Connecting ROS to a real-time control framework for embedded computing

M.M. Bezemer and J.F. Broenink
Robotics and Mechatronics Group,
CTIT Institute, University of Twente,
P.O. Box 217, 7500 AE Enschede, The Netherlands
Email: m.m.bezemer@utwente.nl, j.f.broenink@utwente.nl

***Abstract*—Modern robotic systems tend to get more complex sensors at their disposal, resulting in complex algorithms to process their data. For example, camera images are being used map their environment and plan their route. On the other hand, the robotic systems are becoming mobile more often and need to be as energy-efficient as possible; quadcopters are an example of this. These two trends interfere with each other: Data-intensive, complex algorithms require a lot of processing power, which is in general not energy-friendly nor mobile-friendly.**

In this paper, we describe how to move the complex algorithms to a computing platform that is not part of the mobile part of the setup, i.e. to offload the processing part to a base station. We use the ROS framework for this, as ROS provides a lot of existing computation solutions. On the mobile part of the system, our hard real-time execution framework, called LUNA, is used, to make it possible to run the loop controllers on it.

The design of a 'bridge node' is explained, which is used to connect the LUNA framework to ROS. The main issue to tackle is to subscribe to an arbitrary ROS topic at run-time, instead of defining the ROS topics at compile-time. Furthermore, it is shown that this principle is working and the requirements of network bandwidth are discussed.

I. Introduction

Modern robotic systems have complex sensors in order to perceive their environment as good as possible, resulting in complex algorithms, like environment mapping, visual servoing or path planning, to process the sensor data. Integrating complex algorithms on computing platforms that are hard real-time or part of a mobile or energy-efficient robot, is not a straightforward task. The complex algorithms generally are non-hard real-time, being not able to guarantee that they finish before a deadline is met, making scheduling them together with hard real-time processes impossible. Furthermore, complex algorithms tend to use many resources, like CPU, memory or storage, which as a result consume quite some power. Mobile or energy-efficient robotic systems do not have such amount of energy available in their batteries.

It would be possible to use dedicated hardware devices containing loop controllers, which steer the actuators, for example, ELMO Whistle[1] or Maxon Servoamplifier[2]. However, during development of loop controllers, one wants to be free to implement and experiment with own controllers, which is not feasible when using these hardware devices. Robotic setups that require sophisticated, custom controllers also need their own loop controllers, which are not available as pre-packaged blocks, and also need their own software solutions.

[1]http://www.elmomc.com/products/whistle-digital-servo-drive-main.htm
[2]http://www.maxonmotor.com/maxon/view/content/controls

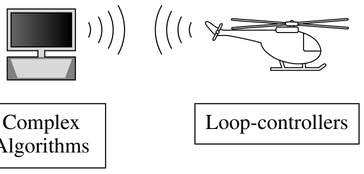

Fig. 1. System overview showing the separation of algorithms.

Another solution to the problem is to offload the complex algorithms to a so-called base station, which is a resource-rich PC that does not have to be mobile or energy efficient. An example of such a setup is depicted in Figure 1, showing a mobile robotic setup, in the form of a helicopter, at the right and the base station at the left. The helicopter and base station are able to communicate wirelessly where the helicopter typically sends location, speed, camera, etc. information, which is processed by the base station. The base station sends new commands to the helicopter as response, so it is able to fulfil its tasks.

For this distributed approach, some software infrastructure is needed. In this work, we use ROS for the complex algorithms running on the rich-resource platform, and our LUNA execution framework [1] for the loop controllers on the embedded platform. Using ROS allows easy combining software parts, and their execution is soft real-time due to the nature of ROS, as that is based on the publisher-subscriber pattern. Programs in LUNA run hard real-time, provided the underlying operating system supports hard real-time execution. However, the connection between these two frameworks is missing.

The focus of this paper is to connect an embedded application to a ROS network. LUNA is used as example framework, resulting in certain design choices, but most of it is generally applicable to any other execution framework or application. Background information, including a discussion on related work and relevant details about ROS and LUNA are provided in Section II. The actual solution to connect LUNA and ROS is discussed in Section III. A proof of principle of discussed in Section IV, followed by this paper's conclusions in Section V.

II. Background

A. Related Work

Combining different frameworks to support different demands w.r.t. timeliness or other properties has been done before. Also several interconnects to specific (real-time) frameworks running on embedded systems and ROS have been made. We review here four different interconnects. Furthermore, we discuss two rapid control-prototyping tools, which are regularly used for embedded hard real-time software design.

R2P [2] is an modular approach to hardware and software for robot prototyping. It focuses on the rapid prototyping of small robots, and uses a CAN fieldbus between the robot computer components, of which each one controls a small robot part, e.g. one motor. It uses ROS for the high-level control, using the straightforward publisher-subscriber pattern. R2P cannot be used in our case, as we need run-time binding of the signals, to prevent the complete LUNA execution engine to be recompiled after every change in the model.

Unity-Link [3] focuses on interfacing FPGA-based controllers to software running on a PC, where ROS is used as middleware. This bridge between ROS-enabled algorithms and hard real-time loop-controller code is quite specific, as the hard real-time code runs on FPGAs. Therefore, it cannot be used for our situation.

Scholl *et al.* [4] deal with connecting integrated wireless sensor nodes to a computer. This work deals with a ROS client only, thus only soft real-time. As we need hard real-time functionality, this approach is *not* usable in our situation.

YARP [5] and Orocos [6], [7] are versatile robot middleware packages, supporting hard real-time and soft real-time behaviour, and extensive ways of configuring and tuning. This results in a quite large footprint, making them not really suitable for UAVs that in general have restricted resources.

LabVIEW[3] is a rapid control-prototyping tool for designing control-system software. By itself LabVIEW is only capable of generating soft real-time signals, as it is running on a regular operating system without hard real-time capabilities. In combination with myRio[4], which is a CPU/FPGA platform, it is possible to let LabVIEW generate setpoints that are used by myRio to generate the hard real-time steering signals for the actuators. LabVIEW is only suitable for modelling the controllers, it is not (easily) possible to model the plant and simulate it together with to controllers to verify the dynamic behaviour of the system.

Simulink[5] is a rapid control-prototyping tool in which a developer draws block diagrams to direct the signal flows to their destination blocks, which use these signals to perform the algorithm calculations. Using the code-generation capabilities, it is possible to have hard real-time execution of the block diagrams on a computing platform. Unfortunately for simulations, the model needs to be annotated with so-called ‘sink blocks’ to visualize the simulation results. This mixes different uses of the models, resulting in a cluttered diagram where the actual (control software) model components becomes hidden between the other non-relevant model contents.

[3]http://www.ni.com/labview/

[4]http://www.ni.com/myrio/

[5]http://www.mathworks.com/products/simulink/

B. ROS

The Robot Operating System (ROS) [8] is a software framework that provides a set of tools and libraries to ease the development of robotic behaviour in software.

One of its strong points is the support of nodes and their interaction via a network of topics, which can be advertised by nodes and subscribed to by other nodes. This helps in building a network of complex algorithms that each provides a part of the overall computations, each using the data that is sent via the topics. This data can consist of sensor information, calculated environment details or planned tasks.

Furthermore, ROS supports a wide range of sensors, varying from simple force, torque or touch sensors to 3D environment sensing sensors, like range finders or cameras. The sensor-driver nodes handle interfacing with these sensors and send the sensors data and control commands through topics.

Usually, a node written in C++ subscribes to a fixed set of topics, which are determined by using their data structures in the code. As a result the data structures of these topics are checked by the C++ compiler, resulting in a robust design. This is due to the compiler being able to match the used fields with the defined fields of a data structure and alert the developers about mismatches, for example a field being removed in the defined structure, or having its data type changed. Additionally, the defined data structures have an MD5 checksum attached to them, calculated using the field information of the structure and thus making the checksum unique. The MD5 checksum is run-time checked with an expected checksum when a node connects to a topic. When these sums do not match, the user is alerted that the expected and defined data structures of the topic do not match.

Due to its complexity and extensiveness, ROS is not capable to provide a hard real-time software environment. The arrival of data on a topic, the scheduled time of a node and so on cannot be guaranteed, as this depends on too many unknown factors. On the other hand, most of the time these things will go as expected or designed, making ROS suitable for soft real-time use.

C. LUNA

LUNA [1] is a hard real-time framework that provides all kind of support for embedded applications, like loop-controller implementations. It is component-based, meaning that functionality that is or is not required can be enabled or disabled, resulting in an as small as possible software footprint.

LUNA also provides a CSP-execution engine, that is capable of executing processes according to the Communicating Sequential Processes (CSP) algebra [9]. The CSP algebra provides mathematical constructs for scheduling concurrent processes and the rendez-vous communication between them. The resulting schedules can be formally verified for correctness, like dead-locks and live-locks. This is used to be able to guarantee that processes are executed before their deadlines, resulting in hard real-time software execution. Another result is that the required execution time can be calculated, which can be used to determine the maximum possible control frequency of the software loop-control implementation for any computing platform.

LUNA-based software is typically developed using MDD techniques, provided by the TERRA tool-suite, making use of the CSP execution engine. As a result, data-flows in LUNA applications are implemented using CSP rendez-vous channels. These channels require both ends to be actively present in order to communicate the data.

III. Design of the ROS to LUNA bridge

Together ROS and LUNA fulfill all requirements to design the software for a complex robotic system, as described in the previous section. Therefore, integration between both frameworks is required, which is provided by a so-called luna-bridge ROS node. The LUNA application connects to this ROS node and is able via this node to communicate with the ROS network. The system overview is shown in Figure 2. The work that is described in this paper, is emphasised in this figure. The design choices for the luna-bridge node and the requirements of the LUNA application are discussed in the remainder of this section.

As mentioned earlier, communication in LUNA CSP applications is typically implemented using rendez-vous channels. It is required for the integration between LUNA and ROS to provide means to connect these rendez-vous channels to the luna-bridge node. ROS does not have rendez-vous communication, so the connection between ROS and LUNA must deal with the mismatch between regular or periodic communication and rendez-vous communication.

The integration between both framework must be as flexible as possible. This implies the following requirements:

- Any primitive data type needs to be supported, so the users are not limited in their designs by a (small) set of data types.
- Complex data types should be build using these primitive data types, further expanding the support different data types.
- Conversion between LUNA rendez-vous channels and ROS topics must be available, since both are the native means for communication.
- Naming conversion must be provided to connect a LUNA channel to a ROS topic field, as their names are not the same by default.

The LUNA application is located on the embedded platform, whereas the luna-bridge is located on a resource-rich platform, as depicted in Figure 2. The resource-rich platform has access to plentiful resources and computing power. Connecting these platforms requires support to communicate. Using existing networking hardware is the most straightforward way to connect the platforms to a robotic network. Nowadays all computing platforms have access to networking hardware, via a regular LAN port or a WiFi adapter. The latter is suitable for mobile robotic platforms that cannot have a cabled connection to the robotic network. An advantage of using existing network hardware is that other components can make use of this infrastructure as well. For example a video camera can send its video stream over the same network, being able to reach all computing platforms as well.

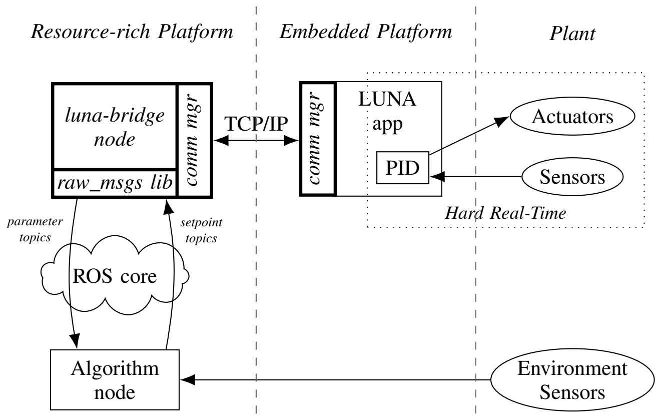

Fig. 2. Architectural overview of the interconnected software platforms.

A. Robotic Network Design

The robotic network, depicted in Figure 2, spans multiple computing platforms. Two of these computing platforms are shown in the figure, connected by a TCP/IP link to depict the physical network topology. This is the minimally required network setup. Of course it is possible to add more platforms to the network if desired.

The resource-rich platform contains the ROS network, to which two nodes are connected. The algorithm node uses the environment-sensor data to perform its complex calculations. In practice it is common to separate complex calculations over multiple nodes, each with their own sub-task to execute. Besides algorithm-related nodes, the ROS network typically contains task-planning related nodes as well, to determine and schedule the short and long running tasks of the robotic system, depending on the purpose of the system and its current environment. The final results of the algorithms, in this situation 'setpoints', are provided via their topics. The luna-bridge node is subscribed to such topics in order to be able to send the results to the embedded platform. Additionally, the luna-bridge node can send parameter values to the algorithm node to properly configure it, depending on the needs of the embedded loop-controllers for example.

The embedded platform contains the hard real-time, embedded loop-controllers. These controllers calculate the steering signals using the results provided by the luna-bridge node and the sensors. Again, only one controller (application) is depicted in the figure, but in practice multiple controllers are typically present, each requiring one or more signals from the luna-bridge node. The example application consists of a simple PID controller, but any other hard real-time controller implementation can be used, depending on the requirements of the actuators and the complexity of the movements of the robotic system.

B. Connection Management

As mentioned in the previous section, it is likely that multiple controllers are present on an embedded platform and/or that these controllers require more than one of the calculated values. Setting up a TCP connection for each controller and its input values, would result in too much resource usage, which especially needs to be prevented on embedded platforms.

Adding a connection manager on both the embedded and resource-rich platforms, solves this issue. The connection manager on the embedded platform establishes a single connection to the connection manager of the luna-bridge. Now the managers are able to collect the data and send it over the single connection. This even allows to bundle data of different topics, and send it using a single TCP packet, further reducing overhead introduced by the TCP headers that accompany each packet.

C. Run-time Topic Binding

As briefly mentioned in Section II-B, the software developer needs to decide at compile-time to which topics the application requires to subscribe to. In other words, the applications are bound to the topics at compile-time. This results in (virtually) no possible mismatches between the actual and the expected data format of the topics, making ROS-enabled programs as robust and smooth as possible with respect to topics.

Unfortunately, the luna-brige node has no definitive information about the topics it is going to connect to when it gets compiled, as this depends on the LUNA applications, which are designed separately (LUNA is used as a framework, i.e. a pre-compiled library). It would be possible for a developer to collect all required topics, depending on the topology of the robotic network and the applications that are connected to it, but that would become an error-prone and tedious task. Especially during development, when the embedded applications tend to change a lot until they are working properly. Another disadvantage would be that the luna-bridge needs to be recompiled every time one of the embedded applications require the information of another topic. In other words, compile=time binding to topics is not useful, and run-time binding to any of the available topics is required by the luna-bridge node.

The solution is to use a custom way of connecting to a topics, circumventing the default subscription method of ROS. This is already done by some software parts of ROS:

- The Python wrappers for ROS[6], as an Python application is not, per se, compiled before being executed, so connecting to topics happens at run-time.
- Some of the ROS commands, that are part of the *ros_comm* package[7], are able to show information about the available topics and monitor them.

The simple ros_comm commands are written in Python, basically providing means to easily access the information that is provided via the Python wrappers. But the more complex commands, like *rosbag*, are written in C++, and therefore use some other mechanism to circumvent the compile-time topic subscriptions. These commands use the rather unknown *topic_tools* package, which has the following description[8]:

> "Tools for directing, throttling, selecting, and otherwise messing with ROS topics at a meta level. None of the programs in this package actually know about the topics whose streams they are altering; instead, these tools deal with messages as generic binary blobs. This means they can be applied to any ROS topic."

This package provides the undocumented *ShapeShifter* API, that contains the actual mechanics to handle the topics as binary blobs. An overview containing the *ShapeShifter* class and related classes is depicted in Figure 3. Using its *getMessageDefinition()* function, it is possible to obtain the string representation of the names and data types of the topic fields at run-time for any topic. This provides some basic means to verify that a topic indeed contains the expected data and its location in the binary blob, as required by the embedded applications. Using the data types of the field, its actual data value can be extracted/decoded from the binary blob.

Due to the complexity it makes sense to provide a separate, reusable software library that provides the functionality of decoding the binary blobs of topics, so any application can just simply request the required data of any topic field. This is exactly what the *raw_messages* library[9] is doing (see Figure 2). It provides a *MessageDecoder* class that is able to decode the raw messages receive from topics and a *MessagePublisher* class to publish messages. The luna-bridge node, makes use of this library to subscribe to or publish at any topic (field) at run-time, when requested by an embedded application connected to it.

The overview of the classes involved by luna-bridge in receiving messages from arbitrary topics is shown in Figure 3. When an application connects to the luna-bridge node (depicted as *LUNABridge*) and requests to receive data, *LUNABridge* instantiates a *TopicListener* and adds it to its *listeners* list. The *TopicListener* subscribes to an arbitrary topic using the *ShapeShifter* class. The *callback()* function is called when a message is received, with the *ShapeShifter* class as message format, which is provided to the *MessageDecoder*. The *MessageDecoder* is able to provide values of the topic fields to *LUNABridge* in order to send them to the LUNA application.

Sending messages from a LUNA application into the ROS network is more straightforward compared to receiving them. When the LUNA application requests to send a value, *LUNABridge* uses the *MessagePublisher* to create a message that can be sent over a topic. The *MessagePublisher* is configured with the topic and field type, this information is used when one of the *publish()* functions is called. These functions check whether the configured type is matching, as an additional safety check. If this is the case, it creates the corresponding ROS message and simply sends it to the topic.

In order for this to work, the message types have to be known in the luna-bridge node on beforehand. All ROS build-in native types are currently supported in the luna-bridge node. Therefore, the messages can only contain a single field consisting of such a build-in type. Each type has its accompanying *publish()* function. Without this limitation the amount

[6]http://wiki.ros.org/rospy
[7]http://wiki.ros.org/ros_comm
[8]http://wiki.ros.org/topic_tools
[9]https://github.com/veger/ros-raw_message

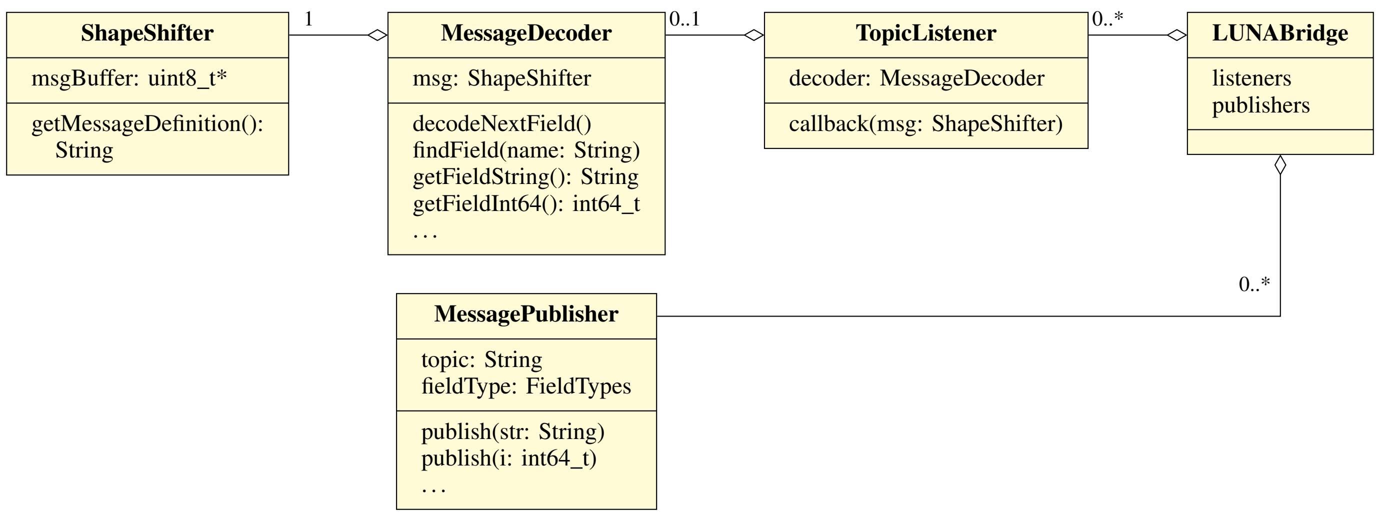

Fig. 3. Class Diagram showing the relation between the different classes used by the luna-bridge node.

of *publish()* functions would exponentially increase, as all combination would require their own function. Of course, it is possible to add several application-specific message types that are required by the robotic system, for example, a message type containing location and rotation information seems sensible to include.

IV. Testing

A. Essential tests

A straightforward test is performed to determine that everything works as intended, i.e. all luna-bridge communication paths are tested, both to the LUNA application and to the ROS network. The data flows of this test are depicted in Figure 4.

The embedded platform is implemented using a Gumstix Overo Fire module[10], running Linux version 3.2.21 and Xenomai 2.6.3. A regular desktop PC with an Intel i7 860 2.8 GHz CPU and 12 GB RAM is used for the rich-resource platform. The desktop PC has Kubuntu 14.10 and ROS Indigo, build from sources[11], installed on it. Both computing platforms are connected using a 1 Gbit/s dedicated LAN.

First, the user is asked for 2 values on the embedded platform. These values are sent to the ROS network via the luna-bridge node on the resource-rich platform. A node on the ROS network performs a complex calculation (a simple addition) using these values. Finally, the result is then sent back to the embedded platform via the luna-bridge, and showed to the user for verification.

The test showed that the luna-bridge node is working as intended. It is able to connect the rendez-vous channels to ROS topics, two channels to send the user input and one channel to obtain the result of the algorithm.

B. Network occupation

The bandwidth required to communicate with the luna-bridge node is neglectable in comparison with the bandwidth

[10]https://store.gumstix.com/index.php/products/227/

[11]https://github.com/veger/ros-builder

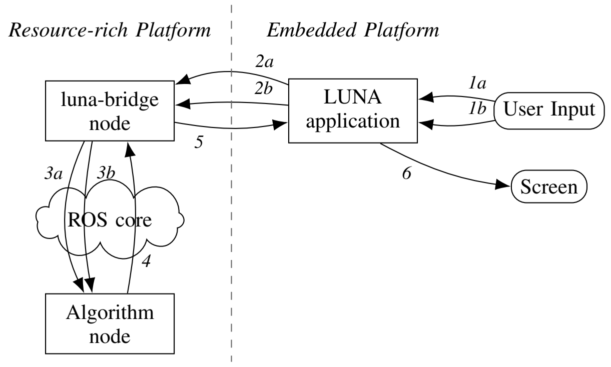

Fig. 4. Data flow diagram of the test setup.

of (WiFi) networks. A typical camera is able to send about 25 to 30 frames per second, but depending on the capabilities and complexity of the algorithms a resource-rich platform might not be able to process the frames at this same rate. Assuming that 25 loop-controllers get their setpoints at a 10 Hz rate, somewhat lower than a camera is able to send the frames, it results in 250 packets per second. In a most simple implementation, these setpoints are not bundled into a single TCP/IP packets but send separately. These 25 packets, each 1500 bytes (or 12,000 bits) large, result in a 3 Mbit/s bandwidth requirement. We expect that this can be covered by a WiFi connection, taking into account the often lower channel capacity due to distance, wheather etc. So, we assume at least 5-10 % of the theoretical maximum of a normal WiFi connection to be available.

A TCP/IP packet is large enough to bundle about 90 setpoints, assuming that 16 bytes of data are required per setpoint. So by bundling the 25 setpoints of the example calculation into one TCP/IP packet, the bandwidth requirement is reduced by a factor 25. Other means like data compression, reducing the setpoint frequency, and so on can further reduce the required bandwidth. Hence it is safe to conclude that the required bandwidth to connect to the luna-bridge node is neglectable.

Since the network has lots of left-over bandwidth available, it makes sense to connect the 'Environment Sensors' to the network as well, in order to communicate their measured data. For example, if the mobile robotic systems has a camera attached, its video stream can be sent over the network. For example, a full HD video stream with H.264 encoding has a bit rate of about 4.5 Mbit/s. Using a 54 Mbit/s WiFi network to connect a mobile robotic system, like a quadcopter, with the resource-rich platform, would have enough bandwidth to theoretically stream up to 12 full HD video streams. However, in practical outdoor situations, such video streams might be too much to let the bundle of setpoints be transported in a normal way. Note that this paper focuses on the network bridge, leaving development of (quad-copter) applications out of the paper's scope.

It is assumed in the previous discussion that the connection strength is optimal; in practice this is often not the case. By implementing quality of service (QoS) mechanisms, it is possible to drop frames from the video streams in order to make sure that the luna-bridge communication is getting through when the available bandwidth is too low. Besides dropping frames when the connection drops, the mobile system needs to be robust enough to handle temporarily connection drops, so it will not continue flying to a direction and get lost, or crash into obstacles.

V. Conclusion

The essential test showed that the principle is working as intended. It is possible to send and receive data between a hard real-time embedded platform to a resource-rich platform. Thereby solving the problem of offloading resource intensive algorithms from a embedded platform to a resource-rich platform.

It is also argued that the available bandwidth is amply sufficient for the required communication between the platforms. There is even additionally bandwidth available to transport data from environment sensors, like a camera, over the same network. In order to use this additional bandwidth safely, quality-of-service mechanisms may be necessary.

Therefore, this solution aids in adding more and more computing platforms to a robotic system, as the the luna-bridge deals with the data transport between the computing platforms.

Future work is to test the luna-bridge solution in more complex research setups. We plan to use it in our ProductionCell, i.e. a scale model of a molding machine, to combine vision with basic control of the setup. A second use-case is to use it on our drones in order to offload the vision algorithms to a ground station.

Further future work is to develop QoS in the form of producing warning messages when the available bandwidth is below a given threshold, such that this can be used in applications to update their strategy (for example, let a quadcopter hoover or even fly back until the connection is strong enough again).

Additionally, we plan to extend our tool chain to support this work, such that it is possible to model the connection between the embedded and resource-rich platforms and include this information into the generated code.

The principle of connecting the platforms is generally applicable, so implementing it for other tool chains should not impose too many problems. So, whether ROS and LUNA are used or other frameworks, or adding support to other (MDD) tool chains does not matter.

References

[1] M. M. Bezemer, R. J. W. Wilterdink, and J. F. Broenink, "LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework," in *Communicating Process Architectures 2011, Limmerick*, ser. Concurrent System Engineering Series, P. Welch, A. T. Sampson, J. B. Pedersen, J. M. Kerridge, J. F. Broenink, and F. R. M. Barnes, Eds., vol. 68, no. WoTUG-33. Amsterdam: IOS Press BV, Nov. 2011, pp. 157–175.

[2] A. Bonarini, M. Matteucci, M. Migliavacca, and D. Rizzi, "R2p: An open source hardware and software modular approach to robot prototyping," *Robotics and Autonomous Systems*, vol. 62, no. 7, pp. 1073–1084, 2014.

[3] 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*, 2013, pp. 3899–3906.

[4] P. M. Scholl, M. Brachmann, S. Santini, and K. Van Laerhoven, "Integrating wireless sensor nodes in the robot operating system," in *Cooperative Robots and Sensor Networks 2014*, ser. Studies in Computational Intelligence, A. Koubaa and A. Khelil, Eds. Springer Berlin Heidelberg, 2014, vol. 554, pp. 141–157.

[5] G. Metta, P. Fitzpatrick, and L. Natale, "Yarp: yet another robot platform," *Int'l J. on Advanced Robotics Systems*, vol. 3, no. 1, pp. 043 – 048, Mar. 2006.

[6] H. Bruyninckx, "Open robot control software: the OROCOS project," in *Robotics and Automation (ICRA), 2001. IEEE International Conference on*, vol. 3. IEEE, 2001, pp. 2523 – 2528.

[7] 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*, vol. 2, Sep. 2003, pp. 2766 – 2771 vol.2.

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

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