Connecting Two Robot-Software Communication Architectures: ROS and LUNA

Communicating Process Architectures 2016, Copenhagen, DK

W. Mathijs van der Werff, Jan F. Broenink

University of Twente
CTIT institute, Robotics & Mechatronics
Enschede, Netherlands
Introduction — Motivation

• Two trends in robotics — Conflicting!
  • More complex algorithms
    - Computer vision, area mapping, planning
  • More light weight, energy efficiency
    - Mobile robots, unmanned aerial vehicles (drones)
• Possible Solution
  • Offloading algorithms to base station
    - Development of algorithms easier
    - More resources, like computer power
    - Easier upgradable
  • Connection between two environments needed
    - Algorithms
      - Robotic Operating System – ROS
    - Loop Controllers, i.e. hard-real time code
      - LUNA Universal Network Architecture -- LUNA
Introduction — Some Background

- **Hard real time**
  - Controlling robots, i.e. fast mechanics
- **LUNA run-time framework**
  - Hard real-time execution, precompiled
  - Design Flow
    - Graphically designed CSP processes in TERRA, and verified
    - Code generated, linked to LUNA lib
- **ROS – Robot Operating System**
  - Open source / large community
  - Publisher - Subscriber pattern: nodes and messages
  - Design Flow
    - Design algorithms and message types
    - Connect nodes via message exchange
    - (re) compile
Introduction — Prototype, earlier made

• Prototype ROS-LUNA bridge made
  • Algorithms in ROS and hard real-time controllers in LUNA
  • Problem: ros :: Publisher pub = n. advertise <template T>(”topic”, 10);
    - so source-code level in ROS to be connected to precompiled library in LUNA
  • Bezemer et al. at ETFA 2015

• Prototype
  • Based on ShapeShifter class
  • Integer LUNA → ROS
  • Limited support messagetypes
    - only basic datatypes
Design and Implementation

- **Essential Requirements**
  - Versatile / Reusable
  - Compiled program
  - SRT - HRT connection
    - Asynchronous data connection

- **Overview**
  - Communication
  - LUNA
  - ROS

---

![Diagram of ROS-LUNA bridge](image)

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.

---

2.1. Connection management and Communication protocol

The communication protocol specifies how data is sent between ROS and LUNA. A straightforward 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 up to 9000 bytes 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, could be used, it would also increase overhead and dependency on third party implementations.
ROS-LUNA Bridge Architecture

- **Overview**
  - Communication
  - LUNA
  - ROS
Implementation – Communication

• Communication Protocol
  • Serialise, Deserialise
    - to fill up TCP/IP packets
    - use bandwidth effectively
    - tailored solution
    - reduce overhead
  • Extendible
  • ROS channels
    - >>

Figure 7. Block diagram of ROS-LUNA bridge.
Implementation — specific channels in LUNA

- LUNA — ROS channels
- Allows modeling in TERRA
- Channel modifications
  - non-blocking write to ROS
    - from HRT to SRT
    - 2 data buffers
  - blocking read from ROS
    - synchronisation...
- Non-blocking read
  - using ALT: ROSread [] SKIP
Implementation — ROS topic listeners...

- **ROS — Topic Listeners**
  - topic = data to transport
  - run-time topic binding
    - specific Publisher
  - specific configuration
    - through the network

- **Implementation**
  - ShapeShifter class
    - publish & subscribe
    - without specifying data type
  - Needs specific
    - serialiser, deserialiser
    - RuntimeBindingPublisher
    - extended TopicListener

In Figure 7 a diagram is depicted showing a schematic overview of the bridge using this ROS network. A Runtime Binding Helper Service is connected following communication to the ROS network. A Runtime Binding Publisher is made available to be sent to the LUNA application. 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 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 the data and publish it.

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 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
Testing

- **Initial Tests**
  - on bandwidth
  - packet loss

- **Verification, Performance**
  - RBP - RuntimeBindingPublisher
  - Performance
    - Publishers
    - Subscribers

- **Demonstration**
  - timing
  - robotic system
Initial Tests

- **Packet loss**
  - to mimic WiFi
- **Additional traffic**
  - network sharing

![Graphs showing response time against loss and additional traffic]
Verification tests

- **Verify RuntimeBindingPublisher**
  - correct serializing / deserializing
- **auto-generated ROS structure of test**
  - time stamp test:
Performance Tests - Publishers

- **Five different implementations of ROS publishers**
  - generic ROS Publisher in C++
  - generic ROS Publisher in Python
  - RuntimeBindingPublisher with prior msg info
  - RuntimeBindingPublisher without prior msg info
  - simplified RuntimeBindingPublisher in Python

- **Tests**
  - average of 100 tests
  - per test 50 x init and publishing of 100 samples
  - 10 tests in 1 run
    - 100 tests in 1 run makes ROS core crash
  - On intel i5@2.53 GHz, 4 GB RAM, Ubuntu 15.10, ROS Jade
Publishers

• **Initialisation**
  - RBPc++ slowest
    - due to external Python helper node
    - RBPc++2: not needed as used from previous call
  - Python slower than C++
    - RBPPython slower than Python
      - additional fu calls needed

• **Runtime**
  - RBPs are comparable
    - only initialisation is different
  - RBP slower than C++
    - due to additional var name look ups
  - Python slowest

**Figure 8.** Performance comparison between different Publisher types.

The measured delay consist, besides the delay introduced by the type of subscriber, also of delays imposed by the publisher present in the time stamp generation node (/timestamp), 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. 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 topics connected to a subscriber. The test initialises both the Python implementations (C++ and Python), after which the measured delay is published on an additional topic (/res). The measured delays include the delays due to network and publisher. 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 determine which Publisher type is the fastest. The measured delay due to external Python helper node, not needed as used from previous call, is done distributed over 4 topics (/sub test 1 to /sub test 4), these topics are connected to both a runtime binding and a normal subscriber are present and connect to one of these subscribers.

**Figure 10.** In initialization both Python implementations seem fastest, followed by the runtime binding C++ implementation is slowest during runtime: it has to iterate over all the callback of the subscriber is based on a template in the normal C++ implementation, while initializing the normal C++ subscriber during runtime: for example, registering the message type is correct.

As third test four different implementations of Subscribers are tested: normal Subscribers (C++), 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. 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 topics connected to a subscriber. The test initialises both the Python implementations (C++ and Python), after which the measured delay is published on an additional topic (/res). The measured delays include the delays due to network and publisher. 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 determine which Publisher type is the fastest. The measured delay due to external Python helper node, not needed as used from previous call, is done distributed over 4 topics (/sub test 1 to /sub test 4), these topics are connected to both a runtime binding and a normal subscriber are present and connect to one of these subscribers.

**Figure 9.** Publishing message type, a custom type containing a header and two fields (refer to Figure 9). Publishing fields. The test initialises both the Python implementations (C++ and Python), after which the measured delay is published on an additional topic (/res). The measured delays include the delays due to network and publisher. 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 determine which Publisher type is the fastest. The measured delay due to external Python helper node, not needed as used from previous call, is done distributed over 4 topics (/sub test 1 to /sub test 4), these topics are connected to both a runtime binding and a normal subscriber are present and connect to one of these subscribers.
Performance Tests - Subscribers

- Four different implementations of ROS subscribers
  - normal subscribers in C++ / Python
  - extended TopicListener in C++ / simple runtime binding in Python

Tests

- custom type: header and 2 float64
- average of 100 test, for initialisation
- 6,000 msg @ 200 Hz:
  - time stamp send as float64
  - published over 4 topics, connected to 2 nodes
    - 1 node C++, 1 node Python
    - both have runtime binding and normal node code
  - received data
    - elapsed time is measured and put in 2nd float64
  - analysed
    - in analysis node
  - delay: publisher + network + subscriber
    - network delay can be subtracted as common factor
Subscribers

- **Initialisation**
  - C++ slowest
    - due to tasks others do at runtime
    - like registering the callback
  - Python seems to optimize
    - due to repeating of runs
- **Runtime**
  - C++ slowest
    - has to iterate over description fields
  - Python faster than RBPc++
    - due to optimizations
- **Overall conclusion**
  - C++ faster than Python
  - RBPc++ is in between
Demonstration Tests

- Robotic setup: vision in the loop
  - our favorite JIWY test setup
    - pan-tilt gimball, DC-motor driven
- RaMstix embedded board:
  - Gumstix over fire, Linux 3.2.21, Xenomai HRT patch 2.6.3
  - FPGA for PWM pulse generation and encoder pulse counting
- Notebook for ROS
- Tests
  - initialisation
  - timing
  - real action
Initialisation JIWY setup

- **Initialisation**
  - of ROS nodes and topics
  - via the ROS-LUNA bridge
  - ROS topic / message graphs
    - before, after LUNA app connects

- **Tests**
  - as expected
Timing tests JIWY setup

- Only ROS-LUNA bridge over the network
- two tasks concurrently
  - transporting images
    - video file and camera images
  - hard-real time task @ higher freq: 500 Hz
    - writing packages to ROS @ 62.5 Hz
- In LUNA
  - priority via PRI ALT
Timing Tests Results

- **Tests**
  - timestamps recorded
  - variation (= jitter) calculated

- **Results - Jitter**
  - at LUNA side
    - HRT Jitter: 0.265 %
    - SRT Jitter: 0.373 %
    - both timed via timer channel
  - on PC - ROS
    - SRT notify: 18.3 %
    - ROS monitor: 21.7 %

- **Results - delays**
  - Round trip 31.5 ms, large variation
    - ROS -> LUNA 15.5
    - inside LUNA 13.4
    - back to ROS 2.6
Complete Robotic system

- **Controlling Robotic Setup**
  - controllers @ 100 Hz
- **System**
  - overview
  - architecture in TERRA

![Diagram of a vision-in-the-loop system distributed over two systems.](image)

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

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 the 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.

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

![Image processing](image)

Visualiztion (Soft real-time)

Controller (Hard real-time)

Sensor data

Actuation

Plant

PC / ROS

Network

Embedded system

Hard real-time

Soft real-time

ROS_imageprocessing

Setpoint_receive

Buffer_send

PositionControllerPan

PositionControllerTilt

IOManager

ROS_JIY_W_monitor

Setpoint data

Visualization data

Video stream

Camera

Controller

Plant

Sensor data

Actuation

Hard real-time

Soft real-time
Results, tracking a green blob
Conclusions and Recommendations

• **ROS - LUNA bridge runs**
  • SRT - HRT connection in a natural way
  • Reusable / Flexible
    - at the price of some more delay
  • Demo application suffers from delay

• **Recommendations**
  • Complete support in TERRA
    - to avoid modifying generated code to use ROS-channels
  • ROS runtime binding
    - can be used in other HRT systems than LUNA
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 were 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. 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.

**Figure 15 Setpoint Receive Blok**

- to read from Im Proc and produce setpoints
Figure 17: signals supporting the JIWY movie

- Pan setpoint vs. Pan encoder
- Tilt setpoint vs. Tilt encoder

4. Conclusion

In this paper, a way to 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 a 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 to run in real time on an embedded system, and allow the execution of CSP. Furthermore, combining these two environments allows to offload parts of the software of a robotic setup to a base-station: this allows the processing inside the robotic setup to remain lightweight and more energy efficient, while complex algorithms can 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 favor of using compiled code. When simple runtime binding subscribers are tested, it appears that the Python implementation is faster, compared to 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, a compilable environment is preferred, as the other parts will benefit from compilation. Verification tests show correct serialization of the messages during runtime, and allow to test whether a ROS environment contains 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...