TÉCNICO LISBOA

Towards an Advanced Interoperability Framework Between the Robotic Middlewares YARP and ROS

Miguel Aragão [1,*]

[1] *VisLab, Institute for Systems and Robotics, Instituto Superior Técnico, Lisbon, Portugal*

Correspondence*:
Miguel Aragão
VisLab, Institute for Systems and Robotics, *Instituto Superior Técnico*, Lisbon, Portugal, miguelaragao91@gmail.com

ABSTRACT

Some middlewares for robotics are starting to be recognized as the best ones around but there is still not one that provides the complete set of services needed by the many different robotic systems.

Interoperability is a possible solution for this problem since it is the capability of several entities, in this case middlewares, to cooperate by providing services and tools to each other, allowing developers to take advantage of the best features of each middleware with ease.

This thesis developed a framework that aims at the interoperability between two important middlewares, YARP and ROS, easing the process of communication between them by automating the code generation and configuration of the messages exchanged, giving the developer a higher level of abstraction on the communication layer. The framework is available on a public repository on GitHub that contains all the documentation needed to use it. For last, there were also some implementations of the framework configured for real platforms.

Keywords: Robotic Middlewares, Interoperability Framework, YARP, ROS, Code Reuse, Cooperation, Code Development Automation

1 INTRODUCTION

A middleware is a piece of software that gives an extra level of abstraction to the developer as a layer between the operating system and the applications. One can see a middleware as an extension of the operating system because it provides new services and abstractions to the outer layers. The name middleware can be interpreted as the software that is in the middle of the applications and the system, as stated by **Gall** (web page last accessed on 28-04-2015).

A middleware most commonly tries to come up with services regarding communication, threading, concurrency and so on, but the most important for this work is the first one - communication.

A software component, or module, is a node of the network where some computation is performed. Middlewares can be seen as the networks and they are helpful because they allow an easier scaling on modular architectures as the case reported by **Hadim and Mohamed** (2006) describing the necessity of a way to work with several wireless sensors on the same network.

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. As **Mohamed et al.** (2008) say, in such a complex architecture like the robotics one, due to the high number of modules and their complexity, the middleware takes a fundamental role by giving the developer the ability of orienting its efforts to the specific purpose of the module. The high number of modules brings a lot of challenges and issues because it is of extreme importance to be able to exchange messages between them, at the same time that multiple threads, and tasks, are coordinated while dealing with complex hardware, and software, with heavy computation algorithms that robots most commonly need. This work focus on two specific middlewares: YARP and ROS.

Besides the abstraction layer that is created by the middleware, there is another advantage that is of extreme importance in this work - code reuse. This is a concept that is not new and as **Ando et al.** (2005) say - users are not interested in middlewares that force each developer to rewrite every module each time a new robot appears. The current state of the development of technology was only possible because of code reuse. It is not easy to imagine a world where everyone that wanted to code some application would have to implement everything from scratch, so we can agree that code reuse is the base for progress as **Cousins et al.** (2010) say. We do not live forever, so it is fundamental that we can start our work from a previous one in order to keep the technology growth.

Research in many fields, and even science generically speaking, are clear examples where humans gather efforts both in terms of knowledge sharing but also reuse of services and theories started by others to achieve the next step of the development. This idea of cooperation introduces a new concept: *interoperability*.

Interoperability is the ability of making several entities work together. **Chen et al.** (2008) define the concept of interoperability as the capability of different systems being able to communicate and take advantage of features of both. Although this work is clearly about a technological example of interoperability there are other examples like the one presented by **Ford and Colombi** (2007) which reports the same concerns and decisions of architectures on a military context. Interoperability is a global solution for completely different problems and it raises similar issues despite the environment and goals of the field.

As stated by **Fitzpatrick et al.** (2008) one of the keys 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 final goal of the work, to ease the process of interoperability between two specific middlewares, ROS and YARP.

This paper presents a framework that eases the process of creating and configuring YARP bottles convertible to ROS messages by providing a generic and flexible tool that is able to wrap data from several sources, apply functions to the existing data, add hardcoded values to the message and so on.

2 RELATED WORK

As stated by **Metta et al.** (2006) YARP stands for Yet Another Robot Platform and was chosen because the laboratory holding my research and work has more than one robot running over this middleware and also because it has a big community with at least 20 laboratories in the world taking advantage of its features.

YARP is an open-source middleware that exists to support and promote the software development for humanoid robots. The main goal is to be able to minimize the difficulties of developing the infrastructure-level software, achieving bigger modularity and code reuse. These main features promote the development and collaboration at research-level maximizing the progress on the area.

Communication is probably the most important feature of the YARP middleware and it is the one responsible for forcing the use of the same protocols for modules to interact with each other.

The Robotic Operating System commonly known as ROS has seen its prototypes being developed at the Stanford University in the mid-2000s and, with the great efforts and contributions from the Willow Garage since 2007 its growth saw a great boost till the state where it is now, having a great open source community that maximizes the potential of the middleware. An important robot to this big expansion was the PR2 from the Willow Garage, which is the most common example of usage of the ROS middleware modules, although many other robots are compatible and configured over ROS. ROS is a middleware and as **Quigley et al.** (2009) say it is not an operating system as the name might indicate because it provides a structured communication layer above the operating system. At the moment, it is considered as one of the best middlewares for robotics mostly due to its exponential growth as stated by **Boren and Cousins** (2011) and it is starting to give the first steps into the industry. It has a big community and it is constantly under development forcing its users to keep up with the new features and improvements.

Its communication is one of the best organized and it allows code reuse mainly because it is extremely easy to identify what are the structures being passed between nodes because the messages structures are publicly defined on human readable files.

On the last couple of years the YARP development team and community has been taking attention to ROS and developed the ability to send (and receive) messages to (and from) ROS. Someone using the most recent version of YARP should be able to receive messages from topics that are converted from the ROS message type to a YARP bottle. The other way around is also possible these days by converting the bottles into ROS messages (see **figure** 1) but this is where I find the big issue that motivates this work (explained on the **subsection** 2.1). This work is described on the web page containing the work of **Fitzpatrick** (web page last accessed on 17-03-2015).

The *Gazebo YARP Plugins* presented by **Mingo et al.** (2014) are yet another example of the efforts of the YARP community on trying to support many different tools, and, although Gazebo (which is a simulator for robotics) is not a software module written for ROS it has almost native support for it and the compatibility with this kind of tools opens the door for other types of cooperation between ROS and YARP like the simulation of hybrid robots like Vizzy by **Aragão et al.** (web page last accessed on 05-05-2015).

The YARP community also presented the work of **Paikan et al.** (2014) that proposes an object that is responsible for selecting one port as the data source from several input ports - the *Port Arbitrator*. Although this work is not directly related to the YARP with ROS work, the architecture from this thesis was influenced by the *Port Arbitrator's* one.

2.1 MOTIVATION

The major problem faced on the YARP to ROS communication is that the messages on the ROS side are expected to follow a specific structure and YARP users usually do not have the concern of following a prespecified structure (see **figure** 2) nor they have an easy way to do that in case they feel the need to. The module, from the YARP with ROS interoperability, that is responsible for the conversion of YARP bottles into ROS messages, is waiting for a specific bottle (there is only one possible bottle configuration for a specific ROS message) but while ROS wraps a lot of data on a single message, YARP does not, so there is the need to gather data from several different sources and that involves coding and synchronization. Besides that, there are still ordering and type compatibility issues to be solved.

3 YARP BOTTLE GENERATOR

The *yarp bottle generator* is the tool to use in order to automate and ease the process of configuring this type of interactions and it is the most generic solution around because it is able to create all kinds of messages and also, it is possible to configure several different data sources. The benefits are huge, from faster progress on the field to time and resources savings. By abstracting developers from the communication layer on a higher level it is possible to target all their focus to the tasks themselves. Code reuse promotes cooperation and helps teams working together for a common goal. The ease of coding motivates more people to use specific middlewares adding extra value to them.

So the main goal of the work is to ease the creation of the YARP to ROS converter's input but it is possible to separate smaller goals in-between like ordering data, organizing data types, wrap data from several sources and point these data sources to the message fields. In the end, the usage of the *yarp bottle generator*, by filling a single configuration file, eases most of the efforts that are needed to assure all the conditions on the bottle, so the framework fulfills its purpose.

There are 6 concepts that can sum up the entire architecture (see **figure** 3 and **figure** 4) of the *yarp bottle generator*: the *configuration file*, the *code generator*, the *generated code*, the *multiplexer*, the *converter* and the *message builder*.

3.1 CONFIGURATION FILE

The configuration file (see **figure** 3) is the most important part for the developer who wants to use the *yarp bottle generator* because it is the only part of the system that is always mandatory to interact with directly. It is where all the options and configurations are defined so it is fundamental to know how to fill all the available fields in order to generate the needed code.

3.2 CODE GENERATOR

It is the core of the system (see **figure** 3) and it is responsible for parsing the *configuration file* and generate the desired code. The developer should not have the need to change this code but it was written taking into account the possibility of adding new features and improve the existent ones.

3.3 GENERATED CODE

The generated code is the source file of the module that will create the bottle in real-time and it can be divided in 3 major parts (see **figure** 4): the *multiplexers* part, the *converters* part and the *message builder* part. The developer is able to use this code directly or with some specific changes, taking advantage of the skeleton of the code in order to add complex functionalities that can not be automatically generated for now. If the *generated code* suffers no changes it should be compiled with ease.

3.3.1 Multiplexer It is the first part of the *generated code* (see **figure** 4) and its function is to get data from a variable number of ports and wrap it in a single structure. As the name indicates, the *multiplexer* works as a common multiplexer, where ports are the inputs and the output is the structure holding all the data. It is possible to configure more than one multiplexer, so the *configuration file* has a field where the developer indicates how many multiplexers he wants to create (see **figure** 5). The data on the output structure will be ordered by the same order as the port names were indicated on the *configuration file*. In most cases the data on the several input ports should be from the same type, so all ports should have the same kind of data (only doubles for example), which is a limitation for now, and should be improved in the future work.

3.3.2 Converter The converter is an entity that allows the developer to apply a function, to the output of each multiplexer, like converting the units, performing an operation or any kind of computation, and so on. The current approach is to have a converter for each multiplexer (see **figure** 4) so in the *configuration file* there is no separated section for *multiplexers* and *converters*. This way the developer only has to care about 1 section where it is possible to configure the fields for the *multiplexer* (the number of ports and its names) but also the fields for the *converter* (see **figure** 5). In order to understand a more high-level view of the converter, it has an input which is the output structure of the *multiplexer* and an output which is the structure after the function was applied to its data (see **figure** 4).

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*. To add the generation code to the class it will need a string to be the unique identifier of the function (the name that we can use in the *configuration file* to select the function) and the strings that should compose the *generated code*.

3.3.3 Message Builder This is the most complex part (see **figure** 4), and its configuration (an example of a *configuration file* on **figure** 5) shows us that. It has several fields but not all are mandatory. The goal of the *message builder* is to define the structure of the bottle and how to fill that structure with data. It is possible to configure several different sources for each field according to its type. In the *configuration file* the section *message* is where the developer defines how to build the bottle and it has only one mandatory field which is the number of fields (see **figure** 5) of the message (this value should match the number of fields of the ROS message that the output topic expects). The number of fields will affect what the parser on the *code generator* is expecting to read.

3.4 DOCUMENTATION

The documentation and examples needed to understand how to fill the *configuration file* can be found at `https://github.com/vislab-tecnico-lisboa/yarp-bottle-generator`.

4 EXPERIMENTS

4.1 YARP TO ROS

This demonstration is a specific one and it focus on a particular ROS message: the *joint_states* message. This is just an example of how generic the module is and how easy is to configure the *generator* to work with different robots. It solves two different case scenarios that will be described on the next paragraphs.

Vizzy is a robot that was built from scratch on VisLab and it is a quite special one. It has a humanoid upper body but a mobile base composed by a *Segway* with two wheels. Besides this special configuration, Vizzy still holds another interesting architecture decision. Its upper body runs over YARP due to hardware compatibility and also to be able to run the same modules that the iCub does, but, its mobile base is running over ROS because it has a lot of support for locomotion and mapping features coordinated with object manipulation as stated on the work of **Chitta et al.** (2010). This dual feature (running over YARP and ROS at the same time) is great for taking advantage of the best services that each middlewares has to offer but it originates some problems. One of those problems is that Vizzy being a mobile robot is often out of our field of vision and most of those times it is important to be able to visualize the robot remotely. This is trivial for the robot position in the world and the position of the mobile base components because ROS has a excellent tool for visualization called Rviz. The problem is that Rviz only has direct access to the state of the base and not to the state of the rest of the body because the data is on the YARP side. This situation is in fact fixed with ease by the *yarp bottle generator* because it is able to feed Rviz (see **figure** 6) with the data from the upper body.

The iCub robots that exist around the world are some of the most advanced humanoids these days and their high number of degrees of freedom motivates researchers to work with them in many different areas. The iCub was developed in parallel with YARP which is the middleware that runs under it. Even though YARP and the existent modules provide the needed tools for controlling all these degrees of freedom it lacks of an important feature - motion planning with collision avoidance. It is an useful feature that is not yet supported through the existent YARP modules and most of the demonstrations and works with the iCub would surely benefit from it. There are two options: spend resources and a considerable amount of time working on a new module for motion planning; or look for existent code (code reuse). It happens that the ROS middleware has an already working version of a package that performs exactly what we want, motion planning with collision avoidance developed by **Sucan and Chitta** (web page last accessed on 05-05-2015) and called *MoveIt!*. In order to generate trajectories, the service expects two things, the 3D URDF (Unified Robot Description Format) model of the robot and a message containing the current state of the robot joints (the *joint_states* message).

The goals of both these demonstrations is clear - to be able to use ROS services fed with data from the YARP network (**figure** 5 shows the configuration file for the Rviz demonstration). The reason why there are two different robots is to prove the high power of configuration that the *yarp bottle generator* allows and how easy it is to get everything to work (see **figure** 6 and **figure** 7).

The demonstration with the iCub can have a second part where the generated trajectory is sent back to YARP so that the robot can actually execute it. Although not generic, I have prepared a YARP module that accepts an array of joints positions and executes it so it is possible to see the complete process. The main goal is being able to use the ROS *MoveIt!* with the data from the actual robot running over YARP (see **figure** 7).

On Vizzy's case there is not the necessity of sending data from ROS to YARP because the main goal is to be able to visualize the robot remotely and with Rviz it is possible to do that with ease (see **figure** 6). This way we can perform objects manipulation and locomotion at the same time and operate the platform remotely with enough tools to be able to detect errors and weird behavior or simply observe how the robot is performing.

4.2 YARP TO YARP

On the YARP to YARP environment the demonstration is more generic because it will not involve the use of existent services and modules. The idea is to prove its usefulness and for that it needs to aim the applications referred on the last chapter. In this case the *yarp bottle generator* will act as a skeleton generator that is able to ease the development on new modules and not as a bridge to connect different services.

The demonstration here is really simple, and it is more of a practical one, with not that many features to talk about. The idea is to create a new module where the developer needs to compute data from some ports. Lets imagine the case where the developer wants to create a module to estimate the position of the end effector by vision and compare the results with the values that are calculated from the joints positions. In this case the developer will have to code all the computation needed and also the communication between the several entities in order to have the links to the data sources (**figure** 8 shows the section of the generated code where the developer should add the computation).

Although the goals are mostly practical ones there are plenty of advantages that the usage of the *yarp bottle generator* should fulfill. First there is the goal of automating the process of coding by generating the skeleton of the module which is indeed common (apart from the specific configurations that are handled by the *configuration file*) to many different modules. Besides this, the first goal leads to a higher level of abstraction which allows faster development and can even motivate people less comfortable with coding to use the middleware. For last it helps developers avoiding errors on the common tasks (of course some users might want to go deep on the communications established and that is also possible by changing the generated code by hand).

4.3 RESULTS

Although there are no obvious quantitative results (see **table** 1 for a comparison between the number of lines of the *configuration file* and the *generated code*), as stated before, there are several points that I consider important in order to try to evaluate the usefulness and overall quality of the *yarp bottle generator*. Of course there will be more feedback when other people outside the project try to take advantage of its features because they will have less knowledge of what it tries to solve and how it is configured. Still, there are practical results and some considerations that can be presented and analyzed in order to classify the success of the demonstrations.

In the case of the YARP to YARP demonstration, the code is generated with success (see **figure** 8) and it is possible to focus exclusively on the module goal which is the estimation of the end effector position (although that part is not even coded). The configuration file has all the options needed to configure a module not aiming the interoperability with ROS so it becomes trivial to generate the code skeleton of most modules.

MoveIt! Planner		**Rviz Visualization**		**YARP to YARP**	
Configuration File	*Generated Code*	*Configuration File*	*Generated Code*	*Configuration File*	*Generated Code*
30	91	30	80	19	70

Table 1. Comparison between the number of lines in the *configuration file* and in the *generated code*

I consider both use cases really useful and capable of easing the developers life and that is the generic evaluation of the demonstrations. Of course the best tests are yet to come because there is the need of people using the *yarp bottle generator* that are not in the project. The ultimate proof is the positive feedback from the community and for now there has been some but it hasn't been widely spread around the laboratories. In terms of time the benefits are obvious because it is as simple as filling a configuration file with a dozen lines and everything works as expected even if the developer has little background in the YARP to ROS interoperability.

It is an easy to use tool (see **figure** 5), and it proves to be generic enough in order to give more than just simple features to its final users. It was designed with expandability in mind, and the addition of new functions was tested during the execution of the demonstrations because the *generator* was missing some options to allow the generation of code for the YARP to YARP case.

5 CONCLUSION

There are several possibilities to continue and extend this work, and they can be divided in three different groups: aesthetic improvements, efficiency optimization and the addition and support of new features.

The aesthetic improvements focus on specific aspects of the *yarp bottle generator* like how it looks to the developers. The most obvious case is the syntax and the organisation of the *configuration file* which was not tested on a large set of users. With time and usage there might be a better solution in terms of usability and it is for sure something to take into account because the better it feels the better it will be.

My considerations about the efficiency optimization are that it is possible to improve the overall performance of the *generated module*, and for that, the YARP team should incorporate the functionalities provided by this work almost as a native service. This would break the need of having an extra bridge on the network improving both the architecture of the system and the efficiency of the communications of course.

For last, the addition and support of new features is of extreme importance, and with time I believe more ideas will come. For now, I can point some features that I already start to miss, like:

- The possibility of accepting ports on the multiplexer that do not behave as an array of elements of the same type;
- To be able to select specific parts of the multiplexer as a data source (like accessing its elements through its position on the output structure);
- A bigger compatibility in terms of complex types;
- A wider set of functions to use on the *converters*.

In technological terms the final result was pretty useful, and it proved to be robust enough to allow the communication between different robots and different middlewares. I guess a bigger set of testers would have been a better proof, but the results speak by themselves. To define a connection to a ROS service from YARP is now a simpler task, even when the message expects a lot of different data sources and conversions, so the main goal was achieved, and the motivation to continue pursuing the improvement of the middlewares for robotics is even bigger now.

REFERENCES

Ando, N., Suehiro, T., Kitagaki, K., Kotoku, T., and Yoon, W. K. (2005), RT-Middleware: Distributed component middleware for RT (Robot Technology), *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS*, 3555–3560, doi:10.1109/IROS.2005.1545521

Aragão, M., Moreno, P., and Figueiredo, R. (web page last accessed on 05-05-2015), Vizzy's Repository - https://github.com/vislab-tecnico-lisboa/vizzy

Boren, J. and Cousins, S. (2011), Exponential Growth of ROS [ROS Topics], *IEEE Robotics & Automation Magazine*, 18, 1, 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, 5, September, 42–49

Chen, D., Doumeingts, G., and Vernadat, F. (2008), Architectures for enterprise integration and interoperability: Past, present and future, *Computers in Industry*, 59, 7, 647–659, doi:10.1016/j.compind.2007.12.016

Chitta, S., Cohen, B., and Likhachev, M. (2010), Planning for autonomous door opening with a mobile manipulator, *Proceedings - IEEE International Conference on Robotics and Automation*, 1799–1806, doi:10.1109/ROBOT.2010.5509475

Cousins, S., Gerkey, B., Conley, K., and Garage, W. (2010), Sharing software with ROS, *IEEE Robotics and Automation Magazine*, 17, 2, 12–14, doi:10.1109/MRA.2010.936956

Fitzpatrick, P. (web page last accessed on 17-03-2015), Using YARP with ROS - http://wiki.icub.org/yarpdoc/yarp_with_ros.html

Fitzpatrick, P., Metta, G., and Natale, L. (2008), Towards long-lived robot genes, *Robotics and Autonomous Systems*, 56, 1, 29–45, doi:10.1016/j.robot.2007.09.014

Ford, T. and Colombi, J. (2007), The interoperability score, *Proceedings of the Fifth conference on systems engineering research*, 1–10

Gall, N. (web page last accessed on 28-04-2015), Origin of the term middleware - http://ironick.typepad.com/ironick/2005/07/update_on_the_o.html

Hadim, S. and Mohamed, N. (2006), Middleware: Middleware challenges and approaches for wireless sensor networks, *IEEE Distributed Systems Online*, 7, 3, 1–23, doi:10.1109/MDSO.2006.19

Metta, G., Fitzpatrick, P., and Natale, L. (2006), YARP: Yet another robot platform

Mingo, E., Traversaro, S., Rocchi, A., Ferrati, M., Settimi, A., Romano, F., et al. (2014), Yarp Based Plugins for Gazebo Simulator, *2014 Modelling and Simulation for Autonomous Systems Workshop (MESAS)*

Mohamed, N., Al-Jaroodi, J., and Jawhar, I. (2008), Middleware for robotics: A survey, in 2008 IEEE International Conference on Robotics, Automation and Mechatronics, RAM 2008, 736–742, doi:10.1109/RAMECH.2008.4681485

Paikan, A., Fitzpatrick, P., Metta, G., and Natale, L. (2014), Data Flow Port Monitoring and Arbitration, *Pasa.Liralab.It*, 5, May, 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*, 3, Figure 1, 5, doi:http://www.willowgarage.com/papers/ros-open-source-robot-operating-system

Sucan, I. A. and Chitta, S. (web page last accessed on 05-05-2015), MoveIt! - http://moveit.ros.org

FIGURES

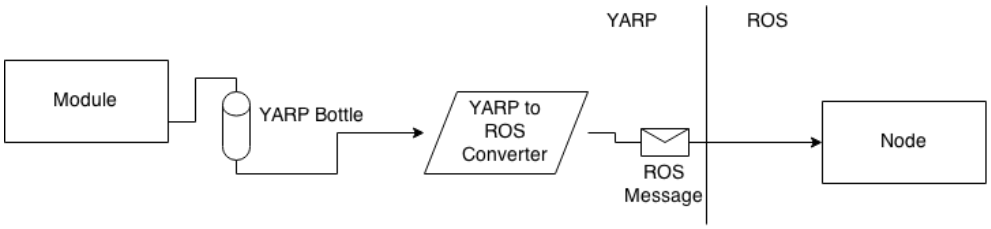

Figure 1. The architecture of the YARP with ROS work by **Fitzpatrick** (web page last accessed on 17-03-2015)

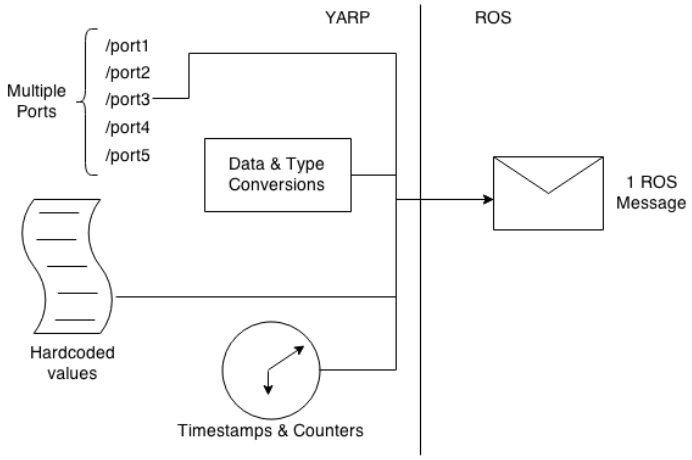

Figure 2. The great motivation of this thesis - 1 ROS message expects many conversions, specific types, data from several YARP sources and ordered fields

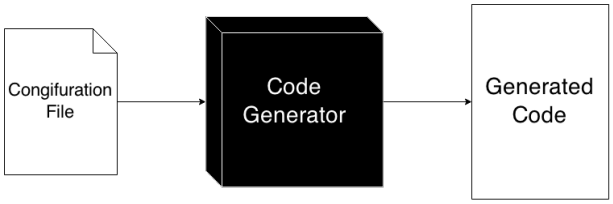

Figure 3. Yarp Bottle Generator architecture

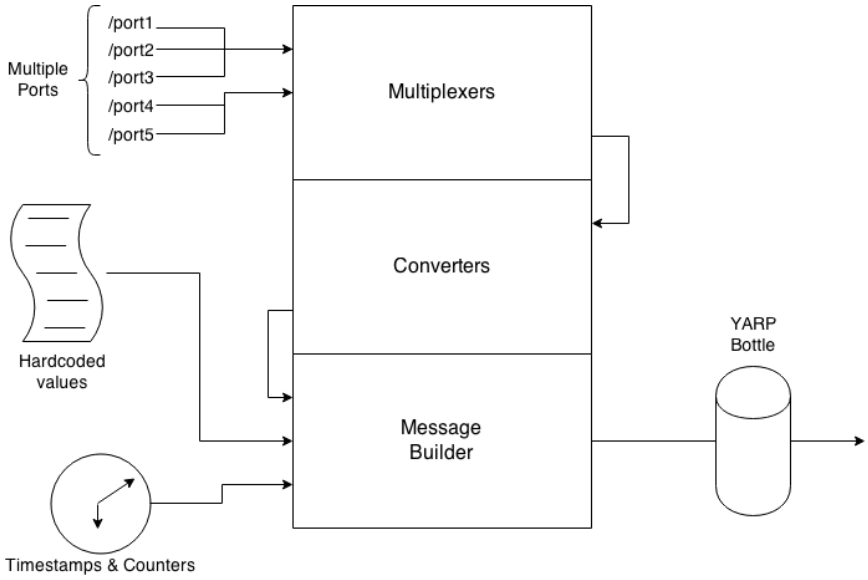

Figure 4. Generated code detailed architecture

```
[general]
output_name = /yarp/joint_states
to_ros = true
num_mux = 1

[mux1]
num_ports = 1
ports = /vizzySim/left_shoulder_arm/state:o
function = deg_to_rad
verbose = true

[message]
num_fields = 5
1_type = msg
1_msg = header
2_type = list
2_msg = "l_shoulder_scapula_joint", "l_shoulder_flection_joint", "l_shoulder_abduction_joint", "l_shoulder_rotation_joint",
"l_elbow_flection_joint", "l_forearm_pronation_joint", "l_wrist_abduction_joint", "l_wrist_flection_joint"
3_type = mux
3_mux = mux1
4_type = list
4_msg =
5_type = list
5_msg =

[header]
num_fields = 3
1_type = counter
2_type = timestamp
3_type = single_value
3_msg = "0"
```

Figure 5. Complete configuration file for the Rviz demonstration

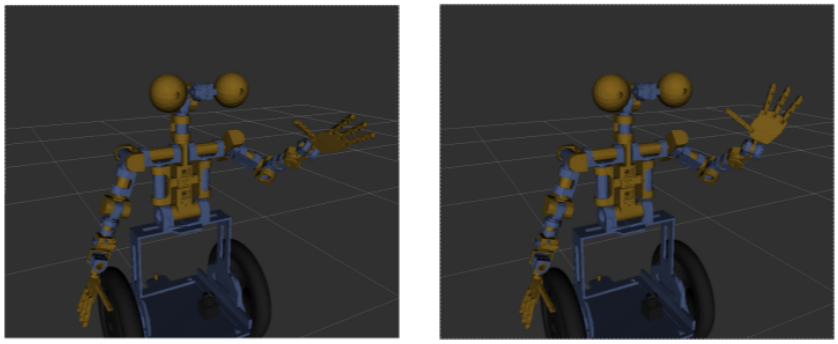

Figure 6. It is possible to take advantage of the ROS tool - Rviz - and understand how the real robot is performing

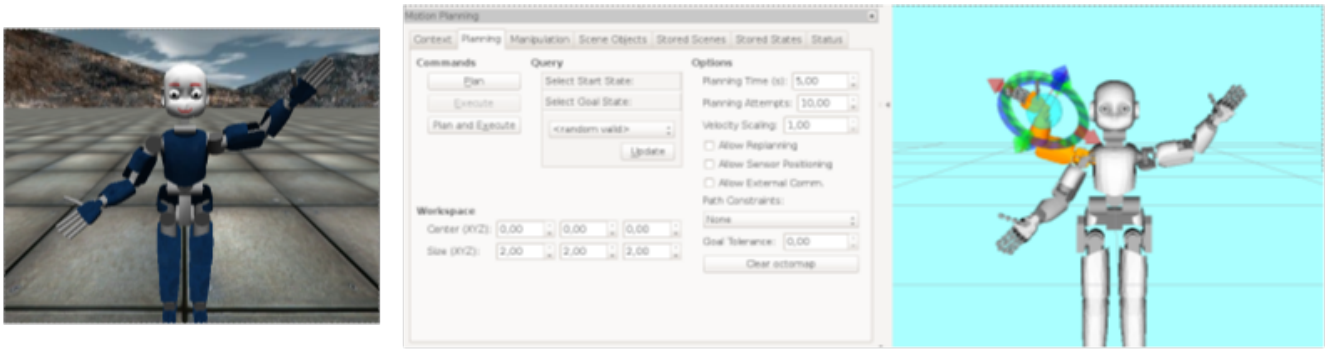

Figure 7. Both arms state is being showed on the MoveIt! Graphical Interface

```
36
37      for(int i = 0; i < reading1Mux1->size(); i++) {
38        mux1.add(reading1Mux1->get(i));
39      }
40      for(int i = 0; i < reading2Mux1->size(); i++) {
41        mux1.add(reading2Mux1->get(i));
42      }
43
44      for(int i = 0; i < reading1Mux2->size(); i++) {
45        mux2.add(reading1Mux2->get(i));
46      }
47
48      for(int i = 0; i < mux1.size(); i++) {
49        break;
50      }
51
52      for(int i = 0; i < mux2.size(); i++) {
53        std::cout << "value on index " << i << ": " << mux2.get(i).asDouble() << std::endl;
54      }
55
56      /* DO SOME COMPUTATION HERE */
57
58      int timestamp = (int) Time::now();
59
60      Bottle message = Bottle();
61
62      /* DO SOME COMPUTATION HERE */
63
64      outputPort.write(message);
65      counter++;
66      Time::delay(0.1);
67    }
68
69    return 0;
70  }
```

Figure 8. Second half of the generated code for the YARP to YARP demonstration