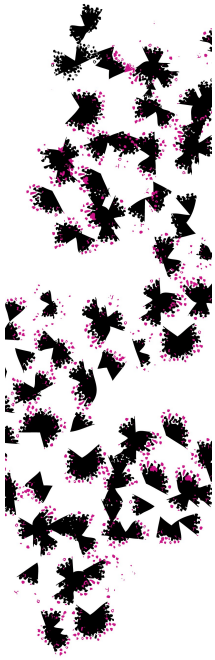


Design of a real-time network channel in LUNA

R. (Robin) Wijnholt

MSc Report

Committee:
Dr.ir. J.F. Broenink
K.J. Russcher, MSc
Dr.ir. P.T. de Boer

October 2017

047RAM2017
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

LUNA is a real-time framework that uses CSP to pass messages from process to process. When communication between two LUNA applications on different hosts is required, no communication method is available that is able to deliver real-time guarantees whilst keeping the CSP methodology in mind.

In this thesis a new network component in LUNA (referred to as *Network Channel*) has been designed and realized that is capable of connecting two real-time LUNA applications. OpenDDS has been used as the communication protocol, which is an open source implementation of Data Distribution System (DDS). Rendezvous communication has been implemented, because this way of message passing is also used in CSP (execution engine of LUNA).

OpenDDS provides QoS settings and can perform real-time communication. An interface for reading and writing to OpenDDS has been realized and has been implemented by the *Network Channel* to implement functionality. The OpenDDS interface is constructed to support bundling of data with the same endpoint. Bundling of data results in better performance as writing to OpenDDS is an expensive operation on the RaMstix (1.34ms per write). A more resource rich platform is faster, and also scheduling the threads with a not real time scheduler on a RAMstix results in faster writes to OpenDDS. Probably mode switches occur on the RaMstix with the Xenomai scheduler and causes higher writer times.

The *Network Channel* implements publish-subscribe, and rendezvous communication. The *DDSReactor* is responsible for sending and receiving messages. A disadvantage is that the loop time of one iteration introduces an extra latency. For the publish-subscribe communication a maximum transmission frequency of 250Hz for 10 channels is obtained where a theoretical maximum sending speed of 400Hz was expected. This is not achieved, as not all data is available in one data bundle, resulting in more than one iteration before all readers are unblocked.

For rendezvous communication a trade-off has been made to only notify the reader's state to minimize the round-trip time of one data exchange. It has been observed that the data sequences do not necessarily contain the data for all readers, and therefore a waiting functionality has been implemented. Waiting for the data of all writers results in the unblocking of the readers at the subscriber side in one iteration, instead of two or maximum three iterations. However at sending rates higher than 200Hz the waiting functionality fails, resulting in incomplete packets and therefore more than one iteration before unblocking. The maximum rendezvous communication with functioning wait functionality has been obtained at 230Hz.

TERRA generates executable code with the LUNA framework from CSP models. A hardware port is added to the TERRA application such that the *Network Channel* can be initialized by the user using the GUI provided by TERRA.

A demo has been realized that controls the youBot base from a RaMstix using an EtherCAT interface and controller in LUNA. An Ubuntu machine is used to execute a LUNA application to send values from a joystick to the RaMstix using the *Network Channel*. It has been observed that periodicity is caused by the SOEM EtherCAT master on the RaMstix, and not necessarily due to Xenomai.

It is recommended to update the waiting functionality to check explicitly for the availability of the subtopic names in a bundle, resulting in bundles that are always complete and therefore faster data exchange for publish-subscribe and rendezvous communication can be achieved. Also by presenting the callback function to the OpenDDS interface and writing as soon as the data is available the loops can possibly be omitted resulting in no extra latency caused by the loops.

Contents

List of acronyms

ADC Analog to Digital Converter

API Application Programming Interface

CAN Controller Area Network

CSP Communicating Sequential Processes

DAC Digital to Analog Converter

DCPS Data-Centric Publish-Subscribe

DDS Data Distribution System

DOF Degrees of Freedom

ENI EtherCAT Network Information

ESI EtherCAT Slave Information

FIFO First In First Out

FPGA Field Programmable Gate Array

GPMC General Purpose Memory Controller

GUI Graphical User Interface

HRT Hard Real Time

I/O Input/Output

IPC Intrinsic Passive Control

LUNA LUNA Universal Networking Architecture

OS Operating System

MDD Model Driven Design

PWM Pulse Width Modulation

RaM Robotics and Mechatronics

RPI Raspberry Pi

ROS Robotic Operating System

RTPS Real-Time Publish-Subscribe

SBC Single Board Computer

SOEM Simple Open EtherCAT Master

TCP Transmission Control Protocol

TERRA Twente Embedded Real-time Robotic Application

UDP User Datagram Protocol

USB Universal Serial Bus

QoS Quality of Service

1 Introduction

1.1 Context

A Single Board Computer (SBC) is used more and more for embedded applications due to their flexibility and ability to perform complex computations. The processing power of these SBCs continues to grow, which makes them popular for robotic setups that need (complex) control. At the Robotics and Mechatronics (RaM) group a SBC is developed called the RaMstix (RaM, 2017a), which is an expansion board for the Gumstix Overo module (Gumstix, 2017). The RaMstix can run Hard Real Time (HRT) control loops using LUNA Universal Networking Architecture (LUNA) in a Xenomai environment (Bezemer, 2011).

LUNA is a hard real-time, multi-threaded, multi-platform, Communicating Sequential Processes (CSP) capable, and a component-based framework. However a real-time communication channel following the heuristics of CSP is not present.

A channel with Robotic Operating System (ROS) for LUNA is already designed by van der Werff (2016), but this solution mainly focuses on connecting a LUNA application with HRT loop controllers to a more resource rich platform that is not necessarily HRT. The disadvantage of using ROS as a communication channel is that it provides no real-time guarantees and therefore the complete system using ROS channel is definitely not real-time.

The work of van de Ridder (2017) on the production cell is an example of a project where real-time communication between those SBCs is important for correct functioning of the total setup. This setup consists of multiple stations each performing their own task, for example a feeder belt, extraction belt, rotation robot and an extraction robot. Every station contains its own LUNA application that has to pass the status of the station to other stations in this setup. Without a reliable communication channel, correct functioning of the setup is not guaranteed.

The work of van de Ridder (2017) is just one of many more projects that could benefit from a proper network component in LUNA. LUNA is a great framework for controlling setups and together with a good functioning communication channel it is easily scaled to bigger projects.

1.2 Goals of the project

This project consists of three parts: the *Network Channel*, the EtherCAT interface and the controller for the youBot. Figure 1.1 shows the relation of these parts.

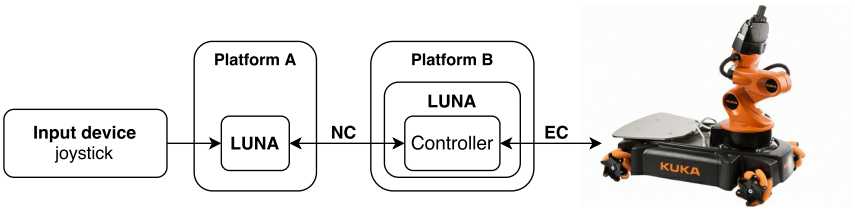


Figure 1.1: The relation between the *Network Channel*, the EtherCAT interface and the controller residing on two platforms. *NC* signifies the *Network Channel* and *EC* signifies EtherCAT.

The goals of this project are formulated as follows:

1. *Design a Network Channel for LUNA*: There is currently no real-time communication method between two LUNA applications that are on different hosts. Therefore a Network Channel is designed that implements this functionality in LUNA.

2. *Provide EtherCAT coupling to LUNA to control the youBot*: EtherCAT is widely used in the industry for real-time applications. The youBot uses EtherCAT to talk to the motor drivers of the youBot joints.

3. *Showcase the Network Channel*: The correct functioning of the *Network Channel*, controller, and EtherCAT interface for the youBot must be shown with a demo.

1.3 Outline of the report

In Chapter 2, background material on LUNA and TERRA, OpenDDS, the RaMstix, and the youBot is given. The requirements of this project are given in Chapter 3 stating the requirements using a MoSCoW format. Chapter 4 contains the design choices, realization and tests of the *Network Channel* divided in the sections: OpenDDS, TERRA, and the LUNA implementation. The design choices, realization and tests of the EtherCAT interface is discussed in Chapter 5. The demo that integrates the *Network Channel* together with EtherCAT interface and the controller to control the youBot is demonstrated in Chapter 6. This report is concluded with a conclusion and recommendations in Chapter 7.

2 Background

The hardware and software used in this project are described in this chapter. Section 2.1 contains information about LUNA and TERRA with a subsection that describes CSP. The fundamentals of OpenDDS are in Section 2.2 and information about the RaMstix is provided in Section 2.3. This chapter concludes with Section 2.4 with information about the youBot.

2.1 LUNA and TERRA

CSP describes how processes interact with each other. LUNA implements CSP as its execution engine. First the basics of CSP are discussed in Section 2.1.1. The fundamentals of LUNA is discussed in Section 2.1.2 and lastly TERRA is treated in Section 2.1.3, which implements a Graphical User Interface (GUI) for drawing CSP processes and can create execution code for the LUNA framework.

2.1.1 CSP

CSP is introduced by Hoare (1978) and is a method to describe a concurrent synchronous communication application. Channels connect CSP processes and can exchange messages by using events. Communication events with variables in CSP are denoted as:

- *c?v*: read variable *v* from channel *c*.
- *c!v*: write variable *v* onto channel *c*.
- *c.v*: Variable *v* on channel *c*, no direction provided.

Events are instantaneous and do not have to be accompanied by a variable. Processes are accompanied by a set of events (called an alphabet) where a process may engage in. The order of operation is not determined by the channels, but by process operators. The basis process operators for CSP are:

- *PAR*: All processes activated at the same time, execution order is fair (no fixed order of execution). Continues to a next process when all processes in a parallel construct are finished.
- *SEQ*: The processes are activated and executed one by one.
- *ALT*: Certain processes are activated depending on the variable on the provided channel.

A graphical example of a CSP model is shown in Figure 2.1. The *SequenceController*, *LoopController* and *Filter* are processes connected with each other by channels that are indicated with the arrows. The direction of the arrow indicate the direction of the variable. The arrows however do not state the order of execution as that is indicated by the connections with a '||' sign that indicates a parallel construct. The question and exclamation mark indicates a reader and writer respectively and the asterisk in the top left corner states that everything that is boxed in that rectangle is repeated indefinitely.

Messages between processes are exchanged only when both processes are active. This results in a synchrony between communicating processes. This way of messaging is referred to in CSP as rendezvous communication.

Complex designs can be constructed with only a few primitive operators and events. More information on CSP can be found in the work of Hoare (1978) and/or Chapter 7 "An overview of CSP" of Nissanke (1997).

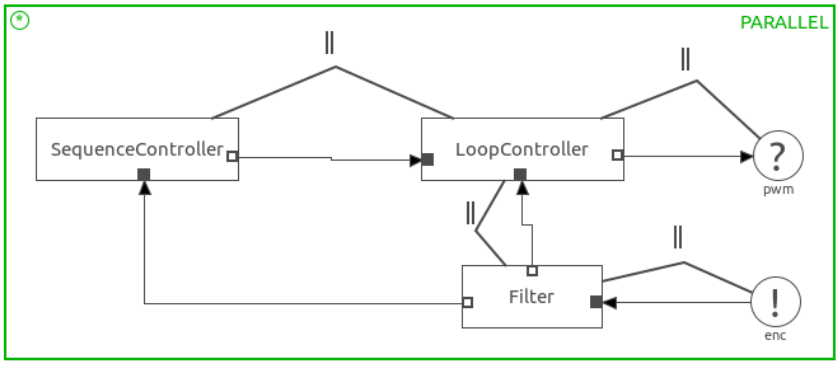


Figure 2.1: A CSP example drawn with TERRA. The processes are connected using parallel constructs that indicate the order of execution, and the arrows indicate the channels and the direction of the variable on that channel.

2.1.2 LUNA

LUNA is a framework that implements CSP as the execution engine. LUNA (Wilterdink, 2011) is the answer to shortcomings of other implementations and is designed with the following requirements in mind:

- Hard real-time
- Multi platform
- Thread support
- Scalability
- CSP execution engine
- Component based

Figure 2.2 shows the architecture of LUNA. CSP is used as the execution engine, but LUNA is made such that other execution engines can be implemented as well. The core components are on layer 1 indicating that layer 2 and 3 are independent of the used Operating System (OS).

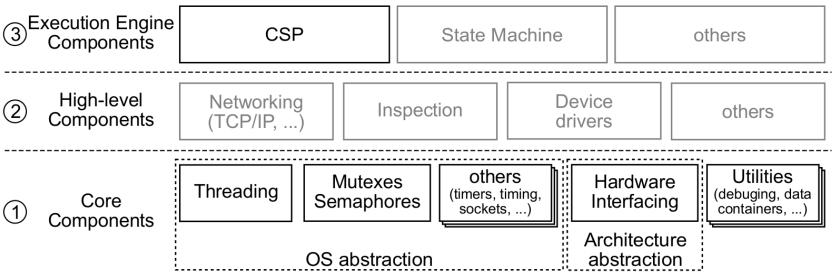


Figure 2.2: Overview of the LUNA architecture (Bezemer, 2011). The light grey blocks are yet to be integrated.

LUNA is designed with threads to support concurrent processes. These threads are used as long as the OS on which LUNA is running can provide them. This way of working is possible ,because the execution engine is separated from the core components.

User threads and OS threads are the two types of threads that can be used. OS threads are resource-heavy, but can run on different cores and have preemptive capabilities. User threads

are light on resources, but must run in a OS thread and are thus running on the same core as the OS thread.

LUNA is responsible to map the processes and constructs onto threads. An example of a LUNA application that is mapped onto threads is shown in Figure 2.3. This figure shows that processes are mapped onto user threads and are put in a user thread container (UTC) before mapping them on a OS thread. The CSP constructs are decentralized implemented, which results in a generic scheduling mechanism without any knowledge of the other CSP constructs.

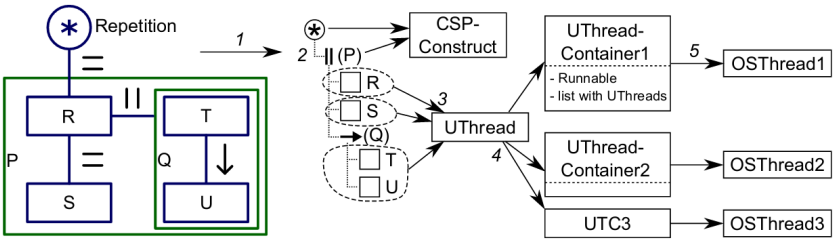


Figure 2.3: The implementation of thread grouping performed in LUNA (Bezemer, 2011). User threads are placed in a user thread container before it is placed on an OS thread.

All processes are able to run parallel in their own thread groups. The efficiency of the application is dependent on how the thread groups are created as only the thread groups are able to run their threads in parallel.

Previous implementations by Kempenaar (2014), Bezemer and Broenink (2015), and van der Werff (2016) were able to connect LUNA to another application using ROS channels. The goal of the network channel presented in this thesis is to connect two HRT LUNA programs whilst trying to maintaining the CSP methodology as much as possible. This has as a result that the Network Channel should also support rendezvous communication as well. An example of rendezvous messaging for a writer and a reader is shown in Figure 2.4.

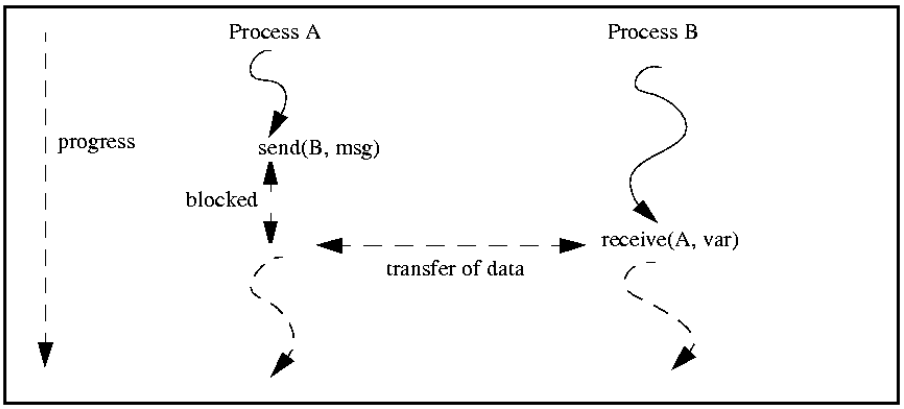


Figure 2.4: Rendezvous messaging shown using two processes A and B (Smyth and Davis II, 1999).

In Figure 2.4 the blocking of process A (sender) is shown, but the same holds for process B (receiver) if that process happens to be activated earlier in time than process A. On the same host this behavior is easily obtained by shared memory or by sharing relevant classes between processes, but for processes residing on different hosts this asks for a different approach, which is discussed in detail in Chapter 4.

Hardware ports in LUNA are implemented by Bezemer (2014) to interface with hardware residing outside of LUNA. The hardware ports consists of a write and read action similar to CSP readers and writers to easily interface with the rest of the LUNA application.

2.1.3 TERRA

TERRA is a Model Driven Design (MDD) tool that enables the user to draw CSP processes and to generate execution code using the LUNA framework. TERRA is a modular collection of tools for designing (control) software for cyber-physical systems, currently providing (RaM, 2017b):

- *Model editors*: Editor for CSP and an architecture model editor.
- *Model Validation*
- *Code validation*: machine CSP and LUNA (C++).
- *Simulation and animation*: log model to visualize system states with an animation.
- *External tool support*: 20-sim and SCXMLgui

For this project TERRA is used to implement a hardware port that is able to add the Network Channel in TERRA. Hardware ports were first added by Bezemer (2014) to get rid of the sand-boxed LUNA applications. The Mesa Anything I/O FPGA board was used in that project as a proof of concept. The design of the hardware port is shown in Figure 2.5. On the left side an architecture meta-model is shown that is used to construct the functionality of the hardware port. The graphical editor part and the code generation are on the right side and deliver the graphical part and code generation.

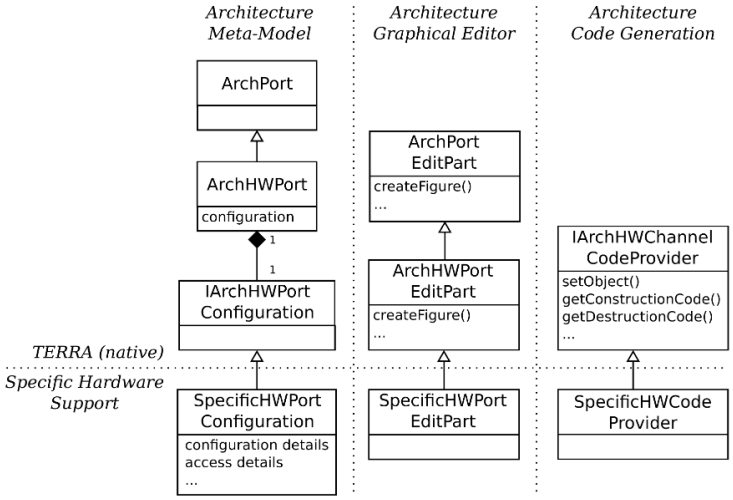


Figure 2.5: The design used for adding hardware ports to TERRA (Bezemer, 2014).

Regular CSP channels provide means to communicate data from a writer to a reader over a channel, hardware channels replace the channel with a hardware implementation as is shown in Figure 2.6.

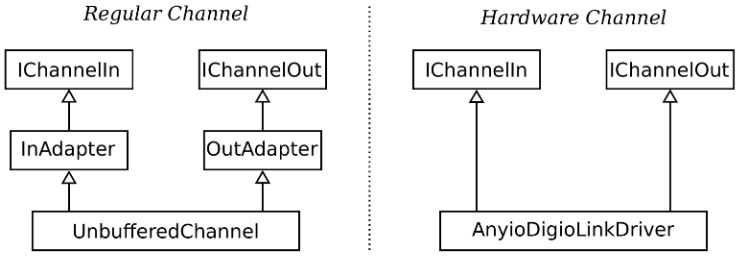


Figure 2.6: A regular CSP channel compared to a hardware channel. Both are providing the same interface to the rest of LUNA (Bezemer, 2014).

Figure 2.6 shows the interfacing of a hardware component to the rest of LUNA by using a *IChannelIn* and an *IChannelOut* interface. This way the actual channel implementation is hidden from readers and writers, resulting in no difference between hardware ports and writers and readers from LUNA perspective. The result is a TERRA code generation that is able to generate the standard readers and writers, even when hardware communication is required.

2.2 OpenDDS

In the search for a suitable communication protocol the following requirements must be met:

- Scalable
- Discovery of endpoints
- Rich set of Quality of Service (QoS) settings
- Real-time capabilities

A lot of communication protocols are available and most of the protocols do not provide all the requirements. ZeroMQ (ZMQ, 2014) provides real-time capabilities, but has no auto discovery for endpoints. Boost.Asio (Kohlhoff, 2017) has the ability to find endpoints using broadcasting, but is not able to provide QoS settings for the user. ZeroMQ and Boost.Asio are protocols that wrap the sockets provided by the OS and therefore do not provide a lot of QoS settings. Also these protocols do not have their own serialization and deserialization and need therefore another library to provide that.

ZeroMQ and Boost.Asio are not suitable for this project but do deliver fast writing speeds as is shown in Busch (2010). This test compares OpenDDS to ZeroMQ and Boost.Asio. All three have a great performance, but OpenDDS does provide the QoS settings that this project needs. OpenDDS also provides endpoint discovery and real-time capabilities as it is based on the OMG Data Distribution Service (DDS) for Real-Time Systems standard. Serialization and deserialization is performed using the IDL data structure OMGIDL (2017).

OpenDDS is completely open source, is used in ROVE (ROVE, 2017) and the new version of ROS uses ROS2 as its communication protocol. Therefore using OpenDDS for this project will result in a new ROS LUNA interface as all DDS vendors should be inter-operable.

OpenDDS is a publish-subscribe service, which is data oriented. The basic entities that needs to be available for every DDS communication are:

- *Topic*: Contains information about a single data type and the distribution and availability of samples.
- *Publisher*: Apply control and restrictions to flow of data from DataWriters.
- *Subscriber*: Apply control and restrictions to flow of data from DataReaders.
- *Datawriter*: Creates Samples of a single application data type.
- *Datareader*: Receives Samples of a single application data type.

To show the relation between the previously mentioned entities Figure 2.7 is provided. For each subscriber or publisher a participant must be available as well as a topic. The participant resides in a DDS domain. Each publisher is accompanied by a datawriter and each subscriber is accompanied by a datalistener.

Each entity has its own QoS settings which are detailed in Chapter 3 of the OpenDDS Developer's Guide (OpenDDS, 2017b). The QoS settings of the publisher must be more demanding than the QoS settings of the subscriber or the writer and listener will not match.

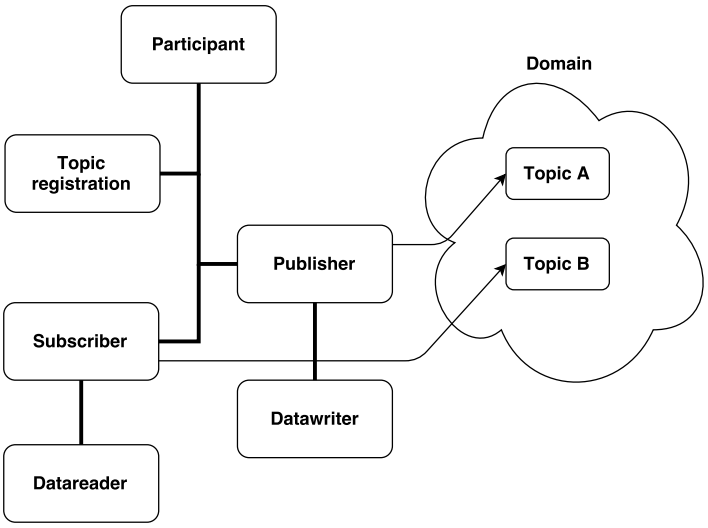


Figure 2.7: Overview of the relation between the entities of DDS, based on Corsaro (2013).

The flow of sending a sample from publisher to subscriber via an OpenDDS topic is as follows:

1. The publisher initiates the flow of data when a data value has been written to the datawriter.
2. The datawriters publication publishes the Samples to the associated subscription(s).
3. Each associated subscriber gives the received sample to its datareader(s) that are associated with the sending datawriter.
4. The flow ends when the application retrieves the data from the datareader.

A conceptual view of the established connection between a DDS publisher and DDS subscriber communicating via a topic is shown in Figure 2.8. The datawriter and datareader are the instances that provide the means to write to, and to read from OpenDDS topics respectively.

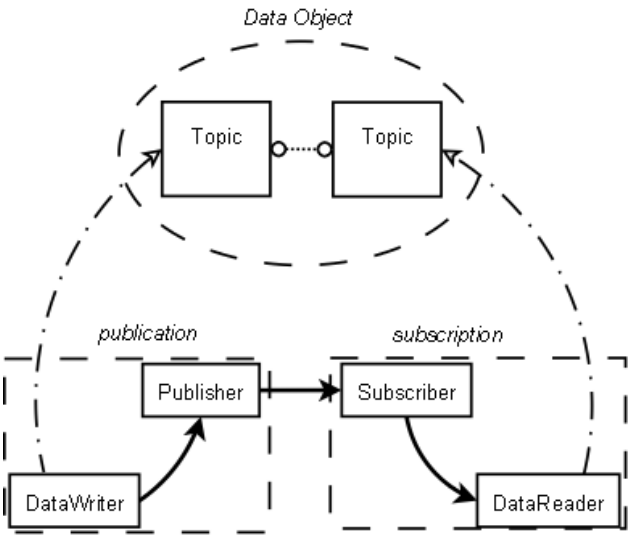


Figure 2.8: A conceptual view of the interconnection between a publisher and subscriber after matching topics (OpenDDS, 2017a).

2.3 RaMstix

The RaMstix (RaM, 2017a) is a platform used to run embedded control software at the RAM group. A functional overview of the RaMstix is shown in Figure 2.9. The RaMstix is an expansion board for the Gumstix Overo module (Gumstix, 2017). The Gumstix Overo module has a single core running at $800Mhz$ and provides WiFi and Bluetooth. Xenomai runs in dual kernel mode alongside the normal kernel on the Gumstix to support HRT applications. The Overo module is connected with a General Purpose Memory Controller (GPMC) to an Field Programmable Gate Array (FPGA). This combination offers the following Input/Output (I/O):

- Universal Serial Bus (USB)/serial debug interface
- 100 Mbit Ethernet
- USB master
- USB slave
- Four dedicated Encoder inputs
- Four dedicated Pulse Width Modulation (PWM)/Stepper outputs
- 16 digital output pins
- 16 digital input pins
- Controller Area Network (CAN) bus interface
- Two 16-bit Analog to Digital Converter (ADC)
- Two 16-bit Digital to Analog Converter (DAC)

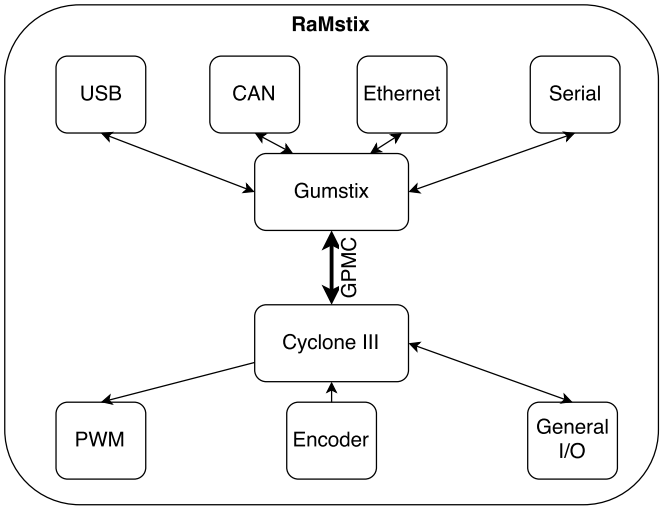


Figure 2.9: A functional overview of the different components available on the RaMstix. The arrow indicate data flow.

The RaMstix is an excellent platform to rapidly test developed software with 20-sim on a setup using the tool 20-sim 4C which allows a controller (or any other model) designed in 20-sim to be uploaded to the RaMstix in a few steps. The inputs, outputs and other variables can be monitored using 20-sim 4C. LUNA is also able to run on a RaMstix, and together with the dual kernel configuration with Xenomai the RaMstix is an interesting platform for this project. For this project LUNA will run on the Overo module and the Ethernet port is used for EtherCAT connection to the youBot.

2.4 YouBot

The youBot is used as a demonstrator to show the functionality of the *Network Channel*. The youBot is a robotic arm on an omni-directional platform designed by KUKA. The arm has five Degrees of Freedom (DOF) and a two-finger gripper. The platform consist of four omni-directional wheels such that it can move in each direction on its xy plane without rotating around the z axis. A picture of the youBot together with the positive joint directions and its range of motion are shown in Figure 2.10.

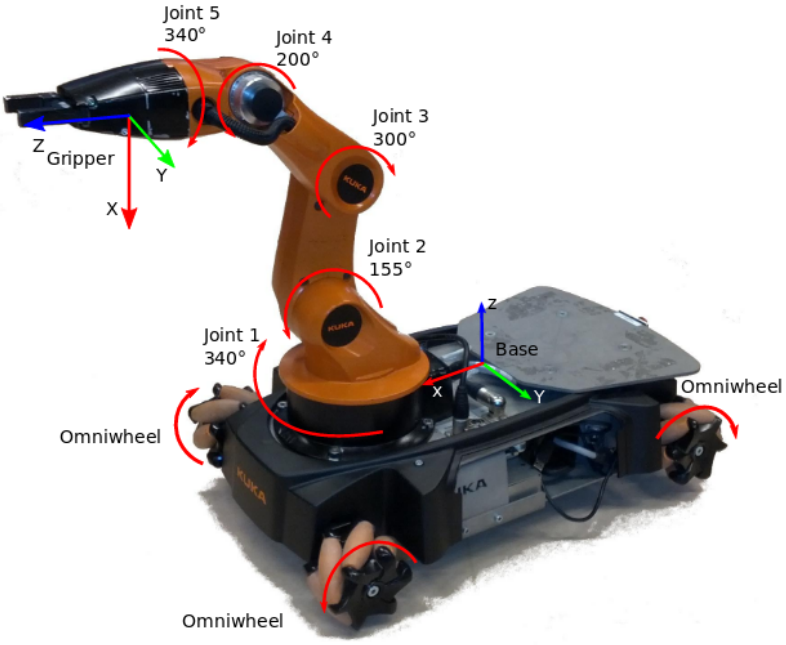


Figure 2.10: The KUKA youBot. Positive direction of the joints are shown with arrows together with the range of motion in degrees for each joint (Frijnts, 2014).

Each joint of the youBot is controlled by a motor driver that is connected to EtherCAT. The motor driver is capable of performing PID control on torque, velocity and position, but also low level control as setting PWM speeds to the joints and reading encoder values is provided. The motor driver on joint 5 has a RS232 connection that is able to control the gripper. An overview of the interconnection of the motor drivers is shown in Figure 2.11

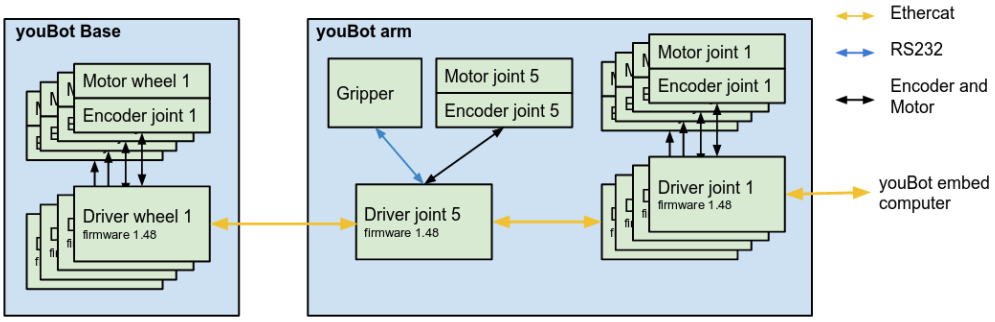


Figure 2.11: All joints have their own motor driver that communicates over EtherCAT with the master (youBot embedded computer). The gripper is controlled via rs232 via motor driver 5 (Frijnts, 2014).

To control the joints over EtherCAT the youBot driver Application Programming Interface (API) provided by KUKA shown in Figure 2.12 can be used.

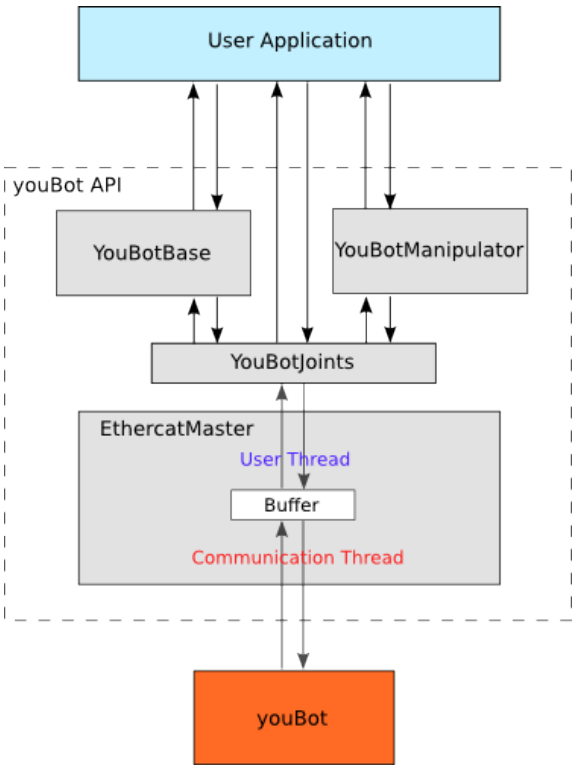


Figure 2.12: Architecture of the youBot API (KUKA, 2015). The API provides joint level control for the youBot.

The youBot uses EtherCAT to control the motor drivers. EtherCAT is an Ethernet based fieldbus system that can achieve real-time communications. The power of EtherCAT lies in the way it handles the data. Each EtherCAT slave extracts relevant data and places his own data into the telegram "on-the-fly". The telegram reaches all slaves before it is returned to the master. EtherCAT almost supports all topologies, including line, tree, star and daisy-chain.

Each EtherCAT slave must be configured at the master and this is done using the EtherCAT Slave Information (ESI) files, which should be provided by the vendor of the EtherCAT slave. All the ESI files are translated to one EtherCAT Network Information (ENI) file with the EtherCAT configuration tool. The ENI file is used by the EtherCAT master to communicate with the connected EtherCAT slaves. The relation of master, slave, ESI, and ENI is shown in Figure 2.13.

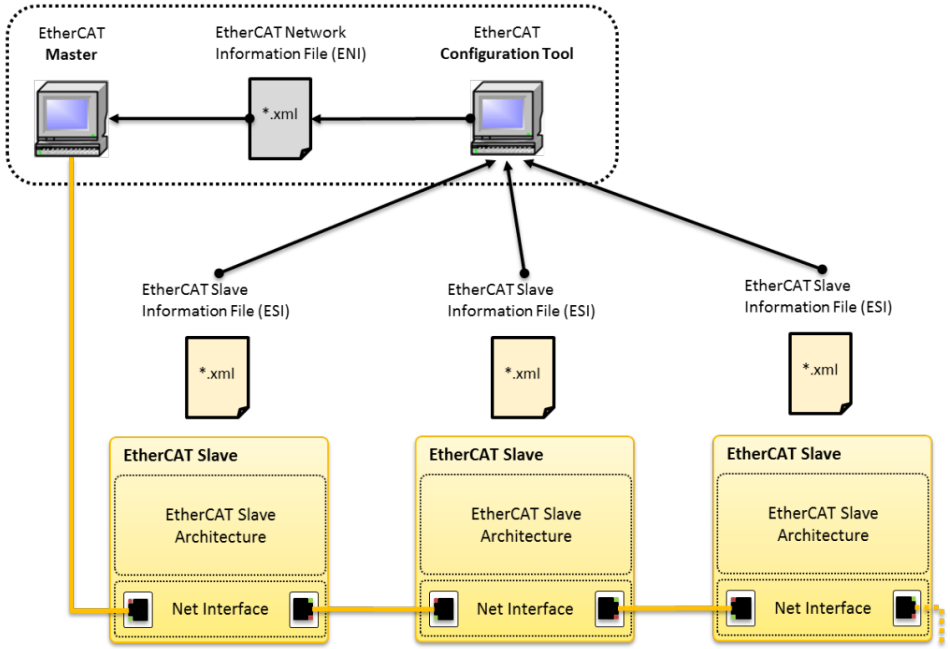


Figure 2.13: The EtherCAT network architecture, providing the realation between master, slave, ESI, and ENI (Beckhoff, 2012).

3 Requirements

The requirements of this project are listed in MoSCoW format per part of this project. The requirements are separated into three parts, namely: *Network Channel*, EtherCAT and the controller that drives the youBot. The relation between those parts is shown in Figure 3.1.

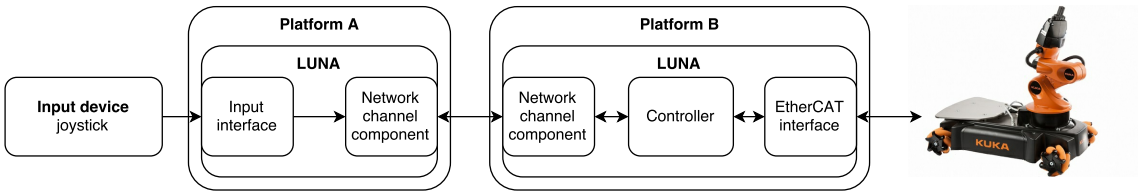


Figure 3.1: This schematic shows the relation between an input device, LUNA with input and *Network Channel* component, and the connection with EtherCAT to the youBot.

The requirements of the *Network Channel* are given in Section 3.1. The requirements of the EtherCAT interface is given in Section 3.2 and lastly the requirements on the youBot controller are specified in Section 3.3.

3.1 Network channel

The *Network Channel* is the main component in this project. The main goal of the *Network Channel* is to connect two LUNA applications for the exchange of data. The requirements belonging to the *Network Channel* are listed below in descending priority.

1. **The Network Channel must be able to perform reliable communication between LUNA applications**: The *Network Channel* must be able to communicate with certain guarantees of the channel performance.

2. **The Network Channel must function on WiFi**: The implementation must work with WiFi as this enables devices to be mobile.

3. **The Network Channel must provide a rich set of QoS settings**: There must be a lot of settings that could change the behavior of the channel, such that the desired channel characteristics can be obtained.

4. **The Network Channel must support rendezvous communication**: LUNA exploits CSP as the execution engine. To follow the CSP method the *Network Channel* should provide rendezvous communication as well.

5. **The Network Channel must be implemented as a component in LUNA**: The *Network Channel* could be implemented by a code block, but it is more convenient to make the *Network Channel* part of LUNA by implementing it as a component. This way it is possible to build the *Network Channel* as an option in LUNA.

6. **The Network Channel should support buffered channels**: Physically separated systems can have different loop frequencies. To be able to communicate between those systems buffered channels should be implemented as well.

7. **The Network Channel should be easy scalable**: To support the use of the *Network Channel* between multiple LUNA applications on several hosts it should be easy scalable. Auto discovery of endpoints has therefore preference.

8. **The Network Channel should be accessible from TERRA**: To use the *Network Channel* in a convenient way it should be available in the TERRA GUI. The high level settings can be handled by the configuration window in TERRA and the user would not have to bother about typing code.

9. **The Network Channel could be inter-operable with ROS2**: The newest version of ROSec: uses DDS as the communication middleware. The new version or ROS promises to be real-time capable. ROS1 is used for many applications at RaM and it may therefore be beneficial to make this *Network Channel* compatible with this newer version of ROS, as it will most probably used for future projects.

10. **A new communication protocol for the Network Channel will not be designed**: A protocol that is designed from scratch will most certainly have issues with maintainability, usability, and portability. An existing protocol will probably have no issues with these as support and updates are mostly provided. Also a community is can be present that is able to help when bugs arise.

3.2 EtherCAT interface

EtherCAT is used to control the youBot motor drivers and therefore some requirements are drafted for the EtherCAT interface in this project.

1. **There must be a coupling between LUNA and an EtherCAT interface to control the motors of the youBot**: The EtherCAT API for the youBot works as is, but it should also be (partly) implemented in LUNA to enable interfacing with the youBot from LUNA.

2. **The EtherCAT interface should have real-time performance**: A goal is to obtain an EtherCAT connection together with LUNA that can perform real-time communication with the youBot. EtherCAT is known for its fast and deterministic cycles and therefore real-time communication should be feasible.

3. **A LUNA component for EtherCAT could be made accessible from TERRA**: If the LUNA component for EtherCAT is designed and it is generic enough it could be implemented in TERRA as well to make it accessible from the GUI to make it easy to use in other applications.

4. **The EtherCAT master from IgH could be used for EtherCAT connection instead of the youBot API**: SOEM is known to have high periodic latency peaks (Spil (2016)). If this is caused by the SOEM implementation another open source implementation can be used like IgH which may solve this problem.

5. **A new EtherCAT master will not be designed.**: No effort is put into designing a new EtherCAT master for communication with the youBot.

3.3 youBot controller

An implementation of a controller is necessary to calculate with inputs the desired outputs Depending on the type of controller the actual implementation may vary.

1. **A youBot controller must be implemented in LUNA**: A controller must be implemented to show a demo of LUNA with the EtherCAT interface controlling the youBot. The kind of controller is less important as the focus of this project is to design a network component in LUNA and not on designing a new kind of controller.

2. **Inverse kinematics could be implemented for the controller**: The controller designed by Spil (2016) is an inverse kinematics controller and could be ported to LUNA.

3. **A new controller could be designed that uses Intrinsic Passive Control (IPC)**: To make the *Network Channel* with a controller more robust for disturbances and connection failures a form of IPC could be implemented.

4 Network Channel in LUNA

4.1 Introduction

The design choices, realization, and tests for every part that constructs the *Network Channel* is provided in this chapter. This chapter contains three sections, and each section contains a design, realization, and tests:

- *OpenDDS*: An interface is designed in Section 4.2 that enables LUNA to write and read from OpenDDS. This interface implements a way to provide rendezvous, and publish-subscribe communication by using OpenDDS as communication protocol and is utilized by the LUNA implementation to actually add functionality to the *Network Channel*.
- *TERRA*: To graphically draw CSP processes TERRA is used. TERRA is able to generate executable code using the LUNA framework. An extra hardware port (DDS port) is added to TERRA in Section 4.3, such that the user can initialize the *Network Channel* from TERRA.
- *LUNA implementation*: The OpenDDS interface is integrated in the actual *Network Channel* in Section 4.4. The functionality of the *Network Channel* is implemented and a component in LUNA is constructed in this section. This section integrates the OpenDDS interface.

An overview of the relations between TERRA, LUNA, and OpenDDS is given in Figure 4.1. TERRA uses the LUNA framework to generate executable code. The generated code is able to instantiate the *Network Channel* that used the OpenDDS interface to provide the communication between LUNA applications.

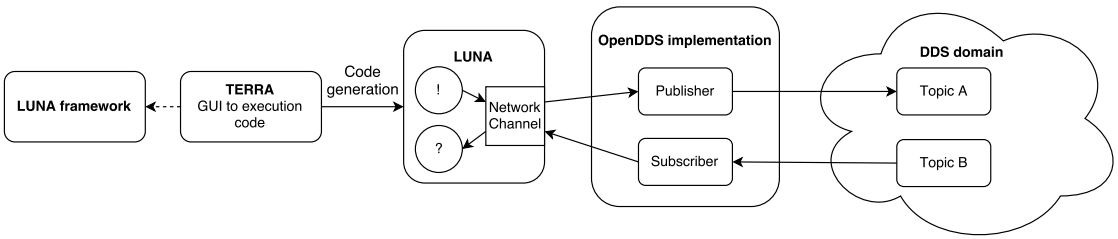


Figure 4.1: Overview of how OpenDDS relates to LUNA and how LUNA relates to TERRA. The solid lines indicate data flow, and the dashed line represents a dependency.

4.2 OpenDDS

4.2.1 Design

The implementation of OpenDDS provides an interface for writing and reading to OpenDDS topics from the *Network Channel* component in LUNA.

The OpenDDS implementation should support publish-subscribe and rendezvous communication. For publish-subscribe communication only one datawriter is instantiated at the publisher and one datalistener at the subscriber. If the topics match, then the data will be exchanged from publisher to subscriber, which is shown in Figure 4.2.

For rendezvous communication in a normal CSP channel the reader and writer know when both are ready to exchange data as they reside on the same host. For the *Network Channel* these states needs to be communicated over the network, which will result in a lot of messages back and forth that all are influenced by network latency and processing time in the application. A

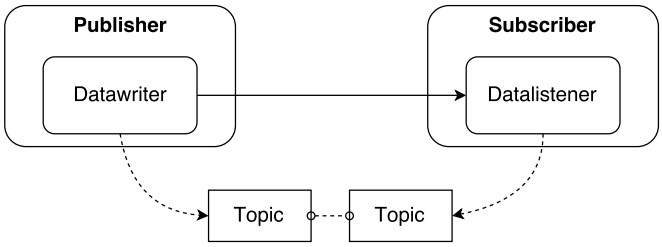


Figure 4.2: An overview of communication via OpenDDS with the publish-subscribe pattern. The data flow is from publisher to subscriber. The topics must match before communication is started.

trade-off is made that only the reader can present its state to the writer, which will result in one less state exchange for every data exchange, which is shown in Figure 4.3.

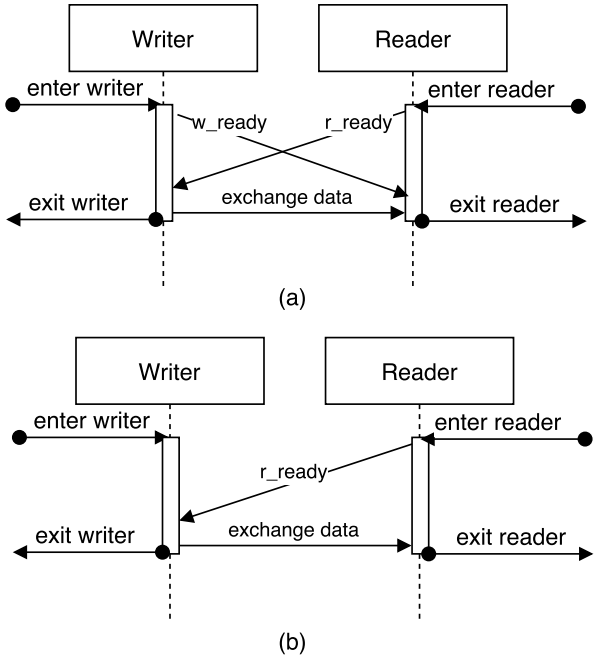


Figure 4.3: The messages that are send over the network when traditional rendezvous messaging is used(a) versus the trade-off that only notifies a ready reader(b).

In the trade-off two flows of data are present, one for state exchange and one for data exchange. The reading side cannot publish data to the writer, and therefore it is necessary to instantiate another writer at the reading side, and an extra listener at the writing side a statewriter and statelistener respectively and is shown in Figure 4.4. It is chosen to instantiate another listener and writer for rendezvous communication, because it separates the publish-subscribe communication from the rendezvous communication.

The blocking of writers and readers is not implemented by OpenDDS, but by the *Network Channel* implementation in Section 4.4.

Two topics are used in OpenDDS to exchange data and the state of the reader and therefore also two data structures are presented. The data structures are presented to OpenDDS using an IDL file (OMGIDL, 2017). It is chosen to implement unbounded sequences for the data exchange, because preliminary tests showed that a DDS topic per writer and reader pair resulted in a bad performance on the RaMstix. With unbounded sequences it is possible to bundle multiple data instances into one packet. Unbounded sequences in IDL show up as arrays in C++ and needs to be allocated. Memory allocation is not a real-time operation, but provides the ability to pack

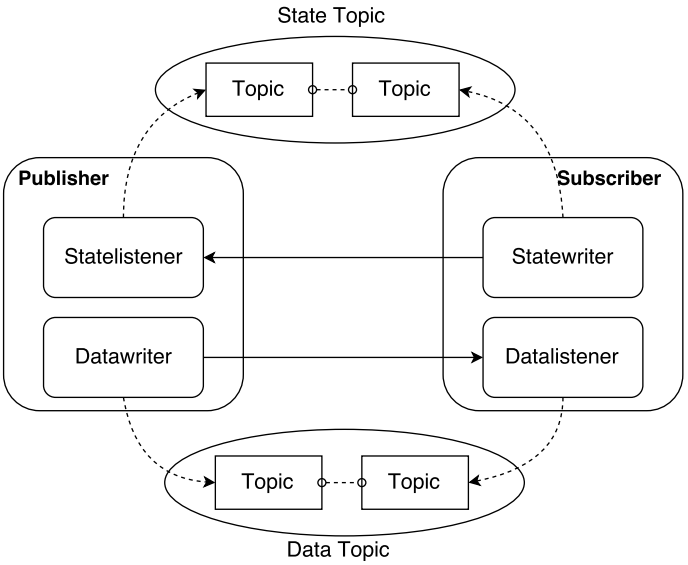


Figure 4.4: An overview of the initialized datawriter, statewriter, datalistener, and statelistener for rendezvous communication. The direction of data is from publisher to subscriber, and the direction of the state is from subscriber to publisher. The state topics and data topics should match before rendezvous communication is can be performed.

multiple data instances in one OpenDDS data structure, which results in less write actions to OpenDDS and therefore probably also in a better overall performance.

4.2.2 Realization

4.2.2.1 IDL file

The IDL format is used to specify the data structures. This IDL file is shown in Figure 4.5 and shows the data struct (*Packet*) with the unbounded sequences and the struct for the state exchange from reader to writer.

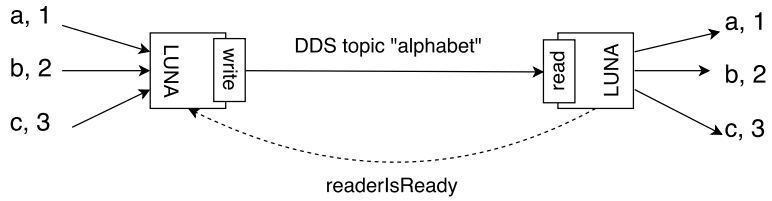
```
module Generic {

#pragma DCPS_DATA_TYPE "Generic::Packet"

typedef sequence<double> sampleList; //define a new type, named sampleList, unbounded
typedef sequence<string> topicList; // define a new type, named topicList, unbounded

    struct Packet {
        sampleList values; //Declare sequence of type sampleList.
        topicList topics; //Declare sequence of type topicList.
        double count; // Declare double value as counter.
    };

#pragma DCPS_DATA_TYPE "Generic::Rendez"
#pragma DCPS_DATA_KEY "Generic::Rendez status"

    struct Rendez {
        double status; //double value of 1 indicates reader is ready to receive
    };

};
```

Figure 4.5: The IDL file that is used to register topics with. If a publisher and subscriber use the same data format (and QoS) than the topics will match. Also the data format for the state exchange for rendezvous communication is provided in this file, and can notify the publisher the subscriber is ready to receive using a value.

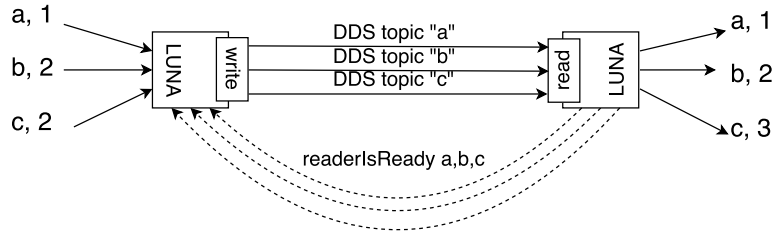
By using unbounded sequences for the data structure it is possible to bundle several subtopics into one DDS topic. With bundling better performance is obtained with respect to separate sending, which is shown in Section 4.2.3. The writers and readers representing those subtopics should however reside in a parallel construct such that the writers and readers are able to un-

block in one iteration. Figure 4.6a shows how the subtopics are bundled on one DDS topic. If rendezvous communication is enabled only one *reader ready* message needs to be send, notifying that all the readers in a parallel construct are ready.

The separate data structure should be used if the data has different endpoints. Also when the readers in one LUNA application are not in a parallel construct it is better to use the separate data structure. For the separate data structure each subtopic results in a separate DDS topic and is shown in Figure 4.6b. Every DDS topic has to present a separate *reader ready* message to the corresponding subscriber if rendezvous is enabled.



(a) A schematic representation of bundling different subtopics into one DDS topic. The single arrow from publisher to subscriber indicates information flow over one DDS topic. One state topic is used to state the readiness of the subscriber.



(b) A schematic overview of the separate data structure when different DDS topics are used for every subtopic. For every DDS topic a DDS publisher and subscriber pair is instantiated, which is illustrated with multiple arrows from publisher to subscriber. Also multiple state topics are used, because every subscriber has to notify its readiness.

Figure 4.6: A schematic that shows the differences between bundled and separate data structures.

4.2.2.2 Software architecture

The software structure of OpenDDS consists of a *Publisher* and a *Subscriber* class. The *Publisher* class is able to write data via a datawriter to a DDS topic and the *Subscriber* is able to receive data via a listener from a DDS topic. A DDS listener is implemented, because a *waitset* and a polling listener introduces extra latencies. The listener however is more resource heavy.

When rendezvous communication is enabled the *Publisher* class provides a statelistener and the *Subscriber* class a statewriter to be able to send and receive state updates. The realized software architecture is shown in Figure 4.7.

The *writeData* function in the *Publisher* class calls the *write* member function in the datawriter to write the data to DDS. The *writeData* function needs two vectors as arguments, one with subtopic names and one with data corresponding to the subtopics. These vectors are placed in arrays that are provided by the IDL file. The algorithm that places these values and topics in the C++ arrays provided by the IDL file is shown in Algorithm 1.

The function *readerIsReady* in the *Publisher* class calls the *readerIsReady* function in the statelistener. If the corresponding subscriber notified it is ready to read than this functions will return true, in all other cases it returns false.

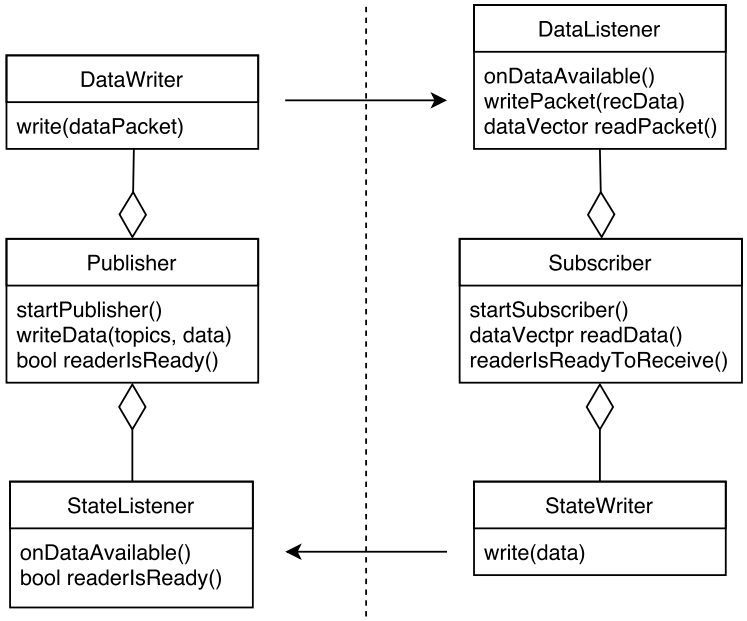


Figure 4.7: Software architecture of *Publisher* (left) and *Subscriber* (right). The listeners and writers are provided by OpenDDS and functions are added for extra functionality. The solid arrows indicate data flow and the dashed line the separation by OpenDDS.

Algorithm 1 Pseudo code of the function *writeData* in the OpenDDS implementation at the Publisher class of the OpenDDS interface.

Input: Two vectors, one dataVector and on topicVector

```
initialize arrays with the length of the dataVector
for every sample in the dataVector do
    place sample in allocated data array
for every topic in the topicVector do
    place topic in allocated topic array
place topic and data array in a dataPacket provided by the datastructure from IDL
Call write in datawriter with the dataPacket as argument
if rendezvous communication is enabled then
    Set the readerIsReady boolean to false in the statelistener
```

The datalistener at the *Subscriber* class fills a *dataVector* with the function *writePacket* whenever data is received and places the *recData* on the *dataVector*. When data was already present in the *dataVector* the received data is appended. The data consists of a subtopic and the corresponding value.

The *readData* in the *Subscriber* class calls the *readPacket* member function of the datalistener and returns the data it has received up until this call. This returned data consists of a vector with strings and the corresponding values. The string represents the topic and the double is the data.

The *readerIsReadyToReceive* member function of the *Subscriber* class sends a message on the state topic via the *write* function of the statewriter to indicate the reader is ready to receive data.

The *startSubscriber* and *startPublisher* functions instantiate the components that are necessary for OpenDDS to start exchanging messages. To instantiate a *Publisher* or *Subscriber* the constructor needs the following arguments:

- *Topic*: The topic name to what this channel is publishing/subscribing.

- *Configuration file*: The path to a configuration file that indicates what kind of transport is used.
- *QoS*: The QoS Reliability can be either "RELIABLE" or "BESTEFFORT", which is the quality of service for datawriter, datalistener and topic. RELIABLE transport is not supported for User Datagram Protocol (UDP) transport.
- *Rendezvous*: A boolean that indicates if an extra writer at the *Subscriber* class and a listener at *Publisher* class for rendezvous communication should be instantiated.

4.2.3 Tests

To test the basic functionality of this implementation a test is performed with five DDS channels. With more than one channel the behavior of multiple listeners can be tested.

The publisher and subscriber are hosted on different RaMstixes and the "Master" runs the Data-Centric Publish-Subscribe (DCPS) service provided by OpenDDS for endpoint detection. The setup for this test is shown in Figure 4.8.

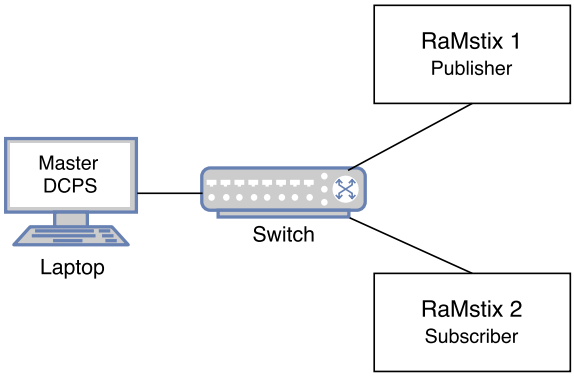


Figure 4.8: The setup used for the test of the OpenDDS implementation. The master, and two RaMstixes connect via a switch to the same network.

Tests are performed to show that the OpenDDS interface is functioning. Publish and subscribe, and rendezvous communication with separate and bundled data structure are tested.

A functional overview of this test is given in Figure 4.9. An application written in C++ calls functions of the OpenDDS interface to test the interface. The C++ writer tries to send its data every 10ms, and the C++ reader actively listens for data and continues if data is received for all the topics.

For these tests Transmission Control Protocol (TCP) transport is used with DCPS for endpoint detection. The Real-Time Publish-Subscribe (RTPS) transport is not used as it does not work with multiple participants on one host (When separate structure is used), and it fails to join multicast addresses from time to time. Time stamping is performed after each write to OpenDDS and after receiving data from OpenDDS. The difference between two timestamps at the C++ writer and two timestamps at the C++ reader are used to show the execution time of each iteration.

The publish-subscribe communication results for the bundled and separate data structure are shown in Figure 4.10. It is observed that the execution time of publishing messages using the separate data structure results in more jitter around the sending frequency of 100Hz (10ms), and is caused by writing five times more often to OpenDDS than when using a bundled data structure.

The C++ reader side also performs better with a bundled data structure than with separate data structure. This is easy to explain, because every listener waits until data is available without

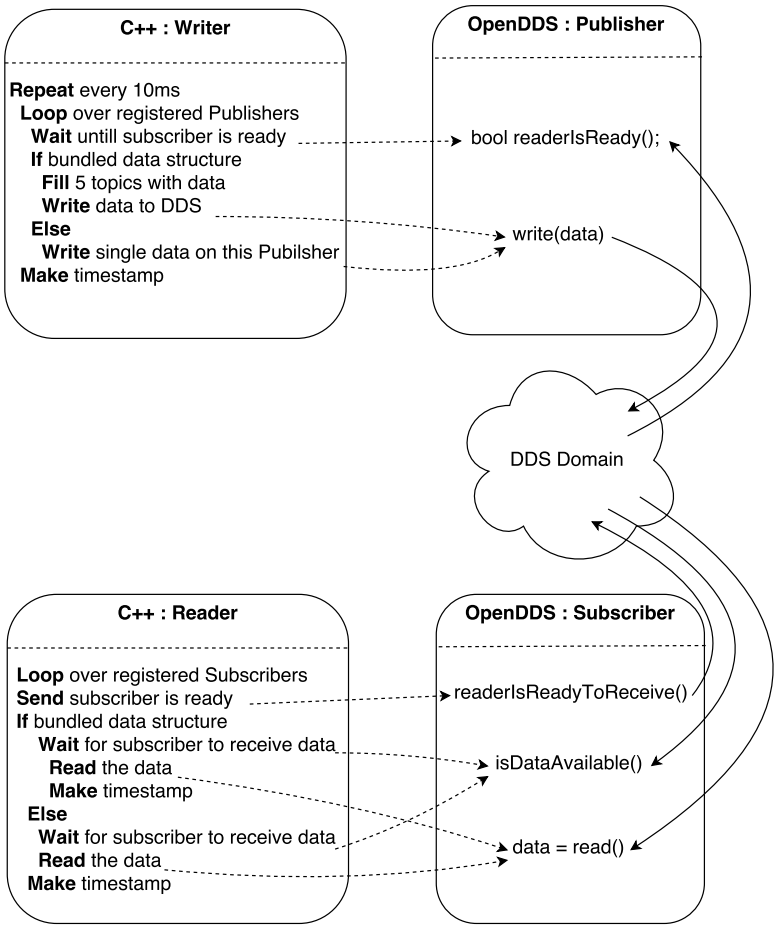


Figure 4.9: An overview of how the OpenDDS interface is used by a C++ program and presented as pseudo code in this figure. The dashed arrows indicate usage of the member function in another class and the solid arrows indicates the flow of data. "Wait until subscriber is ready" at the C++ Writer and "Send subscriber ready" are omitted for the publish-subscribe communication.

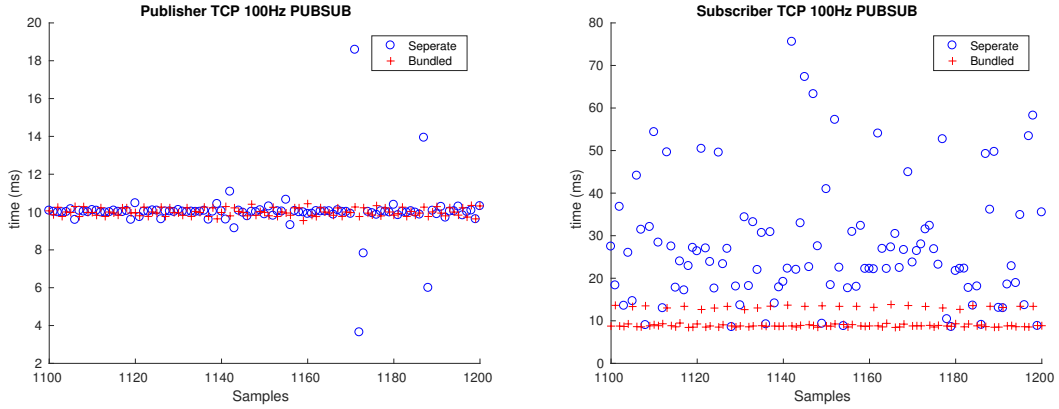
polling the other listeners in the meantime. This waiting for data expresses itself in bands separated by 4ms, which is equal to 250Hz and is the polling frequency for checking if data is available. The extra band 4ms above the 10ms line for the bundled data structure is due to the non availability of the data in the first iteration of the C++ reader and therefore the data is received the next iteration 4ms later.

For rendezvous communication similar behavior is obtained, but the C++ reader and the C++ writer are not decoupled anymore due to the state exchange, so the behavior is also translated to the publishing side and is shown in Figure 4.11.

The bundled data structure is able to maintain a 100Hz sending rate, but for the separate data structure this is not feasible due to long waiting times at the C++ reader for data for every listener. Note again the 4ms separation due to actively listening for the readiness of the reader and the availability of data.

These tests shows that the OpenDDS interface is functioning for rendezvous and publish-subscribe communication. The performance however is not directly translatable to the performance in the *Network Channel* implementation in LUNA that uses this interface.

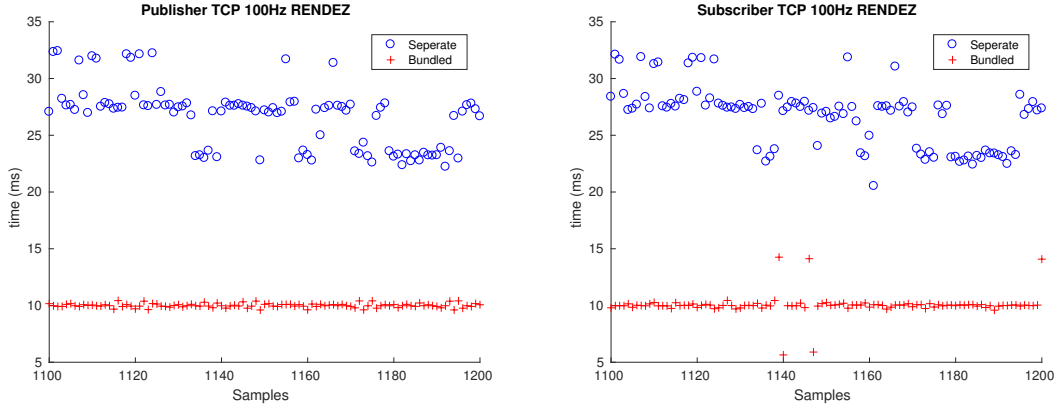
These tests are blocking when no data is available. The implementation in the *Network Channel* will not be blocking when there is no data, because it must also handle all the other readers.



(a) The time between timestamps at the publisher side for the publish-subscribe pattern.

(b) The time between timestamps at the subscriber side for the publish-subscribe pattern.

Figure 4.10: A comparison between a separate and bundled data structure for publish-subscribe communication



(a) The time between timestamps at the publisher side for rendezvous communication.

(b) The time between timestamps at the subscriber side for rendezvous communication.

Figure 4.11: A comparison between a separate and bundled data structure for rendezvous communication

Therefore the implementation in LUNA will not have such a sequential behavior, and therefore a better performance is expected when this OpenDDS interface is implemented for the *Network Channel*.

4.3 TERRA

4.3.1 Design

Bezemer (2014) wanted to get rid of sandboxed modelling software, and therefore hardware ports got implemented, such that the modelling software was able to communicate with the physical world. As a proof-of-principle the Mesa Anything I/O FPGA board was used to show the implementation of the proposed hardware ports in LUNA. The design that is used to add a hardware port to LUNA and TERRA proposed by Bezemer (2014) is shown in Figure 4.12.

The architecture of the TERRA plugin is rather complex and therefore it is chosen to append the *Network Channel* to this proof of principle implementation.

The *Network Channel* at TERRA will show up as *DDS port* to follow the naming convention of the other hardware ports.

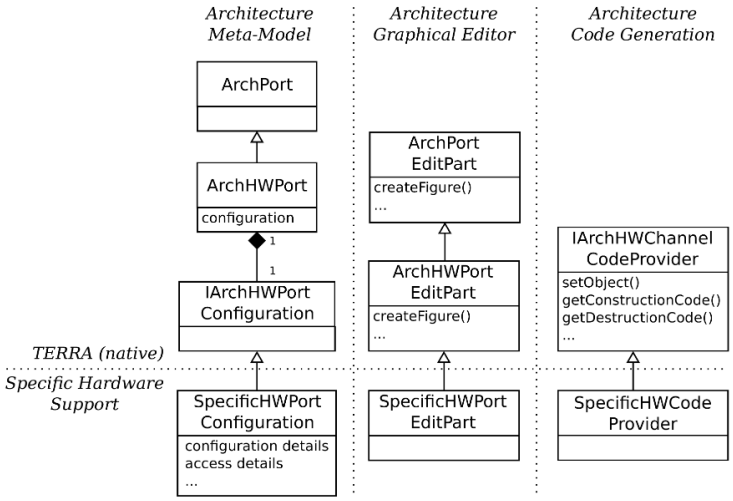


Figure 4.12: The design used for adding Hardware Ports to LUNA and TERRA(Bezemer, 2014).

The TERRA project contains a lot of plugins for its hardware ports, but the plugins relevant for appending a DDS port to the existing Mesa Anything I/O FPGA board hardware port implementation are:

- *nl.utwente.ce.terra.arch.hw.anyio.model*
- *nl.utwente.ce.terra.arch.hw.anyio.editor*
- *nl.utwente.ce.terra.arch.hw.anyio.codegen*

In *hw.anyio.model* a new configuration is added in where the parameters are provided that the *Network Channel* needs for instantiating a *Network Channel*. These parameters are shown in the "Properties" pane in TERRA once a DDS port is placed on the drawing pane of TERRA.

In *hw.anyio.editor* a validator is added that checks if the type that is connected to the channel is supported by the *Network Channel*. All types are set to be valid, such that it is possible to test different data types without changing the TERRA hardware port. The user has to take care of connecting the correct type of channel to the port. Also the *plugin.xml* file is changed such that the DDS port shows up in the TERRA application.

In *hw.anyio.codegen* the actual code generation is performed. The constructor of the *Network Channel* must be placed in a C++ file such that the constructor of the *Network Channel* can instantiate the writer or reader. Also the header file must be added to the top of the generated code.

4.3.2 Realization

In *hw.anyio.model* an extra model is added to *hwanyio.ecore* by copying and renaming another model. The extra model is shown in Figure 4.13 and is called *AnyioDDSConfiguration*.

Next a validator is added in *hw.anyio.editor* that checks if the type connected to the channel is supported. The validator for the DDS port is shown in Figure 4.14.

The *plugin.xml* file is appended with a hardware port configuration for the DDS hardware port, which is shown in Figure 4.15.

Next the part in *hw.anyio.codegen* where the code is generated is changed. First an existing template is copied and changed to match the constructor that initializes the *Network Channel*. In the template folder the *construction.egl* is altered to match the constructor of the Network Channel as is shown in Figure 4.16.

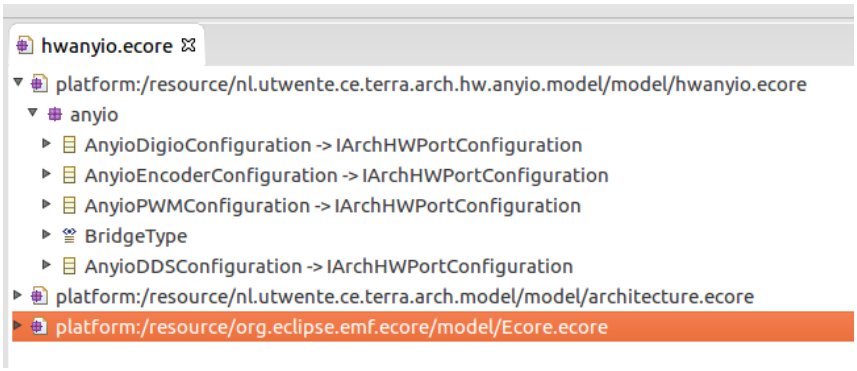


Figure 4.13: An extra model is added to the *hwanyio.ecore* file, and is renamed to *AnyioDDSConfiguration*

```
constraint DDSSupportedUnitType {
  guard: self.unitType.isDefined() and self.configuration.isTypeOf(AnyioDDSConfiguration)
  check: self.unitType.type == CPCType#INT or self.unitType.type == CPCType#UINT32 or
      self.unitType.type == CPCType#BOOL or self.unitType.type == CPCType#UINT16 or
      self.unitType.type == CPCType#UINT8 or self.unitType.type == CPCType#REAL
  message: "Invalid/unsupported unit type (" + self.unitType.type + ")."
}
```

Figure 4.14: The validator for the DDS port is added in *hw.anyio.editor* to make sure the correct type is connected to the DDS port. All types are currently supported for the TERRA DDS port.

```
<hardwareport
      configurationElement="nl.utwente.ce.terra.arch.hw.anyio.model.AnyioDDSConfiguration"
      provider="nl.utwente.ce.terra.arch.hw.anyio.editor.provider.AnyioDDSProvider">
</hardwareport>
```

Figure 4.15: The *plugin.xml* is appended with a hardware port configuration for the *Network Channel*.

```
[%
  var settingsName = channelName + "Settings";
  var configuration = hwPort.configuration;
%]

[% if(hwPort.Direction == PortDirection#INCOMING) { %]
  [%= channelName %] = new LUNA::CSP::DDS::DDSChannel<[%= codeGenerator.getHardwareChannelUnitType() %]>
                              ("[%= configuration.commontopic %]","[%= configuration.owntopic %]",
                              "[%= configuration.config %]", "[%= configuration.qos %]", 1,
                              [%= configuration.buffersize %], [%= configuration.rendezvous %]);
[% } else { %]
  [%= channelName %] = new LUNA::CSP::DDS::DDSChannel<[%= codeGenerator.getHardwareChannelUnitType() %]>
                              ("[%= configuration.commontopic %]","[%= configuration.owntopic %]",
                              "[%= configuration.config %]", "[%= configuration.qos %]", 0,
                              [%= configuration.buffersize %], [%= configuration.rendezvous %]);
[% } %]
```

Figure 4.16: The *construction.egl* file that gives a template for placing the constructor of the *Network Channel* in the generated code.

Then a Java class is copied from the *src* directory in the *hw.anyio.codegen* plugin and altered to implement a code generation for the *Network Channel*. This part adds the constructor and header file for the *Network Channel* actual to a C++ file and is shown in Figure 4.17.

Lastly the *plugin.xml* in the *hw.anyio.codegen* folder is altered such that the code generation is actually able to generate the *Network Channel* constructor and header. The *plugin.xml* file is shown in Figure 4.18

Appendix F shows how the source code of the TERRA application with DDS ports can be obtained.

```
@Override
public List<String> getHeaderFiles() throws Exception
{
    List<String> headers = new ArrayList<String>();
    headers.add("dds-channels/DDSChannel.h");
    return headers;
}
@Override
public String getConstructionCode(String channelName) throws Exception
{
    this.channelName = channelName;
    return processEGLFile(CONSTRUCTION_TEMPLATE, hwPort);
}
```

Figure 4.17: The *AnyioDDSCodeGenerator* that actually places the constructor and header in a C++ file. The *getConstructionCode* calls the file that is shown in Figure 4.16.

```
<provider
      handles="nl.utwente.ce.terra.arch.hw.anyio.model.AnyioDDSConfiguration"
      id="nl.utwente.ce.terra.arch.hw.anyio.codegen.AnyioDDSCodeGenerator"
      provider="nl.utwente.ce.terra.arch.hw.anyio.codegen.AnyioDDSCodeGenerator">
</provider>
```

Figure 4.18: The addition of the DDS code generation to the *plugin.xml* to enable code generation for the *Network Channel*.

4.3.3 Tests

A TERRA application with one DDS port at architecture level is constructed to show the correct functioning of the added DDS port.

When initializing a new architecture model the *Palette* shows up at the right side in TERRA. The DDS Port is located between the other hardware ports and is shown in Figure 4.19.

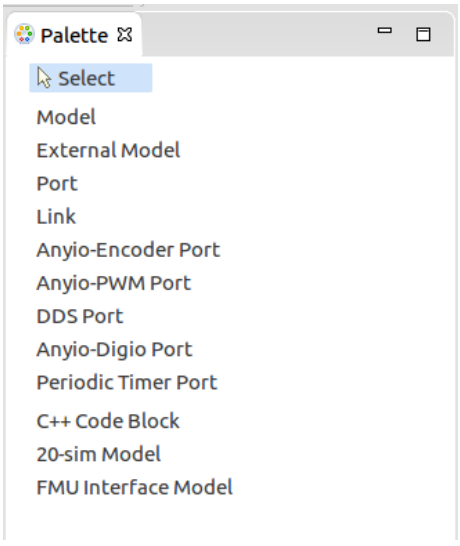


Figure 4.19: The DDS port as it shows up in the *Palette* of TERRA.

After placing a DDS port on the drawing frame in TERRA a properties pane shows up at the right bottom corner that enables the user to set some parameters as is shown in Figure 4.20. The settings entered in this properties pane are passed to the constructor that initializes the *Network Channel*. The buffersize, *commonTopic* (DDS topic), path to configuration file, *ownTopic* (subtopic),acQoS, and rendezvous are configurable parameters.

After code generation the constructor of the class that initializes the *Network Channel* (*DDSChannel*) is generated and filled with the configurations set by the user in TERRA and the header file is placed on top with the other dependencies as can be seen in Figure 4.21.

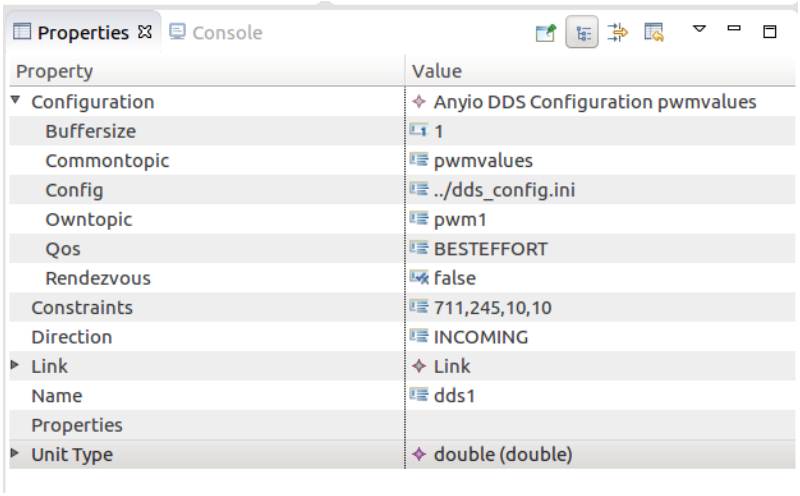


Figure 4.20: The properties pane in TERRA that shows the settings that are passed to the constructor that initializes the *Network Channel*.

```
// Initialize hardware channels
myPublishdds_port_out1_to_dds1Channel =
        new LUNA::CSP::DDS::DDSChannel<double>("pwmvalues","pwm1",
                "../dds_config.ini", "BESTEFFORT", 1, 1, true);
```

(a) The arguments that were set in the TERRA properties pane show up as arguments in the constructor of the *Network Channel* (*DDSChannel*).

```
// Hardware Channels
#include "csp/channels/PeriodicTimerChannel.h"
#include "dds-channels/DDSChannel.h"
```

(b) The code generation succesfully placed the header file (*DDSChannel.h*) of the network channel among the other dependencies.

Figure 4.21: Screenshots of the generated C++ code that call the constructor that initializes the *Network Channel* and places the necessary header files on top.

The user of TERRA is now able to use DDS ports as hardware port in TERRA. The next section describes the implementation of the *Network Channel* in LUNA such that placing a DDS port in TERRA actually delivers functionality to this hardware port.

4.4 LUNA implementation

4.4.1 Design

LUNA is component based, so by adding another component specifically for the DDS *Network Channel* it must be possible to use this *Network Channel* with LUNA. The file structure used by LUNA is shown in Figure 4.22.

```
LUNA
└─components
   └─component name
      ├─Makefile
      └─files
         └─All header and CPP files
```

Figure 4.22: File structure of a component in LUNA.

In the component folder another folder can be created that contains a makefile and a folder that holds the files for the Network Channel. The component folder name is chosen to be *dds-channels* to follow the convention of other components.

The *makefile* is responsible for linking all the project files, such that LUNA knows where to find the necessary files. The *makefile* is also used by the build menu of *LUNA* to show the component as an option to build.

A new software architecture based on ROS channels by van der Werff (2016) is designed for this *Network Channel* and is shown in Figure 4.23. The difference with the software architecture of the ROS channels is that this implementation has an extra class (*DDSManager*) that is instantiated as a *Singleton,* which means that only one object of that class can exist. This extra class is used as a pass through class for all the objects of *DDSChannel* to the *DDSReactor*.

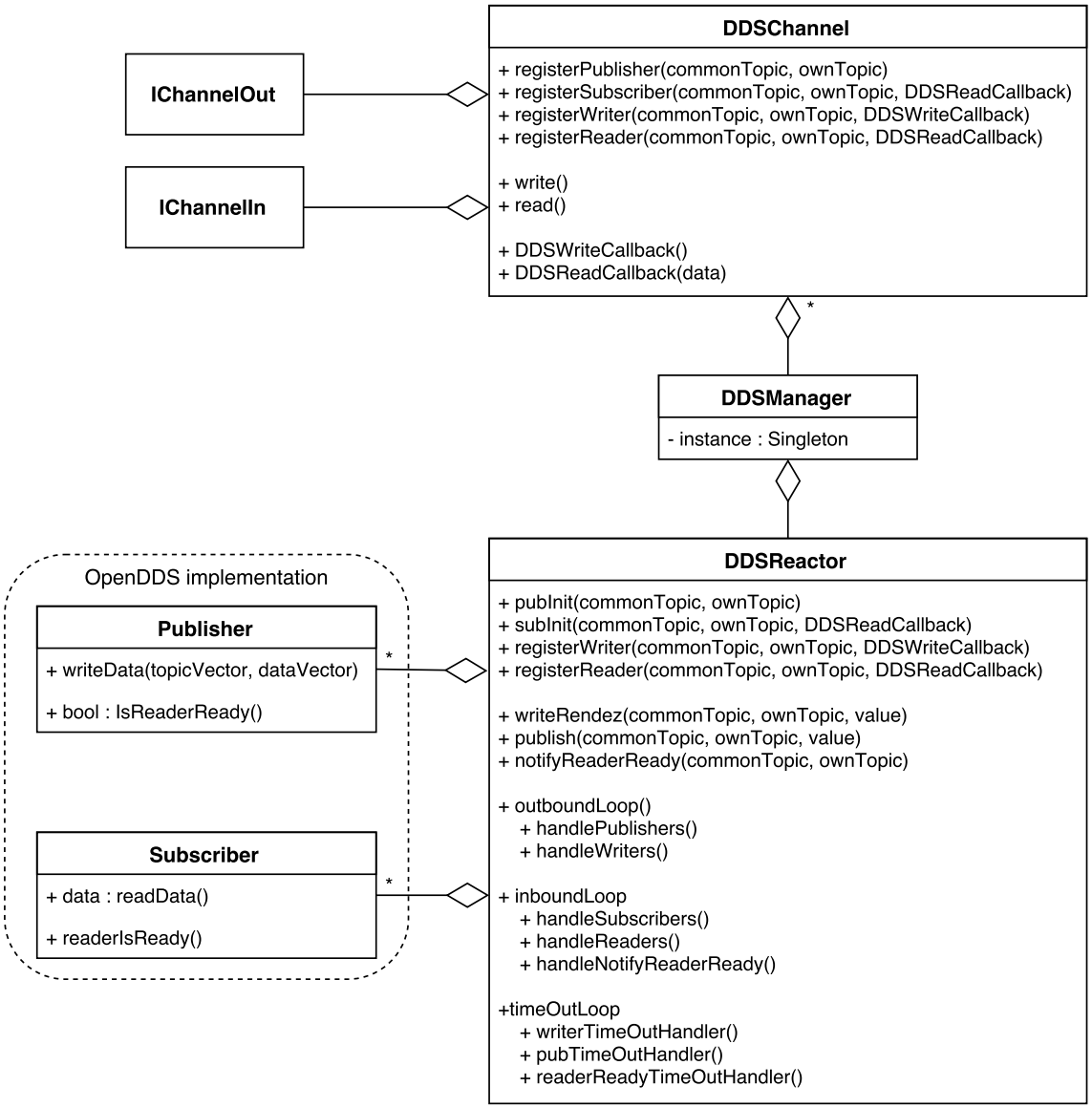


Figure 4.23: An overview of the software architecture used to design a *Network Channel* for LUNA. Only the member functions that provide the functionality are shown.

4.4.1.1 DDSChannel

On the top of Figure 4.23 the *DDSChannel* class is shown. The *DDSChannel* is where the component interfaces with the rest of the LUNA application via the *IChannelIn* and *IChannelOut* interfaces, which are provided by the CSP component in LUNA.

Every DDS port at TERRA level results in an object of the *DDSChannel* class. Invoking a new publisher or subscriber, blocking of readers and writers, and the type of communication (publish-subscribe or rendezvous) are managed by this class. The *DDSChannel* provides the interface to the rest of the LUNA application and therefore blocking of writers and readers must be implemented at this class.

Writers unblock if data is written to the OpenDDS publisher (only for rendezvous), and readers unblock once data is received by the OpenDDS subscriber. The *write* function provided by *IChannelIn* interface invokes a write to OpenDDS. If the writer is configured for rendezvous communication the writer will block until the data is written to a OpenDDS publisher.

At the reader a *read* function is provided by the *IChannelOut* interface that blocks until data is available. Blocking is implemented for publish-subscribe communication as well as for rendezvous communication. The reader with publish-subscribe communication does not necessarily have to block, but it is chosen to block such that loop frequency of the reader is dependent on the send frequency of the writer. With this dependency it is easier analyse the characteristics of the *Network Channel*, and a timer at the subscriber is not necessary.

Once a reader or writer is blocked there must be a way to unblock the reader or writer. Active polling for data can be implemented in the *DDSChannel*, but this results in an extra latency introduced by the polling frequency and also the CPU usage will increase as every reader is polling for data. Therefore callback functions are implemented that are given as an argument to the *DDSReactor* at registering of a blocking writer or reader. A function can be passed as argument by using the *std::function* provided by *C++11*. The callback function for the writer only has to unblock the writer where the callback function of the reader also has to place the received value in the buffer at DDSChannel.

Also a buffer is implemented in the *DDSChannel* that can store data presented by the *DDSReactor*. Whenever the buffer is full, old data will be overwritten.

4.4.1.2 DDSManager

The *DDSManager* class is implemented as a singleton and instantiates an object of the DDSReactor. A class implemented as a *Singleton* can only instantiate one object of the class and therefore all *DDSChannels* objects have access to the same *DDSManager* which makes it practical to use as a pass through class.

4.4.1.3 DDSReactor

The *DDSReactor* in Figure 4.23 takes care of actually publishing and subscribing to OpenDDS. One single thread could handle all the publishing and subscribing, but problems were observed, and therefore multiple threads are implemented which gave less deviation and therefore the multiple thread implementation is used from now on.

The threads that are implemented are: one for writing to OpenDDS, one for reading from OpenDDS and the third one for handling the time-out functionality.

The time-out thread makes sure a message is send to a certain topic if the last publish to that topic was longer ago than a pre-defined time. With a good functioning *Network Channel* no time-outs occur, but if a packet is lost with rendezvous communication a deadlock can occur. This deadlock can be resolved by re-sending to the topic that caused the time-out.

The *DDSReactor* instantiates OpenDDS classes for publishing and/or subscribing to a topic.

4.4.1.4 Communication types

For the communication over the *Network Channel* two types of communication should be available:

- *Publish and Subscribe*: This is the most basic communication pattern and also implemented by most other communication protocols. The publisher publishes data without knowing the subscriber was ready. The subscriber simply receives the data without notifying its readiness.
- *Rendezvous*: CSP is used in LUNA and therefore rendezvous communication is used in the LUNA application. Therefore the *Network Channel* should support rendezvous communication as well to maintain the synchrony between two systems following CSP theory. The publisher for rendezvous communication waits until the subscriber is ready to receive before sending the data.

For the publish-subscribe communication the writer at *DDSChannel* is not blocking and the reader at *DDSChannel* blocks until the registered callback function with data is called from the *DDSReactor*. A timing sequence that shows how messages are exchanged for the publish-subscribe communication is shown in Figure 4.24.

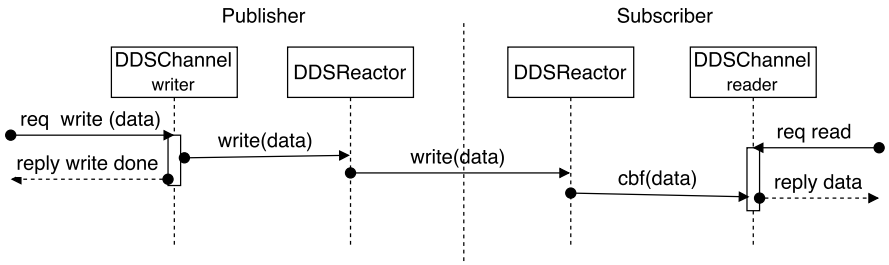


Figure 4.24: A timing sequence that shows how publish-subscribe communication is performed. The writer does not block and the reader unblocks as soon as the callback function is called from the DDSReactor.

For rendezvous communication the writer on *DDSChannel* is blocking until the data is send on the DDS publisher at the *DDSReactor*. The reader on *DDSChannel* is blocked until the callback function with the received data is called from the *DDSReactor*. The reader at *DDSChannel* states its readiness to the DDS subscriber at the *DDSReactor*. Once the readiness of the reader is received at the DDS publisher the *DDSReactor* is allowed to send the data to the DDS subscriber. The behavior of the writer and reader is shown using a timing sequence in Figure 4.25.

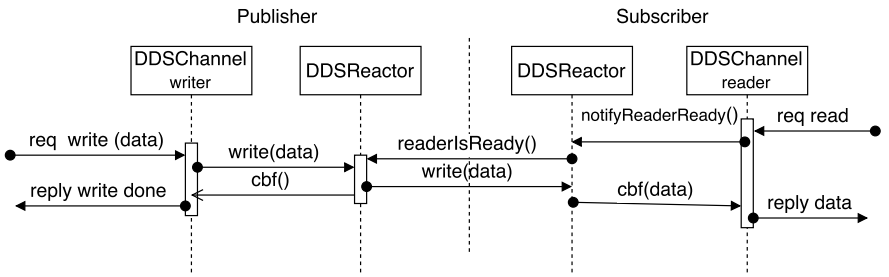


Figure 4.25: A timing sequence that shows how rendezvous communication is performed. Writer and reader at *DDSChannel* block until data is written to OpenDDS or received from OpenDDS respectively.

4.4.1.5 Publishing data

Multiple instances of the *DDSChannel* call the same member function in the *DDSReactor* to initiate a publish onto a DDS topic. The *DDSChannel* objects are running concurrent and therefore a queue must be implemented that guarantees thread safety.

To make it a thread safe operation one could use the mutex wrapper *lock_guard* supported by *C++11*. If a thread enters this function than other threads wait with entering this function until the lock is ended. The same functionality but different implementation is also provided by the *lockfree* implementation by the Boost library (Blechmann, 2011), and by the First In First Out (FIFO) queues of MoodyCamel (Moodycamel, 2014).

MoodyCamel compared its implementation with the *lock_guard* and *lockfree* from the Boost library. Also Qihoo360 (2017) tested these three implementations. Both tests conclude that MoodyCamel has the fastest writing and reading speeds when a queue is accessed by multiple threads. Therefore the FIFO queue implementation of MoodyCamel is used for the implementation of the queues.

Both communication types uses queues for sending messages. Every message that has to be published contains three arguments:

- *commonTopic*: The DDS topic that the data should be published to. This topic must be registered by a OpenDDS publisher.
- *ownTopic*: This is the subtopic on a *commonTopic*. Multiple *ownTopics* can reside in one *commonTopic* as this is supported by the data structure provided by OpenDDS.
- *Data value*: The value that the writer wants to send to the reader.

Multiple *ownTopics* can reside in one *commonTopic*. The functionality of bundling of *ownTopics* into a bundle is shown in Figure 4.26.

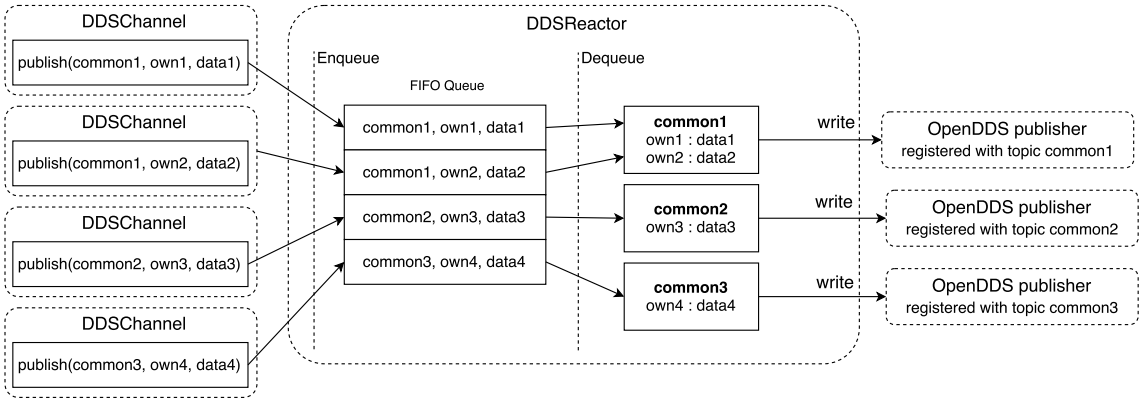


Figure 4.26: The functionality of publishing a message using queues. Messages are placed on the queue from multiple instances of the *DDSChannel*. The first items placed in a queue will be dequeued first as well (FIFO). Note that the messages are bundled by the *commonTopics* before writing to the corresponding OpenDDS publisher. The *DDSManager* is left out as as it only passes the messages.

The *outboundLoop* in the *DDSReactor* is responsible for dequeueing the messages from the queue.

Every iteration of the *outboundLoop* all the messages on the queue will be dequeued by the *handlePublishers* function for publish-subscribe communication and by the *handleWriters* function for rendezvous communication. All *ownTopics* on the queue with the same *commonTopic* are bundled until all the messages are dequeued.

The default implementation dequeues all the messages and sends them to the OpenDDS publisher. However, a problem arises if not all the messages were presented yet to the queue and therefore the amount of messages for a certain DDS topic is not equal to the amount of registered writers. For rendezvous messaging this will results in a deadlock as not all readers are unblocked in one iteration, which means that the reading side does not state its readiness again and the writing side send no data as it waits for the readiness of the reader. Therefore for rendezvous messaging a wait function must be implemented that waits for all registered writers to have presented their data to the queue. The wait function is illustrated for four writers and readers writing to the same *commonTopic* and is shown in Figure 4.27.

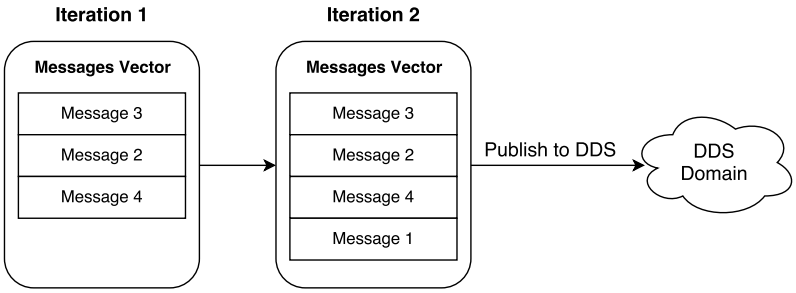


Figure 4.27: An illustration of the wait functionality for one *commonTopic* and four registered writers. In the first iteration not all four writers have presented their data to the queue and in the second iteration the bundle is complete as all messages from all the writers are available.

4.4.1.6 Subscribing to data

Where the *outboundLoop* is responsible for sending messages the *inboundLoop* at the *DDSReactor* is responsible for receiving the messages and placing the data on the callback function that corresponds to the received *ownTopic*. The data is processed once it is available and therefore no queue is needed for receiving data.

However, one queue is implemented that handles the *reader is ready* messages for rendezvous communication. The reader at *DDSChannel* invokes the call to notify the corresponding writer by calling *notifyReaderReady* function in the *DDSReactor*, which enqueues the *commonTopic* and is matched to a registered OpenDDS publisher topic at dequeueing. Once matched, a *readerIsReady* message on that topic is published. After this message the reader will block until data is received via the callback function.

For publish-subscribe communication the reader at the *DDSChannel* will block without notifying its readiness.

4.4.1.7 Time out

The *timeOutHandlerLoop* takes care of sending a message to a certain topic if the last publish to that topic was longer ago than a pre-defined value. This time-out occurs when a deadlock is obtained by for example missing packages. If a deadlock occurs it is resolved by the *timeOutHandlerLoop*. If rendezvous communication is used the synchrony is lost for a moment, but it is more important to have data than to be in a deadlock forever. The *timeOutHandlerLoop* calls in every iteration three functions, namely:

- *writerTimeOutHandler()*: This function checks if the last publish to a topic for rendezvous communication was not longer ago than a pre-configured time-out time.
- *pubTimeOutHandler()*: This function checks if the last publish to a topic for publish-subscribe communication was not longer ago than a pre-configured time-out time.

- *readerReadyTimeOutHandler()*: This function checks if the last *reader ready* notification to a topic was not longer ago than a pre-configured time-out time.

4.4.2 Realization

The essence of the realization is discussed in this section. More details on the implementation are given with pseudo code and flow charts in Appendix A.

4.4.2.1 Component integration in LUNA

Following the generic file structure of a component a new folder is added with the name *dds-channels* to follow the naming convention of other components. Inside this folder a *makefile* is present that specifies which files are needed for the *dds-channel* component. OpenDDS is used and therefore LUNA needs the location of the OpenDDS binaries and headers. The *setenv.sh* script that is provided by OpenDDS sets the environment variables. In *rules.mk* (located in top directory of LUNA), it is checked if the environment variables are set.

The result is that the *Network Channel* can be build with LUNA, because it is presented as a build option in the LUNA build configuration.

4.4.2.2 LUNA implementation

The software architecture for the *Network Channel* as is shown in Figure 4.23 consists of three classes, and depending on the setup also a *Publisher* or *Subscriber* object is instantiated.

The most important functionality of the *Network Channel* is implemented in the looping threads. Three threads are used in the *DDSReactor*:

- *inboundLoop*: This loop handles all incoming messages and is started when the first subscriber is instantiated.
- *outboundLoop*: This loop handles all outgoing messages and is started when the first publisher is instantiated.
- *timeOutHandlerLoop*: This loop is started as soon as the first publisher or subscriber is instantiated. This loop sends a message to a DDS topic if the last write to that topic was longer ago than a pre-defined time.

Those three threads implement the core functionality of the *Network Channel* and call the implemented functions that enables the *Network Channel* to perform publish-subscribe and rendezvous communication.

The *outboundLoop* dequeues the messages from a Moodycamel (2014) queue. The messages on a queue contain a *commonTopic*, *ownTopic*, and a data value. The *outboundLoop* implements the following functions:

- *handlePublishers()*: This function implements the dequeueing of the publish-subscribe queue, which is filled by the implemented *publish* function in the *DDSReactor*. All messages are written to the corresponding DDS topic (*ownTopic*) by this function.
- *handleWriters()*: This function implements the dequeueing of the rendezvous queue, which is filled by the implemented *writeRendez* function in the *DDSReactor*. All messages are written to the corresponding DDS topic (*ownTopic*) if the corresponding is ready to read. Also the waiting functionality is implemented in the *writeRendez* function.

The *handleWriters()* function also implements a waiting functionality. This waiting functionality makes sure that the bundle contains an equal amount of data as there are *ownTopics*. The pseudo code of how the waiting functionality is implemented is shown in Algorithm 2.

This algorithm waits another iteration if not all the data was present in the queue at time of dequeueing.

Algorithm 2 Pseudo code of the function *handleWriters* that dequeues the messages on the queue with waiting enabled. The writer is written to OpenDDS if the *dataVector* size is equal to the amount of registered writers.

```
1:  for every entry of publisher in publisherVector do
2:      if accompanying subscriber is ready then
3:          while queue with messages is not empty do
4:              Dequeue a message
5:              if message commonTopic is equal to the publisher topic then
6:                  append ownTopic to topicVector
7:                  append value to dataVector
8:                  for every entry in the callbackVector do
9:                      if callback topic is equal to the ownTopic of the message then
10:                         Add callback to callback vector.
11:             else
12:                 add message to reenqueueVector.
13:         if dataVector size is equal to number of registered writers then
14:             write data to write function of publisher
15:             Handle all callback functions in callbackVector.
16:             clear topicVector, dataVector and callbackVector.
17:         if re-enqueue vector is not empty then
18:             re-enqueue this vector in the message queue.
```

As also mentioned in the design section, callback functions are presented to the *DDSReactor* to unblock a writer or reader. The *inboundLoop* receives the messages and implements the following functions:

- *handleSubscribers()*: This function receives the message for the publish-subscribe communication. With the callback of the reader that is provided at the registration the reader can be unblocked.
- *handleReaders()*:This function receives the message for the rendezvous communication. With the callback of the reader that is provided at the registration the reader can be unblocked.
- *handleReaderReadyNotification()*: This function dequeues the queue with *reader is ready* messages. This queue is filled by the *notifyReaderReady* function implemented in the *DDSReactor*. All messages are written to the corresponding DDS topic by this function.

The *timeOutHandlerLoop* makes sure the last publish was not longer ago than a pre-defined time (currently hard coded on 10s), and calls the following functions:

- *pubTimeOutHandler()*: This function re-publishes a value if the previous publish to a publish-subscribe topic was longer ago than a pre-defined time.
- *writerTimeOutHandler()*: This function re-publishes a value if the previous publish to a rendezvous topic was longer ago than a pre-defined time.
- *readerReadyTimeOutHandler()*: This function presents the state again if the state was presented longer ago than a pre-defined time.

The three threads and the functions in those threads implement most of the functionality of the *Network Channel*. The implementation of the connection between the *DDSChannel* and the *DDSReactor* is not discussed in this section, and also the implementation of the functions is not detailed in this section.

Stating the implementation of the *Network Channel* in words would merely result in a repetition of the design section, and showing the actual implementation would result in a long realization section due to all flow charts and pseudo code listings. Therefore it is chosen to present the realization in Appendix A. This appendix contains a detailed implementation by using pseudo code and flow charts to show the realization of the member functions and the relation of *DDSChannel*, *DDSManager* and *DDSReactor*.

4.4.3 Tests

A summary of the performed tests is discussed in this section. A more extensive analysis of the results from the tests is discussed in Appendix B.

Four tests are conducted to investigate the performance of this *Network Channel* implementation:

- *Transport*
- *Bundled data validation*
- *Latency analysis*
- *Stress test*

All tests use the setup that is shown in Figure 4.28, with exception of the load test in the Stress test. The setup consists of a DCPS server that provides the endpoint discovery for the OpenDDS instantiations and two RaMstixes of which one is the publisher and one is the subscriber.

The testing setup consists of 10 parallel writers for the publisher and 10 parallel readers for the subscriber at TERRA level.

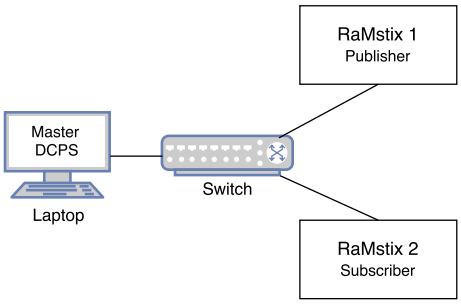


Figure 4.28: The setup used for all the tests. The "Master" hosts the DCPS server, which is used for endpoint discovery. One of the RaMstixes executes the publisher code and the other the subscriber code.

4.4.3.1 Transports

The first test contains of a transport test that investigates the differences between UDP and TCP transport. UDP delivers no guarantees in terms of reliability, where TCP offers reliable transport of data (Kurose and Ross, 2012).

For rendezvous communication it is observed that UDP indeed introduces some packet loss as the unblocking of the readers takes sometimes more than one iteration, where the TCP transport shows no such behavior. The subscriber for rendezvous communication with UDP versus TCP transport is shown in Figure 4.29

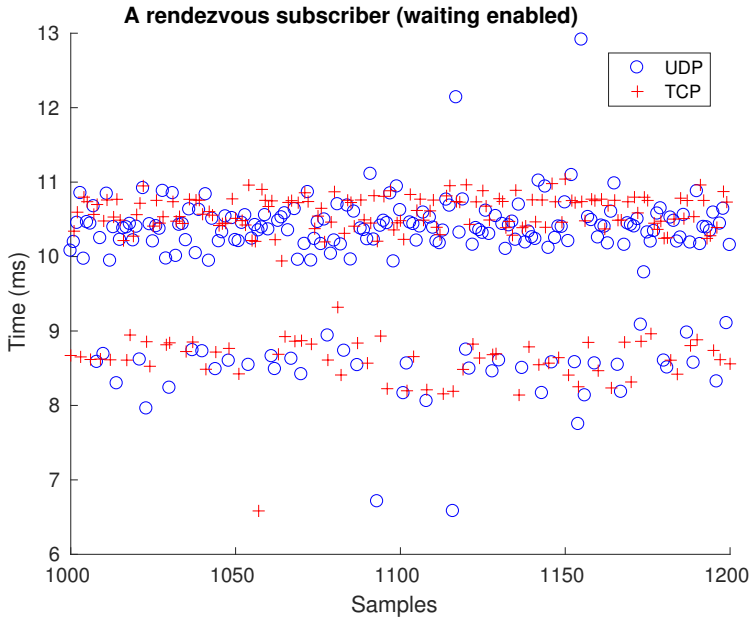


Figure 4.29: The time between two timestamps for a rendezvous subscriber using different transports (UDP and TCP).

4.4.3.2 Bundled data validation

This test validates the functionality of the bundling of data for a certain *commonTopic*. It is observed that waiting for a full packet has less impact on the publishing side as less writes are performed to OpenDDS. Also at the subscribing side is observed that the reader is unblocking in one iteration instead of unblocking in one or more than one iteration without the waiting functionality.

A test is executed with 100 000 samples at a sending rate of 100 Hz to validate the functioning of data bundling. It is observed that in all cases that the data from all the writers were present in a bundle, and therefore bundling works as expected at 100Hz.

4.4.3.3 Latency analysis

The latency analysis is performed to show the latencies in the *Network Channel* for the publisher and subscriber.

It is observed that writing to OpenDDS is an expensive operations, because a lot more jitter on the publishing side is observed in comparison with a comparable implementation with ZeroMQ by van de Ridder (2017).

A latency analysis is performed to investigate how long certain actions at a publisher or subscriber take. Details of the investigated latencies are discussed in Appendix B.

It is observed that the differences in latencies between UDP and TCP are neglectable.

It is also observed that the time to write to OpenDDS takes indeed a significant amount of time, namely 1.3413ms of the total latency of 2.5ms, from invoking a publish to actual publishing on OpenDDS. To check if this is a OpenDDS limitation or an implementation issue, the same test is repeated on the RaMstix with the default linux scheduler (OTHER) instead of the Xenomai threads (FIFO). The latency test is also performed on a laptop running Ubuntu.

It is observed that the laptop (with more resources) is able to write faster to OpenDDS with a lower standard deviation. But also by using the default linux scheduler (OTHER) on a RaMstix,

the average writing time has also dropped. The results of the three implementations is shown in Figure 4.30.

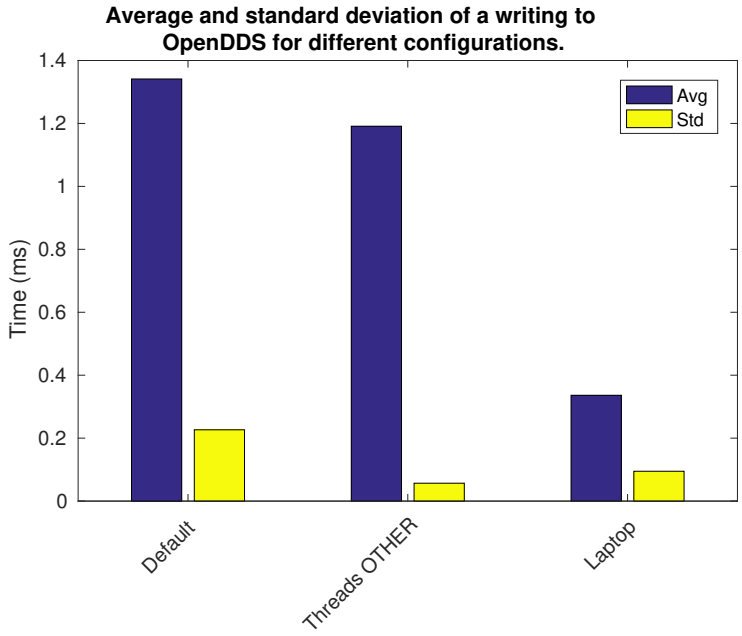


Figure 4.30: Average time and standard deviation over 1600 samples for three different configurations for writing to OpenDDS on a TCP transport. "Default" is the FIFO scheduler with priority 88. The left bar presents the average value and the right bar the standard deviation. The writing times for UDP are comparable.

The overall performance with the OTHER scheduler on the RaMstix is worse due to rescheduling of the OTHER threads by higher priority threads, however it shows that by using the Xenomai scheduler higher writing times and higher standard deviations are obtained. The Xenomai scheduler encounters mode switches (Xenomai, 2014), which probably cause these extra latencies with more jitter.

At the subscribing side is observed that the time between calling the callback function and actual unblocking of the reader in the *DDSChannel* is rather high. This unblocking is shown in Figure 4.31.

These bands are not clearly linked to an event and therefore hard to analyse. The RaMstix is probably busy with other processes with higher priority threads such that the threads that unblock the reader are getting rescheduled.

4.4.3.4 Stress test

The stress test consists of two tests. One with introducing extra load on the network and one with increasing the sending frequency.

For the load test a Raspberry Pi (RPI) is added to the setup. The setup is shown in Figure 4.32.

The load test showed that by saturating the network with data more jitter was observed at the writer and the reader side. The writer has a more busy network stack, resulting in more jitter at the writing times and the reader receives the data less deterministic resulting in more jitter at the receiving side. A more thorough analysis is performed in Appendix B.

The second test consists of several sending frequencies to find a maximum sending frequency. The same test setup as the first three test is used again. It is observed that the maximum sending speed for UDP is somewhat lower than the maximum for TCP, which is caused by lost packages in the network, resulting in more iterations before all readers are unblocked.

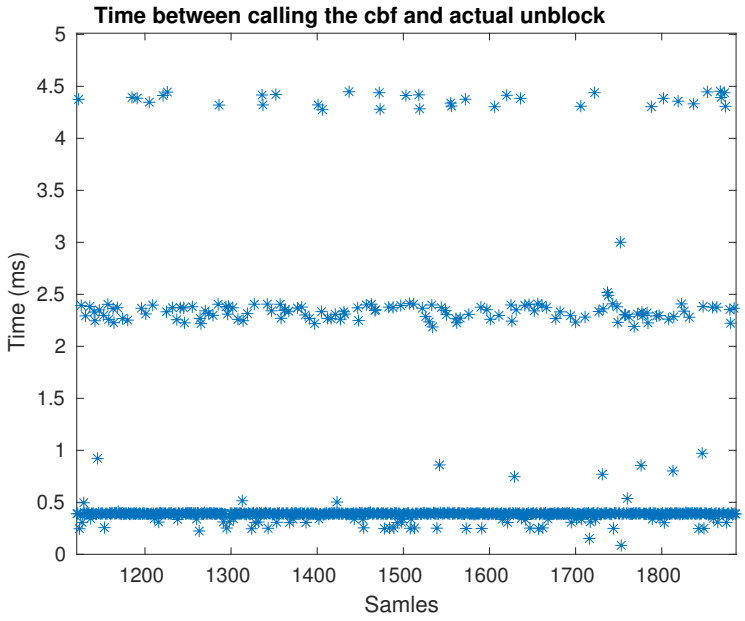


Figure 4.31: The difference between the timestamp just after calling the callback function and the actual unblocking of the reader in the *DDSChannel*.

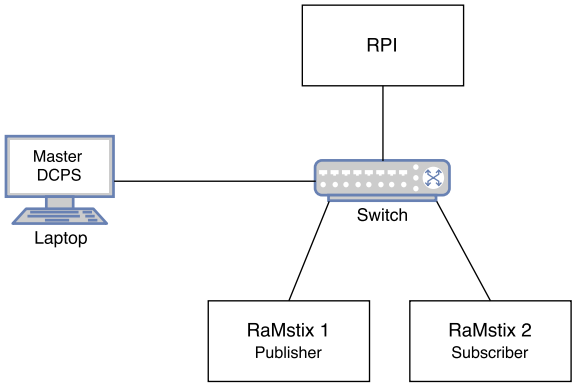


Figure 4.32: Test setup for introducing extra load to the network.

For publish-subscribe communication a maximum sending frequency of around 250Hz is obtained and for rendezvous communication a maximum of 200Hz. However for rendezvous communication and sending frequencies higher than 200Hz the waiting for complete packets fails. It is recommended to explicitly check for the available topics instead of the amount of data in the bundle. A more thorough analysis is performed in Appendix B.

4.5 Conclusion

An OpenDDS interface is provided that supports publish-subscribe and rendezvous communication. This interface is used by the LUNA implementation to add the functionality.

A DDS port is appended to the Mesa AnyIO hardware port in TERRA. TERRA is able to include the *Network Channel* in the executable code, such that the user is able to use the *Network Channel* without adding the code manually. TERRA generates executable code, which contains also the constructor and header file of the *Network Channel*.

The LUNA implementation integrates the OpenDDS interface to provide the functionality of the *Network Channel*. Bundling of data is implemented as instantiating a separate DDS topic

for every writer and reader resulted in a bad performance on resource constraint platform as the RaMstix.

From tests of the *Network Channel* is observed that TCP transport is indeed reliable. Also it is observed that writing to OpenDDS takes a significant amount of time and is caused by the Xenomai scheduler that causes mode switches. Also the RaMstix is a resource constraint platform, resulting in slower writing times than a more resource rich platform.

A maximum writing and reading frequency of 250Hz is obtained for publish-subscribe communication and 200Hz for rendezvous communication. Faster writing and reading frequencies were obtained for the rendezvous communication, but resulted in failing of the waiting functionality. It is therefore recommended to update the waiting functionality to check explicitly for topic names, instead of counting the amount of messages in a bundle.

Also it is observed that with saturating the network with extra traffic the subscriber more often unblocks in three iterations. Three iterations were sometimes also present with no traffic on the network. However, due to the looping threads a clear difference between traffic and no traffic on the network is not obtained as one loop iteration takes a certain amount of time that may be more than the extra latency on the network. With more traffic on the network the writing side introduces more jitter due to a more busy network stack.

It is recommended to get rid of the loop threads by presenting the callback functions to the OpenDDS interface and by writing to OpenDDS once a complete packet is present. This will result in a better performance as the latencies introduced by the looping threads are omitted. It will also result in a better analysis of external influences on the *Network Channel* as external influences will influence the writing and reading times directly. An extra thought should be given in how to handle multiple instances of OpenDDS publisher and subscriber pairs when the loops are omitted.

5 EtherCAT interface

In this chapter the development of an interface is provided for communication with the youBot. Also a controller is presented that is able to move the youBot base using the joystick values as input.

5.1 Design

The youBot is used as demonstrator for the *Network Channel* and therefore an interface must be provided that interfaces between LUNA and the EtherCAT master to control the youBot joints. The relation between LUNA, the EtherCAT interface and the youBot is shown in Figure 5.1.

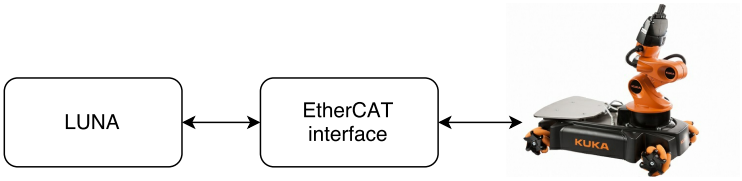


Figure 5.1: The relation between LUNA, the EtherCAT interface and the youBot.

For the design of the EtherCAT interface two approaches are considered: designing a component in LUNA, or use a C++ code block provided by LUNA. The latter is chosen for the following reasons:

1. The youBot provides no ESI file that describes how the motor drivers should be controlled via EtherCAT (Chapter 2.4). Therefore the control of the joints is hard coded in the youBot API and the implementation of an EtherCAT component in LUNA is therefore not generic.

2. The focus of this project is on designing a *Network Channel* component in LUNA. A C++ code block implementation is implemented faster than writing a new component for LUNA.

3. Using a code block the same interface as in Spil (2016) can be used, such that the differences between the interface in LUNA and 20-sim can be investigated.

Spil (2016) designed an EtherCAT interface specifically for the youBot and integrated this in 20-sim 4C. Periodic peaks in the latency of the execution time were observed in this implementation. By using the same EtherCAT interface in LUNA instead of 20-sim 4C it can be investigated whether or not this latency is caused by the execution engine or by the EtherCAT master on RaMstix.

The youBot API is used to extract the commands that needs to be send to the EtherCAT master for joint control. The Simple Open EtherCAT Master (SOEM) EtherCAT master is used as it is also used by the youBot API. The same EtherCAT interface as Spil (2016) is used and therefore only direct control of the youBot motors is provided in this interface such that a controller (for example the inverse kinematic controller from Spil (2016)) is able to control the motors at joint level.

5.2 Realization

The EtherCAT interface from Spil (2016) is used and provides the following functionalities:

- *Set* functions for setting: joint current, joint position and joint velocity.

- *Get* functions for getting: joint current, joint position and joint velocity.
- Function to send a mailbox message for selecting options in the youBot motor drivers.
- Functions to initialize base and manipulator.
- Function to stop the youBot.

Spil (2016) provides a youBot interface that uses SOEM as EtherCAT master and C functions to provide the interface for joint control. The youBot API was used to study the correct commands to interface with the youBot via EtherCAT. The provided C functions in the youBot interface and the realization with the SOEM EtherCAT master is shown in Figure 5.2.

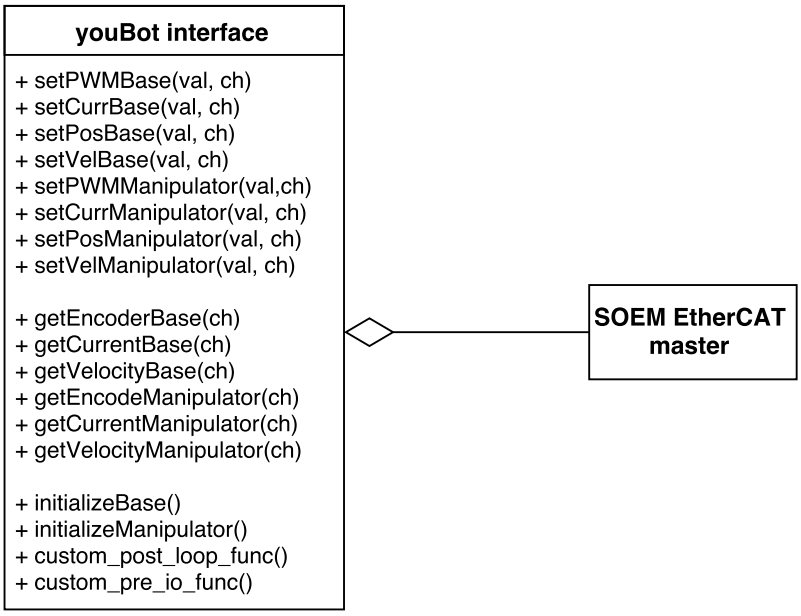


Figure 5.2: The software architecture for the EtherCAT interface. The member functions provide: PWM, current, position and velocity control for the base and manipulator. Also the functions that initializes the base and manipulator, and the functions that actual receives and sends the EtherCAT frames are shown. "val" signifies the value and "ch" signifies the channel that indicates the motor driver.

A library is made from these C functions such that it can be integrated in every C++ application (including LUNA).

The execution time for the EtherCAT interface with SOEM on a RaMstix and laptop are measured to investigate and compare these results to the execution time obtained by the EtherCAT interface in 20-sim 4C from Spil (2016).

A discussion of the execution time of this EtherCAT interface in LUNA is provided in Chapter 6.

5.3 Test

5.3.1 Setup

A C++ application is written that utilizes the EtherCAT interface such that it can be executed on a RaMstix and a laptop.

The constructed test setup with pseudo code of the C++ application is shown in Figure 5.3. No LUNA is used such that only the behavior of the EtherCAT interface can be investigated. The program is compiled for the non real time default linux scheduler (OTHER) such that no Xenomai is used.

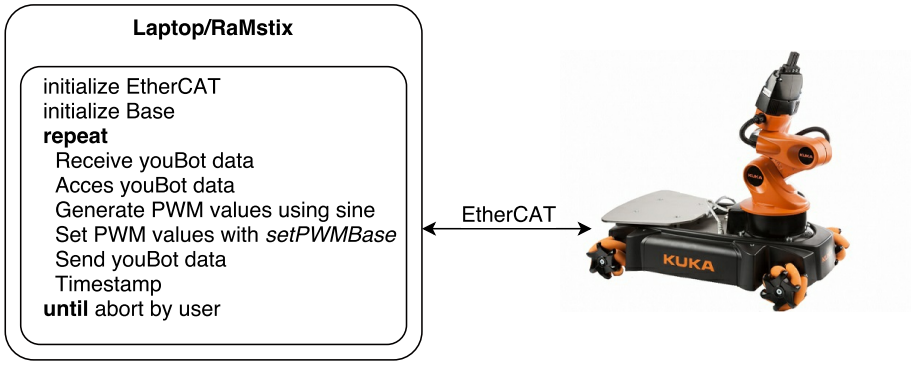
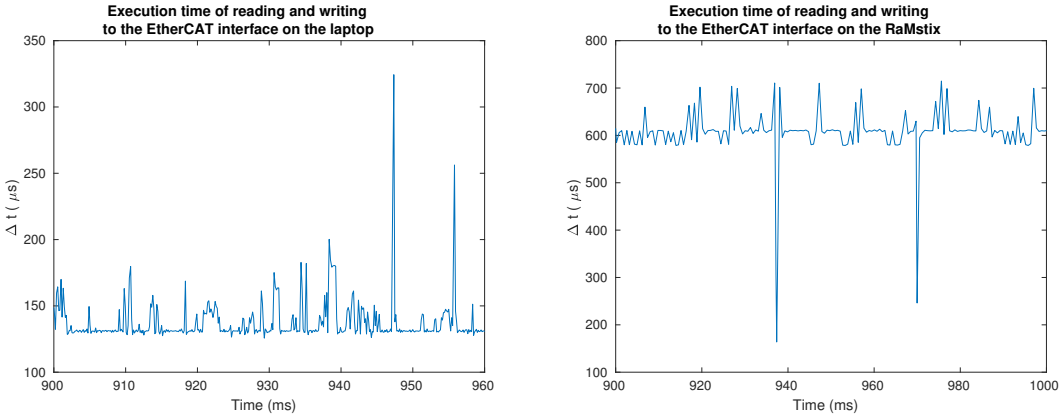


Figure 5.3: The setup consists of a C++ application that is executed on the RaMstix and laptop. The C++ application reads and writes to the EtherCAT interface. A sine function is used to generate PWM values for the wheels.

5.3.2 Results

The loop is timestamped such that the execution time of a loop can be calculated. The execution time of a loop on the laptop is shown in Figure 5.4a and for the RaMstix in Figure 5.4b.



(a) The execution time of the C++ application executed on a laptop.

(b) The execution time of the C++ application executed on a RaMstix.

Figure 5.4: The execution times of the C++ application that communicates with the EtherCAT interface to control the youBot joints. This application is executed on a laptop and on a RaMstix.

The test with the laptop resulted in no periodic behavior and faster execution times with respect to the same application on the RaMstix. Periodicity is obtained on the RaMstix as every 15 cycles a peak is observed with an increase in latency of approximately 100μs.

The results of the measurements of the execution time of the EtherCAT interface in 20-sim 4C executed by Spil (2016) is shown in Figure 5.5.

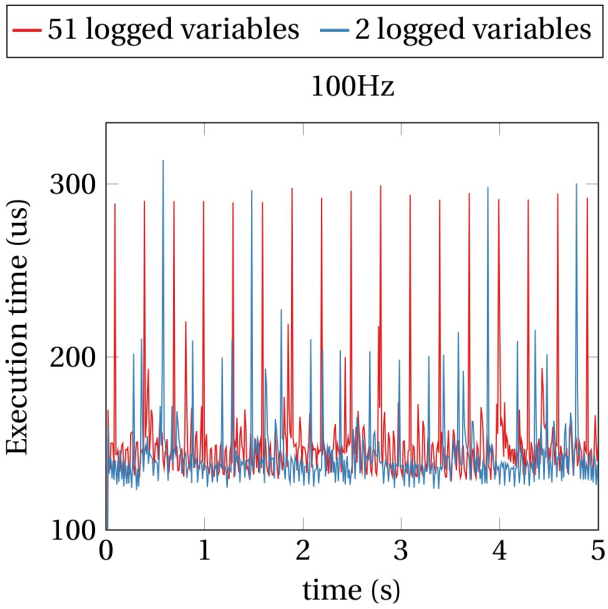


Figure 5.5: The execution time of the EtherCAT interface when executed with 20-sim 4C on a RaMstix (Spil, 2016).

In the execution time of the EtherCAT interface in 20-sim 4C is observed that the latency peaks are obtained every 30 cycles with an increase in latency of $50-100\mu s$.

5.3.3 Discussion

No periodicity is obtained when the application that interfaces with the EtherCAT interface is executed on a laptop. Therefore the periodicity of the EtherCAT SOEM master is caused by the RaMstix, where peaks are obtained every 15 cycles with heights of $100\mu s$. This application is not compiled for Xenomai and therefore the EtherCAT SOEM master introduces the periodic execution times even when no Xenomai kernel is used. The next chapter should conclude whether the same periodic behavior is obtained when the EtherCAT interface is used in a LUNA application, which uses Xenomai.

The 20-sim 4C implementation has latency peaks every 30 cycles which is interesting as this test observed a periodicity every 15 cycles. The height of the peaks however are comparable.

20-sim 4C utilizes the XMLRPC daemon to send variables to the monitor. The latency peaks at every 30 cycles imply that the XMLRPC daemon probably tries to update the variables every 30 cycles, resulting in periodic latency peaks every 30 cycles as the EtherCAT master and XMLRPC daemon both try to access the network.

The EtherCAT interface is compiled to a library resulting in an EtherCAT interface that can be used within LUNA to control the motors of the youBot. The interface provides real-time performance as the execution time is bounded, however some periodicity is obtained. It is chosen to not implement a component for EtherCAT as it is not generic enough and no EtherCAT interface with IgH is designed.

The next chapter investigates the execution time of the EtherCAT interface in LUNA such that a better comparison can be made between the EtherCAT interface executed in LUNA and in 20-sim 4C.

6 EtherCAT interface and Network Channel combined

6.1 Introduction

An integration of the EtherCAT interface and the *Network Channel* is discussed in this chapter. The EtherCAT interface is able to send PWM values to the motors of the youBot. To be able to actually control the youBot a controller is realized that converts values from an input device to PWM values for the motor joints on the youBot.

To show the functionality of the controller a setup is used consisting of a laptop connected via USB to the joystick and via Ethernet to the youBot using EtherCAT as is shown in Figure 6.1.

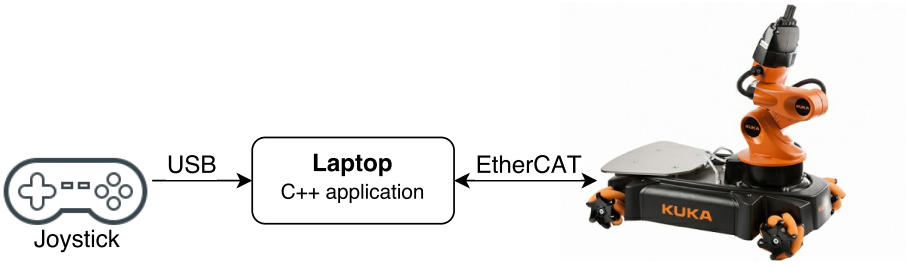


Figure 6.1: A laptop executes a C++ application that reads the joystick values, calculates the wheel PWM values using the controller and sends them to the EtherCAT interface to actual control the youBot.

After the joystick and controller are functioning a second setup is used to integrate the EtherCAT interface, controller and the *Network Channel* in a LUNA application on one RaMstix and the joystick and *Network Channel* in a LUNA application on the laptop as is shown in Figure 6.2. Measurements of the execution loop of the application on the RaMstix are performed to compare the execution times with the work of Spil (2016). This setup is also used for a demo as it demonstrates the *Network Channel*, controller and the EtherCAT interface in one setup. How to obtain the demo is described in Appendix D, and if building from source code is required, then Appendix E describes how to obtain the source code.

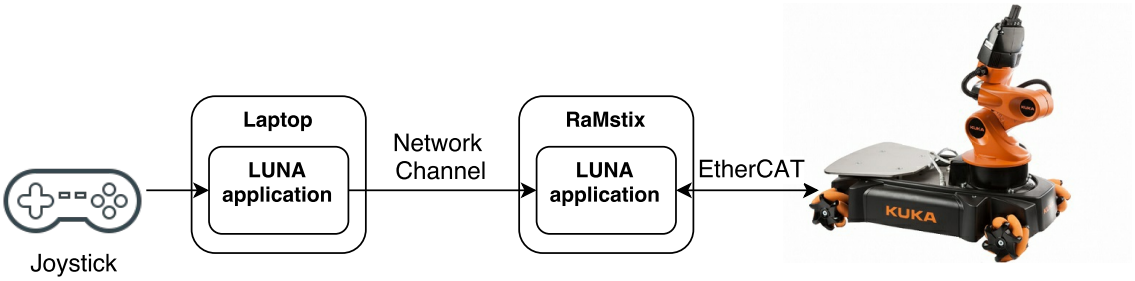


Figure 6.2: The laptop executes a LUNA application that reads the joystick values and sends them on the *Network Channel*. The RaMstix receives the values from the *Network Channel*, calculates PWM wheel velocities using the controller and sends them to the youBot using the EtherCAT interface.

6.2 Design

6.2.1 YouBot

A complete setup would consist of control for the complete youBot, but due to lack of time only the youBot base is implemented. The youBot base consists of four omni-directional wheels of which each wheel can be controlled separately by the EtherCAT interface provided in Chapter 5.

6.2.2 Input device

Several input devices can be used for the control of the youBot, for example:

- Geomagic Touch (Geomagic Touch, 2016): A 6 DOF haptic device.
- Omega 7 (Force Dimension, 2017): Also a 6 DOF haptic device.
- Xbox 360 controller: A controller used for playing games on the Xbox 360.

The Geomagic Touch and the Omega 7 are both haptic devices and deliver 6 DOF, which are very suitable for controlling a manipulator as they can also provide force feedback. Only the base of the youBot is controlled and therefore a Xbox controller suffices as input device.

The Xbox controller can be read out by using a library provided by Noakes (2017). The Xbox controller together with the axes and button numbers as they show up with the joystick interface is shown in Figure 6.3. Axis 0 and 1 are used to move the youBot in the xy plane and the Left Trigger (LT) and Right Trigger (RT) are used to rotate the youBot around its z axis.

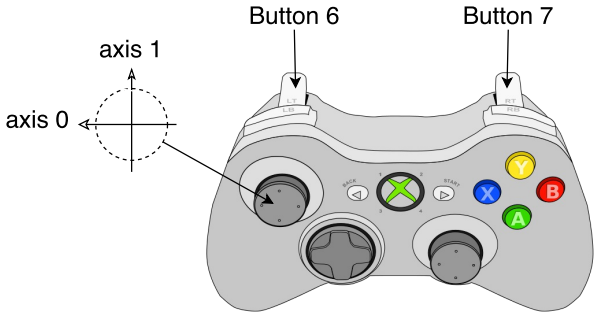


Figure 6.3: Xbox controller that is used for this setup with an overlay of axis and button numbers as they are used by the joystick interface. Button 6 and 7 are used for turning the youBot, and axis 0 and 1 for moving the youBot in the xy plane (Jishenaz, 2013).

6.2.3 Controller

The joystick interface delivers values from the axes and buttons that needs to be converted to wheel PWM values in a meaningful way to control the youBot.

It is chosen to use the function *cartesianVelocityToWheelVelocities* provided by the youBot API as a starting point. A new function called *joystickToMotorSpeed* is constructed in C++ that takes longitudinal (*long*), transversal (*trans*) and angular (*angle*) inputs between -1 and 1 and delivers wheel PWM values from -100 to 100.

The result is a controller that is able to move and rotate the youBot base using the values originating from a joystick.

6.2.4 Test design

First the controller and joystick are tested without the *Network Channel* and LUNA. This way it is ensured that the controller and joystick are functioning before they are implemented in the integration test. The test setup consists of a laptop running a C++ application that is reading joystick values, and places them onto the *setPWMBase* function of the EtherCAT interface. This setup is shown in Figure 6.4.

After the testing the controller with a C++ application two LUNA applications are constructed to show the integration of the *Network Channel*, EtherCAT interface and the controller. An overview of the setup is given in Figure 6.5. The joystick is connected to the laptop as the RaMstix does not provide support for the Xbox controller.

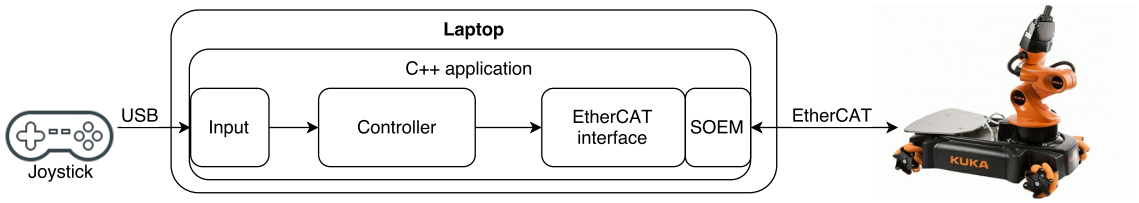


Figure 6.4: The C++ application executed on a laptop to check the functionality of the controller.

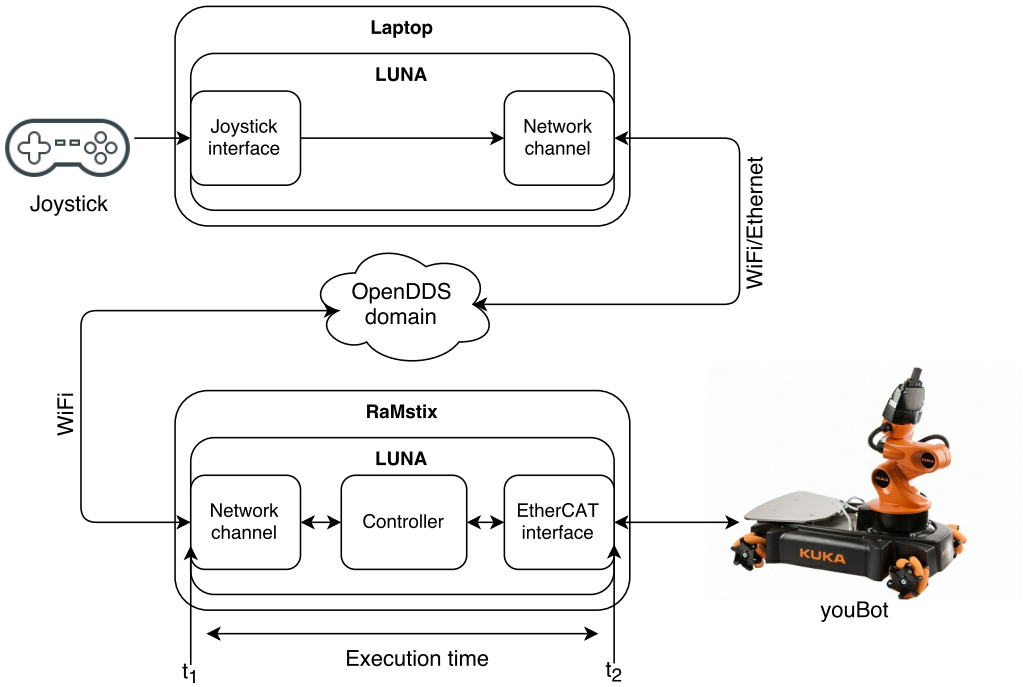


Figure 6.5: The setup that is used for the demo. The joystick is connected to the laptop and the youBot to a RaMstix. The RaMstix is communicating over WiFi with the laptop as the Ethernet port is used by the EtherCAT interface.

Note that the LUNA application at youBot side logs the time at t_1 and t_2. The difference between those timestamps is the execution time and is used to compare this implementation with the EtherCAT interface in 20-sim 4C from Spil (2016).

The LUNA application on the laptop is responsible for:

- Reading joystick axes and buttons.
- Sending these values to the *Network Channel*.

The LUNA application on the RaMstix has more tasks and is responsible for:

- Receiving joystick values from the *Network Channel*.
- Conditioning of joystick values.
- Check of the liveliness of the values and limit checking to add safety.
- Calculating the PWM values for the youBot base joints using the conditioned joystick values.
- Placing PWM values on the EtherCAT interface to actuate the youBot base motors.

6.3 Realization

6.3.1 Controller

It is chosen to use the function *cartesianVelocityToWheelVelocities* as a starting point. This function provides the base kinematics of the youBot. The inputs for this function originates from the joystick and consists of a longitudinal (*long*), transversal (*trans*) and angular (*angle*) value, all mapped between -1 and 1. With these inputs the wheel PWM values are calculated with a composition of these inputs as is shown in Equation 6.1. The omni-directional wheels with corresponding wheel numbers and the direction of the longitudinal, transversal and angle arguments is shown in Figure 6.6.

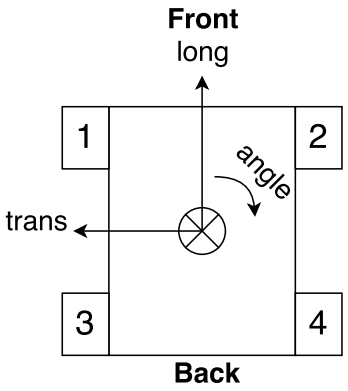


Figure 6.6: Top view of the youBot base with motor numbers and the direction of the longitudinal, transversal, and angle.

$$w_1 = -long + trans + angle \tag{6.1a}$$

$$w_2 = long + trans + angle \tag{6.1b}$$

$$w_3 = -long - trans + angle \tag{6.1c}$$

$$w_4 = long - trans + angle \tag{6.1d}$$

The result of Equation 6.1 represent the wheel ratios and not the PWM values that should have a value between -100 and 100. To obtain a PWM value in this range, first the wheel ratios are normalized by using the greatest absolute wheel value and dividing all wheel velocities by this factor. The normalization of the wheel velocities is shown in Equation 6.2.

$$w_{max} = max(abs(w_1, w_2, w_3, w_4)) \tag{6.2a}$$

$$w_{1,norm} = \frac{w_1}{w_{max}} \tag{6.2b}$$

$$w_{2,norm} = \frac{w_2}{w_{max}} \tag{6.2c}$$

$$w_{3,norm} = \frac{w_3}{w_{max}} \tag{6.2d}$$

$$w_{4,norm} = \frac{w_4}{w_{max}} \tag{6.2e}$$

From the input arguments the magnitude is calculated and is used to calculate the speed. The normalized wheel velocities are multiplied with this magnitude times 100 to obtain PWM values between -100 and 100. The calculation of the magnitude and the calculation of the resulting PWM values is shown in Equation 6.3.

$$speed = \sqrt{long^2 + trans^2 + angle^2} \times 100 \tag{6.3a}$$

$$w_{1,pwm} = w_{1,norm} * speed \tag{6.3b}$$

$$w_{2,pwm} = w_{2,norm} * speed \tag{6.3c}$$

$$w_{3,pwm} = w_{3,norm} * speed \tag{6.3d}$$

$$w_{4,pwm} = w_{4,norm} * speed \tag{6.3e}$$

The resulting PWM values are given to the *setPWMBase* function from the EtherCAT interface to actuate the joints of the youBot.

6.3.2 Controller and joystick test

To test the controller a C++ application is written that reads the joystick values using the joystick interface from Noakes (2017). These joystick values are scaled to represent values between -1 and 1 and is presented to the controller. The controller calculates the PWM values for the wheels and is provided to the EtherCAT interface to control the youBot. This controller test is shown using pseudo code and is shown in Figure 6.7.

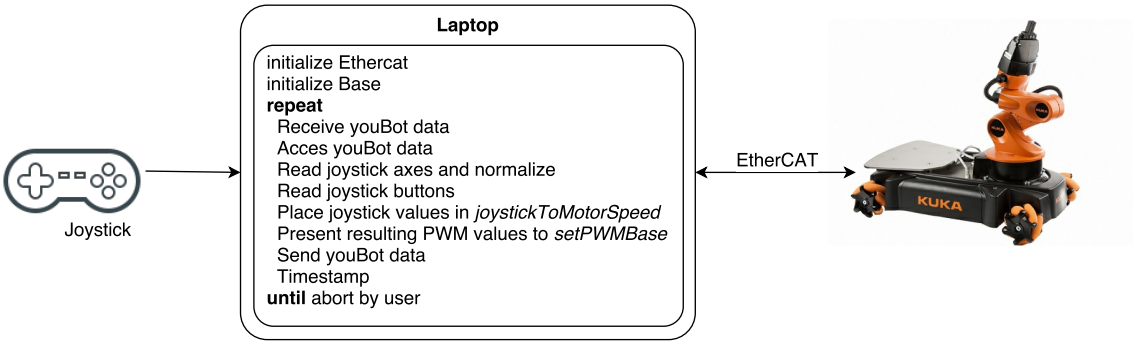


Figure 6.7: The C++ application presented as pseudo code that reads the joystick values, presents them to the controller and finally presents the resulting PWM values to the EtherCAT interface.

The objective of this test is to test if the joystick interface and controller work as expected.

6.3.3 Integration test

The setup presented in Figure 6.5 is realized.

The LUNA application running on the laptop is shown in Figure 6.8 as a TERRA model. The laptop is responsible for reading the joystick values and sending them over the *Network Channel* to the RaMstix.

The LUNA application consists of a timer that starts the sequential construct every time the timer expires. After the timer the *cpp_generate_direction* code block is entered and reads the joystick values, places those on variables that are being read by the parallel writers. The parallel

writers are connected to the *Network Channel* for communication with the RaMstix host with the youBot.

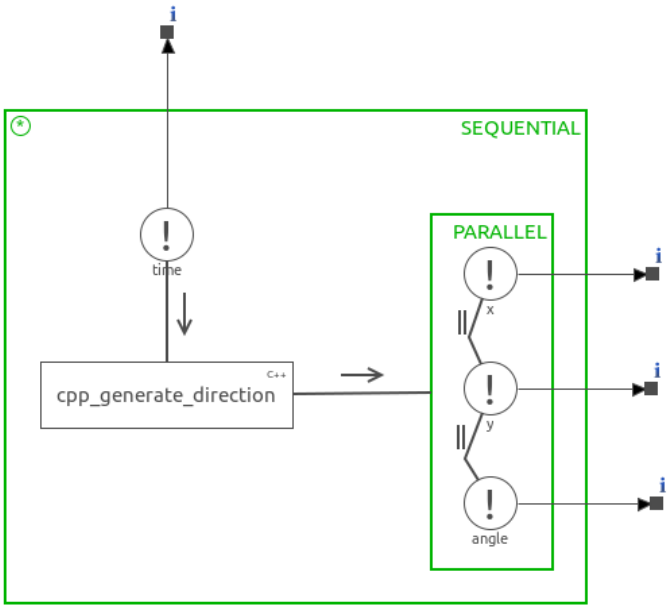


Figure 6.8: The LUNA application running on the laptop that reads the joystick values and sends the values to the network channel

At initialization of the *cpp_generate_direction* code block a joystick object is instantiated using the joystick interface from Noakes (2017). The code that is executed every iteration of the *cpp_generate_direction* code block is shown as pseudo code in Algorithm 3.

Algorithm 3 Pseudo code of the execute part of *cpp_generate_direction* C++ code block in TERRA.

```
1:  if there is no joystick found then
2:      print error message and block
3:  if There is a joystick event then
4:      if Event is button then
5:          if event button number is 7 AND button is pressed then
6:              set angular to -1
7:          else if event button number is 6 AND button is pressed then
8:              set angular to 1
9:          else if event button number is 7 OR event button number is 6 then
10:             set angular to 0
11:     if Event is axis then
12:         if event axis number is 0 then
13:             set x to event value
14:         else if event axis number is 1 then
15:             set y to event value
```

The values of the axes read from the joystick are signed 16 bit integers. The angle has a value of 1 (RT is pressed), -1 (LT is pressed) or 0 (non pressed), which indicates which direction the youBot should turn. The conditioning and safety check are performed by the controller on the RaMstix.

The LUNA application running on the RaMstix is shown in Figure 6.9 as a TERRA model. The RaMstix receives the joystick values from the *Network Channel*, provides a conversion and im-

plements safety before the values are provided to the EtherCAT interface to control the youBot base.

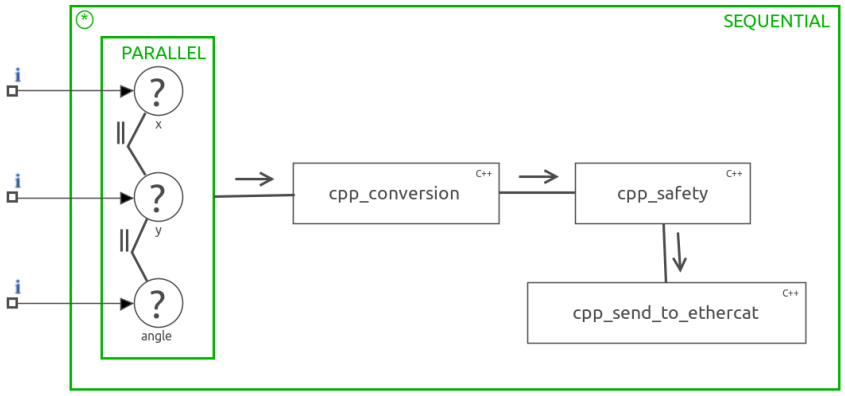


Figure 6.9: The LUNA application that receives the joystick values, converts them, adds safety and sends it to EtherCAT. Timestamps are performed at entering the *cpp_conversion* code block and at exiting the *cpp_send_send_to_ethercat* code block.

The LUNA application on the RaMstix has:

- *Parallel readers*: The parallel readers are responsible for receiving the values from the *Network Channel*. The received values were send on the *Network Channel* by the LUNA application on the laptop.
- *cpp_conversion*: The values are normalized to present a value between -1 and 1. Also a deadzone is implemented as the joystick axes are not zero when untouched.
- *cpp_safety*: Safety is added that makes sure the values received by the parallel readers are changing enough, such that the youBot stops whenever the values freeze for a certain amount of time.
- *cpp_send_to_ethercat*: Lastly PWM values are calculated using the controller. The values are written to the motor drivers of the youBot base by using the *setPWMBase* function in the EtherCAT interface.

After receiving the values from the network channel the *cpp_conversion* code block handles the conversion of raw joystick values to values between -1 and 1. The joystick sends values other than zero if the joystick is not touched, and therefore a deadzone is implemented. The deadzone implementation makes sure the youBot is not moving when values under a certain threshold are received. The angle is scaled down by a factor of two as this gave a rotation speed of the youBot base that is not too aggressive. The C++ code of the *cpp_conversion* code block is shown in Listing C.1 in Appendix C.

After the execution of the *cpp_conversion* code block the *angle_conv* has a value between -0.5 and 0.5 and the *x_conv* and *y_conv* have a value between -1 and 1. After the conversion the *cpp_safety* is entered and provides:

- *Liveliness check*: If the values *x_conv* and *y_conv* do not change for a certain amount of time the youBot stops moving.
- *Limit check*: A limit is provided for *angle_conv*, *x_conv* and *y_conv* to make sure the values do not exceed -1 and 1.

The code of the *cpp_safety* block is provided in Listing C.2 in Appendix C.

The code block after *cpp_safety* consists of the controller and uses the *setPWMBase* of the EtherCAT interface of the youBot. This code block is called *cpp_send_to_ethercat* and its functionality is shown in pseudo code in Algorithm 4. The PWM values of the motors are calculated using the conditioned values from the joystick. The calculated PWM values are send to the motor drivers of the youBot base with the EtherCAT interface.

Algorithm 4 Pseudo code of the execute part of *cpp_send_to_ethercat* C++ code block that utilizes the controller to calculate the motor PWM values and sends these PWM values to the EtherCAT interface to control the youBot base joints.

1: Initialize the joystick
2: Initialize EtherCAT
3: Initialize youBot Base
4: Call *joystickToMotorSpeed* with x, y and angular as arguments
5: Give resulting PWM values to *setPWMBase* of the EtherCAT interface

6.4 Tests

6.4.1 Controller and joystick test

The C++ application that combines the EtherCAT interface with the controller presented in this chapter is executed. The response between a change in input of the joystick and the actual moving of the base is fast. The youBot moves and turns as is commanded by the joystick. No significant latencies are observed with the eye and therefore this C++ application successfully showed the correct functioning of the EtherCAT interface with the controller.

6.4.2 Integration test

6.4.2.1 Results

The LUNA application that reads the joystick values and sends them on the *Network Channel* is executed on the laptop and sends the values to the RaMstix with a rate of 100 Hz.

The LUNA application that reads values from the *Network Channel*, performs conversion, adds safety, and sends it on the EtherCAT interface and is executed on the RaMstix.

Spil (2016) measured the execution time of his implementation, and to be able to compare results the same kind of measurement is performed in this setup. The difference between t_1 and t_2 as shown Figure 6.5 is the execution time and is used to compare these results with the measurements performed in Spil (2016).

The latency test of Spil (2016) is shown in Figure 6.10a and the latency test of this implementation is shown in Figure 6.10b.

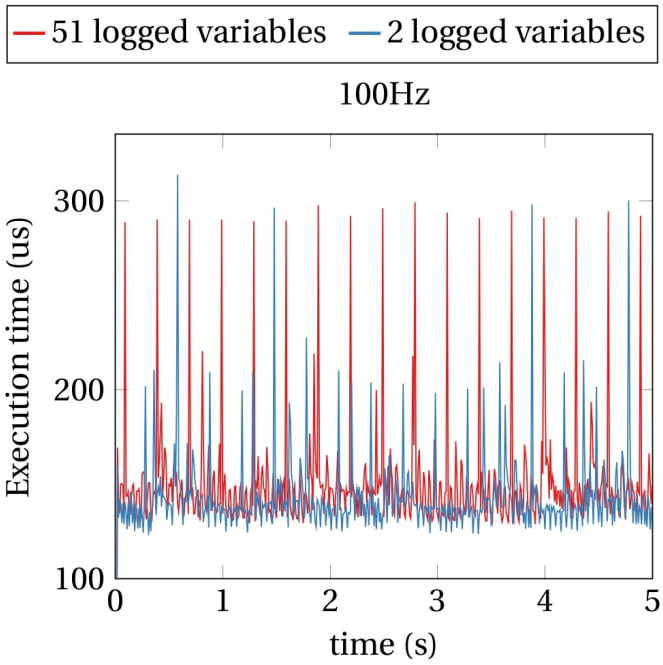
The peaks observed by the measurements in Figure 6.10a are clearly periodical as peaks are observed every 30 cycles. Peaks of equal height for 51 logged samples are observed with an height of $50 - 100\mu s$.

The execution times of this implementation as is shown in Figure 6.10b do not show such a clear periodical behavior, but latency peaks are observed of $100 - 400\mu s$. However some periodicity can be observed between 2.5 and 4.5 seconds as the peaks are separated by 15 cycles.

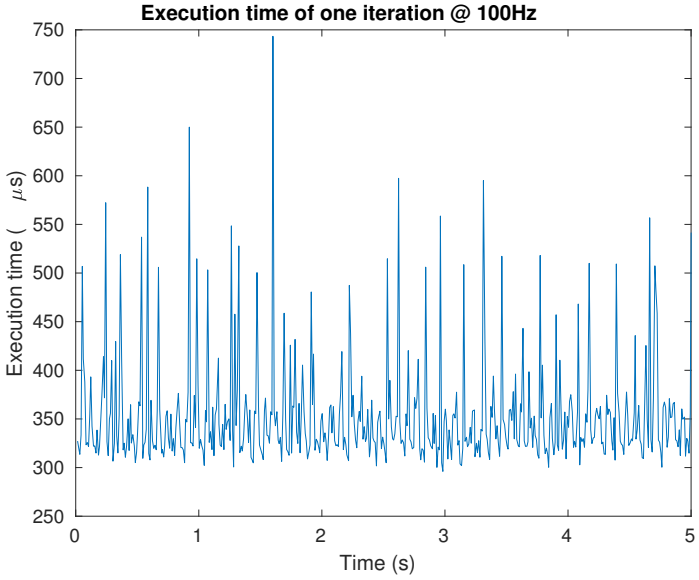
6.4.2.2 Discussion

In Chapter 5 the EtherCAT interface on the RaMstix is executed in a C++ application without LUNA. Latency peaks were observed every 15 cycles with peak heights of $100\mu s$.

This chapter introduced the EtherCAT interface in LUNA on the RaMstix. Again periodic behavior with latency peaks separated by 15 samples are observed, but with increased peak heights



(a) Execution time of the 20-sim 4C applications with the EtherCAT interface (Spil, 2016).



(b) Execution time of the LUNA application with the EtherCAT interface.

Figure 6.10: Execution times for the EtherCAT interface in 20-sim 4C (Figure 6.10a) and in LUNA (Figure 6.10b).

of $100 - 400\mu s$. The higher latencies in the peaks are probably caused by the switch from Xenomai to the linux network stack, as the network stack is not managed by Xenomai. Also the periodic behavior is not that clear anymore as the LUNA application itself also introduces varying latencies.

The periodicity is not as clear as is observed in the implementation of Spil (2016). Latency peaks separated by 15 cycles are observed in LUNA, where the peaks are separated by 30 cycles for the EtherCAT interface in 20-sim 4C.

It can be concluded that the EtherCAT interface on the RaMstix introduces periodic latency peaks every 15 samples. The EtherCAT interface in 20-sim 4C with the XMLRPC daemon that

sends variables to the 20-sim 4C monitor updates probably every 30 cycles resulting in latency peaks at every 30 cycles as both the EtherCAT master and the XMLRPC daemon try to access the network stack every 30 cycles.

The C++ application that uses the EtherCAT interface on the RaMstix without Xenomai still obtains periodic latencies. The statement made by Spil (2016) that the periodicity is caused by the network stack that is not managed by Xenomai is therefore not true. The height of these latency peaks however *can* be caused by using the network stack from Xenomai as the test *with* LUNA introduced higher latency peaks than the implementation *without* LUNA.

These tests show that the latency peaks are also visible when only the default linux scheduler is used on the RaMstix, so therefore the latencies are caused by the EtherCAT interface, where the height of the latencies can still be caused by the switch to the linux network stack from Xenomai.

6.5 Conclusion

The functioning of the controller and the joystick interface is shown and executed using a C++ application on the laptop. No latencies observed with the eye are observed.

Also the test that integrates the *Network Channel*, controller and joystick is executed and the result is a functioning demo showing the *Network Channel* that sends the joystick values from the laptop to the RaMstix, such that the youBot base can be moved.

The measurements in the integration tests show that also a periodicity is obtained with the EtherCAT interface in LUNA. However this periodicity is not the same as observed by Spil (2016). The peaks of the latencies for the execution time with LUNA are separated by 15 cycles, where the separation of the peaks in the work of Spil (2016) are separated by 30 cycles. Therefore the latency peaks observed by Spil (2016) are observed by using the EtherCAT interface and the XMLRPC daemon for communication variables to the 20-sim 4C monitor. The assumption that the network stack introduced the periodic behavior is not completely true as the EtherCAT interface on the RaMstix without Xenomai also shows periodic behavior.

Using the integration test it is shown that the *Network Channel* also functions over WiFi. Latencies are increased as wireless communication introduces more latencies. The inverse kinematics of Spil (2016) or a IPC controller are not implemented as there was no time and a more simpler controller only controlling the youBot base sufficed for this research.

7 Conclusion & Recommendations

7.1 Conclusion

The goals stated in the introduction are met as a *Network Channel* for LUNA is designed, an EtherCAT coupling in LUNA to control the youBot is implemented, and a demo is constructed to show the functioning of the *Network Channel*, the controller, and the EtherCAT interface.

TERRA is extended to add the *Network Channel* to the GUI, such that the user is able to use the *Network Channel* by placing a *DDS port* on the drawing pane of TERRA.

The *Network Channel* is realized by using OpenDDS as the communication protocol. An interface to OpenDDS is realized and is used by the *Network Channel* component in LUNA to add the functionality.

The *Network Channel* for LUNA is able to perform publish-subscribe and rendezvous communication. Bundling of multiple subtopics into one DDS topic is supported. To make sure a packet contains all the data of the registered writers a waiting functionality is implemented for rendezvous communication, which results in the unblocking of all readers in one iteration and results in a bounded execution loops at the subscriber side.

A demo is designed that reads joystick values on an Ubuntu machine, and uses the *Network Channel* to send these values to a RaMstix that moves the youBot base by using a controller and an EtherCAT interface using the SOEM EtherCAT master.

7.2 Recommendations

A *Network Channel* is realized in this project, but no HRT guarantees can be given, because OpenDDS and the network stack are not supported by Xenomai. Also Ethernet and WiFi have no hard real-time capabilities.

The next step for the *Network Channel* could involve introducing IPC as a layer above this *Network Channel* to make sure the total system is always stable, such that IPC together with this *Network Channel* implementation is safe to use in robotic setups. Even though the *Network Channel* has no hard real-time capabilities, the total system will always be stable due to the energy based approach of IPC. The youBot has torque control and therefore the youBot can be used as a demonstrator for the *Network Channel* together with IPC in a future project.

The functioning *Network Channel* is provided, but some recommendations can be made.

- *Explicit checking for topic names in data packages*: By explicitly checking the topics present in a data bundle it can be guaranteed that all data is available such that the readers at subscribing side unblock in one iteration, which effectively results in a higher communication speed.
- *Implement asynchronous writing and reading*: By implementing the callback functions to the OpenDDS interface and a write action to OpenDDS once all data is present there is no loop dependency anymore, resulting in a better performance as the latencies due to the loops are omitted.
- *Add support for network stack and OpenDDS in Xenomai*: Mode switches occur as OpenDDS uses not Xenomai supported functions, and also the network stack is not supported. If OpenDDS and the network stack are redesigned for Xenomai, faster writes to OpenDDS and less latency at the network stack can be obtained.
- *Resolve RTPS issue*: The RTPS endpoint discovery of OpenDDS fails on the RaMstix as the RaMstix is not always able to connect to multicast addresses. RTPS delivers discovery

without central broker, and therefore resolving this issue will result in a easier to scale *Network Channel*.

- *Investigate different QoS settings for OpenDDS*: right now only the *best effort* QoS setting for writer, listener and topic is used. OpenDDS provides a lot more QoS setting which may result in a better overall channel performance.
- *Add waiting parameter to TERRA*: As the waiting functionality for data was introduced later in the project it is not yet available as a parameter of the DDS port at TERRA level.
- *Add support for different data types*: Only double data values are supported by the data structure at this time and therefore every other type needs a cast to a double.
- *Use another platform than the RaMstix*: When multiple listeners are instantiated the CPU load is increased significantly. Also a write action to OpenDDS takes a significant amount of time. When using a more resource rich platform a better performance will be obtained.

A LUNA implementation realization

A.1 Component integration in LUNA

Following the generic file structure of a component a new folder is added with the name *dds-channels* to follow the naming convention of other components. Inside this folder a *makefile* is present that specifies which files are needed for the *dds-channel* component. OpenDDS is used and therefore LUNA needs the location of the OpenDDS binaries and headers. The location of these files is provided by the *setenv.sh* that is provided by OpenDDS. By running this script, environment variables are set that are used by the *makefile* files. In *rules.mk* (located in top directory of LUNA), it is checked if the environment variables are set as is shown in Figure A.1. The LUNA component shows up as an option to build with LUNA and is shown in Figure A.2.

```
ifeq ($(CONFIG_COMPONENT_dds-channels),y)
        ifeq ($(ACE_ROOT),)
                _NULL:=$(shell $(call ERROR_MESSAGE, \$$(ACE_ROOT) is not set. Please run setenv.sh to set this variable))
                $(error $$(ACE_ROOT) is not set. Please run setenv.sh to set this variable)
        endif
        TARGET_CFLAGS += -I $(ACE_ROOT)

        ifeq ($(TAO_ROOT),)
                _NULL:=$(shell $(call ERROR_MESSAGE, \$$(TAO_ROOT) is not set. Please run setenv.sh to set this variable))
                $(error $$(TAO_ROOT) is not set. Please run setenv.sh to set this variable)
        endif
        TARGET_CFLAGS += -I $(TAO_ROOT)

        ifeq ($(DDS_ROOT),)
                _NULL:=$(shell $(call ERROR_MESSAGE, \$$(DDS_ROOT) is not set. Please run setenv.sh to set this variable))
                $(error $$(DDS_ROOT) is not set. Please run setenv.sh to set this variable)
        endif
        TARGET_CFLAGS += -I $(DDS_ROOT)
endif
```

Figure A.1: Rules.mk checks whether or not the OpenDDS environments variables are set if the *dds-channels* is configured for building with LUNA.

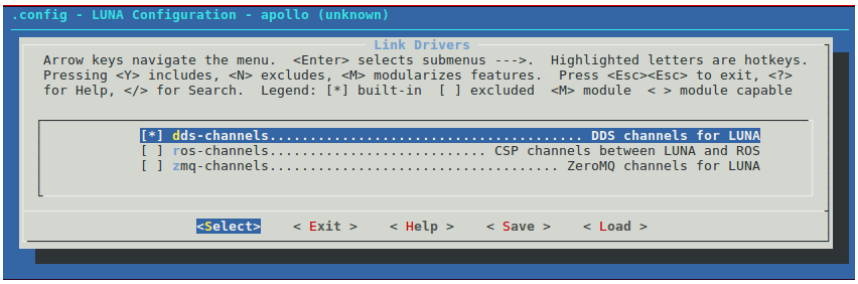


Figure A.2: The option to build the *dds-channels* with LUNA.

A.2 LUNA implementation

As shown in Figure 4.23 the software architecture for the *Network Channel* consists of three classes, and depending on the setup also a *Publisher* or *Subscriber* object is instantiated.

First the registering of a DDS port in the *DDSReactor* is treated and next the implementation of the *inboundLoop*, *outboundLoop* and *timeOutHandlerLoop* is treated.

Suppose a TERRA model is constructed that is responsible for sending data, and therefore the DDS ports at the sending side are configured to register publishers. Another TERRA model is constructed that is responsible for receiving the data, and therefore the DDS ports at the receiving side are configured to register subscribers. TERRA generates executable code from the models using the LUNA framework. When executing this code first the publishers and subscribers are instantiated before the communication is started.

The following sections describe the realization of:

- *Registering of a subscriber*
- *Registering of a publisher*
- *Publishing of data*
- *Subscribing for data*

Registering a subscriber
Two communication types are available for a DDS port configured as a subscriber:

- *Publish and subscribe communication*: The function *subInit* is called and instantiates a new OpenDDS subscriber with a datalistener if the *commonTopic* is not registered.
- *Rendezvous communication*: The function *registerReader* is called and instantiates a new OpenDDS subscriber with a datalistener and statewriter if the *commonTopic* is not registered.

The *subInit* and the *registerReader* function needs a *commonTopic*, *ownTopic*, *QoS* and a *callback* function as arguments. Every time *subInit* or *registerReader* is called from the *DDSChannel* it is checked if the *commonTopic* is already registered. If that is not the case than a new OpenDDS subscriber is instantiated and appended to a vector with the other OpenDDS subscribers. The provided *callback* function together with *ownTopic* is placed onto a callback vector. When data is available for a certain *ownTopic* the callback function can be extracted from the vector and can unblock the corresponding reader. The *subInit* or *registerReader* functions also start the inbound and time-out loop if they weren't started yet. The flow chart of how a subscriber is registered is shown in Figure A.3.

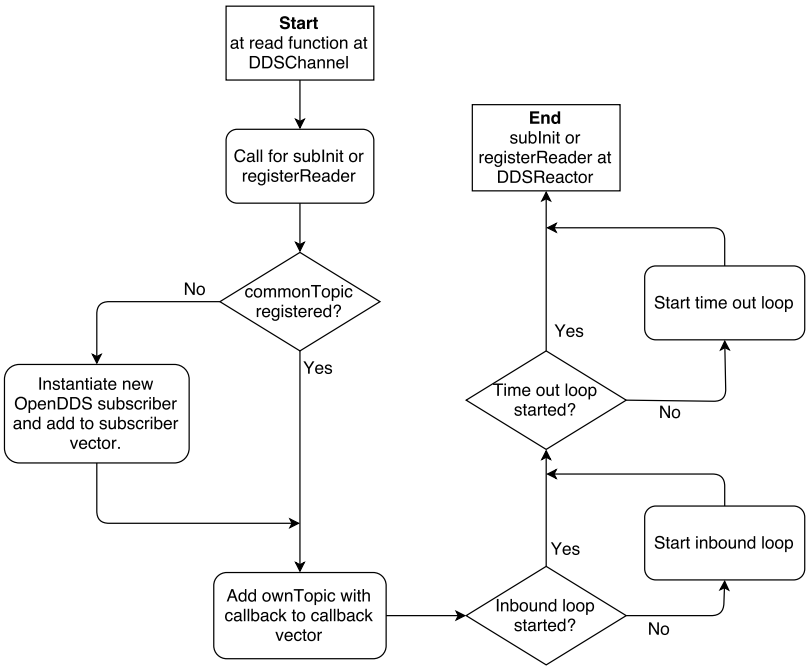


Figure A.3: A flowchart that shows how a DDS port configured for subscribing is registered in the *Network Channel* for rendezvous, and publish-subscribe communication.

Registering a publisher
Also two communication types for a DDS port configured as a publisher are available:

- *Publish and subscribe communication*: The function *pubInit* is called and instantiates a new OpenDDS publisher with datawriter if the *commonTopic* is not registered.

- *Rendezvous communication*: The function *registerWriter* is called and and instantiates a new OpenDDS publisher with datawriter and statelistener if the *commonTopic* is not registered.

The *pubInit* and *registerWriter* function both need a *commonTopic*, *ownTopic* and *QoS* as arguments. A new OpenDDS publisher is instantiated and added to the subscriber vector with other instances if that *commonTopic* was not registered. The *pubInit* needs no callback as the writer at *DDSChannel* has no blocking functionality, but the *registerWriter* function however needs a callback function as it blocks until the data is actually written to the OpenDDS publisher. The flow chart of how a publisher is registered is shown in Figure A.4.

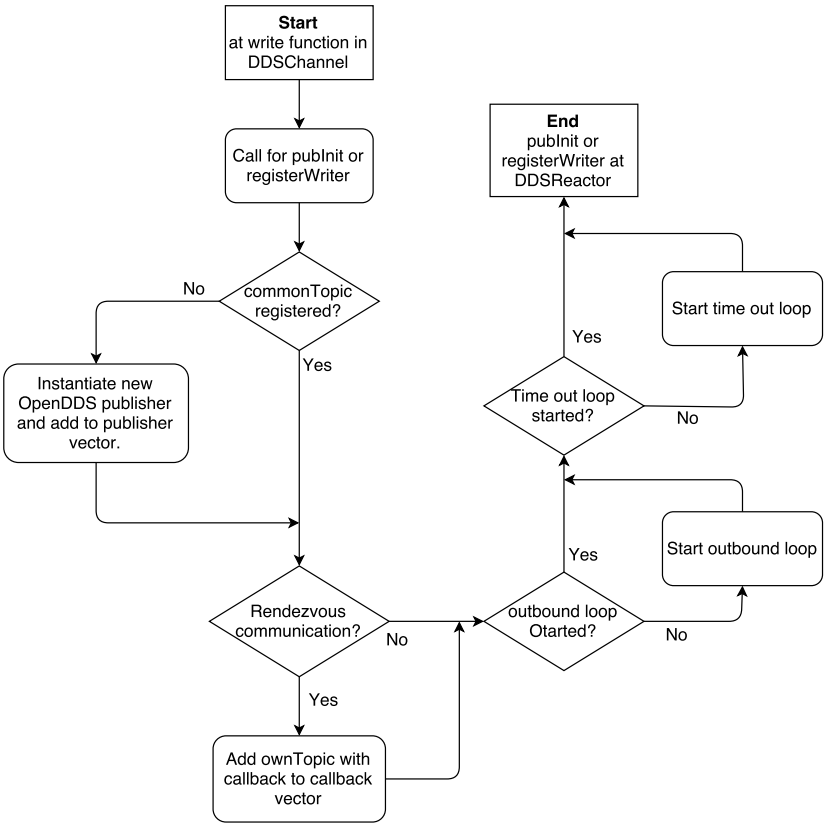


Figure A.4: A flowchart that shows how a DDS port configured for publishing is registered in the *Network Channel* for rendezvous, and publish-subscribe communication.

Publishing data

The publish of data to the *Network Channel* starts when the *write* function in *DDSChannel* provided by the *IChannelIn* interface is called by the rest of the LUNA application. In this *write* functions the *publish* function is called for publish-subscribe communication and the *writeRendez* function for rendezvous communication. Both writing functions need a *commonTopic*, *ownTopic* and a value as arguments. A flowchart of the implementation of the writer at *DDSChannel* is shown in Figure A.5. It is clear that the rendezvous write function *writeRendez* blocks after the call to write where the publish-subscribe write function continues after the call to write.

The *DDSReactor* receives all calls to enqueue a message from all the instantiated DDSChannels. To be able to handle all these calls and keep it thread safe the FIFO queues from Moodycamel (2014) are implemented. It is fast and easy to use as only one header file needs to be included. The arguments from the write functions at *DDSChannel* are placed in this queue and is dequeued by the *outboundLoop*.

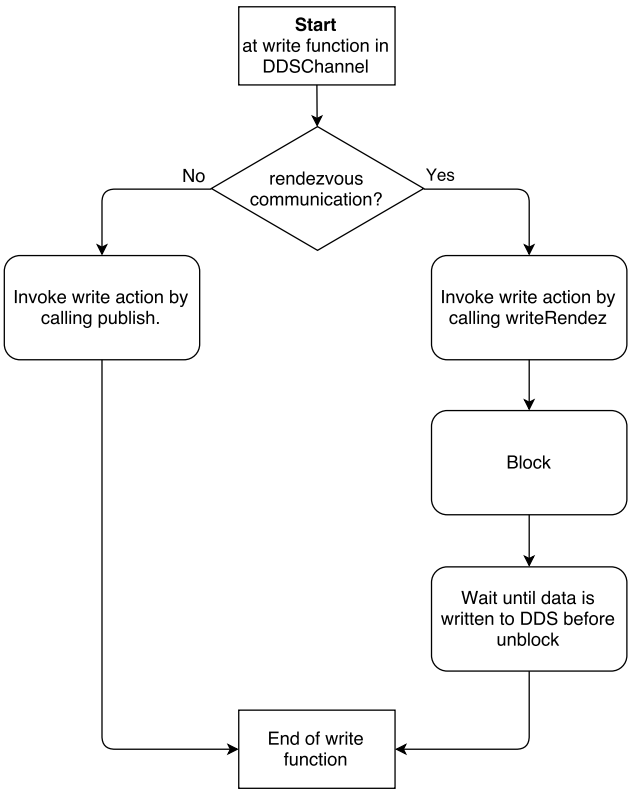


Figure A.5: A flowchart of publishing data in the *DDSChannel* for rendezvous and publish-subscribe communication.

The queue is emptied every iteration such that all the data in the queue is written to a DDS topic. The function in the outbound loop that handles the publish-subscribe publishers is called *handlePublishers* and the function for handling the rendezvous publishers is called *handleWriters*.

The difference between rendezvous and publish-subscribe is that for rendezvous communication an extra check is necessary to check if the corresponding reader is actually ready to receive.

Pseudo code for the *handlePublishers* function for publish-subscribe communication is shown in Algorithm 5 and the pseudo code for the *handleWriters* for the rendezvous communication is shown in Algorithm 6.

Algorithm 5 Pseudo code of the dequeueing of messages for publish-subscribe in the function *handlePublishers*.

```
1:  for every intance of publisher in publisherVector do
2:      while queue with messages is not empty do
3:          Dequeue a message
4:          if message commonTopic is equal to the publisher topic then
5:              add ownTopic to topicVector
6:              add value to dataVector
7:          else
8:              add message to reenqueueVector
9:      if there is data in dataVector then
10:         write data to write function of publisher
11:     if reenqueueVector is not empty then
12:         re-enqueue this vector in the message queue
```

Algorithm 6 Pseudo code of the function *handleWriters*. Dequeueing of messages for rendezvous communication without waiting.

```
1:  for every entry of publisher in publisherVector do
2:      if accompanying subscriber is ready then
3:          while queue with messages is not empty do
4:              Dequeue a message
5:              if message commonTopic is equal to the publisher topic then
6:                  add ownTopic to topicVector
7:                  add value to dataVector
8:                  for every entry in the callbackVector do
9:                      if callback topic is equal to the ownTopic of the message then
10:                         Add callback to callbackVector.
11:             else
12:                 add message to reenqueueVector.
13:         if there is data in dataVector then
14:             write data to write function of publisher
15:             Handle all callback functions in callbackVector.
16:         if size of dataVector is smaller than number of registered writers then
17:             Manually set the reader is reader to receive data.
18:         if re-enqueue vector is not empty then
19:             re-enqueue this vector in the message queue.
```

Note for the rendezvous communication that the readiness of the reader is manually set to ready (line 16 in Algorithm 6) if the *dataVector* was smaller than the amount of registered writers. This is necessary as not all the readers will unblock if no data is present for all the readers., and the writer will not write again because the *reader is ready* boolean is set to false after a write to OpenDDS.

The reset of a *reader is ready* boolean is not the most elegant way to solve this, and therefore a wait function is introduced. The wait function waits until all registered writers have presented their data. The *handleWriters* function with the wait function implemented is shown in Algorithm 7.

Note that with waiting the data is written to the OpenDDS publisher as soon as the vector with the data has the same length of the registered writers. The assumption is made that the data comes in bundles of all registered writers. This is an assumption that is validated in the test section of this chapter, Section 4.4.3. This waiting approach only works for one rendezvous topic, because the amount of *ownTopics* are counted and not explicitly checked for presence. By using multiple rendezvous topics the queue is certainly not ordered per *commonTopic* which will result in unwanted behavior.

Subscribing for data

The *read* function at *DDSChannel* provided by the *IChannelOut* interface receives the data via a callback function that places the data on a buffer. If data is already available on the buffer it is extracted and presented to the rest of the LUNA application, if no data is available than it will behave differently depending on the chosen communication method.

For rendezvous communication the *read* function calls the *notifyReaderReady* function to notify the corresponding subscriber it is ready to receive data before blocking. This call places a *reader is ready* message on a FIFO queue. Dequeueing of the *reader is ready* messages is performed by the *handleReaderReadyNotification* function in the inbound loop. Algorithm 8 shows how the messages are dequeued and written to the correct topic.

Algorithm 7 Pseudo code of the function *handleWriters* that dequeues the messages on the queue with waiting enabled. The writer is written to OpenDDS if the *dataVector* size is equal to the amount of registered writers.

```
1:  for every entry of publisher in publisherVector do
2:      if accompanying subscriber is ready then
3:          while queue with messages is not empty do
4:              Dequeue a message
5:              if message commonTopic is equal to the publisher topic then
6:                  append ownTopic to topicVector
7:                  append value to dataVector
8:                  for every entry in the callbackVector do
9:                      if callback topic is equal to the ownTopic of the message then
10:                         Add callback to callback vector.
11:             else
12:                 add message to reenqueueVector.
13:         if dataVector size is equal to number of registered writers then
14:             write data to write function of publisher
15:             Handle all callback functions in callbackVector.
16:             clear topicVector, dataVector and callbackVector.
17:         if re-enqueue vector is not empty then
18:             re-enqueue this vector in the message queue.
```

Algorithm 8 Pseudo code of dequeueing the messages in the *handleReaderReadyNotification* function for sending the reader ready notifications.

```
1:  for every entry of subscriber in the subscriberVector do
2:      while queue with notifyreaderready messages is not empty do
3:          Dequeue a message
4:          if message commonTopic is equal to the publisher topic then
5:              Increment the counter.
6:              if counter is equal to registered readers then
7:                  Call readerIsReadyToReceive on the subscriber.
8:          else
9:              add message to reenqueueVector.
10:     if reenqueueVector is not empty then
11:         re-enqueue this vector in the message queue.
```

The reader is unblocked as soon as data is received and the callback function is called at the *DDSReactor*.

For publish-subscribe communication the reader just blocks at entering the read function and unblocks whenever the callback function with data is called by the *DDSReactor*. The flow diagram of the *read* function in the *DDSChannel* is provided in Figure A.6.

The *buffersize* is a configurable unit that is set by the user at TERRA level. The *buffersize* states how many samples of data the buffer can hold before data is overwritten. The default size of the buffer is one. With a buffer size of one there is only place for one data value. If the callback function is called twice before the read action is entered one sample is lost as the second sample overwrites the first sample. Buffers can be used when two systems are connected that run on different frequencies, such that all the data is still available even when the receiving application is slower than the sending application.

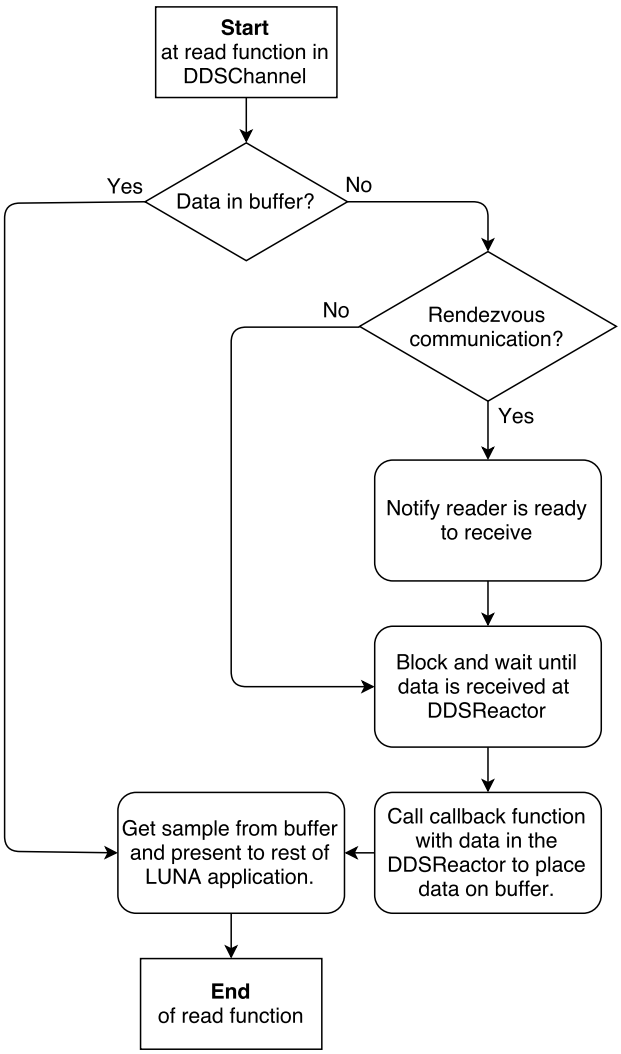


Figure A.6: A flowchart of the *read* function in *DDSChannel* for rendezvous and publish-subscribe communication.

The functions that receive the data are *handleSubscribers* and *handleReaders* for publish-subscribe and rendezvous communication respectively. The approach of both functions is the same and is shown in Algorithm 9. The functions dequeue a different *subscriberVector*.

Algorithm 9 Pseudo code of receiving data from OpenDDS to the *Network Channel* for the *handleSubscribers()* and *handleReader()* function.

```
1: for every entry of subscriber in subscriberVector do
2:     if the subscriber has received data then
3:         Read the data from the subscriber
4:         for every entry in the data do
5:             for every entry in the callbackVector do
6:                 if topic of data is equal to the topic of the callback entry then
7:                     Place the data entry in the correct callback
```

B LUNA implementation tests

B.1 Introduction

To validate the functionality of the *Network Channel* some tests are performed. The tests that are performed are:

- *Transport*: As DCPS is used for matching endpoints, TCP and UDP transports are supported. This test shows the differences between the transports using the setup described earlier.
- *Bundled data validation*: Waiting for complete packets must be validated, and therefore a test is performed that shows if waiting for messages results in complete packets with the data from all the registered writers.
- *Latency analysis*: It has been found that the implementation of van de Ridder (2017) is able to perform writes with less deviation. An analysis on the latency is performed to have a better insight in the latencies of the *Network Channel* using OpenDDS.
- *Stress test*: By introducing higher sending rates and extra traffic onto the network the *Network Channel* implementation is stress tested.

With these tests the performance can be evaluated, but also the limitations of this implementation of the *Network Channel*.

B.2 Test setup

The basic setup for most of the tests consists of two RaMstixes. One RaMstix is the publisher and the other one is the subscriber. A laptop is used to run the DCPS server to match endpoints. An overview of the test setup is given in Figure B.1.

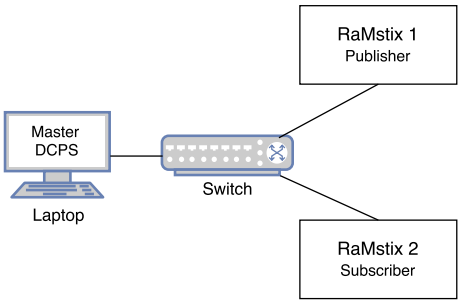
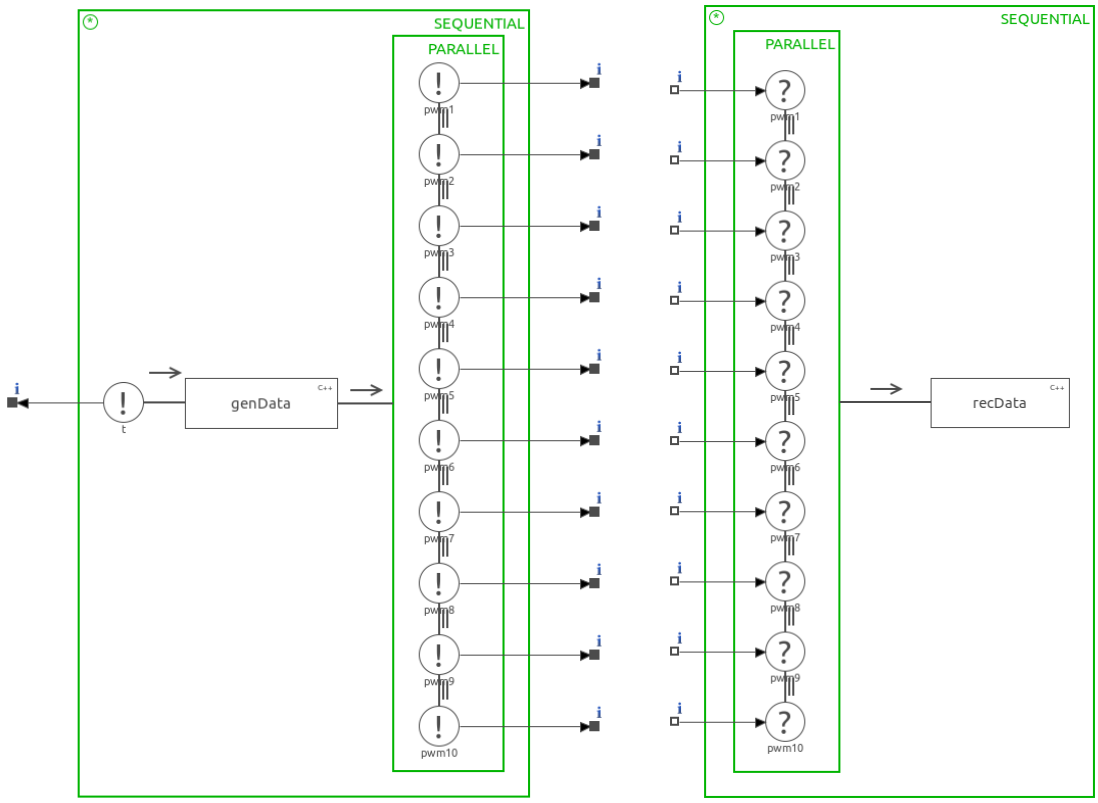


Figure B.1: The setup used for all the tests. The "Master" hosts the DCPS server, which is used for endpoint discovery. One of the RaMstixes executes the publisher code and the other the subscriber code.

To test the performance of the *Network Channel* an application with TERRA is drawn that consists of 10 DDS ports on each side. 10 channels were chosen as this gives clear differences between the communication types.

The readers and writers of these 10 channels are in a parallel construct to be able to test the waiting functionality as well. The IO-SEQ pattern is used, because it is probably also used in other LUNA applications as it reduces the risk of a deadlock. IO-SEQ means that first all the readers or writers are executed in parallel, before calculations are performed.

Figure B.2a shows the publishing side of the test application and Figure B.2b shows the subscribing side.



(a) The publisher side in TERRA. The ports on the right side connect to DDS ports at architecture level. The single port on the left is the timer port that enables the publisher to run at a given frequency. The *genData* code block generates sequential data for the writers and performs the timestamp after each data generation.

(b) The subscriber side in TERRA. The ports on the left side connect to the DDS ports at architecture level. the *recData* code block collects the received data from the reader and performs the timestamp.

Figure B.2: The TERRA test application that is used for publishing and subscribing to 10 channels for publish-subscribe and rendezvous communication.

The *recData* and *genData* code blocks perform timestamps. The *genData* codeblock generates sequential data and logs the current time directly after generating the data. The *recData* codeblock performs a timestamp as soon as it enters the codeblock (directly after the unblocking of all readers). It is chosen to measure the time between two write and two read iterations as a measure of performance as it does not need a synchronized clock between the two systems.

The inbound and outbound loop are configured to run at 400Hz as empirical testing shows that a loop with this frequency does not introduces a too high load on the CPU whilst still able to dequeue the queues often enough. The sending rate of the publisher is chosen to be 100Hz as it is multiple of the loop frequency and shows distinct differences between publish-subscribe and rendezvous communication. These parameters hold for all tests in the next section unless mentioned otherwise.

DCPS and RTPS are the two implementations for endpoint detection in OpenDDS. RTPS requires no central broker and DCPS hosts a service that DDS clients connect to. RTPS uses multicast and for some reason it fails to connect to multicast addresses from time to time and therefore DCPS is used for these tests. DCPS supports TCP and UDP as transport mechanisms.

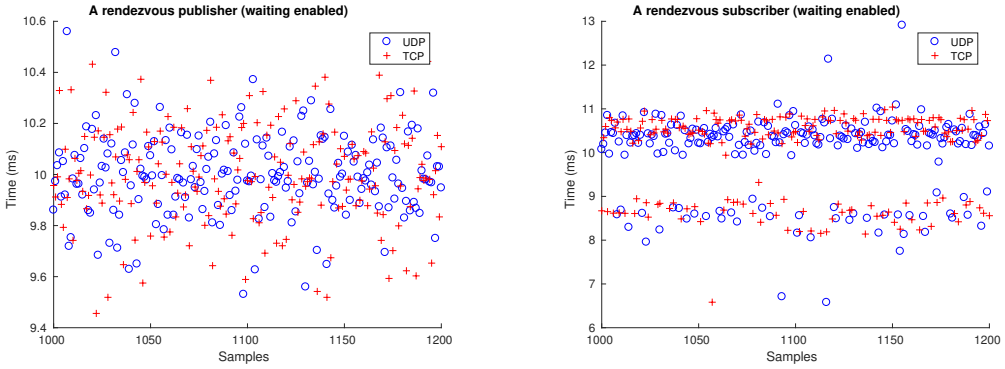
B.3 Tests

B.3.1 Transport

DCPS supports TCP and UDP as transport mechanisms. UDP delivers no guarantees in terms of reliability, where TCP offers reliability with: flow control, re-transmission of lost packets by negative acknowledgements and has duplicate free transmission due to a point to point connection (Kurose and Ross, 2012). UDP is used when losing packets is not disastrous for the application (video, telephone). UDP is usually faster as it does not have to provide reliable transport. TCP is mostly used in applications where all the data must be available after transfer (file exchange).

The test setup proposed at the beginning of this chapter with ten channels is used and executed for rendezvous, and publish-subscribe communication.

For the publish-subscribe communication no differences where obtained and therefore only the results of the rendezvous communication with waiting enabled is shown in Figure B.3.



(a) The time between two timestamps for a rendezvous publisher sending with different transports (UDP and TCP).

(b) The time between two timestamps for a rendezvous subscriber using different transports (UDP and TCP).

Figure B.3: The test that shows the performance of waiting versus no waiting for sending data at 100Hz.

With UDP transport it is observed that the readers do not always unblock in one iteration, which is indicated by a execution time greater than 10ms ($\approx 12ms$ and at $\approx 13ms$), which must have been caused by losing a packet as no such outliers are observed for TCP transport.

Bounded execution times at the subscriber are observed when using TCP transport. UDP transport introduces packet loss and therefore the execution times vary as not all readers are unblocked in one iteration. In Section B.3.3, it is shown that there is almost no difference in writing time with TCP versus UDP transport. Therefore it is recommended to use TCP over UDP.

B.3.2 Bundled data validation

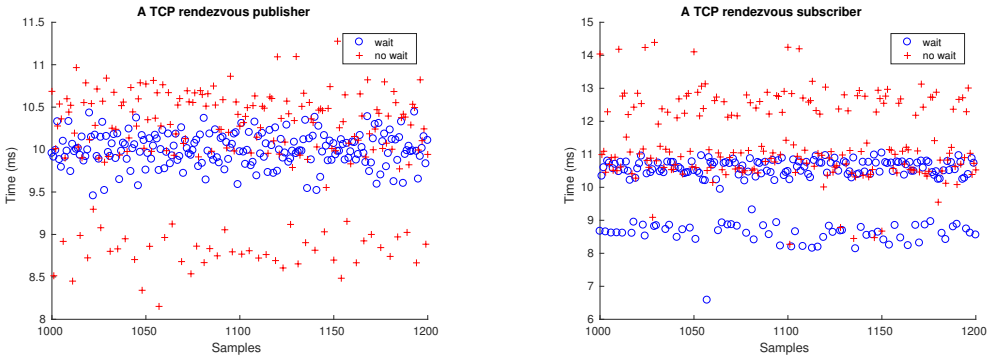
In Section 4.2, it is shown that bundling of data is probably more efficient as only one listener has to be asked if data is available for a *commonTopic*. Also preliminary tests of OpenDDS on the RaMstix showed that multiple listeners resulted in a fairly high CPU load and also bad performance.

When publish-subscribe communication is used the publisher will publish whatever is on the queue at the moment of dequeueing, but for rendezvous communication two implementations are available:

- Publish whatever is on the queue. This follows the same method as publish-subscribe communication.

- Wait for all registered writers to have presented their data. When waiting for the availability of all data the writers and readers can be unblocked at once.

The result of waiting versus not waiting is shown in Figure B.4.



(a) The rendezvous publisher that publishes the messages with and without waiting for all the writers to have presented their data.

(b) The rendezvous subscriber receives the data. The difference of waiting versus no waiting at the writing side is shown.

Figure B.4: The test that shows the performance of waiting versus no waiting for sending data at 100Hz with rendezvous communication.

At the publisher side (Figure B.4a), it is observed that waiting for a full packet results unblocking of the writers constantly around 10ms, without unblocking in an extra iteration. When waiting is not enabled, the data sequences may not contain all the data. Resulting in a mismatch between sending *reader is ready* messages and unblocking of the readers and writers resulting in unwanted behavior.

From the measurements at the subscriber side in Figure B.4b, it is observed that all readers are unblocked within 10ms for the waiting functionality as no data points are present above this line. The small offset from 10ms is caused by the *inboundLoop* and *outboundLoop* on the different systems being not in sync.

Also some samples 2.5ms below the 10ms line are observed and are caused by the unblocking of the readers one iteration earlier than normal. It may happen that the values are published one iteration earlier to OpenDDS if the *readerIsReady* message was already received, and the messages were already on the queue. The result is the unblocking of the readers one iteration earlier.

When the waiting functionality is not enabled the bundle does not necessarily contains the data for all the readers and therefore it may take up to three iterations before the readers are unblocked (data points at 14ms).

Rendezvous communication with waiting functionality seems to work correctly, but needs a validation. The validation consists of sending 100 000 data bundles to the subscriber at a 100Hz sending rate. The publisher saves the *ownTopics* that were present in a packet. A snapshot of this file is shown in Figure B.5, every line represents a packet that is written to OpenDDS.

The complete file is analysed with MATLAB to check if all the *ownTopics* are available for every package. The analysis shows that all the packages contain all the data of the registered writers, which means that the waiting functionality performs without problems.

However, the limitation of this implementation is that only one rendezvous wait publisher is supported on one host. With more rendezvous wait publishers the queue will not be ordered anymore and this will lead to unwanted behavior. Actively comparing *ownTopic* for every *com-*

```
pwm10 : pwm6 : pwm2 : pwm1 : pwm9 : pwm8 : pwm7 : pwm5 : pwm4 : pwm3 :
pwm10 : pwm6 : pwm2 : pwm1 : pwm9 : pwm8 : pwm7 : pwm5 : pwm4 : pwm3 :
pwm10 : pwm6 : pwm2 : pwm1 : pwm9 : pwm8 : pwm7 : pwm5 : pwm4 : pwm3 :
pwm10 : pwm6 : pwm2 : pwm1 : pwm9 : pwm8 : pwm7 : pwm5 : pwm4 : pwm3 :
pwm5 : pwm4 : pwm3 : pwm10 : pwm6 : pwm2 : pwm1 : pwm9 : pwm8 : pwm7 :
pwm1 : pwm9 : pwm8 : pwm7 : pwm5 : pwm4 : pwm3 : pwm10 : pwm6 : pwm2 :
pwm10 : pwm6 : pwm2 : pwm1 : pwm9 : pwm8 : pwm7 : pwm5 : pwm4 : pwm3 :
pwm10 : pwm6 : pwm2 : pwm1 : pwm9 : pwm8 : pwm7 : pwm5 : pwm4 : pwm3 :
pwm4 : pwm3 : pwm10 : pwm6 : pwm2 : pwm1 : pwm9 : pwm8 : pwm7 : pwm5 :
pwm2 : pwm1 : pwm9 : pwm8 : pwm7 : pwm5 : pwm4 : pwm3 : pwm10 : pwm6 :
pwm10 : pwm6 : pwm2 : pwm1 : pwm9 : pwm8 : pwm7 : pwm5 : pwm4 : pwm3 :
pwm10 : pwm6 : pwm2 : pwm1 : pwm9 : pwm8 : pwm7 : pwm5 : pwm4 : pwm3 :
pwm10 : pwm6 : pwm2 : pwm1 : pwm9 : pwm8 : pwm7 : pwm5 : pwm4 : pwm3 :
```

Figure B.5: A snapshot of the file that is produced at the publisher. Every line is a bundle containing the subtopics that has been written to an OpenDDS publisher. Note that the bundles are not ordered as the writers are in a parallel construct.

monTopic with what is registered solves this problem, but is not implemented. Also the assumption is made that the the messages for one *commonTopic* is ordered in the queue.

The wait function is also implemented for the publish-subscribe communication, but is less crucial as it needs no synchrony between writer and reader. A slightly different approach was taken resulting in probably an off-by-one error, but is nevertheless shown in Appendix I.

So when using rendezvous communication between two hosts for one *commonTopic* the best practice is to use the wait function as it results in:

- Less deviation at writing side due to less writes to the OpenDDS publisher, which also results in the unblock of all writers in one iteration.
- Better performance at the subscriber side due to the unblocking of all readers in one iteration. Also one listener needs to be instantiated, resulting in a lower CPU load.

B.3.3 Latency analysis

When comparing the *Network Channel* with a basic implementation of ZeroMQ from van de Ridder (2017), it is obtained that the OpenDDS publisher has a lot more deviation for writing at 100Hz. The differences are clearly visible as is shown in Figure B.6.

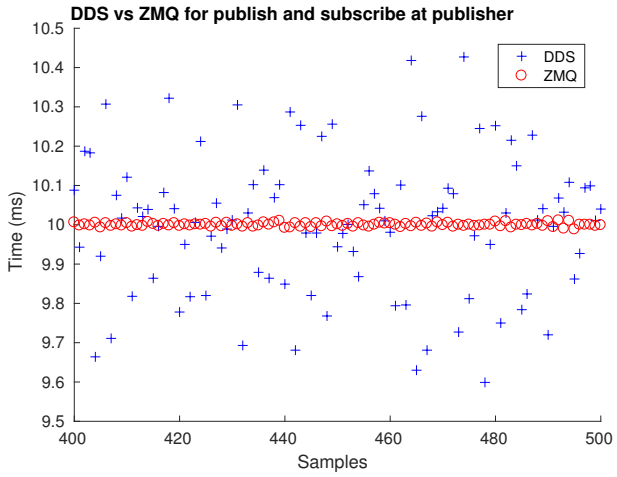
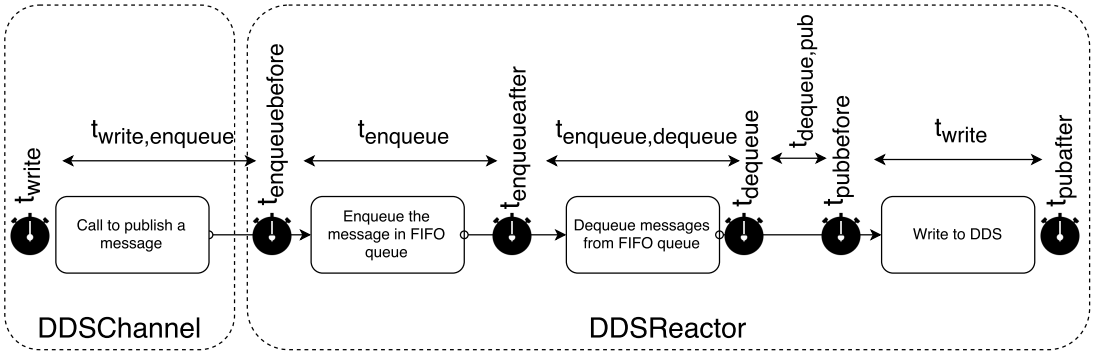


Figure B.6: A comparison between of the ZeroMQ implementation by van de Ridder (2017) and the OpenDDS implementation of this project for publish-subscribe communication with 5 channels.

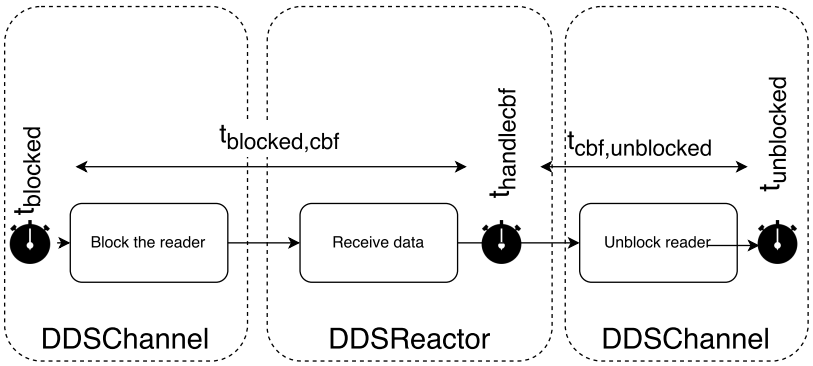
However the accumulated latencies at the writer do not exceed 10ms as can be seen from the write at every 10ms a fairly big standard deviation is obtained around this 10ms line.

The only difference is the protocol that is used, and therefore a write action to OpenDDS adds a lot more overhead than a write to ZeroMQ. ZeroMQ wraps only the sockets, where OpenDDS provides a lot of QoS settings, which support this statement.

From Figure B.6 is concluded that writing to OpenDDS probably is an expensive operation and therefore a latency analysis on the *Network Channel* implementation is performed. Timestamps log the time at important places of the *Network Channel* to measure the time of a certain action. The place of the timestamps for publish-subscribe communication is shown in Figure B.7. This analysis is also performed for rendezvous communication and is shown in Appendix J as the average latencies are in the same order of magnitude.



(a) The location of the timestamps for the publisher.



(b) The location of the timestamps for the subscriber.

Figure B.7: The place of the timestamps for analysing the latencies for the publish and subscribe communication. The stopwatches indicate a timestamp and the arrows between timestamps indicate the difference between timestamps.

The test with 10 channels is used and executed with TCP and UDP transport to also investigate the differences between the transports. Ten channels are used for the test, but only the latencies experienced by one *ownTopic* is investigated. This way the latencies experienced by one *ownTopic* can be investigated for a 10 channel setup.

The average latencies and the standard deviations of these latencies for UDP and TCP transport is shown in Table B.1.

From Table B.1, it is clear that writing to OpenDDS (t_{write}) is indeed an expensive operation on the RaMstix. To investigate if this latency is introduced by the chosen thread implementation

Table B.1: Latencies calculated over 2000 samples with a sending rate of 100Hz for the publisher with publish-subscribe communication. The labels correspond to the difference in timestamps as shown in Figure B.7a.

	Latencies at the publisher (ms)			
Δt	**TCP**		**UDP**	
	Avg	Std	Avg	Std
$t_{write,enqueue}$	0.0307	0.0025	0.0322	0.0026
$t_{enqueue}$	0.0084	0.0011	0.0087	0.0013
$t_{enqueue,dequeue}$	1.1051	0.5838	1.0366	0.5226
$t_{dequeue,pub}$	0.093	0.0205	0.1114	0.0198
t_{write}	1.3413	0.2260	1.2876	0.2098

this test is repeated for different thread configurations for the *outboundLoop* and *inboundLoop*. The configurations that are tested, including the default, are:

- *Default*: The publisher is executed on the RaMstix with the threads scheduled by the FIFO scheduler at priority 88.
- *Threads OTHER*: The publisher is executed on the RaMstix with the threads scheduled with the default Linux scheduler (OTHER).
- *Laptop*: The publisher is executed on the laptop and uses therefore the default Linux scheduler (OTHER).

The average time and standard deviation of the write action to OpenDDS for all five configuration is shown in Figure B.8.

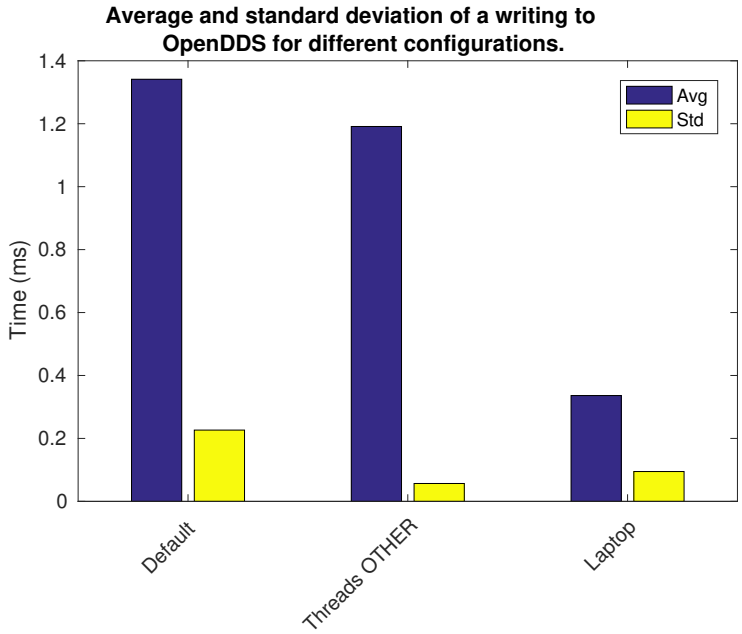


Figure B.8: Average time and standard deviation over 1600 samples for three different configurations for writing to OpenDDS on a TCP transport. The left bar presents the average value and the right bar the standard deviation. The writing times for UDP are comparable.

The configuration with the laptop has the lowest writing latency as it has the most resources. For the RaMstix implementations the threads running with the OTHER scheduler have lower

average writing times and a lower standard deviation with respect to the RaMstix implementation running with real-time threads.

By investigating the *Network Channel* with spurious relaxes (Xenomai, 2014), it is obtained that the write to OpenDDS contains a *gettimeofday()* function that causes a system call which is not handled by the Xenomai kernel, and therefore a mode switch occurs (Xenomai, 2014). Mode switches are unwanted when short and bounded response times are required. Also the network stack is not handled by Xenomai and results in mode switches as well. By scheduling the threads by the OTHER scheduler no mode switches occur as the call originates from a non real-time thread, such that a somewhat lower average writing time and a lower standard deviation is obtained.

Although the average latency and standard deviation are lower for OTHER threads, the overall performance drops drastically as it is not possible anymore to perform rendezvous communication at 100Hz. Probably the OTHER thread gets rescheduled by the higher priority threads, which causes slower loop times as is shown in Figure B.9 that shows a rendezvous writer.

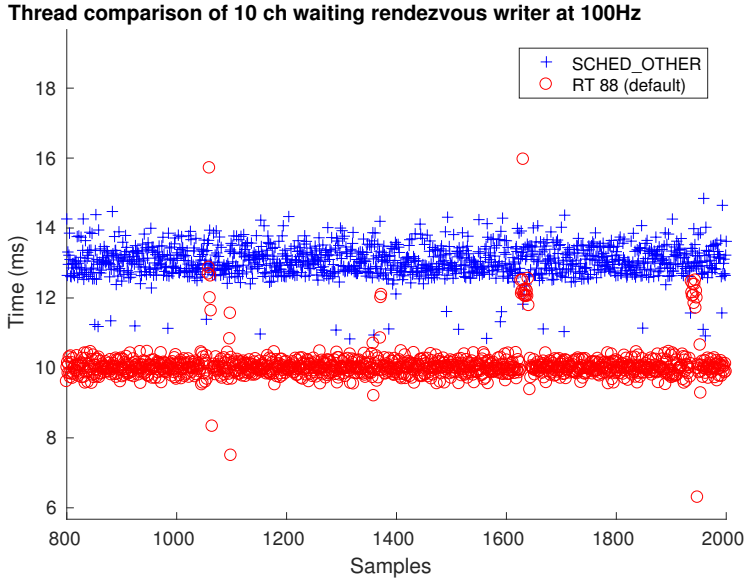


Figure B.9: A rendezvous writer with two different thread implementations, both trying to write at 100Hz.

The default scheduling policy results in the best performance although the writing time to OpenDDS is somewhat slower than with the OTHER scheduler. The OTHER scheduler introduces a bad overall performance as the OTHER scheduled threads are getting rescheduled by higher priority threads.

Therefore the default scheduling policy is used for the *Network Channel*.

This latency for writing to OpenDDS is a limiting factor for the maximum reachable sending speed. The other significant latency at the writer side is caused by dequeueing the message ($t_{enqueue,dequeue}$) and is caused by the following:

1. A message is added asynchronously to the queue and is dequeued by a loop that runs at 400Hz. Therefore a variable delay is obtained between 0 and 2.5ms.

2. The function that dequeues the message groups all the messages with the same *commonTopic*. And depending on how much messages of the same *commonTopic* are available this may take several dequeue operations.

The result is an average dequeue time that lies around 1ms. There is no significant difference between UDP and TCP in terms of average latencies. there is also no significant difference in the standard deviation as well: 0.2098ms for UDP and 0.2260ms for TCP transport.

The latencies for the subscriber are shown in Table B.2. The reader blocks whenever the LUNA application is ready to read the data. Unblocking is performed when data is received and the callback function is called. The time between being blocked and calling the callback function ($t_{blocked,cbf}$) takes the longest as the readers block after receiving data and get unblocked after receiving data again. If data was send with 10ms separation by the publisher a 10ms blocking time is expected. However, the actual latency is somewhat shorter, because with 10 readers in a parallel construct the reader with the analysed *ownTopic* could have been the last to block and the first to unblock.

Table B.2: Average latencies over 2000 samples at the subscriber for publish-subscribe communication. The labels correspond to the difference in timestamps shown in Figure B.7b.

	Latencies at the subscriber (ms)			
Δt	**TCP**		**UDP**	
	Avg	Std	Avg	Std
$t_{blocked,cbf}$	8.2921	1.3780	8.4870	1.2314
$t_{cbf,unblocked}$	0.9214	1.0893	0.8144	1.0333

The time from the callback function to the actual unblocking ($t_{cbf,unblocked}$) is higher then expected. The only thing between those two timestamps is placing the value in a buffer and unblocking the reader. The time it takes to unblock the reader after calling a callback function $t_{cbf,unblocked}$ is shown in Figure B.10. From this figure can be seen that several bands are present separated by 2ms from each other. Most of the samples are located around 0.40ms, which is the minimum time between the callback function and unblocking.

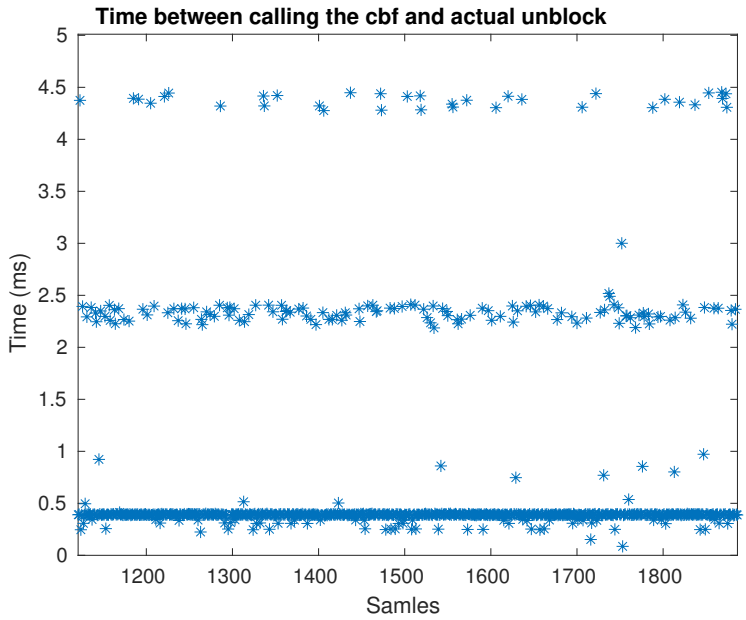


Figure B.10: The difference between the timestamp just after calling the callbackfunction and the actual unblocking of the reader in the *DDSChannel*.

These bands are not clearly linked to an event and therefore hard to analyse. The RaMstix is probably busy with other processes with higher priority threads such that the threads that unblock the reader are rescheduled. The same behavior is also obtained in the *handleWriter*

function for rendezvous communication. These applications are running on a RaMstix and therefore not a lot of resources are available, which makes it plausible that this behavior is caused by a lack of resources.

This latency analysis shows that it indeed takes a significant amount of time to write to OpenDDS. This latency is not solvable as it is an OpenDDS implementation, but a queue at the OpenDDS interface can be implemented that places the messages in another queue. The disadvantage of this approach is that yet another loop is implemented that introduces another delay. The actual writing time to OpenDDS is not increased and an extra latency is introduced, so therefore this is not implemented.

The *Network Channel* using OpenDDS is functioning on the RaMstix, but better performance will be obtained when a more resource rich system is used as is indicated with the laptop test. Most SBCs nowadays have a multi-core processor running on a higher clock frequency than the RaMstix, which will probably result in better performance. Also porting OpenDDS and the network stack to Xenomai will result in a better performance as it will result in no mode switches.

B.3.4 Stress tests

The first test adds extra traffic to the network while keeping the sending rate at 100Hz. The second test increases the sending frequency to check what the behavior of the *Network Channel* is at higher sending rates.

Test 1: Load test
The load test introduces extra traffic on the network by using *Iperf* (Iperf, 2017) to show the influence of extra traffic on the network for the performance of the *Network Channel*. 10 channels are used with a sending rate of 100Hz. The influence of extra traffic on the network is investigated and therefore the sending rate is kept constant.

One device hosts a *Iperf* server and another device the *Iperf* client. The client is sending data to the server using UDP at a rate that is configurable by the user.

Preliminary runs of *Iperf* indicated that the maximum amount that can be add to the network is around $67Mbps$. Therefore for this test loads of $25Mbps$ and $50Mbps$ are introduced. A Raspberry Pi 3 (RPI) is used for this test as it has a $100Mbps$ Ethernet connection. The laptop is only used for the DCPS server and not for *Iperf* as the gigabit Ethernet introduced extra packet loss with *Iperf*. The setup of this test is shown in Figure B.11.

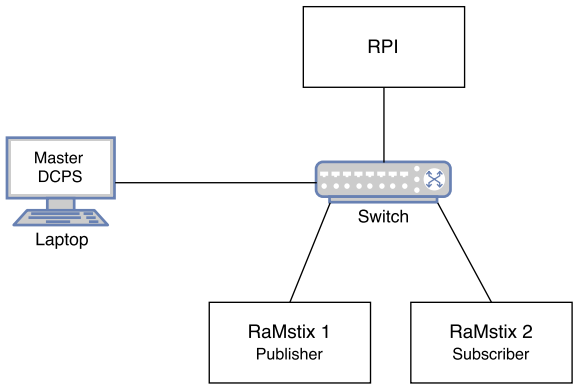


Figure B.11: Test setup for introducing extra load to the network.

The *Iperf* command used for starting a server is shown in Listing B.1 and the command used for starting a client for UDP traffic is shown in Listing B.2.

```
./iperf3 -s -i 1
```

Listing B.1: The command that starts an *Iperf* server

```
./iperf3 -c [serverIP] -u -t 1200 -b [load]M -i 1 -w 128k
```

Listing B.2: The command that starts an *Iperf* client for creating *UDP* traffic. The *serverIP* and *load* is depending on the location of the server and the load the user wants to induce on the network. Note that the socket buffer size is increased to 128k as it introduced less package loss.

Four different configurations are tested for the load test to show the influence of each test for the publisher and subscriber:

- *Client on RPI, server on Publisher*
- *Client on Publisher, server on RPI*
- *Client on Subscriber, server on RPI*
- *Client on RPI, server on Subscriber*

LUNA devices using the *Network Channel* can receive or send to multiple DDS topics and therefore these tests shows the performance of the network when other devices are using the same network.

The test is performed for publish-subscribe communication as the reader and writer are decoupled such that they do not influence each others performance. Also the influence of adding traffic to the network can be analyzed with publish-subscribe communication as multiple packages are needed to unblock all readers. This way the resulting frequency at the reader is a measure for the extra latency introduced by the network.

The difference between two timestamps at the publisher and at the subscriber side is again used of a measure of the performance. All the tests averaged out to 100Hz, but the standard deviations were different. The results of the publish-subscribe communication for the sending side is shown in Table B.3 and for the reading side in Table B.4.

The command line output of *Iperf* shows the actual bandwidth. By analyzing this output the following is observed:

- When the RaMstix is an *Iperf* server, it is not able to receive $50 Mbps$ UDP traffic from the RPI client as packets are dropped, but the data is induced on the network according to the *Iperf* client. Therefore the RaMstix is dropping the packets.
- When the RaMstix is an *Iperf* client, it is not able to produce $50 Mbps$ UDP traffic to the RPI server and the resulting traffic will be less than $50 Mbps$. ($38 Mbps$ for *Pub to RPI* and $40 Mbps$ for *Sub to RPI*.)

Table B.3: The standard deviation of the execution time of the writing side over 2500 samples under additional traffic on the network induced by *Iperf*.

Publisher	**Standard deviation (ms)**			
$Mbps$	RPI to Pub	Pub to RPI	Sub to RPI	RPI to Sub
0	0.1782	0.1782	0.1782	0.1782
25	0.3058	0.1816	0.1828	0.1816
50	0.3256	0.1828	0.1911	0.3256

Table B.4: The standard deviation of the execution time of the reading side over 2500 samples under additional traffic on the network induced by *Iperf*

Subscriber	**Standard deviation (ms)**			
$Mbps$	RPI to Pub	Pub to RPI	Sub to RPI	RPI to Sub
0	0.9982	0.9982	0.9982	0.9982
25	1.4053	1.3232	1.0822	1.3145
50	1.6306	1.6466	1.6494	1.4053

It is observed that the standard deviations at the publisher in Table B.3 increases with the increase of traffic on the network. This must be caused by extra traffic on the network, resulting in more work for the network stack and therefore more jitter in the real-time LUNA application.

However the increase in standard deviation is also obtained when the extra traffic is not directed to the publisher. The switch regulates the amount of data, resulting in extra latencies in the network stack not running in Xenomai, which results in more jitter in the real-time LUNA application.

Also when the traffic is directed to the RPI less deviation at the publisher is observed, with respect to the traffic directed to the RaMstixes (publisher and subscriber). This is explicable as the RaMstixes are not able to generate $50Mbps$ when the *Network Channel* is in operation. The RPI however is able to generate $50Mbps$ on the network. More traffic on the network results therefore in more standard deviation at the publisher.

The subscribing side shown in Table B.4 does not show the same clear distinctions between the four configurations. However in all configurations the standard deviation increases with increasing load on the network.

This increase in deviation is caused by less deterministic writes by the publisher due to the extra traffic on the network, and therefore also more variation in time in the unblocking of the readers is obtained. Probably also some extra latency is introduced by the switch, but this behavior is not measured.

By inspecting the graphs of the execution times no high outliers are observed, which indicates that the *Network Channel* is still able to communicate reliable with more data on the network, without increasing the amount of iterations before unblocking of all readers occur.

For rendezvous communication these deviations are obtained twice, once for the state exchange and once for the data exchange. However, rendezvous communication introduces waiting for packets, which results in unblocking of all readers at once, and therefore if the standard deviation stays within the bounds of one loop cycle almost no performance degradation will be observed as a loop cycle introduces more latency than the latency introduced by extra traffic.

The standard deviation at the subscriber is in the order of 1ms as two bands are obtained. To show these bands Figure B.12 is provided that shows the *RPI to Sub* results for the subscribing side. These bands are obtained as a data packet may contain not all the data for all the readers, resulting in unblocking one iteration earlier or later. It can also be observed that with $50Mbps$ of extra traffic on the network more often the unblocking of all readers is obtained in three iterations as the extra traffic introduced extra latencies.

The observation in all configurations for publisher and subscriber is that adding extra traffic to the network results in higher standard deviations at publisher and subscriber side. Rendezvous had to communicate state and data and therefore the extra latencies due to the network will be encountered twice. However for publish-subscribe communication the unblocking of the readers can take up to three iterations, where the unblocking of rendezvous communication

Figure B.12: The Subscriber for the *RPI to Sub* configuration that shows the results of added traffic to the network and bands due to the unblocking of readers in a loop.

only takes one iteration resulting in less influence of the extra traffic on the network than for publish-subscribe communication.

Test 2: Frequency test
The second stress test investigates the influence of higher sending rates with respect to the performance. Again a publisher and subscriber with 10 channels are used to investigate the maximum communication frequency for publish-subscribe and rendezvous communication. The frequency is increased from 100Hz up to 300Hz in steps of 50Hz. Again two RaMstixes are used, one runs the publisher application and the other one the subscriber application. First the measurements of the rendezvous communication are discussed and next the results for the publish-subscribe communication.

Table B.5 shows the results of the test for rendezvous communication. The observations made from this table are:

1. Maximum send and receive frequency is approximately 230Hz for TCP and 200Hz for UDP transport.
2. Overwrites occur from 200Hz, and more overwrites are observed for UDP transport.
3. Sending and receiving frequency are not equal for UDP transport.
4. Standard deviation for 100Hz is greater than other frequencies.

Note that this table also shows the amount of overwrites. The data dat is send from the writers are sequential, meaning that the data is incremented by one each send action. If at the subscriber the difference between two consecutive is greater than one, a value on that topic is overwritten. Only the data of one channel is logged, and therefore the overwrite column states the amount of overwrites for one channel. The overwrite column is an indication of the amount of overwrites that actually occur.

From Table B.5 can be seen that for TCP transport a maximum sending rate of approximately 230Hz is obtained and for UDP a lower rate of 200Hz is obtained as more packets are dropped in the network, indicated by more overwrites.

Table B.5: The average achieved frequency and standard deviation for rendezvous communication calculated over 2000 samples.

Rendezvous communication					
	TCP				
	Writer		**Reader**		
Frequency	**Avg (Hz)**	**Std (ms)**	**Avg (Hz)**	**Std (ms)**	**Overwrites**
100	100.0	0.2278	100.0	0.9763	0
150	150.0	0.1633	150.0	0.4569	0
200	200.0	0.2603	200.6	0.3457	10
250	233.4	0.2095	229.0	0.6088	73
300	233.8	0.2013	222.61	0.6424	58
	UDP				
	Writer		**Reader**		
	Avg (Hz)	**Std (ms)**	**Avg (Hz)**	**Std (ms)**	**Overwrites**
100	100	0.1705	100.0	0.8799	0
150	150	0.1434	150.0	0.5737	0
200	200.0	0.3838	198.3	0.2325	2
250	223.7	0.4516	200.9	0.2400	230
300	223.8	0.4446	196.2	0.4218	278

The network alone introduces a latency of approximately 0.3ms as shown in Appendix G, which is encountered for the *reader is ready* package and the data. The time it takes to enqueue data and sending it to OpenDDS is also encountered for the *reader is ready* package and the data, and is approximately 2.5ms (From Table B.1). The theoretical sending speed is therefore 5.6ms, or 178Hz. This theoretical sending speed is less optimistic, and can be explained by:

- The writing time for a *reader is ready* message is probably shorter, as it contains no unbounded sequences and multiple data elements.
- The latency analysis consists of *printf* statements, which due not contribute to performance.

Therefore it is possible that a maximum sending rate of 230Hz is obtained.

With rendezvous transport the sending and receiving side are coupled and therefore the same send and receive frequencies are expected. However when the maximum sending speed is approached the rendezvous communication is experiencing overwrites in the buffer at the reader side as not all readers are unblocked at the same time, which indicates a failing waiting function for higher sending frequencies.

The result is a lower receiving frequency as the data bundles are not complete, resulting in multiple iterations before all the readers are unblocked. As mentioned before active checking of *ownTopics* should be performed in the *DDSReactor* to solve these overwrites. More overwrites are detected with UDP transport than with TCP transport as UDP also introduces packet loss, resulting in an even lower receiving frequency. Package loss with UDP is also concluded from Section B.3.1.

The standard deviation for the 100Hz communication is greater as the band below the 10ms is present the complete test. The other sending frequencies show similar behavior but appear later in time. Figure B.13 shows the difference between timestamps for the rendezvous reader with TCP transport to show the presence of these bands. The 250Hz and 300Hz do not show

bands of 2.5ms any more as dequeuing and writing to OpenDDS takes longer than 2.5ms, resulting in a loop frequency dependent on the dequeue and write times of OpenDDS.

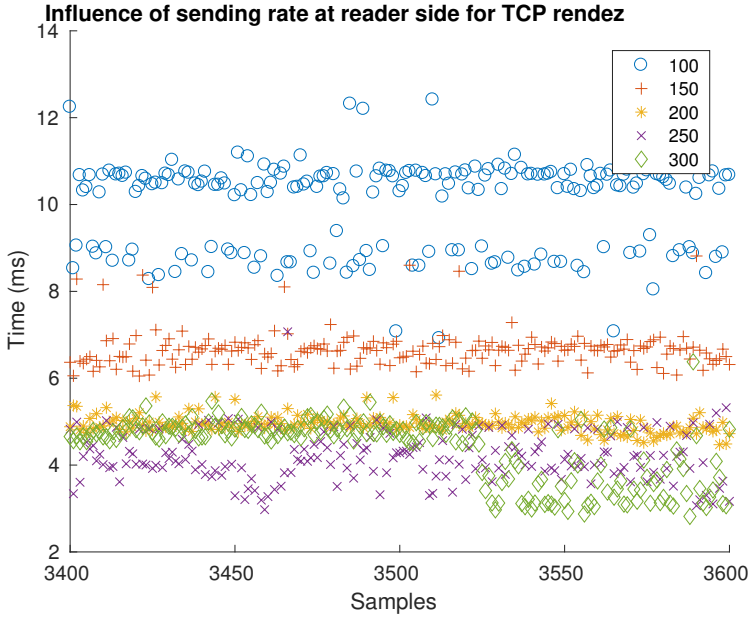


Figure B.13: The rendezvous reader with TCP transport for different sending frequencies.

These bands occur if a message is send one iteration earlier as discussed in Section B.3.1, but bands can also occur when the sending frequency is not a multiple of inbound and outbound loop frequencies in the *DDSReactor* as is illustrated in Appendix H.

So when using rendezvous communication the user should use TCP transport and choose a frequency lower than 200Hz to guarantee the functioning of the wait function. The result is less deviation at the subscriber side as the readers are unblocked in one iteration. Furthermore it is recommended to update the waiting functionality to explicitly check the presence of the topics.

The same test is performed for the publish-subscribe communication. The results of this test are shown in Table B.6 and the following is observed from this table:

1. The writer is able to achieve all the commanded frequencies.
2. The reader is able to receive with 247Hz for TCP and 250Hz for UDP.
3. UDP seems to be able to communicate faster without overwrites.

The writer in publish-subscribe communication is not blocking, which means that the write action is finished once the data is enqueued. For faster writing speeds this enqueueing takes longer as another process is simultaneous dequeueing as there is more data in the queue. The writer has an increase in standard deviation for higher frequencies as it is more busy with dequeueing and writing to OpenDDS resulting in more deviation at the LUNA application that writes. The writer is able to achieve the commanded sending frequency as the latencies introduced by enqueueing are smaller than the sending frequency.

The maximum speed for receiving is a somewhat higher than for the rendezvous communication as no *reader is ready* message needs to be send by the reader. When bundles with all the *ownTopic* are published from sender to receiver it is expected to reach a frequency around 400Hz as the time to publish a message onto a DDS topic is approximately 2.5ms (Table B.1). However not all data has to be available in one data bundle for publish-subscribe communication, resulting in the unblocking of all the readers in more than one iteration. If it takes two

Table B.6: The average achieved frequency with the standard deviation for publish-subscribe communication calculated over 2000 samples.

Publish and Subscribe communication					
	TCP				
	Writer		**Reader**		
Frequency	**Avg (Hz)**	**Std (ms)**	**Avg (Hz)**	**Std (ms)**	**Overwrites**
100	100.0	0.1872	100.0	1.0023	0
150	150.0	0.1877	150.0	1.1963	0
200	200.0	0.1887	200.0	1.4310	0
250	250.0	0.3518	244.5	1.2365	46
300	300.0	0.8468	247.0	1.3313	429
	UDP				
	Writer		**Reader**		
	Avg (Hz)	**Std (ms)**	**Avg (Hz)**	**Std (ms)**	**Overwrites**
100	100.0	0.1713	100.0	0.9418	0
150	150.0	0.1882	150.0	1.1364	0
200	200.0	0.1715	200.0	1.3104	0
250	250.0	0.1800	250.0	1.0997	0
300	300.0	0.6962	225.0	1.6352	666

cycles to unblock all readers than it takes another 2.5ms, resulting in a receiving frequency of 200Hz. The readers can unblock however with one data bundle, resulting in a maximum receiving speed of around 250Hz. To obtain higher receiving frequencies active checking of topic availability must be performed, resulting in complete packets and unblocking of all readers in one iteration.

The data is only flowing from publisher to subscriber and no states are exchanged, and therefore probably less UDP packets are lost, that results in a better performance for UDP than for TCP. However for the 300Hz communication the TCP transport has less overwrites than the UDP transport which indicates that with higher publishing rates the TCP transport is more reliable.

B.4 Conclusion

Four tests are conducted to investigate the performance of this *Network Channel* implementation. The conducted tests are:

- *Transport*
- *Bundled data validation*
- *Latency analysis*
- *Stress test*

By using rendezvous communication the differences between UDP and TCP transport are investigated. The conclusion is that TCP communication results in less jitter on the subscriber side as TCP does not lose any packets and therefore all readers always unblock at the same time. The latency analysis shows that the writing time to OpenDDS with UDP and TCP transport are approximately equal. Also when using TCP transport at higher sending rates less overwrites occur at the subscriber than with UDP transport. Therefore TCP introduces reliable transport resulting in bounded loop times at the subscriber side.

The waiting functionality is functioning correctly for sending rates at 100Hz and resulted in the unblocking of the readers in one iteration. However when the sending rate is increased to 200Hz, overwrites are observed that indicate that not all data for all subtopics was available in a bundle at some point.

For higher sending rates the waiting functionality fails to add all data samples for the different subtopics. Therefore it is recommended to update the waiting functionality to explicitly check for all the topics to be available in a packet. This will result in faster loop times at the subscriber as one iteration is sufficient to unblock all readers.

With the latency test it is observed that writing to OpenDDS is a rather expensive operation. By using a more resource rich system the writing time can be drastically reduced. Also implementing the default Linux scheduler (OTHER) instead of the default real-time scheduler (FIFO) resulted in a decrease of writing time and standard deviation. However the overall performance dropped drastically probably due to re-scheduling of the OTHER thread. The real time FIFO threads causes mode switches due to OpenDDS and the network stack, which results in higher writing times and more standard deviation with respect to the OTHER scheduler.

Another significant latency is obtained at the subscribing side that consists of calling the callback function in the *DDSReactor* that provides the data to the *DDSChannel* ($t_{cbf,unblocked}$). This latency is not directly linked to an event, and is probably caused by the re-scheduling of the thread by a higher priority thread, but this is not validated.

Extra traffic on the network results in more standard deviation at writer and reader. The increase of standard deviation at the writer is caused by a more busy network stack, resulting in more jitter in the execution time of the LUNA application.

Adding extra load to the network resulted in a maximum of three iterations before all the readers were unblocked, which was also obtained without load. However unblocking in three iterations is more often observed and is caused due to the extra load on the network. Adding extra traffic on the network introduces extra latencies, but does not significantly reduce the performance with respect to no load on the network.

It is recommended to get rid of the threads that continuously loop. The performance of the *Network Channel* will increase as the latencies caused by the loop time of a thread is not present. Also the influence of external parameters can be better investigated if no loops are present.

To omit the looping threads the callback functions at the *DDSChannel* should be provided to the OpenDDS interface, such that a reader can be unblocked directly after receiving the data. For the writers a write to OpenDDS should be performed when all topics for a *commonTopic* are present, resulting in a write independent of a loop.

The requirements at the beginning of this thesis for the *Network Channel* are all met as reliable communication between LUNA applications is obtained with TCP as transport mechanism. OpenDDS provides a rich set of QoS settings that may even introduce more reliability for the *Network Channel*, but this needs to be more investigated. Also rendezvous communication is implemented with a trade-off that only the reader presents its readiness to the writer. The *Network Channel* is implemented as a component in LUNA and also supports buffered channels. At TERRA level is a DDS hardware port added such that the *Network Channel* can be used in a CSP model in TERRA. Scalability is provided, because the *DDSReactor* on one host can send and receive to multiple DDS topics when publish-subscribe communication is used and the DCPS server provides auto discovery of endpoints. It is however not tested if the *Network Channel* is able to communicate with a ROS2 client using DDS, but it should theoretical be inter-operable.

C Code of the integration test

The C++ code of the *cpp_conversion* block from the LUNA application that runs on the RaMstix of the integration test is shown in Listing C.1. The values are normalized to represent a value between -1 and 1. And also a deadzone is implemented as the joystick axes never return zero if the joystick is not touched.

```
//Variables set in the constructor
double deadzone = 5000;
double max_joystick = 2^15;
double max_val = 1;
double scaling = max_val/(max_joystick-deadzone);
//End variables set in the constructor

if(abs(x) > deadzone) {
    x_conv = (abs(x) - deadzone) * scaling;
    x_conv = (x < 0) ? -x_conv : x_conv;
} else {
    x_conv = 0;
}

if(abs(y) > deadzone) {
    y_conv = (abs(y) - deadzone) * scaling;
    y_conv = (y > 0) ? -y_conv : y_conv;
} else {
    y_conv = 0;
}

angle_conv = angle_scale * angle;
```

Listing C.1: The C++ code of the *cpp_conversion* code block that converts the values received by the Network Channel to values between -1 and 1.

The C++ code of the *cpp_safety* block of the LUNA application on the RaMstix for the integration test that checks if the values do not exceed limits and it also checks the liveliness of the values as is also shown in Listing C.2. When the values for the x and y direction haven't changed for several iterations the output is zero, which causes the youBot to stops its motion.

```
\\ Values set in constructor
float epsilon = 0.001;
int maxCount = 200;
\\ End values set in constructor

if(prev_x == (float)x_conv){
    counter_x++;
}else {
    counter_x = 0;
}

if(prev_y == (float)y_conv){
    counter_y++;
}else {
    counter_y = 0;
}

if((float)(1 - x_conv) < epsilon ||
            (float)(1 + x_conv) < epsilon){
    counter_x = 0;
}

if((float)(1 - y_conv) < epsilon ||
            (float)(1 + y_conv) < epsilon){
    counter_y = 0;
}

prev_x = x_conv;
prev_y = y_conv;

x_conv = (counter_x > maxCount) ? 0 : x_conv;
y_conv = (counter_y > maxCount) ? 0 : y_conv;

//safety on limits.
if(x_conv > 1){
    x_conv = 1;
} else if (x_conv < -1){
    x_conv = -1;
}
if(y_conv > 1){
    y_conv = 1;
} else if (y_conv < -1){
    y_conv = -1;
}
if(angle_conv > 1){
    angle_conv = 1;
} else if (angle_conv < -1){
    angle_conv = -1;
}
```

Listing C.2: The C++ code of the *cpp_safety* code block that makes sure the values are within bounds and no freezing of values occurs.

D Demo

This appendix describers how the demo can be run on an RaMstix and a ubuntu machine[1] by using the pre-compiled binaries. Building from source code is also possible and discussed in Appendix E.

OpenDDS 3.11 must be available on both systems. For OpenDDS on the RaMstix a pre-build version is provided in a git repository: https://git.ram.ewi.utwente.nl/wijnholtr/OpenDDS-3.11_RAMstix_Yocto_build.

OpenDDS 3.11 can be installed on the Ubuntu machine by following the instructions on the OpenDDS.org website.

Clone or copy this repository to the RaMstix to provide the RaMstix with an install of OpenDDS. Open the *setenv.sh* file and provide the correct paths to the OpenDDS directory on the RaMstix.

Git clone the repository containing the binaries: https://git.ram.ewi.utwente.nl/wijnholtr/Demo

Copy the laptop directory to anywhere on the Ubuntu machine and copy the RaMstix directory to anywhere on the RaMstix.

The Ubuntu and RaMstix directory both contain a *bin* directory locating an executable. Before running the executable the environment variables of OpenDDS needs to be set by sourcing the *setenv.sh* located in the installation folder of OpenDDS 3.11 on the RaMstix and Ubuntu machine.

There must be a device on the network that hosts the DCPS service (can be the RaMstix, Ubuntu machine or another machine with OpenDDS installed) and is started with (after sourcing *setenv.sh*):

```
$DDS_ROOT/bin/DCPSInfoRepo -ORBListenEndpoints iiop://:12345
```

Listing D.1: The command to start an DCPS server. The *setenv.sh* file should have been sourced first.

In the laptop and RaMstix directory a file called *dds_config.ini* is present. This file indicates the transport to use, and where the DCPS service is hosted. Change the line with *DCPSInfoRepo* such that it contains the correct IP address. Leave the port number as is also shown in Listing D.2. Also make sure that both *dds_config.ini* files contain the same transport mechanisms.

```
DCPSInfoRepo=[IP address of DCPS service]:12345
```

Listing D.2: The line that should read the correct IP address in the *dds_config.ini* file.

Connect the Xbox controller to the Ubuntu machine. Now open a terminal on the Ubuntu machine and navigate to the OpenDDS directory to source the *setenv.sh* file. Then navigate to the *bin* directory that contains the executable for the laptop. Run the application by running the command: *./top_arch_generator*.

Make sure the RaMstix is connected to a WiFi network (*wpa_supplicant*) that is in the same local network as the Ubuntu machine and connect the youBot with an Ethernet cable to the RaMstix Ethernet port. Also make sure that the Ethernet address is a fixed address and in another subnet than the WiFi network.

SSH to the RaMstix or use the serial interface. Type the command *sudo su* to enable root privileges. Navigate to the OpenDDS directory on the RaMstix to source the *setenv.sh* file. Then

[1] Ubuntu 16.04, Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz

navigate to the *bin* directory that contains the executable for the RaMstix. Run the application by running the command: *./top_arch_controlloop*. Root privileges are necessary as the SOEM EtherCAT master needs root privileges for accessing the network stack.

If everything went well a communication over the Network Channel is initiated and the youBot can be moved by using the joystick. If for some reason the execution of the binaries didn't work, then the application should be build from source code as is shown in Appendix E

E Building demo from source

For building the demo from source the following dependencies are required:

- A build of LUNA for RaMstix and laptop.
- Source code for demo on RaMstix and laptop.
- The Yocto environment to cross-compile for the RaMstix.
- The joystick interface from Noakes (2017).
- OpenDDS 3.11 for RaMstix and Ubuntu machine.

A pre-requisite for both platforms is to have an installation of OpenDDS 3.11. For the installation on the Ubuntu machine follow the instructions provided by `OpenDDS.org`.

A repository containing the dependencies above, together with pre-compiled versions of LUNA for linux-x64 and xenomai-arm-v7 is made available and can be git cloned from
`https://git.ram.ewi.utwente.nl/wijnholtr/Demo_source`

In principle this repository contains all the dependencies, but if a custom LUNA build with *dds-channels* is required then the LUNA project can be cloned from:
`https://git.ram.ewi.utwente.nl/wijnholtr/LUNA`

Following the instructions provided by the RaM website LUNA can be compiled:
`https://www.ram.ewi.utwente.nl/ECSSoftware/luna.php`

E.0.1 Compiling for Ubuntu

Open a terminal on the Ubuntu machine and source the *setenv.sh* script in the OpenDDS installation directory on the Ubuntu machine as shown in Listing E.1 to set the environment variables of OpenDDS.

```
source setenv.sh
```

Listing E.1: The command to set the environment variables for openDDS.

Navigate to the *demo_laptop* directory with a terminal and run *make clean; make*, which builds the project and places the executable in the *bin* folder.

Provide the correct transport and IP address of the DCPS server in the *dds_config.ini* file and plug in the Xbox 360 controller. The DCPS service can be started with the commands shown in Listing E.2. Note that the device running the DCPS service has to have a OpenDDS installation.

```
$DDS_ROOT/bin/DCPSInfoRepo -ORBListenEndpoints iiop://:12345
```

Listing E.2: The command to start an DCPS server. The *setenv.sh* file should have been sourced first.

Navigate to the *bin* folder and run the executable with the command provided in Listing E.3.

```
./top_arch_generator
```

Listing E.3: The command to start the application that reads the joystick and sends it on the *Network Channel*

The result is a LUNA application that is able to send the joystick values to the RaMstix using the *Network Channel*.

E.0.2 Compiling for RaMstix

Navigate with a terminal on the Ubuntu machine to the cloned repository *Demo_source* and run the command shown in Listing E.4.

```
./poky-glibc-x86_64-ramstix-20sim-image-
cortexa8hf-vfp-neon-toolchain-1.8.2.sh
```

Listing E.4: The command to build the Yocto SDK environment.

Install the SDK in the *ramstixSDK* folder, which is already located in the *Demo_source* directory.

Change the *sysroot* variable in the *tools.mk* file (line 13 and 14) of the LUNA build in the folder *lunuabuilds/luna-xenomai-arm-v7-Posix* to match the absolute path of the *ramstixSDK*.

Also change the *setenv.sh* file in the OpenDDS directory provided for the RaMstix to match the absolute path of the OpenDDS installation directory.

Navigate in a terminal on the ubuntu machine to the *demo_ramstix* folder and source the *envAndXenomaiDir* with the command provided in Listing E.5 to set the environment variables of the RaMstix SDK and OpenDDS.

```
source envAndXenomaiDir
```

Listing E.5: The command to set environment variables of OpenDDS and the RaMstixSDK.

Then enter the commands *make clean; make* to start building the application for the RaMstix. The resulting executable is now located in the *bin* directory. If errors are obtained, double check if the paths provided in *tools.mk* and *setenv.sh* are correct.

Make sure the RaMstix is connected to a WiFi network (*wpa_supplicant*) that is in the same local network as the Ubuntu machine and connect the youBot with an Ethernet cable to the RaMstix Ethernet port. Also make sure that the Ethernet address is a fixed address and in another subnet than the WiFi network.

Copy the executable, *dds_config.ini* and the OpenDDS build for the RaMstix to the Ramstix, with for example *scp*. Change the *setenv.sh* on the RaMstix again to match the absolute path of the OpenDDS directory on the RaMstix.

On the RaMstix first enter *sudo su* to elevate to root privileges. Next source the *setenv.sh* to set the environment variables of OpenDDS. Lastly navigate to the location of the executable and run the command shown in Listing E.6.

```
./top_arch_controlloop
```

Listing E.6: The command to start the LUNA application on the RaMstix.

Make sure the DCPS IP address in the *dds_config.ini* file is correct and that the same transport is used as is specified on the Ubuntu machine.

F Getting TERRA with DDS ports

This appendix describes how to obtain the TERRA project to use the DDS ports in TERRA.

1. Create a folder somewhere on your machine. This folder is referred to as *root* folder.
2. Git clone this project in the root folder:
   https://git.ram.ewi.utwente.nl/wijnholtr/TERRA
3. Install a fresh Eclipse LUNA or git clone this repository:
   https://git.ram.ewi.utwente.nl/wijnholtr/Eclipse_LUNA
4. Start Eclipse LUNA.
5. In an empty java workspace in Eclipse: File > Import > Existing Projects into Workspace
6. Browse to the cloned TERRA project and import to the workspace.
7. Click on the "Run TERRA" button (green play button) at the right top of Eclipse to start TERRA.
8. When an architecture model is created the "DDS port" will show up at the right side and models can be constructed.
9. A LUNA build must be available to actually generate the executable code. The LUNA project can be obtained from https://git.ram.ewi.utwente.nl/wijnholtr/LUNA. Also pre-compiled binaries are available and instructions to download them is provided in Appendix E.

G Propagation delay on Ethernet

A ping request is used to approximate the latency between two RaMstixes. A ping request uses the ICMP protocol which is located at the same layer as IP, which is the *Network layer* (layer 3) of the OSI model. The *Network Layer* is depicted in the OSI model shown in Figure G.1

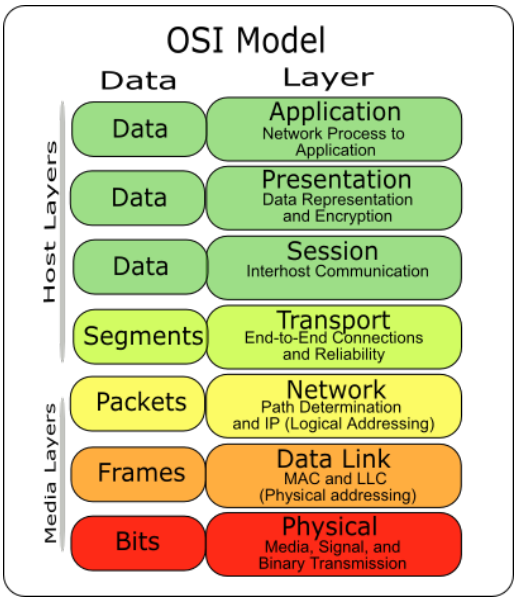


Figure G.1: An OSI 7 model showing the seven layers of networking. (Hewitt (2005))

When performing a ping request between two devices, the *Network Layer*, *Data link layer* and *physical layer* are passed four times. Twice for the requesting device, and twice for the device that is pinged. The assumption is made that the network is symmetrical, and therefore the half of the observed ping latency is the time between the *Network Layers* of the devices.

The result of the ping request is shown in Figure G.2. The setup consists of two RaMstixes and a switch. The same setup is used for all the tests.

```
ram@overo:~$ ping 192.168.1.102
PING 192.168.1.102 (192.168.1.102) 56(84) bytes of data.
64 bytes from 192.168.1.102: icmp_seq=1 ttl=64 time=0.474 ms
64 bytes from 192.168.1.102: icmp_seq=2 ttl=64 time=0.616 ms
64 bytes from 192.168.1.102: icmp_seq=3 ttl=64 time=0.603 ms
64 bytes from 192.168.1.102: icmp_seq=4 ttl=64 time=0.625 ms
64 bytes from 192.168.1.102: icmp_seq=5 ttl=64 time=0.619 ms
64 bytes from 192.168.1.102: icmp_seq=6 ttl=64 time=0.609 ms
64 bytes from 192.168.1.102: icmp_seq=7 ttl=64 time=0.624 ms
64 bytes from 192.168.1.102: icmp_seq=8 ttl=64 time=0.626 ms
64 bytes from 192.168.1.102: icmp_seq=9 ttl=64 time=0.597 ms
64 bytes from 192.168.1.102: icmp_seq=10 ttl=64 time=0.626 ms
64 bytes from 192.168.1.102: icmp_seq=11 ttl=64 time=0.606 ms
64 bytes from 192.168.1.102: icmp_seq=12 ttl=64 time=0.624 ms
64 bytes from 192.168.1.102: icmp_seq=13 ttl=64 time=0.628 ms
^C
--- 192.168.1.102 ping statistics ---
13 packets transmitted, 13 received, 0% packet loss, time 12008ms
rtt min/avg/max/mdev = 0.474/0.605/0.628/0.051 ms
```

Figure G.2: The output of the ping request between two RaMstix devices via a switch.

From Figure G.2 can be seen that the average latency is 0.605 ms resulting in a latency of approximately 0.3 ms from device to device. This latency is used as a measure of latency for sending data from device to device.

To investigate the time on the network stack the tool *ku-latency* (Vilimpoc (2008)) is executed on a RaMstix. The result is shown in Listing G.1.

```
time_kernel                    : 1497353708.265088
time_user                      : 1497353708.265194
Total Average                  : 30280/294 = 102.99 us
Rolling Average (32 samples)   : 105.47 us
```

Listing G.1: The latency introduced between the user space and kernel space of the linux network stack.

The time it takes for a message to pass through the kernel space to the user space takes about $105.47\mu s$. This is negligible with respect to the time it takes for the ping request. Also probably some overlap is present between the *ku-latency* test and the ping request. Therefore this latency between kernel and user space is neglected.

H Dependency of reactor loop frequency and sending frequency

It is important to choose a suitable sending rate with respect to the reactor frequency. The sending frequency must be a multiple of the reactor frequency as otherwise aliasing will occur that results in bands around the expected send frequency at the subscriber.

The result of a mismatch between the reactor loop frequency and the sending rate of the publisher is illustrated by Figure H.1. The reactor loop frequency in the first test is configured at 250 Hz and in the second test at 400 Hz. In both cases the publisher was sending at 100 Hz.

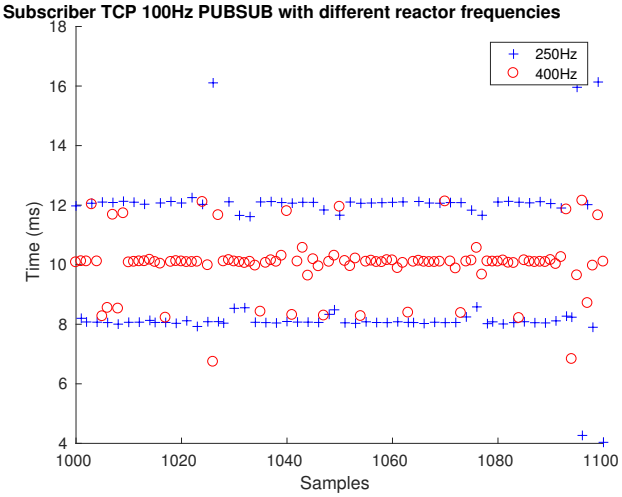


Figure H.1: Two different reactor frequencies that show the aliasing when the sending frequency is not a multiple of the reactor frequency.

With a reactor loop frequency of 250 Hz it basically means the data is samples every 4 ms. The sending rate is at 10 ms so bands occur at 8 and 12 ms (when inbound and outbound loop are in sync). When having a reactor loop at 400 Hz this is not obtained as the reactor frequency is a multiple of the send frequency. Note that sometimes the readers are unblocked in two loop iterations, and sometimes a second set of messages was presented a little faster that resulted in earlier unblocking. This results is two bands around the 10 ms line separated by 2.5 ms. The reactor loop frequency is a hard coded value and is therefore not that easy to change, therefore it is advised to select a send frequency that is a multiple of the reactor frequency.

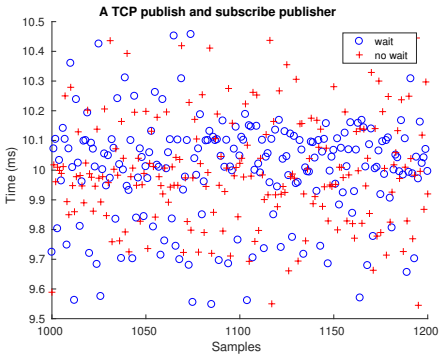
I Waiting for publish and subscribe communication

The results of the measurements with waiting with publish and subscribe communication is shown in Figure I.1. The publishing side behaves similar as the rendezvous case, but the subscriber does not unblock always in one loop iteration. The bands around the 10 ms line show that not all topics are available in one packet. Also there is some data between 18-20 ms which indicates that some samples must have been lost as it took several iterations before all readers were unblocked.

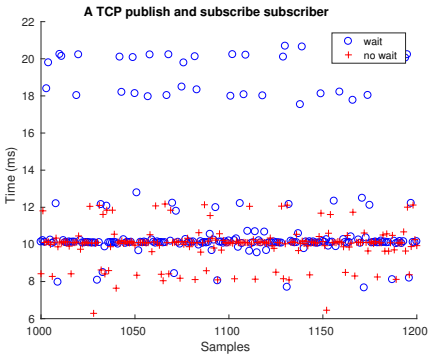
Probably some samples are discarded at the sending side as this waiting functionality has a slightly different implementation. This implementation makes sure that exactly the amount of data values are written to OpenDDS. If more data is available it performs writes in multiple iterations. The rendezvous waiting communication is not implementing this feature as it performed in all times at 100 Hz sending rate. However the implementation of this feature probably has a *off-by-one* error.

When introducing the check of the availability of every topic in a data bundle this functionality is not necessary anymore as all the data bundle will consist of exactly all the topics.

There is however a lack of time to debug this problem. Waiting for complete packets for publish and subscribe communication is less crucial as no *reader is ready* message needs to be exchanged.



(a) The publish and subscribe publisher that publishes data with and without waiting for writers to have presented all their data.

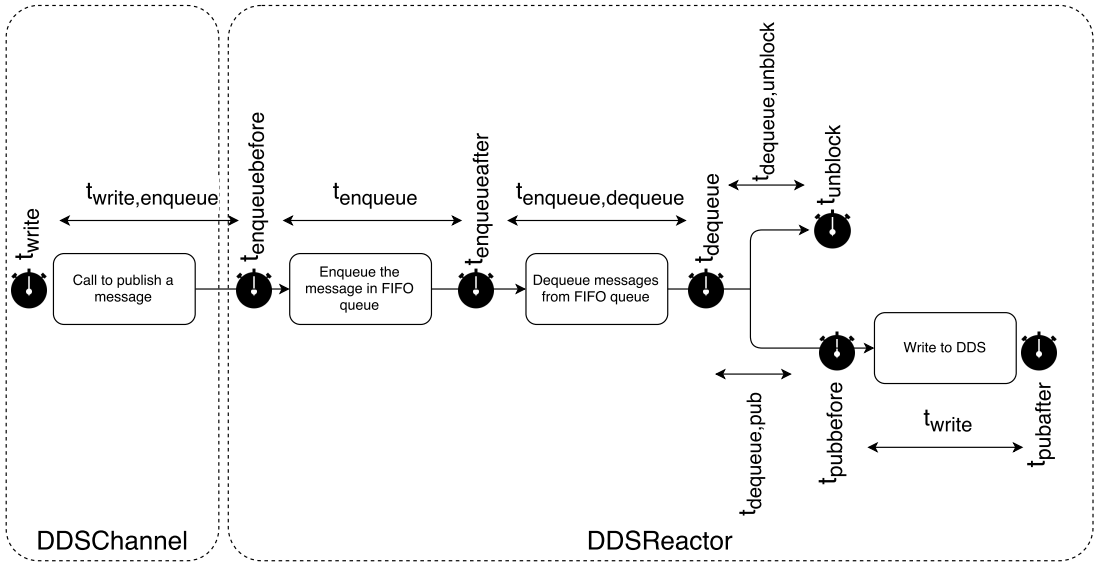


(b) The result at the subscriber for waiting or not waiting for all data to be present at the publisher.

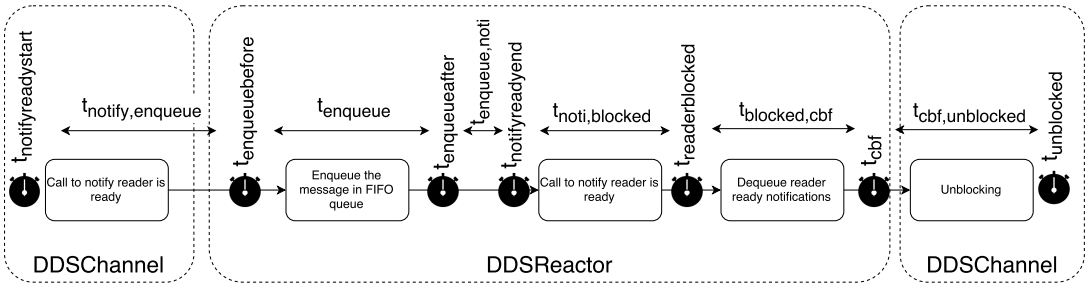
Figure I.1: The test that shows the performance of waiting versus no waiting for sending data at 100Hz.

J Latency analysis on rendezvous communication

The location of the timestamps for the rendezvous communication are shown in Figure J.1.



(a) The location of the timestamps for the publisher.



(b) The location of the timestamps for the subscriber.

Figure J.1: The location of the timestamps for analyzing the latencies for the rendezvous communication. The stopwatches indicate a timestamp and the arrows between timestamps indicate the difference between timestamps.

The average latencies for the different sections are shown for the rendezvous publisher in Table J.1 and for the rendezvous subscriber in Table J.2.

From Table J.1 can be seen that t_{write}, $t_{enqueue}$, and $t_{write,enqueue}$ are comparable to the publish and subscribe communication. The time between enqueueing and dequeueing ($t_{enqueue,dequeue}$) however is around 2.5 ms, which is explicable as the rendezvous communication introduces a wait, that results in the dequeueing of a message one iteration later if not all the data was available. If this happened in the beginning than it will always take one complete iteration to fill a data bundle resulting in 2.5 ms. Also the time between dequeueing and publishing ($t_{dequeue,pub}$) is somewhat higher as sometimes the reader is not ready to receive yet, resulting in somewhat higher dequeueing to publish time.

The values at the subscriber are as expected when it is compared to the times for publish and subscribe communication. However a very high callback function to unblocking is obtained, which was also observed for the publish and subscribe communication. It is not linked to an event. The explanation for the publish and subscribe was that the RaMstix was probably busy

Table J.1: Latencies at the publisher for rendezvous communication

	Average latency (ms)	
Δt	**TCP**	**UDP**
$t_{write,enqueue}$	0.0363	0.0393
$t_{enqueue}$	0.0084	0.0084
$t_{enqueue,dequeue}$	2.5685	2.5562
$t_{dequeue,unblock}$	1.8264	1.9437
$t_{dequeue,pub}$	0.3042	0.2564
t_{write}	1.1024	1.2366

with other processes which resulted in rescheduling of the call for the callback function. A higher average latency is obtained for this callback function to actually unblocking than for the publish and subscribe communication. The rendezvous subscriber has also to publish a state exchange to OpenDDS that induces an extra load on the RaMstix, which is the only probable cause of this extra latency.

Table J.2: Latencies at the subscriber for rendezvous communication

	Average latency (ms)	
Δt	**TCP**	**UDP**
$t_{notify,enqueue}$	0.0099	0.0106
$t_{enqueue}$	0.0055	0.0055
$t_{enqueue,noti}$	0.0152	0.0153
$t_{noti,blocked}$	0.0058	0.0056
$t_{blocked,cbf}$	10.008	11.1084
$t_{cbf,unblocked}$	3.0989	5.8849

Bibliography

Beckhoff (2012), ETG2200 Slave Implementation Guide, http://www.ethercat.org/pdf/english/ETG2200_V2i0i0_SlaveImplementationGuide.pdf.

Bezemer, M. (2011), *LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework*, Ph.D. thesis.

Bezemer, M. and J. F. Broenink (2015), Connecting ROS to a Real-Time Control Framework for Embedded Computing, Technical report.

Bezemer, M. M. (2014), Hardware Ports - Getting Rid of Sandboxed Modelled Software.

Blechmann, T. (2011), Chapter 22. Boost.Lockfree - 1.63.0, http://www.boost.org/doc/libs/1_63_0/doc/html/lockfree.html.

Busch, D. (2010), Object Computing, Inc. - Middleware News Brief - June, 2010, http://mnb.ociweb.com/mnb/MiddlewareNewsBrief-201004.html.

Corsaro, A. (2013), OpenSplice DDS Tutorial – Part II.

Force Dimension (2017), Force Dimension - Products - Omega.7 - Features, http://www.forcedimension.com/products/omega-7/features.

Frijnts, S. D. (2014), Upgrading the Safety Layer and Demo of the youBot Robot, Pre-msc report 007ram2014, University of Twente.

Geomagic Touch (2016), Geomagic Touch (Formerly Geomagic Phantom Omni) Overview, http://www.geomagic.com/en/products/phantom-omni/overview.

Gumstix (2017), Overo® FireSTORM-P COM, https://store.gumstix.com/coms/overo-coms/overo-firestorm-p-com.html.

Hewitt, O. c. b. J. (2005), English: OSI RM Model. Intent: This Was Created to Clearly Show Layers in the OSI Model.

Hoare, C. A. R. (1978), Communicating Sequential Processes, **vol. 21**, no.8, pp. 666–677, ISSN 0001-0782, doi:10.1145/359576.359585.

Iperf (2017), iPerf - The TCP, UDP and SCTP Network Bandwidth Measurement Tool, https://iperf.fr/.

Jishenaz (2013), English: SVG Drawing of an Xbox 360 Controller. Useful for Making Button Mapping Diagrams.

Kempenaar, J. J. (2014), *Communication Component for Multiplatform Distribution of Control Algorithms*, Msc report 001ram2014, University of Twente.

Kohlhoff, C. (2017), Boost.Asio - 1.64.0, http://www.boost.org/doc/libs/1_64_0/doc/html/boost_asio.html.

KUKA (2015), API Architecture - youBot Wiki, http://www.youbot-store.com/wiki/index.php/API_architecture.

Kurose, J. F. and K. W. Ross (2012), *Computer Networking: A Top-Down Approach (6th Edition)*, Pearson, 6th edition, ISBN 0-13-285620-4 978-0-13-285620-1.

Moodycamel (2014), A Fast General Purpose Lock-Free Queue for C++, http://moodycamel.com/blog/2014/a-fast-general-purpose-lock-free-queue-for-c++.

Nissanke, N. (1997), *Realtime Systems*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, ISBN 978-0-13-651274-5.

Noakes, D. (2017), A Minimal C++ Object-Oriented API onto Joystick Devices under Linux.

OMGIDL (2017), OMG IDL, http://www.omg.org/gettingstarted/omg_idl.htm.

OpenDDS (2017a), DDS Overview, http://opendds.org/about/dds_overview.html.

OpenDDS (2017b), OpenDDS Developer's Guide, http://download.objectcomputing.com/OpenDDS/OpenDDS-latest.pdf.

Qihoo360 (2017), Benchmark Lockfree versus Mutex, https://github.com/Qihoo360/evpp/blob/master/docs/benchmark_lockfree_vs_mutex.md.

RaM (2017a), RaMstix FPGA Board Documentation: RaMstix Overview, https://www.ram.ewi.utwente.nl/ECSSoftware/RaMstix/docs/index.html.

RaM (2017b), Robotics and Mechatronics - ECS Software - TERRA, https://www.ram.ewi.utwente.nl/ECSSoftware/terra.php.

ROVE (2017), Robotics and Mechatronics - RoVe, https://www.ram.ewi.utwente.nl/research/project/rove.html.

Smyth, N. and J. S. Davis II (1999), CSP Domain, https://ptolemy.eecs.berkeley.edu/papers/99/HMAD/html/csp.html.

Spil, T. (2016), User-Input Device Based Command and Control of the youBot Using a RaMstix Embedded Board.

van de Ridder, L. (2017), Design and Implementation of Embedded Control Software for a Demonstrator Using a Model-Driven Approach, in progress.

van der Werff, W. (2016), Connecting Two Robot-Software Communicating Architectures: ROS and LUNA.

Vilimpoc, M. (2008), Measuring Latency in the Linux Network Stack between Kernel and User Space., https://vilimpoc.org/research/ku-latency/.

Wilterdink, R. (2011), *Design of a Hard Real-Time, Multi-Threaded and CSP-Capable Execution Framework*, Ph.D. thesis, University of Twente.

Xenomai (2014), Finding Spurious Relaxes – Xenomai, https://xenomai.org/2014/06/finding-spurious-relaxes/.

ZMQ (2014), Distributed Messaging - Zeromq, http://zeromq.org/.