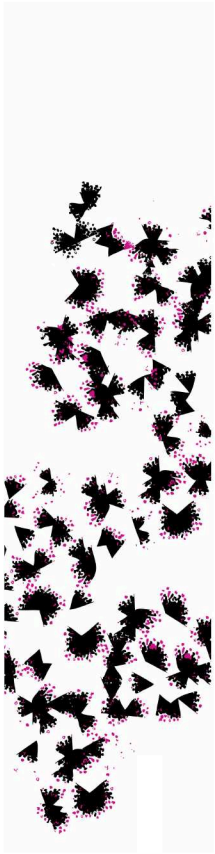

ROBOTICS AND MECHATRONICS

Connecting ROS to the LUNA embedded real-time framework

W.M. (Mathijs) van der Werff

MSc Report

Committee:
Prof.dr.ir. S. Stramigioli
Dr.ir. J.F. Broenink
Dr.ir. T.J.A. de Vries
Ir. E. Molenkamp
Z. Lu, MSc

July 2016

018RAM2016
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE.

MIRA CTIT
BIOMEDICAL TECHNOLOGY AND TECHNICAL MEDICINE

Summary

In this document, a report is presented showing the design, implementation and test results of a 'bridge' between ROS and LUNA, an embedded real-time capable framework.

Two trends can be distinguished in modern robotics: One is the need for more computational power, used to run algorithms to process data from complex sensors. The other trend are more mobile and energy-efficient setups. These trends seem to conflict: computational power generically comes at the cost of weight and energy efficiency.

One of the solutions is to separate computational expansive tasks, and offload them to a resource-rich base station, while execution of tasks with strict deadlines remain close to the robotic setup on an embedded processor.

To test this way of working, a 'bridge' is made between the Robotic Operating System (ROS), capable of running its many open-source algorithms and programs on a resource rich platform, and an embedded processor running an embedded application, designed using the real-time capable framework LUNA, developed at the Robotics and Mechatronics group of the University of Twente. This bridge is able to configure ROS-nodes based on initialization commands issued by the embedded application, performing runtime binding to nodes in ROS. Configuring the system based on the embedded application, allows reuse of the bridge in other systems by simply replacing the application.

The implementation of this ROS-LUNA bridge is verified and tested, showing correct runtime binding and communication between the two environments. To show the usability of the bridge a test setup with vision in the loop was made. In this test setup, camera data is send over the network and processed in ROS. The results are send to the embedded system as setpoint for loopcontrollers, controlling the axis of a pan/tilt camera.

These tests show proper functioning of the design and implementation, and give an example on how both environments could be used together.

It is advised to further improve and integrate the designed ROS-LUNA bridge. Mainly the integration of the bridge into TERRA would allow users to connect ROS and LUNA with more ease. It is also recommended to analyse the effects of the delays imposed by long-range communication further, and implement more advanced controllers and a safety layer to control robotic setups using the ROS-LUNA bridge over large distances.

Contents

1 Introduction

In modern robotics, two main trends are distinguishable. The first one is the increasing complexity of the setups: more sensors are added to perceive and interact better with the environment of the robotic setup. These sensors generate more data which should be processed by new and computationally heavier algorithms. For example, a robotic setup could use a camera as sensor to perceive its environment. Computer Vision algorithms run on these images allow the extraction of useful data. The second trend is mobility and energy efficiency: for example, Unmanned Aerial Vehicles (UAV) need to be both lightweight and energy efficient, to allow it to operate as long as possible.

Both trends seem to conflict: executing more computational expensive tasks, increases the processing power of the system, which causes it to consume more energy and increase the weight of the system by the additionally needed powersource. One way to circumvent this conflict, is to offload the computationally expansive tasks to a base station. These tasks are generically Soft Real Time (SRT) (Buttazzo (2011)), meaning that the delays between robotic setup and base station are not disastrous. The Hard Real-Time (HRT) tasks, like controlling the actual motors in the setup, can remain inside the setup on a simpler and energy-efficient embedded processor.

Such a base station can be implemented using a system running the Robotic Operating System (ROS)[1]. ROS is an open-source environment allowing fast development of robotic setups, by using algorithms and programs in a networked structure. At the Robotics and Mechatronics group, a framework for real-time applications has been developed. This framework, called LUNA (Bezemer et al. (2011)), is able to run HRT tasks on embedded devices. This framework is used in a graphical environment called TERRA.

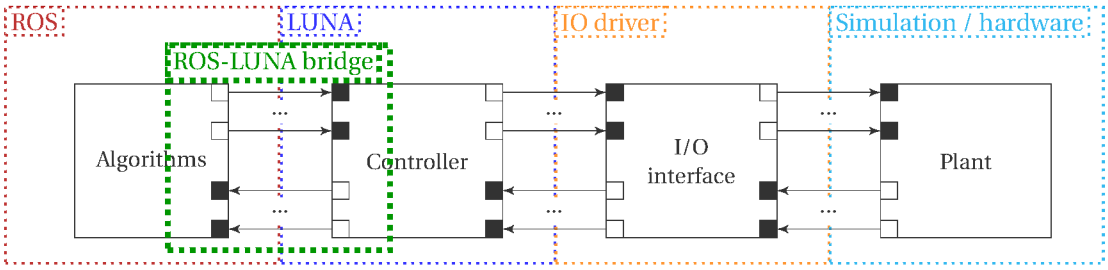

Figure 1.1: The ROS-LUNA bridge in the global scope of the TERRA toolsuite.

Combining ROS and LUNA through a 'bridge', allows separation of SRT and HRT, and allows offloading of computationally expensive tasks from embedded processors to a base station. It also extends the functionality of TERRA: the idea behind TERRA is to have one toolsuite to design the structure a robotic system (or more generic: Cyber Physical Systems) spanning multiple domains and systems (refer to Figure 1.1). The bridge allows connectivity to a platform in which algorithms can be programmed more easily.

An initial version of this bridge between ROS and LUNA was made and presented in Bezemer and Broenink (2015). It showed basic communication by sending a variable from LUNA to ROS.

1.1 Project goals

In this assignment, this bridge is further designed and improved. The bridge should both support versatile data in its communication, and work around the difference in design trajectories

[1] http://www.ros.org/

between ROS and LUNA: the user should be able to use the freedom and adaptability of ROS, while LUNA applications are linked to a precompiled LUNA library. This can be achieved by splitting the implementation of the bridge into two parts: one part is implemented at the embedded side in the LUNA framework, and the second part is implemented at the base stations' side, in ROS. The implementation in ROS is configured through a network connection by the LUNA application: this also allows multiple robotic setups to use the same base station. This implementation is made in such way that it uses so called *runtime binding*: allowing dynamic configuration of the bridge. This runtime binding differs from the normal way connections are setup inside ROS: in general parts in ROS are checked during compile time and generate header-files, which should be included in other parts of the system when communication between them is needed. The definitions in these header-files are not known in the LUNA application, making the LUNA bridge responsible to load the definitions during runtime.

1.2 Report layout

In this report, it is explained how the implementation of the bridge implements the runtime binding. The main body of this report is formed by a paper in chapter 2 written for the CPA conference of 2016 [2]. In this paper, the design, implementation, tests and conclusions of the ROS-LUNA bridge are presented. Separate from the conclusion and recommendations presented in paper, some additional conclusions and recommendations are given in chapter 3.

Further elaborations on the design, implementation and tests of the bridge are presented in appendix A. Furthermore, in appendix B a set of additional appendices are given, containing practical information on how to use the ROS-LUNA bridge and an additional overview containing the derived requirements from the project proposal. Both appendix A and appendix B contain an overview with their subsections listed.

1.3 Reading order

It is advised when the reader is not familiar with CSP, LUNA and TERRA, to read appendix A.1 first.

The preferred reading order is further: the paper in chapter 2, additional elaborations on the design and implementation from appendix A.2, further elaborations regarding testing from appendix A.3, the conclusion and recommendations from chapter 3.

For a quick overview of the requirements and how these where achieved, refer to appendix B.1. For details on how to use the ROS-LUNA bridge, refer to appendices B.2 through B.6.

[2] http://www.wotug.org/

2 Paper: 'Connecting two robot-software communicating architectures: ROS and LUNA'

On the next page, the paper "Connecting two robot-software communicating architectures: ROS and LUNA" is added. This paper is written for the CPA conference (Communicating Process Architectures) [1].

[1] http://www.wotug.org

Submitted to: Communicating Process Architectures 2016
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2016

Connecting two robot-software communicating architectures: ROS and LUNA

W. Mathijs van der Werff and Jan F. Broenink

Robotics and Mechatronics, CTIT Institute, Faculty EEMCS,
University of Twente, The Netherlands

Abstract. Two current trends in modern robotics and other cyber-physical systems seem to conflict: the desire for better interaction with the environment of the robot increases the needed computational power to extract useful data from advanced sensors. This conflicts with the need for energy efficiency and mobility of the setups. A solution for this conflict is to use a distribution over two parallel systems: offloading a part of the complex and computationally expensive task to a base station, while timing-sensitive parts remain close to the robotic setup on an embedded processor. In this paper, a way to connect two of such systems is presented: a bridge is made between the Robotic Operating System (ROS), a widely used open source environment with many algorithms, and the CSP-execution engine LUNA. The bridge uses a (wireless) network connection, and provides a generic and reconfigurable way of connecting these two environments. The design, implementation in both environments, and tests characterizing the bridge are described in this paper.

Keywords. CSP, LUNA, Embedded, ROS

Introduction

Modern robotics rely more and more on data from complex sensors and algorithms to perceive their environment as clear as possible: algorithms like environment mapping, path planning and visual servoing rely on computational-expensive functions to retrieve the desired information from the data of the sensors. These algorithms are generically non hard real-time [1]: for example, when they are used as reference or as setpoint in a control loop. The complexity increases the requirements the computing system needs to have: more memory, more processing power and more energy are needed.

This conflicts with another trend in robotics: the need for more mobile and more energy-efficient setups. These mobile setups, like Unmanned Aerial Vehicles (UAV), have less resources at their disposal, in favour of being light-weight and energy-efficient. These devices are generically powered by batteries and are controlled using embedded processors.

One solution to perform the complex tasks inside a modern robot, is to add dedicated and tailored hardware to perform these complex tasks. This is expensive however, and may not be available. Also, during development of a robotic setup, the developer needs to be able to change the configuration easily, while replacing or modifying custom hardware is time consuming.

Another solution is to split the system into two parts, and use two separate systems to run the tasks. The computationally expensive tasks are offloaded to a base station, while the hard

real-time parts, like loop controllers, remain close to the setup on an embedded processor (refer to figure 1). The Robotic Operating System (ROS) [2] is an open source environment.

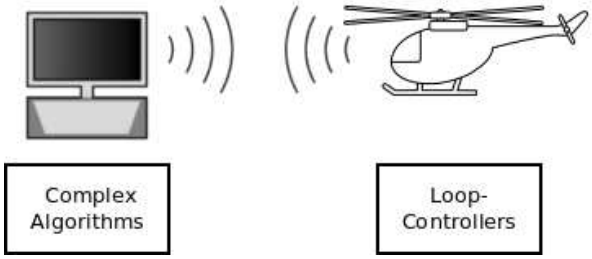

Figure 1. System overview showing separation of tasks over two systems.

ROS is network based, allowing the already available algorithms and implementations of new algorithms and functions to easily connect. It is therefore most suitable to be used in such a base station. The LUNA Universal Network Architecture (LUNA) [3] is a real-time capable framework, developed at the Robotics and Mechatronics group of the University of Twente. This framework is capable to run real-time tasks on (embedded) processors, and is therefore usable to implement the hard real-time tasks.

The main issue in combining code for these two systems is how both environments are used in development: ROS gives the user the ability to easily change its configuration, but needs to run configuration files and has to be recompiled when changes occur. With LUNA, the user builds an application which reuses functionality from the pre-built LUNA library: it is therefore not possible to include definitions generated by ROS, since these differ from system to system, and even over time on the same system, and can therefore not be included in the LUNA library. A method is needed to combine both environments using a more dynamic approach: a method of binding the two systems during runtime is needed.

The design of a bridge between ROS and LUNA is explained in the work-in-progress paper by Bezemer *et al.* [4], describing an initial design and a basic test showing proper functioning.

In this paper, the design of the bridge is further improved and tested. First, some background information is given about LUNA, ROS, the previous version of the ROS-LUNA bridge, and other related work. Then, in section 2, the design and design choices of the improved ROS-LUNA bridge are illustrated. In section 3, tests are described and their results analysed, which prove the proper functioning, show the performance, and demonstrate a typical use of the bridge. Finally, conclusions are drawn about the bridge in section 4.

1. Background

To place the design of the bridge into perspective, background information is given on LUNA, the ROS, and the current version of the ROS-LUNA bridge. Also, some related work and alternative environments is given.

1.1. LUNA

LUNA (LUNA Universal Networking Architecture) [3] is a hard real-time framework, providing support for all kinds of embedded applications. It is component based, allowing parts that are not used to be turned off, resulting in an as low as possible footprint.

LUNA provides a CSP-execution engine, making it able to execute processes according to the Communicating Sequential Process (CSP) algebra [5]. The CSP algebra provides mathematical constructs for scheduling and uses rendezvous channel communication be-

tween these constructs. The resulting schedules can be formally verified (using tooling as FDR3 [6]), making it possible to check correctness and rule out unwanted behaviour like deadlocks and livelocks. These checks can be used to guarantee execution of the processes before their deadlines, making it possible to run hard real-time tasks.

Developing LUNA-based applications is generally done using Model Driven Techniques (MDD), provided by the TERRA (Twente Embedded Real-time Robotic Application) tool suite. TERRA allows easy use of the CSP-execution engine of LUNA, allowing the structure to be drawn instead of programmed by hand. In Figure 2 the architecture of a simple Producer/Consumer example is drawn, with the implementation of each submodel depicted below the component in the architecture.

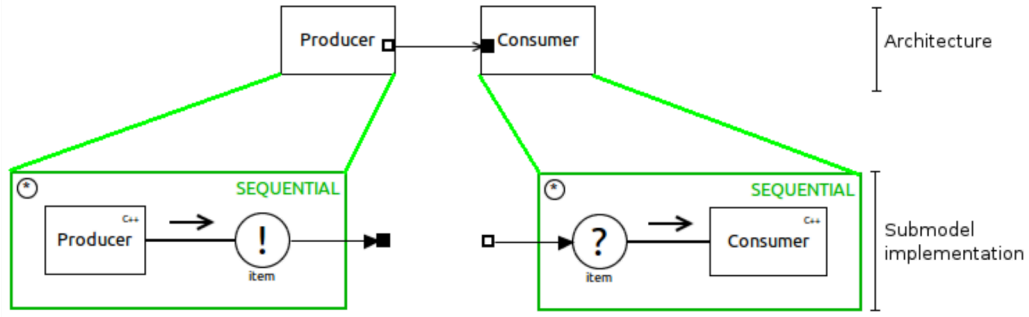

Figure 2. Producer/Consumer architecture and submodels, drawn in TERRA.

Using CSP allows an easy decomposition of the structure of a program into a set of sequential and parallel tasks. Support for more advanced structures (e.g. timed channels, (guarded) alternatives, prioritized parallel) is present, allowing also complex structures to be decomposed. Adding blocks with custom C++ code allows the user to add the functionality of the program to the structure defined with the CSP constructs. Furthermore, embedding converted 20-sim[1] models is supported, allowing for easy implementation of digital controllers.

1.2. Robotic Operating System

The Robotic Operating System (ROS) is a software environment that provides a set of tools to connect multiple parts of a robotic setup. ROS supports a wide range of sensors and algorithms. For the user it is also easy to add new parts, since Python and C++ (amongst others) are supported as programming languages.

The different parts of the setup (called Nodes) interact through a network structure. Communication is based mainly on the publish/subscribe pattern: a node can publish data on a specific topic, or listen to a topic. The exchange of data is done through so-called messages, which are defined by the user in an additional textfile, describing the layout of the data.

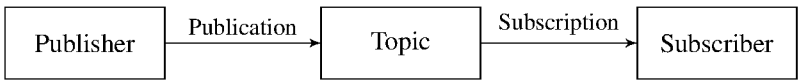

Figure 3. Publish/Subscribe structure in ROS.

When ROS is compiled, these messages are transformed to header files, where the user-defined fields are placed in *structs*. Also, serialization and deserialization functions are added, for sending over the ROS-network as arrays of bytes.

[1] `http://www.20sim.com/`

Using these automatically generated functions makes the communication more robust. To make the communication even more reliable, MD5 checksums of the message type are added. These checksums are checked during runtime, when a Subscriber connects to a Publisher. This allows verification whether both the Publisher and Subscriber use the same message definition, and thus use the same serialization/deserialization functions. This assures the correctness of the received data. Publishers and subscribers are generically instantiated using the message type: in C++ for example a publisher on topic "chatter" with String from the std_msgs package is made using[2]:

```
ros::NodeHandle n;
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter",
    1000);
```

A subscriber listening to this same topic, is instantiated with:

```
ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
```

In the specified callback function ("chatterCallback"), the type of the received message should be specified:

```
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  //Code to handle data from the msg
}
```

When a message type changes (or a new one is added), the ROS environment should be rebuild, to update the changes in these message classes. When a program uses a messagetype, it should also be rebuild, to update the definitions and the checksums.

Due to the complex and reconfigurable structure of ROS, it is not capable to provide hard real-time tasks: the timed execution of a node and the arrival of data cannot be guaranteed, as this depends on too many unknown factors. Most of the time the system will function fast enough however, making ROS suitable for soft real-time tasks.

The ease of use of ROS comes at the cost of more overhead, making it less suitable to run on embedded processors: these processors tend to have less resources at their disposal, in favor of energy consumption, weight and cost.

1.3. Combining ROS and LUNA

The work presented in Bezemer *et al.* [4] is already able to connect an embedded (LUNA based) application during runtime to ROS. Runtime binding to a ROS publisher is performed through the *MessagePublisher* class. This *MessagePublisher* class has a switch construct to determine the variable type of the received variable from LUNA, and generates a ROS *Publisher* with the corresponding message type. This allows to publish on topics with basic message types.

Subscribing to topics is done by using the *TopicListener* and *MessageDecoder* class. The *MessageDecoder* uses raw data of the message, provided by the *ShapeShifter* class present in ROS. The raw data consists of a serialized version of the variable data of the message. Along with the raw data, a message definition (a textstream containing the type and name of each field in the message) is send. This definition is used in the *MessageDecoder* class to iterate through the raw data, until the desired field inside the message definition is found, and the correct bytes can be selected from the raw data. Since the size of a serialized variable needs

[2] http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(c++)

to be known to iterate through the raw data stream, it is only possible to listen to topics with a message type containing standard data types, like $ints$, $bools$ and $strings$.

A communication managing component is used to send and receive data over a TCP connection, as soon as the data is made available. Calculations are presented, showing reduction in bandwidth when multiple variables are packed into one TCP packet.

To connect the received data in LUNA to CSP, a CSP channel is modified to perform its read and write operations through the communication managing component. This modified channel is hardcoded to use the desired datatype of the variable.

The simple tests described indicates correct functioning of the bridge: variables are send from a LUNA application to ROS, and values are returned and received by the LUNA application. Since the bridge only supports basic types, it does not support the full functionality and freedom ROS combined with LUNA could offer. When the bridge is further improved, it could be used in all types of systems: indirectly allowing CSP constructs through the LUNA framework to interact with the real world, by using algorithms and functions present in the open source environment of ROS.

1.4. Related Work

Connecting different (embedded) environments is also done before in other projects.

Unity-Link [7] combines FPGA based controllers with software running on a PC, where ROS is used as middleware. This solution to add real-time control is rather specific: it only works when (re)configurable hardware is present in the device, while many embedded systems favor an embedded processor over programmable logic.

Scholl *et al* [8] combine multiple devices with small resources to form a wireless sensor network, and connect this network to ROS. These devices are programmed to use fixed data structures in the communication to ROS. This is less useful for a bridge between ROS and LUNA, since LUNA should be able to use more dynamic data structures. Furthermore, only the ROS client is used, resulting in a soft real-time environment.

YARP (Yet Another Robot Platform) [9] and OROCOS (Open RObot COntrol Software) [10], [11] are versatile robot middleware environments. Support for both hard- and soft real-time tasks is available, and it supports an extensive way of configuring. It is less suitable for mobile setups however: it has a larger footprint and has therefore higher requirements on the processor.

In Einhorn *et al.* [12] MIRA is presented as a new middleware for robotic applications. Through a custom implementation of so called *reflection* in C++, it is able to optimize the serialization and deserialization processes in the communication between distributed parts of the application, making it faster in terms of latency and computation time compared to other middlewares like YARP and ROS. It lacks a large community, making it less favourable compared to an environment like ROS. Although the middleware is able to run on different environments, it is not designed and tested for real-time purposes, making it less suitable for embedded controllers, and therefore also less suitable as a complete solution for a robotic system.

In Wei *et al.* [13] a real-time extension is made to ROS, called RT-ROS. A multicore system is used in this approach: one part of the cores runs a generic Linux distribution, while simultaneously a real-time Linux distribution Nuttx is running on the other part of the cores. The Nuttx environment is adapted, so it is able to compile ROS nodes. In this setup, a combination of two environments is made on one processor. The used test setup uses a multicore processor and a processor architecture (Intel Core 2 Duo) that is commonly found in desktop PCs, and is therefore less suitable to be used in mobile robotics, limiting the possibilities to use this approach.

The ROSpackage ROSSerial[3] provides a method to connect embedded devices (like Arduinos) to the ROS network. Runtime binding is performed through the *ShapeShifter* class, or using *rospy*, a Python implementation of ROS. The embedded side needs to be informed about the setup of ROS (regarding the message structure) before compilation. This is achieved by including a special set of libraries, which are generated by a script. This increases the overhead, which is a problem in systems with sparse resources [14]. Furthermore, each time a message definition is added or modified, the conversion needs to be redone, which causes the program depending on them to be recompiled. Since LUNA is a provided to the end user a a pre-compiled library, it is therefore not possible to use ROSSerial.

2. Design of the ROS - LUNA bridge

The ROS-LUNA bridge needs to connect the CSP environment of LUNA to the topics of ROS: allowing CSP-channel constructs (Writer/Readers) to send/receive data from an external source located in a ROS topic. Connecting CSP channels to fields in Subscribers and Publishers in ROS should be reusable, to allow easy integration into the TERRA tool suite. Furthermore, support for flexible (re)configuration and versatile data types should be present, allowing reuse of the bridge in future projects.

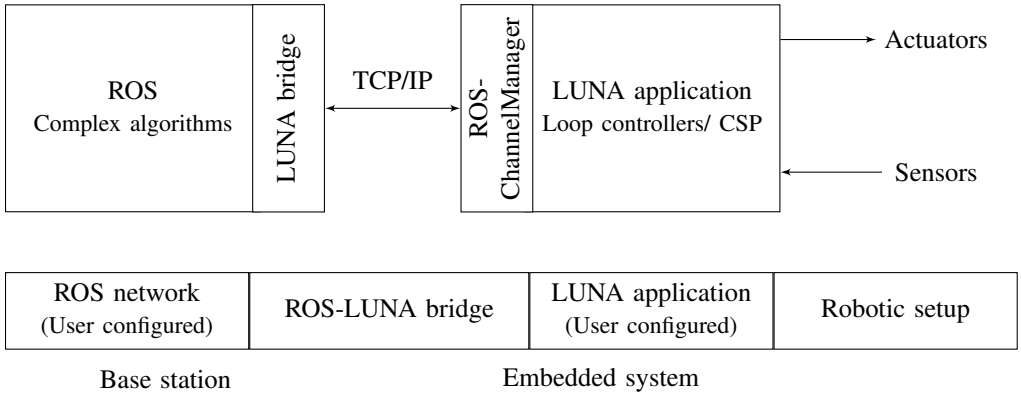

Figure 4. Global overview of the ROS-LUNA bridge.

As depicted in Figure 4, the design of the ROS-LUNA bridge is spread out over three subsystems: an implementation in ROS (LUNA_bridge), an implementation in LUNA (ROSChannelManager), and a link over a TCP/IP network specified by a communication protocol.

2.1. Connection management and Communication protocol

The communication protocol specifies how data is send between ROS and LUNA. A straight forward approach is to make a TCP link between the two sides of the system for each variable, and send each new value in a separate packet as soon as it becomes available. This would lead to too large overhead however: TCP connections were designed to be reused, and the maximum size of a TCP packet (theoretically: 2^{16} bytes, but is limited by the Maximum Transfer Unit [15]. The MTU for ethernet is 1500 bytes) allows combining of variable values in one packet. The communication protocol defines how multiple variables are serialized into one packet, and how their values are retrieved during deserialization. Although widespread serialization methods, like JSON[4] could be used, it would also increase overhead and dependency

[3] `http://wiki.ros.org/rosserial`
[4] `http://www.json.org/`

on third party implementations. A tailored solution is preferred, which reduces overhead by specifically supporting just the communication type of this bridge.

Variables are serialized by placing the type, name length and datalength represented by one byte each in a buffer. In a secondary buffer the name of the variable is added, followed by the variable value represented as byte array. Once the packet needs to be send, both buffers are copied into the TCP packet, preceded by a headerfield, with a predefined layout. This header identifies the type of packet, and the sizes of both buffers. These sizes are used in deserialization: allowing to extract the two buffers from a stream of bytes. With the 3 bytes per variable in the first buffer, the name and data are extracted from the second buffer. Using the name, earlier registered callbacks are called, which will copy the byte array into an actual variable using the size of the received data.

2.2. LUNA-side

Sending to, and receiving data from ROS needs to be usable with CSP constructs offered in LUNA: this allows better integration in TERRA, allowing the end user to use the graphical design environment to design his application. Furthermore, the way how data is send and received is important: writing to ROS might be performed from a hard real-time task in LUNA, and needs to be handled quickly and without locking. Reading data from ROS should lock however: it is of no use to read data when it is not yet available. Integration is possible by using custom code blocks, managing the sending and receiving of data, inside the model in TERRA. Although this would have reduced the changed needed in LUNA and TERRA, it would have been less user friendly, since the user has to copy these code blocks and re-derive the accompanying CSP structure when new models are designed.

Since sending and receiving data has similarities with the CSP writer and reader, a custom channel type (a ROSChannel) was derived to support communication to ROS. This channel is implemented as templated class, making it possible to define the variable type of the channel based on its connected reader or writer. Non blocking writes are performed using a dual buffer: this assures when one buffer is used in sending data over the network, the other one will still be usable to write to from another process. With a user specified interval, the buffers are switched. This allows to reduce bandwidth by grouping variables, and still allows the specification of the maximum time a value is delayed by the buffer.

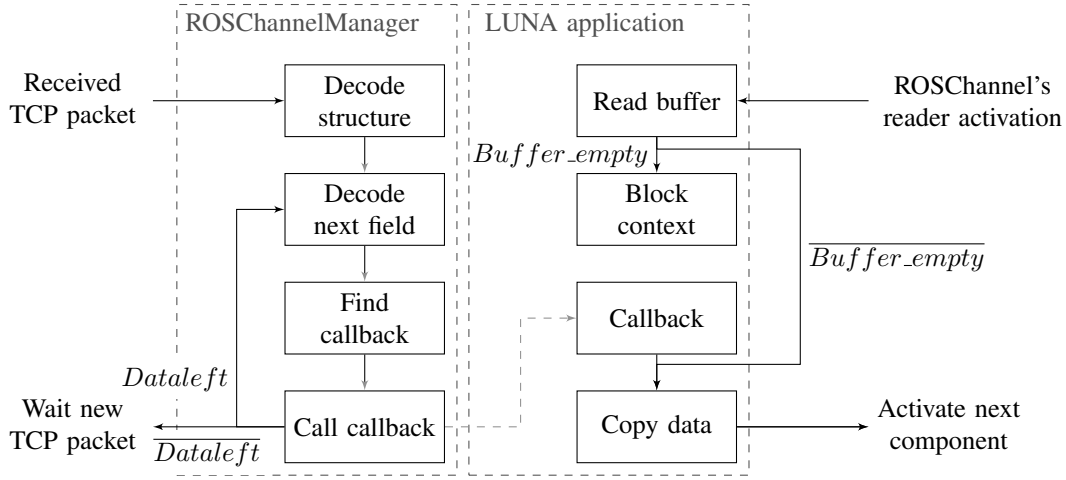

Figure 5. Schematic representation of receiving network data combined with CSP read operations.

A block diagram of the blocking read is depicted in Figure 5. The read operations consist either of directly copying data when it is available, or by placing a callback and blocking the context of the reader. When data is received, the callback is called, unblocking the reader and

allowing the data to be copied. Finally, the next (CSP) component could be activated. Since

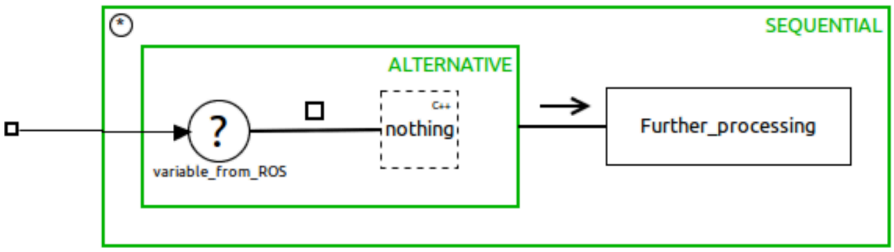

Figure 6. Example of unguarded ALT structure used to perform a non-blocking read on the ROSChannel.

in some cases it is desirable to do a non-blocking read (e.g. reuse an old value when no new value is present), it is possible to compose the reader in an unguarded alternative structure, as depicted in Figure 6. This allows the sequential process to continue when the reader is blocked, by executing an empty model instead. Since multiple ROSchannels could be present in a LUNA application, a single component (called ROSChannelManager, implemented as singleton object) is added to implement the buffers, register and call the callbacks, handle the actual TCP connection and use the defined communication protocol.

2.3. ROS-side

Sending to, and receiving data from a LUNA application needs to be combined with the communication structure in ROS: during runtime, publishers and subscribers need to be made. The message type of these publishers and subscribers need to be configured from the LUNA application: the same bridge could be used in multiple projects with different LUNA applications. A method is needed to bind publishers and subscribers to a messagetype during runtime: normally this is done during compile time, by instantiating the publisher or subscriber object with the message type's class. One way to perform this runtime binding, is to use a code generation tool to make a large switch structure, which combines the name of a messagetype, to the instantiation of an actual object. The tool also need to generate get and set functions, since a message could exist of multiple fields. Using this type of code generation results in a large code file and program, since all possible messages are coded inside it. Also, using code generation adds another step in the design process: each time message definitions change in ROS, the code generation need to be rerun and the compile process of ROS restarted. Another way is to use an interpreted language, like Python. Since the implementation is then also interpreted, it is able to load new classes during runtime and generate objects based on the name of the message type. This reduces performance however: interpreted languages are generically 4 - 5 times slower compared to compiled programs. The reduced perforamance is not ideal in a forward path.

The *ShapeShifter* class in ROS provides a method to publish and subscribe data without a predefined messagetype. It requires however a custom implementation of the serialization and deserialization of the message's variables, and the checksums and message definition need to be set by the user: these are normally specified in the generated header files of the message type. Two classes were derived, performing these actions during runtime. For the publisher, the RuntimeBindingPublisher was derived; for the subscriber the TopicListener was extended. The RuntimeBindingPublisher (RBP) calls a ROSservice in Python when a new message type is desired during runtime: this service is able to load the definition and checksum of this type, and replies it to the RBP. The RBP stores the definition, and the checksum. Furthermore, the message definition is analysed, and added field for field into a map. Using recursion, nested message types are also added. Since this only needs to be

done when new message types are used, the latency introduced by using Python will only occur during runtime. When a new publisher is made, the retrieved data is used to configure a shapeshifter into the right format. The mapped structure of the message is used to store received data from LUNA, and allow when all data for one message is received to serialize the data and publish it.

When a ShapeShifter is used to receive data as subscriber, the received data will consists of raw data (an array of bytes), containing all the data of the message. Furthermore, the definition of the message is also received. The TopicListener implements recursive methods to analyse this message structure, allowing the correct bytes to be selected from the raw data. The methods determine whether a field in the structure is of basic data type (e.g. int, bool, string etc.). When it is not a basic data type, a nested message is found, and recursion is called on the definition of this nested message. This is repeated, until only basic types are found, or the desired field is retrieved. From all the preceding fields, the data type is used to determine the location in the raw message. This allows to select the correct bytes, which are then copied and made available to be send to the LUNA application.

2.4. Overview of the ROS-LUNA bridge

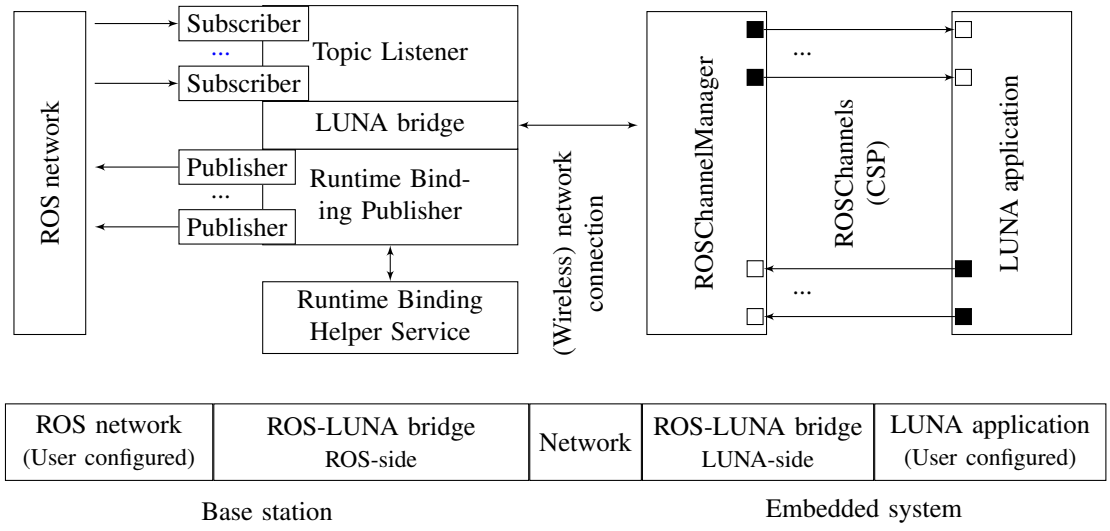

Figure 7. Block diagram of ROS-LUNA bridge.

In Figure 7 a diagram is depicted showing a schematic overview of the bridge using this design. The ROS side adds a series of needed publishers and subscribers during runtime, allowing communication to the ROS network. A Runtime Binding Helper Service is connected as ROS service in Python, allowing the configuration of the Runtime Binding Publisher. The LUNA side of the bridge has a series of CSP incoming (white)- and outgoing (black) ports in the ROSChannelManager, which are able to connect to CSP channels in the LUNA application.

Since the bridge is fully configured during runtime by the LUNA application, hard coded configuration of the bridge is omitted: this allows to reuse the same bridge node in ROS for different LUNA applications. The implementations allows the bridge to be almost invisible to the user: the LUNA application is configured to connect to specific ROS nodes, and the bridge handles this. This results in a clear connection between the algorithms the user uses in ROS, and the CSP structures used in the LUNA application.

3. Testing

Two series of tests are performed on the design of the ROS-LUNA bridge. The first series is used to show correctness and compare performance of the implementation at the ROS side of the system. The second series uses a more complete setup, where the correctness and performance of the bridge is shown using an actual connection between a ROS to an embedded LUNA application through the bridge. Also, a demonstrational setup is described and tested, showing that the bridge is possible to be used in a distributed application, by using both platforms in an area the perform well in: at the embedded site, loop controllers are implemented based on CSP structures, while ROS is used to perform complex algorithms, represented by an image processing algorithm.

3.1. Test 1: Checking runtime binding

To verify the correctness and the performance of the implementation of the runtime binding publisher (RBP), two subtests were designed and executed. The first test verifies the correct serialization during runtime using code generation: a C++ file is generated, which contains code to make a serialized message for each message type present on the system, both for the generic way using a normal publisher and by using the new RBP. The resulting serialized messages are compared, and in three separate lists saved whether the message type serialized correctly, failed, or was unsupported (for example, when it contained an array). The list with failed message types was used to further improve the implementation, until the failed list was empty.

A second test was performed, to compare the different implementations of ROS Publishers. A total of 5 types can be distinguished: the generic ROS Publisher in C++, the generic ROS Publisher in Python, the RBP (both with and without prior stored knowledge about the message type) and a simple version of runtime binding implemented in Python. The test is done by measuring the time needed for initialization, and measuring the interval needed to publish a message for each publisher type. Inside the published message, the intervals from the initialization and the previous publish are stored, allowing an external subscriber node to handle and store the timestamps. An average over 100 samples is taken to measure the time needed for publishing. In one test, the initialization and publishing of 100 samples is repeated 50 times, using a different topic name each time. This test is repeated 10 times: running one large test results in too many topics($10 * 50 * 5 = 2500$) being registered at the ROS core, resulting in the system to crash.

This test results in an average over 500 initializations and 50000 publications of each publisher implementation.

The test is carried out on generic notebook (synopsis of specifications: Intel i5@2.53GHz, 4GB RAM, Ubuntu 15.10, ROS Jade).

The results are depicted in Figure 8. In initialization, the Runtime Binding Publisher in C++ (RBP_{C++}) is slowest: this is due to the call to the external Python helper node. In $\text{RBP}_{C++,2}$, this call is not needed since the messagestructure is reused from a previous call: this results in an initialization time just a little higher compared to the generic C++ implementation. Python is also slower compared to generic C++ Publisher. The additional calls needed to load the message modules during runtime cause the runtime binding version in Python to also be slower compared to the generic Python implementation.

After initialization, it can be seen that both RBP C++ implementations have comparable results for publishing: this is expected, since only the initialization changed and the normal publish call did not. When RBP is compared to both Python implementations, it can be seen that the RBP is faster. Compared to a generic C++ implementation, it is slower however. This is due to additional lookups that need to happen to map the name of a variable to the variable, which are not needed in the generic C++ publisher. From these measurements, it can

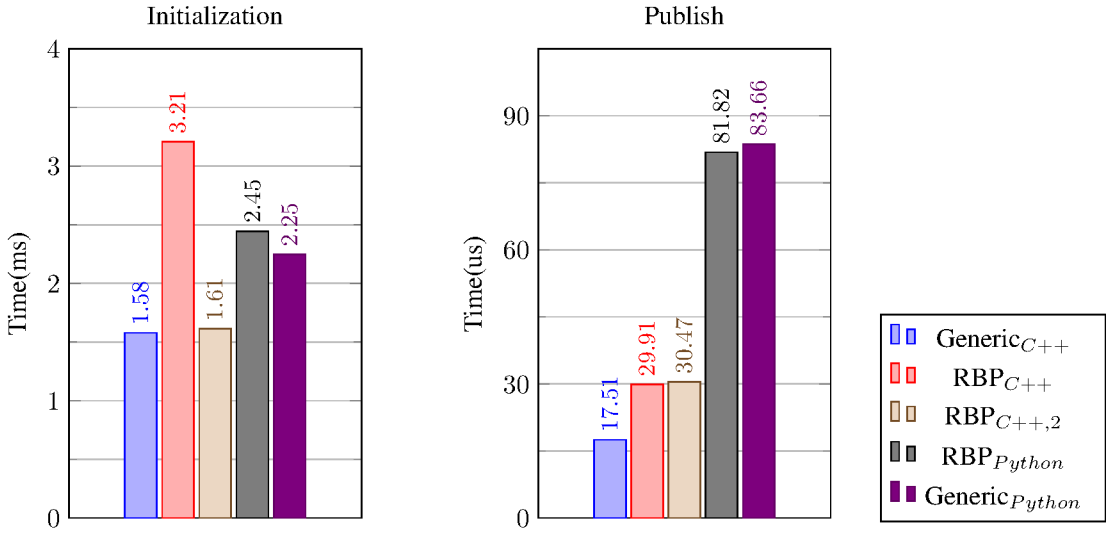

Figure 8. Performance comparison between different Publisher types.

be determined that the Publish function of RBP is between 70-74% slower compared to its implementation in C++, but is roughly 64% faster compared to both Python implementations.

As third test four different implementations of Subscribers are tested: normal Subscribers implemented in C++ and Python, the implementation using the TopicListener and a simple implementation of a runtime binding subscriber in Python. All subscribers use the same message type, a custom type containing a header and two *float64* fields. The test initializes each type of Subscriber 100 times, and measures the average time needed for initialization. A secondary test is started, which publishes 6000 messages at a rate of 200Hz, containing the current time stamp in one of the *float64* fields (refer to Figure 9. Publishing is done distributed over 4 topics (/subscriber_test_1 to /subscriber_test_4), these topics are connected to two nodes, implemented in either C++ (/test_subscribers_cpp) or Python (/test_subscribers_python), where both a runtime binding and a normal subscriber are present and connect to one of these topics. When a message is received, the timestamp is extracted, and compared to the current timestamp. This difference is published on an additional topic (/CPPS_result, /RBS_result, PythonS_result, PythonRB_result). The messages on these topics are received by an analysis node (/test_subscribers_analysis), where they are stored in a CSV file for further analysis. The measured delays consist of the delay imposed by the publisher present in the time stamp generation node (/test_subscribers_timestampgeneration), the delay in the network, and the delay the subscriber types has. Since the publisher and network are the same on average, the measured average delay is usable to identify the performancedatatype difference between subscriber types.

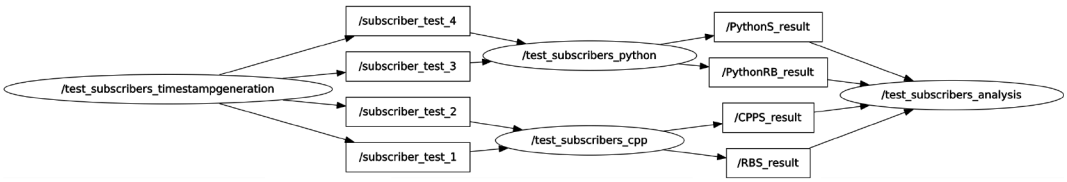

Figure 9. Auto-generated ROS graph of test setup measuring the delays the different types of subscribers impose.

The results of both tests are presented in Figure 10. In initialization both Python implementations seem fastest, followed by the runtime binding implementation in C++. The normal C++ subscriber initializes slowest. Python seems to be able to use optimizations, and the

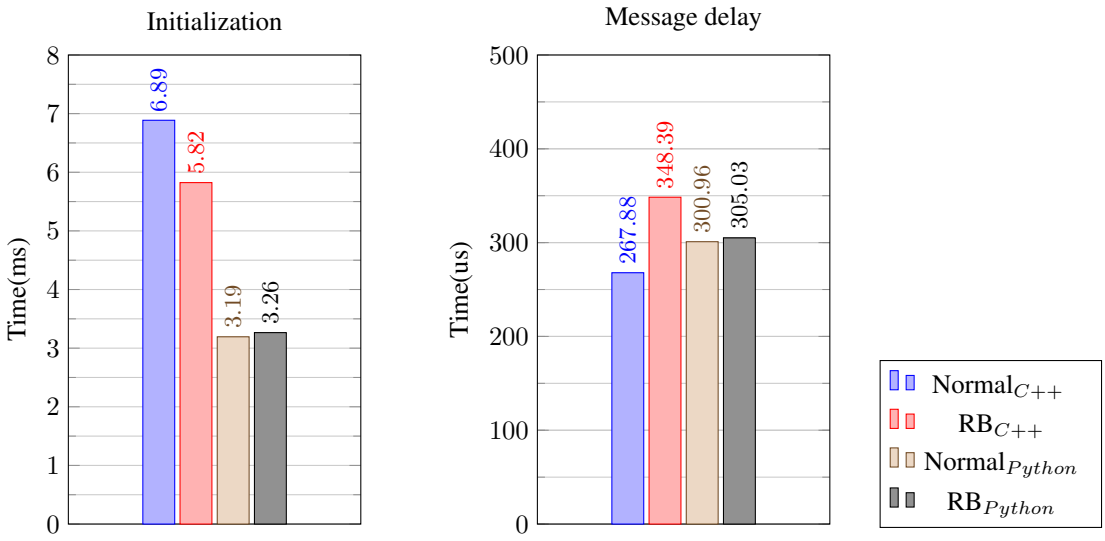

Figure 10. Performance comparison between different Subscriber types.

runtime binding implementation has less work to do in initialization: the normal subscriber has to register its callback initialized with the correct type, while the runtime binding version uses functions at runtime. This is visible in the measured message delays: there the normal C++ implementation is fastest. The runtime binding C++ implementation is slowest: it has to iterate over the description fields to find the correct data that it is listening to. The Python implementations are faster compared to the runtime binding implementation, probably due to optimizations. It is expected, when actual processing is done on the received data, the total execution time of a Python node will be higher, compared to a node in C++.

No large difference are present between both implementations in Python: the runtime binding is rather basic, adding almost no additional delays. Furthermore, Python is already an interpreted environment, allowing easy runtime binding add just a small increase in overhead.

3.2. Test 2: complete system

To test a setup closely related to a real world application, a test setup demonstrating vision in the loop was devised. Refer to Figure 11. It consists of a camera combined with image processing, which will provide feedback about the state of the plant to the controller. Data from the controller is send to a visualization node (e.g. using rqt_plot) to inform the user about the state of the system. Using the physical location of a node and whether it is real-time or not, a mapping is performed, dividing the system over ROS and the embedded system.

The same notebook named in test 1 is used as resource rich platform. As embedded system, a board (called RaMstix) containing a Gumstix Overo Fire[5] module with Linux 3.2.21 and Xenomai patch 2.6.3 is used. A 100MBit/s dedicated network is used in most tests, where the notebook is configured both as DHCP server and NTP[6] server, allowing time-synchronization between the two platforms.

3.2.1. Initialization

The first part of the test is to determine whether the initialization is correct. ROS nodes are started that will perform visualization (*ROS_monitor*) and a node containing the image processing (*ROS_imageprocessing*). The *ROS_monitor* node receives a message type containing a Header and 3 $float$ values. The *ROS_imageprocessing* publishes a message type containing

[5] `https://www.gumstix.com/`
[6] `http://www.ntp.org/`

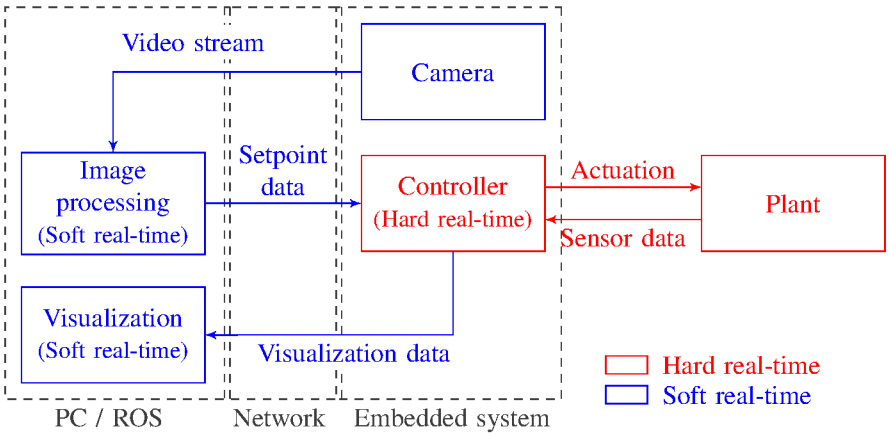

Figure 11. Block diagram of a vision-in-the-loop system distributed over two systems.

a Header and two $float$ values containing setpoints for the plant. Alongside these two nodes, the luna_bridge node is running accompanied by the rlb_helper node, containing the helper node to perform runtime binding. This setup results in the (simplified) graph depicted in left in Figure 12. The LUNA application on the embedded system is configured to send initialization instructions to let the luna_bridge node connect to the two setpoint fields inside the ROS_imageprocessing node, and to make publishers for the ROS_monitor node. When these commands are received, it results in the structure depicted right in Figure 12: the nodes are now connected.

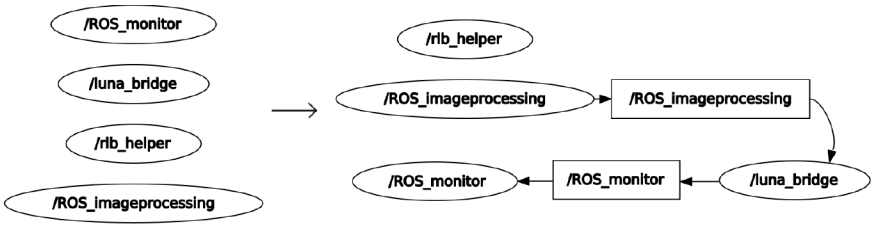

Figure 12. Auto-generated ROS graphs showing node overview before (left) and after (right) the LUNA application connects.

3.2.2. Timing analysis

A second test is performed to analyse the timeliness in the different parts of the system. To perform this, the LUNA application is configured to receive values from the *ROS_imageprocessing*, store these values and reply them in soft real-time. Parallel with this task, a hard real-time task with higher frequency is performed, emulating the controller. Since timeliness is analysed, no actual controller and no plant is connected to the setup. The timestamps from these actions are saved for further analysis. The nodes running in ROS also store the timestamps. For better repeatable test, the camera on the embedded system is replaced with a videofile, which is streamed from the embedded system using gstreamer. The stream is converted to a virtual webcam at the PC, and used in the ROS_imageprocessing node. This structure is depicted in Figure 13.

Only the LUNA_application block is implemented in LUNA, the other parts are just representations of the different links present in the setups. The frequency of the HRT task is set to 500Hz, and the frequency of writing packages to ROS is set to 62.5 Hz.

Inside the LUNA application (refer to Figure 13), three sequential processes are composed inside a PriPar setting. The hard real-time task (HRT_TASK) receives highest priority.

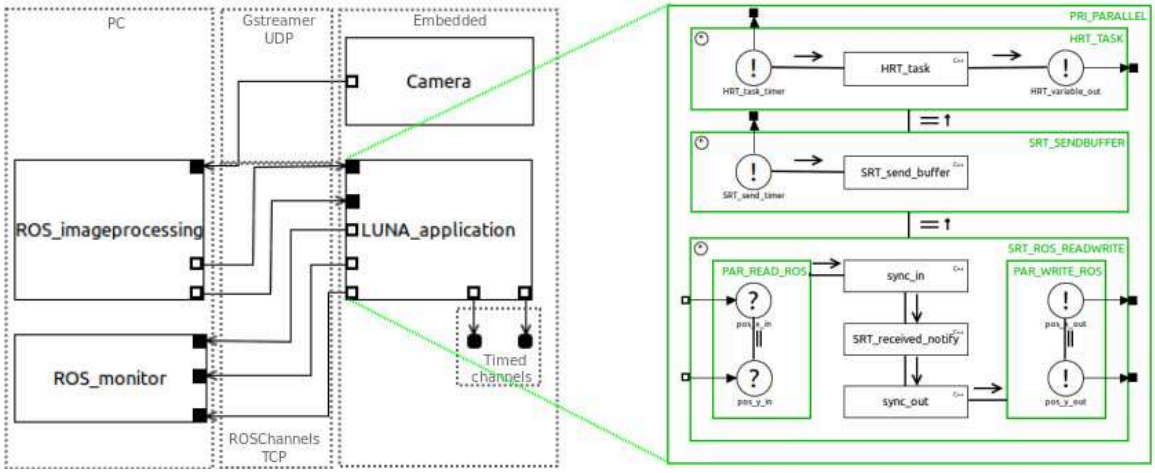

Figure 13. Overview of the total system, drawn in TERRA. Implementation of the CSP-based application and distribution over systems are added for clarity.

Inside this process, a timestamp is recorded, allowing to measure the frequency of the process, and the observation of the deviation in start time (jitter). To make synchronization of measurement data over multiple processes easier, also a unique value is written to the output buffer using the HRT_variable_out variable. The period of this process is controlled through the first writer, which is connected to a TimerChannel. This TimerChannel is activated after its specified period, letting the writer at the start of the process wait until the period indicates the process should start.

The second process (SRT_SENDBUFFER) is the process which controls when data should be written to ROS. It would be possible to make this write conditional (where a condition checks whether a write is needed, e.g. when there are a certain amount of variables present in the buffer), but for simplicity a TimedChannel is used again.

The third process (with lowest priority, SRT_ROS_READWRITE) asynchronously receives values from ROS using two readers. These readers are connected to ROS using the ROSChannels, and receive the X and Y position from the image processing node. The readers are placed in a Parallel composition, and the received values are stored in intermediate variables. When both readers are finished, a code block copies these intermediate variables to the actual variables. This assures synchronized update of variables originating from the same ROS message.

After receiving these values, the time stamp is recorded, and the same values are written back to ROS using writers connected to the ROS monitor node. This allows the measurement of the round-trip time.

The timestamps at the ROS side of the setup are also recorded. The time stamp when the X and Y position are published is recorded, and the time when the ROS monitor receives a value is monitored. Using the values and order of the data in the messages, it is possible to determine the delays in the system. Analysing the difference in start time (ΔT) between two successive executions, allows to measure the jitter (J).

Since different frequencies are being observed, the jitter of different periods needs to be compared relative to their period:

$$J = |\Delta T - \overline{\Delta T}|$$

$$J_{relative} = 100\% * \frac{J}{\overline{\Delta T}}$$

In Table 1 the results are depicted of these jitter measurements.

Table 1. Jitter measured at multiple parts of the setup.

	HRT_task	SRT_send_buffer	SRT_received_notify	ROS_imageprocessing	ROS_monitor
$\overline{\Delta T}(ms)$	20.0	16.0	66.7	66.7	66.6
$std(\Delta T)(ms)$	0.0635	0.0730	15.2	1.97	17.6
$\overline{J}(ms)$	0.0530	0.0598	12.2	1.58	14.5
$\overline{J_{relative}}$	0.265%	0.373%	18.3%	2.38%	21.7%

Table 2. Delay measurements.

	ROS_imageprocessing → SRT_receive_notify	SRT_receive_notify → SRT_send_buffer	SRT_send_buffer → ROS_monitor	Total RTT
Average (ms)	15.5	13.4	2.6	31.5
Stdev (ms)	10.0	3.3	4.6	11.7
Max (ms)	76.6	15.2	26.6	89.3
Min (ms)	5.5	0.5	0.5	9.8

The results show, that the HRT task (HRT_task) has the least jitter: 0.265%. The SRT task which sends the buffer (SRT_send_buffer) also has has a low value for the jitter: 0.373%. These two tasks are purely located on the embedded system inside the LUNA application, and are activated by a TimedChannel: therefore the low jitter complies with the expectation. The image processing (ROS_imageprocessing) is running on a non real-time PC, and therefore has higher jitter. When the data is send over the network, this jitter increases: the process that receives the data (SRT_received_notify)) has a jitter of 18.3%. Sending the data back to the ROS monitor introduces again an increase in jitter: the visualization node has a relative jitter of 21.7%.

Both the increase in jitter when data is send over the network, and the high jitter in the execution of the imageprocessing show the need for a combined setup, where a real-time capable framework is used for the real-time tasks.

The delays between three different parts of the system are interesting: the delay between publishing the results from the image processing and receiving these values (ROS_imageprocessing → SRT_receive_notify), the delay between receiving the values and sending values back (SRT_receive_notify → SRT_send_buffer), and the delay between starting transmission from LUNA and receiving them in the ROS monitor (SRT_send_buffer → ROS_monitor). Refer to Table 2.

In this setup, there is an average round trip time of 31.5 ms. The largest part from this delay is present in sending from the image processing node to LUNA. The second largest delay is present between receiving and returning the values inside LUNA. This occurs due to the buffering: data is buffered for 0.016 seconds. When data arrives at the start of this period, it has to wait for the whole period before it is send back. The maximum delay in this test is 15.2 ms, which is within this 16 ms period. Sending data back to ROS is faster than receiving: on average 2.6 ms is needed to send data back. The delays have a large standard deviation. This coincides with the measured jitter in the previous test: the deviations in the network makes the jitter increase inside the nodes.

3.2.3. Controlling a robotic setup

In the next test, the LUNA_application from the previous test was modified. The hard real-time task was replaced with a controller, and connected to a real robotic setup. This setup, named JIWY, is a pan/tilt camera controlled by two motors. The LUNA application executes the controller at a rate of 100Hz, for which the controlloops where derived. The architecture is changed, to fit the new controllers (PanPositionController and TiltPositionController) and a

block to interface with the IO of the encoders and the PWM of the motors. Refer to image 14 A block is added to send data to ROS after a specified time, and a block to generate setpoints

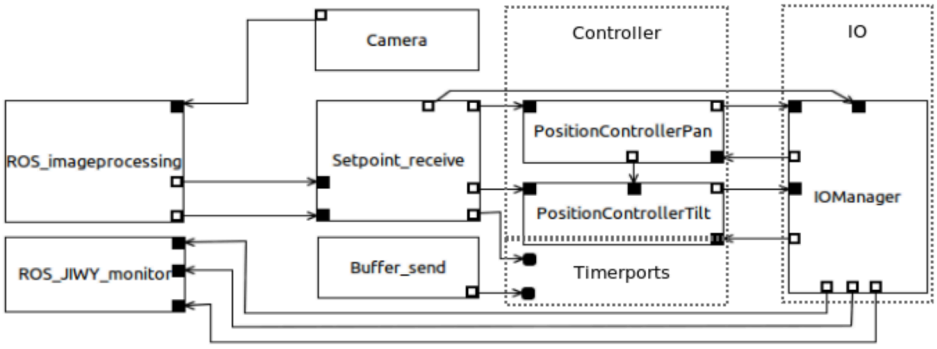

Figure 14. Architecture of setup to control a JIWY setup, drawn in TERRA

is added. Generating these setpoints is done at 100Hz, and uses the last received setpoints from ROS, allowing the system to easily update the setpoints, without the need to wait for non real-time data from ROS (Figure 15). The last received setpoint values are updated in var_sync, assuring synchronized update of the pan and tilt setpoints. The controllers will wait until these setpoints are placed on their channels, causing the controllers to also run at 100Hz.

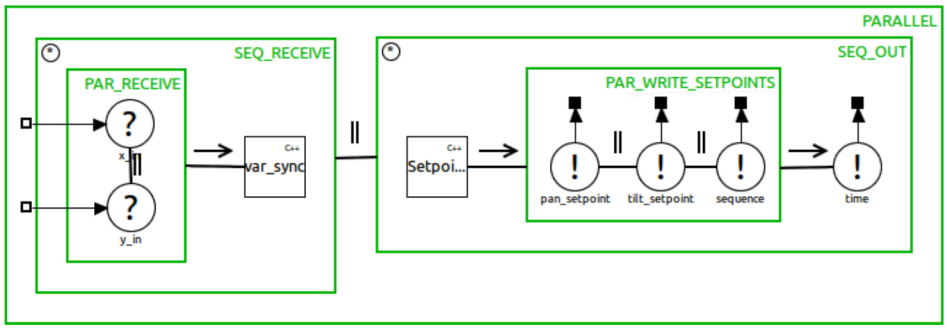

Figure 15. CSP diagram for generating setpoints and receiving new setpoints from ROS.

3.2.4. Setting orientation of JIWY from ROS

Using the same videostream as in the timing analysis, it was possible to let the JIWY setup follow the same trajectory as the green dot present in the videostream.

The setpoint and encoder values in pan direction are fairly similar: some small settling effects are present on the encoder values. A series of setpoints is set, and the controller can overshoot this setpoint due to its integral action. It tries to steer back to the setpoint, until the next setpoint arrives. Refer to Figure 16.

A larger error is present in the tilt direction: although the same pattern is followed, a scaling error is present. This is caused by an inproperly tuned controller, which has a DC gain.

3.2.5. Object tracking using JIWY

An additional test was done by slightly modifying the previous test: the filestream was replaced by the actual camera in the JIWY setup. Since the image data now also will change due to the rotation of the camera, the image processing was adapted to publish the differ-

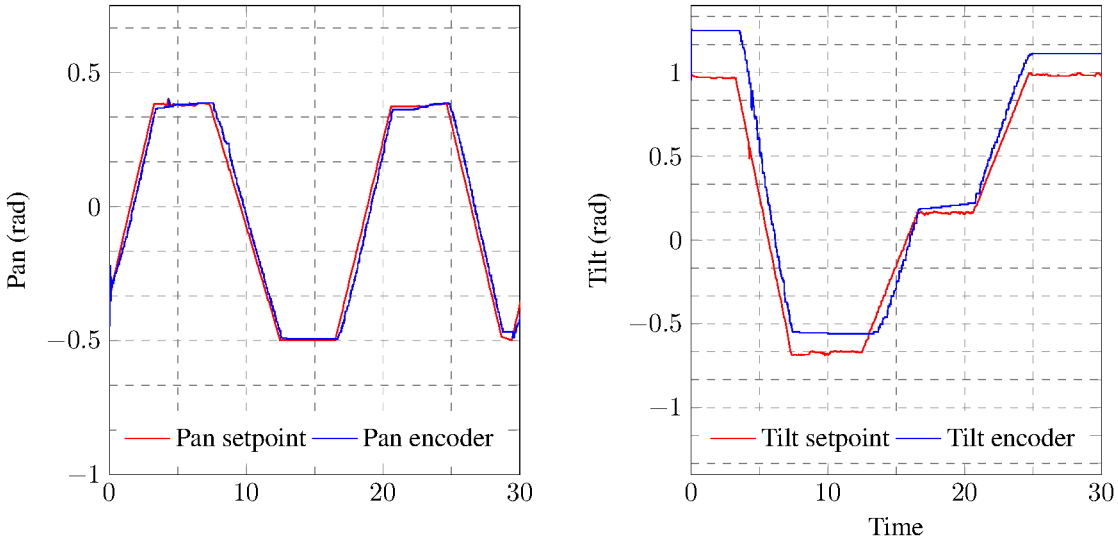

Figure 16. Pan and tilt setpoint from image processing versus the encoder values.

ence between the location of a green blob and the center of the frame. Furthermore, the network link was also replaced by a wireless one, causing delays from the network to be less predictable.

Using this setup, it was possible to follow a moving green dot present in the cameras view. A graph depicting the pan and tilt setpoints compared to the pan and tilt encoder values is presented in Figure 17.

Since the difference in location is published to the robotic setup, the summation of this difference is compared to the encoder values.

As seen in Figure 16, both pan and tilt seem to follow the calculated setpoint roughly: the added delay combined with the incremental update of the setpoint, which is calculated by taking the difference of the location of the green blob and the center of the frame, results in a delayed and smoothed response. Measuring the location of the green blob with respect to the center of the frame, effectively adds an additional P-type controller over the whole system, which includes the delay caused by the network. When delays become too high, this could reduce the stability of the system (refer to ten Berge *et al.* [16]). Since all setups that use a wireless or long distance connection could suffer from these type of delays, it is not possible to counter this effect inside the bridge. It is better to use a more complex type of controller, which holds its stability even with uncertain delays: for example by adding passivity layers and an energy balance for safety, as proposed in Franken *et al.* [17].

4. Conclusion

In this paper, a way two combine two different environments is proposed, implemented and tested. The implementation allows to connect the Robotic Operating System with LUNA, a real time CSP-execution framework. The implementation is made in such way that it is reusable in future applications, by supporting a high degree of freedom through the support of basic data types, and the runtime binding to arbitrary ROS message types during runtime. Combining ROS and LUNA allows to use both systems in the area they perform well: ROS has a lot of algorithms and a large community, while LUNA based applications are able run in real time on an embedded system, and allow the execution of CSP. Furthermore, combing these two environments allows to offload parts of the software of a robotic setup to a basestation: this allows the processing inside the robotic setup to remain lightweight and more energy efficient, while complex algorithms could still be used.

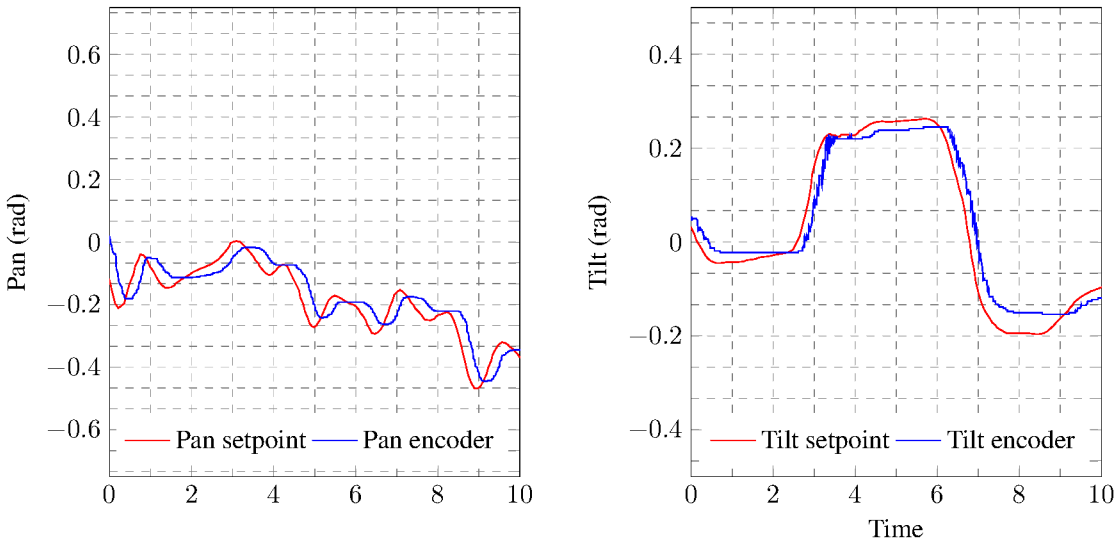

Figure 17. Pan and tilt setpoint from image processing versus the encoder values.

Tests show that the implemented runtime binding is slower compared to a generic C++ publisher: this is as expected, since additional steps needed to perform runtime binding were added. The implementation is faster compared to the Python implementation, showing the favour of using compiled code. When simple runtime binding subscribers are tested, it appears that the Python implementation is faster, compared the runtime binding subscriber. This is probably caused by optimizations present in the Python implementation, allowing simple data types to be received faster. When the implementation is combined with other parts into a larger application, an compilable environment is preferred, as the other parts will benefit from compilation. Verification tests shows correct serialization of the messages during runtime, and allow to test whether a ROS environment contain message types that are not usable yet.

A test setup closely related to a real world application, namely controlling a robotic setup using vision, shows correct functioning and the usability of the bridge: a pan/tilt camera is connected to an embedded system, which streams the camera data over a (wireless) network to a resource rich platform running ROS. The image processing in ROS detects the location of a green dot, and send setpoints through the ROS-LUNA bridge back to the embedded system, which uses these setpoints to update the setpoints in the controller. The controller uses these setpoints to move the pan and tilt axis to the correct orientation, and send data back to ROS, allowing visualization of the state of the setup for the user. Combined this resulted in a cyber-physical system tracking an object. The added delay by the network causes the motions to be non-ideal: although at no point control was lost over the setup, the setpoints do not exactly match the systems response. When delays become too high, it might lead to instabilities however. Since all long range communication will suffer from these type of delays, it is advised to make a more advanced controller, by adding a passivity layer and an energy balance. Such systems have proven to remain stable, even when unpredictable network delays are present in a setup.

Currently, the system is partly designed in the graphical environment TERRA: parts of the generated code are modified after code generation to use this new LUNA bridge through the new and improved ROSChannels. These channels are setup in such way, that they can further be integrated in the TERRA tool suite. This increases the ease-of-use for the end user: he will be able to design the structure of the robotic setup in one tool, even when it spans multiple environments. The runtime binding provided by the ROS-LUNA bridge, used to connect to an arbitrary ROS topic during runtime and without having upfront knowledge

about the message definition, is reusable for other embedded systems as well, without the use of LUNA. This allows future expansion of ROS with embedded devices, when these devices (e.g. microcontrollers) are not able to run LUNA.

References

[1] G.C Buttazzo. *Hard real-time computing systems*, chapter 1, pages 1–13. Springer, 3th edition, 2011.

[2] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A.Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[3] M. M. Bezemer, R. J. W. Wilterdink, and J. F. Broenink. LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework. In P.H. Welch, A. T. Sampson, J. B. Pedersen, J. M. Kerridge, J. F. Broenink, and F. R. M. Barnes, editors, *Communicating Process Architectures 2011, Limmerick*, volume 68 of *Concurrent System Engineering Series*, pages 157–175, Amsterdam, November 2011. IOS Press BV.

[4] M. M. Bezemer and J. F. Broenink. Connecting ros to a real-time control framework for embedded computing. In *2015 IEEE 20th Conference on Emerging Technologies and Facotry Automation*, pages 1–6. IEEE, September 2015.

[5] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[6] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A. W. Roscoe. Fdr3: a parallel refinement checker for csp. *International Journal on Software Tools for Technology Transfer*, 18(2):149–167, 2015.

[7] A. B. Lange, U. P. Schultz, and A. S. Sørensen. Unity-link: A software-gateware interface for rapid prototyping of experimental robot controllers on fpgas. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, Tokyo, Japan, November 3-7, 2013*, pages 3899–3906, 2013.

[8] P. M. Scholl, M. Brachmann, S. Santini, and K. Van Laerhoven. Integrating wireless sensor nodes in the robot operating system. In A. Koubaa and A. Khelil, editors, *Cooperative Robots and Sensor Networks 2014*, volume 554 of *Studies in Computational Intelligence*, pages 141–157. Springer Berlin Heidelberg, 2014.

[9] G. Metta, P. Fitzpatrick, and L. Natale. Yarp: yet another robot platform. *Int'l J. on Advanced Robotics Systems*, 3(1):043 – 048, March 2006.

[10] H. Bruyninckx, P. Soetens, and B. Koninckx. The real-time motion control core of the Orocos project. In *Robotics and Automation, 2003. Proceedings. ICRA '03. IEEE International Conference on*, volume 2, pages 2766 – 2771 vol.2, September 2003.

[11] H. Bruyninckx. Open robot control software: the OROCOS project. In *Robotics and Automation (ICRA), 2001. IEEE International Conference on*, volume 3, pages 2523 – 2528. IEEE, 2001.

[12] E. Einhorn, T. Langner, R. Stricker, C. Martin, and H. M. Gross. Mira - middleware for robotic applications. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2591–2598, Oct 2012.

[13] Hongxing Wei, Zhenzhou Shao, Zhen Huang, Renhai Chen, Yong Guan, Jindong Tan, and Zili Shao. Rt-ros: A real-time {ROS} architecture on multi-core processors. *Future Generation Computer Systems*, 56:171 – 178, 2016.

[14] André Araújo, David Portugal, Micael S. Couceiro, and Rui P. Rocha. Integrating arduino-based educational mobile robots in ros. *Journal of Intelligent & Robotic Systems*, 77(2):281–298, 2014.

[15] J.F. "Kurose and K.W." Ross. *"Computer Networking: A top down approach"*, chapter 3.

[16] M. H. ten Berge, B. Orlic, and J. F. Broenink. Co-simulation of networked embedded control systems, a csp-like process-oriented approach. In *Proceedings of the IEEE Int'l Symposium on Computer Aided Control Systems Conference, CACSD 2006*, pages 434 – 439. IEEE Control Systems Society, 2006.

[17] M. C. J. Franken, S. Stramigioli, R. Reilink, C. Secchi, and A. Macchelli. Bridging the gap between passivity and transparency. page 36. Robotics: Science and Systems V, Seattle, USA, 2009.

3 Final conclusion and recommendations

In this work a way to combine two different environments is proposed, implemented and tested. ROS allows many complex algorithms to be used in soft real time, while LUNA offers the capability to design hard real-time applications for embedded systems. The implementation of the bridge between these two environments allows versatile communication and it is designed to be reusable in future systems.

The implementation takes the different design trajectories of ROS and LUNA into account: ROS is a versatile environment, while LUNA applications are build using a pre-compiled library: it is therefore not known upfront what the exact definitions in ROS are in the LUNA applications. To work around this problem, runtime binding is implemented. This allows the LUNA application to connect to ROS topics by just specifying the names of the message type and the name of the topic. Using this method, also allows reusability of the bridge: the same bridge is usable when another LUNA application connects.

3.1 Recommendations

Further optimizations are possible, to increase the usability of the ROS-LUNA bridge further. Integration of the ROS-LUNA bridge into TERRA would allow users to define systems distributed over LUNA and ROS with more ease.

The implementation could be optimized further: the *TopicListener* scans each received message until it finds the desired field, and counts how many bytes in the received data need to be skipped to select the correct data. Optimization is possible by caching the definition: this caching is hard to implement however, since fields with dynamic size could be present in a message, which need to be accounted for.

Increasing the complexity of the supported ROS messages in the RuntimeBindingPublisher, by adding support for arrays would also allow for more setups to be usable with the bridge: it would then be possible to read camera data in the LUNA application and send the frame as array of pixels to ROS through the bridge. This reduces needed configuration at the embedded system: in the current setup a separate Gstreamer application needs to be started in the current implementation. However, support for arrays also means the extension of the supported data types in TERRA: only single values can currently be used on a channel.

Monitoring the delays introduced by long range communication should be analysed further, and compensated for by using more advanced controllers (ten Berge et al. (2006)). Adding safetylayers (Franken et al. (2009), Brodskiy (2014)) in the underlying structure of the ROS-LUNA bridge allows the user to implement one type of these advanced controllers.

A Appendix

As extension to the paper presented in chapter 2, appendices are added in this chapter to further elaborate on the design, implementation and testing of the ROS-LUNA bridge. Furthermore, in appendix B information is added on the practical usage of the ROS-LUNA bridge.

In appendix A.1 an introduction in CSP, LUNA and TERRA is added for the novice reader in these subjects.

In appendix A.2 additional remarks and further elaborations on the design and implementation of the ROS-LUNA bridge are appended.

In appendix A.3 additional measurement data and tests are appended.

Furthermore, in appendix B a set of additional appendices are presented.

A.1 Introduction into CSP, LUNA and TERRA

A small introduction to CSP, LUNA and TERRA is given below. Using an example, the combination of CSP, LUNA and TERRA is further explained.

A.1.1 Communicating Sequential Processes

Communicating Sequential Processes (CSP) is a language to formally describe process and interaction patterns in concurrent systems. It allows to define a process' structure in a set of of smaller sub-processes. These processes are composed in mainly Sequential and Parallel structures. Using this algebra allows formal verification of the process: for example, checks for dead-lock and live-lock could be performed.

Communication between processes is implemented using channel communication: this allows one process (the Writer) to use a write operation on a channel, while on the other side of the channel a read is performed in another process (by a Reader). The communication is based on rendezvous: both the writer and reader should be present for the transaction of data to succeed. Otherwise the components will wait until the other component is available. This assures synchronisation between multiple parallel processes, and allows data to be passed between the processes.

A.1.2 LUNA

LUNA Universal Networking Architecture (LUNA)[1] (Bezemer et al., 2011) is a real-time capable framework, also providing a CSP execution engine. It is component based, has support for different operating systems(Linux, QNX, Xenomai(partly) and Windows(partly). It supports hardware with the x86, x64 and ARM architectures. Due to its small footprint, is is able to run on embedded processors.

A.1.3 TERRA

The Twente Embedded Real-time Robotic Application (TERRA)[2] is an environment allowing the design of robotic applications. TERRA has an graphical interface, allowing the user to define the structure of his application. Graphical tools are present to construct submodels using CSP symbols. Code generation allows these submodels to be converted to code, using the LUNA library as execution engine. This allows easy development of CSP based models. Code generation to machine readable CSP code is also possible, making it possible to check a submodel with formal tools like FDR3.

A.1.4 Example of designing a model with CSP, LUNA and TERRA

An example is given to further explain CSP, LUNA and TERRA.

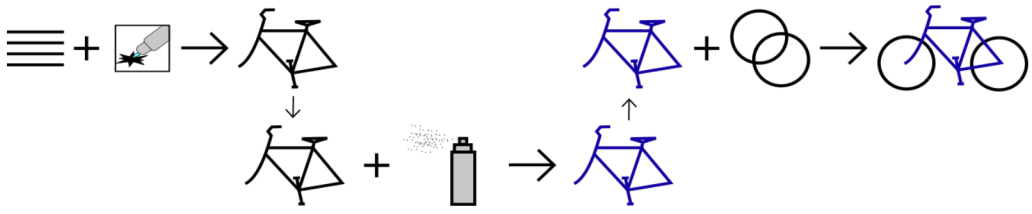

Figure A.1: Simplified production line of a bicycle: two mechanical operations (welding and adding wheels) with in between a painting operation.

[1] LUNA website: https://www.ram.ewi.utwente.nl/ECSSoftware/luna.php
[2] https://www.ram.ewi.utwente.nl/ECSSoftware/terra.php

Imagine a simple production line of a bicycle. Refer to Figure A.1. The production consists of three separate steps: weld the frame, paint the frame and attach wheels. These tasks are separated over two processing units: one unit is able to perform mechanical operations (e.g. the welding and attaching the wheels), the second unit is able to paint the frame. The execution order is important: first the frame needs to be welded, before paint can be applied. The wheels can only be added in the last stage, since they do not need to be painted.

Since the sequential operations are distributed over two processing units, and the operations are performed on the same product, it is import that the processing units are synchronised and wait for their turn to perform their task. This example is modelled in the graphical environment of TERRA, using submodels and CSP for synchronisation. Refer to Figure A.2.

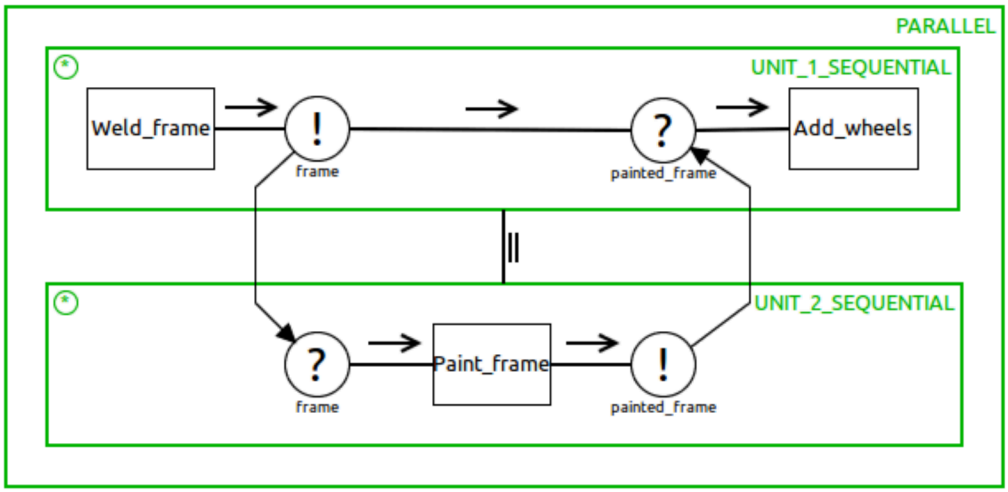

Figure A.2: CSP structure modelling a bicycle production line.

The two processing units are represented as two parallel processes(UNIT_1_SEQUENTIAL and UNIT_2_SEQUENTIAL), each containing a set of sequential steps. Since the processes are in a parallel construct, both will start directly as soon as the process is activated. When UNIT_1_SEQUENTIAL starts, it will begin by executing the task which will weld the frame. While the frame is being weld, UNIT_2_SEQUENTIAL will not be able to start painting, as the frame is not ready. To let the execution of the process halt, a read is performed through a reader (marked by a (?)): this symbolises the request of the welded frame. As long as it is not ready, the Reader will block the process.

When the welding is finished in UNIT_1_SEQUENTIAL a writer (indicated by (!)) is executed, placing the frame on the channel. Since the reader on this channel is already waiting, it is unblocked, allowing both UNIT_1_SEQUENTIAL and UNIT_2_SEQUENTIAL to continue. In UNIT_1_SEQUENTIAL, the painted frame is needed to assemble the wheels: a read using a reader on a channel makes the process wait for the painted frame. In UNIT_2_SEQUENTIAL, the received frame will be processed by the Paint_frame node.

Once this process finished, it writes the painted frame on the rendezvous channel to UNIT_1_SEQUENTIAL through a writer. Both processes are again able to continue. Since no more steps are present in UNIT_2_SEQUENTIAL, it restarts: this is indicated by the 'recursion' mark, depicted by a (*) in the process. In UNIT_1_SEQUENTIAL, the last step is activated (Add_wheels). When this step is finished, the bicycle is complete. The UNIT_1_SEQUENTIAL process is started again, allowing the next bicycle to be made.

When this setup is simulated on a single core system, activation order of parallel process is non-deterministic. Using the rendezvous-channel communication, allows to specify the wanted

execution order: the readers and writers block the execution of the sequential processes until their counter part unblocks them, indicating their data is available.

When the code generation in TERRA is started, classes are made for each submodel and for the mainmodel. The main code constructing the mainmodel using the submodels and the LUNA library is depicted below. It is added to show the work done by code generation to convert a model, making it able to use the LUNA library.

```
MainModel::MainModel() :
    Parallel(NULL)
{
  SETNAME(this, "MainModel");

  // Initialize channels
  mya_writer_to_a_readerChannel = new UnbufferedChannel<int, One2In, Out2One>();
  myb_writer_to_b_readerChannel = new UnbufferedChannel<int, One2In, Out2One>();

  // Initialize model objects
  myAdd_wheels = new Add_wheels::Add_wheels();
  SETNAME(myAdd_wheels, "Add_wheels");
  myPaint_frame = new Paint_frame::Paint_frame();
  SETNAME(myPaint_frame, "Paint_frame");
  myWeld_frame = new Weld_frame::Weld_frame();
  SETNAME(myWeld_frame, "Weld_frame");
  mya_reader = new Reader<int>(&frame, mya_writer_to_a_readerChannel);
  SETNAME(mya_reader, "a_reader");
  mya_writer = new Writer<int>(&frame, mya_writer_to_a_readerChannel);
  SETNAME(mya_writer, "a_writer");
  myb_reader = new Reader<int>(&painted_frame, myb_writer_to_b_readerChannel);
  SETNAME(myb_reader, "b_reader");
  myb_writer = new Writer<int>(&painted_frame, myb_writer_to_b_readerChannel);
  SETNAME(myb_writer, "b_writer");

  // Create UNIT_1_SEQUENTIAL group
  myUNIT_1_SEQUENTIAL = new Sequential(
    (CSPConstruct *) myWeld_frame,
    (CSPConstruct *) mya_writer,
    (CSPConstruct *) myb_reader,
    (CSPConstruct *) myAdd_wheels,
    NULL
  );
  SETNAME(myUNIT_1_SEQUENTIAL, "UNIT_1_SEQUENTIAL");

  // Make UNIT_1_SEQUENTIAL recursive
  Recursion<CSPConstruct>* myUNIT_1_SEQUENTIALRecursion = new Recursion<
      CSPConstruct>(myUNIT_1_SEQUENTIAL);
  SETNAME(myUNIT_1_SEQUENTIALRecursion, "UNIT_1_SEQUENTIAL-recursion");
  myUNIT_1_SEQUENTIALRecursion->setEvaluateCondition(true);

  // Create UNIT_2_SEQUENTIAL group
  myUNIT_2_SEQUENTIAL = new Sequential(
    (CSPConstruct *) mya_reader,
    (CSPConstruct *) myPaint_frame,
    (CSPConstruct *) myb_writer,
    NULL
  );
  SETNAME(myUNIT_2_SEQUENTIAL, "UNIT_2_SEQUENTIAL");

  // Make UNIT_2_SEQUENTIAL recursive
  Recursion<CSPConstruct>* myUNIT_2_SEQUENTIALRecursion = new Recursion<
      CSPConstruct>(myUNIT_2_SEQUENTIAL);
  SETNAME(myUNIT_2_SEQUENTIALRecursion, "UNIT_2_SEQUENTIAL-recursion");
  myUNIT_2_SEQUENTIALRecursion->setEvaluateCondition(true);
```

```
    // Register model objects
    this->append_child(myUNIT_1_SEQUENTIALRecursion);
    this->append_child(myUNIT_2_SEQUENTIALRecursion);

    // protected region constructor on begin
    // protected region constructor end
  }
```

The generated code itself is documented, allowing to read it with ease. At the start, two channels are defined. Next, the set with model objects is made: the reader and writer objects receive the defined channels as parameter. These model objects are placed in the sequential and recursive structures. Finally, both sequential constructs are added (or registered) as child to the object being created. This object is of type parallel, making the children of the object to run in parallel.

With TERRA and LUNA, also more advanced CSP constructs are possible. For example: Alternatives, allowing the program to choose between two processes based on guards, timed execution using a time channel, PriParallel structures indicating one process is more important than another, interfacing to IO using IO channels, using submodels and importing 20-sim code. With these features, it is possible to define many systems to control a robotic application.

A.2 Design and implementation of the ROS-LUNA bridge

Details about the design and implementation of the ROS-LUNA bridge are given in chapters 2.2 and 2.3. The design of the bridge is split in three subsystems: the communication protocol, the implementation in ROS, and the implementation in LUNA. As further elaboration on the design, the serialization and deserialization processes in the communication protocol are explained more in depth in appendix A.2.1.

The implementation in ROS is further elaborated with more in depth explanation of the runtime binding in appendix A.2.2. A way to support the Service/Client structure with the bridge is also presented.

Since the implementation of the LUNA side in chapter 2.2.2 was rather complete, no additional information is given in this appendix regarding its design.

A.2.1 Communication protocol and management

Serialization and deserialization of variables

In serialization two aspects need to be accounted for: multiple variables can be present in one stream, and the variables can be of a different type (and data size). A way to distinguish variables in such a stream is implemented, by splitting the payload of the message into two parts: one part, named the variable description field, contains information describing the type, name length and data size of the serialized data of each variable. The second part, the actual payload, contains both the serialized name and data of the variable.

Serialization in the communication protocol is done by appending bytes to the variable description field: one byte containing a number representing the variable type, as declared in the communication protocol. A second byte contains the length of the serialized name, and a third byte contains the length of the serialized data. In the payload the name is appended, followed by a serialized version of the data. This serialization is done by copying the data from the variable to an array of characters.

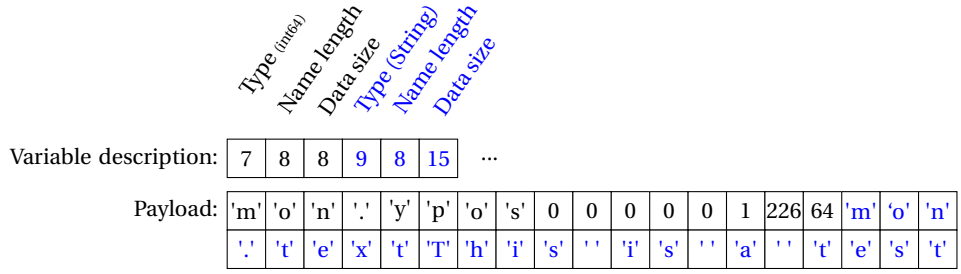

Figure A.3: Example of serialization of variables with a variable description and a payload.

In Figure A.3 a visualization of the serialization of two variables, ''mon.ypos''(*int64*) and ''mon.text''(*string*), is given. The variable values are 123456 and "This is a test": for readability, bytes based on a character are represented by their character, while other bytes are represented as their decimal value.

When the serialized data is send, a predefined header structure is prepended to the variable description field and payload. This header structure contains fields indicating the type of packet and the sizes of the field: the succeeding parts are of dynamic length, and their layout are retrievable by the receiver using these lengths.

Further deserialization of the actual variables is done by iterating over the variable description field: the bytes indicating name length and data size are used to select the name and the raw data from the payload. Once the name and data are selected, it is possible to lookup earlier

registered callbacks, and allow these callbacks to either cast the selected data using the variable type, or copy the raw data into an actual variable using its data size. When only the data size is used, it will be possible to use data formats not specified in the protocol, allowing for more flexible datatransfer at the cost of possible validation of the data type. This is not an issue when the data types are correctly setup on both sides of the system during initialization.

A.2.2 ROS side

The ROS-side of the bridge needs to be able to subscribe to and publish message types without knowing upfront (that is, during compile-time) what type it is: it is not known what type of message the LUNA application is going to use. A way to determine and load the definitions during runtime binding is needed. In chapter 2 section 3, the implementation of the runtime binding using the *TopicListener* and *RuntimeBindingPublisher* are presented. How the *TopicListener* analyses a message during runtime is further explained using an example.

TopicListener example

The message definition retrieved by the *ShapeShifter* class contains besides the definition of the message type, also the definition of all the nested message types. This section of the definition is used in a recursive function to determine the size of non-primitive types. The recursion ends when a primitive data type is found. This recursive function also needs to keep track of the position inside the raw data: there exist data types with dynamic sizes (for example: strings). In ROS, these dynamic sized data types are prepended with an *uint32_t* containing the size of the data type. When a dynamic field is encountered in the recursive function, the value of this *uint32_t* is extracted from the location in the raw data. Keeping track of the position in the raw data is done, by summation of the sizes of all the previously found datatypes.

As example a custom type '*NestedMessage*'is decoded. The structure and data in this example is:

```
1 bool field_1 = true;
2 TwoInt32 field_2 = {int32 data = 1234567;
3                     int32 data2 = 7654321;};
4 int32 field_3 = 112233;
```

The goal is to find the value of field_3 in the raw data. The *MessageDecoder* receives this message from the *ShapeShifter* as:

Byte #	0	1	2	3	4	5	6	7	8	9	10	11	12
Byte value	[001]	[135]	[214]	[018]	[000]	[177]	[203]	[116]	[000]	[105]	[182]	[001]	[000]

Table A.1: Raw data bytes from the NestedMessage.

The message also contains the format of the message (for clarity, comment fields are left out):

```
1 bool field_1
2 TwoInt32 field_2
3 int32 field_3
4 ================...================
5 MSG: luna_bridge/TwoInt32
6 int32 data
7 int32 data2
```

The message decoder analyses the format by starting with line 1: it finds *field_1*, of type *bool*. This is a primitive type, of size 1 byte. The decoder continues, as *field_1* is not the desired field.

Next, line 2 is analysed. *Field_2* is found, with $TwoInt32$ as type. This type is not a primitive type, therefore the size is determined by calling the recursive function. This recursive function tries to find the line which separates the message definitions ("===...===="). It then determines whether the line after this line contains the desired type: otherwise, the next separation needs to be found.

In this case, line 4 contains this separation, and line 5 indicates that the description is for $TwoInt32$. Next, the contents of the definition are analysed by the function: it finds two fields of type $int32$: both are primitive types, and therefore their size is known.

When another nested type would have been found, the recursive function would have started to look for the definition, until only basic data types would have been found.

In this case, the recursive function returns that the size of $TwoInt32$ is 8 bytes (2x4). The function continues on line 3 of the messagedefinition, and finds the desired field_3 of type $int32$. Before this field, 1+8 bytes where found: it is now known that the data corresponding to field_3 starts at byte 9, and is 4 bytes long:

[105][182][001][000]

Casting the data to an $int32$ (taking read direction into account), results in the correct value of 112233.

The position and the size of the found field are stored in the *MessageDecoder*. The placed callback is called, which allows the function that registered a Subscriber to this topic and field to copy the values directly in a variable of the correct type, through casting or *memcpy()*. In the case of the ROS-LUNA bridge, the data received from ROS is added as byte sequence as described in section 2.1 from chapter 2: it is therefore not useful to copy the data into a variable, but is directly added to the outputbuffer.

RuntimeBindingPublisher

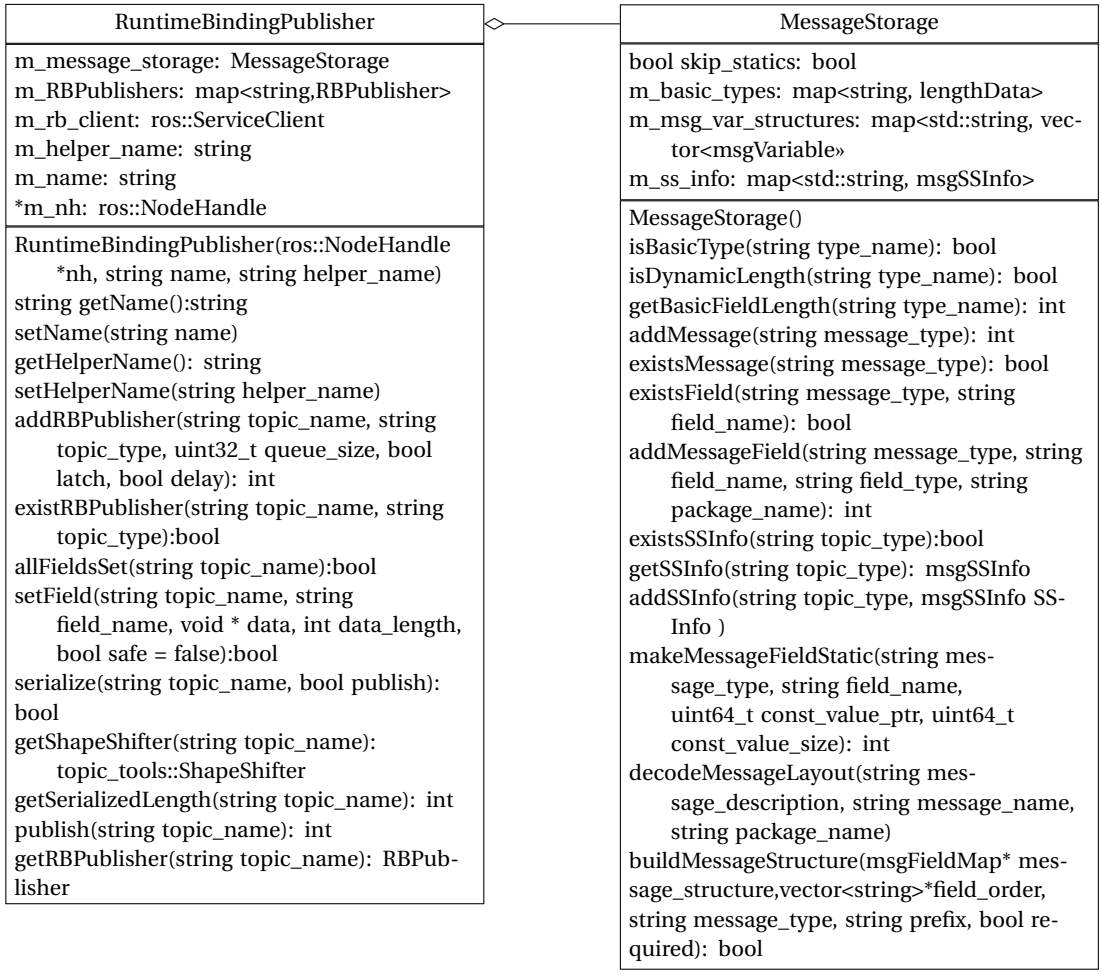

Figure A.4: Classdiagrams implementating runtime binding for the publishers.

In Figure A.4 an overview of the implementation of the *RuntimeBindingPublisher* is given. The *RuntimeBindingPublisher* provides functions to add publishers (function addRBPublisher()) using the topic name and topic type specified as string. A set function allows to set a value in the message of an added publisher. The class has a *MessageStorage* object: this structure contains functionallity to convert a received message description and convert it to a usable map.

In the class' headers some custom types are declared: this allows shorter code. The definitions and where they are used for are listed below.

```
/**
  *@brief: type definition to store store the type of message with a ShapeShifter
  */
  typedef struct {std::string type_description;std::string md5;} msgSSInfo;
/**
  *@brief: type definition to store information about low level variables
  */
  typedef struct{int64_t length;bool dynamic;} lengthData;
/**
  *@brief: type definition to store information of a field inside a message
  */
  typedef struct{std::string field_name;std::string field_type;bool required;std::
      string package_prefix; bool is_static; std::string const_data;} msgVariable;
```

```
/**
  *@brief: type definition to store all field of a message
  */
  typedef struct {int raw_data_length; std::string type_name; bool is_set; bool
      required; std::string raw_data;} msgField;
/**
  *@brief: type definition to map the name of a message to it msgField object
  */
  typedef std::map<std::string, msgField> msgFieldMap;

/**
  *@brief: structure to store data regarding a publisher added during runtime
  */
  typedef struct{ std::string topic_name; std::string topic_type; ros::Publisher
      publisher; topic_tools::ShapeShifter shape_shifter; MessageStorage::
      msgFieldMap message_fields;std::vector<std::string> fieldOrder;} RBPublisher
      ;
```

ROS services

In ROS, it is possible to use so called Services[3] instead of the publish/subscribe structure. With services, it is possible to call a function made available in an other node as a client. The client which performs this call, will send a service message to the server: this service message is a combination of a request and response message. The called function receives this message, handles the data, and sends the service message with its response set back to the client. Meanwhile, the client will halt until either a time-out occurs or the response is received. Support for the services could be implemented in two ways: either fully support both connecting to services as client and serving clients inside the LUNA bridge and LUNA application, or by adding a compatibility or conversion node only at the ROS side, converting implemented Publisher/-Subscriber structures to Service/client structures.

The first option results in the fastest execution and most robust and reusable setup, but takes longer to implement:

- Serialization/Deserialization is harder in services: normal messagetypes could use the *ShapeShifter* class. For Service message there does not yet exist such a class.
- Calling a service as client, results in the clients' process being locked, until the time-out occurs or a response is received. The lock means, that further communication to other topics and network communication is also halted. A way to circumvent this, is by completely rewriting the structure of the LUNA bridge node, allowing it to create separate threads.
- Letting a ROS node call a service present in a LUNA application, needs to be handled somehow in the CSP by a combined action of Reading and Writing data. This means adding another new component in the LUNA library.

Furthermore, services are generically used to change or request parameters: actions carried out incidentally. This reduces the need for fast execution: it will therefore suffice to use a conversion node.

In figures A.5 and A.6 conversion from and to a service call is depicted. Using these conversions, it is possible to mimic a ROS service in LUNA or to mimic a ROS client in LUNA.

Conversion from a service call to LUNA, is done by registering the service inside the conversion node. When a ROS node performs a call on this service, the callback is called in the conversion

[3] http://wiki.ros.org/Services

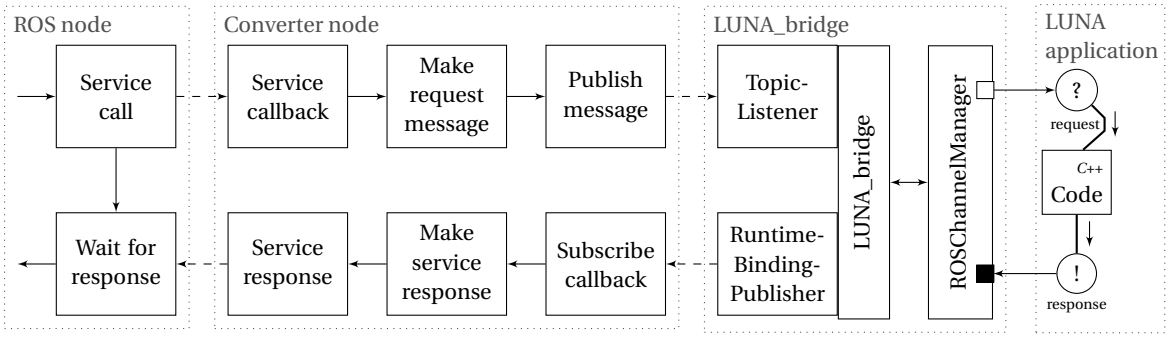

Figure A.5: Mimicking a service in LUNA using a conversion node in ROS.

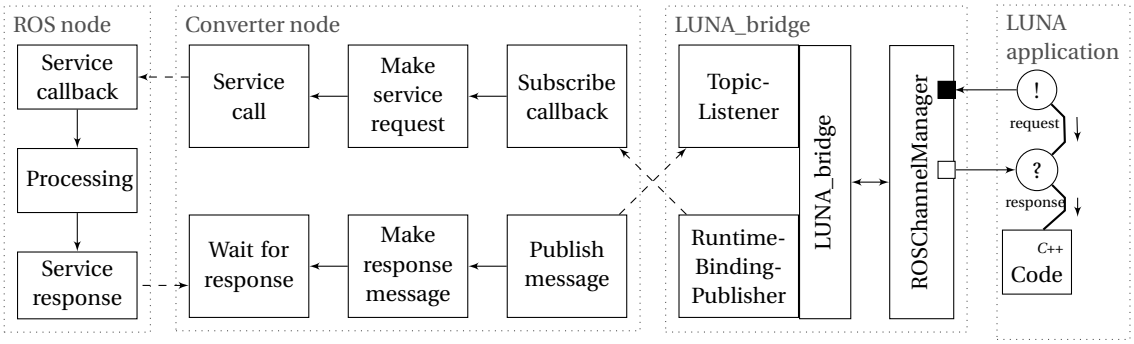

Figure A.6: Mimicking a client in LUNA using a conversion node in ROS.

node. This callback activates a conversion, copying the data from the service request to a message with the same format. This message is published on a topic, which the LUNA bridge is listening to through a *TopicListener*. Through the LUNA bridge and *ROSChannelManager*, the data of the request will be forwarded to de CSP structure in the LUNA application: in the figure, this is represented by one reader: to read more fields, more parallel readers could be used. When the data is read, a model or C++ code executing the service functionallity is activated. When this block is done, the results are written back to ROS, using CSP writers, connected to the *ROSChannelManager*. Using the RuntimeBindingPublisher, the result is published. Inside the converter node, this message is received using a subscriber and a callback. This callback copies the data in the response message to the response field inside the service. Finally, the service response is send back, allowing the client to receive the outcome of the service and continue.

Conversion to a service call in ROS, is done in similar manner: only in this case, the LUNA application issues a request using the *RuntimeBindingPublisher*, and waits for a response using the *TopicListener*. The conversion node provides these topics, and convert data to and from a service call.

A.3 Tests

Tests and results characterizing and verifying the bridge are presented in chapter 2 section 3. Additional tests and figures regarding the delay, jitter and network are added in this appendix.

A.3.1 Delay and jitter measurements

The results of jitter and delay tests of a setup using the ROS-LUNA bridge are presented in Table 1 and 2 in chapter 2. For visualisation purposes, part of these measurements are depicted in Figure A.7 and A.8.

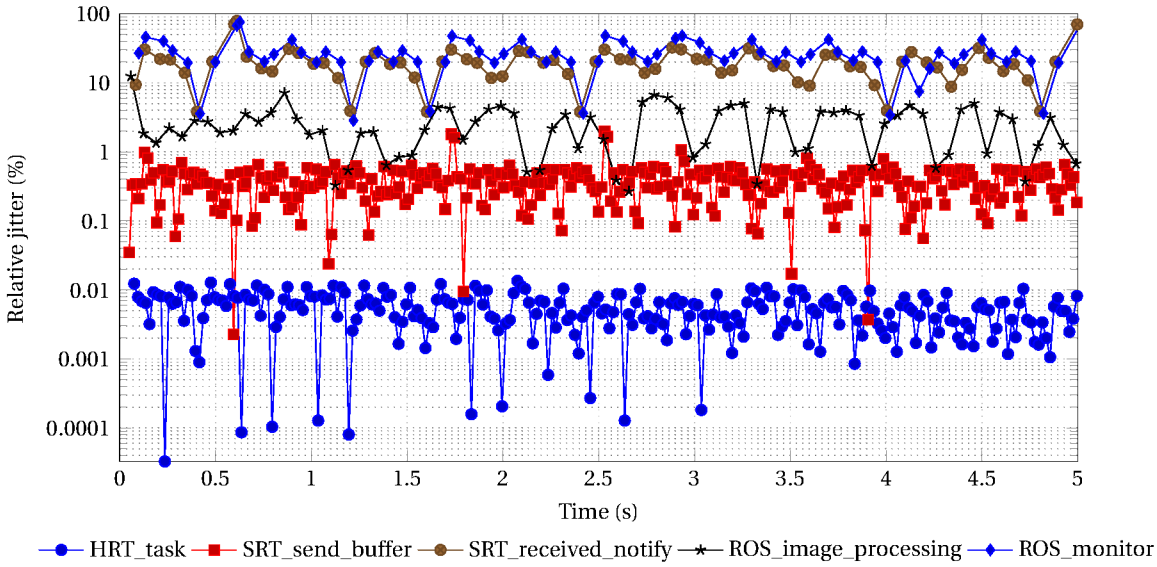

Figure A.7: Plot of jitter measurements.

Table A.2: Delays introduced by writing to the network.

	LUNA bridge (ROS)	LUNA application
Average (ms)	0.610	0.267
Stdev (ms)	0.480	0.0991
Max (ms)	2.74	0.917
Min (ms)	0.180	0.0518

In Table A.2 the delays introduced by writing to the network are presented. The column "LUNA bridge (ROS)" is determined by measuring the difference between the callback of the *TopicListener* and the timestamp when the data was converted and the buffer was written to network. This delay is part of the delay "ROS_imageprocessing → SRT_receive_notify" delay present in Table 2 of chapter 2.

The delay needed to find the field using the *TopicListener*, convert it and write it to the network is low compared to the total delay: 0.610 ms out of 15.5 ms is needed. This shows, that the delay in sending data from a ROS to LUNA is mainly determined by the ROS network and the normal network.

The column "LUNA application" in Table A.2 contains the delays introduced by sending the buffer to the network inside the LUNA application. The delay is part of the delay "SRT_send_buffer → ROS_monitor" from Table 2 in chapter 2.

The delay introduced by converting the buffer to a TCP packet and writing it to the network are low compared to the total delay: 0.267 ms out of 2.6 ms is needed.

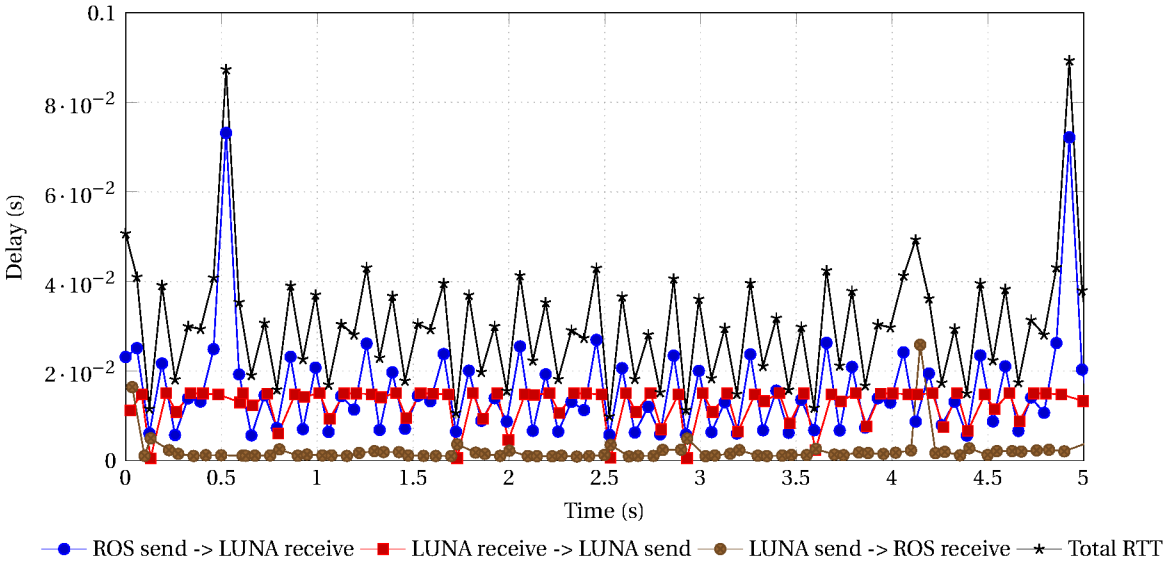

Figure A.8: Delay measurements.

Note: in Figure A.7 the y-axis is plotted on a logarithmic scale, since the range of the relative jitter over the multiple parts of the system is high.

Further interpretation and explanation of these measurements are present in chapter 2 section 3.2.

A.3.2 Controlling a robotic setup

In chapter 2 section 3.2.3, a test setup is described using vision in the loop to demonstrate the ROS-LUNA bridge. The a blockdiagram of the layout and a picture of the test setup are depicted in figure A.9 and A.10.

Figure A.9: Blockdiagram of the topology of the test setup using vision in the loop.

On the PC, image processing is performed on a videostream send from the RaMstix. This imageprocessing sends locations of green objects over the network, used as setpoint on the embedded system for its controllers. These controllers make the pan/tilt camera in the JIWY rotate, allowing it to track the green object.

Further test results and description of this test setup are found in appendix A section 3.2.3.

A.3.3 Networking tests

Network connections, especially wireless connections, cause delays when used in a system. Since the medium of the connection could also be in use by other systems, these delays are non-deterministic. When more systems try to use the same data bandwidth on the network, delays might increase due to queuing. Also, some network connections could suffer from packetloss: especially in wireless connections like WiFi, packetloss will be present when a sent packet

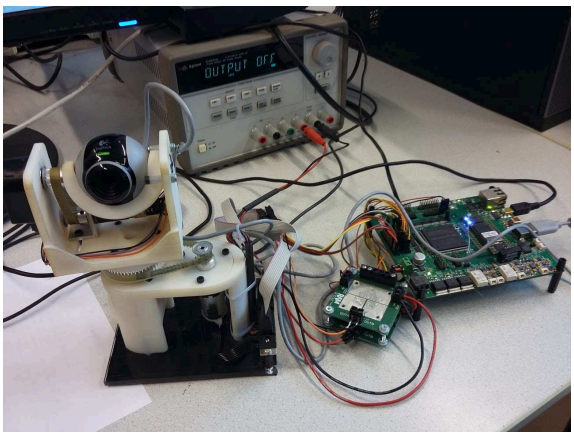

Figure A.10: Image of the JIWY setup (left) connected to a RaMstix board(right).

collides with the packet of another sender. To simulate the behaviour of (wireless) networks, and analyse the effects on the response, two tests where done.

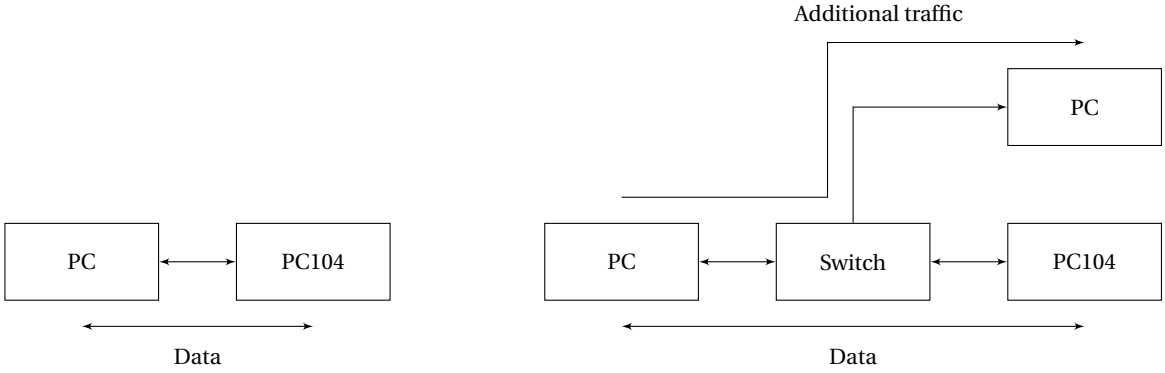

Figure A.11: Simulating the influence of packetloss (left side) and additional traffic (right side) on the response time of the initial version of the ROS-LUNA bridge.

Simulating packetloss

The first test is used to analyse the influence of packetloss, emulating a wireless network. The test setup is depicted in the left side of Figure A.11: a connection is made between a PC (running ROS and a luna_bridge [4] node) and a PC104 (running a LUNA application) through a 100Mbit/s ethernet network. Data is send from the PC to the PC104. Packetloss is simulated by issuing a command on the PC, which ill-configures the networking card, instructing it to drop packets with a certain change. This change is configured in a range from 0% to 50 %. Command used on the PC to configure the change of packetloss:

```
$: sudo tc qdisc change dev eth0 root netem loss 1%
```

The PC104 runs a LUNA application, which connects to the PC using the ROS-LUNA bridge. It sends a value to ROS, and waits for a reply. The time interval between sending and receiving is measured: this gives the response time of the complete system. One part of this response time is the delay introduced by the network, another part of the measured delay is the response time of the ROS-LUNA bridge's software. Keeping the software the same throughout the test, and only sweeping the change of packetloss, allows to see the influence of just the packetloss

[4] The version of Bezemer and Broenink (2015) with slight modification was used

in the network on the total response-time. Refer to the left side of Figure A.11. Sending data is repeated 200 times for each configuration of packetloss. The first response time is discarded: in this case, the ROS-LUNA configuration also needs to initialize objects and the connections, increasing the response time. The average of the response time is printed by the PC104: normally during development, the SSH connection provided by the QNX momentics toolsuite is used for visualization of the terminal of the PC104. Since this SSH connection also connects through the ethernet, it is also subjected to the ill configuration of the network card. Therefore, a separate monitor connected to the PC104 is used for visualization.

Simulating additional traffic in network

The second test is used to analyse the influence of additional traffic over the network: the test setup is depicted in Figure A.11. The setup simulates the network being shared with another datastream, for example streaming video. The same basic setup is used as in the first test: the PC is still connected to a PC104. On the PC a program generating (invalid) TCP packets (called `mausezahn` [5]) at a configurable rate is running in parallel with the LUNA-bridge. Handling incorrect datapackets occupies hardware resources: therefore, the additional traffic generated is sent to a second PC. To let the first PC connect to both the PC104 and second PC, a switch is used. Mausezahn is capable to send packets at a certain interval: to measure the actual used bandwidth on the network, `bmon` is used after the `Mausezahn` application is started with the desired configuration. Next, the same application as in the first test is used to measure the response time.

The command used to start `mausezahn`

```
$: sudo mz eth0 -B 192.168.1.30 -c 0 -t tcp "dp=1-1023,␣flags=syn" -P "PAYLOAD" -d
    INTERVAL
```

Where PAYLOAD contains a sequence of characters and INTERVAL the value at which rate the packets should be send. This will result, together with some overhead in `Mausezahn`, in the measured additional traffic.

Test results

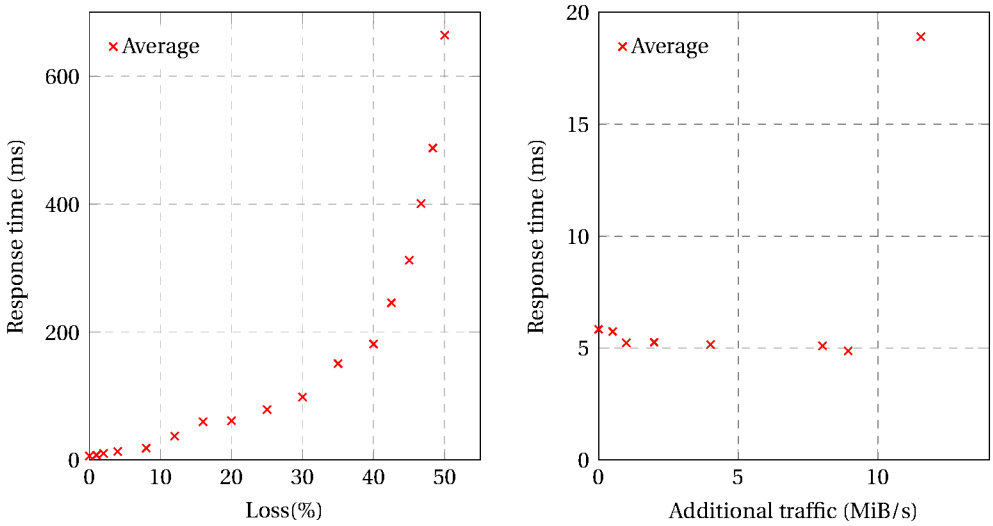

Figure A.12: Response time of the ROS-LUNA bridge versus configured change of packet loss (left graph) and additional traffic (right graph).

[5] http://man7.org/linux/man-pages/man8/mausezahn.8.html

The results of both tests are depicted in Figure A.12. The graph on the left shows the response time versus the configured change of packet loss. The loss of packets clearly influences the measured response time. The graph on the right shows the response time versus the configured additional traffic. The additional traffic causes no significant increase in response time from 0 to 8MiB: some small variations are present, but are not significant. However, when the additional traffic neared the maximum capacity of the network (100 Mbit/s, equal to 11.92 MiB/s [6]). This shows that as long as the network is not near complete congestion, no increase in response time is expected.

[6] MiB=MebiByte. MiB is 2^{20} = 1048576 bytes, compared to 1000000 bytes in a normal MB.

B Additional appendices

In this chapter, additional appendices are added. These appendices focus on the usage of the different part of the designed ROS-LUNA bridge. Also an additional overview is giving, linking the MoSCoW list from the project proposal to the actual implementation.

In appendix B.1, an annotated MoSCoW list is added. This list allows a quick overview on how the set requirements are met, and were in the document this is described.

In appendix B.2 an example is given on how to use the *RuntimeBindingPublisher* class in ROS.

In appendix B.3 an example is given on how to use the *TopicListener* class in ROS.

In appendix B.4, documentation is added on how to compile LUNA for the RaMstix and how to use it.

In appendix B.5, documentation is added on how to use ROSchannels in the current version of TERRA.

In appendix B.6, documentation is added on how Gstreamer was used in this assignment.

B.1 Requirements

In the project proposal, a MoSCoW list was made to indicate the requirements on the then to-be-designed bridge. Since the description of the final design is spread out over the paper and the appendices, an overview is made below. This overview depicts an annotated MoSCoW list: the requirements are combined with both a description in *italics*, and information on how the bridge handles this requirement, also referring to sections in the paper and other appendices where the implementation and testing is further described.

B.1.1 Annotated MoSCoW requirements

A **Must have**

I Versatile communication library

The bridge must have a library which can encode and decode a wide scale of variables and messages.
Using a protocol defining the variables and fields inside messages by their name and data size, allows support of versatile datatypes and messages. Refer to chapter 2 section 2.1.

II Extensible communication library

The library must have programming that allows easy extension and adaptation.
Providing the option to either specify the variables' type by using a definition in the library or by its size, allows to extend the library with datatypes not implemented in the communication protocol. Refer to chapter 2 section 2.1.

III LUNA side communication based on CSP structures

LUNA side of the bridge must be implemented in CSP structures, to allow verification using formal tooling like FDR3.
The implementation using a channelmanager to convert incoming data to channels allows CSP readers and writers to be used to communicate with ROS. Refer to chapter 2 section 2.2.

IV Compatibility with Publish/Subscribe structure

The bridge must connect the variables in LUNA with messages in ROS, and be able to publish these messages on topics and subscribe to topics.
The implementation of runtime binding in subscribers and publishers, using the *TopicListener* and *RuntimeBindingPublisher* classes, allows support of Publish/Subscribe structures in ROS based on variables of LUNA. Refer to chapter 2 section 2.3.

V Automatic configured publish/subscribe structure ROS side

The ROS side should initialize the needed Publishers and Subscribers, to reduce complexity at LUNA side.
The LUNA_bridge node receives initialization commands from LUNA, and makes Publishers and Subscribers based on these commands. This allows the LUNA application to configure the ROS side of the setup with a simple set of commands. Refer to chapter 2 sections 2.2, 2.3 and 3.1.

VI Independence between ROS and LUNA

Since the LUNA library must be able to work without recompiling, the changeable dependencies of ROS must be omitted.
The implementation in ROS is made to perform runtime binding, based on the name of a topic and the name of its messagetype. This circumvents the need to include libraries from ROS in LUNA.

VII Working test setup (using JIWY)

To demonstrate correctness and how the bridge works, a demo setup must be made, based on JIWY.
An application distributed over ROS and LUNA was written to control a JIWY setup. Using an setpoint generating algorithm in ROS, the controller setpoints of the controllers present in LUNA are able to make the JIWY setup rotate its axes. Refer to chapter 2 section 3.2.3.

VIII Network usage analyzed

To show network statistics like delay, bandwidth utilization, network usage should be tracked and analysed.
No direct network statistics was added: the measurements from appendix A.3.3 indicate that bandwidth and network utilization is hard to link to the performance of the link. Measuring the delay could indicate the quality of the link. Using the ROS and LUNA environment allows to add measurements easily by the user.
Using a writer sequentially followed by two readers in LUNA, allows the user to write a time stamp to ROS. This time stamp is subtracted in a ROS node with the current time, and published with the current time stamp through the bridge to the two readers. The LUNA application should then perform an action based on the measured delays, for example activate a failsafe option in case the link becomes weak.

B **Should have**

I Compatibility with Server/Client structures

The Server/Client structure allows a reply on a topic. This allows for inquiring specific parameters from LUNA, or using ROS as computational device.
Using the conversion techniques as described in appendix A.2.2, it is possible to convert client/server structures in ROS into the supported Publish/Subscribe structures.

II ROS control through bridge

It would be useful to configure ROS automatically through a LUNA app. Things like security of the ROS system should be kept in mind.
The communication protocol supports sending commands to ROS: handling them in ROS is not yet implemented. It would also be possible to run a separate configuration node, which receives commands from LUNA to start new nodes or issue reconfiguration commands through a terminal commands.

III Advanced network configurations

It would be useful for the user to get more advanced control over the network interface: dynamically set send frequency of packets, port configuration, host configuration.
A submodel in the LUNA application is used to configure at which intervals data is send from LUNA to ROS: this interval is specified by the user. In the source code, the port and IP-address are also configured. In future integration into TERRA, these field could be configured from the graphical user interface of TERRA.

IV Validation of timing and correctness network/ packages

TCP should guarantee correct and in order arrival op packages. Link loss and too large delays should be checked. Should be combined with warnings to the user when problem with link is detected.

V Demo with shared network connection with high load

Instead of using the network just for the data send over the bridge, a test should be performed where the network is also used to send large amounts of data, for example a video feed.
The test setup containing the JIWY pan/tilt camera is used, combined with a gstreamer pipeline sending data from the embedded system over the network to ROS. This stream adds an additional load on the network, and allows the setup to use vision in the loop. Refer to chapter 2 section 3.2.3

VI Tests / examples using other setups

To persuade new users to use LUNA combined with ROS, converting multiple example setups should be useful.
Only the JIWY setup is converted to use vision in the loop in a combined system with LUNA and ROS. A manual is written on how to combine a LUNA application with ROS through the ROSChannels. Refer to appendix B.5.

C **Could have**

I TERRA integration

Integration to TERRA blocks would allow easier integration into LUNA projects of an user. This would however take a lot of effort and time, and might be implemented in a future project.
No direct integration into TERRA was yet feasible due to time related matters: developing for TERRA requires time to get acquainted with the development tools. Using the manuals presented in chapters B.4 and B.5 will allow someone already familiar with TERRA development to integrate the implementation in LUNA into the code generation and interface of TERRA.

II Graphical bridge configuration interface for user

Although the bridge can be configured using code, for setting up experiments it might be useful to have an GUI, where most important settings could be changed to the bridge.
Future extensions of TERRA would allow the user to design the structure of both the ROS algorithms and its connections to the LUNA application. This reduced the need for an additional graphical interface.

III Graphical message decoding interface for user

When changes are made to either ROS or to LUNA, it might cause names to change. A GUI which allows to see unconnected nodes of the same type, and configure a layer in between these nodes is useful.
In ROS, there are already commandline options present to connect and reroute topics. Combined with graphical tools like `rqt`, allows the user to graphically see the structure in ROS, reducing the need for an additional interface.

IV Graphical interface for network analysis

For verification and adaptations in the settings of the network connection, it would be useful for the user to have the state of the network accessible in an interface.
Since it was chosen to let the user implement network analysis himself by letting him measure delays using CSP structures and a ROS node, no interface was made to show the network state.

V Compatibility with actionlib structure(Service/Client with program state feedback)

The Server/Client structure is extended in the actionlib library, which allows for preemption, and state messages. Extending the ROS-LUNA bridge to also support actionlib, would make the usage of a ROS system for its resources (just like accessing a su-

per computer to perform computational expensive tasks) better. Using the same sort of conversion techniques presented in appendix A.2.2 to convert Service structures could be used to convert actionlib structures.

VI Fully compilable code

To make the ROS side more efficient, it would be useful to derive a full implementation in C++. This will take time, since versatile communication is easily implemented in interpreted languages, but are harder C++.
Using the implementation of runtime binding in the *TopicListener* and *RuntimeBindingPublisher* allows almost a full implementation in C++. Only during initialization of the Runtime Binding Publisher, a connection is made to a Python node. Once a message type received from this node is analysed, its cached version is used, making the implementation faster. Refer to chapter 2 sections 2.3 and 3.1.

VII Demo with ported ROS code to LUNA

Some effort is put into making a real-time library for ROS by others. Some of the applications that use this library, could be ported to use LUNA instead, and connect to ROS through the bridge.
In the assignment, all effort was put into the bridge itself. No time was put into converting code of others.

D **Won't have**

I Verification of ROS message generation at LUNA side

A check during LUNA application compile time (or in case of using TERRA: code generation time), to check whether messages are complete, is useful. Sophisticated tricks would be needed to perform these verifications, without making the LUNA library depend on ROS. No effort was put into implementation: although it would be possible to extend TERRA to import message definition, and let it verify the correct configuration of the initialization commands.

II Tests connecting other platforms to LUNA

Some other environments have a bridge to ROS: for example, Matlab has an interface. Also other platforms could directly be accessed through a bridge based on the ROS-LUNA bridge. This is out of the scope of this assignment.
No other platforms were combined with the LUNA and ROS bridge.

B.2 Using the runtime binding publishers

The *RuntimeBindingPublisher* class allows to add publishers during runtime, by specifying the name of the publisher and the topic type as string. The generated publisher will bind to this topic during runtime.

Setting fields in the message of these publishers is also done by using just the name of the field.

B.2.1 Example

For clarification, a codelisting is presented below, demonstrating code to generate a publisher during runtime. A topic *jiwy_setpoint* is created, with topic type *luna_bridge/jiwy_position*. This message consists of a standard *Header* structure (containing a sequence number and a timestamp), alongside two doubles representing the x and y rotation of the JIWY setup.

```
//Include the runtime binding
#include "luna_bridge/RuntimeBindingPublisher.h"

using namespace runtime_binding;
//Small program to show runtime binding
int main(int argc, char **argv)
{
    //Init ROS
    ros::init(argc, argv, "rlb_helper_tester");
    // Make a Node Handle, needed for ROS
    ros::NodeHandle nh;
    // A rate of 100Hz should be enough
    ros::Rate loop_rate(100);
    ROS_INFO("Starting␣RuntimeBinding␣Publisher␣test.␣Make␣sure␣a␣'helper'␣node␣is␣
        running␣as␣node␣'rlb_helper'!");

    //Make the actual object that can perform runtime binding!
    //The RBP is going to connect to a python node called 'rlb_helper'
    RuntimeBindingPublisher RBP(&nh,"rlb_RBP","rlb_helper");

    // Example: we want to have a publisher with topic 'jiwy_setpoint' and type '
        luna_bridge/jiwy_position'
    std::string desired_topic_name="jiwy_setpoint";
    std::string desired_topic_type="luna_bridge/jiwy_position";

    // Create the publisher with buffersize 1 and latching enabled
    RBP.addRBPublisher(desired_topic_name,desired_topic_type,1,true,false);
    //Define the data to be send
    double xpos=0.12345;
    double ypos=0.2468;
    uint32_t sequence=1;
    //get the timestamp
    ros::Time timestamp = ros::Time::now();
    //Other fields will automatically be set to 0's

    //Set the data inside the RBP, using a ptr to each variable and their sizes
    RBP.setField(desired_topic_name,"xpos",(void *)&xpos,sizeof(xpos),true);
    RBP.setField(desired_topic_name,"ypos",(void *)&ypos,sizeof(ypos),true);
    RBP.setField(desired_topic_name,"header.seq",(void*)&sequence,sizeof(sequence),
        true);
    RBP.setField(desired_topic_name,"header.stamp",(void*)&timestamp,8,true);
    //Publish the message!
    RBP.publish(desired_topic_name);
    //Publishing the next message is as simple as repeating all the set commands
        with new data
}
```

When this example is run, data will be published on the *jiwy_setpoint* topic. Visualization of the data is possible, by adding a subscriber to this topic. This is simply done by using the build-in `rostopic` echo command:

```
>$: rostopic echo jiwy_setpoint
        header:
          seq: 1
          stamp:
            secs: 1467233508
            nsecs: 426700937
          frame_id: ''
        xpos: 0.12345
        ypos: 0.2468
```

The output of the `rostopic` echo shows the received message, as set in the example code.

B.3 Using TopicListener

The *TopicListener* class allows to listen to fields inside a topic, without knowing their type up-front. A code listing on how a callback can be placed on such a field is listed below.

```
#include "luna_bridge/TopicListener.h"
#include "ros/ros.h"
using namespace luna_bridge;
void callback(TopicListener* listener)
      {
        //The referenced listener contains the ptr to the desired data, copy it.
        double xpos=0;
        int desired_data_size=8;
        if(listener->getDataSize()==desired_data_size)
        {
          //Found and desired sizes coincide!
          memcpy(&xpos,(void*)listener->getDataPtr(),8); // A cast could also be
              used
          ROS_INFO("Received xpos [%0.9f]",xpos);
          //Normally, the callback contains more advanced options, e.g. allowing to
              copy data to the network buffer
        }else{
          ROS_ERROR("Found a field with wrong size! desired [%d] vs received [%d]",
              desired_data_size,listener->getDataSize());
        }
      }

int main(int argc, char **argv)
{

   //Init ROS node
   ros::init(argc, argv, "jiwy_setpoint_listener");
   // Make a Node Handler
   ros::NodeHandle nh;
   //Make the topiclistener, let it listen to the xpos value
   TopicListener* listener = new TopicListener(nh, "jiwy_setpoint", "xpos",&
       callback);//std::bind(&TopicListenerExample::callback, this, std::
       placeholders::_1));
   //Spin ros, as the callback takes care of the rest
   ros::spin();
   return 0;
}
```

The registered callback is called, as soon as the registered field *xpos* is found. The *TopicListener* contains a pointer to the found position in the raw data. and the size of the data. This pointer and the found size are used to copy the data to a variable.

With the example code of appendix B.2 still running, executing this code snippet will result in:

```
>$: rosrun luna_bridge exampleopicListener
      [ INFO] [1467238709.451963128]: Received xpos [0.123450000]
```

This shows that the correct value is decoded..

B.4 Compiling LUNA and LUNA applications for RaMstix

When LUNA needs to run on a device with Linux as operating system, a patch should be applied to the kernel to make it real-time. This patch is called Xenomai[1] . The current image present on the RaMstix is patched with Xenomai version 2.6.3. The kernel version is a bit old however: this causes problems when normal crosscompiling is done. Also, the code generation of TERRA is currently mainly used to generate code for QNX. Although a large part of the functionality is present in the code generation for Xenomai, some fixes need to be applied to the generated code to make it work. This appendix explains the steps taken to make LUNA applications run on the RaMstix, running Linux kernel 3.2.21 with Xenomai 2.6.3.

B.4.1 Prerequisites

- It is expected that the user already has TERRA setup correctly and has the LUNA source (located at `~/LUNA/LUNA`).
- It is expected the install occurs on a Ubuntu system (or inside a Ubuntu VM). Other operating systems might work as well, but are not tested.
- It is also expected that cross compile tools are setup correctly. When this is not the case, some of the compile commands will fail, and indicate which part of the cross compiler is not installed. (Needed are: binutils-arm-linux-gnueabi gcc-arm-linux-gnueabi g++-arm-linux-gnueabi).

B.4.2 Downloads

- Download Xenomai version 2.6.3 from http://download.gna.org/xenomai/stable/.
- Download Xenomai version 2.6.2.1 from http://download.gna.org/xenomai/stable/.
- Download Linux kernel 3.2.21 from https://www.kernel.org/pub/linux/kernel/v3.x/.

The second download is needed, since Xenomai version 2.6.3 does not have a patch for kernel 3.2.21, while the RaMstix still runs Xenomai 2.6.3 and kernel 3.2.21. This is fixed by taking the interrupt pipeline patch (ipipe) from Xenomai 2.6.2.1 and add it to Xenomai 2.6.3. This is not ideal, but no other fix is present at this time.

B.4.3 Directory structure setup

- A couple of directories need to be created, execute in a terminal:

```
$: sudo mkdir /opt/xenomai-2.6.3
$: sudo chown YOUR_USERNAME /opt/xenomai-2.6.3
$: mkdir /opt/xenomai-2.6.3/build
$: mkdir /opt/xenomai-2.6.3/staging
$: mkdir /opt/xenomai-2.6.3/kernel
$: mkdir /opt/xenomai-2.6.3/src
```

- For quick reference to these directories, execute:

```
$: export build_root=/opt/xenomai-2.6.3/build
$: export staging_dir=/opt/xenomai-2.6.3/staging
$: export linux_tree=/opt/xenomai-2.6.3/kernel
$: export xenomai_root=/opt/xenomai-2.6.3/src
```

[1] http://www.xenomai.org/

- Extract the contents of the downloaded xenomai-2.6.3 archive to `$xenomai_root`
  - Note: the contents of the archive should be directly in `$xenomai_root` , not inside a subfolder.
- Extract the contents of the downloaded kernel-3.2.21 archive to `$linux_tree`.
  - Note: the contents of the archive should be directly in `$linux_tree` , not inside a subfolder.
- Open the archive of the downloaded Xenomai-2.6.2.1 archive, extract file `xenomai-2.6.2.1/ksrc/arch/arm/patches/ipipe-core-3.2.21-arm-4.patch` to directory `$xenomai_root/ksrc/arch/arm/patches/`.

B.4.4 Patching the kernel

Change directory to `$xenomai_root` and run the kernel patch script:

```
$ : cd $xenomai_root
$ : scripts/prepare-kernel.sh --arch=arm --linux=$linux_tree
```

- The script should suggest the correct ipipe patch (`/opt/xenomai-2.6.3/src/ksrc/arch/arm/patches/ipipe-core-3.2.21-arm-4.patch`)

B.4.5 Preparing the kernel build

- Change directory to `$linux_tree`

```
$: cd $linux_tree
```

- Copy the correct initial configuration for the processortype:

```
$: cp arch/arm/configs/omap2plus_defconfig .config
```

- Run the configuration script:

```
$: make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- menuconfig
```

- A menuconfig tool will be opened in the terminal. This allows to easily set some options:
  - Goto System Type âĂŤ> TI OMAP2/3/4 Specific Features.
  - Select Typical OMAP configuration, TI OMAP3 and Gumstix Overo Board.
  - Deselect all other configurations.
- Exit the menuconfig tool, and save the configuration upon exit.

B.4.6 Building the kernel

- Make the kernel:

```
$: make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- modules
```

  - Note: the command might fail with "fatal error:`linux/compiler-gcc5.h`: No such file or directory". This is caused by the cross compiler being too new (version 5 instead of 4) for this old kernel. The proposed fix is to either revert back to an older compiler, or to apply a small fix by copying `linux/compiler-gcc4.h` into `linux/compiler-gcc5.h`:

```
$: cp include/linux/compiler-gcc4.h include/linux/compiler-gcc5.h
```

- Execute the make command again.
- Note: another error might occur, indicating "`/opt/xenomai-2.6.3/kernel/arch/arm/include/asm/ftrace.h`:51 : multiple definitions of 'return_address'"[2]. The proposed fix is:
  - Open `/opt/xenomai-2.6.3/kernel/arch/arm/include/asm/ftrace.h`
  - Change the line:

    ```
    extern inline void * return_address ( unsigned int level)
    ```

    into:

    ```
    static inline void * return_address ( unsigned int level)
    ```

  - Save the file.
  - Open the file `arch/arm/kernel/return_address.c`.
  - Delete or comment out the section:

    ```
    void * return_address ( unsigned int level)
    {
      return NULL;
    }
    ```

  - Save the file.
  - Execute the make command again.

B.4.7 Building Xenomai

- Change directory to `$build_root`:

  ```
  $: cd $build_root
  ```

- Configure thebuild and start building:

  ```
  $: $xenomai_root/configure CFLAGS="-march=armv7-a" LDFLAGS="-march=armv7-a"
      --build=i86-pc-linux-gnu --host=arm-linux-gnueabi
  $: make DESTDIR=$staging_dig install
  ```

- Note: after some time, your sudo password is requested.
- After completion, this will result in a Xenomai build usable with LUNA.

B.4.8 Crosscompiling LUNA

- LUNA expects the path to the build Xenomai to be present in `$XENOMAI_DIR`:

  ```
  $: export XENOMAI_DIR=$staging_dir
  ```

- Change the directory to the LUNA source and start the menuconfig, e.g.:

  ```
  $: cd ~/LUNA/LUNA
  $: make menuconfig
  ```

[2] https://github.com/zanezam/boeffla-kernel-cm-bacon/commit/ef4fea130eeb70eff4f3a549fd3f6e9b11437550

- Due to the old kernel used on the RaMstix, cross compiled programs might run into an error, indicating CXXABI on the RaMstix is too old: the cross compiled programs used functionality not yet implemented in the CXXABI present on the RaMstix' OS. The newer CXXABI implements safety measures allowing bufferoverflows to be found more easily. This functionality needs to be disabled to make LUNA applications work on the RaMstix.
  - In the menuconfig, select Advanced configuration options > Additional Compiler Commands.
  - Set the field to:

    ```
    $: -fno-caller-saves -U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=0
    ```

  - Use exit to return to the root of the menu.
- Some other options need to be set:
  - Enter Target System.
  - Select "xenomai-arm-v7" as target system.
  - Use exit to return to the root of the. menuconfig
  - Enter Target Images.
  - Enable Install Local and set its configuration to `~/LUNA/LUNA/bin`.
  - Use exit to return to the root of the menuconfig.
  - Enter Advanced configuration options.
  - Enable Target Options and enter its submenu.
  - Select "Use software floating point by default".
  - Make sure Target Optimizations is set to:

    ```
    -Os -pipe -march=armv7-a -mcpu=cortex-a8 -ftree-vectorize -ftree-
        vectorizer-verbose=3 -fdump-tree-vect -mfpu=neon
    ```

  - Use exit to return to the root of the menuconfig.
  - Make sure the correct settings are present under Core, Debug, High-level and Networking for your project.
  - Exit menuconfig and save upon exit.
  - Build LUNA:

    ```
    $: make V=s
    ```

  - Note: an error might occur indicating "bash: arm-linux-gnueabi-gcc-ar: command not found". This is fixable by making a link and restarting make:

    ```
    $: sudo ln /usr/arm-linux-gnueabi/bin/ar /usr/local/bin/arm-linux-
        gnueabi-gcc-ar
    $: make V=s
    ```

  - This should result in a compiled LUNA library.

B.4.9 Configuring TERRA

- Start TERRA.
- Configure TERRA so it will generate code for Xenomai using the just build LUNA library:
  - Open Window -> Preferences.

- Select TERRA -> LUNA in the dialog.
- Select "Xenomai" under Target platform.
- Set "Binary location" to the directory libLUNA.a was created for Xenomai, e.g. `~/LUNA/LUNA/bin/luna-xenomai-arm-v7-Posix/lib/`.
- Set "Header files location" to the directory the header files were created for Xenomai, e.g. `~/LUNA/LUNA/bin/luna-xenomai-arm-v7-Posix/include/`.
- Save the settings by pressing Ok.
- (Re)Generate the code for your whole project: the archm file and all the submodels.

B.4.10 Adapting generated csode

- Some fixes need to be applied to the generated code, to make it work after crosscompiling:
  - The crosscompiling should include C/C++ libraries statically: this prevents problems with compatibility on the older kernel.
  - Open the Makefile of the archm model or the main cspm model. This file is located in the generated directory.
  - Add in the protected region underneath Compiler/Linker flags:

```
LDFLAGS+=-static-libstdc++ -static-libgcc
```

  - Save the file.
  - Open the MainModel.cpp file of the archm or main cspm file (located in src).
  - Add beneath the SETNAME(this,"YOUR_APPLICATIONNAME") function:

```
OSScheduler::Instance()->setSchedulerPolicy(FIFO);
```

  - This instructs Xenomai to use a real-time scheduler. Otherwise (amongst others) timer channels will fail.
  - Note: this last fix is applied outside of a protected region. When codegeneration is run again, this fix needs to be reapplied.

B.4.11 Compiling the LUNA application

- Return to the terminal again.
- Change directory to the root of your TERRA project.
- Run:

```
$: make clean
$: make
```

- Binaries and libraries will appear in `bin/` and `lib/` inside the project folder after successful compilation of the project.
- Use for example scp to copy both directories to the RaMstix (assuming an ssh server is running).

B.4.12 Additional notes on executing the LUNA application on the RaMstix

- It is useful to add an entry to `/etc/hosts` pointing "roshost" to the current IP of the system running ROS.
- Some large applications seem to have startup problems, ending in an error mentioning a "RealTime Signal 36". This is not yet debugged. However running the applications in GDB seems to work fine.
- Since the threading model is based on Posix, context switches are expensive. This causes performance degradation when many threads are used in large programs.
- Standard stack size for each pthread seems to be 8MB on many systems. Since each CSP component is implemented using one pthread, it may issue problems with the 256MB RAM present on the RaMstix. It is advised to set the thread-size to a smaller value when this happens, for example:

  ```
  $: ulimit -s 512
  ```

  Note: for normal users it is only possible to decrease the stacksize.

B.5 Using ROS with TERRA

The current version of TERRA does not have graphical object and codegeneration for the connections to ROS. While TERRA is still in development, this feature will be added. In the meantime the connection to ROS could be used by applying some tricks. These tricks are explained by using a simple example setup modeled in TERRA, and adapting its generated code so the ROSChannels of LUNA could be used. In the step by step explanation it is expected that the user has some experience with TERRA.

B.5.1 Global steps

Globally, the workflow is:

- Model the system in an archm file, using submodels both the LUNA part of the application and for the representation of the nodes located in ROS.
- Generate code.
- Remove the ROS submodels, adapt and replace the channels between ROS and LUNA submodels by ROSChannels in the generated code.

B.5.2 Making and changing an example application

The sample application will connect to a Publisher node in ROS (ROS_pub). When a value from this topic is received in LUNA, it is send back to ROS to a Subscriber (ROS_sub). Both ROS topics have std_msgs/Int32 as their messagetype. This messagetype contains one field named "data", of type int32.

Make a model for the embedded (LUNA) application

- Create if desired a new TERRA project.
- Make one csp model, containing the actual LUNA application, named "LUNA_app.cspm". In this case, it is a simple structure: a sequential Reader and Writer, receiving/writing data from ROS.
- Add an incoming and outgoing port in LUNA_app.
- Add a CSP Reader and Writer in a sequential structure (Reader -> Writer).
- Connect the ports to the Reader and Writer.
- Give the Reader and Writer a name(e.g. valueReader and valueWriter)
- Add a unit with data type integer, and add a variable "value" with this unittype.
- Set the variable in the Reader and Writer to "value".
- Group the Reader and Writer, and make the group recursive.
- Give the incoming and outgoing ports recognizable names, e.g. value_in_port and value_out_port. These portnames will appear in the generated code which needs adaption.
- Make the diagram Shareable and set the name to LUNA_app, so the model is usable with the archm. Refer to figure B.1.
- Execute the code generation.

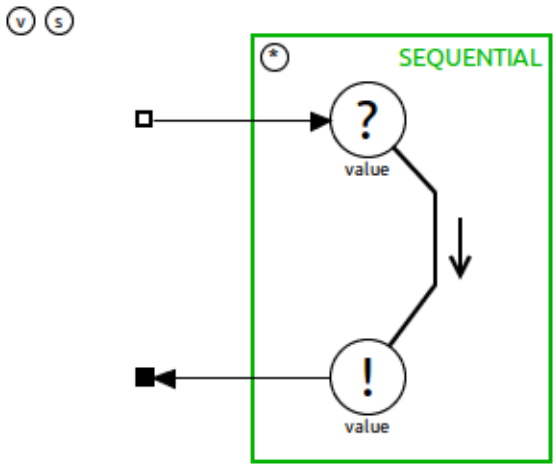

Figure B.1: Structure of the application to be run on the embedded system.

Make a model for buffer control

- Make a second csp model, named "Buffer_send.cspm". This model will periodically call the function that sends the buffered data to ROS.
- Add a unit with datatype "time period" inside the model, and add a variable "timer" with this unit.
- Add a Writer and an incomming port, and connect them with an channel.
- Name the Writer and incomming port(e.g. timeWriter and timePort), and set the variable in the writer to the timer variable.
- Place a C++ Code Block in a sequential construct after the Writer, and name the c++ block (e.g. "send").
- Group the writer and c++ code block, and make the group recursive.
- Make the diagram shareable, set the component name to "Buffer_send", and generate code.
- Open the generated cpp file of the C++ block (e.g. `Buffer_send/src/send.cpp`).
- Add in the "protected region additional headers":

```
#include "ros-channels/ROSChannelManager.h"
using namespace LUNA::ROS;
```

- Add inside the "protected region execute code":

```
ROSChannelManager::Instance()->sendPacket();
```

- Save and close the file. Refer to figure B.2.

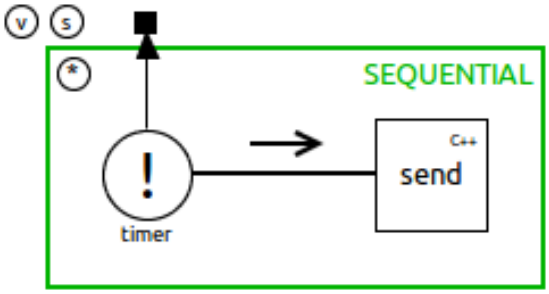

Figure B.2: Structure of the outgoing buffer controller.

Make a model representing ROS publisher node

- Make a cspm model representing the ROS publisher, named "ROS_publ.cspm". This model will only be used to allow code generation (providing the start of a channel in TERRA), and will be removed from the generated code.
- Add a unit with the same definition used in LUNA_app and add variable of this type.
- Add a Writer, set the variable and give it a name (e.g. tbrWriter).
- Add an incoming port, and connect it to the Writer with a channel. Give the port a recognizable name, e.g. data_port.
- Make the diagram shareable, set the component name to "ROS_publ", and generate code. Refer to figure B.3.

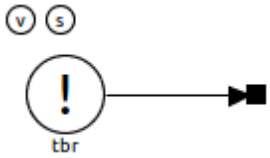

Figure B.3: Model representation of the ROS publisher node.

Make a model representing ROS subscriber node

- Make a cspm model representing the ROS subscriber, named "ROS_sub.cspm". This model will again only be used to allow code generation (providing the end of a channel in TERRA), and will be removed from the generated code.
- Add a unit with the same definition used in LUNA_app and a variable of this type.
- Add a Reader, set the variable and give it a name (e.g. tbrReader).
- Add a port, set it to outgoing, and connect it to the Reader with a channel. Give the port a recognizable name, e.g. data_port.
- Make the diagram shareable, set the component name to "ROS_sub", and generate code. Refer to figure B.4.

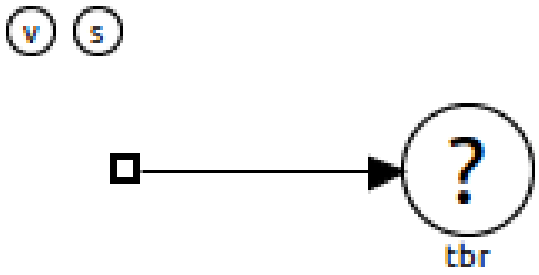

Figure B.4: Model representation of the ROS subscriber node.

Make an architecture model

- Make an architecture model, named "application.archm".
- Add the four previously made cspm files as external models, and set their names to the name of the model.
- Connect the port of ROS_publ("data_port") to the port of LUNA_app(value_in_port).
- Connect the port of ROS_sub("data_port") to the port of LUNA_app ("value_out_port").
- Add a Periodic Timer Port, and connect it to the port of Buffer_send.
- Give the Periodic Timer Port a name, and set its interval. This interval specifies the frequency at which data will be send back to ROS: inside this interval, data written to ROS will be buffered. This interval is a trade of between latency, processing time needed to start sending data, and the size of the buffer. In many cases, a frequency between 30 and 60 Hz should suffice. Refer to figure B.5.
- Generate code from the archm model.

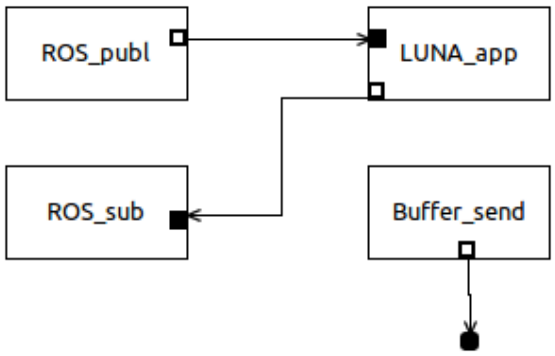

Figure B.5: Architecture model to combine the parts of the application.

Adapt generated code

After the modelling, the generated code needs to be adapted to implement the actual ROSChannels.

Mainmodel.cpp

- Open the MainModel.cpp of the code generation of the archm model (located in `application/src`).

- Delete the initialization of both ROS nodes(probably named "myROS_publ" and "myROS_sub") and their SETNAME calls, which are located somewhere beneath the line "Initialize model objects".
- Delete the same ROS nodes' objects from the parallelGroup.
- In the destructor, remove also the delete commands for these ROS nodes (located beneath "Destroy model objects").
- Find the channel connecting the port of the ROS_publ ("data_port") to the port of LUNA_application (value_in_port). This channel is probably named:

```
myROS_publdata_port_to_LUNA_appvalue_in_portChannel
```

- Notice the occurence of the port names in the channel.
- This channel should implement a subscriber in ROS: it is going to listen to the data field inside the ROS_publ topic. Replace the declaration of the new Unbuffered channel with a declaration of a new ROSChannel:

```
new ROSChannel<int>("ROS_pub", "data",1,ChannelsProtocol::NodeTypes::
    SUBSCRIBER);
```

  The template indicates the datattype. The first argument the topic this subscriber needs to connect to. The second argument represent the name of the field inside the topic. The third parameter indicates the length of the buffer: currently only 1 is tested. The last argument indicates that this channel is a subscriber in ROS. This is indicated by using the enumeration provided in the ChannelsProtocol.

- Generically, this needs to be done for each channel connecting a CSP Reader to a ROS topic
- Find the channel connecting the port of the ROS_sub to the outgoing port of LUNA_app (value_out_port). This is probably named:

```
myLUNA_appvalue_out_port_to_ROS_subdata_portChannel
```

- This channel should implement a publisher in ROS: it is going to add data in a message, and when the message is completed, it is going to be published on the specified ROS topic. This also means, that when a messagetype consisting of multiple fields a channel needs to be made for each field.
- Replace the UnbufferedChannel generation with a ROSChannel, indicating the name of the ROS publisher we are making, and the field this channel writes to:

```
new ROSChannel<int>("ROS_publ", "data",1,ChannelsProtocol::NodeTypes::
    PUBLISHER);
```

- In this example, the messagetype only consists of the data field, so only one channel needs to be converted.
- After these adoptions of the generated code, some additional code needs to be added: the code contains instructions for the bridge to be set up correctly.
- First, a connections should be made. Place the following code below `SETNAME(this,"MainModel");:`

```
int status = ROSChannelManager::Instance()->connectToROS("roshost",
    LUNA_BRIDGE_DEFAULT_PORT);

if(status != SUCCESS)
 {
   switch(status)
   {
     case LUNA_ERROR_1:
       LOG(LUNA::LOG_ERROR, "Invalid socket\n");
       break;
     case LUNA_ERROR_2:
       LOG(LUNA::LOG_ERROR, "Could not set hostname\n");
       break;
     case LUNA_ERROR_3:
     case LUNA_ERROR_4:
       // Error already printed by ROSChannelManager
       break;
     default:
       LOG(LUNA::LOG_ERROR, "luna_bridge - System error: %s (%d)\n",
           strerror(status), status);
       break;
   }
 }
```

- Note: in connectToROS, "roshost" is either the IP address of the ROS host, or the name specified in `/etc/hosts`. LUNA_BRIDGE_DEFAULT_PORT is the default port(12345) the LUNA bridge is listening to.

- Next, code instructing the ChannelManager to setup callbacks and send instructions to ROS for configuration, needs to be added: For sending data to ROS, per topic:

```
ROSChannelManager::Instance()->connectToTopic("ROS_sub","std_msgs/Int32"
    ,ChannelsProtocol::NodeTypes::PUBLISHER,"");
```

  For each channel receiving data from ROS:

```
ROSChannelManager::Instance()->connectToTopic("ROS_publ","data",
    ChannelsProtocol::NodeTypes::SUBSCRIBER,"");
```

  In both cases should the first argument coincide with the arguments provided by their declared ROSChannels. The second argument of the topics data is written to define the messagetype (prepended with the package the message is in) in ROS. The second argument for the channels receiving data from ROS specify the field inside the topic which should be listened to.

- In the header's protected region, add the following includes and namespace usages:

```
#include "ros-channels/ROSChannel.h"
#include <unistd.h>
#include "csp/CSP.h"
#include "debug/Debug.h"
#include "interfaces/LunaError.h"
#include "ros-channels/ChannelsProtocol.h"
#include "ros-channels/ROSChannelManager.h"
#include "scheduler/OSScheduler.h"
#include "threading/OSThread.h"
#include "threading/Runnable.h"
#include "threadblocker/OSThreadBlocker.h"
using namespace LUNA;
using namespace LUNA::CSP;
```

```
using namespace LUNA::CSP::ROS;
using namespace LUNA::ROS;
using namespace LUNA::Threading;
```

MainModel.h

- Next, open the MainModel.h from the application.archm (`application/include/MainModel.h`).
- Find the definitions of the ROS models (below "Model objects"), and delete them.
- Find the definitions of the channels(below "Channel definitions") connecting LUNA to ROS, and change their datatype from UnbufferedChannel to the ROSChannel. Also change the template instruction to correspond with the declarations made inside the MainModel.cpp. Change:

```
UnbufferedChannel<int, One2In, Out2One> *
    myLUNA_appvalue_out_port_to_ROS_subdata_portChannel;
UnbufferedChannel<int, One2In, Out2One> *
    myROS_publdata_port_to_LUNA_appvalue_in_portChannel;
```

  to:

```
ROSChannel<int> *myLUNA_appvalue_out_port_to_ROS_subdata_portChannel;
ROSChannel<int> *myROS_publdata_port_to_LUNA_appvalue_in_portChannel;
```

- Below the #includes, add:

```
#include "ros-channels/ROSChannel.h"
using namespace LUNA;
using namespace LUNA::CSP;
using namespace LUNA::CSP::ROS;
using namespace LUNA::ROS;
using namespace LUNA::Threading;
```

Compile

When done correctly, it will now be possible to completely make the application by running make inside the folder of the project:

```
$: make clean
$: make
```

Some of the applied changes needed to be made outside of protected regions. This means, that when the structure of the archm changes and/or code generation is executed again, these changes are lost, and need to be made again.

Additional notes

- The bridge is tested using RaMstix, with LUNA compiled for Xenomai 2.6.3
- QNX support is not fully tested: Initial implementations were tested using QNX6.5 on a PC104, using a QNX6.6 compiler. The LUNA bridge contains code which is not supported in the compiler used in QNX6.5. Although forward compatibility should be achievable with some tricks between QNX6.5 and 6.6, problems occured when IO ports needed to be used: an error occured, indicating a thread malfunctioning due to too large thread-name. The threadname seemed to be set to random values, indicating the possibility of a bufferflow somewhere, overwriting part of the threadname. It was therefore decided to move on to the RaMstix platform.

B.6 Using gstreamer with the ROS-LUNA bridge

The demo setup for the ROS-LUNA bridge uses vision in the loop. The camera, present in the JIWY setup, is connected to the RaMstix, while the image processing runs in ROS on the resource-rich platform. The camera data needs to be streamed between the two systems. Gstreamer is an application most suitable for this purpose. Gstreamer has two main versions: the older 0.10 and the newer 1.0. Since the Linux image on the RaMstix does not support version 1.0, version 0.10 is used.

B.6.1 RaMstix side

To stream the camera data from the RaMstix to the PC:

```
$: sudo gst-launch-0.10 -v -e v4l2src device=/dev/video0 ! 'video/x-raw-yuv,width
    =160,height=120' ! ffmpegcolorspace ! jpegenc ! image/jpeg,width=160,height
    =120,framerate=30/1 ! rtpjpegpay ! udpsink host=roshost port=5000
```

Some notes:

- A modified version of UDP (named Real-time Transport Protocol, or RTP[3]) is used to send the data: this protocol is designed to deliver video or audio data over IP networks.
- The gstreamer pipeline is processed by the processor: the size of the images sent influences the processor load. It was determined that a image size of 160x120 (or qqvga) has a typical load of 15-20% on the processor, leaving enough processing power for other processes.
- Higher resolutions might be achievable when processing the pipeline is partly implemented in the hardware, but no effort was put into realization since qqvga resolutions was good enough for the demosetup.
- In `/etc/hosts` an entry was made for roshost, linking it to the IP of the system running ROS.
- It might be that some of the bins (parts of the pipeline) are not available on the RaMstix: these need to be installed, for example using apt-get. Make sure these plugins are installed:
  - gstreamer0.10-ffmpeg
  - gstreamer0.10-plugins-bad
  - gstreamer0.10-plugins-base
  - gstreamer0.10-plugins-good
  - gstreamer0.10-plugins-ugly
  - gstreamer0.10-tools
  - gstreamer0.10-x
  - libgstreamer-plugins-base0.10-0
  - libgstreamer0.10-0

Additionally, some tests used a video file (`/home/ram/R_L-B-movie3.m4v`, encoded as mpeg2) as source. The command to stream a video file is:

```
$: gst-launch-0.10 -v -e filesrc location=/home/ram/R_L-B-movie3.m4v ! mpeg2dec !
    ffmpegcolorspace ! jpegenc ! image/jpeg,width=160,height=120,framerate=30/1 !
    rtpjpegpay ! udpsink host=roshost port=5000
```

[3] http://www.networksorcery.com/enp/protocol/rtp.htm

The video is only streamed once: to achieve a repeated stream, a bash script could be created containing the gstreamer pipe in a loop:

```
while :
do
 gst-launch-0.10 -v -e filesrc location=/home/ram/R_L_B_movie3.m4v ! mpeg2dec !
      ffmpegcolorspace ! jpegenc ! image/jpeg,width=160,height=120,framerate=30/1 !
       rtpjpegpay ! udpsink host=roshost port=5000
done
```

B.6.2 PC side

On the PC, the stream needs to be received and converted back to actual images. This could be done by using the gstreamer plugins inside the image processing. Another, easier way, is to use a video4linux2 (v4l2) sink as destination in a receiving gstreamer pipeline. This creates a virtual camera device on the PC, allowing other programs to interface more easily to the camera data. Before the v4l2-sink is usable, it needs to be initialized (also make sure v4l2loopback-utils is installed).

```
$: sudo modprobe -r v4l2loopback
$: ls /dev/video*
$: sudo modprobe v4l2loopback
$: ls /dev/video*
```

Using this sequence of commands, results in the kernel loading the v4l2loopback device. It also prints two lines containing available camera devices: one before the module was loaded, the other one afterwards. This is the device that is usable in for example OpenCV or in a webcam application. The camera device needs to be connected to the stream. The gstreamer pipeline is:

```
$: gst-launch-0.10 -v -e udpsrc port=5000 ! application/x-rtp, encoding-name=JPEG,
     width=160, height=120, payload=26 ! rtpjpegdepay ! jpegdec ! queue !
     ffmpegcolorspace !  v4l2sink device=/dev/video1
```

This connects the stream to `dev/video1`

Bibliography

ten Berge, M. H., B. Orlic and J. F. Broenink (2006), Co-Simulation of Networked Embedded Control Systems, a CSP-like process-oriented approach, in *Proceedings of the IEEE Int'l Symposium on Computer Aided Control Systems Conference, CACSD 2006*, IEEE Control Systems Society, pp. 434 – 439, ISBN 0-7803-9797-5, doi:10.1109/CACSD-CCA-ISIC.2006.4776685.
http://doc.utwente.nl/57680/

Bezemer, M. M. and J. F. Broenink (2015), Connecting ROS to a real-time control framework for embedded computing, in *2015 IEEE 20th Conference on Emerging Technologies and Facotry Automation*, IEEE, pp. 1–6, ISBN 978-1-4673-7928-1.

Bezemer, M. M., R. J. W. Wilterdink and J. F. Broenink (2011), LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework, in *Communicating Process Architectures 2011, Limmerick*, volume 68 of *Concurrent System Engineering Series*, Eds. P. Welch, A. T. Sampson, J. B. Pedersen, J. M. Kerridge, J. F. Broenink and F. R. M. Barnes, IOS Press BV, Amsterdam, pp. 157–175, ISBN 978-1-60750-773-4, ISSN 1383-7575, doi:10.3233/978-1-60750-774-1-157.
http://wotug.org/papers/CPA-2011/Bezemer11/Bezemer11.pdf

Brodskiy, Y. (2014), *Robust autonomy for interactive robots*, Ph.D. thesis, University of Twente, Enschede.

Buttazzo, G. (2011), *Hard real-time computing systems*, Springer, chapter 1, pp. 1–13, 3th edition, ISBN 978-1-4614-0675-4.

Franken, M. C. J., S. Stramigioli, R. Reilink, C. Secchi and A. Macchelli (2009), Bridging the gap between passivity and transparency, Robotics: Science and Systems V, Seattle, USA, p. 36.