R5-COP

Reconfigurable ROS-based Resilient Reasoning Robotic Cooperating Systems

Implementation and Documentation of ROS Bridges

(PIAP) Adam Dąbrowski, (PIAP) Rafał Kozik,
(SMR) Heico Sandee, (PIAP) Sebastian Tabaka

Project	R5-COP	**Grant Agreement #**	621447
Deliverable #	D31.40	**Dissemination Level**	PU
Contact Person	Adam Dąbrowski	**Organization**	PIAP
E-Mail	adabrowski@piap.pl	**Date**	31.10.2016

Document History

Document History			
Ver.	**Date**	**Changes**	**Author**
0.1	25.10.2016	First version	Adam Dąbrowski
0.2	25.10.2016	ROS driver to Universal Robot	Heico Sandee, Adam Dąbrowski
0.3	26.10.2016	Content on ROS2 bridge	Adam Dąbrowski, Rafał Kozik
0.4	27.10.2016	Content on JAUS bridge	Adam Dąbrowski, Sebastian Tabaka
0.5	28.10.2016	Finalized for internal review	Adam Dąbrowski
1.0	31.10.2016	Applied review feedback, finalized	Adam Dąbrowski

Table of Contents

Cover page **1**

Document History **2**

Table of Contents **4**

List of Acronyms **5**

1 Introduction **6**
1.1 Summary (abstract) . . . 6
1.2 Purpose of this document . . . 6
1.3 Partners involved . . . 6

2 Bridging considerations **7**
2.1 Drawbacks of bridging and possible alternatives . . . 7
2.2 ROSbridge . . . 8

3 Bridging with JAUS **9**
3.1 JAUS standard . . . 9
3.2 Differences and similarities with ROS . . . 9
3.3 State of ROS-JAUS bridges . . . 9
3.4 ROS-JAUS bridge project . . . 10
3.5 Overview of architecture . . . 10
3.6 Adding new messages . . . 11
3.7 Limitations and shortcomings . . . 11

4 Bridging with ROS2 **13**
4.1 ROS2 overview . . . 13
4.2 Differences and similarities between ROS and ROS2 . . . 13
4.3 ROS2 ros1_bridge package . . . 13
4.4 Submitted improvements . . . 13
4.5 Drawbacks and limitations of ros1_bridge . . . 14

5 Bridging with native security protocol **15**
5.1 Overview . . . 15
5.2 Differences and similarities between RTRC and ROS . . . 15
5.3 Drawbacks and limitations . . . 15

6 Interfacing with Modular Link Controller **16**
6.1 Overview . . . 16
6.2 Interfaces . . . 16
6.3 Advantages and drawbacks . . . 16

7 Bridging ROS with Universal Robots control platform **18**
7.1 Introduction . . . 18
7.2 ROS driver for Universal Robot . . . 18
7.3 Architecture of the driver and ROS interface . . . 19
7.4 ROS messages . . . 20
7.5 Benefits . . . 21

8 Conclusions **22**

List of Acronyms

ROS	Robot Operating System
JAUS	Joint Architecture for Unmanned Systems
SAE	Society for Automotive Engineers
RTRC	Real Time Robot Control
DDS	Data Distribution Service
QoS	Quality of Service
IDL	Interface Description Language
TCP	Transmission Control Protocol
TCPROS	(TCP + ROS) TCP Transport layer for ROS Messages and Services
UDP	User Datagram Protocol
MLC	Modular Link Controller
MLF	Modular Link Framework

1 Introduction

1.1 Summary (abstract)

Robotic engineers often don't have the luxury of developing an entire system from the start with unified design. Instead, they need to integrate existing parts, some of which can be closed to development for various reasons, such as having a proprietary software. Similar problem occurs when updating middleware to a newer, incompatible version, such as in the case of upgrade from ROS to ROS2. The preferred way to do the migration in a complex system would often be gradual, package by package, while maintaining functionality and stability of the system during the migration. Creating a generic driver to a platform in a way most suitable for ROS also requires a specialized interface. While many aspects of such integrations and updates are specific to a particular case, some issues are common, especially when the migration involves well-defined standards. In the document, work on several different ROS bridges is described. Analysis on each of these bridges limitations is provided as well.

1.2 Purpose of this document

This document is an outcome of task 31.5: Interfacing to non-ROS middlewares. The work presented in this deliverable aims to tackle each of scenarios where bridging is required: when integrating with a non-ROS standard protocol, when integrating with a non-standard protocol, when bridging two systems through a common interface, when creating a robotic platform driver and when migrating up to ROS2 In the scope of the task, ros1_bridge packet in ROS2 repository has been further developed to support services, and ROS-JAUS open source project has been published. The document can be of benefit to anyone interested in using or developing ROS bridges.

1.3 Partners involved

Partners and Contribution	
Short Name	**Contribution**
PIAP	Most of the document content
SMR	Content on ROS driver for Universal Robot
TUBS	Review

2 Bridging considerations

When considering a bridge between two middlewares, it is important to understand their conceptual design as well as behavior of their interfaces. Often, differences between two potential bridge ends make the idea of bridging unfeasible, because it is not possible to find a satisfying translation of protocols and concepts. Some challenges of bridging include:

- Different communication patterns, such as publish-subscribe vs direct messaging. One end of a bridge may require negotiation of connection with a peer, while the other end has no connection concept at all.
- Incompatible information packaging, such as when one system reacts to a particular package of information and requires the entirety of it, while the other one separates the data into several messages.
- Lack of support for same communication guarantees and quality of service settings.
- Semantic differences in message types, such as when one system sends a command to control motors with meters per second of desired speed, but the other uses percentage of maximum effort. It is not possible to accurately translate between the two without additional knowledge (of maximum speed).
- Very different sets of standard messages. The envisioned purpose of the middleware has strong influence on which messages are supported in the library - the sets can have little in common, so there is a need to resort mostly to custom messages

With all these challenges present, directly bridging from protocol to protocol is often not a valid solution. To connect two middlewares of vastly different design, an additional abstraction layer is required to hold states and understand concepts and parameters of both worlds. Even with such a layer, there can be an inevitable loss of information or quality when translating from one middleware to another.

2.1 Drawbacks of bridging and possible alternatives

With two nodes (or peers) communicating over common middleware, there is a certain performance characteristic, depending on the medium and communicated data characteristics. With bridge between two middlewares, the time and resource cost of processing roughly doubles, as message is now sent and received twice (one for each middleware). On top of that, there is an added cost of processing and translating the message in the bridge code, but it is usually not very expensive. If one of middlewares is, performance-wise, much better than the other, the advantage will be lost completely for the faster middleware, resulting in significant worsening of performance from the perspective of the faster node.

Sometimes, a valid alternative is to adapt the other node to the same middleware, enabling direct communication with no bridge. On the other hand, in some cases, performance is not a key aspect of the system or is not substantially affected by the use of the bridge, such as in the cases of low-intensity, low-bandwidth communication. It is important to analyze the system and weight benefits and drawbacks of bridging, as bridging in general is not a good solution to every problem.

2.2 ROSbridge

Rosbridge_suite package[1] is an important bridge in ROS ecosystem. It allows to connect ROS and non-ROS programs through media like websockets, with the use of JSON API. It is also used to connect two separate ROS systems in order to get around the issue of single master. There are certain limitations coming from the use of JSON, such as issues with bridging data that requires high resources and performance. There are some issues that can be solved to improve the package[2].

Since the package is already used and well known, we have decided to direct our efforts in bridging towards the new ros1_bridge in ROS2 instead.

[1] http://wiki.ros.org/rosbridge_suite
[2] https://github.com/RobotWebTools/rosbridge_suite/issues

3 Bridging with JAUS

3.1 JAUS standard

The Joint Architecture for Unmanned Systems (JAUS) defines communication protocols for unmanned vehicle systems as well as some of their internal components. It employs service oriented architecture approach and defines a set of standards, which are owned by Society of Automotive Engineers (SAE) under Aerospace Standards Unmanned Systems Steering Committee. A JAUS system is made up of subsystems connected to a common data network[3].

An important part of the standard is JAUS Service Interface Definition Language, which is used to define what a service does and how it is intended to operate. JAUS Service Sets encompass thematic groups of messages.

3.2 Differences and similarities with ROS

- JAUS Service Sets are somewhat similar to ROS packages.
- JAUS uses event based messaging similar to publish and subscribe of ROS and performs online discovery as well.
- There are major differences between standard messages in JAUS and ROS, which creates difficulty in mapping. An example is a standard mobile base control message, which in ROS contains target speeds, while in JAUS it specifies percentage of effort. Automatic translation between such these two corresponding messages is not possible without knowledge of maximum speeds available to the controlled platform.
- JAUS provides means, such as JSS Core Access Control, to facilitate exclusive controlling of services such as mobility. In ROS, there are no built-in ways to achieve it, though it is possible to add it on top of the middleware i.e. by introducing sessions like in OpenRAVE.
- JAUS is a closed standard while ROS is open. Many other characteristics follow from these, such as community type and direction of development.
- JAUS implementations are not free.

3.3 State of ROS-JAUS bridges

Existing ROS-JAUS bridges are proprietary or outdated. None of these solutions seems to have seen any significant use. Following libraries were identified:

- OpenJaus jROS[4] - must be bought with OpenJaus SDK, which is not open despite the name. There is little or no information on anyone using the bridge, which means that it's either not used or used in closed projects. jROS contains ROS packages for each of JAUS Service Sets (i.e. Core, Mobility, Manipulators), which provide mapping of JAUS messages to ROS topics.

[3] http://openjaus.com/support/understanding-jaus/
[4] http://openjaus.com/products/jros/

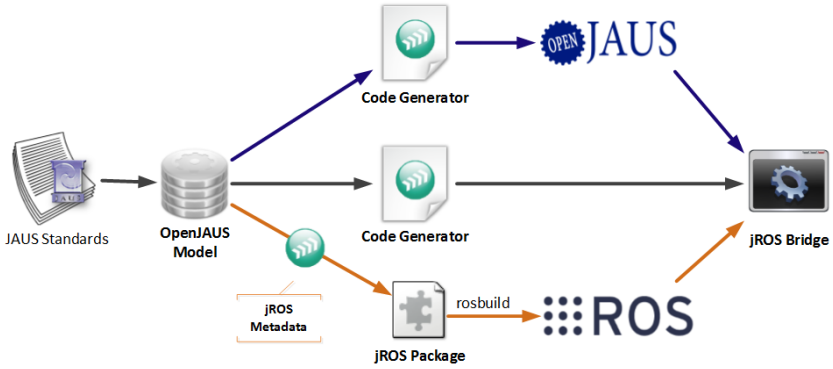

Figure 1: Architecture of jROS. Source: Open JAUS LLC.

- ROStoJAUSBridge - there is a report on Army Research Laboratory concerning creation of the bridge, dated 2012. There is a reason to think it was implemented - however, no software is available[5].
- University of Arizona ROS-to-JAUS - targets Ubuntu 12.04 and ROS Groovy, lacks supports 3 years ago, doesn't seem to build. Supports only a some messages for intended use with a vehicle employed in a project that spawned the bridge[6].
- Case Western Reserve University cwru_jaus - very old, targets old versions, 6 years since last commit, experimental phase[7].

Since available solutions are not adequate, a bridge was implemented as a part of the task.

3.4 ROS-JAUS bridge project

The ros-jaus-bridge open source project is available `https://github.com/piappl/ros-jaus-bridge`. It contains implementation of the bridge library as well as several messages and it is open to extension through adding new messages. Users can configure both the ROS topics and JAUS component addresses as they wish.

3.5 Overview of architecture

While the project doesn't provide bridge generation, it offers more flexibility in message translation to potential users in form of message plugin system. Translations can be defined between JAUS and ROS in an extendable way, allowing to potentially use some of standard messages on both sides (code generation results in non-standard messages). This gives the user more power, but also requires some degree of skill to add a new message and compile it. Simple interfaces and examples are provided to make the task easier.

[5] http://www.arl.army.mil/arlreports/2012/ARL-MR-0812.pdf

[6] http://catvehicle.arizona.edu/bridge-communication-between-robot-operating-system-ros-and-joint-architecture-unmanned-systems-jaus

[7] https://github.com//cwru-robotics/cwru-ros-pkg/tree/fuerte/cwru_experimental/cwru_jaus

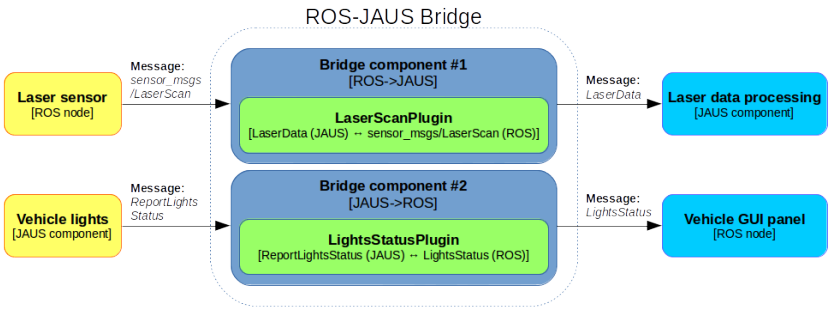

Figure 2: Example use of the bridge. Source: PIAP

A simplified architecture is illustrated on the figure below:

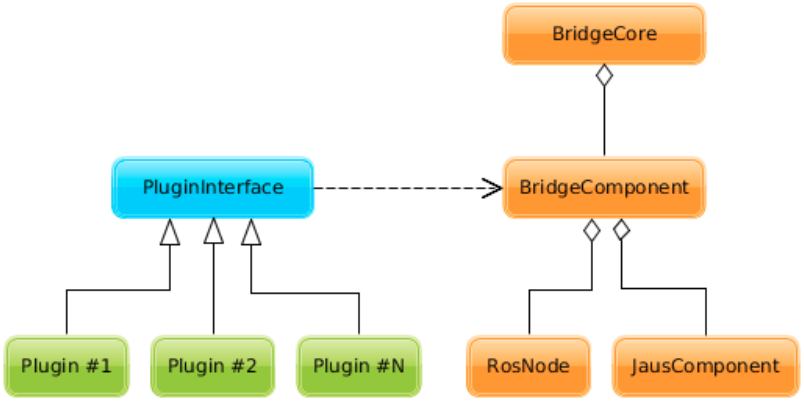

Figure 3: Simplified bridge architecture. Source: PIAP

3.6 Adding new messages

Adding a new message happens through creation of appropriate plugin, in which the author defines the translation, implementing the provided PluginInterface. The user implements both sides of translation according to provided interface. Such plugin is then loaded by the bridge, which recognizes the type and sets the message channel up.

3.7 Limitations and shortcomings

The bridge does not solve the problem of interfacing between ROS and JAUS in a complete way. The differences in standard messages are causing further complications. For example, the JAUS WrenchEffort message is constructed in a following way:

```
int16 PropulsiveLinearEffortX
int16 PropulsiveLinearEffortY
int16 PropulsiveLinearEffortZ
int16 PropulsiveRotationalEffortX
int16 PropulsiveRotationalEffortY
int16 PropulsiveRotationalEffortZ
int16 ResistiveLinearEffortX
```

```
int16 ResistiveLinearEffortY
int16 ResistiveLinearEffortZ
int16 ResistiveRotationalEffortX
int16 ResistiveRotationalEffortY
int16 ResistiveRotationalEffortZ
```

The WrenchEffort message is a rough equivalent of ROS cmd_vel topic geometry_msgs/Twist message, but gives percentage of effort for each joint instead of values in meters per second. One solution is to create a custom ROS message mirroring WrenchEffort and subscribe to it in the robot node, which understands what percentage effort means in terms of speed, knowing the maximum speed of its motors. In this case, the bridge does not provide transparent integration, but demands implementation adjustments on both sides of the bridge. It does eliminate the compile dependency (i.e. the ROS node does not need to link with JAUS library), but the dependency on the structure of the original message remains. Fortunately, JAUS messages are subject to standard and not expected to change.

Another shortcoming is that the bridge requires the user to implement translation for each missing, required message. From one point of view, the power to interpret and adapt the message is left to the user, on the other hand, an automatic generation would reduce the required time for implementation. Further work on the project, whether carried out by original authors or any other contributors, should include message generation but leave the possibility of manual translation as well.

Finally, a performance penalty is introduces, as it is always the case with bridges. The bridge itself is lightweight and does not incur any unnecessary penalty to delivery time. It should be valid to use to most common use cases.

4 Bridging with ROS2

4.1 ROS2 overview

An extensive ROS2 description and evaluation has been presented in the D31.20 "ROS2.0/DDS Evaluation".

4.2 Differences and similarities between ROS and ROS2

Some shortcomings of ROS are listed in D31.30 "Transport Layer Middleware", D31.20 is also good source of in-depth comparison between ROS and ROS2. The middlewares differ mostly in communication layer, with ROS2 employing DDS instead of native TCPROS. On the other hand, there are multiple similarities, coming from the common publish-subscribe pattern and, unsurprisingly, from ROS2 being the continuation of ROS, with ROS2 design aimed to provide as much compatibility as possible.

Differences in communication layer are fundamental and prohibitive of direct communication. Thus, a bridge is required to connect ROS systems to ROS2.

4.3 ROS2 ros1_bridge package

Aware of the importance of interoperability between old and new version, the developers of ROS2 have created the ros1_bridge package [8], which connects ROS and ROS2 topics together. It does so by listing all topics and matching them between ends based on names and types. However, some aspects of the ROS2 DDS such as Quality of Service cannot be translated by the bridge, since ROS doesn't support them. ROS2 nodes communicating with ROS nodes must relax their expectations and default to default reliable communication settings, as these best fit what TCPROS provides.

4.4 Submitted improvements

PIAP team keeps contact with ROS2 developers and volunteered to extend the bridge to support services (request/reply calls). An issue [9] has been raised and the following exchange resulted in adding new functionality to the ros1_bridge package. The bridge, on top of relaying messages as previously, now connects requests and replies both ways, as illustrated:

[8] https://github.com/ros2/ros1_bridge
[9] https://github.com/ros2/ros1_bridge/issues/33

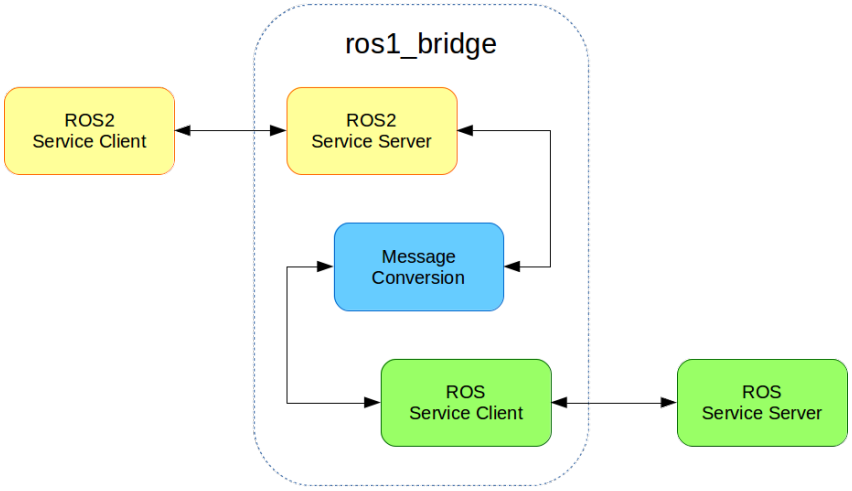

Figure 4: Services in ros1_brige: ROS client to ROS2 server. Source: PIAP.

To create a bridge for ROS service, the ros1_bridge creates a ROS2 service server and a ROS service client. The ROS2 service server listens for requests from other ROS2 nodes and forwards these requests to the ROS service server. After a response is received, it is forwarded back to the client. Bridges for ROS2 services are created in the similar fashion. Message conversion can be challenging, as they may contain complex types.

4.5 Drawbacks and limitations of ros1_bridge

Drawbacks of the ros1_bridge are as expected: loss of Quality Of Service settings over the bridge and a relatively minor performance hit. Similarity of the middlewares, with one being the continuation of another, removes problems that are commonly associated with bridging such as differences in sets of messages or incompatible communication models. An important issue is the current maturity of the bridge package (alpha), which is similar to that of ROS2 in general. It is expected to develop and stabilize with the subsequent releases. From our experience, the bridge seems to be a promising, valid solution for ROS2-ROS1 interfacing. The most important consideration for integrators is proper understanding of what QoS loss means for their systems: when it is acceptable to integrate with the older ROS node through the bridge; and when it is worth the effort to port the old node to the new middleware.

5 Bridging with native security protocol

5.1 Overview

Real Time Robot Control (RTRC), developed and used at PIAP, is a representative non-standard security robot protocol. Built on top of UDP, it provides connection logic and parametrization of communication properties.

5.2 Differences and similarities between RTRC and ROS

Property	**ROS**	**RTRC**
Communication Model	Publish-subscribe	Connection oriented
Message generation	From idl files	None
Confirmation mechanism	TCP	Custom, configurable
Focus fields	Autonomous, research	Teleoperated, security
Availability	Open standard	Closed standard

Native security protocol is conceptually very different from ROS. Semantic messages can be exchanged only after connection is negotiated and they are always intended for a single recipient. This and other characteristics of RTRC can not be modeled by a protocol translation bridge. To properly integrate RTRC and ROS systems, a communication abstraction layer has been introduced, able to understand and handle both paradigms. Through the system of capabilities, events and notifications important to registered logic layer listeners, information is exchanged between middlewares.

The architecture of communication layer and more of the rationale behind this particular solution is described in detail in D31.30 ”Transport layer middleware”. The solution extends beyond RTRC and ROS, providing support for ROS2, rosbridge and (partially) JAUS.

5.3 Drawbacks and limitations

The solution of using an additional abstraction of communication layer is commonly used in cases where multiple implementations are supported. Such is the case with emerging ROS2 and their rmw library, abstracting DDS implementations. While benefits of supporting various protocols in a transparent way are obvious, there are drawbacks of this approach as well. The most important one is that it introduces complexity. The architecture of such system is by necessity more sophisticated. For all new custom messages, there is an effort required to implement them in the abstract layer as well, which needs to be done by the middleware developer. Even in an accommodating, well-designed architecture it comes as an added cost. The cost can be accepted for the control and power that such approach gives, but it is important to analyze whether the associated effort is worth the benefit: other reasons might weight in on the decision. Creating this kind of architecture is expensive and it should be clear in the analysis that no other approach (i.e. bridging, porting, dropping support for the other middleware) is preferable.

6 Interfacing with Modular Link Controller

6.1 Overview

Modular Link Controller, a major part of Modular Link Framework, is a system that abstracts the platform specifics and allows to operate on functionality (skill) level. It is described in multiple D36.x deliverables. The MLC is relevant to bridging, as it allows to connect robots of previously incompatible interfaces to control systems. Putting out a common interface, it lowers the cost of of communication, upgrading from many-to-many architecture with multiple robots and control systems to many-to-one plus one-to-many architecture. In this way, MLC acts as a middleware between two ROS systems, which are expected to interface through it and not directly.

6.2 Interfaces

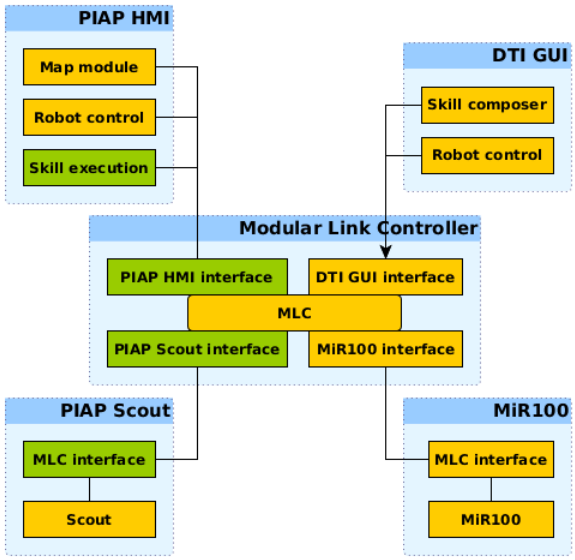

Figure 5: Interfacing through MLC: green nodes were added during integration. Source: PIAP.

The integration of PIAP Scout and HMI with MLC, described in D31.20 chapter 6, has been extended to include the following functionalities:

- Listing and executing MLC skills
- Engaging the emergency stop
- Receiving the robot status
- Receiving and displaying a list of known waypoints on the map

6.3 Advantages and drawbacks

The MLF offers a solution to a problem of integrating ROS systems, allowing to execute skills over multiple platforms. The cost of using MLF is of course adding another layer where errors

can occur, but what is less obvious is that there is still a substantial effort needed in order to interface with MLF. It comes from the fact that ROS has only a small set of standard messages, which in no way covers the needs of various systems build on ROS. The consequence is that ROS systems usually have a substantial number of custom messages which they operate on. Two such systems will have different custom messages and the challenge is, again, to translate one set to another. For example, a RobotInformation message required by PIAP HMI, containing a sort of introduction data of the robot and facilitating multi-robot operations, was not previously supported by MiR100 platform or the MLF itself. It was added during the integration, which imposed cost for all 3 involved systems. The best solution would of course be to use standard messages for interfacing, but right now this would severely limit the functionality of the entire system. The issue with small set of standard ROS messages was also pointed out as problematic during the recent ROSCon 2016. Considering this, the MLF offers a good solution for interoperability and empowers a more generic approach to robot control.

7 Bridging ROS with Universal Robots control platform

7.1 Introduction

To deploy generic robot components in ROS on a real system, it is essential to have a robot driver. There are several examples of ROS drivers for industrial robot platforms, most of them have been developed as part of the ROS Industrial project[10] and have been created according to the ROS Industrial Driver specification[11].

A great advantage of these drivers is that the ROS interfaces are standardized, so that a developer can control a robot without the need to understand its specifics. However: this standardization comes with a cost. Since the standard has to work on a wide range of industrial robots, the drivers cannot rely on any functionally already present in the robot controller. This means that the drivers only allow communication on joint level: they accept sampled reference trajectories for each joint and report the joint state. For a simple application where the robot has to move from point to point in Cartesian space, developers are already required to implement some form of inverse kinematics and motion planning themselves. These tools are available in ROS, but are often generic solutions, while many robot platforms come with dedicated and tested solutions for that platform that have much better performance.

7.2 ROS driver for Universal Robot

An analysis of the performance of the existing ROS driver and possible alternatives were analyzed before[12], but even though the analysis addresses some important performance issues and proposes new implementation, it still adheres to ROS Industrial Driver specification and communicates on joint level with the robot. In addition however, it also offers an interface where the developer can send commands in the native script language of the robot.

SMR has developed a ROS driver for the Universal Robot platform[13], based on generating script commands for each motion using the low-level driver proposed in the analysis to execute them on the robot.

[10] http://rosindustrial.org/
[11] http://wiki.ros.org/Industrial/Industrial_Robot_Driver_Spec
[12] T. T. Andersen, "Optimizing the Universal Robots ROS driver," Technical University of Denmark, 2015.
[13] http://www.universal-robots.com/

7.3 Architecture of the driver and ROS interface

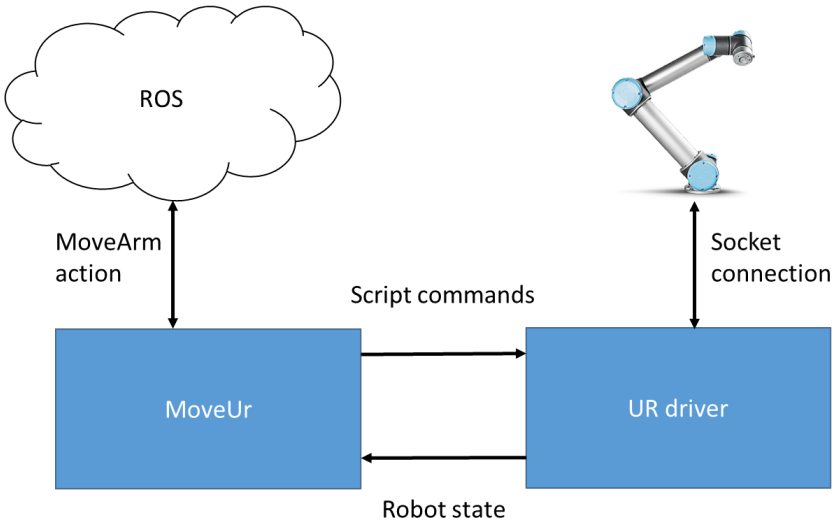

Figure 6: Schematic layout of the Universal Robot driver. Source: SMR.

The driver consists of two components:

1. UR driver - a low-level driver that translates the robot state from a socket connection to ROS topics and that can send script commands to the robot controller using a socket interface. An existing package ur_modern_driver [14] is used for this.
2. MoveUr - a high-level driver that exposes a robot-independent interface that can be used to set goals for the robot.

The ROS interfaces are as follows:

1. Script commands - a ROS topic with String type.
2. Robot State - a collection of ROS topics, including messages on joint position, cartesian position and wrenches.
3. MoveArm action - ROS Action interface, describing a trajectory.

The flow of the driver is roughly as follows:

1. External process sends a goal to MoveUr.
2. MoveUr translates this goal in a script command that the Universal Robot can execute.
3. This script command is published to the UR driver, which sends it via socket message to the robot.
4. During execution, the UR driver records the state of the robot and streams this to MoveUr.
5. MoveUr monitors the robot state and returns the result of the goal to the external processes.

[14] https://github.com/ThomasTimm/ur_modern_driver

7.4 ROS messages

MoveArm action interface, with MoveArm[] waypoints is a vector defining a trajectory:

```
# The path consists of MoveArm messages, which can either be
# Cartesian goal definitions or joint goal definitions.
# Velocities, accelerations, (efforts) and goal tolerances can be
# specified for each waypoint
MoveArm[] waypoints
---
# Result definition
int16 IMPROPER_DEFINITION=-2
int16 FAILED=-1
int16 CARTESIAN_POSITION_REACHED=1
int16 WRENCH_REACHED=2
int16 JOINT_POSITION_REACHED=3
int32 result
sensor_msgs/JointState joint_state
geometry_msgs/PoseStamped tcp_pose
string tcp_frame_id
---
# Feedback
```

Each waypoint can be either a CartesianWaypoint:

```
## Goal definition
# reference_frame defines the frame and pose in which the goal is specified
# tcp_frame specifies the frame and offset w.r.t. that frame
# The goal is that reference_frame.pose and tcp_frame.pose coincide
geometry_msgs/PoseStamped reference_frame
geometry_msgs/PoseStamped tcp_frame

# Desired velocity in arbitrary frame in [rad/s] or [m/s]
geometry_msgs/TwistStamped velocity

# Desired acceleration in arbitrary frame in [rad/s^2] or [m/s^2]
geometry_msgs/AccelStamped acceleration

## Tolerances
#Orientation of tolerance is along tcp_frame
geometry_msgs/Twist pose_tolerance

# Wrench tolerance
# Lower and upper limit of the wrench
# If all six values are between the specified limits, this goal has
# succeeded. If no check is desired, both can be zero
geometry_msgs/WrenchStamped lower_wrench_limit
geometry_msgs/WrenchStamped upper_wrench_limit
```

..or JointWaypoint:

```
## Goal definition
# Array with joint names
string[] joint_names

# Array with desired joint positions [rad] or [m]
float64[] positions

# Array with maximum joint velocities [rad/s] or [m/s]
float64[] velocities
```

```
# Array with maximum joint accelerations [rad/s^2] or [m/s^2]
float64[] accelerations

# Array with joint torque tolerance
# Lower and upper limit of the torques
# If all six values are between the specified limits, this goal has
# succeeded. If no check is desired, both can be zero or empty
float64[] lower_torque_limits
float64[] upper_torque_limits

## Tolerances
float64[] tolerance
```

7.5 Benefits

In comparison to existing drivers, the new Universal Robot ROS driver has the following advantages:

- Generating a point-to-point trajectory is much simpler, since the developers only needs to set a number of waypoints either in Joint space or in Cartesian space, rather than having to implement an inverse kinematics solution and trajectory generator to create a sampled reference signal in joint space
- By relying on the trajectory generator and kinematics engine of the robot itself, the motions are more reliable and more accurate, for example when executing a straight line in Cartesian space.
- This driver uses the force feedback functionality of the robot to stop the robot when it touches something. This proves very useful for pick-and-place applications where for example the exact size of the product can vary.
- By using script commands, this driver can easily be extended to include more functionality already present in the robot language. For example: The robot can be put in compliancy mode, where the robot can be made compliant in certain degrees-of-freedom.

8 Conclusions

Several solutions to problems requiring bridging with ROS were introduced in the document. Since in each case a problem is different, approaches vary and represent a wide spectrum of solutions. Developing further an existing, well supported library is preferred to creating a separate solution, so that's what was done in the case of ros1_package. The existing solution for driver was not suitable and so a new ROS interface to Universal Robot was implemented, learning from an analysis of the former library. With a lack of free alternatives to a closed, proprietary library, a GitHub project of ROS to JAUS bridge has been developed. For protocols which are too different to easily translate, a communication abstraction layer was designed to facilitate communication between systems that differ in concepts as well as protocols. Finally, the MLC shows great value in robotic systems integration and has been extended by adding interfaces to PIAP Scout robot and PIAP operator's console.

Bridges and various approaches to bridging have their limitations, which were analyzed case by case. With different standards, there is a challenge in translating messages. For middlewares that differ deeply, like in the case of ROS and native security robot protocol, an additional communication abstraction can be employed, but it is expensive and the need should be carefully considered. All bridges incur some performance penalty, which might be prohibitive for some systems. Only bridges between middlewares similar enough can be fully transparent to the user. The goal was for each of the implemented bridging solutions to provide value. Altogether, the case of ROS interfacing is advanced through the work described in this document.