frontiers in Robotics and AI

CODE
published: 28 October 2016
doi: 10.3389/frobt.2016.00064

Middleware Interoperability for Robotics: A ROS–YARP Framework

Miguel Aragão, Plinio Moreno and Alexandre Bernardino*

LARSyS, Instituto de Sistemas e Robótica (ISR/IST), Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal

Middlewares are fundamental tools for progress in research and applications in robotics. They enable the integration of multiple heterogeneous sensing and actuation devices, as well as providing general purpose modules for key robotics functions (kinematics, navigation, and planning). However, no existing middleware yet provides a complete set of functionalities for all robotics applications, and many robots may need to rely on more than one framework. This paper focuses on the interoperability between two of the most prevalent middleware in robotics: YARP and ROS. Interoperability between middlewares should ideally allow users to execute existing software without the necessity of (i) changing the existing code and (ii) writing hand-coded "bridges" for each use case. We propose a framework enabling the communication between existing YARP modules and ROS nodes for robotics applications in an automated way. Our approach generates the "bridging gap" code from a configuration file, connecting YARP ports and ROS topics through code-generated YARP bottles. We support YARP/ROS and ROS/YARP sender/receiver configurations, which are demonstrated in a humanoid on wheels robot that uses YARP for upper body motor control and visual perception, and ROS for mobile base control and navigation algorithms.

Keywords: robotic middlewares, interoperability framework, YARP, ROS, code reuse, code development automation, code:C++, license: GNU Free Documentation License

OPEN ACCESS

Edited by:
Lorenzo Natale,
Istituto Italiano di Technologia, Italy

Reviewed by:
Juxi Leitner,
Queensland University of Technology, Australia
Fulvio Mastrogiovanni,
University of Genova, Italy
Stéphane Lallée,
*Institute for Infocomm Research (A*STAR), Singapore*

****Correspondence:***
Plinio Moreno
plinio@isr.tecnico.ulisboa.pt

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *03 February 2016*
Accepted: *06 October 2016*
Published: *28 October 2016*

Citation:
Aragão M, Moreno P and Bernardino A (2016) Middleware Interoperability for Robotics: A ROS–YARP Framework. Front. Robot. AI 3:64. doi: 10.3389/frobt.2016.00064

1. INTRODUCTION

Robotics midlewares such as the *Robot Operating System* (ROS) (Quigley et al., 2009) and *Yet Another Robot Platform* (YARP) (Metta et al., 2006) are currently two of the main frameworks for research and development on robotics platforms and are deployed in hundreds of robots worldwide. They play a key enabling role on building complex applications requiring multiple distinct hardware and software tools but are still under active development and far from providing a complete set of functions for general purpose robots. YARP has been more used in the domain of humanoid robots and developmental robotics, where skills such as visual and tactile perception, human robot interaction, dexterous manipulation, and legged locomotion are central, whereas ROS has higher focus on mobile robots and provides more tools on navigation, depth perception and planning. Thus, rather than competitors, these middlewares should be seen as complementary and many robotic platforms may benefit from using functions from both. However, trying to use these software tools simultaneously is not an easy task mainly due to fundamental differences in the communication architecture, i.e., how messages from different software modules are represented and transmitted

between them. Thus, there is the need to address, in a systematic way, the requirements for a proper interoperability[1] framework between YARP and ROS.

As stated by Fitzpatrick et al. (2008), one of the key aspects for the longevity of middlewares is the ability to adapt to the new environments and niches so interoperability can be seen as a step in the right direction. This is basically the goal of this work, to ease the process of interoperability between two specific middlewares, ROS and YARP. We address the interoperability in the context of existing ROS nodes and YARP modules that can bring new skills to a robot by exchanging messages between them. Ideally, the developer (YARP or ROS) should not need to change the existing code and/or write code for "bridging" the gap between the middlewares. We present a solution that avoids changing the existing code and/or writing hand-coded bridges. Since in most of the cases, the user has more familiarity with one of the middlewares (YARP or ROS), we aim at simplifying the programing of data sending/receiving. Issues such as concurrency and real-time communication are not addressed on this work. Our approach is based on a configuration file where the user describes the "bridge" in a multiple inputs to one output fashion. Using the configuration file, our software generates automatically C++ code that enables the communication between existing ROS nodes and YARP modules. Our approach is constructed upon the interoperability *YARP with ROS*, proposed by Fitzpatrick (2016) and developed by the YARP team, which supports the run-time conversion of YARP bottles to ROS messages.

Our framework provides both YARP/ROS and ROS/YARP as sender/receiver use cases. The YARP/ROS use cases illustrate how to visualize in Rviz (a ROS application) (i) the current state of the robot joints controlled with YARP and (ii) the current gaze point of a robotic head. The ROS/YARP use case shows how to send target gaze points from a ROS topic to a robotic head controller implemented in YARP.

2. BACKGROUND AND RELATED WORK

Naur and Randell (1969) define *middleware* as a piece of software that gives an extra level of abstraction to the developer through a layer between the operating system and the applications.

The majority of the communication middlewares are based on an Interface Definition Language (IDL) approach[2] and have several implementations.[3] Although the IDL is an agnostic standard, the implementations require libraries for all the supported languages/operating systems. The dependency on specific libraries has shifted the problem from the IDL to the adoption of a particular implementation. Vendors and designers encourage the users to adhere to a particular middleware while not considering other options, so the interoperability between middlewares is not usually addressed.

In the robotics context, Ceseracciu et al. (2013) describe the middleware as the entity which provides the glue that holds all the software modules together. Furthermore, as noted by Mohamed et al. (2008), a middleware helps collaborative development since each developer may orient its efforts to a specific module. However, in systems with a large number of modules, a high effort is put on the messaging system, in terms of efficiency and coordination mechanisms. In this work, we focus on the communication services that allow to send and to receive information between software components in the YARP and ROS middlewares.

Metta et al. (2006) introduced YARP, an open-source middleware initially designed to provide an abstraction layer to the communications between modules. It has evolved into a multi-purpose middleware that provides libraries, interfaces, and utilities that act as the control system of a robot. Its main showcase is the iCub robot, which uses YARP as its "circulatory system". Recently, the robots Coman and Vizzy have adopted it, adding them to the more than 20 labs that use the iCub and YARP for research. The basic communication objects in YARP are *Ports* that send *Bottles* over the network. The Bottles may be constructed by hand or using the *Thirft Interface Definition Language (IDL)* that allows to define the bottle from a struct (i.e., list of types).

The Robotic Operating System (ROS) is an open-source middleware that had large contributions from the Willow Garage since 2007. As noted by Boren and Cousins (2011), it has grown exponentially since then and turned into the most popular middleware in robotics while doing its first steps into the industry. In addition to the large set of libraries, interfaces, and utilities, ROS provides several state-of-the-art mobile robotics and motion planning algorithms. Its main showcase is the PR2 from Willow Garage, which is the common example of usage of the majority of ROS modules. The ROS popularity has widened the number of robots partially or totally supported and configured over ROS. As described in Quigley et al. (2009), the basic communication objects in ROS are *Topics* that send *messages* over the network. The messages are constructed from a list of types defined in a text file. The allowed types include standard primitive types (integer, floating point, Boolean, etc.), arrays of primitive types, and other types defined in a message file.[4]

The most recent developments by the YARP team, described in Fitzpatrick (2016), allow to communicate bidirectionally (read/write from/to ROS topics) for all the basic types and some of the non-basic types such as images. There are two options to translate types between YARP and ROS: (i) generate ROS-compatible types at compilation time using Thrift ILD or (ii) generate ROS-compatible types at run-time using Bottles.

However, both options require custom code written purposefully for each particular set of messages, which is an effortful process in systems with many communication links. Aspects such as conversion between data types, memory layout of the messages, metadata information, packaging/unpackaging of messages (many-to-one or one-to-many conversions), all have to be hand coded by the developer. Instead, we propose a semi-automated way to generate the interface code. Through a text configuration

[1] Interoperability is defined by Chen et al. (2008) as *the capability of different systems being able to communicate and take advantage of features of both*.

[2] For instance, CORBA and IIOP http://www.omg.org/spec/.

[3] For instance, OpenMAMA www.openmama.org and DDS http://portals.omg.org/dds/.

[4] http://wiki.ros.org/Messages.

file, using a simple interface language, the developer describes just the minimal set of information required to perform the conversion between message formats.

In the following section, we describe the automatic code generator that allows to exchange information between existing YARP modules and ROS nodes.

3. YARP BOTTLE GENERATOR

Our framework is based on a tool that generates C++ code from a configuration text file that describes the set of inputs (i.e., ports/topics and their types), data conversions that may be applied to the inputs, and a detailed specification of the output (i.e., list of types, hierarchically defined). We denote this tool as the *yarp bottle generator*, which after parsing the configuration file creates a C++ file that after compilation and subsequent execution acts as a bridge between YARP and ROS. From the point of view of Software Patterns, our approach acts as a Mediator Pattern, that transforms data between a source and a destination while lowering the coupling level. The generated code allows to communicate between YARP port objects and ROS publisher/subscriber objects developed previously, reducing the maintenance/refactoring load.

The generated code reads data from several inputs (YARP ports or ROS topics) and constructs a YARP Bottle as output. Note that the *yarp bottle generator* has two modes of operation: (i) from ROS topics to a YARP port (ROS–YARP mode) and (ii) from YARP ports to a ROS topic (YARP–ROS mode). On the one hand, reading from/sending to YARP ports is straightforward because our code generator is based on YARP bottles. On the other hand, reading from/sending to ROS topics needs an additional conversion, which is handled by the run-time YARP to ROS converter `yarpidl_rosmsg`, introduced by Fitzpatrick (2016).

Our generator abstracts YARP and ROS developers from dealing directly with interoperability issues. The main concepts of the bottle generator are: the *hub*, the *converter*, and the *output builder*, which are explained in the following subsections.

3.1. Main Concepts of the Generated Code

Figure 1A illustrates the main concepts of the generated code and their interaction for creating the output. The output message is assembled from the information provided by the converters (units and data types conversion between the different systems), constants (additional data required by ROS), timestamps, and counters (time and sequence information required by ROS and YARP). Each converter has a hub associated to it, which collects information from several YARP ports/ROS topics and merges/splits/reorders data elements according to the requirements of the different platforms. In the following, we explain in detail the hubs, converters, and output builder modules.

3.1.1. Hub

A hub reads data coming from several ports/topics and stores the data in a YARP bottle. The data are ordered in the bottle sequentially according to the list of port names indicated in the *configuration file*. The user can define as many hubs as needed in the configuration file according to the needs. The role of hubs is mostly related to (i) the memory layout of the output and (ii) the converter functions to be applied. For instance, a hub reads the data coming from several YARP ports that contain the motor encoder values of a robot head and arms. This hub reads the ports and stores the values sequentially in a bottle. After building the hub bottle, the following step is the conversion.

The hub idea is similar to the YARP Port Arbitrator of Paikan et al. (2014), because the arbitrator and the hub read data from several ports. The main difference is that the hub stores the data in a bottle, while the arbitrator selects one of the ports to be sent according to a rule.

3.1.2. Converter

The converter receives a bottle from the hub and then applies a function to every element of the bottle. The conversion function was designed to perform tasks such as unit conversion (e.g., degrees to radians), string conversion (e.g., lowercase to uppercase), and so on. The current approach is to have a converter for each hub so in the *configuration file* each *hub* has the field "function" where its corresponding converter function is defined. The conversion result is stored in the same bottle received by the converter.

The currently implemented functions include (i) degrees to radians, (ii) string to floating point number, and (iii) empty function. It is possible to add new functions to the *converter* generator in order to extend the set of available *converters*. In order to add a new function, the developer should add the generation code to the class *yarp-bottle-generator/src/dataconvertergenerator.cpp*.[5] A new converter function is described by a unique identifier (i.e., string) that will be used in the *configuration file* and the corresponding strings that are written in the generated code. We implemented the converter function in the C++ language to improve efficiency of complex operations on data intensive types, such as images and videos, in future applications of the bridge. The converter was inspired by the YARP Port Monitor of Paikan et al. (2014), which provides this functionality between YARP ports in run-time.

3.1.3. Output Builder

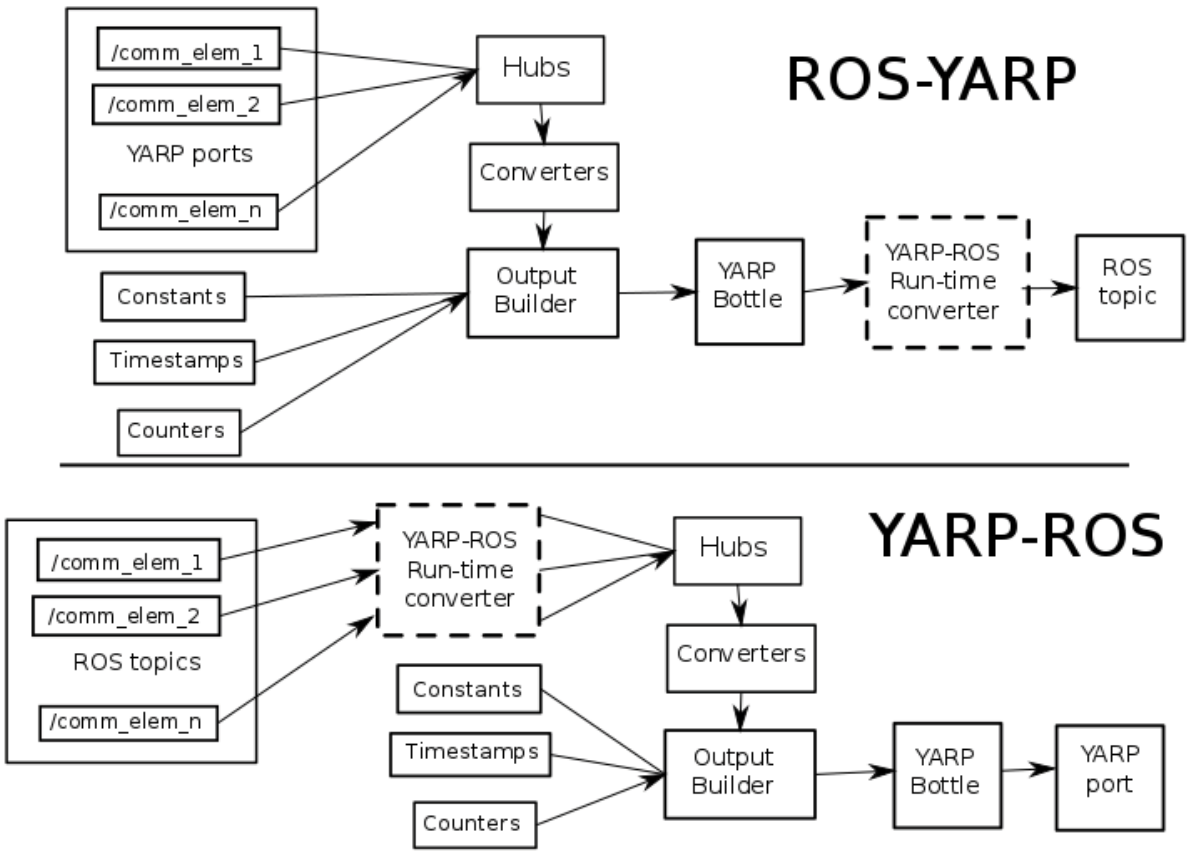
The output builder constructs a YARP bottle using a hierarchical structure of data types, defined in the *configuration file*. The hierarchy of allowed types includes the hubs, constants, timestamps, and counters, as illustrated in **Figure 1A**.

The structure of the hierarchy is comprised by the root's data type and its children, which may have children themselves. The hierarchical data type is denoted as *msg* (borrowing from the ROS message definition), which means that the definition of this type is a list of data types. Following the same idea, the child data type may have an *msg* in it, so the hierarchy expands one more branch.

The remaining data types can be divided into (i) containers (list, hub) and (ii) basic types (timestamp, counter, and single_value). The containers have several values wrapped in one bottle,

[5] http://vislab-tecnico-lisboa.github.io/yarp-bottle-generator/doxygen/doc/html/classDataConverterGenerator.html#a64845e3ce285461b22eabb32af5899ff.

A



B

Section	Name	YARP-ROS	ROS-YARP
[general]	output_name	ROS topic name	YARP port name
	to_ros	true	false
	num_mux	user defined (mandatory)	
	rate	user defined (optional)	
	from_ros_topics	false	true
[mux<integer >]	num_ports	user defined (mandatory)	
	ports	YARP ports (mandatory)	ROS topics (mandatory)
	function	none, deg_to_rad or none_double (mandatory)	
	verbose	true/false (optional)	true/false (optional)
[message]	num_fields	user defined (mandatory)	
	<integer>_type	single_value, timestamp, counter, list, mux, msg	
	<integer>_msg	mandatory if <integer>_type not equal to "mux"	
	<integer>_mux	mandatory if "<integer>_type = mux"	

FIGURE 1 | Top image shows the architecture of the generated code. Bottom table shows the overview of the configuration parameters. **(A)** Generated code detailed architecture for the main use cases. The dashed boxes represent the previously implemented YARP with ROS Fitzpatrick (2016) interoperability tools. **(B)** Overview of configuration file parameters. If the message section has a hierarchical structure, additional sections must be added and the instructions on how to do this are available online (see text footnote 8).

and the basic types are just single elements added to the output. The counter type is a sequential integer, while the single_value is a constant (i.e., string or any number type). The examples in Section 4 will show the application of the components explained in this section.

3.2. Software Architecture

Figure 5A shows the Unified Modeling Language (UML) structure diagram of the yarp bottle generator. The root of the hierarchy is the YarpCodeGenerator, an abstract class from where all the remaining classes inherit the function *generatedCode()*

A

```
[general]
output_name = /motors_inertial_port
to_ros = true
num_mux = 2
rate = 60
ros_msg_name = MotorsInertial
from_ros_topics = false

[mux1]
num_ports = 1
ports = /dummy_head/state:o,
function = deg_to_rad
verbose = false

[mux2]
num_ports = 1
ports = /inertial
function = none_double
verbose = false

[message]
num_fields = 2
1_type = mux
1_mux = mux2
2_type = mux
2_mux = mux1
```

```
float64[] inertial
float64[] encoders
```

B

```
int32 joint
float64 angle
```

```
[general]
output_name = /fakebot/motor/rpc:i
to_ros = false
num_mux = 1
rate = 60
from_ros_topics = true

[mux1]
num_ports = 1
ports = /fakebot_motor_control_ros
function = none
verbose = true

[message]
num_fields = 3
1_type = single_value
1_msg = "set"
2_type = single_value
2_msg = "pos"
3_type = mux
3_mux = mux1
```

FIGURE 2 | Top part shows the YARP–ROS configuration file, and bottom part shows the ROS–YARP configuration file. **(A)** Left side: configuration file example. In this case, there are two sensors: (i) a fake motor control board interface that sends the motor encoder values to a YARP port (mux1) and (ii) a fake inertial sensor that sends readings to a YARP port (mux2). Readings from both sensors are sent to a ROS topic that reads the MotorsInertial message file (right side of the figure). The MotorsInertial file indicates that the data are composed of two floating point arrays. **(B)** ROS–YARP configuration file example. In this case, there is a fake robot, whose motors are controlled by a ROS program. The ROS message contains the joint number and the position value to be commanded, which is shown on the left side of the figure. The right side of the figure shows the configuration file for the ROS–YARP bridge, where there the position values for a motor are read from a ROS topic configured in mux1 (/fakebot_motor_control ros). The read values are sent to a YARP port (/fakebot/motor/rpc:i) configured in the section message.

that produces the strings that correspond to the automatically generated code. The code generation is divided into the following classes: CommonBeginningGenerator, PortMuxGenerator, DataConverterGenerator, BottleCreatorGenerator, ChildGenerator, and CommonEndGenerator (see code examples of **Figures 3** and **4**). The class CommonBeginningGenerator handles the code of the headers and main function code. The class PortMuxGenerator handles the code that connects and reads the data values from the YARP ports/ROS topics for all the hubs (Section 3.1) defined in the configuration file. The class DataConverterGenerator handles the code of the converter function (Section 3.2) for the elements of a hub. The classes BottleCreatorGenerator and ChildGenerator handle the output builder (Section 3.3). The class BottleCreatorGenerator handles the code for building the message and sending it through the network (YARP port/ROS topic), at the top level of the message hierarchy (root of the tree). The class ChildGenerator performs the same task but at the leaves of the message hierarchy. Finally, the class CommonEndGenerator handles the code that finishes the main function. The documentation of the classes described above is available online.[6]

3.3. Managing Middleware Source/Destination

This work is largerly motivated by our robot Vizzy, described in Moreno et al. (2016), which has YARP and ROS running at the same time. YARP controls its upper body, and ROS controls its mobile platform. We will provide examples of both the YARP/ROS and ROS/YARP cases. In addition, we implemented the

[6]http://vislab-tecnico-lisboa.github.io/yarp-bottle-generator.

remaining combinations (ROS/ROS and YARP/YARP) that support a mix of existing ROS topic and YARP port tools. The YARP/YARP case is equivalent to a customized mix of the commands `yarp merge` and yarp sample and the conversion utility of the Port Monitor of Paikan et al. (2014). The ROS/ROS case is equivalent to a customized mix of the topic tools[7] `mux`, `throttle`, and `transform`.

The critical issue for managing the source/destination is that connections between ports/topic are done differently on each middleware. On the one hand, YARP does not assume either read or write status of a port, so the sender–receiver connection is done by the user. On the other hand, ROS topics have to be declared as either publishers or subscribers, so the *roscore* server does the connection sender–receiver according to the topic definition. This difference is considered in our code, connecting to YARP ports when sending/receiving data to/from YARP ports and letting the roscore server manage the connections when sending/receiving data to/from ROS topics.

3.4. Configuration File

The concepts explained in Section 3.1 correspond to sections or parameters of the configuration file. **Figure 1B** summarizes the sections (in square brackets) and their parameters, taking into account the source–destination middleware configuration. More details on how to write your own configuration file are available online.[8]

4. USE CASES, EXPERIMENTS, AND KNOWN ISSUES

This section shows two tutorial use cases and two real use cases of the code generation. The tutorial use cases (Sections 4.1 and 4.2) allow the user to grab the important concepts and test the programs in just one pc. The tutorials' ROS side repository and instructions to run the examples are available online.[9] The real use cases are described in the github wiki of the repository,[10] which are more complex and show the usage of the code generation in real-time robotics applications. Section 4.3 analyses the additional computational resources required by our approach, followed by known issues and limitations section.

4.1. YARP–ROS Case: Reading YARP Devices from ROS

Figure 2A shows the configuration file for reading two YARP ports and converting them into a MotorsInertial ROS message (right side of **Figure 2A**). The generated program shown in **Figure 3** reads from the YARP ports `/dummy_head/state:o` and `/inertial` and creates a MotorsInertial message, which is published on the ROS topic `/motors_inertial_port`. After reading data from the ports, the program converts the angles to radians and fills up the arrays `inertial` and `encoders` of the MotorsInertial message with the converted values.

The configuration file contains four sections: [general], [mux1], [mux2], and [message]. The general section describes

- The output entity name (string `output_name`), in this case a ROS topic.
- The source/destination middlewares flags (`from_ros_topics = false`,`to_ros = true`). In this case, we send from YARP ports to ROS topics.
- The number of hubs (integer `num_mux`), in this case 2.
- The rate of execution of the generated code (number `rate`), in this case 60 Hz.
- The ROS message name (string `ros_msg_name`), for ROS to know what type to expect.

The subsequent sections of the configuration file describe each of the hubs, which in this example are two (`mux1` and `mux2`). The hub attributes are

- Number of ports (integer `num_ports = 1` for both hubs).
- Name of the ports/topics (string `ports = /dummy_head/state:o` for mux1 and `ports =/inertial` for mux2). For several ports, the user writes comma separated names.
- Converter function `function = deg_to_rad` for mux1, which converts the read encoder angles from degrees to radians. The converter function of mux 2, `function = non-e_double` parses the coming data as a floating point number.
- Verbose flag (string `verbose = false`), in case the user wants to print on the screen the values red by the hub.

The rest of the file describes how the output's structure is built. The section `message` needs to know the number of types (integer `num_fields`), which is this case is the number of fields of `MotorsInertial`. In this simple example, each field corresponds to the data stored by the hub, so we just need to define `type` and `mux`, as follows:

- The first type `1_type = mux` corresponds to the data stored in a hub. In order to know which hub corresponds to this field, the line `1_mux = mux2` assigns the data from the inertial sensor.
- The second type `2_type = mux` and `2_mux = mux1` assigns the data from the motors to the second field of the message.

Figure 3 shows the code generated by this configuration file example. The code guarantees that there is successful connection to all the YARP input ports, waiting until all the YARP ports are connected to the bridge ports. If the code connects to all the inputs, moves to a loop that (i) reads the input data, (ii) builds the output message, and (iii) sends the message. It is important to remark that for the output port is not necessary to check the connection status, because ROS automatically connects the publisher to the subscriber as soon as they are available.

4.2. ROS–YARP Case: Controlling YARP Devices from ROS

The right side of **Figure 2B** shows the configuration file for reading the ROS message MotorControl (on the left side of **Figure 2B**) from a ROS topic, converting the message into a YARP Bottle that

[7] http://wiki.ros.org/topic_tools.

[8] https://github.com/vislab-tecnico-lisboa/yarp-bottle-generator#customize-your-own-configuration-file.

[9] https://github.com/vislab-tecnico-lisboa/yarp_bottle_generator_ros_examples.

[10] Items 3, 4, and 5 on https://github.com/vislab-tecnico-lisboa/yarp-bottle-generator/wiki.

```
#include <iostream>
#include <yarp/os/all.h>
#include <math.h>

using namespace yarp::os;

int main(int argc, char *argv[]) {
  Network yarp;
  /*Creation of the reading inputs' ports*/
  BufferedPort<Bottle> receiverBuff1Mux1;
  bool receiver1Mux1Ok = receiverBuff1Mux1.open("/yarp_ros_tutorial/mux1/receiver1");

  BufferedPort<Bottle> receiverBuff1Mux2;
  bool receiver1Mux2Ok = receiverBuff1Mux2.open("/yarp_ros_tutorial/mux2/receiver1");
  /*Creation of the writing output port*/
  Port outputPort;
  outputPort.promiseType(Type::byNameOnWire("MotorsInertial"));
  outputPort.setWriteOnly();
  bool outputOk = outputPort.open("/motors_inertial_port@/yarp/yarp_ros_tutorial");

  bool allConnected = false;
  std::vector<std::vector<bool> > portsConnected;
  portsConnected.resize(2);
  int totalConnections=0;
  portsConnected[0].resize(1);
  portsConnected[0].assign(1,false);
  totalConnections += 1;
  portsConnected[1].resize(1);
  portsConnected[1].assign(1,false);
  totalConnections += 1;
  /*while loop that gurantees that the bridge gets connected to all input ports*/
  while (!allConnected){
    int connectedNumber=totalConnections;
    if (!portsConnected[0][1]){
      portsConnected[0][1] = yarp.connect("/dummy_head/state:o,", receiverBuff1Mux1.getName());
      if (!portsConnected[0][1])
        connectedNumber--;
    }    if (!portsConnected[1][1]){
      portsConnected[1][1] = yarp.connect("/inertial", receiverBuff1Mux2.getName());
      if (!portsConnected[1][1])
        connectedNumber--;
    }    if (connectedNumber == totalConnections)
      allConnected = true;
    std::cout << ".\n";
    Time::delay(1);
  }  int counter = 0;
  /*while loop that reads the inputs, converts data and write output*/
  while(true){
    Bottle* reading1Mux1 = receiverBuff1Mux1.read();

    Bottle* reading1Mux2 = receiverBuff1Mux2.read();

    Bottle mux1;
    Bottle mux2;

    for(int i = 0; i < reading1Mux1->size(); i++) {
      mux1.add(reading1Mux1->get(i));
    }

    for(int i = 0; i < reading1Mux2->size(); i++) {
      mux2.add(reading1Mux2->get(i));
    }
    /*Converting from degrees to radians*/
    for(int i = 0; i < mux1.size(); i++) {
      mux1.get(i) = mux1.get(i).asDouble() / (180/3.1415926);
    }
    /*Converting from string to double*/
    for(int i = 0; i < mux2.size(); i++) {
      mux2.get(i) = mux2.get(i).asDouble()    }
    /* Creation and storage of data in the output Bottle */
    double timestamp = (double) Time::now();
    Bottle message = Bottle();
    Bottle& list_1 = message.addList();
    for(int i = 0; i < mux2.size(); i++) {
      list_1.add(mux2.get(i));
    }
    Bottle& list_2 = message.addList();
    for(int i = 0; i < mux1.size(); i++) {
      list_2.add(mux1.get(i));
    }
    /* Writing the Bottle to the output port */
    outputPort.write(message);
    counter++;
    /* Delay provided by the rate configuration parameter */
    Time::delay(0.01666666666666666);
  }
  return 0;
}
```

FIGURE 3 | Example of the generated C++ code by running the yarp-bottle-generator executable with the configuration file yarp_ros_tutorial example in Figure 2A.

commands a motor of the fakebot robot. The generated program reads the joint and angle values from the topic `/fakebot_motor_control_ros`, then sending the command "set pos joint angle" to the RPC YARP port `/fakebot/motor_rpc:i`. The main differences with respect to the previous example are

- The general section has the flags `to_ros = false` and `from_ros_topics = true`.
- No converter function applied to the hub (`function = none`).
- The [message] generated has two string values that are constant (`1_msg` and `2_msg`).

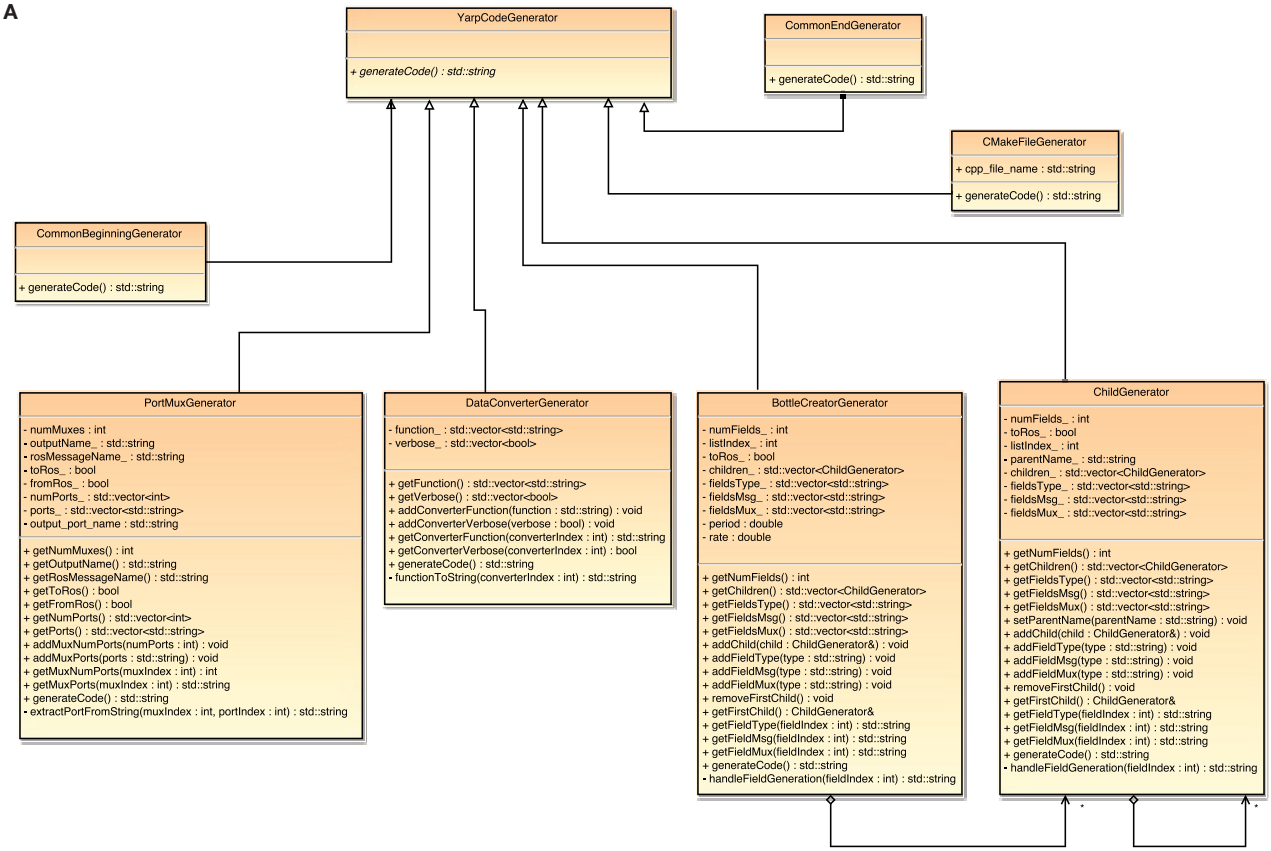
4.3. Computational Performance

Our approach aims at reducing the workload of the programmer (i.e., having to change the original software in order to be compatible with both middlewares), who does not need to care anymore about the message format translation. There is a small amount of computational and network resources allocated to the bridging code, namely, the creation of two additional ports that act as the

```
#include <iostream>
#include <yarp/os/all.h>
#include <math.h>

using namespace yarp::os;

int main(int argc, char *argv[]) {
  Network yarp;
  /*Creation of the reading inputs' port*/
  BufferedPort<Bottle> receiverBuff1Mux1;
  receiverBuff1Mux1.setReadOnly();
  bool receiver1Mux1Ok = receiverBuff1Mux1.open("/fakebot_motor_control_ros@/mux1/receiver1");
  /*Creation of the writing output port*/
  Port outputPort;
  outputPort.setWriteOnly();
  bool outputOk = outputPort.open("/writeBuff ");
  /*while loop that gurantees that the bridge gets connected to all input ports*/
  std::cout << "Waiting for output..." << std::endl;
  bool connectSuccess = false;
  while(!connectSuccess) {
    connectSuccess = yarp.connect(" /writeBuff", "/fakebot/motor/rpc:i");
    Time::delay(1);
    std::cout << ".\n";
  }
  std::cout << "Connection successfuly established." << std::endl;

  int counter = 0;
  /*while loop that reads the inputs, converts data and write output*/
  while(true){
    Bottle* reading1Mux1 = receiverBuff1Mux1.read();

    Bottle mux1;
  /*No conversions*/
    for(int i = 0; i < reading1Mux1->size(); i++) {
      mux1.add(reading1Mux1->get(i));
    }

    for(int i = 0; i < mux1.size(); i++) {
      std::cout << "Dummy break. Sorry for my laziness." << std::endl;
      break;
    }

    /* Creation and storage of data in the output Bottle */

    double timestamp = (double) Time::now();

    Bottle message = Bottle();

    message.add("set");

    message.add("pos");

    for(int i = 0; i < mux1.size(); i++) {
       message.add(mux1.get(i));
    }

    /* Writing the Bottle to the output port */

    outputPort.write(message);
    counter++;
    /* Delay provided by the rate configuration parameter */
    Time::delay(0.016666666666666666);
  }

  return 0;
}
```

FIGURE 4 | Example of the generated C++ code by running the yarp-bottle-generator executable with the configuration file ros_yarp_tutorial example in Figure 2B.

A



B

	Total time read mean (var) ms	Write output mean (var) ms
Use-case Section 4.1 1 port reading, 1 port writing	$2.638 \cdot 10^{-3}(1.252 \cdot 10^{-9})$	$6.401 \cdot 10^{-2}(2.795 \cdot 10^{-7})$
Use-case Section 4.1 with three control boards 3 ports reading, 1 port writing	$2.906 \cdot 10^{-3}(1.633 \cdot 10^{-9})$	$6.254 \cdot 10^{-2}(3.522 \cdot 10^{-7})$
Use-case Section 4.2 1 port reading, 1 port writing	$1.932 \cdot 10^{-3}(4.46 \cdot 10^{-9})$	$0.183(2.548 \cdot 10^{-6})$

C

	Total rate read mean KB/s	Write output mean KB/s
Use-case Section 4.1	22	7
Use-case Section 4.1 with three control boards	36	13
Use-case Section 4.2	0.160	0.99

FIGURE 5 | The top image shows the UML class diagram, the middle table shows the computational overload of the generated code, and the bottom table shows the network overload of the generated code. **(A)** UML class diagram. **(B)** Computational overload. The results show the additional execution time of reading the inputs and writing the output Bottle. **(C)** Network overload.

bridge. We discuss qualitatively the additional memory needed, and we evaluated quantitatively the execution time and network bandwidth required by the two additional ports. **Figures 5B,C** show the additional load in CPU time and network transmission, where we remark the very low additional CPU load and network load. It is important to mention that the computational load is in the microsecond order of magnitude, having only one exception (0.183 ms) because of writing to an RPC port that takes longer than the regular ports. On the one hand, **Figure 5B** shows that the computational overload increases when the number of ports read increases, but the rate is not linear. On the other hand, **Figure 5C** shows that the network overload increases linearly with the number of ports being read.

4.4. Known Issues and Limitations

The code does not have issues in its latest version. The most important issues solved include (i) the bidirectionality support by adding the ROS–YARP communication, (ii) perform the conversion between YARP timestamps and ROS timestamps, and (iii) the type of the ROS message was not defined properly. The complete list of the solved issues is at the github repository.[11] The main limitation of our approach is that we do not have tools for checking the format conversion specified in the configuration file. For instance, if the user in the configuration file associates a wrong type between the middlewares, the code is generated without any warnings/errors. Thus, the user has to verify the association between types. Another limitation of our approach is the applicability area, which is limited to two middlewares only. In the context of robotics applications, the main goal of the middleware is the correct streaming of sensors and actuators data for control systems, so abstractions such as meta-messages that will work for several middlewares are difficult to implement because of the low-level streaming approach (e.g., different data code/decode for the same type of sensor).

[11] https://github.com/vislab-tecnico-lisboa/yarp-bottle-generator/issues?q=is%3Aissue+is%3Aclosed.

5. CONCLUSION AND FUTURE WORK

We introduced a software tool that improves the interoperability between YARP and ROS, reducing the coding effort and changing existing software. Our approach is based on a configuration file written by the user that is the input of our software, which generates C++ code. The code enables the communication between YARP modules and ROS nodes and works in the multiple inputs and one output fashion. We show examples of a robotic platform using YARP and ROS, communicating existing YARP control software with ROS visualization and actionlib software. Future work should address (i) the automatic parsing of the output message from ROS msg files and Thrift IDL files, and (ii) the automatic generation of YARP code for supporting more complex data types such as images and point clouds. In addition, the actionlib ROS library is the standard way to implement remote monitoring over a process, which allows to cancel, read feedback, and overwrite goals to the actionlib server. The automatic generation of code that supports the communication of YARP code with the ROS actionlib servers and clients will ease the interoperability between YARP-based control and the MoveIt motion planning interfaces of Sucan and Chitta (2016).

AUTHOR CONTRIBUTIONS

All authors listed, have made substantial, direct and intellectual contribution to the work, and approved it for publication.

FUNDING

This work was supported by FCT (UID/EEA/50009/2013), partially funded by the FCT Ph.D. programme RBCog and FCT project AHA (CMUP-ERI/HCI/0046/2013).

REFERENCES

Boren, J., and Cousins, S. (2011). Exponential growth of ROS [ROS topics]. *IEEE Robot. Autom. Mag.* 18, 19–20. doi:10.1109/MRA.2010.940147

Ceseracciu, E., Domenichelli, D., Fitzpatrick, P., Metta, G., Natale, L., and Paikan, A. (2013). A middle way for robotics middleware. *J. Softw. Eng. Robot.* 5, 42–49.

Chen, D., Doumeingts, G., and Vernadat, F. (2008). Architectures for enterprise integration and interoperability: past, present and future. *Comput. Ind.* 59, 647–659. doi:10.1016/j.compind.2007.12.016

Fitzpatrick, P. (2016). *Using YARP with ROS*. Available at: http://www.yarp.it/yarp_with_ros.html

Fitzpatrick, P., Metta, G., and Natale, L. (2008). Towards long-lived robot genes. *Rob. Auton. Syst.* 56, 29–45. doi:10.1016/j.robot.2007.09.014

Metta, G., Fitzpatrick, P., and Natale, L. (2006). Yarp: yet another robot platform. *Int. J. Adv. Robot. Syst.* 3, 43–48. doi:10.5772/5761

Mohamed, N., Al-Jaroodi, J., and Jawhar, I. (2008). "Middleware for robotics: a survey," in *IEEE International Conference on Robotics, Automation and Mechatronics, RAM 2008* (Chengdu), 736–742.

Moreno, P., Nunes, R., Figueiredo, R., Ferreira, R., Bernardino, A., Santos-Victor, J., et al. (2016). "Vizzy: a humanoid on wheels for assistive robotics," in *Robot 2015: Second Iberian Robotics Conference: Advances in Robotics*, Vol. 1. (Cham: Springer International Publishing), 17–28.

Naur, P., and Randell, B. (eds) (1969). *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 Oct. 1968*. Brussels: Scientific Affairs Division, NATO.

Paikan, A., Fitzpatrick, P., Metta, G., and Natale, L. (2014). Data flow port monitoring and arbitration. *J. Softw. Eng. Robot.* 5, 80–88.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., et al. (2009). Ros: an open-source robot operating system. *ICRA Workshop Open Source Softw.* 3, 5.

Sucan, I. A., and Chitta, S. (2016). *MoveIt!* Available at: http://moveit.ros.org

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.