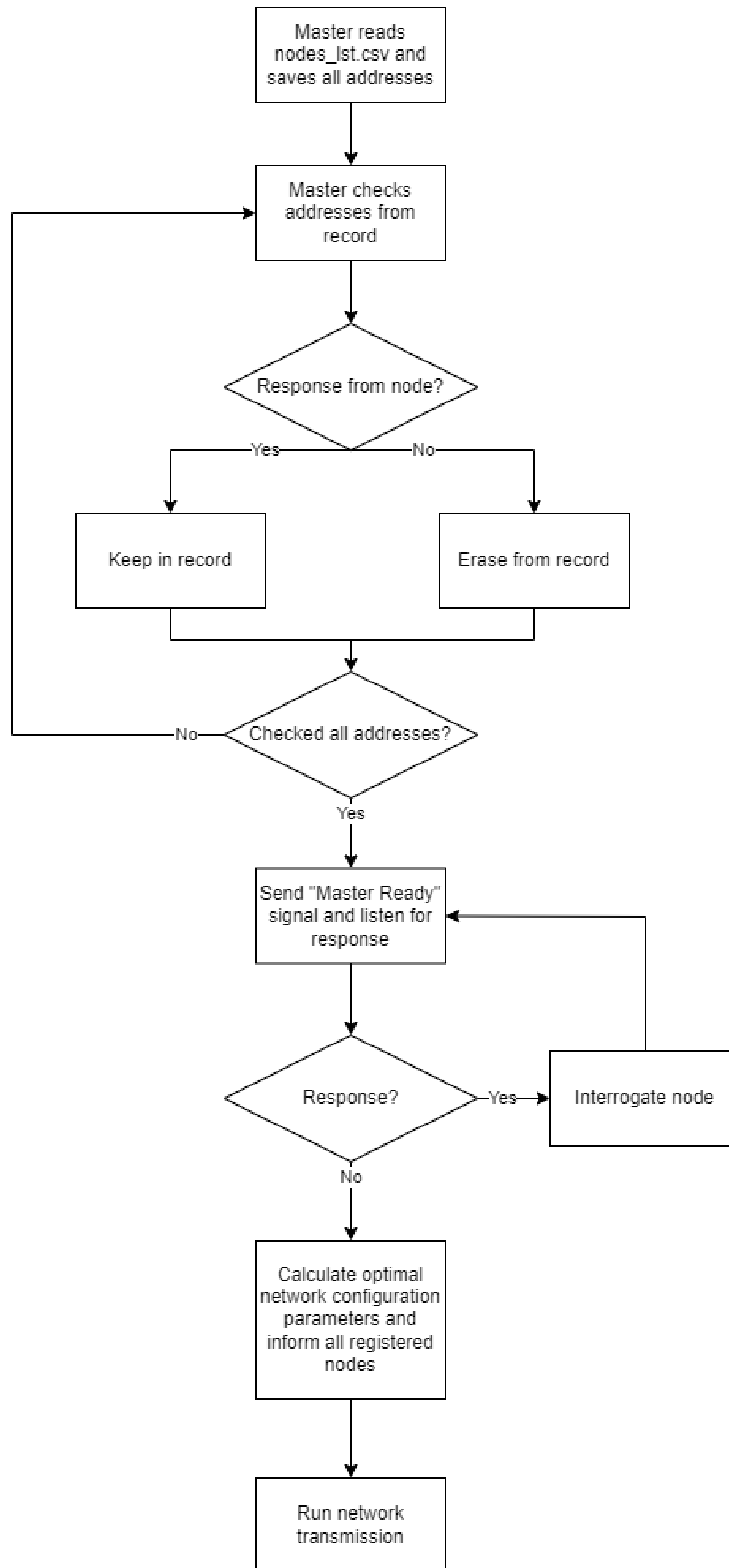


Figure 18: Complete protocol logic for the master side

The master node is the brain behind the operation in the protocol. All communication to, from and between the nodes goes through the master, and the master decides and calculates the relevant network parameters such as the designated time slot and ID for all nodes. The master also keeps track of which nodes are active in the network and keeps a record of the nodes saved locally in a csv-file. Each line corresponds to one node and has the following standard format:

```
MAC-address, Alias, Min. Frequency, Max. Frequency, Supported Protocols
```

```
48:0f:cf:01:f6:fc,node_1,18000,34000,SUP
48:0f:cf:01:f7:fc,node_2,18000,34000,SUP
48:0f:cf:01:f8:fc,node_3,18000,34000,SUP
```

Listing 6: Example of a node_lst.csv-file saved locally on the master

The reasoning behind the inclusion of maximum and minimum frequencies is in the case of the application of the protocol in a network consisting of different modems. This way, the master will have full control of which frequency bands can be used in the communication and configure the network accordingly. However, the group never created a proper implementation for this due to its complexity.

Figure 18 shows the logic that formed the basis for the development of the master script in C++. First of all, the master has to acquire the list of nodes previously active in the network, which is done by reading the csv-file. The first branch of the chart shows the process in which the master checks all the saved node addresses in order to check if it has the correct information and whether the node is active or not. Active nodes will respond with an ”OK” or the correct information, whilst inactive ones will not respond and therefore be removed from the records.

When all addresses have been checked, the configuration of the network parameters can commence. The master calculates the time slot based on the elapsed time of the different node checks. The longest elapsed time multiplied by some factor between one and two becomes the time slot. This guarantees that data transmission between the master and each node can be finished without interruption from other nodes. The final time slot duration was determined using the following logic for every node check.

```
auto start = chrono::steady_clock::now();
janus_tx(send_command);
response = janus_rx(7);
auto time_elapsed = chrono::duration_cast<chrono::seconds>(chrono::steady_clock::
    now()-start);
if(time_elapsed>prev_time_elapsed){
    prev_time_elapsed = time_elapsed;
}
time_slot_duration = prev_time_elapsed.count()*1.6;  //After all checks are done
```

The factor with which the time slot is multiplied depends on how many transmissions back and forth between the master and the node will be expected. The group wanted to be on the safe side and used a factor of *1.6*, which increases the time slot duration by 60% compared to the originally measured duration. Node ID is also determined in the configuration phase and is simply based on the index of the node in the address list of the master.

```
int set_config(){        //Sets the configured parameters for the network, as well
    as node IDs
    for(int i  = 0; i < nodes_addresses.size();i++)      //Sent to each node
        string node_config_data = std::to_string(time_slot_duration) + "|" + std::
    to_string(i+1);   //ID is based on the index of the node in the address list
```

```
        string send_command = "//SUP-MC//"+nodes_addresses[i]+";" +
    node_config_data + ">>";
        janus_tx(send_command);
    }
    return 0;
}
```

Listing 7: The logic for sending the configured parameters to the nodes

After the check of all the nodes is completed, the master broadcasts a "master ready"-signal to all the nodes. Any node that has not yet been registered with the master will respond to this ready signal and go through the registration process. This process involves the node sending all relevant information to the master and the master making a new entry in nodes_lst.csv for the new node. If the master does not receive any response from any nodes, it will assume all nodes have already been registered and checked and will proceed to send the configured network parameters to each node.

After the configuration, the network is ready for transmission, and the master goes into transmission mode. In this mode, the master will send a "Master Ready"-command to each node in turns, to which the nodes will be able to reply with any new data they have gathered. This process is run by a while-loop in the code and runs continuously until the program is stopped. Designating the order of nodes for transmission is the most important task of the master, and by making the master call upon each node it is more or less guaranteed that there will be no collisions between packets.

```
void network_transmission(){        //Network transmission commences here..
    cout << "Network transmission starting with time slot " << time_slot_duration
    << "ms.\n";
    while(true){             //Transmission runs continuously
        for(int i = 0; i<nodes_addresses.size();i++){       //For each node, do
    this
            int id = i+1;                          //Node ID is determined from their
    index in the address list
            string master_ready_cmd = "//SUP-MT//"+nodes_addresses[i]+";"+ std::
    to_string(id) +">>";
            janus_tx(master_ready_cmd);
            string response = janus_rx(time_slot_duration);
            response_check(response,nodes_addresses[i]);        //Check the
    response from the node
        }
    }
}
```

Listing 8: The logic handling network transmission from the master

Because of the difficulty of synchronizing the clocks of each node due to different distances, the group decided to adopt another model for the protocol. This entailed the master itself signaling to each node when it is their turn to transmit and when it is ready to receive whatever data the node has to offer. This model rendered the time slot duration and node ID redundant. However, the variables were kept for the purpose of potential future developments.

The protocol has a standardized format for commands and information flow. Both the master and the slave are assigned their own commands to send depending on the nature of the request or transmission.

<table>
<tr><th>Command</th><th>Description</th><th>Use</th></tr>
<tr><td>//SUP-MD//nodeMac;masterMac|
alias|min_f|max_f|protocol>></td><td>Node Request</td><td>Sent to each node during node check.
Checking if the master has correct info.</td></tr>
<tr><td>//SUP-MR//masterMac;>></td><td>Master Ready
(After Check)</td><td>Sent to every node to indicate that the
master is ready for transmission.</td></tr>
<tr><td>//SUP-MC//nodeMac;timeSlot|nodeID>></td><td>Send Config</td><td>Sent to each node informing about the
calculated time slot and node ID.</td></tr>
<tr><td>//SUP-MT//nodeMac;nodeID>></td><td>Master Ready
(Transmission)</td><td>Sent to each node when master is ready for
transmission to commence. Request data.</td></tr>
<tr><td>//SUP-MA//nodeMac;masterMac>></td><td>Node Registration
Ack</td><td>Sent from the master to acknowledge new
node registration process.</td></tr>
</table>

Table 6: List of commands sent from the master

Table 6 lists the various commands sent from the master to the nodes and their intended use in the developed scripts. These are the commands that the slave nodes currently can recognize and respond to, but more commands can easily be added through the script.

3.6.3 The Slave

Every node in the network connected to the master is regarded as a slave. They are mounted on the ROV together with the modem and the different sensors they gather information from. The slave nodes only respond to messages they receive from the master. Their logic is, therefore, less complex than the master's logic but still built in a similar manner. Each slave node has a MAC address, an alias and some fundamental properties such as maximum and minimum frequency supported. They can also be customized to support other protocols as well by adding additional commands to the script.

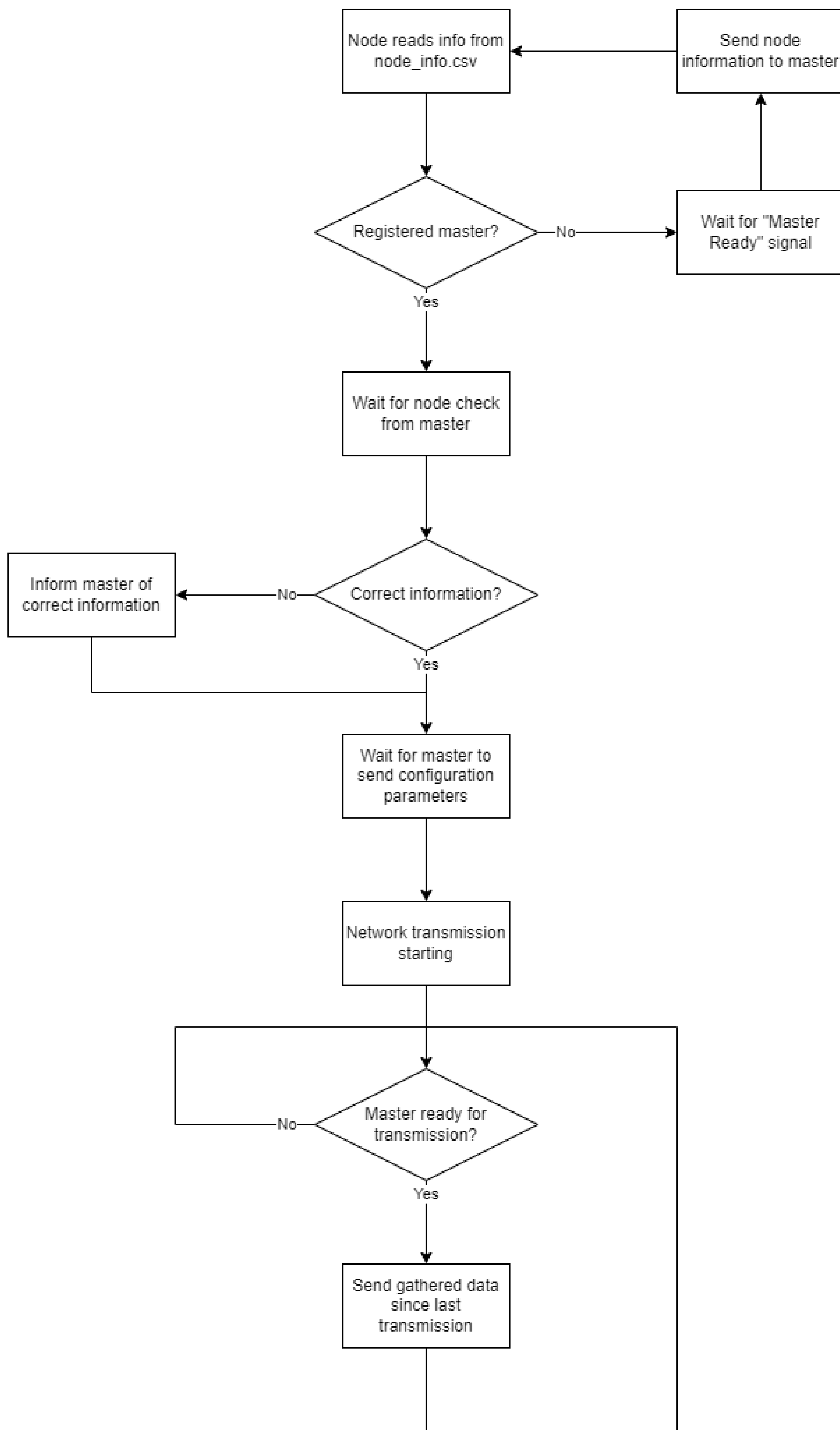


Figure 19: Complete protocol logic for the slave nodes

Figure 19 shows the logic developed for the slave nodes. It was realized through coding in a C++ script, just like with the master. The slaves' information is kept saved locally in a csv-file and

contains the MAC address of the master as well as the basic node information. This file is read every time the node is initialized and the information is saved in variables for easy access. In the csv-file the master is always located on the first line of the file, and its alias is always "master".

```
48:0f:cf:01:f9:fc,master
48:0f:cf:01:f6:fc,node1,18000,34000,SUP
```

Listing 9: Example of a "node_info.csv"-file saved locally on every node

When reading the node_info.csv-file the slave will first check if a master has previously been registered, and saves its MAC address in a variable. The node will also save its own information in variables so that they can easily be sent to the master when called upon. If the csv-file contains a registered master, the node will proceed to wait for the master node check. When the node receives its check from the master, it will compare the received information to its own information. If it matches up the node will answer with *//SUP-NC//OK;>>*. If the information is incorrect, the node will send the correct information to the master.

In the event that no master is registered, the node will wait for the "Master Ready"-signal. Normally the node will ignore this signal, but when no master is registered, it will reply with a command asking to be registered. When the master receives this message, it will reply with an acknowledgment, to which the node replies with its own information. The master will register the node in its records and run an extra node check on the node.

The original plan of building the protocol as a TDMA-protocol involved selecting time slots for each node to communicate with the master in a given order. For this to function, the master would have to communicate the size of this time slot and where each node stands in the order. As mentioned previously, this is done in the configuration phase, which culminates in the master sending each node its calculated network parameters. After this configuration has been received by all nodes, network transmission can begin.

Since the group later moved away from the idea of basing the protocol on TDMA, the only logic needed for the slave in the network transmission is a while-loop. The while-loop continuously listens for the master to send the ready-signal, and when it is received the node will transmit the latest data received from the sensors.

```
void network_transmission(){
    cout << "Transmission commencing" << endl;          //For debugging purposes
    while(true){
        response_check(janus_rx(time_slot_duration));   //Checking all received messages
    }
}
```

Above is the simple implementation for the network transmission in the slave nodes' scripts.

<table>
<tr><th>Command</th><th>Description</th><th>Use</th></tr>
<tr><td>//SUP-NC//nodeMac;alias|
min_f|max_f|protocol>></td><td>Node Config</td><td>Sent to master after node
check. Contains all information or ”OK”.</td></tr>
<tr><td>//SUP-NT//nodeMac;data1|data2|...>></td><td>Node Data Transmission</td><td>Send to master during
transmission.</td></tr>
<tr><td>//SUP-NR//nodeMac;>></td><td>Node Registration</td><td>Sent to master after ”Master
Ready” to inform of a new node
wanting to be registered.</td></tr>
<tr><td>//SUP-NI//nodeMac;alias|
min_f|max_f|protocol>></td><td>New Node Information</td><td>Send to master for
registration of new node.</td></tr>
</table>

Table 7: List of commands sent from the slave nodes

Table 7 shows the commands used by the slaves in communication with the master.

3.6.4 ROS 2 Implementation

Another goal of the group was to offer a full-fledged implementation of ROS 2 compatibility in the product. This compatibility would make it easier for the developed system to be woven in with the already existing system parts, which already utilized ROS 2 for communication and control. The idea was to include publishers and subscribers in the master and slave scripts that would run in parallel with the protocol logic and convey the data received all the way from the sensors to the operator.

On the slave side, sensors publish their data and findings to a topic, to which a subscriber in the slave script subscribes and receives the data. The data would then be transmitted to the master through the modems during the nodes' specified time slot. The master would then publish the received data to another topic through the publisher embedded in the same script. The operator is the final link in this chain, being a subscriber to the last topic and receiving all the transmitted data and writing it to the terminal or some other medium of choice.

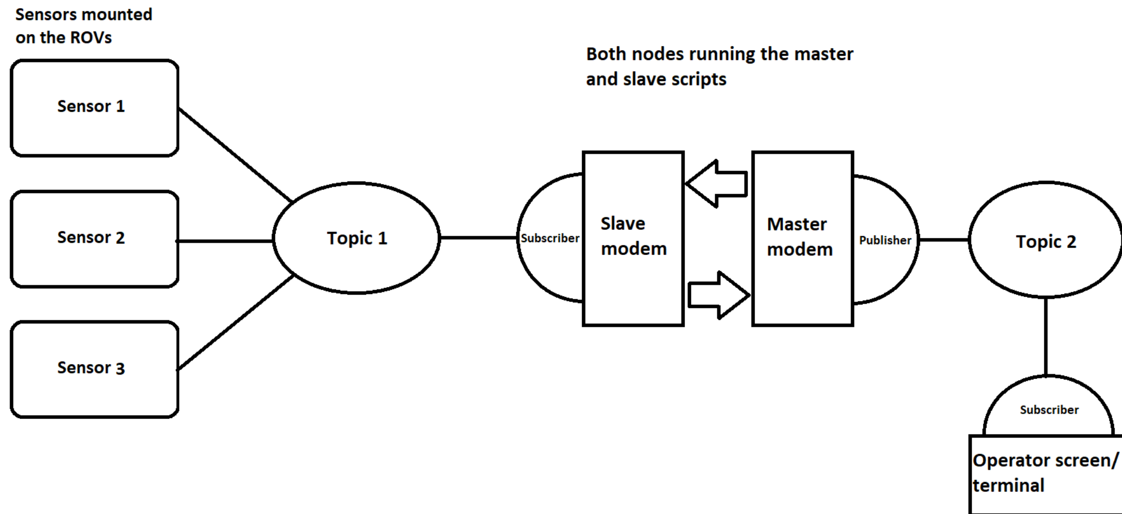


Figure 20: Ideal implementation of ROS 2 in the network

Figure 20 shows the theoretical architecture of the ROS 2 implementation in the system. The communication between the ROV mounted sensors and *Topic 1* is part of a separate project where the sensor values are published to a ROS topic, while the rest of the chain was to be developed alongside the protocol.

```
class MinimalPublisher : public rclcpp::Node        //The code related to the ROS 2 subscriber
{                                                   //Commented out in the final code due to lack of proper testing
public:
  MinimalPublisher()
  : Node("minimal_publisher"), count_(0)
  {
    publisher_ = this->create_publisher<std_msgs::msg::String>("masterTopic", 10);
    MinimalPublisher::timer_callback();
  }

private:
  void timer_callback()
  {
    auto message = std_msgs::msg::String();
    message.data = data1 + "; " + data2 + "; " + data3 + "; " + data4;    // Publishing up to four fields of data
    //RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
    publisher_->publish(message);
  }
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
  size_t count_;
};
```

Above is a draft of an implementation of a publisher originally meant to be integrated into the script for the master. It is a slightly modified version of the simple publisher available in the official ROS 2 documentation. This publisher is designed to publish the data stored in the different data-

variables once in the topic called *masterTopic* when called upon, unlike the original publisher who publishes in a loop with a given time interval. [18]

```
int node_run(int argc, char * argv[]){
    rclcpp::init(argc, argv);
    rclcpp::spin_some(std::make_shared<MinimalPublisher>());
    rclcpp::shutdown();
    return 0;
}
```

The function *node_run()* is called whenever the master wants the node to publish information. It is also a modified version of the *main()*-function in the simple publisher from the ROS 2 documentation. The only difference between the two is how the data processing in the node is started, with *spin_some* instead of just *spin*. The difference between these two methods lies in the way they execute the node. The *spin*-method executes the node so that the data processing continues indefinitely, even when there is no more data left to process. This would leave the node running until the script is stopped and locking the program, which is not optimal given the circumstances.

On the other hand, the *spin_some*-method only runs the node and executes the immediately available work until it is idle, such as publishing a simple data-string before it shuts down[17]. This means that the program would not be in danger of being locked into a loop where the publisher is waiting for new data to arrive. While not an optimal solution, this means that the implementation of the publisher in the master script is no problem in theory. A ROS 2 implementation in this part of the system can therefore be easily attained.

Moving on from the master to the slave nodes where the idea was to implement a subscriber in a similar manner, the group found this to be a more complex task than previously thought. In contrast to the publisher where the task can be easily defined as "publishing a message", the subscriber's task is to continually listen for new messages and process them. This means that when the subscriber is called, the program will be locked in a listening state until it is terminated.

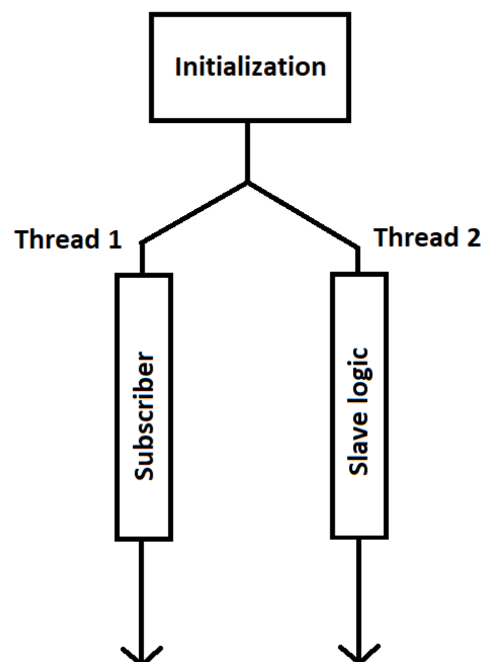


Figure 21: Basic idea of the two threads

Due to this, the *spin_once*-solution would not work on the subscriber, and therefore another solution was needed. Multithreading was the obvious answer for the group, where one thread would handle the subscriber and eventual data processing while the other thread would handle the communication with the master and the protocol logic. ROS 2 offers its own variant of threading through the different *executors* available for development. The *executors* manage the resource use linked to the running of nodes and can be used to assign different threads to different nodes so they can run more or less in parallel.

The use of *executors* relies on all the code being arranged into nodes in order to be executed in parallel. Because of the complexity of the already developed code for the

slave logic at this point, the group opted to solve the
issue by using the *threading*-library available through C++, as it was easier to implement with the existing code.

Unlike the publisher, the subscriber code is a nearly non-modified version of the simple subscriber class from the ROS 2 documentation[18]. This class should, in theory, be easy to fit into the slave script had it not been for the issues with the program being in a locked state while the subscriber is listening.

```
class MinimalSubscriber : public rclcpp::Node           //This code is related to
    the ROS 2 implementation into the slave, and uses threading to run
{                                                       //both the subscriber and
    the slave logic, the name of the main-function must be changed to
public:                                                 //protocol(int i) to work
    properly with this implementation.
  MinimalSubscriber()
  : Node("minimal_subscriber")
  {
    subscription_ = this->create_subscription<std_msgs::msg::String>(
      "slaveTopic", 10, std::bind(&MinimalSubscriber::topic_callback, this, _1));
  }

private:
  void topic_callback(const std_msgs::msg::String::SharedPtr msg) const
  {
    //RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg->data.c_str());
    string received = msg->data.c_str();                      //format = num|num|num|
    num, not set in stone
                                                              //receiving up to four
    data fields
    string tmp;
    stringstream ss(received);                        //Processing data and ordering
    it for sending through modems
    vector<string> words;
    while(getline(ss,tmp,'|')){
        words.push_back(tmp);
    }
    data1 = words[0];
    data2 = words[1];
    data3 = words[2],
    data4 = words[3];
    //std::cout << "changing data" << std::endl;        //For debugging purposes
  }
  rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
 };
```

This subscriber listens to the topic *slaveTopic* and assigns all data it receives to the *data*-variable. The subscriber runs continually until the script is terminated, and the data is handled in another function of the slave script, where it is prepared for transmission to the master. Further processing of the data is handled by the master upon reception.

The data processing is also done in this subscriber, ordering the data into a set format that can easily be sent in a command string through the modems. Using this solution requires some tweaks to the code and includes threading.

```
int ros(int i)
```

```
{
  std::cout << "Subscriber starting\n";
  rclcpp::init(0, 0);
  rclcpp::spin(std::make_shared<MinimalSubscriber>());
  rclcpp::shutdown();

  return 0;
}

int main(){
    thread subscriber(ros,0);
    thread slave(protocol,0);

    subscriber.join();
    slave.join();
    return 0;
}
```

When this implementation is used two more functions must be included as shown above, and the normal *main()*-function changes name to *protocol(int i)*. This introduces the subscriber through one thread and initializes and runs the slave logic on another thread. This solution is only tested manually and not with the library, so problems may occur when they are run together.

4 Results

4.1 Library description

To handle the automatic transmission of data through JANUS and SDMSH, the group developed a library in C++ called *janusxsdm*. The library defines a class containing all necessary processes for automatic transmission to take place and is included in the master and slave scripts, where it is called every time the node needs to communicate with another node. This class defined in the library is named ”connection” and takes five arguments when declared:

1. modemIP - The ipv4 adress of the acoustic modem
2. JANUSPATH - The absolute or relative path from the code location to the JANUS executables
3. SDMPATH - The absolute or relative path from the code location to the sdmsh executable
4. rxPort - An arbitrary network port used to link *janus-rx* and sdmsh
5. txPort - An arbitrary network port used to link *janus-tx* and sdmsh

The class also has eight methods:

1. sdmconf() - sets the source level of the modem to its lowest setting.
2. setPreamble() - configures the correct preamble for JANUS.
3. sendSimple(message) - Transmits the value of the *message* variable.
4. listenSimple(message) - Writes receved message to the *message* variable.
5. listen(message, timeout) - Writes receved message to the *message* variable. Cancels listener upon timeout reached.
6. decode(buffer, message) - Designed to take a *buffer* of 16-bit JANUS-encoded data and decode to *message* variable.
7. printHeader() - Prints the frameheader of the SDM protocol.
8. sdmStop() - Sends an interrupt command to the modem.

A simple example of implementation:

```
#include<iostream>
#include<fstream>
#include<stdlib.h>
#include<string.h>
#include<chrono>
#include<unistd.h>
#include<signal.h>

#include "lib/janusxsdm/janusxsdm.cpp"

//Program paths
```

```
std::string JANUSPATH = "lib/janus-c-3.0.5/bin/";
std::string SDMPATH = "lib/sdmsh/";

using namespace std::chrono_literals; //Allows for use of 10s, 100ms ++ when
    specifying listener timeout

int main()
{
    std::cout << "Tester\n";
    //Declaring a connection object
    janusxsdm::connection myModem("192.168.0.189", JANUSPATH, SDMPATH, 9988, 9989);

    //Modem configuration
    myModem.sdmconf();
    myModem.setPreamble();

    std::string returnMessage; //Declaring a returnVariable

    myModem.sendSimple("Hello world!"); //Sending message "Hello world!"

    //Printing returnmessage on success or cancels if nothing is received in 30
    seconds
    if(myModem.listen(returnMessage, 30s))
    {
        std::cout << "Message was: " << returnMessage << std::endl;
    }
}
```

The example script above acts both as the sender and the receiver of the message "*Hello World!*. The first part is the standard way of configuring the modems and *SDMSH*. The connection object *myModem* sends the desired message through JANUS and the modems by using the method *sendSimple*. The last part acts as the listener for the message and has a time-out interval of 30 seconds. When the message is received, it is printed to the terminal.

When compiled and built this example script yields the following output when executed with a repeater in the other end.

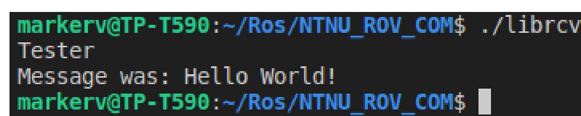


Figure 22: Result of a transmission using the library

4.2 Protocol

The group's protocol is based on a master-slave structure, where the master signals to the nodes in turns when they can transmit data. Processing of the data is handled both in the slaves and the master, where the slaves gather the data and structure it into a predetermined order. The master only formats the data it receives into the desired form for publishing on a ROS 2-topic, which is in turn displayed on an operator screen.

The master and the slaves go through a fixed process upon initialization. The communication itself is based on standardized commands. This is to ensure that all information kept by both the

master and the slave is correct, as well as make sure communication is possible between all the nodes. The initialization serves as a precursor to the actual network transmission, and the only data exchanged in this phase is considered configurational.

The process of initialization follows two distinct patterns depending on whether the slave already has a master registered from before or not. The two patterns are visualized as simplified flow charts in figure 23 and 24 below.

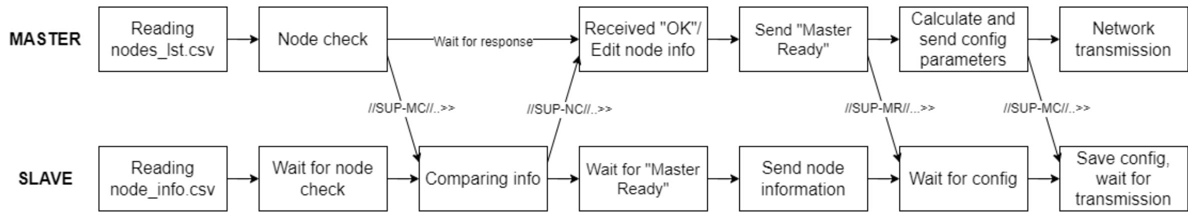


Figure 23: Chart showing normal initialization of master and slave

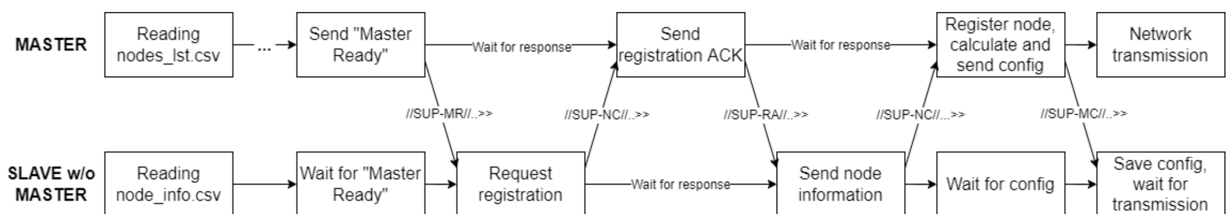


Figure 24: Chart showing initialization of master and a slave without a registered master

The relevant commands sent back and forth are also included in these diagrams to understand the system better.

The network transmission phase also follows its own distinct pattern, where the master asks all nodes in turn whether they have any data to transmit. The nodes will answer the master with any new data they have gathered and return to an idle state, waiting for the next transmission window.

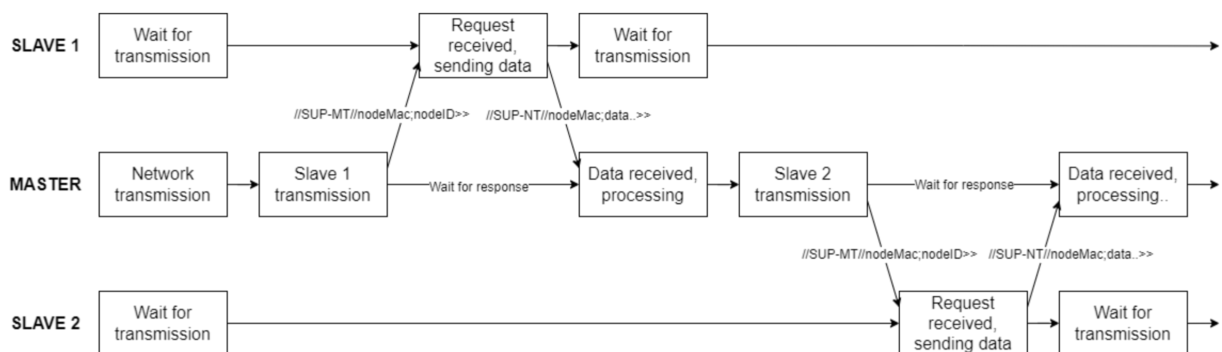


Figure 25: Chart showing an example of the flow of communication in the network during network transmission

In the end, the group did not end up with a full-fledged integration of ROS 2. The code is written and when it was tested with an earlier development of the JANUS configuration the transmission was a success. Data was published by ROS 2, gathered by a modem, transmitted to the other

modem via JANUS and collected through the ROS 2 subscriber. Unfortunately, the RPI's stopped functioning and the groups' opportunity to continuously test the product with ROS 2 vanished. The resulting ROS 2 product is a functioning publisher and a limited subscriber that only works if the program is threaded.

The complete code can be found on *www.github.com/markerv/NTNU_ROV_COM*

5 Discussion

5.1 Raspberry PI

The original project plan was to use the Raspberry PI 4 mainly because it features LPDDR4 SDRAM with options for larger capacities [25] and its support for Ubuntu Desktop 22.04 LTS [15]. Compared to the Raspberry PI 3B+, equipped with LPDDR2 SDRAM [24] and lacking support for the Desktop version of Ubuntu. However, due to all Raspberry PI models being out of stock worldwide, the group used two Raspberry PI 3B+'s which were borrowed from one of the group members.

The support for Ubuntu desktop was preferable in the development phase as it would facilitate easier testing directly on the hardware. However, in deploying the solution, one would prefer Ubuntu server since the graphical interface would have no use and contribute to increased power consumption in a system powered by batteries.

To circumvent the lack of GUI on the Raspberry PI the group used laptops with Ubuntu. Unfortunately, this is not optimal since the laptops have an AMD64 system architecture, while the Raspberry PI has ARM64. Due to the difference in system architecture, one can not guarantee that applications and libraries used in development on a laptop will work as intended on a Raspberry PI.

5.2 Networking

As mentioned earlier in the report, the modems are configured with static IP addresses from the manufacturer, and the Ethernet-IP of the PI3 was set to static. Alternatively, a DHCP server could have been set up on the Raspberry Pi, with the acoustic modem IP set to static while other units on the network would receive dynamic addresses. This would improve the system's modularity as other functionalities could be added and removed without risking IP conflicts. However, the group decided against it, as it would have been time-consuming to make it work properly, and manually assigning the addresses through the Ethernet adapters proved to be sufficient.

5.3 Brand specific implementation

JANUS has been an important part of the project, and a lot of the time has been allocated to JANUS. However, there is an alternative to using JANUS that would be a lot less time-consuming to implement; Using TCP streams with the EvoLogics standard, e.g. with netcat like the group did early in the project. This would limit the system to only work with EvoLogics modems, which is why JANUS is being used instead. Using JANUS means that the modem only has to recognize the preamble to know when there is an incoming message. This, in turn, means that a system can have modems from different manufacturers communicating with each other without issues.

Much of the work with JANUS revolved around the lack of or the poor quality of the documentation available related to implementation on the EvoLogics modems. While much data and resources are available on januswiki.com, some of it is outdated and not in sync with the latest version. This caused the group to spend a good portion of time trying commands and flags that no longer

existed.

A concrete example of this was in the early phases when the group was still getting acclimatized to using JANUS and the flags related to the executables. Trying to find the flag for appending data to the protocol header, the group consulted the read-me available on januswiki.com, appended on page 60. In the read-me, the flag *–packet-payload* was assigned as the flag for appending data to the header, but when tried, the appended data did not follow the header. It was by a coincidence that one group member executed the tx-executable wrong, causing it to return a list of usable flags, and the group found the *–packet-cargo*-flag, which turned out to be the correct flag. While this might be the result of an update where the read-me was not reviewed, it would have saved the group a lot of time had the documentation been up to date.

5.4 Implementation difficulties

The project utilized the Evologics 18/32 USBL modems, but the client also had access to modems made by SubNero. The reasoning was that they had access to three modems by Evologics and only two from SubNero and that at least three were required to create a network. However, during the project, only two modems were used as the difficulty of developing a solution for the Evologics modems slowed the project progression.

The implementation guide offered by Evologics for JANUS with SDMSH uses executables with configuration flags. It offered no guidance on implementing its SDM library, and the libraries themselves were scarcely commented on. The JANUS libraries were also barely documented and offered little direction on how to use it.

An attempt was made to use the SDM and JANUS libraries, but the lack of documentation made the solution impossible to implement, given the group's ability and time frame. Instead, a solution where the code launches the JANUS and SDMSH executables were implemented.

Launching executables from code as opposed to direct integration with libraries resulted in less control over the executing process and is a sub-optimal solution. For future projects, the group recommends either a project team with greater experience with C and C++ or to change to another solution. The SubNero modems are a good candidate as they use the UnetStack framework, which features good guides for the use of their libraries in addition to supplying a variety of tools for monitoring and simulation.[22]

5.5 Limited access

During the first few weeks of working with the modems in a room with a water tank, the group was informed that the room would be inaccessible for three weeks due to lab work. This resulted in the group performing tests on the modems exclusively in air. Mostly, it was not an issue, but the quality of the transmission went down due to signal noise from the surroundings. With the SubNero modems, this would have posed less of an issue because it uses the UnetStack framework with support for UnetSim. UnetSim allows for the simulation of underwater networks, where the user can simulate an entire network from one computer. [22]

Towards the end of the project, another student required the modems. In the end, the group could use the modems for three days on even weeks (week 12, 14, 16 and 18) and two days the following

week. Although it did not affect the project work in a significant way, it did require the group to adjust and make changes to the plan. This is another issue that would have been completely solved with the use of UnetSim.

The group had limited access to the modems in the sense that they had to be retrieved and delivered to the same office each day. The group did not have access to the office, so someone had to be there when the group wanted to pick up or drop off the modems. It is reasonable to want them locked in an office overnight with regards to the price and importance of the modems. Although, this meant that the group could only perform work on the modems when someone else's schedule allowed for it. In addition to this, the modems had to be reconfigured with the right settings every time the group was working on them since someone else might have used them in the meantime.

5.6 The Protocol

5.6.1 TDMA

The group had originally planned for the protocol to be fully based on a TDMA-type protocol, where the master's role would be minimal, and all nodes would keep track of the time individually and automatically start transmission themselves. However, this proved to be harder than anticipated to implement with regards to synchronizing the clocks of the nodes with a signal from the master, given that the distance between the master and each node is not the same. Differing distances between the nodes meant that synchronizing all nodes simultaneously would require a lot of work, which was therefore scrapped.

5.6.2 Testing

Due to the problems the group had while transmitting data and valid packets through the air, a proper test of the developed protocol scripts was hard to accomplish. This, coupled with difficulties utilizing the *listen*-function from the library, made the protocol testing less productive than the group would have wanted.

The only tests of the protocols were carried out by manually changing the *response* variable returned by the *janus_rx*-function for every test. This means that the automatic aspect of the protocols was never properly tested, and the logic was never tested in other conditions than optimal. Troubleshooting was only limited to the issues unveiled by the manual tests so that no guarantee can be given for the functionality of the protocol in all types of circumstances. As a direct consequence, the logic in both the master and the slaves may be incompletely implemented in the scripts or may even be flawed altogether.

5.7 ROS 2, experience and future solutions

ROS 2 was an important part of this project, and a lot of time was spent on it. Configuring Raspberry PIs and JANUS scripts to collaborate with ROS 2 was especially time-consuming. As mentioned earlier, the RPIs suddenly stopped running ROS 2. This was attributed to them being used a lot before the project, and it is assumed that a new pair would work. This assumption stems from the fact that the ROS 2 code still functioned on the groups' private computers.

In the end, the product did not involve ROS 2. Despite this, there was developed functioning code. As mentioned in the "Results" chapter, the publisher performs well while the subscriber obstructs every other program. Had more time been available, the group would have solved this with executors. The "Implementation" chapter mentioned why the group opted out of utilizing executors. It would still be a viable solution given more time, and the ROS 2 galactic documentation page gives a lot of insight into how executors could be used. Mentioning that "The MultiThreadedExecutor creates a configurable number of threads to allow for processing multiple messages or events in parallel". [16]

In short, the MultiThreadedExecutor relies on "callback groups". These "callback groups" are stored as class members. That is how the executor can trigger them. Then one specifies which group a subscriber belongs to, and the executor uses its threads to the callback as many groups as one need, within certain limits. The scheduling is quite easy to understand as well. The rule of thumb is FIFO, but in some cases, the schedule is handled in a "Round-robin fashion". The flow diagram explains the semantics. [16]

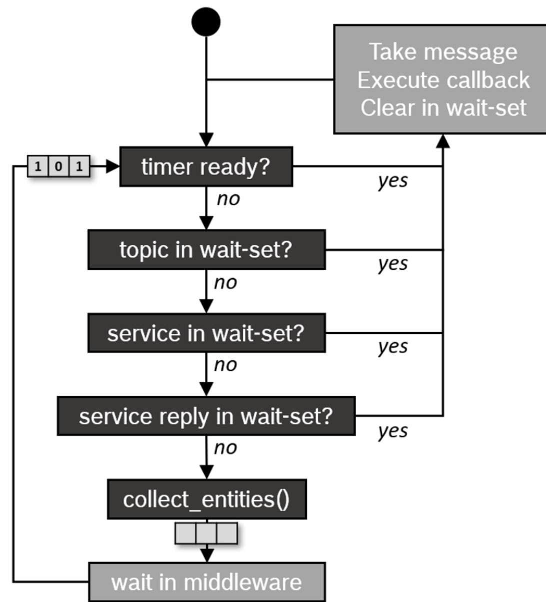


Figure 26: Executor schedule [16].

It should not be too big of a challenge for future groups to implement the subscriber through executors. The logic is sound, so the biggest issue would probably be getting acquainted and comfortable with ROS.

The alternative solution to this was using the *threading*-library belonging to C++, which became the solution the group chose. While being a working solution, it is not an optimal solution. Given that the *janusxsdm*-library uses the forking of processes to function it may not be optimal to introduce more threads into the code. The few tests the group could run using this solution did not uncover any problems. However, since the tests were conducted mostly manually and for short periods, any long-term stability issues have yet to surface.

This, combined with the problems surrounding the Raspberry Pis and running of ROS 2, led the group to drop the implementation from the final code draft altogether. Instead, the parts of the code relating to ROS 2 have been commented out for eventual later development, as the group did not want to include code that was not tested thoroughly enough but still enable future developers to have a crack at it.

6 Conclusion

To determine what was accomplished in the course of this project and whether it satisfies the set objective, the most important question is if the project follows the original goal. At the beginning of the year, the group set out to:

Develop a standardized solution for underwater wireless networking using Evologics S2C Underwater Acoustic Modems as receiver/transmitter modules to exchange all types of data between multiple ROS 2 based systems.

The objective included an agreement with the client that, if possible, the group would expand the project scope. This expansion involves incorporating libraries, bit error correction, multiple transmission protocols and GSM interfacing.

The project resulted in a standardized solution for underwater transmission, albeit without data sent between multiple ROS 2 based systems. A code that performed the transmissions manually did succeed in integrating with the ROS 2 system. The transition to automatic transmission would probably have been smoother with a better opportunity to use the modems on the group's own accord, as well as access to the intended RPI 4s.

The protocol is functional, and the group did a good job using JANUS with the modems, considering how little documentation was available. In addition, the advancement from Evologics AMA to JANUS made it possible for group members with differing grasp of the technology to develop their understanding alongside one another.

To conclude this project, it would make sense to look at some individual aspects. The group is satisfied with the effort that has been put into the project. The number of hours and the quality of the collaboration has been great. Everyone has communicated well and done their utmost to contribute. Each of the group members has acquired new skills and valuable experience. Therefore it should be noted that the group is pleased with the work in and of itself.

On the other hand, the resulting product has not been as good as the group anticipated when the project began. Unforeseen difficulties mentioned previously in the report made for a disappointing finished product. Even though it delivered the promised attributes and the group was able to fulfill their task, there is dissatisfaction among the members regarding the result.

References

[1] BlueRobotics. ‘Ardusub’. (n.d), [Online]. Available: https://www.ardusub.com/. (accessed: 02.05.2022).

[2] BlueRobotics. ‘Bluerov2’. (n.d), [Online]. Available: https://bluerobotics.com/store/rov/bluerov2/. (accessed: 02.05.2022).

[3] BlueRobotics. ‘Bluerov2 heavy configuration retrofit kit’. (n.d), [Online]. Available: https://bluerobotics.com/store/rov/bluerov2-upgrade-kits/brov2-heavy-retrofit-r1-rp/. (accessed: 02.05.2022).

[4] BlueRobotics. ‘Lumen subsea light’. (n.d), [Online]. Available: https://bluerobotics.com/store/thrusters/lights/lumen-sets-r2-rp/. (accessed: 02.05.2022).

[5] BlueRobotics. ‘Newton subsea gripper’. (n.d), [Online]. Available: https://bluerobotics.com/store/rov/bluerov2-accessories/newton-gripper-asm-r2-rp/. (accessed: 02.05.2022).

[6] Brian Gerkey. ‘Why ros 2?’ (2015), [Online]. Available: https://design.ros2.org/articles/why_ros2.html. (accessed: 02.05.2022).

[7] Digi-Key. ‘Raspberry pi 3 model b+’. (n.d.), [Online]. Available: https://www.digikey.no/no/products/detail/raspberry-pi/RASPBERRY-PI-3-MODEL-B/8571724. (accessed: 08.05.2022).

[8] EvoLogics GmbH. (2018), [Online]. Available: https://evologics.de/product/s2c-r-18-34-usbl-2. (accessed: 08.03.2022).

[9] EvoLogics GmbH. (2022), [Online]. Available: https://evologics.de/software/ama. (accessed: 15.05.2022).

[10] EvoLogics GmbH. ‘Sdmsh wiki’. (9.01.2020), [Online]. Available: https://github.com/EvoLogics/sdmsh/wiki. (accessed: 18.05.2022).

[11] D. Hendler. ‘Using cidr notation in cloud networks’. (21.04.2021), [Online]. Available: https://docs.rackspace.com/support/how-to/using-cidr-notation-in-cloud-networks. (accessed 18.05.2022).

[12] ‘Janus-compile-and-run’. (2020), [Online]. Available: https://github.com/EvoLogics/sdmsh/wiki/SDM-:-JANUS-Compile-and-Run. (accessed: 12.05.2022).

[13] JAOM. ‘Usbl underwater positioning - concept and basic’. (2021), [Online]. Available: https://www.youtube.com/watch?v=op6M6C7kKgE. (accessed: 14.05.2022).

[14] J. Kurose and K. Ross, *Computer networking: A top-down approach, global edition*, 7th ed. London, England: Pearson Education, Jan. 2021.

[15] C. Ltd. ‘Install ubuntu on a raspberry pi’. (n.d.), [Online]. Available: https://ubuntu.com/download/raspberry-pi. (accessed: 06.05.2022).

[16] Open Robotics. ‘Executors’. (2022), [Online]. Available: https://docs.ros.org/en/galactic/Concepts/About-Executors.html. (accessed: 15.05.2022).

[17] Open Robotics. ‘Rclcpp namespace reference’. (n.d), [Online]. Available: https://docs.ros2.org/galactic/api/rclcpp/namespacerclcpp.html#ad48c7a9cc4fa34989a0849d708d8f7de. (accessed: 14.05.2022).

[18] Open Robotics. ‘Writing a simple publisher and subscriber (c++)’. (n.d), [Online]. Available: https://docs.ros.org/en/galactic/Tutorials/Writing-A-Simple-Cpp-Publisher-And-Subscriber.html. (accessed: 14.05.2022).

[19] J. Potter, J. Alves, D. Green, G. Zappa, I. Nissen and K. McCoy, 'The janus underwater communications standard', in *2014 Underwater Communications and Networking (UComms)*, 2014, pp. 1–4. DOI: 10.1109/UComms.2014.7017134.

[20] M. Stojanovic. 'Underwater acoustic communication'. (2003), [Online]. Available: http://web.mit.edu/millitsa/www/resources/pdfs/ency.pdf. (accessed: 13.05.2022).

[21] B. Stroustrup. 'Lecture: The essence of c++', Youtube. (6.05.2014), [Online]. Available: https://www.youtube.com/watch?v=86xWVb4XIyE. (accessed: 07.05.2022).

[22] SubNero. (2022), [Online]. Available: https://subnero.com/products/unet.html. (accessed: 14.05.2022).

[23] The Raspberry Pi Foundation. 'Install raspberry pi os using raspberry pi imager'. (2022), [Online]. Available: https://www.raspberrypi.com/software/. (accessed: 15.05.2022).

[24] The Raspberry Pi Foundation. 'Raspberry pi 3 model b+'. (n.d.), [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/. (accessed: 06.05.2022).

[25] The Raspberry Pi Foundation. 'Raspberry pi 4 tech specs'. (n.d.), [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/. (accessed: 06.05.2022).

[26] The University of Rhode Island. (2021), [Online]. Available: https://dosits.org/people-and-sound/history-of-underwater-acoustics/the-first-studies-of-underwater-acoustics-the-1800s/. (accessed: 19.05.2022).

[27] The University of Rhode Island. (2021), [Online]. Available: https://dosits.org/people-and-%20sound/history-of-underwater-acoustics/the-discovery-of-underwater-acoustics-pre-1800s/. (accessed: 13.05.2022).

[28] 'Underwater usbl positioning systems'. (n.d.), [Online]. Available: https://evologics.de/usbl. (accessed: 12.05.2022).

[29] Wikipedia contributors. 'Netcat'. (26.04.2022), [Online]. Available: https://en.wikipedia.org/w/index.php?title=Netcat&oldid=1084736372. (accessed: 12.05.2022).

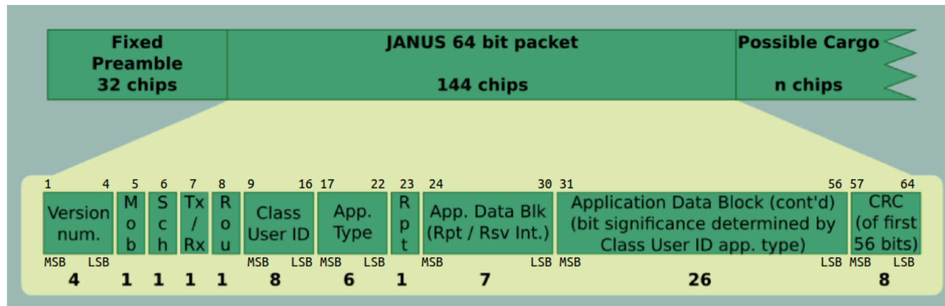
[30] Wikipedia contributors. 'Robot operating system'. (5.04.2022), [Online]. Available: https://en.wikipedia.org/w/index.php?title=Robot_Operating_System&oldid=1081050080. (accessed: 12.05.2022).

[31] Wikipedia contributors. 'Visual studio code'. (3.05.2022), [Online]. Available: https://en.wikipedia.org/w/index.php?title=Visual_Studio_Code&oldid=1086042161. (accessed: 12.05.2022).

[32] J. Wilson. 'How to configure static ip address on ubuntu 20.04'. (2021), [Online]. Available: https://www.rosehosting.com/blog/how-to-configure-static-ip-address-on-ubuntu-20-04. (accessed: 08.05.2022).

Appendix

A JANUS BIT ALLOCATION TABLE



The JANUS baseline bit allocation table is available through januswiki.com after logging in.

B JANUS README

README - JANUS Tool Kit 3.0.1

Giovanni Zappa

2013-03-25

JANUS Tool-kit version 3.0.1 provides a Reference implementation of JANUS standard version 3 transmitter and a sample implementation of decoder. Latest version of the JANUS Reference implementation can be downloaded prior registration on http://www.januswiki.com on *File Galleries* > *Code repository* > *Most Recent Toolkit*.

It is coded in "C" and in Matlab. The sources of the "C" version are available in the file "janus-c-3.0.1.zip", has to be build using *CMake* and it only requires libraries *FFTW3*. Matlab code is available on the same site in the file "janus-m-3.0.1.zip".

On http://www.januswiki.com is possible to find documentation on JANUS standard.

"C" code installation

The "C" code implementation is composed of a library "libjanus.a" providing all the basic functionality, two sample command line implementation "*janus-tx*" and "*janus-rx*" respectively for generating the transmission waveform and decoding it, plus a plugin for the "reference implementation message 16" in the "*libplugin_016_00.so*".

If CMake finds a Matlab installation will be also generated "*janus_tx_mex*" and "*janus_rx_mex*" to be used with Matlab.

B JANUS README

Compilation on Linux or other UNIX

The "C" code has been successfully compiled on different Linux version.

Once downloaded "janus-c-3.0.1.zip" form `http://www.januswiki.com`, unzip the file.
Use CMake for the configuration. CMake command line accepts the standard as well as custom variables.
Most useful standard variables are:

- `CMAKE_BUILD_TYPE` which can be `Release` or `Debug`.
- `CMAKE_INSTALL_PREFIX` which allow to install JANUS on a path different from the default `/usr/local`.

CMake Custom command-line options are:

- specify the Matlab installation directory if not in the default path
  eg: `-DMATLAB_DIR=/usr/local/matlab80`
- force using single precision floating point where possible
  `-DJANUS_REAL_SINGLE=1`
- use single precision floating point only in the FFTs operations `-DJANUS_FFTW_SINGLE=1`
- avoiding the use of function *alloca* (broken on some systems) `-DJANUS_WITHOUT_ALLOCA=1`
- to avoid compiling on 32 bits machine "position-independent code" (PIC) add `-DJANUS_NO_PIC=1`

A possible compiling procedure is:

```
unzip -v janus-c-3.0.1.zip
cd janus-c-3.0.1
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make
```

and finally

```
make install
```

The last command might need superuser privileges if installed outside the user path.

JANUS executables need to find the pulgins describing the behaviour of the different JANUS "class id" and "application type", **it is strongly recommended to perform the installation of JANUS**.
Otherwise will be required to add the directory containing the plugins to the environment variable `LD_LIBRARY_PATH` or to system "ld.so.conf".
Those operation might be anyway needed on some broken Linux versions.

Compilation on Windows

JANUS reference implementation has been installed on Windows using CMake 2.6 or newer and "Visual Studio Express 2012".

FFTW compilation under Windows

JANUS took-kit needs to link to the FFW3 library. There are some pre-compiled version or is possible to compile it.
The sources (fftw-3.3.3.tar.gz) can be downloaded from the website `http://www.fftw.org/download.html`.
The Visual Studio solution (fftw-3.3-libs-visual-studio-2010.zip) or the pre-compiled versions are available at `http://www.fftw.org/install/windows.html`
If decided to compile the FFTW3 libraries from sources, follow those steps:

B JANUS README

```
...\ FFTW-3.3.3
├─aclocal.m4
├─api
│ └─api.h
├─fftw-3.3-libs
│ └─libfftw-3.3
│   ├─libfftw-3.3.sln
│   └─x64
│     └─Release
```

- Unpack the sources `fftw-3.3.3.tar.gz` as in the figure.
- Inside the extracted folder `fftw-3.3.3` copy the content of file `fftw-3.3-libs-visual-studio-2010.zip`.
- Following the instructions contained into the Readme.txt, open with Visual studio the solution file `libffftw-3.3.sln` in the directory `ffrw-3.3-libs`.
- Build the solution in the Release version.
- Choose an installation directory for the libfftw3 which for simplicity from now is called `fftw3_install_directory` and create inside the directories `bin`, `include` and `lib`.
- Copy inside `bin` the compiled file `fftw-3.3.3\ffrw-3.3-libs\libfftw-3.3\x64\Release\libfftw-3.3.dll`. In case of 32 bits system the structure will be `fftw-3.3.3\ffrw-3.3-libs\libfftw-3.3\Release\libfftw-3.3.dll`.
- Copy in `include` the contents of `fftw-3.3.3\api`
  Copy in `lib` the file from `fftw-3.3.3\ffrw-3.3-libs\libfftw-3.3\x64\Release\libfftw-3.3.lib`.
  After the installation the folder should present the following structure:

```
...\ fftw3_install_directory
├─bin
│   libfftw-3.3.dll
│
├─include
│   api.h
│   apiplan.c
│   configure.c
│   execute-dft-c2r.c
│   ...
│   plan-r2r.c
│   print-plan.c
│   rdft2-pad.c
│   the-planner.c
│   version.c
│   x77.h
│
└─lib
    libfftw-3.3.lib
```

- To allow CMake and the operating system to find the FFTW3 libraries open `regedit` (just type "regedit" from search or execute of your start menu).
- Open the tree called `HKEY_CURRENT_USER` or `HKEY_CLASSES_ROOT` in case of a system wide installation which requires "administrator" privileges.
- Look for `Environment`.
- Create a new "String Value" and set Name to `FFTW_DIR` and Data to the full path of your `fftw3_install_directory`.
- Edit on regedit the entry `PATH` and prepend "`fftw3_install_directory\bin;`" to your existing PATH.

B JANUS README

JANUS compilation

As well as for Linux the source file "janus-c-3.0.1.zip" can be downloaded form http://www.januswiki.com.

- Extract `janus-c-3.0.1.zip`.
- Run CMake-gui; with `Browse Source` find the folder where JANUS source is unpacked.
- Copy the source path on the following line (`Where to build the binaries`) and append `\build`.
- Press `Configure`, eventually confirm you want to create the `build` folder.
- Select "Visual Studio 11" (if possible Win64) or the appropriate project type for your compiler on the new appearing window and click on `Finish`.
- It is possible to change some of CMake internal variables from the gui.
  For example `CMAKE_INSTALL_PREFIX` allows to select the installation directory. For those variables refer to Compilation on Linux or other UNIX.
- Press `Generate` and close CMake.
- Run Visual Studio and open the solution file `JANUS.sln` inside the build directory specified to CMake.
- Select your `Solution Configuration`, `Solution Platform` and run `Build Solution`.
- To install JANUS tool-kit, select `INSTALL` in the solution explorer and with right mouse key build it.

Matlab code

Matlab code is stored in the "janus-m-3.0.1.zip". Unpack the zip file and move with Matlab on folder `janus-m-3.0.1` or use `addpath()` command to be able to use JANUS functions.

Building the "C" source on a system with a Matlab installation (might be necessary to specify `MATLAB_DIR=<path_to_matlab_directory>`) will generate the files "*janus_tx_mex*" and "*janus_rx_mex*" which allow to run the "C" compiled code inside Matlab.

As usual the directory containing those files needs to be added with `addpath()` or be present in working directory.

Using JANUS Tool-Kit

JANUS tool kit interface use similar option whenever possible for the transmitter, receiver either in Matlab and in "C" version.

Typically "C" command-line options start with `"--"` and words are separated with `"-"`, while on Matlab use `"_"` as word separator.
Eg: The "C" option `--stream-driver` on Matlab becomes `'stream_driver'`

Command-line options

Following options control the JANUS modulation and are common between transmitter and receiver:

"C" option	Matlab option	Function	default
--pset-file	pset_file	Parameter Set File	
--pset-id	pset_id	Parameter Set Identifier	0
--pset-center-freq	pset_center_freq	Center Frequency (Hz)	0
--pset-bandwidth	pset_bandwidth	Bandwidth (Hz)	0
--chip-len-exp	chip_len_exp	Chip Length Dyadic Exponent	0
--sequence-32-chips	sequence_32_chips	Initial sequence of 32 chips (hex)	0x6BC4D788

- `pset-file` must point to a valid `"CSV"` file with standard central frequency and band.

B JANUS README

- pset-id must select a valid entry of *pset-file*.

all the following options are not required:

- pset-center-freq override central frequency. Changing from default value may cause not only incompatibility with other devices but hardware damages on transmitters.
- pset-bandwidth override available bandwidth. As for previous option is strongly suggested to not use it.
- chip-len-exp force chip duration to multiplied by $2^{chip-len-exp}$, slowing transmission rate but probably increasing reliability of communications. Same value should be set on all the other devices.
- sequence-32-chips change the $0/1$ sequence transmitted in the first 32 chips acquisition signal. Different settings will cause incompatibility with other devices.

The following options allow the user to control the user interface and the waveform generation:

”C” option	Matlab option	Function	default
--verbose	verbose	Verbose level	0
--stream-driver	stream_driver	Stream Drv (nul/alsa/pulse/raw/wav/wmm)	wav
--stream-driver-args	stream_driver_args	Stream Driver Arguments	janus.wav
--stream-fs	stream_fs	Stream Sampling Frequency (Hz)	44100
--stream-format	stream_format	Stream Format	S16
--stream-channels	stream_channels	Stream Channels configuration	1
--stream-channel	stream_channel	Stream Active Channel	0
--stream-passband	stream_passband	Stream Passband signal	1
--config-file	config_file	Configuration file	

- verbose allows to get more internal informations, 0 is the minimum, 1 is consider normal verbosity and 2 more information including raw chips probability and plotting in Matlab.
- stream-driver select the device media for input/output stream (see the following media table).
- stream-driver-args media configuration (see the following media table).

Driver	Typical option	Notes	Matlab	”C”	”MEX”
alsa	default | plughw:0,0	ALSA Linux sound system		X	
fifo		Only useful if at library level			
mat	janus.mat	Save/Load Matlab file	X		
mem	janus	Use Matlab variable	X		
null		Fake dirver only for testing	X	X	X
pulse		PULSE sound system		X	
raw	janus.raw	Save/Load raw binary file	X	X	X
wav	janus.wav	Save/Load RIFF WAV file	X	X	X
wmm	-1	Windows Multimedia sound system		X	

- stream-fs stream sampling frequency, must be compatible with the selected media. Needs to satisfy Nyquist sampling theorem (stream-fs should be a least twice the maximum frequency).
- stream-format sample binary format representation (as in the following table):

Format	Option value
Signed 8 bit PCM	S8
Signed 10 bit PCM	S10
Signed 12 bit PCM	S12
Signed 14 bit PCM	S14
Signed 16 bit PCM	S16
Signed 24 bit PCM	S24
Signed 24 bit PCM in LSB of 32 Bit words	S24_32
Signed 32 bit PCM	S32
32 bit IEEE floating point [-1.0, 1.0]	FLOAT
64 bit IEEE floating point [-1.0, 1.0]	DOUBLE

B JANUS README

- `stream-channels` number of channels the of the input/output stream.
- `stream-channel` which channel number should be used.
- `stream-passband` use passband signal or baseband
- `config-file` loads file with all input configurations.

Transmitter specific options are:

”C” option	Matlab option	Function	default
--pad	pad	Enable/Disable Padding of output	1
--wut	wut	Enable/Disable Wake Up Tones	0
--stream-amp	stream_amp	Stream Amplitude Factor (0.0 - 1.0]	0.95
--stream-mul	stream_mul	Stream samples multiple of given number	1
--packet-mobility	packet_mobility	Mobility	0
--packet-tx-rx	packet_tx_rx	Tx/Rx capability	1
--packet-forward	packet_forward	Forwarding capability	0
--packet-class-id	packet_class_id	Class User Identifier	16
--packet-app-type	packet_app_type	Application Type	0
--packet-reserv-time	packet_reserv_time	Reservation Time	0.0
--packet-repeat-int	packet_repeat_int	Repeat Interval	0.0
--packet-app-data	packet_app_data	Application Data	0x0000000
--packet-app-data-fields	packet_app_data_fields	Application Data Fields	
--packet-payload	packet_payload	Optional Payload	

- `pad` insert short silence at the beginning and end of signal.
- `wut` enable/disable transmission of wake-up tone. (It may create false detection with consequent loss of the following packet with sample decoder implementation).
- `stream-amp` stream amplification factor.
- `stream-mul` force generated signal to have multiple of the given integer number of samples (needed for some signal generators).
- `packet-mobility` sets mobility packet flag.
- `packet-tx-rx` sets packet flag indicating the capability of decoding JANUS signals.
- `packet-forward` sets packet flag indicating the capability of forwarding packets.
- `packet-class-id` sets class user identifier, selecting application or nation $[0 \sim 255]$. Togeither with the *application type* determine the meaning of *application data* field and payload.
- `packet-app-type` sets application type for the specific *packet-class-id* $[0 \sim 63]$.
- `packet-reserv-time` JANUS packet reserving channel for the requested number of seconds. This option is incompatible with *packet-repeat-int*.
- `packet-repeat-int` a JANUS packet will be retransmitted after the requested number of seconds. This option is incompatible with *packet-reserv-time*
- `packet-app-data` force application data field to be set with the provided bits in hexadecimal form.
- `packet-app-data-fields` comma separated list of couples $< field >=< value >$ interpreted by the plugin specified by *packet-class-id* and *application data* to fill the *app-data* field
- `packet-payload` string to encode into the following payload.

”C” option	Matlab option	Function	default
--doppler-correction	doppler_correction	Enabled/Disabled Doppler Correction	1
--doppler-max-speed	doppler_max_speed	Doppler Correction Maximum Speed [m/s]	5
--channel-spectrogram	channel_spectrogram	Enabled/Disabled channel spectrogram	1

B JANUS README

- doppler-correction enable / disable Doppler correction.
- doppler-max-speed if Doppler correction enabled, limit speed estimation to the specified value expressed in $[m/s]$.
- channel-spectrogram enable / disable channel estimation around the JANUS 26 carrier frequencies and Media Access Control (MAC).

Examples

The following command line examples are provided as examples. Unfortunately different version of command line interpreters may need different quotations.

To be able to run the same examples on Windows and Linux move to the installation directory, your binary CMAKE_INSTALL_PREFIX/bin or default /usr/local/bin on Linux or to your CMAKE_INSTALL_PREFIX/bin if specified or the default C:\Program Files\JANUS\bin. Only to be able to run the following examples, on Windows is necessary to have the directory \tmp, while on Linux to add the path the binary directory:
export PATH="<CMAKE_INSTALL_PREFIX>/bin:$PATH".

"C" version

Basic examples

```
> janus-tx
```
To get the command line help of transmitter.

```
> janus-rx
```
To get the command line help of receiver.

```
> janus-tx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
```
Packet generation. This is the minimalistic command line, it relies on the default values. The result is a 'WAV' file called janus.wav on the same directory of the executable. This command will fail in case the user has not write premission in installation path.

```
> janus-tx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver-args /tmp/test_janus.wav
```
In this case the 'WAV' file created is /tmp/test_janus.wav overcoming the permission problem of previous example.

```
> janus-rx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver-args /tmp/test_janus.wav
```
To decode the previously created file /tmp/test_janus.wav.

Receiver commands

```
> janus-rx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver-args /tmp/test_janus.wav --verbose 2
```
To decode the previously created file /tmp/test_janus.wav with increased verbosity, showing the single chip error probabilities.

```
> janus-rx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver-args /tmp/test_janus.wav --doppler-correction 0
```
To decode the previously created file /tmp/test_janus.wav without Doppler correction.

```
> janus-rx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver-args /tmp/test_janus.wav --verbose 1
    --doppler-max-speed 3
```

To decode the previously created file `/tmp/test_janus.wav` with Doppler correction estimated in the range $[-3 \sim 3]m/s$.

```
> janus-rx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver-args /tmp/test_janus.wav --channel-spectrogram 0
```

To decode the previously created file `/tmp/test_janus.wav` without running MAC check.

Playing with different media

Connecting with a cable your audio output with audio line-in, after verifying input and output levels, is possible to transmit and decode real-time JANUS signals.

On Linux and other UNIX

```
> janus-tx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver alsa --stream-driver-args default --stream-format S16
    --stream-fs 48000
```

To play the JANUS waveform using *alsa* sound though the `default` device using a sampling rate of 48KHz and sample format signed integer represented with 16 bits.

```
> janus-rx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver alsa --stream-driver-args default --stream-format S32
    --stream-fs 48000
```

To decode the JANUS waveform using *alsa* sound though the `default` device using a sampling rate of 48KHz and sample format signed integer represented with 32 bits.

```
> janus-tx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver pulse --stream-driver-args "" --stream-fs 48000
    --stream-format S16
```

To play the JANUS waveform using *pulse* sound though the `default` device using a sampling rate of 48KHz and sample format signed integer represented with 16 bits.

```
> janus-rx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver pulse --stream-driver-args "" --stream-fs 48000
    --stream-format S32
```

To decode the JANUS waveform using *pulse* sound though the `default` device using a sampling rate of 48KHz and sample format signed integer represented with 32 bits.

On Windows

```
> janus-tx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver wmm --stream-driver-args -1 --stream-fs 48000
    --stream-format S16
```

To play the JANUS waveform using *pulse* sound though the `default` device using a sampling rate of 48KHz and sample format signed integer represented with 16 bits.

```
> janus-rx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver wmm --stream-driver-args -1 --stream-fs 48000
    --stream-format S32
```

To decode the JANUS waveform using *pulse* sound though the `default` device using a sampling rate of 48KHz and sample format signed integer represented with 32 bits.

B JANUS README

Physical layer options

```
> janus-tx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver-args /tmp/test_janus.wav --chip-len-exp 3
```

The generate waveform use the dyadic chip length option with exponent 3, chips length will be $2^3 = 8$ times longer.

```
> janus-tx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver-args /tmp/test_janus.wav --stream-channels 2
    --stream-channel 0
```

Generates a stereo waveform modulating the signal only on first channel.

```
> janus-tx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver-args /tmp/test_janus.wav --pad 0
```

Generates a waveform without padding at the beginning and the end.

```
> janus-tx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver-args /tmp/test_janus.wav --stream-passband 0
```

Generates a stereo waveform modulated in baseband.

JANUS packet options

```
> janus-tx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver-args /tmp/test_janus.wav --packet-mobility 1
    --packet-tx-rx 1
```

Sets the into the packet mobility and ability to receive JANUS signal flags.

On Linux

```
> janus-tx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver-args /tmp/test_janus.wav --packet-class-id 16
    --packet-app-type 0 --packet-app-data-fields 'Station
    Identifier=1,Parameter Set Identifier=17'
```

Force generation of JANUS packet of *class id* 16 and *application type* 0 (those are already default values). Filling the application fields with *"Station Identifier"=1* and *"Parameter Set Identifier"=17*. It cause loading the plugin library file `libplugin_016_00.so` if loadable (either in a standard library directory or in folder contained in `LD_LIBRARY_PATH` or installed under the specified `CMAKE_INSTALL_PREFIX`). Plug-ins are not mandatory, are needed to interpret correctly the field name in *packet-app-data-fields* and especially for decoding to know the eventual payload length. The plugin name is identified by *packet class identifier* and *packet application type* in this case respectively 16 and 0.

```
> janus-tx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver-args /tmp/test_janus.wav --packet-class-id 16
    --packet-app-type 0 --packet-app-data-fields 'Station
    Identifier=1,Parameter Set Identifier=17' --packet-payload 1234
```

Force generation of JANUS packet as in the previous example with payload text "1234"

On Windows

```
> janus-tx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver-args /tmp/test_janus.wav --packet-class-id 16
    --packet-app-type 0 --packet-app-data-fields "Station
    Identifier=1,Parameter Set Identifier=17"
```

Force generation of JANUS packet of *class id* 16 and *application type* 0 (those are already default values). Filling the application fields with *"Station Identifier"=1* and *"Parameter Set Identifier"=17*. It cause loading the plugin library file `libplugin_016_00.dll` if loadable (either in a standard library directory or installed under the specified `CMAKE_INSTALL_PREFIX`). Plug-ins are not mandatory, are needed to interpret correctly the field name in *packet-app-data-fields* and especially for decoding to know the eventual payload length. The plugin name is identified by *packet class identifier* and *packet application type* in this case respectively 16 and 0.

```
> janus-tx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver-args /tmp/test_janus.wav --packet-class-id 16
    --packet-app-type 0 --packet-app-data-fields "Station
    Identifier=1,Parameter Set Identifier=17" --packet-payload 1234
```

Force generation of JANUS packet as in the previous example with payload text "1234"

Again system independent

```
> janus-tx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver-args /tmp/test_janus.wav --packet-class-id 16
    --packet-app-type 0 --packet-app-data 0x000040443 --packet-payload
    1234
```

As in the above example, where application field are directly passed to command line.

```
> janus-tx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver-args /tmp/test_janus.wav --packet-reserv-time 1
```

JANUS packet request reservation time of 1 second.

```
> janus-tx --pset-file ../share/janus/etc/parameter_sets.csv --pset-id 1
    --stream-driver-args /tmp/test_janus.wav --packet-repeat-int 10
```

Announcing a retransmission of a JANUS packet in 10 seconds, especially useful for beacon applications.

Examples for Matlab version

Basic examples

As already mentioned between the "C" and Matlab version, option have same name and behaviour, differentiating only typographically.

To be able to run Matlab functions, move to the installed Matlab source directory or add the same to the path (`addpath`).

```
> [tx_pkt, tx_state] = simple_tx('parameter_sets.csv', 1, ...
   'stream_driver_args', '/tmp/test_janus.wav')
```

In this case the `'WAV'` file created is `/tmp/test_janus.wav`.

```
> [rx_pkt, rx_state] = simple_rx('parameter_sets.csv', 1, ...
   'stream_driver_args', '/tmp/test_janus.wav'), ...
   rx_payload_ascii = char(rx_pkt.payload)
```

To decode the previously created file `/tmp/test_janus.wav`.

Receiver commands

```
> [rx_pkt, rx_state] = simple_rx('parameter_sets.csv', 1, ...
   'stream_driver_args', '/tmp/test_janus.wav', 'verbose', 2), ...
   rx_payload_ascii = char(rx_pkt.payload)
```

To decode the previously created file `/tmp/test_janus.wav` with increased verbosity, showing the single chip error probabilities.

B JANUS README

> ```
> [rx_pkt, rx_state] = simple_rx('parameter_sets.csv', 1, ...
>   'stream_driver_args', '/tmp/test_janus.wav', 'verbose', 1, ...
>   'doppler_correction', 0), ...
>   rx_payload_ascii = char(rx_pkt.payload)
> ```
> To decode the previously created file `/tmp/test_janus.wav` without Doppler correction.

> ```
> [rx_pkt, rx_state] = simple_rx('parameter_sets.csv', 1, ...
>   'stream_driver_args', '/tmp/test_janus.wav', 'verbose', 1, ...
>   'doppler_max_speed', 3), ...
>   rx_payload_ascii = char(rx_pkt.payload)
> ```
> To decode the previously created file `/tmp/test_janus.wav` with Doppler correction estimated in the range $[-3 \sim 3]m/s$.

Matlab variables and files

> ```
> [tx_pkt, tx_state] = simple_tx('parameter_sets.csv', 1, ...
>   'stream_driver', 'mat', 'stream_driver_args', '/tmp/test_janus.mat')
> ```
> Save the JANUS waveform into the Matlab file `/tmp/test_janus.mat`

> ```
> [rx_pkt, rx_state] = simple_rx('parameter_sets.csv', 1, ...
>   'stream_driver', 'mat', 'stream_driver_args', '/tmp/test_janus.mat')
>   rx_payload_ascii = char(rx_pkt.payload)
> ```
> Decodes the file generate in previous example.

> ```
> global test_janus;
> [tx_pkt, tx_state] = simple_tx('parameter_sets.csv', 1, ...
>   'stream_driver', 'mem', 'stream_driver_args', 'test_janus', ...
>   'stream_fs', 48000)
> ```
> Save the JANUS waveform into the Matlab file `/tmp/test_janus.mat`

> ```
> [rx_pkt, rx_state] = simple_rx('parameter_sets.csv', 1, ...
>   'stream_driver', 'mem', 'stream_driver_args', 'test_janus', ...
>   'stream_fs', 48000), ...
>   rx_payload_ascii = char(rx_pkt.payload)
> ```
> Decodes the file generate in previous example.

Physical layer options

> ```
> [tx_pkt, tx_state] = simple_tx('parameter_sets.csv', 1, ...
>   'stream_driver_args', '/tmp/test_janus.wav', ...
>   'chip_len_exp', 3)
> ```
> The generate waveform use the dyadic chip length option with exponent 3, chips length will be $2^3 = 8$ times longer.

> ```
> [tx_pkt, tx_state] = simple_tx('parameter_sets.csv', 1, ...
>   'stream_driver_args', '/tmp/test_janus.wav', ...
>   'stream_channels', 2, 'stream_channel', 0)
> ```
> Generates a stereo waveform modulating the signal only on first channel.

> ```
> [tx_pkt, tx_state] = simple_tx('parameter_sets.csv', 1, ...
>   'stream_driver_args', '/tmp/test_janus.wav', ...
>   'pad', 0)
> ```
> Generates a waveform without padding at the beginning and the end.

> ```
> [tx_pkt, tx_state] = simple_tx('parameter_sets.csv', 1, ...
>   'stream_driver_args', '/tmp/test_janus.wav', ...
>   'stream_passband', 0)
> ```
> Generates a stereo waveform modulated in baseband.

B JANUS README

JANUS packet options

```
> [tx_pkt, tx_state] = simple_tx('parameter_sets.csv', 1, ...
    'stream_driver_args', '/tmp/test_janus.wav', ...
    'packet_mobility', 1, 'packet_tx_rx', 1)
```

Sets the into the packet mobility and ability to receive JANUS signal flags.

```
> [tx_pkt, tx_state] = simple_tx('parameter_sets.csv', 1, ...
    'stream_driver_args', '/tmp/test_janus.wav', ...
    'packet_class_id', 16, 'packet_app_type', 0, ...
    'packet_app_data_fields', ...
    'Station Identifier=1,Parameter Set Identifier=17')
```

Force generation of JANUS packet of *class id* 16 and *application type* 0 (those are already default values). Filling the application fields with *"Station Identifier"=1* and *"Parameter Set Identifier"=17*. It cause loading the plugin library file `libplugin_016_00.m` in directory `plugins`). Plug-ins are not mandatory, are needed to interpret correctly the field name in *packet-app-data-fields* and especially for decoding to know the eventual payload length.

```
> [tx_pkt, tx_state] = simple_tx('parameter_sets.csv', 1, ...
    'stream_driver_args', '/tmp/test_janus.wav', ...
    'packet_class_id', 16, 'packet_app_type', 0, ...
    'packet_app_data_fields', ...
    'Station Identifier=1,Parameter Set Identifier=17', ...
    'packet_payload', '1234')
```

Force generation of JANUS packet as in the previous example with payload text "1234"

```
> [tx_pkt, tx_state] = simple_tx('parameter_sets.csv', 1, ...
    'stream_driver_args', '/tmp/test_janus.wav', ...
    'packet_class_id', 16, 'packet_app_type', 0, ...
    'packet_app_data', '000040443', 'packet_payload', '1234')
```

As in the above example, where application field are directly passed to command line.

```
> [tx_pkt, tx_state] = simple_tx('parameter_sets.csv', 1, ...
    'stream_driver_args', '/tmp/test_janus.wav', ...
    'packet_reserv_time', 1)
```

JANUS packet request reservation time of 1 second.

```
> [tx_pkt, tx_state] = simple_tx('parameter_sets.csv', 1, ...
    'stream_driver_args', '/tmp/test_janus.wav', ...
    'packet_repeat_int', 10)
```

Announcing a retransmission of a JANUS packet in 10 seconds, especially useful for beacon applications.

Mex Matlab version

Mex files are compiled togeither with the "C" file if found a Matlab installation.

Options are the same of Matlab code, with the exception of Matlab files and variables.

The files are installed from the `CMAKE_INSTALL_PREFIX` in `share/janus/matalb/mex`. Run Matlab in the directory containing the MEX file or add that folder to Matlab path.

To run it:

```
> [tx_pkt, tx_state] = janus_tx_mex('parameter_sets.csv', 1, ...
    'stream_driver_args', '/tmp/test_janus.wav')
```

In this case the `'WAV'` file created is `/tmp/test_janus.wav`.

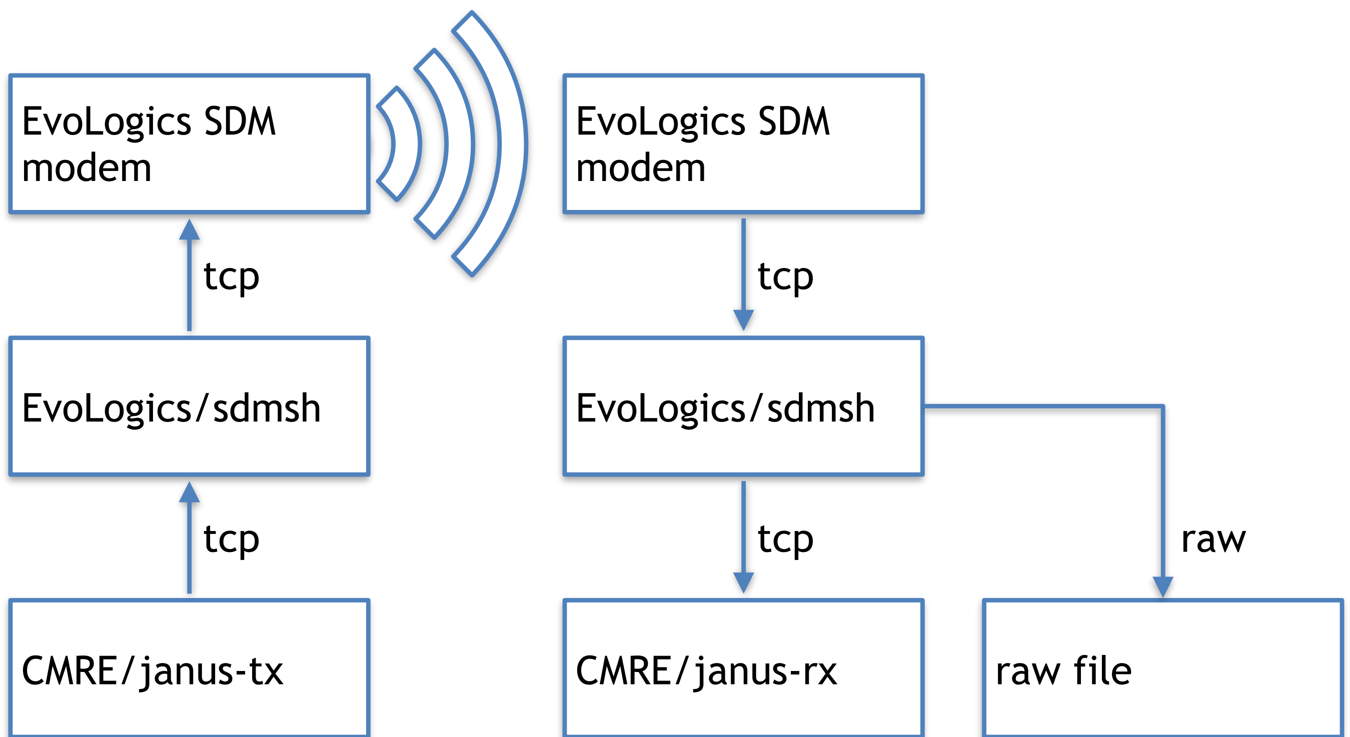
```
> [rx_pkt, rx_state] = janus_rx_mex('parameter_sets.csv', 1, ...
    'stream_driver_args', '/tmp/test_janus.wav'), ...
  rx_payload_ascii = char(rx_pkt.payload)
```

To decode the previously created file `/tmp/test_janus.wav`.

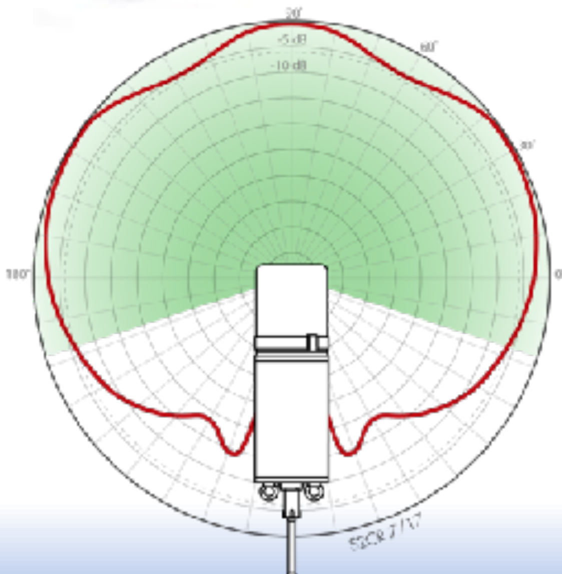
Integration experience of EvoLogics Software Defined Modem Shell with Janus-C implementation

Oleksiy Kebkal, Maksym Komar
EvoLogics GmbH

JANUS with EvoLogics SDM



Evologics S2CR 7/17 Underwater acoustic modem

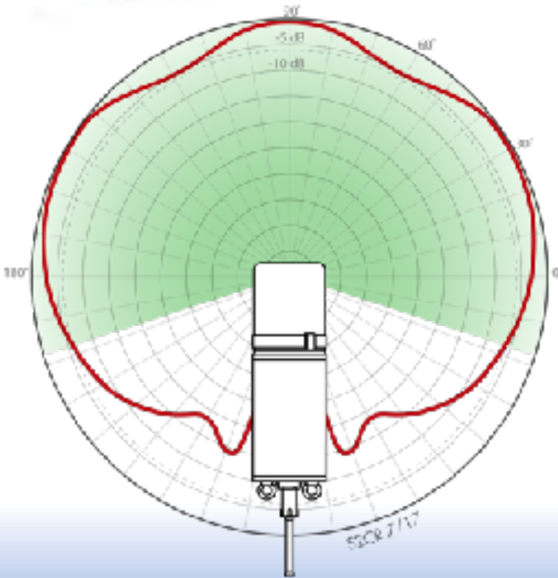


Operating Range	8000m
Frequency Band	7 - 17 kHz
Transducer Beam Pattern	hemispherical
Acoustic Connection	up to 6.9 kbit/s
Physical layer	Chirp QPSK
MAC layer	D-MAC

Applications

- Long-range operations
- Data link for deep-sea AUVs and ROVs
- Underwater observatories
- Underwater acoustic sensor networks

Evologics Software Defined Modem (SDM)



- Fully functional acoustic modem, compatible with other EvoLogics modems
- Can be switched into SDM mode:
  - configure PHY parameters
  - configure a reference to trigger signal detection
  - transmit and receive arbitrary waveforms
  - SDM USBL: receive waveforms on USBL receivers

Evologics/sdmsh

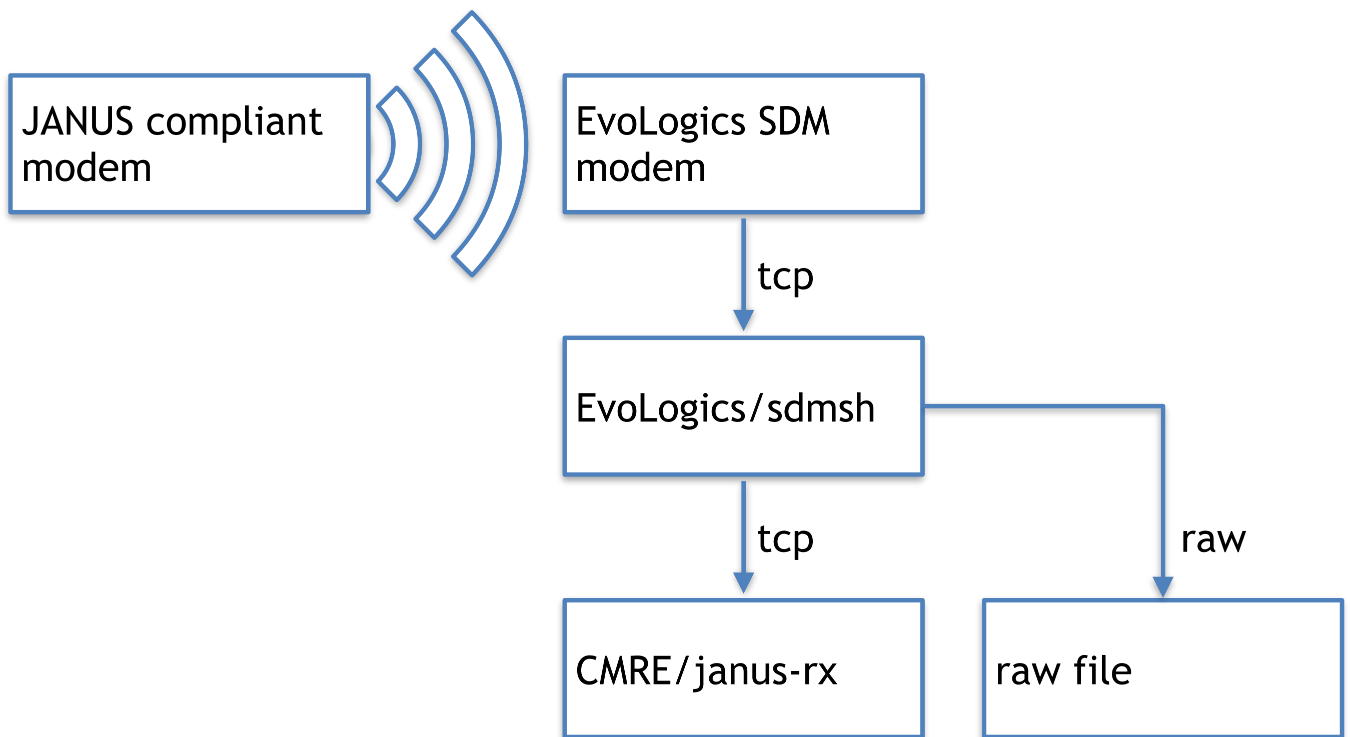
- software defined modem shell
- implements EvoLogics SDM protocol
- scripting support
- IO streams: file, tcp socket
- open source
- https://github.com/EvoLogics/sdmsh

CMRE janus-c

- JANUS signals: generation, detection and demodulation
- IO streams: file in different formats, alsa, pulse audio
- well designed and flexible implementation
- available as tarball on januswiki

- EvoLogics patchset
  - IO streams: +tcp sockets
  - standard compliance fixes

JANUS with EvoLogics SDM



JANUS Interoperability Fest

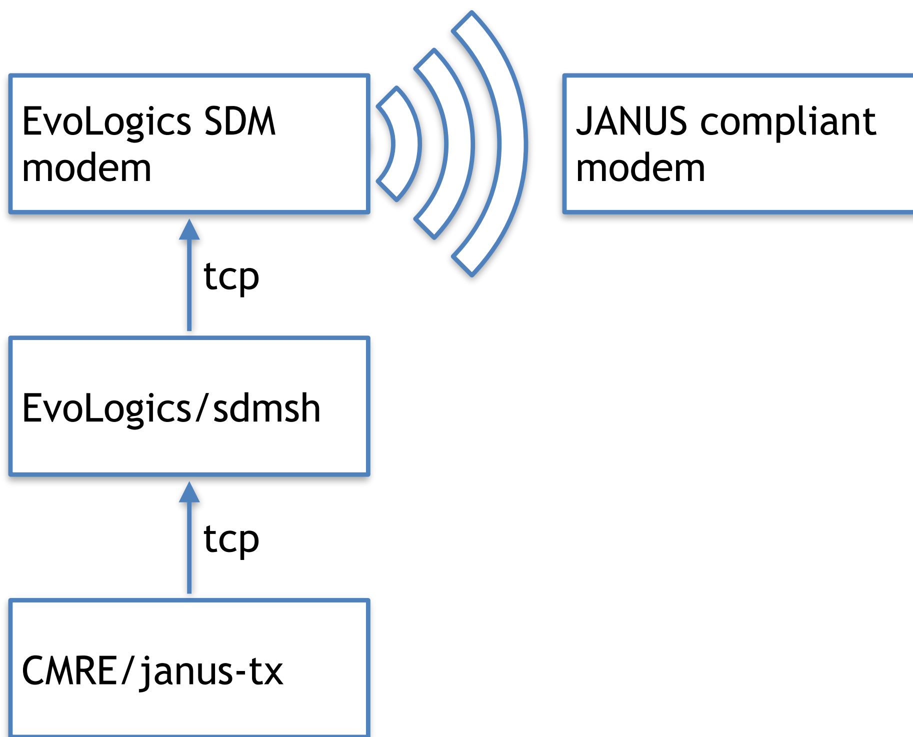
15-May-2018

```
[ OK] teledyne
[ OK] cmre-doppler-2ms
[ OK] cmre-doppler-4ms
[ERR] thales-cargo-136bytes
[ OK] cmre-wakeup-tone
[ OK] cmre-wakeup-tone-minus-24db
[ OK] cmre-wakeup-tone-minus-12db
[ OK] cmre-cargo-8+8
[ OK] cmre-cargo-8+16
[ OK] cmre-cargo-8+32
[ OK] cmre-cargo-doppler
[ERR] cmre-cargo-doppler
```

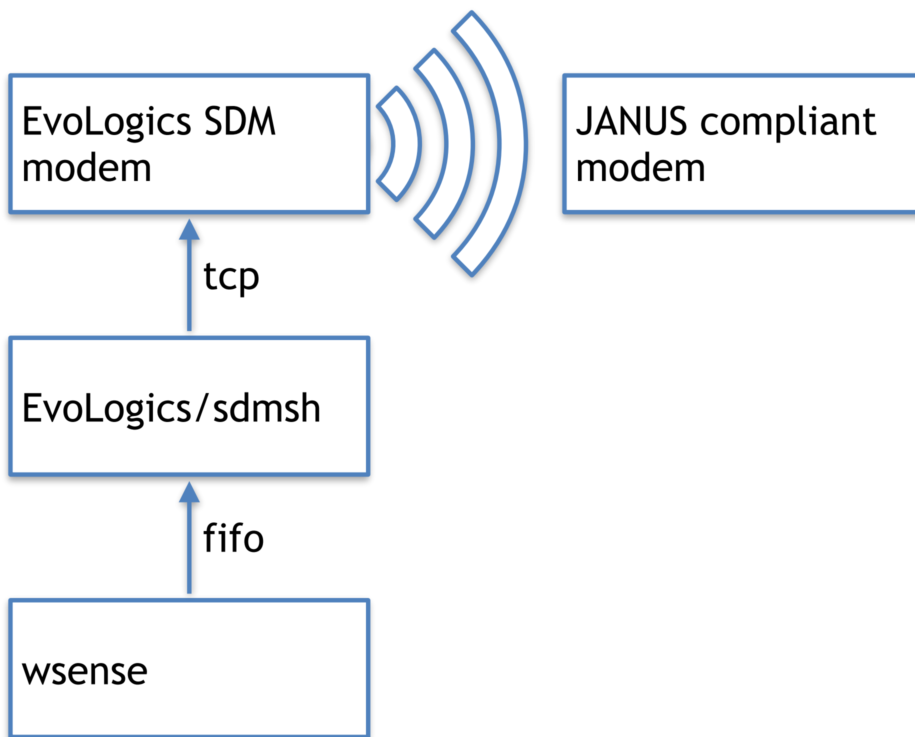
16-May-2018

```
[ OK] cmre-cargo-8-to-32-doppler-4ms
[ OK] cmre-cargo-8-to-32-doppler-2ms
[ERR] cmre-cargo-8-to-32-doppler-2ms
[ OK] wartsila-cargo-8+8
[ERR] wartsila-cargo-8+8
[ OK] wartsila-cargo-8+16
[ OK] wartsila-app-type-0-cargo-8+16
[ OK] wartsila-cargo-8+32
[ OK] teledyne
[ OK] telesub-cargo-8+8
[ OK] telesub
[ OK] telesub-cargo-8+32
[ OK] evologics-rx-cargo
[ OK] evologics-rx-cargo-static-payload
[ OK] teledyne
```

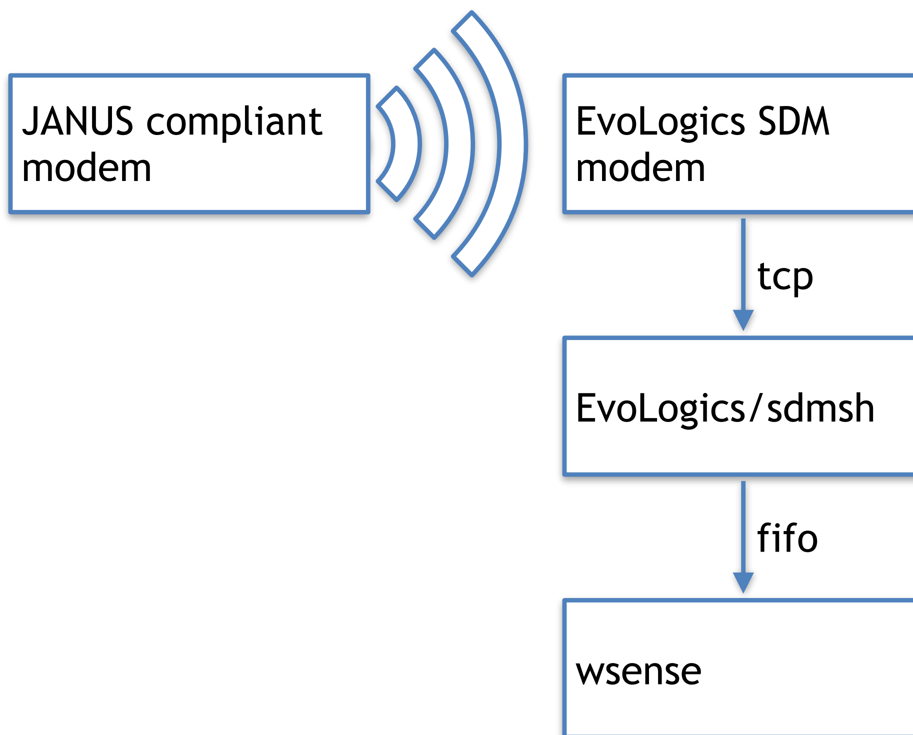
JANUS with EvoLogics SDM



JANUS with EvoLogics SDM



JANUS with EvoLogics SDM



CMRE janus-c drawbacks

- available only outdated version 3.0.1 from 2013
- official repository is not available

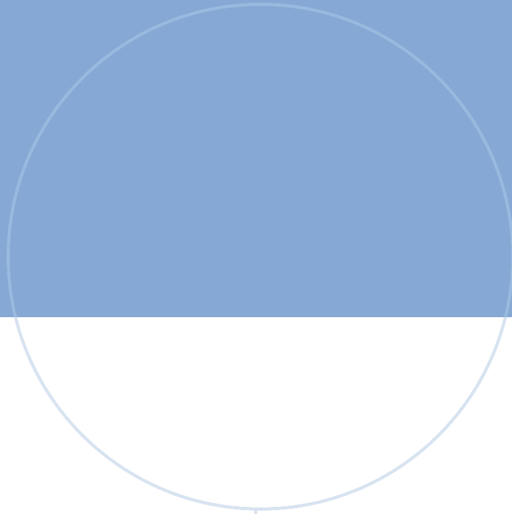
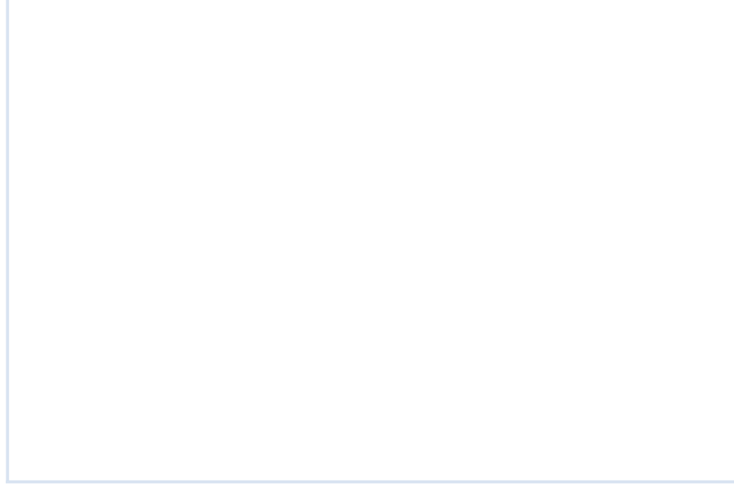
- official repository and maintainer are needed
- unofficial repository maintained by volunteer maintainer could be helpful, if accepted by community

Conclusions and future plans

- EvoLogics has JANUS compliant reference PHY level implementation
- SDM signal triggering with reference signal is not practical for JANUS
- EvoLogics plans to manufacture bilingual modem with JANUS support
  - JANUS signal triggering [Q4 2018]
  - Busy channel detection
  - Modulation / demodulation [Q4 2018]
  - MAC implementation

Thank you

D PROJECT POSTER

NTNU
Norwegian University of
Science and Technology

Underwater Communication Protocol

Finn Christian Eriksen, Markus Næss Ervik, Pål Ivar Delphin Kværnø Fosmo, Aleksander Misje Furnes
NTNU Trondheim, May 2022

History

The fact that sound can be heard under water as well as in air was noted by Aristotle nearly 2000 years ago. In the 1400's Leonardo Da Vinci observed that ships could be heard through the ocean over great distances through a long tube submerged in water.

Wartime developments in the 1900s led to large scale research within the field, This culminated in the development of the underwater telephone by the US in 1945, regarded as the first underwater communication system.

Task

Develop a standardised solution for underwater wireless networking using Evologics S2C Underwater Acoustic Modems as transceiver nodes, exchanging all types of data between multiple ROS2 based systems.

Acoustic Modems

An acoustic modem is a wireless transceiver that operates in the audible frequency range. It features a lower bitrate than traditional wifi but in turn suffers less signal loss over longer ranges making it ideal for subsea applications.

JANUS

The JANUS communication protocol is an open-source signalling method developed as an acoustic communication standard allowing modems from differing manufacturers to communicate under water. This protocol utilizes frequency key shifting in order to transmit data through sound waves.

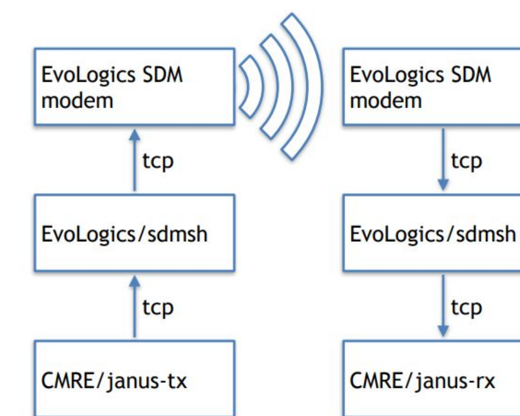
Link Layer

The link layer utilizes multiple tcp servers to process and distribute the system data.
Application data is encoded using JANUS before being passed to a local SDMSH server which in turn directs the data to the modem.

Incoming data from the acoustic link is received to a SDMSH session and redirected to the JANUS decoder.

All data flow is managed in a C++ library easing the implementation of complex logic.

Link Layer Architecture

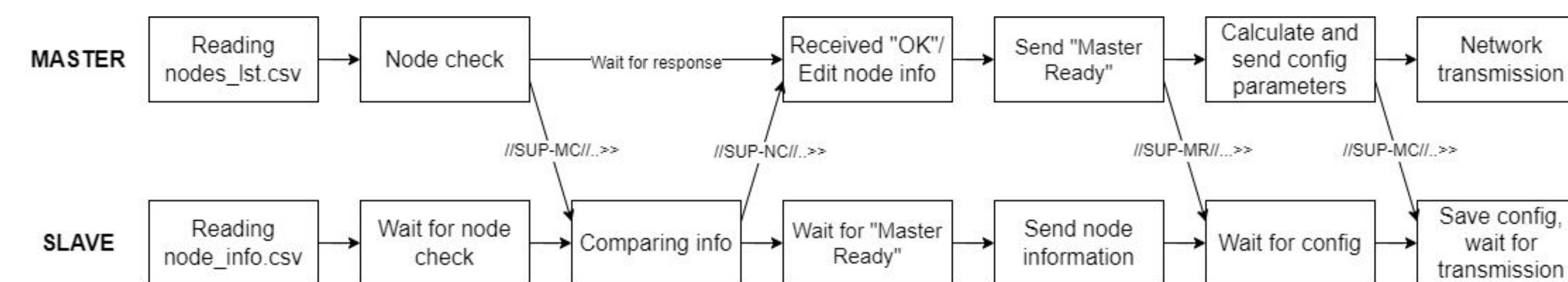


Protocol

The developed protocol is based on a master-slave structure where the master signals to the nodes in turns when they can transmit data.

The master keeps a record of all previously active nodes saved locally, and upon initialization every node is checked for availability.
The initialization process also forms the basis for the network transmission, as the master calculates relevant configurational parameters.

Network Layer Illustration



Sources

[10] EvoLogics GmbH. 'SDMSH wiki'. (9.01.2020), [Online]. Available: https://github.com/EvoLogics/sdmsh/wiki. (accessed: 18.05.2022).
[16] J. Potter, J. Alves, D. Green, G. Zappa, I. Nissen and K. McCoy, 'The janus underwater communications standard', in 2014 Underwater Communications and Networking (UComms), 2014, pp. 1–4. doi: 10.1109/UComms.2014.7017134.
See paper for full reference list.