Communicating Process Architectures 2016
K. Chalmers, J.B. Pedersen et al. (Eds.)
Open Channel Publishing Ltd., 2016

Connecting Two Robot-Software Communicating Architectures: ROS and LUNA

W. Mathijs van der Werff [1] 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.

[1]Corresponding Author: *W.M. van der Werff , Robotics and Mechatronics, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands*; E-mail: `w.m.vanderwerff@student.utwente.nl`

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.

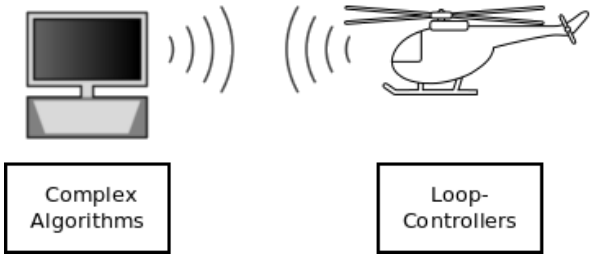

Figure 1. System overview showing separation of tasks over two systems.

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 suitable to implement the hard real-time tasks.

The main issue in combining 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. This is especially needed when changes occur to messagedefinitions: the reconfiguration and recompilation allows the nodes to communicate using the changed messagedefinition. With LUNA, the user builds an application which uses functionality from the pre-built LUNA library: it is therefore not possible to include definitions generated by ROS, as the definitions should then also be included in the LUNA library. Since these differ from system to system, and even over time on the same system, it would mean a recompilation of the whole LUNA library every time the configuration in ROS changes. This makes the reusability and development of LUNA applications harder. A solution 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, ROS, and the current version of the ROS-LUNA bridge. Also, some related work and alternative environments are presented.

1.1. LUNA

LUNA (LUNA Universal Networking Architecture) [3] is a hard real-time framework, providing support for quite some 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 between 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.

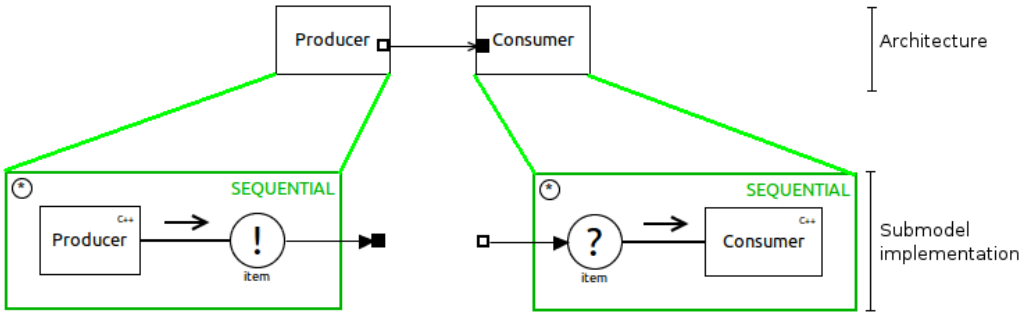

Figure 2. Producer/Consumer architecture and submodels, drawn in TERRA.

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. Adding new software by the user is also simple, 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 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 text file, describing the layout of the data.

When ROS is compiled, these messages are transformed into header files, where the user-defined fields are placed in *structs*. For serialization and deserialization, functions are

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

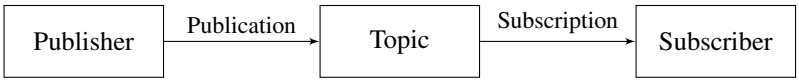

Figure 3. Publish/Subscribe structure in ROS.

added, for sending over the ROS-network as arrays of bytes. For the user it is also easy to add new parts, since Python and C++ (amongst others) are supported as programming languages.

Using these automatically generated functions makes the communication robust. To verify whether the same messagedefinition is used by both sides of the communication, MD5 checksums 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. 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 publishing 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

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

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 text stream containing the type and name of each field in the message) is sent. 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 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 hard coded to use the desired data type of the variable.

The simple tests described indicate correct functioning of the bridge: variables are sent 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 favour 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 *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 new version of 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.

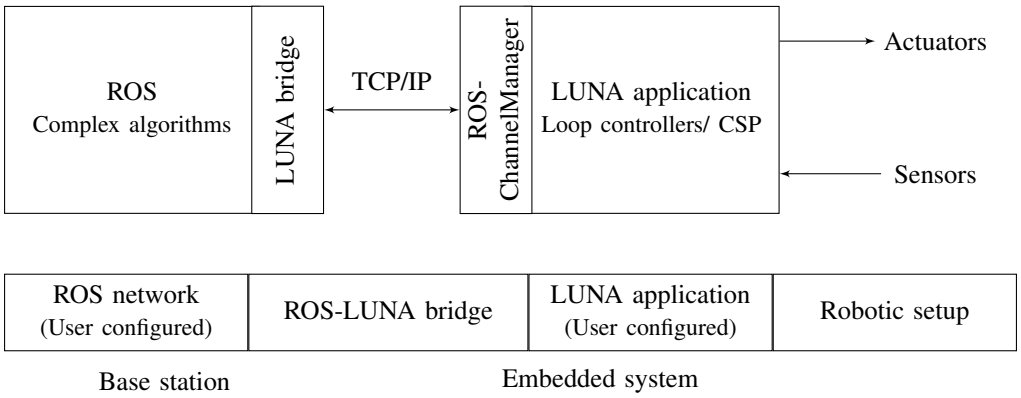

Figure 4. Global overview of the ROS-LUNA bridge.

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 sent 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 Fast Ethernet is 1500 bytes, and upto 9000 bytes

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

in Gigabit Ethernet) 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 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 data length 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 sent, both buffers are copied into the payload of an actual TCP packet. The payload is preceded with an additional header 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 sent 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 block 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 a templated class, making it possible to define the variable type of the channel based on its connected reader or writer. Writing to the network is an unpredictable task, since the hardware may not be available as it is a shared resource. To make write operations non blocking, two buffers are added. One buffer is marked to be accessible for write operations. After an user specified period, the filled buffer will be made available to a soft real time process responsible for actually sending the data over the network. In the mean time, the second buffer is marked to be accessible for the write 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 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.

[4] `http://www.json.org/`

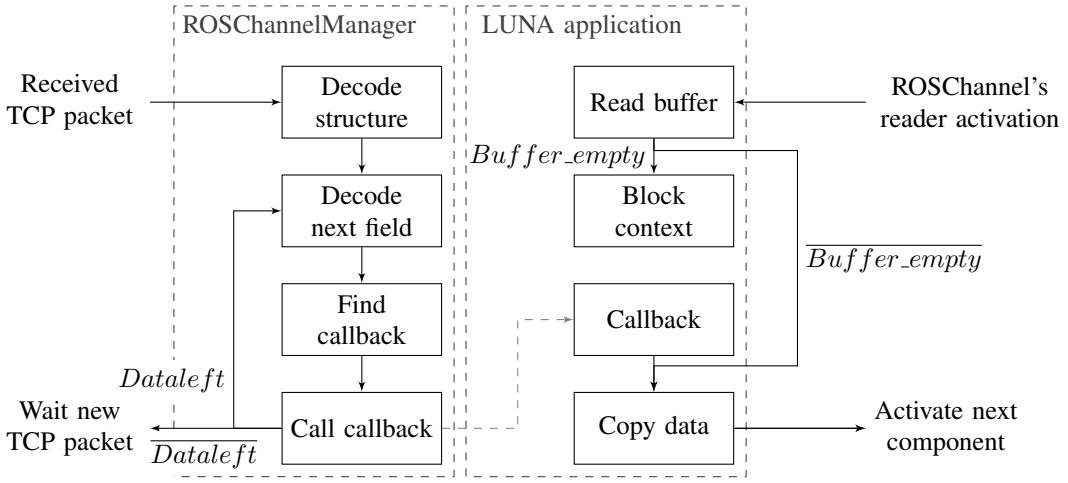

Figure 5. Schematic representation of receiving network data combined with CSP read operations.

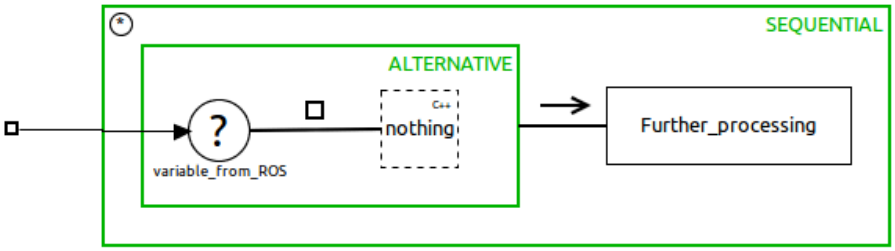

Figure 6. Example of unguarded alternative structure used to perform a non-blocking read on the ROSChannel.

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 performance 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 consist 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 sent to the LUNA application.

2.4. Overview of the ROS-LUNA bridge

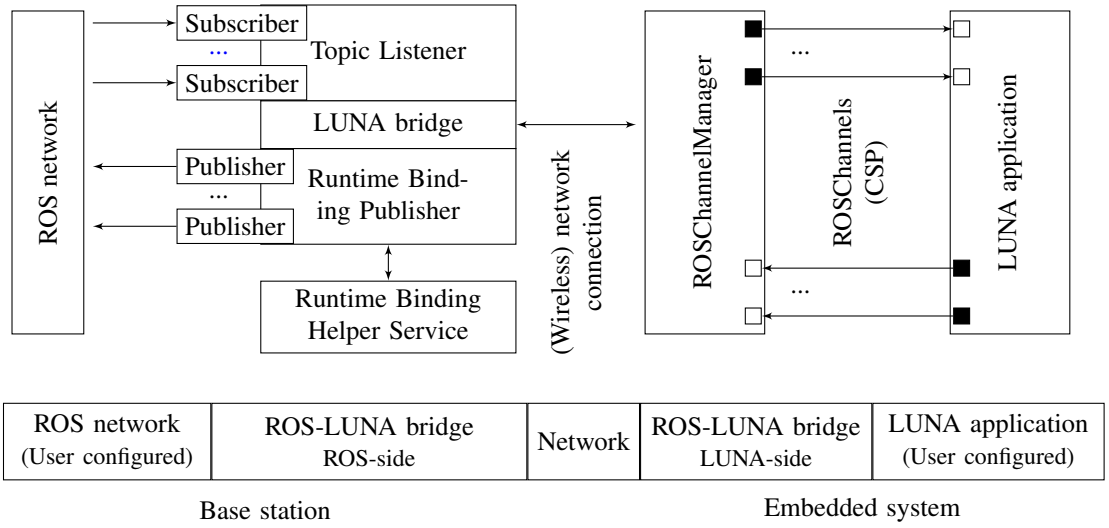

Figure 7. Block diagram of ROS-LUNA bridge.

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 incomming and outgoing ports, represented by white and black squares in the ROSChannelManager. These ports connect to the LUNA application through ROSChannels.

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 implementation 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 verify 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 demonstration 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 they perform well. The embedded side consist of hard real-time loop controllers implemented using CSP structures and the LUNA library. ROS is used to perform a complex task, represented by an image processing algorithm.

3.1. Test 1: Checking runtime binding

To check the implementation and performance 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. It is stored whether the message type was correctly serialized, 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 50,000 publications of each publisher implementation.

The test is carried out on generic notebook (Intel i5@2.53 GHz, 4 GB 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.

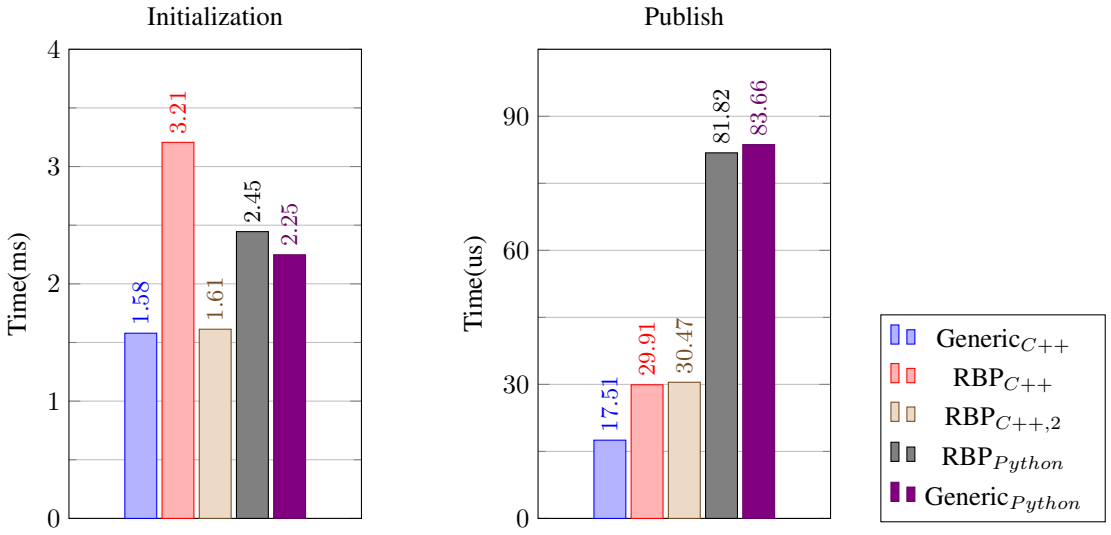

Figure 8. Performance comparison between different Publisher types.

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 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 6,000 messages at a rate of 200 Hz, containing the current time stamp in one of the *float64* fields (refer to Figure 9). Publishing is done distributed over 4 topics (/sub_test_1 to /sub_test_4), these topics are connected to two nodes, implemented in either C++ (/sub_test_cpp) or Python (/sub_test_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 (/res_cpp_N, /res_cpp_RB, /res_pyt_N, /res_pyt_RB). The messages on these topics are received by an analysis node (/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 (/timestamp_generation), the delay in the network, and the delay the subscriber types has. The measured delay consist, besides the delay introduced by the type of subscriber, also of delays imposed by the network and the publishers. Since the network and publisher will have the same delay on average over all the tests, the difference in measured delays could be used to compare the performance of the subscribers. 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. It is expected that RB$_{C++}$ and both implementations in Python perform some of the tasks performed during initialization of the normal C++ subscriber during runtime: for example, registering the callback of the subscriber is based on a template in the normal C++ implementation, while

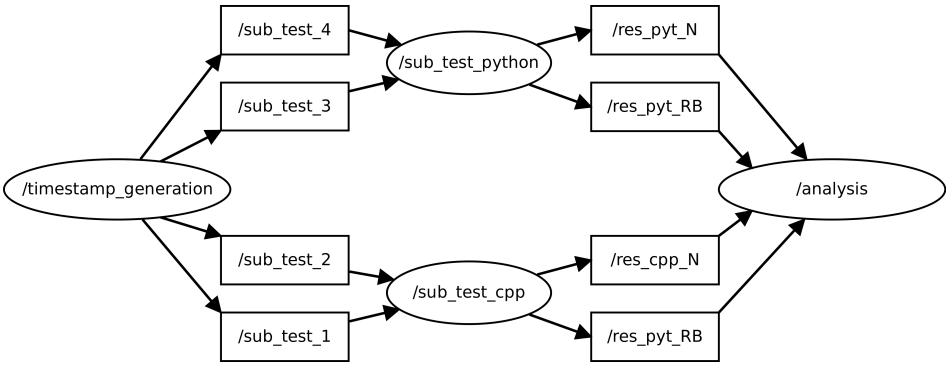

Figure 9. ROS graph of test setup measuring the delays the different types of subscribers impose.

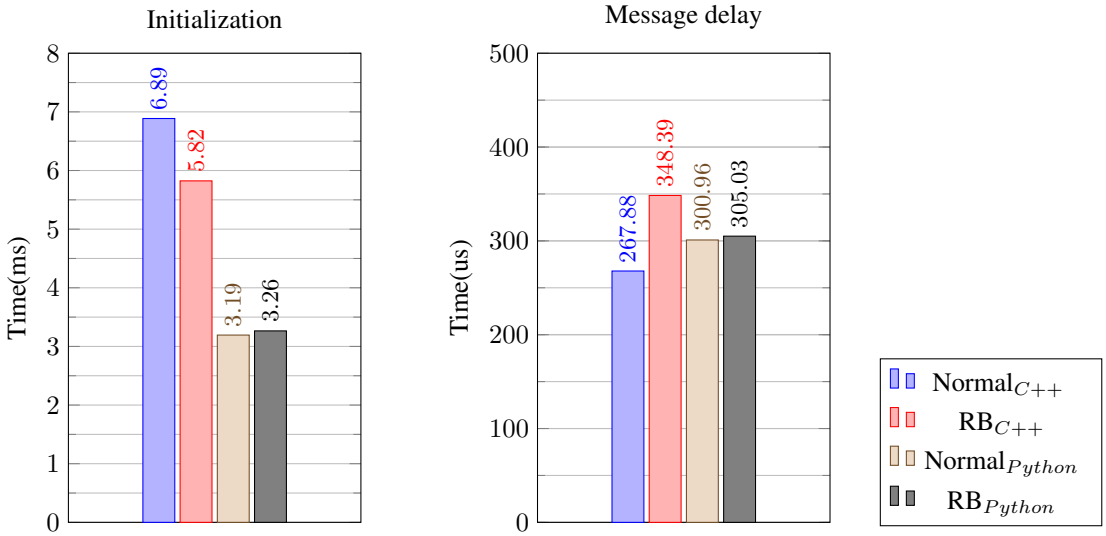

Figure 10. Performance comparison between different Subscriber types.

the other implementations have a generic callback, and have to perform an additional check whether the messagetype is correct.

This results in faster initialization, but reduced performance in during runtime. Furthermore, Python seems to be able to use optimizations, since the test is repeated multiple times, definitions are already loaded in the interpreter.

The runtime binding C++ implementation is slowest during runtime: 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 is present between both implementations in Python: the runtime binding is rather basic, adding almost no additional delays in the interpreter. 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 sent 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.

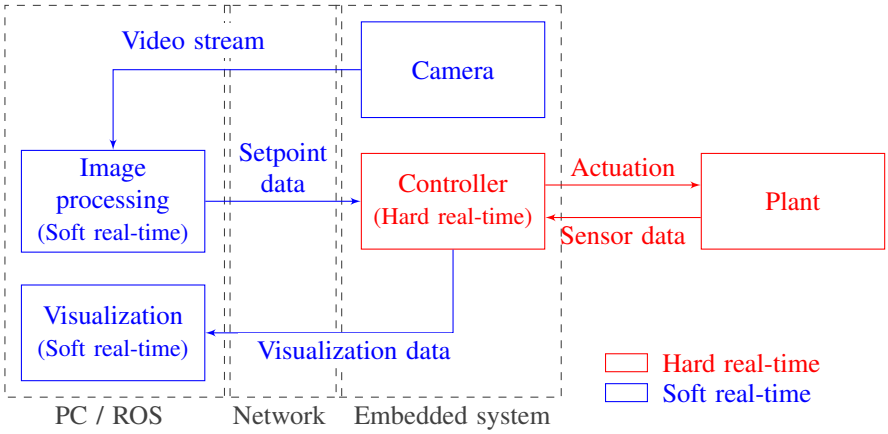

Figure 11. Block diagram of a vision-in-the-loop system distributed over two systems.

The same notebook mentioned 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 100 MBit/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 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.

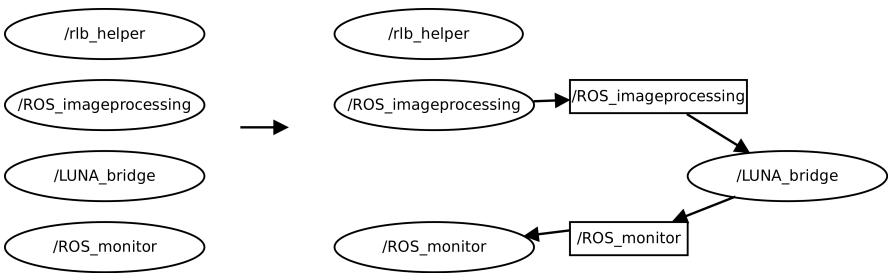

Figure 12. ROS graphs showing node overview before (left) and after (right) the LUNA application connects.

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

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 time stamps. For better repeatable test, the camera on the embedded system is replaced with a video file, which is streamed from the embedded system using gstreamer (an open source multimedia framework)[7]. 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.

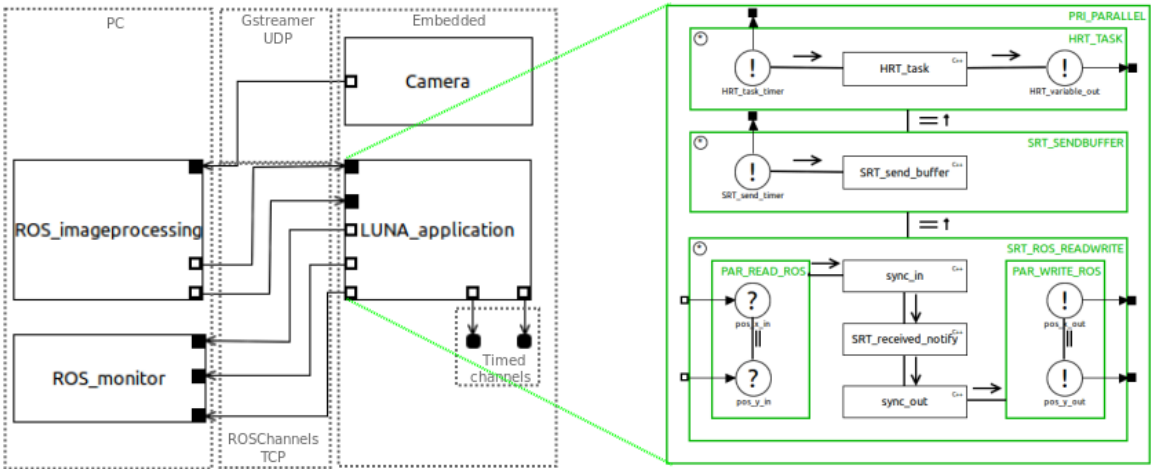

Figure 13. Overview of the total system, drawn in TERRA. Implementation of the CSP-based application and distribution over systems are added for clarity.

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 500 Hz, 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. Inside this process, a time stamp 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. It receives the asynchronous data from the network and uses readers to convert it to synchronous variables. These readers are connected to ROS using the ROSChannels, and receive the X and Y position from the image processing node. The readers are

[7] https://gstreamer.freedesktop.org/

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%

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 time stamps 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.

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 sent 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 sent 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 $\rightarrow$ SRT_receive_notify), the delay between receiving the values and sending values back (SRT_receive_notify $\rightarrow$ SRT_send_buffer), and the delay between starting transmission from LUNA and receiving them in the ROS monitor (SRT_send_buffer $\rightarrow$ 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 s. When data arrives at the start of this period, it has to wait for the whole period before it is sent back. The maximum delay in this test is 15.2 ms,

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

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 make 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 100 Hz, for which the control loops 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 Figure 14. A block is added to send data to ROS after a specified time, and a block to generate setpoints

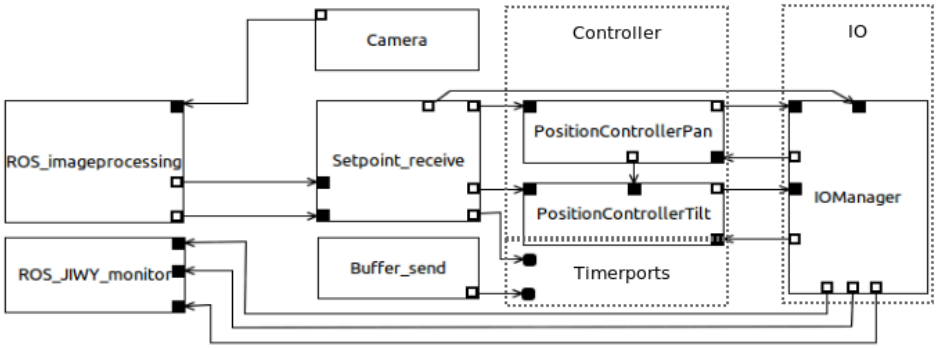

Figure 14. Architecture of setup to control a JIWY setup, drawn in TERRA

is added. Generating these setpoints is done at 100 Hz, 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 100 Hz.

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.

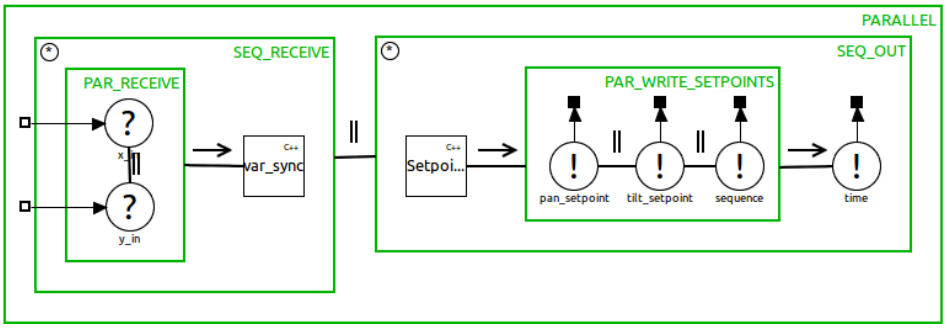

Figure 15. CSP diagram for generating setpoints and receiving new setpoints from ROS.

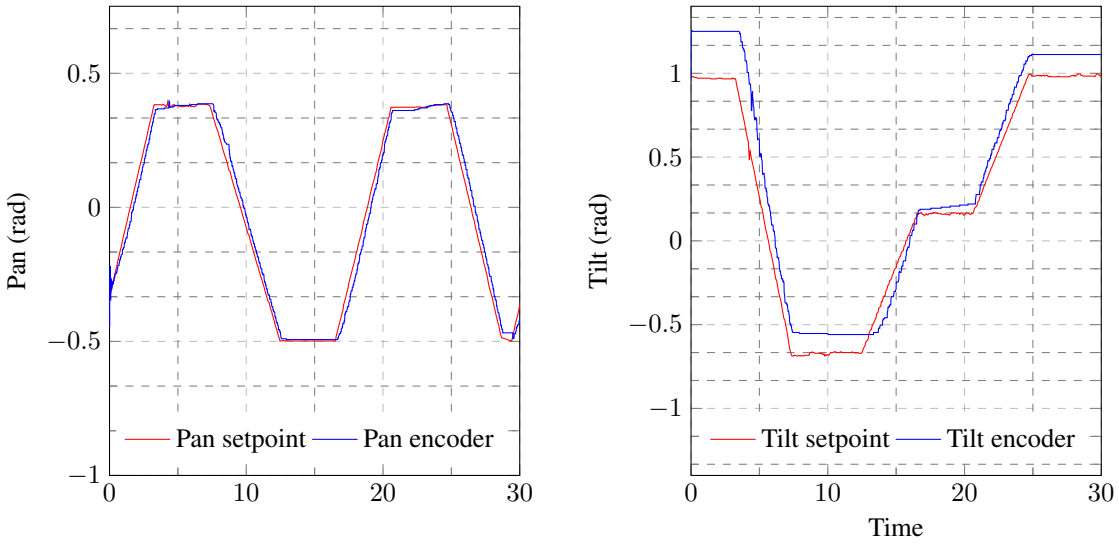

Figure 16. Pan and tilt setpoint from image processing versus the encoder values.

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 difference 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,

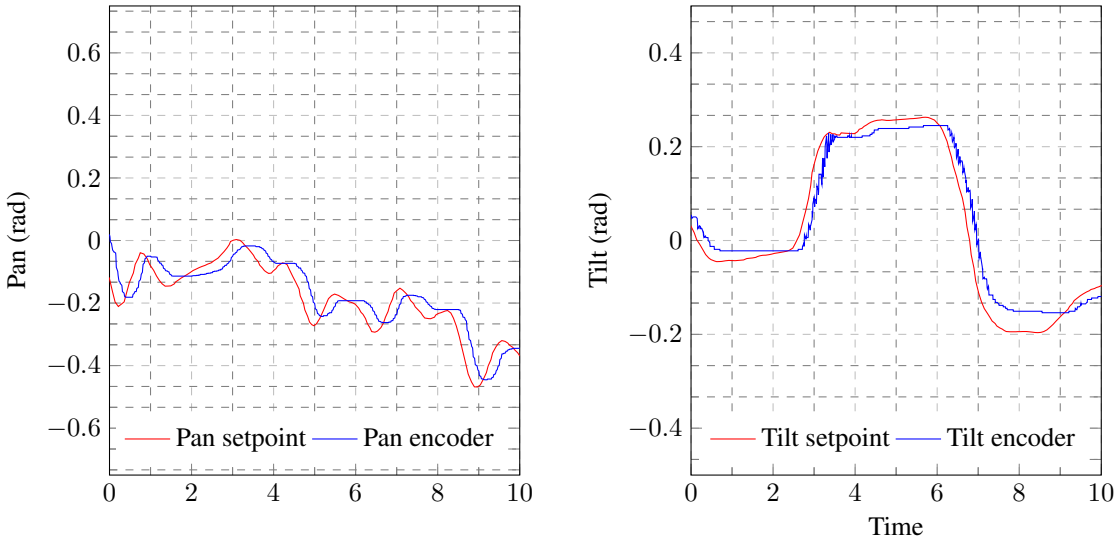

Figure 17. Pan and tilt setpoint from image processing versus the encoder values.

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.

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 sends 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 3.5, pages 268–271. Pearson, 5 edition, 2010.

[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.