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

Miguel Moreira de Figueiredo Osório de Aragão

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors:
Doctor David Manuel Martins de Matos
Doctor Alexandre José Malheiro Bernardino

Examination Committee

Chairperson: Doctor José Manuel da Costa Alves Marques
Supervisor: Doctor Alexandre José Malheiro Bernardino
Member of the Committee: Doctor Rodrigo Martins de Matos Ventura

June 2015
Acknowledgements

This work was developed in the context of the laboratory VisLab from the Institute for Systems and Robotics, Instituto Superior Técnico, Lisbon, and it would have not been possible without all the knowledge and material it provided to me. I will always remember everyone from the laboratory because they helped me on my work almost everyday.

I would like to specially thank Doctor Alexandre Bernardino, Plínio Moreno and Ricardo Nunes for all the hours they have spent helping me achieving my goals and performing my experiences. Thank you for all the talks that boosted my motivation on the work.

I would also like to thank my supervisor Doctor David Matos for the many interesting talks about the work and other issues about the field of informatics and for the guidance that allowed this thesis to be more complete.

Finally I also want to thank my friends and family for the efforts on helping me through my entire life and studies.

Lisbon, June 29, 2015
Miguel Aragão
For my friends, parents, sister and girlfriend.
Resumo

Certas middlewares para robótica começam a ter uma posição de destaque, sendo consideradas como as melhores do mercado, mas de facto ainda não existe uma que forneça todos os serviços necessários para os diferentes sistemas de robótica.

Uma possível solução para este problema é a interoperabilidade que é a capacidade de várias entidades, middlewares neste caso, cooperarem, providenciando serviços e ferramentas umas às outras, permitindo assim que os programadores consigam utilizar as melhores funcionalidades de cada middleware com facilidade.

Esta tese desenvolveu uma framework que se foca na interoperabilidade entre duas middlewares importantes, o YARP e o ROS, facilitando o processo de comunicação entre elas através da automação da geração de código, e a configuração das mensagens trocadas, providenciando assim, ao programador, um nível de abstração superior na camada de comunicação. A framework está disponível num repositório público do GitHub onde é possível encontrar a documentação completa para a sua utilização. Por fim, foram também configuradas implementações da framework em plataformas reais.
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.
Palavras Chave

Robotic Middlewares
Interoperability Framework
YARP
ROS
Code Reuse
Cooperation
Code Development Automation
# Index

## 1 Introduction

1.1 Context .............................................. 1

1.2 Motivation and Goals .................................. 1

1.3 Thesis Structure ...................................... 2

## 2 YARP and ROS Overview

2.1 Middleware for Robotics .................................. 3

2.2 YARP .................................................... 6

2.2.1 Overview ........................................... 6

2.2.2 Ports ................................................ 6

2.2.3 Name Server ......................................... 7

2.2.4 Bottles .............................................. 7

2.3 ROS .................................................... 7

2.3.1 Overview ........................................... 8

2.3.2 Nodes ................................................ 8

2.3.3 Topics .............................................. 9

2.3.4 Services ............................................ 9

2.3.5 Messages .......................................... 9

2.4 Interoperability ......................................... 9

2.4.1 Generic Interoperability ............................ 9

2.4.2 YARP and ROS Interoperability .................. 10

2.5 Summary ............................................. 13
List of Figures

2.1 The architecture of the YARP with ROS work by Fitzpatrick (2015) ............ 12
2.2 The great motivation of this thesis - 1 ROS message expects many conversions, specific types, data from several YARP sources and ordered fields ............. 13

3.1 Yarp Bottle Generator Architecture .............................................. 17
3.2 Configuration File Sections ......................................................... 18
3.3 Code Generator Architecture ...................................................... 18
3.4 Generated Code Architecture ..................................................... 22
3.5 Complete Data Flow with the Generated Module ................................. 23
3.6 An example of the general section of a configuration file ...................... 24
3.7 An example of the mux sections of a configuration file ........................ 25
3.8 An example of the message specification sections of a configuration file ...... 26

4.1 Configuration file for the Rviz demonstration ................................... 32
4.2 Visualization of Vizzy’s upper body on the ROS tool Rviz from a remote location .................................................................................................................. 33
4.3 It is possible to take advantage of the ROS tool and understand how the real robot is performing .......................................................... 33
4.4 YARP robot iCub using the ROS tool MoveIt! ............................... 34
4.5 Both arms state is being showed on the MoveIt! Graphical Interface .......................... 35
4.6 First half of the generated code for the YARP to YARP demonstration .... 36
4.7 Second half of the generated code for the YARP to YARP demonstration ... 37
# List of Tables

2.1  An analogy for the ROS terminology ........................................... 8

3.1  Some examples of ROS messages fields and how the *message builder* can be configured for those fields ........................................... 21

4.1  Comparison between the number of lines in the *configuration file* and in the generated code ........................................... 37
1 Introduction

This chapter focuses on the context and motivations of the work. It also shows how the work will be presented in the next chapters.

1.1 Context

A middleware is a layer between the operating system and the software applications. The concept of middleware is not exclusively for robotics and it is applied in many different fields of the technology. It is possible to establish a parallel with the operating systems as both provide services and abstractions to the outer layers. On the last decade many different middlewares for robotics were developed but most of them have been forgotten or limited to applications on a specific robot.

A big problem in robotics is that there is still not a perfect middleware for all the robots in the world so it is difficult to progress at a higher rate because there is a lot of software modules and algorithms that can not be ported directly to other robots. We face a similar contest to the one between the operating systems. Now we have 3 clear winners (Windows, MacOS and Linux) but it was not like this before. In the robotics middleware case we still have not clear winners although some have better chances to succeed, like YARP and ROS.

Instead of creating the perfect middleware it is possible to try some other solutions before. Interoperability is a real possibility and it can be seen in almost every human interaction. It is the capability of entities cooperating with each other. It can be applied to the cooperation between robotic middlewares so that they can use services and tools from each other at a low cost.

The developed framework, named Yarp Bottle Generator, is presented and illustrated in robotic applications. The framework is fully documented and publicly available on GitHub at https://github.com/vislab-tecnico-lisboa/yarp-bottle-generator.

1.2 Motivation and Goals

YARP and ROS are the case study of this thesis and it happens that it is possible to communicate between YARP and ROS through an interoperability service provided by the YARP
team. It is possible to send a message from YARP that is converted into a ROS message, and vice versa, but the exchange from YARP to ROS is not as straight forward as it might look.

The communication from ROS to YARP has no major issues because the messages in YARP accept any kind of structure but ROS does not. A ROS message expects specific data types, a specific order of the data, data from many different sources (in YARP the data is split through different modules) and there are also compatibility issues on the units of the data. Till now there were no way to ease this process so it becomes slow and difficult to configure a message in YARP to be sent to ROS.

The goal of the work is to create a framework that eases all this process and allows a trivial configuration of the modules that want to send messages to ROS. This way most developers can take advantage of ROS and YARP services despite the environment they are coding in.

Although this is a specific approach for two specific middlewares, the work and its architecture can be seen as a generic solution to solve compatibility issues with other middlewares, and, in the future, an adaptation of this work could lead to other examples of interoperability.

1.3 Thesis Structure

Chapter 2 - YARP and ROS Overview - gives a deep context of the many concepts involved on the work and briefly presents the work developed by others towards a compatibility between YARP and ROS.

Chapter 3 - Yarp Bottle Generator - holds the definition of the framework and how everything works. It has several diagrams explaining the architecture of the final solution and the language specification.

Chapter 4 - Case Studies - first it presents a brief discussion of the main applications of the yarp bottle generator and its biggest advantages, and after, it covers some specific examples of usage and its results. It explains in detail how the framework is used on different robots and in different case scenarios.

Chapter 5 - Conclusions - the last part of this document holds a discussion about possible additions and improvements to be included in the future work, and, my final words about the achieved results.
2

YARP and ROS Overview

This chapter covers the most relevant concepts to fully understand the work developed on this thesis. It tries to contextualize the work and its goals with the current state of the technology. It also presents some related work both on robotics and other fields. Since the work focuses on two specific middlewares, YARP (Yet Another Robot Platform) and ROS (Robotic Operating System), the approach will be, first to give a more generic context, and after, a more fine-grained view of both these middlewares.

2.1 Middleware for Robotics

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 (2005).

Each field of the technology should have slightly different middlewares because there is not a single one with all the functionality needed (there is still not an universally accepted operating system so, in the near future, we do not expect a universal middleware at all).

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 order to analyse if a service should be part of the middleware layer one should think about its usability on several cases. If the large majority of the software modules implements some specific task, that means the service provided by that task, should probably be provided by the middleware itself.

In the robotics context, Ceseracciu, Domenichelli, Fitzpatrick, Metta, Natale, and Paikan (2013) describe the middleware as the entity which provides the glue that holds all the soft-
ware modules together. As Mohamed, Al-Jaroodi, and Jawhar (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.

Due to the specific architecture of robotic systems, being composed of several complex modules working with each other to achieve a goal, the communication services provided by the middleware can probably be considered the most important ones from the available set and this considerations are far from new as reported by Schlegel (2006) that discusses the communication issues at the time when there were no satisfactory middlewares for robotics. The main goal of the communication layer is to provide all the tools to exchange messages between software modules with ease. In the end, a valid middleware for communication should allow the developer to easily send and receive data from several sources so that its focus stays on the module and not on how to send and receive messages. Besides the abstraction of the communication layer, these middlewares provide another feature which is fundamental in a robotics system, real-time communication. So, it is not just an abstraction but also a specific feature on the layer that would call for even more effort from the developer in order to achieve an acceptable result by himself. Although most middlewares for robotics have the high-level goal of providing communication services, they do not follow the same approach for the low-level implementation. This way each middleware has its own way of dealing with messaging and the data that is exchanged in the network.

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, Suehiro, Kitagaki, Kotoku, and Yoon (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, Gerkey, Conley, and Garage (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.

Similarly to the operating systems, middlewares allow code reuse, but during many years there was not a robotics middleware that was the one chosen by the majority of the robotics teams. This lead to the appearance and fall of many platforms, a period similar to the war between operating systems (there is still not a unique operating system although most people use one of three, Windows, MacOS or Linux). Even though ROS is facing a good period on the robotics community there are still several middlewares around with great communities and
with a big set of software services like the YARP one. ROS now starts to give the first steps outside the research world and that might be an important step for its recognition as one of the best middlewares for robotics.

Having started this chapter from a high-level context to a more specific one, it is now time to start pointing my goals. Code reuse is the problem that got my attention and I will give you the needed context to fully understand my idea and proposal. The last few years lead us to a state of the development where we have a big set of modules that can not be ported directly between all the robots, and this is tremendously inefficient as said by Smart (2007) that considers to be almost impossible to have well-implemented algorithms without code sharing. The problem of achieving a state where there is a lot of code on many different middlewares is not new and trying to reuse it is something that has to take into account both time consumption and monetary costs because not taking advantage of what exists already will require extra efforts. I will cover this problem on the next sections and chapters because my proposed work aims directly at trying to take advantage of two specific middlewares, YARP and ROS. Both these choices were based on their potential and community around them. ROS is stepping into the industry and has one of the biggest communities on the robotics field. Besides this, it has a big set of services that have already been aproved by other applications and projects. YARP is probably the best middleware focused on the humanoids development and there are at least 20 laboratories using it with the iCub robot. It has an active development and it is an open-source project with many followers and contributors.

Both middlewares follow a similar approach on the architecture of the system. YARP and ROS networks can be seen as graphs where nodes are computational elements (nodes in ROS and modules in YARP) and links represent the connections between those elements (topics in ROS and ports in YARP). Being middlewares, they provide several services to abstract the developer from most of the common tasks. We can separate the services in two different kinds: native services and module services. Native services are the ones which provide the abstraction from inner layers. Usually these services are deeply attached to the middleware itself and they should be developed by the team responsible for the middleware. Module services are the ones that were developed over the middleware. These services provide an higher level of abstraction and are often developed by teams outside of the group responsible for the middleware itself. One of the main goals of middlewares is the possibility of code reuse, that allows developers to take advantage of existing modules without major changes, so it is common to associate module services with the middleware where they run, but, be aware that they are not native services although it should be possible to use them on any system that uses the same middleware. The focus of my work is on the native services, more specifically on the communication layer, because they are the ones that allow us to develop and use the module services with ease.
2.2 YARP

As stated by Metta, Fitzpatrick, and Natale (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.

2.2.1 Overview

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.

It is a pretty stable option in terms of robotics middleware these days and in the specific field of the humanoids it is definitely to be taking into account. It has a strong and active development team in the iCub Facility at the Istituto Italiano di Tecnologia in Genova, Italy and a lot of users and contributors on laboratories around the world. Its communication layer is powerful enough to be used even in non-robotic environments although it is not its main goal - this strong communication layer enforces YARP as a middleware oriented to the communication services as Middleware.org (2008) says a Message Oriented Middleware. In terms of the message structuring, there is still not an approach globally considered as the best one, so, for now, we can only state the advantages and disadvantages of the several approaches available. Currently YARP has support in all three major operating systems (Windows, Mac OS and Linux) and that is an almost unique feature in comparison to the other available middlewares.

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.

2.2.2 Ports

In a YARP network there is a fundamental concept that one should keep in mind to understand most of the architecture of the communication layer, the concept of port. A port is the basic structure for all the communications in a YARP network. There are several kinds of ports but they all share a common goal which is to provide the capability of modules being able to communicate with each other. Any module which needs to send or receive a message will only need an instance of a port keeping the abstraction about the low-level transactions happening under it. Ports can exchange messages under several protocols but the most common are TCP,
UDP, multicast and XML/RPC. RPC allows modules to act as services that have an input and output pointing to the same port in the network. In the point of view of the middleware architecture the communication is pretty simple, a port can send and receive messages and modules can instantiate the needed ports.

2.2.3 Name Server

There is another entity which is responsible for managing all the ports, a central server that handles all the needed routing. This server is called the name server and it has a fundamental role on the YARP network. It works with a naming convention, so ports are recognized by their names, and it provides information like location and type of connections between the network, allowing communication between the several entities.

2.2.4 Bottles

One can not send any kind of message over ports, so there is the need to understand the rules about the message structure. In YARP there are no message definition, by other words, messages do not need to follow a specific structure (like type or order) defined on some external file or object (not all middlewares follow this same approach). For this approach it was created an important element: the bottle. A bottle is the thing that can be sent over the YARP network and it refers to a specific object with the ability of holding any kind of data. Since there are no specification of the messages exchanged, the developers have to know the ports they are handling with and care about the specific types of the objects they are exchanging, and also, their order inside the message and its size. This approach makes the writing of documentation, along modules, of high importance so that others can reuse them with relative ease.

2.3 ROS

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, that 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, Conley, Gerkey, FAust, Foote, Leibs, Berger, Wheeler, and Mg (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.

### 2.3.1 Overview

ROS is definitely the right middleware to start most projects these days. It has the biggest community and offers almost everything that developers miss from the other middlewares. Its communication is one of the best organized and it follows a publish/subscribe model. 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. The set of available modules, besides the native services (like communication), is also one of the best around. In a question of a couple of hours we can configure a robot simulation and apply some reasonably complex algorithms for several kinds of applications and demonstrations.

ROS has a strong support in terms of mobile robots and as Cousins (2010) says it is one of the big reasons to choose over ROS, but it has many more powerful and useful features. Similarly to the YARP modules, ROS has the nodes, and the services they provide are built over the native services. The main focus will be the communication layer that is the base for all the operations over the computational elements because, as in the YARP case, it establishes the network that allows the exchanges between the several parts of the system components. There are four really important concepts to understand in a ROS network: nodes, topics, services and messages.

### 2.3.2 Nodes

Instead of the YARP modules in ROS we have nodes. Nodes are the parts of the software that execute computation. The ROS network is based on peer-to-peer connections so it is easy to imagine the entire system as a graph where each module, is represented as a node, and the connections as the edges. In ROS there is an object that has almost the same function as YARP ports, the topics. The table 2.1 is an analogy from the paper by Mayachita, Widyarini, Sono, Ibrahim, and Adiprawita (2013) which gives us an interesting parallel between the ROS concepts and other concepts of our day-to-day life.

<table>
<thead>
<tr>
<th>Terminology</th>
<th>Analogy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>User in a forum</td>
</tr>
<tr>
<td>Topic</td>
<td>Thread in a forum</td>
</tr>
<tr>
<td>Message</td>
<td>The interchanged messages</td>
</tr>
<tr>
<td>Publish message</td>
<td>Posting a message on a forum thread</td>
</tr>
<tr>
<td>Subscribe message</td>
<td>Reading a post on a forum thread</td>
</tr>
</tbody>
</table>

Table 2.1: An analogy for the ROS terminology


2.4. INTEROPERABILITY

2.3.3 Topics

A topic is an object, that is mapped on the name service and allows connections between nodes (see table 2.1). If one node wants to publish/subscribe to a specific topic, it only needs to know his symbolic name. The way messages are sent in a ROS network is, as said before, based on a publish/subscribe model. If a node wants to send a message (publish) it will write it on a certain topic; on the other side, if any node wants to read a message (subscribe) it will read from a certain topic.

2.3.4 Services

There is still one special communication which stands for synchronous transactions, where this model is not appropriated. The services exist to solve this. On a typical service there is an object composed by a string name and a pair of strictly typed messages (both request and response). One of the main differences from topics is that only one node can advertise a specific service.

2.3.5 Messages

ROS follows a different approach from YARP in terms of the messages structure. In YARP we have no definition of the messages, and the developer is able to send bottles with the data he wants but in ROS the developer has to create a specific message file. The same applies to services as their structure is also defined in an external file. A topic is bound to a certain type of message so there are no problems trying to understand what will be the content of a message because we only need to analyse the external file which contains the description. In order to instantiate a specific message on a node the developer only has to initialize an object that is created for each message file. This way the developer handles the messages as simple C++ types and keeps abstracted from the underlying processes.

2.4 Interoperability

Till now the concepts were generically presented in order to have the right background to understand the context of the proposed work, and, on this section, we can read about the concept which is aimed more deeply by the work.

2.4.1 Generic Interoperability

Interoperability is, generically speaking, the ability of making several entities work together. Chen, Doumeingts, and Vernadat (2008) define the concept of interoperability as the
CHAPTER 2. YARP AND ROS OVERVIEW

capability of different systems being able to communicate and take advantage of features of both. The cooperation from all the parts involved in a generic network is preferable because it allows an easier integration, but, even if some entity of the network is not ready to cooperate it is still possible to use its services with something like a translating mechanism. Interoperability is not only a technological concept but also something that people face everyday on multiple occasions. The concept is so generic that we can go from fields like human interactions to the technology keeping the same goal of cooperation between entities. Most interactions that are put in work for a specific task can be considered as part of an interoperability network because it shows us a group of entities working together taking advantage of the available features around them. 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.

Although we can think about so many different examples of interoperability they all share the same goal of working for a common task and that is the clear advantage of it, the cooperation. By cooperating it is possible to gather the best features from a set of available providers and achieve the best results with less effort and cost. Many working environments are based on cooperation and many people consider it as the only way to achieve the progress. The need for cooperation is not new and the existence of discussions as the one by Wegner (1996) proves that it is something critical for software development for a long time now, but still, it is not as straightforward to solve. Basically the concept behind interoperability is the ability of sharing efforts and knowledge to reach better results and that is the biggest concern of most companies and working entities.

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. In technological examples of interoperability there are specific problems and the one that this thesis aims is the compatibility issues that arise when two different middlewares try to communicate with each other.

2.4.2 YARP and ROS Interoperability

In robotics, one can define several levels of interoperability like the cooperation between several hardware pieces, several modules working for a specific goal or even cooperation between several middlewares. On this work the focus is on the interoperability between middlewares. This specific case of interoperability can be seen as a higher level of abstraction in terms of cooperation because it assumes that the several parts that compose a robotics system inside the same middleware have already the needed support to allow their task oriented interoperability.
In order to understand how interoperability between middlewares should work let's imagine that several car models are different middlewares, and that car components are like the services provided by the middlewares. There are a lot of different cars but they all share a similar architecture because their components have similar goals. These days companies try, instead of creating all these components from scratch for each car, to take advantage of existing ones even though they were initially created for other systems. Although cars can not share their components without losing them, it is possible to order an equal component and use it on other model. Interoperability between middlewares will be similar to this example because services from different middlewares should be used over different environments like using some ROS service through YARP. As stated by Fitzpatrick, Metta, and Natale (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.

The main challenges while trying to allow interoperability over robotics middleware are basically the same presented in the generic overview but applied to the specific context of robots and their community. One of those problems is that before ROS reached the state as the middleware to use for a robotic system, most teams tried to isolate their implementations in order to win the war as the best solution around. Another problem is that due to the complexity of the solutions, priorities while developing a middleware were not put on interoperability but rather on functionality.

More recently, with the growth of ROS both around the robotics industry and around the research environments people started to feel the need to develop solutions to take advantage of what has been done before even if it was built over different middlewares. The easiest way to start this kind of inter-operation would be by natively support it, so the teams responsible for the middlewares would have a fundamental role on this approach. That means both costs and time for those teams and maybe even more important the will to do that. Some middlewares are already capable of communicating with others, mainly with ROS and I believe this will be the tendency in the times to come. Besides this native support it is still possible to try new ways of ease the processes, almost like an extension of the available services. This way we can approach the progress and development counting on old services and modules even if they are based on different systems. This is basically the final goal of the work, to ease the process of interoperability between two specific middlewares, ROS and YARP.

Multi-agent systems have a huge role on many different fields, and in robotics they are fundamental because it is possible to have robots with different features cooperating while performing complex tasks. As stated by Iigo-Blasco, Diaz-Del-Rio, Romero-Ternero, Cagigas-Muiz, and Vicente-Diaz (2012) both YARP and ROS are middlewares that allow multi-agent control and management, so the interoperability between them will aim at an even bigger niche of robots and their functionalities. Being YARP a middleware mainly aiming at humanoid robots and ROS a more generic one, the focus will be on how to use ROS nodes with robots...
running over YARP. Although the exchange of messages on both ways (YARP to ROS and ROS to YARP) is really important, this work focus on the exchanges from YARP to ROS because they are the ones which raise more complex issues for the reason that ROS expects specific structures and that is not a behavior present on the YARP network. Basically, YARP is fine receiving any ROS message because its ports do not expect a specific message, but, on the other side, ROS expects specific messages, with specific data types, with specific order and specific data sources, and there is no way to configure this kind of messages with ease without the framework developed on this thesis. YARP robots can benefit from several features of modules developed over ROS like motion planning and execution, trajectory planning, collisions avoidance and several other examples. The beauty of the interoperability lies in the ability to easily take advantage, in one middleware, of services that become available on the other.

Figure 2.1: The architecture of the YARP with ROS work by Fitzpatrick (2015)

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 2.1). This work is described on the web page containing the work of Fitzpatrick (2015).

The Gazebo YARP Plugins presented by Mingo, Traversaro, Rocchi, Ferrati, Settimi, Romano, Natale, Bicchi, Nori, and Tsagarakis (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, Moreno, and Figueiredo (2015).
2.5 Summary

This chapter has presented the most important concepts of the background of the work and it should be target of a careful reading for a complete understanding of the next chapters.

The middlewares chosen for this work were YARP and ROS due to their strong community and availability on my laboratory. Besides that both this middlewares are on the right path to become stable choices to develop new robotic projects.

The work focus on the interoperability between middlewares, mainly on the reuse of ex-
isting services on different environments - code reuse. This example of cooperation allows middlewares to take advantage of services from other middlewares easing the development and configuration of robotic systems.

The motivation comes with the difficulties faced while sending messages from YARP to ROS due to the huge differences on the messages expected by this middlewares. The ROS network is waiting for a structured message with specific types and a specific order, and with many data sources split by several different modules of the YARP network. Without the framework developed there is no easy way to configure this messages on the YARP network.
Yarp Bottle Generator

This chapter will present the proposed work by covering all the concepts around it and the major architectural decisions and implementations. The final result was a framework named as *yarp bottle generator* (see figure 3.1). It is a generic approach to solve a specific problem and it follows a typical software architecture aiming code modularity and the possibility of extensions and other kinds of future work.

The problem is clear. The module, from the YARP with ROS interoperability, that is responsible for the conversion of YARP bottles into ROS messages (see figure 2.1), 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 (see figure 2.2), YARP does not, so there is the need to gather data from several different sources and that involves coding and synchronization, and also, there are still ordering and type compatibility issues.

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 it is possible to configure several different data sources (see figure 3.5). 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.

I believe this tool is the most generic solution available at the moment for YARP with ROS so I expect usage from YARP and ROS developers. Besides already existing users being able to ease their coding processes, I think this tool can bring other users to both middlewares because it improves the availability of several services from different middlewares turning those environments into more complete solutions.

### 3.1 Proposal

The main goal of the *yarp bottle generator* is to ease the process of configuring the communication between YARP modules and ROS nodes. In order to do this the *generator* automates all the tasks needed to create the bottle to be converted. The YARP to ROS converter (see figure
CHAPTER 3. YARP BOTTLE GENERATOR

2.1) waits for a specific bottle but, before this work there were many complications when trying to “feed” this converter (see figure 2.2). So the main goal of the work is to ease the creation of the 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 (see figure 3.2), eases most of the efforts that are needed to assure all the conditions on the bottle, so the framework fulfills its purpose.

The main problem was already identified and it is due to the big differences between the composition of messages in YARP and ROS (see figure 2.2). In order to create the right bottle there is the need to create a module that does all the configuration and wraps the data from the several sources. The automation will be made by generating this module without the need of coding it (see figure 3.1), so the developer will only have to configure the available options on a configuration file. This module will have as the input a configuration file and as the output the generated code, which will be able to do all the operations in order to create the right bottle and send it to a ROS topic (see figure 3.5). In terms of a generic overview of the architecture we have the configuration file which serves as the input of the code generator (see figure 3.3) that will produce the generated code (see figure 3.4).

This generic overview of the architecture allows us to imagine how the system works for the code generation process but there is still one important step in order to describe the complete process of creating the bottles (see figure 3.5). First the developer generates the module with the generator but then how is the bottle actually created and filled with the data from the specified sources? The approach is the creation of the bottles in real-time by running the generated module, acting as a bridge between the YARP to ROS converter and the other YARP modules. The generated code should be compiled with ease with the default makefile present on the destination folder of the generator. This way, unchanged generated code, so purely generated code, is compiled with no extra efforts and works as expected. It is still possible to access the generated code and add some specific functionality to extend the ones already present there (see figure 4.7), but the developer will have to be careful enough to maintain the code with no errors in order to compile and use it after.

3.2 Architecture

There are 6 concepts that can sum up the entire architecture of the yarp bottle generator (see figure 3.1): the configuration file (see figure 3.2), the code generator (see figure 3.3), the generated code, the multiplexer, the converter and the message builder (see figure 3.4 for the last 4 concepts). Not all these concepts are software modules that compose the system but they all represent an important part of it and it is fundamental to know them in order to understand the entire architecture and how the system works.
3.2. ARCHITECTURE

3.2.1 Configuration File

The configuration file 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 (see figure 3.2). 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. It is the only mandatory part to interact with because the generated code should work fine with no changes, so the *configuration file* is the only thing that has an effect on the *generated code*. The file is divided by sections, and each section has several fields. These fields are the ones which will be configured allowing the generation of the desired modules (see figure 3.6, figure 3.7 and figure 3.8).

3.2.2 Code Generator

It is the core of the system and it is responsible for parsing the *configuration file* and generate the desired code (see figure 3.1). 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. The code generator can be separated in 3 parts (see figure 3.3). For reading the configuration file it uses the ini files parser from the Boost C++ libraries. The module generates the code by printing specific strings containing C++ code to the output file. The makefile to compile the generated code is static and it does not depend on the configuration file so if the code generator is unchanged, it should output a file compatible with the existing one.

3.2.3 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 3.4): 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 but any extra change will be done at its own risk, so the developer has to be careful with it. After compilation the code should run during execution and be kept
3.2. ARCHITECTURE

alive for as long as the communication is needed (see figure 3.5).

3.2.3.1 Multiplexer

It is the first part of the generated code and its function is to get data from a variable number of ports and wrap it in a single structure (see figure 3.4). The YARP community presented the work of Paikan, Fitzpatrick, Metta, and Natale (2014) that proposes a similar object but instead of wrapping the data from several ports it is responsible by selecting one port as the data source from several input ports - the Port Arbitrator. 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 3.6). The fields needed to configure each multiplexer are the number of input ports and its names (see figure 3.7). 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.

3.2.3.2 Converter

The converter is an entity that allows the developer to apply a function, to the output of each multiplexer (see figure 3.4), 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 so in the configuration file there is no separated section for multiplexers and converters (see figure 3.7). 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: the function and the verbose. The function is configured by specifying the name of the function that the developer wants to apply to the data on the multiplexer output structure. It is possible to extend the set of available functions by coding its generation on the converter generator at the code generator. The verbose is just a field that expects a boolean in order to print more info about the data or not. 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 3.4). It is possible to apply no conversion to the structure by choosing the function none.

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.

3.2.3.3 Message Builder

This is the most complex part, and its configuration 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 (see figure 3.8) is where the developer defines how to build the bottle and it has only one mandatory field which is the number of fields 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. For each field the parser will expect to read one field specifying the type of the message variable from a set of possible types: single_value, list, mux, timestamp, counter or msg (see table 3.1 to check some examples for specific ROS message fields). After getting the field type, the parser will look for the fields that the specific type expects. The single_value expects the msg field and the developer should assign a hard coded value there. The list expects the msg field and the developer should assign a hard coded list where each element is separated by a comma. The mux expects the mux field and it should be configured with the number that identifies the mux which outputs the structure that the developer wants to fill the message field with. The timestamp does not expect any extra field and it fills the message field with a timestamp (the idea for future work is to be able to specify the approach to calculate the best timestamp). The counter also does not expect any extra field and it only fills the message field with the index of the current bottle (to help coordinating the age of the bottles). Finally the msg type means that the type of the message variable is another message, and it expects an extra field msg where the developer defines the name of the section where he will define that message.

3.3 Implementation

Now that all the concepts were explained it is possible to understand the behavior of all of the entities and how the cooperation between them all happens. So in this section it will be explained where the concepts are applied and where they are located in terms of the system architecture. This way it is possible to understand how the modules work and how the data flows, both during the generation process (see figure 3.1) and during the real-time execution (where the bottle is created and sent to the network - see figure 3.5).
3.3 IMPLEMENTATION

<table>
<thead>
<tr>
<th>ROS Message</th>
<th>Message Builder</th>
</tr>
</thead>
</table>
| Header header | x_type = msg
x_msg = header_section
(…)
[header_section]
(…)
| string robot_name | x_type = single_value
x_msg = “baltazar”
| string[] links_names | x_type = list
x_msg = “left_shoulder”, “left_elbow”, “left_wrist”
| float64 max_value | x_type = single_value
x_msg = 24.5
| float64[] velocities | x_type = list
x_msg = 20.0, 10.3, 67.5
| float64[] velocities | x_type = mux
x_mux = mux1
| time current_time | x_type = timestamp
| int32 index | x_type = counter

Table 3.1: Some examples of ROS messages fields and how the message builder can be configured for those fields

3.3.1 Parser

It is located on the main program of the code generator (see figure 3.3) and it is responsible for reading the data from the configuration file.

It stores all the configurations in local variables in order to instantiate the several generators with the needed attributes. The parsing of the configuration file is done by sections (see figure 3.2) in order to maintain code modularity allowing the addition of new options and fields with ease.

3.3.2 Library

The library is the set of available generators and it is an entity that is instantiated also on the main program of the code generator (the library provides the code generators that can be seen on figure 3.3). It is composed of 5 generators: the common initial generator, the multiplexer generator, the converter generator, the message builder generator and the common final generator. Both common generators are responsible for generating the code that is common for all the modules that are generated including some initializations and common variable declarations (they exist in order to have a modular system where the generators only generate what they are supposed to).

The library provides all the tools to generate the desired code. All the objects are code generators and they all have a function which returns a piece of code. The common generators build the
CHAPTER 3. YARP BOTTLE GENERATOR

3.3.3 Main Program

The parser and the library work together in order to generate the final module. The parser gets the configurations from the configuration file and instantiates the generators from the library. The main program is responsible for defining and coordinate the work of these two entities so that in the end it gets all the generated code (see figure 3.3).

The main program is then responsible for writing the generated code into a file with a predefined name and location. The current options on the configuration file does not allow the user to specify the name and location of the generated code because it would need some additional logic in order to support the generation of the makefile to compile the module. This way the file that is written is directly compiled by the available makefile (see figure 3.1).

3.3.4 Generated Module

The generated module is the final piece of the yarp bottle generator (see figure 3.4) and it should be compiled with ease in case the developer does not change the source code. The generated code will be stored by default on a specific location and it will have all the needed files to be compiled with ease so it will be trivial to have the executable program.
The executable should run on a YARP network and it will expect a ROS topic to be available in order to start creating bottles to be sent. Besides the ROS topic, the multiplexers will wait for YARP ports so it is fundamental that they have been instantiated before. The module works as a bridge so data comes from YARP ports and then the bottle that is built is sent to the YARP to ROS converter (see figure 3.4). This YARP to ROS converter is invisible to the regular user, so it is just like if we are sending a message to another YARP port which is in fact a ROS topic. This module will be constantly publishing the bottle to the ROS topic because the execution is kept on an infinite loop, where the data is updated at each iteration and wrapped in a new bottle. As soon as the user stops the module the ROS topic will stop receiving the data from the YARP network.

![Diagram](image)

**Figure 3.5: Complete Data Flow with the Generated Module**

### 3.4 Language Specification

The language specification is the definition of how the fields of the configuration file should be configured. It is the only part of the system that needs to be configured and it follows a typical syntax of configuration files with some special cases in order to adapt the options to the system needs. There are 3 major parts (some may have more than one section) on the configuration file (see figure 3.2): the general part, the mux part and the message part.

#### 3.4.1 General Part

In order to support the process of understanding the syntax of the general part check the figure 3.6.
CHAPTER 3. YARP BOTTLE GENERATOR

This part has one section called general and it has 3 variables: output_name, to_ros and num_mux.

3.4.1.1 output_name

This variable expects the name of the output topic (see line 2 of figure 3.6). This means that in the end of each cycle of the generated module the created bottle will be sent to the ROS topic with the name specified on this variable.

3.4.1.2 to_ros

This variable expects true or false (see line 3 of figure 3.6), true when the module is supposed to send a message to a ROS topic and false when the output is a YARP port.

3.4.1.3 num_mux

This variable holds the number of multiplexers that the developer will configure on the next sections (see line 4 of figure 3.6). It is possible to create the generated module with no multiplexers by assigning the value 0 as the number of multiplexers.

```plaintext
1  [general]
2  output_name = /output_topic
3  to_ros = true
4  num_mux = 2
```

Figure 3.6: An example of the general section of a configuration file

3.4.2 Mux Part

In order to support the process of understanding the syntax of the mux part check the figure 3.7.

The parser will expect to find a number of mux sections equal to the number defined on the num_mux variable from the general section. The name of the sections will be mux plus its index as a prefix (for example, mux1, mux2 and so on). Each section has 4 variables: num_ports, ports, function and verbose. Since for each multiplexer there is an associated converter the variables for the converter are configured on the correspondent multiplexer section. The num_ports and ports are the variables for the multiplexer configuration and the function and verbose are the ones for the converter.
3.4.2.1  num_ports

This value expects the number of input ports of the multiplexer (see line 7 of figure 3.7). It is needed to allow the correct parsing of the names of the ports. Although it accepts the value 0 there is no use on it because a multiplexer with no input ports will be pointless.

3.4.2.2  ports

This variable will contain the names of all the input ports of the multiplexer (see line 8 of figure 3.7). Each name should be separated by a comma and in the end, the number of commas has to be equal to the num_ports value minus 1.

3.4.2.3  function

The function variable is the description or name of the operation that is going to be performed against the multiplexer data (see line 9 of figure 3.7). There is a finite number of available functions but the developer might add new ones by adding its code to the library.

3.4.2.4  verbose

This variable expects true or false (see line 10 of figure 3.7) and it just tells the code generator if it should generate code for printing debug information about the data that goes through the converter.

```
6  [mux1]
7    num_ports = 2
8    ports = /port_number_1, /port_number_2
9    function = deg_to_rad
10   verbose = true
11
12  [mux2]
13    num_ports = 3
14    ports = /port_number_3, /port_number_4, /port_number_5
15    function = none
16   verbose = false
```

Figure 3.7: An example of the mux sections of a configuration file

3.4.3  Message Part

In order to support the process of understanding the syntax of the message builder part check the figure 3.8.

There might exist more than one section for the message builder depending on the message structure but there is a mandatory section that has the name of message. There will be more
than one section when one of the fields of the message is in fact another message (so messages have fields, and fields can have a primitive type or another ROS message). All the sections will have the same variables: num fields and one or more variables for each of the fields (at least one for each field - the table 3.1 shows some examples of this configuration to help you understand when to use each type).

![Figure 3.8: An example of the message specification sections of a configuration file](image)

3.4.3.1 num fields

This field expects the number of fields of the ROS message (see line 19 of figure 3.8). This number should match the exact number of fields that the message file on the ROS package has because that is the only way the bottle can be converted to the desired ROS message.

3.4.3.2 Other Fields

The num fields value will influence the number of variables that the developer has to configure for the message builder. There will be at least one variable for each field (see lines 20, 22, 24, 26, 28 and 30 of figure 3.8) to define its type having as name the field index plus field
(for example, 1\_field, 2\_field, 3\_field and so on). There are 6 types: single\_value, list, mux, msg, timestamp and counter.

The timestamp and counter types does not expect extra variables but others do (see lines 35 and 36 of figure 3.8). Both single\_value and list types expect the extra variable msg (for example, 1\_msg - see lines 23, 27 and 31 of figure 3.8) where the developer can add hard coded data to the message (a single value or a list of values separated by commas). The mux expects the mux variable (for example, 1\_mux - see lines 25 and 29 of figure 3.8) where the developer should indicate the index of the mux whose output he wants to fill the message field with. The msg also expects the msg variable (for example, 1\_msg - see lines 21 and 40 of figure 3.8) where the developer defines the name of the section that will configure the other ROS message (so this type applies when the message field is not of a primitive type but of another ROS message type).

### 3.5 Summary

This chapter gave a detailed explanation of the concepts that compose the yarp bottle generator and in the end it presented the syntax of the configuration file. It serves as the official documentation of the framework but it can be complemented by visiting the public repository on GitHub at https://github.com/vislab-tecnico-lisboa/yarp-bottle-generator.

The framework has several entities that were covered in this chapter but most users will only have to deal with the configuration file. The rest of the entities that compose the complete architecture of the yarp bottle generator will probably be changed only by developers who want to add extra functionalities to the generated modules.
This chapter will first aim at covering all the advantages that come from the *yarp bottle generator* and its benefits for future work and for the current development. After, the chapter will cover specific use cases that demonstrate the advantages of using the *yarp bottle generator*. These are the demonstrations that were performed in order to evaluate the process of configuring the generators and how they worked at execution time. There will be two demonstrations, one regarding the YARP and ROS interoperability and the other one regarding the usage of the *yarp bottle generator* exclusively on the YARP network.

### 4.1 Discussion

Although this work is focused on two specific middlewares it is possible to adapt the approaches for other integrations so I consider even more important to be able to identify the strongest features in order to be able to adopt the solutions for the most suitable case scenarios. In order to support this summary of the most important advantages it will be preceded by a section containing the several applications where the *generator* can be applied.

#### 4.1.1 Applications

In terms of applications we can focus on the generic possibilities or on specific ones. This section tries to give a generic overview keeping the specific examples for the demonstration chapter. The idea is that a generic overview abstracts the reader from specific concepts keeping its focus on the real applications and advantages. The generic applications are not completely generic because they will focus on specific middlewares but keep in mind that a similar module could work on other environments.

An obvious application is of course the motivation of the work which is the interoperability between YARP and ROS. The *yarp bottle generator* allows the developers to easily configure connections between YARP modules and ROS nodes. It automates all the processes needed and gives an extra value to both networks. It is now trivial to send a message from YARP to ROS so it is one of the main applications of the module.

Although the module was developed in order to ease the process of configuring the communication between YARP and ROS it can be extremely useful for an usage exclusively inside
the YARP network. It can be extremely powerful because it gives an extra level of abstraction to the developers because they can generate all the code structure and focus all their efforts on the goals of their project. All the communication will be configured by the generator and the developer will only complete the section of the code where the execution should occur. There is another useful feature which is the possibility of understanding the contents of the bottles exchanged between the modules without having to read the source code. In YARP there are no message specification (like in ROS messages) and by reading the configuration file it is possible to understand how the bottle will be structured.

Another application is a not so obvious one because it expects the imagination and creation of others. It is the possibility of extending the current work on ways that I do not even imagine now. There are some future work that I will present on the last sections but I believe there is much more than that. The module was developed thinking about extensibility so it should be just a matter of ideas in order to improve the current features and code new ones. The application is basically the possibility to work as the ground of other works, so starting with the yarp bottle generator try to come with something better and even more useful.

Another application is the possibility of integrating the concepts with YARP so that we can use this service natively by using YARP. This way there would not be a bridge but a more complete service from the middleware. It would be an improvement for both entities because it would increase the value of the middleware by having an higher-level of abstraction and, for the yarp bottle generator, it would be a recognition of its original value. By integrating the module directly with YARP there would probably be more efficient processes so it would also provide a better solution than the current one.

The other application is the application of a similar approach to the automation of the communication on the other way, so from ROS to YARP. It can be seen as a motivation to complete the automation of the code to perform that kind of communications. Although I do not see as many problems when sending messages from ROS to YARP because the second one does not expect any kind of specific message (so it is possible to send any ROS message and the converter will always work). Even though there are not as many problems, the higher-level of abstraction plus some more configurations would be enough to motivate efforts for that specific work. When this happens we will have automation on both directions and both middlewares will be able to cooperate with ease.

4.1.2 Advantages and Benefits

Now that we have an overview of the applications it is possible to select both the advantages and benefits of using the yarp bottle generator and the advantages of the approach of trying to automate the development by building tools with that goal.

One of the most evident and maybe the one which motivates the interoperability between
4.2. YARP TO ROS

different middlewares the most, is code reuse. Code reuse is a fundamental feature in order to develop complex systems because it saves us time and resources. With this module we can use ROS services and feed them with data from the YARP network. Instead of trying to match the set of services from ROS on YARP, it is possible to use them with low effort. Also, the capability of abstracting the developers from the communication issues, keeps the interaction as a transparent one, almost like a communication between modules inside the same network. It means code reuse at its maximum.

The benefits for both middlewares are of course huge because both have an extra set of services that increase their value. More value means more users, bigger community and in the end it means more developers around and a consequently progress on the field.

Of course the big advantage of the generator itself is the capability of easing the process of configuring the communications. This allows several kinds of developers to take advantage of the interoperability between ROS and YARP, because there is no need to have deep knowledge on the YARP network to be able to use a ROS service with a YARP robot. Users get to try different middlewares and it works as publicity for them.

Another big advantage of this yarp bottle generator is the fact that it tries to be as generic as possible. There is not a specific message that can be created but instead it can generate any kind of message. It goes against the approach from the recent work of the YARP team, that allows developers to configure a small set of ROS ready messages on the YARP network. Maybe a fusion between both works would result in a better solution, getting the generic approach from the yarp bottle generator and the integration with YARP from the other approach because it works as a native service.

The benefit of being highly modular is that it is possible to add new features with ease. Most experienced developers can even add functionality to the existent modules and improve the current solutions. The modularity also allows the possibility of integration with other environments, like other middlewares for example (of course these other middlewares should follow a similar culture on the communication layer).

4.2 YARP to ROS

This demonstration is a specific one and it focus on a particular ROS message: the joint_states message (see figure 4.1). 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.
4.2.1 Description

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, Cohen, and Likhachev (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 (see figure 4.2). 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 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 plan-
4.2. YARP TO ROS

Figure 4.2: Visualization of Vizzy’s upper body on the ROS tool Rviz from a remote location

ning with collision avoidance. It is an useful feature that is not yet supported through the ex-
istent 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 hap-
pens that the ROS middleware has an already working version of a package that performs ex-
actly what we want, motion planning with collision avoidance developed by Sucan and Chitta
(2012) and called MoveIt! (see figure 4.4). 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).

Figure 4.3: It is possible to take advantage of the ROS tool and understand how the real robot
is performing

4.2.2 Goals

The goals of both these demonstrations is clear - to be able to use ROS services fed with
data from the YARP network (see figure 4.1). 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.

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! (see figure 4.5) with the data from the actual robot running over YARP.

Figure 4.4: YARP robot iCub using the ROS tool MoveIt!

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 (see figure 4.3) it is possible to do that with ease. 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.

4.3 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.

4.3.1 Description

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.
4.4 Results

Although there are no obvious quantitative (the table 4.1 shows a comparison between the number of lines in the configuration file and in the generated code) results 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.

4.4.1 Practical Results

Regarding the demonstrations aiming the tests on interoperability between YARP and ROS the yarp bottle generator shows as the tool to use because it is generic enough to allow an easy and powerful configuration of the connections between ROS nodes and YARP modules (see figure...
4.1. In the demonstration two systems were configured and both modules were generated with success allowing the usage of two ROS services, Rviz tool and the MoveIt! planner.

In the case of the YARP to YARP demonstration, the code is generated with success and it is possible to focus exclusively on the module goal (see figure 4.7) 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.

4.4.2 Overall Evaluation

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 has not 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 (see figure 4.1) and everything works as expected even if the developer has little background in the YARP to ROS interoperability.

```cpp
#include <iostream>
#include <yarp/os/all.h>
using namespace yarp::os;
int main(int argc, char *argv[]) {
    Network yarp;
    BufferedPort打了receiverBuff1Mux1;
    bool receiverMux1Ok = receiverBuff1Mux1.open("/generatedCode/mux1/receiver1");
    BufferedPort打了receiverBuff2Mux1;
    bool receiverMux2Ok = receiverBuff2Mux2.open("/generatedCode/mux1/receiver2");
    BufferedPort打了receiverBuff1Mux2;
    bool receiverMux2Ok = receiverBuff1Mux2.open("/generatedCode/mux2/receiver1");
    Port outputPort;
    outputPort.writeOnly();
    bool outputOk = outputPort.open("/pose_estimation");
    yarp.connect("/icubSim/cam/right", receiverBuff1Mux1.getName());
    yarp.connect("/icubSim/cam/left", receiverBuff2Mux1.getName());
    yarp.connect("/icubSim/right_arm/state:0", receiverBuff1Mux2.getName());
    int counter = 0;
    while(true){
        Bottle* reading1Mux1 = receiverBuff1Mux1.read();
        Bottle* reading2Mux1 = receiverBuff2Mux1.read();
        Bottle* reading1Mux2 = receiverBuff1Mux2.read();
        Bottle* reading2Mux2 = receiverBuff1Mux2.read();
    }
    Bottle mux1;
    Bottle mux2;
}
```

Figure 4.6: First half of the generated code for the YARP to YARP demonstration
4.4. RESULTS

```cpp
for(int i = 0; i < reading1Mux1->size(); i++) {
    mux1.add(reading1Mux1->get(i));
}
for(int i = 0; i < reading2Mux1->size(); i++) {
    mux1.add(reading2Mux1->get(i));
}
for(int i = 0; i < reading1Mux2->size(); i++) {
    mux2.add(reading1Mux2->get(i));
}
for(int i = 0; i < mux1.size(); i++) {
    break;
}
for(int i = 0; i < mux2.size(); i++) {
    std::cout << "value on index " << i << ": " << mux2.get(i).asDouble() << std::endl;
}

/* DO SOME COMPUTATION HERE */
int timestamp = (int) Time::now();
Bottle message = Bottle();
/* DO SOME COMPUTATION HERE */
outputPort.write(message);
counter++;
Time::delay(0.1);
return 0;
```

Figure 4.7: Second half of the generated code for the YARP to YARP demonstration

<table>
<thead>
<tr>
<th>MoveIt! Planner</th>
<th>RViz Visualization</th>
<th>YARP to YARP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration File</td>
<td>Generated Code</td>
<td>Configuration File</td>
</tr>
<tr>
<td>30</td>
<td>91</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison between the number of lines in the configuration file and in the generated code

The biggest down points are specific configurations that are not yet supported and that will be discussed on the future work. Besides that there are some feedback on the aesthetic feeling that the configuration file gives to the developer and those issues will also be pointed out on the next chapter.

It is an easy to use tool, 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.
4.5 Summary

This examples of applications are important in a work like this because there is not much quantitative results to present and the practical results show if the framework works or not. Both examples show the potential of the yarp bottle generator and the discussion about the possible applications helps us understanding how it can be applied on the current existing systems.

The goal was to prove how generic is the configuration file by providing different case studies for different robots. In the end the results were positive since the services from ROS were successfully used with the data from the YARP network.
5 Conclusions

This is the final chapter of my thesis and it holds some considerations about the future work and expandability of the current features and also a conclusion resuming my final words and thoughts.

5.1 Final Words

In technological terms the final result 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 of the framework usefulness but the results achieved on the closest projects show us how generic the solution is.

The usage of the ROS services by YARP robots proves the possibility of code reuse over this two middlewares which was one of the main goals of the work. That code reuse, and the ease on the process of configuring that usage of services, leads the interoperability between YARP and ROS to a better state of the development. 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. Besides that, the YARP to YARP examples show us that the code automation is also improved by the non ROS oriented skills of the framework.

5.2 Future Work

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.

Of course there are other considerations that can be done, and with usage from other people there will be definitely more use cases and issues that will ask for improvements and fixes, but this is a start and it can help other people to follow the work and take advantage of this thesis.
Bibliography


