Universidade de Aveiro, Departamento de Electrónica, Telecomunicações e Informática, 2015

Aplicação de Controlo e Monitorização

Control and Monitoring Application

Autores / Authors
José Horácio Fradique Duarte, Nº Mec 62406
Fernando José Fradique Duarte, Nº Mec 48257

Aplicação de Controlo e Monitorização

Control and Monitoring Application

Relatório de Projeto em Informática da Licenciatura de Engenharia Informática da Universidade de Aveiro, realizado por José Horácio Fradique Duarte e Fernando José Fradique Duarte sob a orientação do professor José Nuno Panelas Nunes Lau, Professor Auxiliar do Departamento de Electrónica, Telecomunicações e Informática e do professor Artur José Carneiro Pereira, Professor Auxiliar do Departamento de Electrónica, Telecomunicações e Informática.

.Palavras-Chave / Keywords

Dynamic Anthropomorphic Robot with Intelligence–Open Platform
Humanóide / Humanoid
Robótica / Robotics
Robot Operating System
Sistema de Controlo e Monitorização / Control and Monitoring System

Agradecimentos / Acknowledgements

Gostaríamos de agradecer aos nossos supervisores de projecto, o Professor Nuno Lau e o Professor Artur Pereira a pronta disponibilidade que sempre demonstraram para nos ajudar a ultrapassar os problemas com que nos deparámos no desenrolar das várias fases do projeto.

We would like to thank our project supervisors, Professor Nuno Lau and Professor Artur Pereira for the availability they always showed to help us overcome the problems we faced during the several stages of the execution of the project.

Resumo / Abstract

A utilização de robôs com forma humanóide tem sido alvo de um interesse crescente por parte de inúmeros centros de investigação, a nível mundial, em diversas áreas de aplicação. Iniciativas internacionais tais como a RoboCup Liga Humanóide, palco de competição entre alguns dos modelos humanóides mais avançados do mundo, vieram dar ainda mais visibilidade a este ramo da robótica. Recentemente assistimos ao aparecimento das primeiras plataformas deste tipo ditas abertas.

A Universidade de Aveiro tem várias equipas de investigação ligadas a esta área e participa regularmente em eventos nacionais e internacionais com excelentes resultados.

Este projeto centra-se no desenvolvimento de uma aplicação de monitorização e controlo para o robô humanóide, em plataforma aberta, DARwin-OP, bem como a sua integração com o meta sistema operativo, de código aberto, Robot Operating System (ROS).

Neste documento apresentamos o estudo e levantamento efetuados sobre as capacidades desta plataforma. De seguida, descrevemos e analisamos a implementação da aplicação de monitorização e controlo, a sua arquitetura e o protocolo de comunicação que usa. Finalmente, discutimos a integração realizada com o sistema operativo ROS.

Depois da realização deste projeto, compreendemos melhor os problemas, as dificuldades e os desafios que se colocam na àrea da robótica e pensamos ter adquirido uma base de conhecimento que pode ser aplicada em trabalhos futuros.

The use of human-like robots or humanoids has received significant attention in recent years by several worldwide research centers in many fields of application. International initiatives such as the RoboCup Humanoid League competition, stage for some of the most advanced autonomous robots in the world, have contributed largely for the rapid growth in popularity of this field of robotics. Recently, the first open platforms of this kind have made their appearance.

The University of Aveiro has several research teams in this field and participates regularly in national and international events with excellent results.

This project is focused on the development of a control and monitoring application for the open humanoid platform robot DARwin-OP, as well as its integration with the Robot Operating System (ROS), an open-source meta-operating system.

In this report we present the study performed to evaluate the capabilities of this platform. Next, we describe and analyze the implementation of the control and monitoring application, its architecture and communication protocol. Finally, we discuss the integration process with ROS.

In general, this project allowed the creation of a solid foundation for future work and the understanding of the problems, difficulties and challenges found in the field of robotics.

Table of Contents

List of Figures

Abbreviations

API	Application Programmer Interface
CAD	Computer-aided design
CPU	Central Processing Unit
DARwin-OP	Dynamic Anthropomorphic Robot with Intelligence–Open Platform
DoF	Degrees of Freedom
DXL	Dynamixel
FIRA	Federation of International Robot-soccer Association
FK	Forward Kinematics
FSR	Force Sensing Resistors
ICRA	International Conference on Robotics and Automation
IDE	Integrated Development Environment
IEETA	Institute of Electronics and Informatics Engineering of Aveiro
IP	Internet Protocol
IRIS	Intelligent Robotics and Systems Laboratory
JSON	JavaScript Object Notation
LED	Light-Emitting Diode
LEI	Licenciatura em Engenharia Informática
OS	Operating System
Qwt	Qt Widgets for Technical Applications
RGB	Red-Green-Blue Color Model
ROS	Robot Operating System
SDK	Software Development Kit
SSD	Solid State Drive
TCP	Transmission Control Protocol
UI	User Interface
UPENN	University of Pennsylvania
URDF	Unified Robot Description Format
USB	Universal Serial Bus
XML	Extensible Markup Language

Chapter 1

1. Introduction

Humanoid robotics is an exciting and challenging research field governed by very ambitious goals. Since its inception considerable progress has been made. Proof of that is that new technical innovations and remarkable results achieved by universities, research institutions and companies are being made public on a daily basis [15].

Many international initiatives, in the form of competitions, such as the RoboCup Humanoid League are not only taking the humanoid research to the spotlight as well as serving as a catalyzer: by raising the complexity of the technical challenges that the competing teams must solve, on one hand, while at the same time instigating the idea that investigators should share their research amongst themselves and everybody else.

It is no wonder that reproducible humanoid robots systems endowed with rich and complex sensorimotor capabilities, able to mimic human movement and perform relatively difficult tasks, are now being made available [15]. One such example is the Dynamic Anthropomorphic Robot with Intelligence–Open Platform (DARwin-OP), subject of this report.

1.1 Context

This is an Informatics Project, integrated with the Bachelors of Informatics Engineering (LEI) lectured at the University of Aveiro, which focuses on humanoid robotic platforms, more specifically on the Dynamic Anthropomorphic Robot with Intelligence–Open Platform (DARwin-OP).

The project was developed at the Intelligent Robotics and Systems Laboratory (IRIS), at the University of Aveiro, during the course of the second semester of the 2014/2015 academic calendar.

1.2 Motivation

The University of Aveiro has had a very successful path in the field of robotics over the years, having participated and won several prizes and awards in national and international competitions. Recently two DARwin-OP platforms were acquired for research and education purposes.

The motivation for this project is to lay out the foundations for further work and research, involving this specific robotic platform, by developing a useful control and monitoring tool as well as to provide a first exploratory insight into the capabilities and potential of this new and exciting robotic platform.

1.3 Goals

The main objective of this project is to design and develop an intuitive and easy to use control and monitoring application, fully integrated with the real robot. The second goal aims to integrate the ROBOTIS framework with ROS via the implementation of a wrapper library. We also expect to report the advantages and disadvantages of such integration.

1.4 Document Structure

This report is organized as follows:

Chapter 2 introduces the DARwin-OP platform in terms of its hardware and software structure.

Chapter 3 outlines similar or related work already developed over this platform as well as the technologies used during the development of our solution.

Chapter 4 focuses on the system requirements and architecture of the control and monitoring application.

Chapter 5 details the implementation of the control and monitoring solution and presents the protocol of communication between its different modules.

Chapter 6 discusses the integration of DARwin-OP with ROS.

Chapter 7 presents the conclusions and future work.

Chapter 2

2. DARwin-OP

DARwin-OP is a modular miniature humanoid robot, depicted in Figure 1, and stands for Dynamic Anthropomorphic Robot with Intelligence–Open Platform. The specific version targeted by this project has approximately 45.5 centimeters of height and weights 2.9 kilograms. It has dynamic motion ability, several sophisticated sensors integrated and comes packaged with many code samples and predefined behaviors which make it ideal for research and educational purposes, which, as explained in [5], was one of the main goals driving its conception and design.

On top of that DARwin-OP was designed has an open platform, meaning that all the code (except for the DYNAMIXEL (DXL) firmware), the schematics of the robot, including circuit diagrams, mechanical Computer-aided Design (CAD) files and parts information are freely accessible to anyone. DARwin-OP is highly customizable and fairly popular due to its many participations in several international competitions, such as: the IEEE International Conference on Robotics and Automation (ICRA) Robot Challenge [16], RoboCup Humanoid League [2], Federation of International Robot-soccer Association (FIRA) [17] and Humabot [18]. A new version of this platform ROBOTIS OP2 was recently announced [19].

Figure 1 DARwin-OP

2.1 Platform Overview

In this section we present a general overview of the DARwin-OP platform in terms of its hardware and software architectures. At the end of the section we mention some useful links for more information and details concerning these subjects.

2.2 Hardware Architecture

As illustrated in the mechanical design scheme in Figure 2 and Figure 3, depicted below, DARwin-OP is based on a modular network structure composed of two controllers. The main controller is a SBC-fitPC2i equipped with a 1.6 GHz Intel Atom Z530 Central Processing Unit (CPU) running Ubuntu 9.10 Karmic, the sub controller is a CM-730 manufactured by ROBOTIS equipped with an ARM CortexM3 STM32F103RE 72MHz CPU.

All devices, the camera being the exception, are connected to the sub controller which acts as an access gateway. The communication between the two controllers is established via Universal Serial Bus (USB).

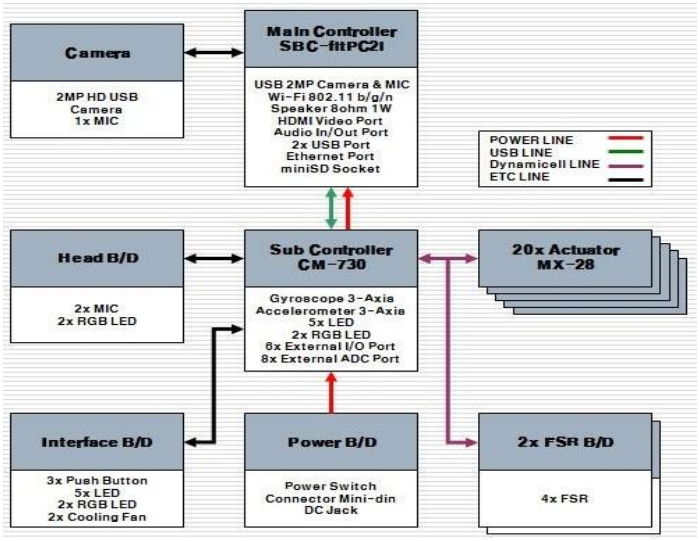


Figure 2 System block diagram (a) [4]

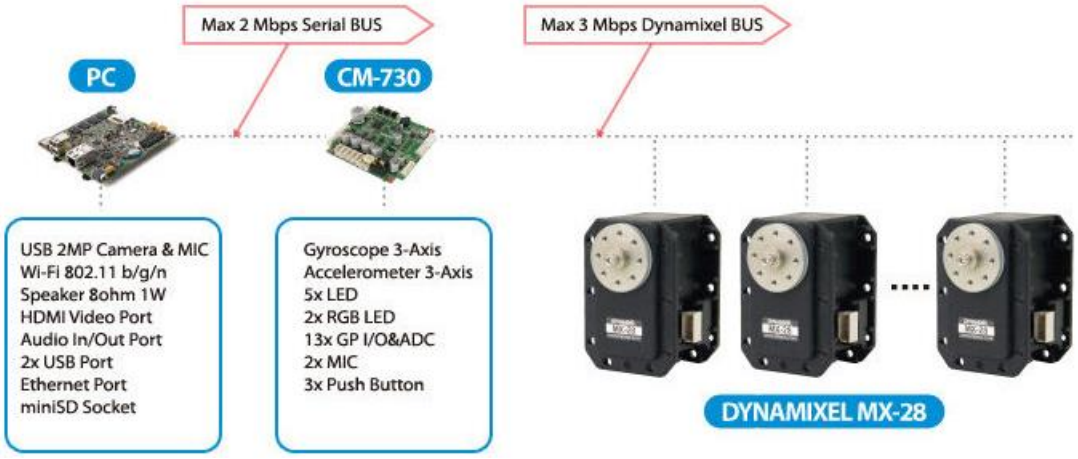


Figure 3 System block diagram (b) [2]

2.3 Sensors

DARwin-OP has a myriad of sensors and interfaces as shown in Figure 4 below. The following sensors can be found natively integrated in the platform:

- A three axis gyroscope and accelerometer mounted in the upper body, for posture estimation and balancing;
- A USB-based high-definition camera for image processing and a total of three microphones, one for voice recognition with the purpose to receive commands and two additional ones for sound localization, located in the head;
- A speaker;
- Full-color range RGB LED's in the eyes and forehead can be programmed to show specific colors. There are also status LED's and buttons in the back panel of the upper body;
- Four Force Sensing Resistors (FSR) modules can optionally be placed on each foot for ground reaction force measurements;
- Additional sensors can be attached via external I/O.

Figure 4 Integrated sensors and interfaces [2]

2.4 Actuators

DARwin-OP as a total of 20 Degrees of Freedom (DoF): 2 in the head, 3 in each arm and 6 in each leg, spread over 20 actuators. The actuators are high performance networked servomotors Dynamixel (DXL) MX-28 (in Figure 5), an improved version of the RX-28 series also produced by ROBOTIS. DXL has a very versatile expansion capability and can be used for multi-joint robot systems such as robotic arms, robotic hands, bi-pedal robots and others [2]. The firmware of the DXL is the only non-open source component of the DARwin-OP package.

Figure 5 The MX-28 [2]

2.5 Demonstration Programs

To showcase the capabilities of the robot, DARWIN-OP comes preconfigured with four modes of operations and ten pre-programmed actions [20]:

- Demonstration-ready mode means the robot is ready to initiate the demonstration;
- In Autonomous Soccer mode DARwin-OP pursues and kicks a red ball, playing soccer by itself;
- In Interactive Motion mode DARwin-OP performs eight pre-programmed motions sequentially, presented in Figure 6. The robot will "talk" while playing these motions;
- In Vision Processing mode DARWIN-OP performs the same motions as in Interactive motion mode, but individually, depending on the color(s) on the card presented in front of the robot.

Figure 6 Pre-Programmed actions [20]

2.6 DARwin-OP Framework SDK

Having been developed mainly as a platform intended to ease research and investigation and aid researchers, DARwin-OP offers a framework that can be used: for further development, as a reference/starting point or modified/extended as needed. This Application Programmer Interface (API) is composed of several parts divided into modules. The sub-controller is not addressed here because it was not part of the objectives of this project to work at that level and is therefore not relevant to the discussion. When referring to the DARwin-OP Framework we mean implicitly main-controller programming.

The framework is open source and comes with several code examples that showcase some of the capabilities of the robot and the framework itself. It is hierarchical, composed of several independent modules and operating system agnostic to facilitate portability to any existing or future system. A high level overview of

the framework is depicted in Figure 7. The programmer may however use any other independently developed software.

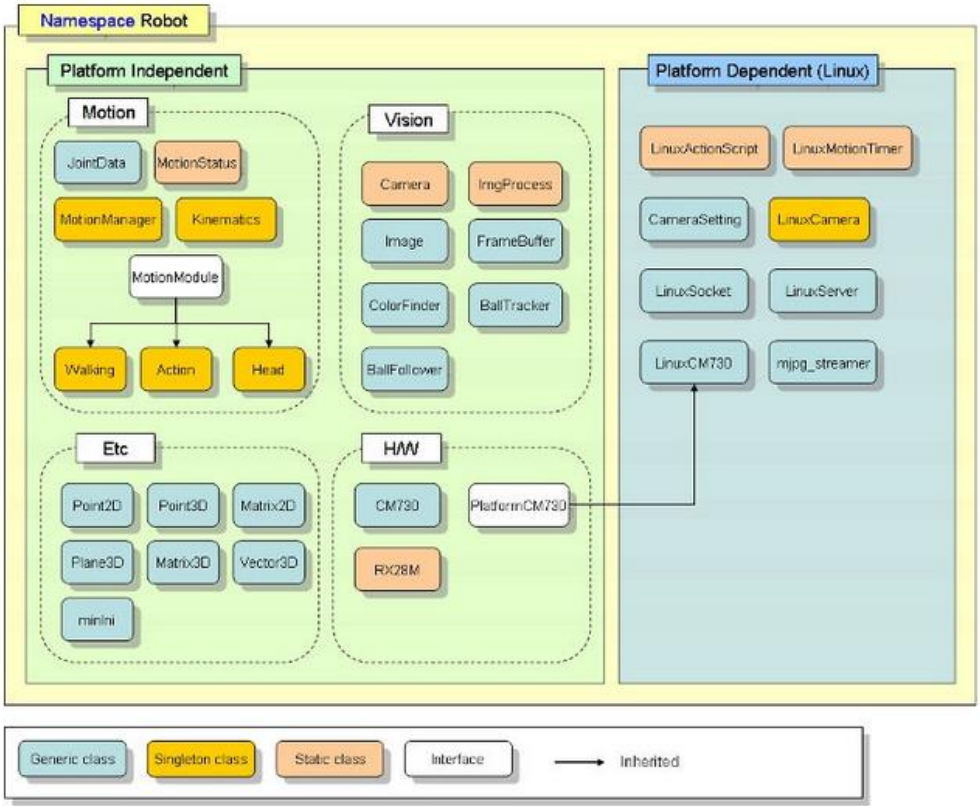


Figure 7 Software Framework for DARwin-OP [21]

- The **Motion** module is used to perform specific pre-configured motions such as walking and controlling the movements of the head.
- The **Vision** module is responsible for controlling the camera and image processing and acquisition.
- The **Etc** module contains mostly mathematical tools and also file read/write facilities used by the other modules.
- The **HW** module deals with the communication with the sub-controller and the servo-motors of the robot.

2.7 Problems

The main problem faced during the exploratory phase and subsequent development was the lack of proper documentation for the provided framework and code samples.

Other issues were encountered when trying to mix the different pre-programmed actions in custom ways other than the ones already provided.

DARwin-OP's onboard 4GB Solid State Drive (SSD) is not nearly enough to install all the needed software components.

The Operating System (OS) installed by default, Ubuntu 9.10, is utterly outdated making the integration with newer software solutions very difficult or in most cases impossible and the upgrading process of the OS to a newer version is error prone and not well documented.

Furthermore, the way the access to the CM-730 sub-controller was programmed in the framework does not allow having more than one client monitoring/controlling the robot at any given time, which proved to be painful when the project entered the ROS integration phase.

Another noticeable drawback of the framework is the use of tight while loop structures in detriment of an event driven programming architecture, to check starting and ending actions: animations, motion, etc. This polling architecture is very inflexible and not the most desirable performance wise. Threads were used to circumvent some of these limitations.

2.8 More Information

DARwin-OP is a fairly wide subject which cannot be fully explored here. For more information on the platform capabilities and startup, the following link should be visited http://support.robotis.com/en/product/darwin-op.htm. The framework and the included code examples can be downloaded from http://sourceforge.net/p/darwinop/code/HEAD/tree/.

Chapter 3

3. State Of The Art

Being a fairly recent platform, DARwin-OP as however received a lot of attention. Some of the most exciting projects involving this platform were carried out by universities, competing as teams in the many international competitions currently taking place. Also, important steps in the area of simulation were taken, with several projects aiming to fully integrate DARwin-OP in the most powerful and used open-source or commercial simulation software applications available.

3.1 Related Projects

Several projects have been developed involving the DARwin-OP platform. In this section we present an overview of the most relevant ones. When appropriate, we also discuss some of the advantages and disadvantages of each one.

3.1.1 Webots

Webots, from Cyberbotics Ltd., is a development environment used to model, program and simulate mobile robots. It offers a large choice of simulated sensors and actuators to equip each robot as well as a varied collection of commercial robot models. Several robots of different types can be used together and interact in complex, highly customizable setups and simulated real environments [6]. An example is depicted in Figure 8 below.

Using Webots the programmer can first test the robot behavior in a physically realistic world before transferring the program into the real robot, also using Webots. Although not being a free product, a license for education can be purchased and Webots remains a very popular product amongst universities and research centers worldwide. Recently, and as reported in [4], DARwin-OP was fully integrated with Webots.

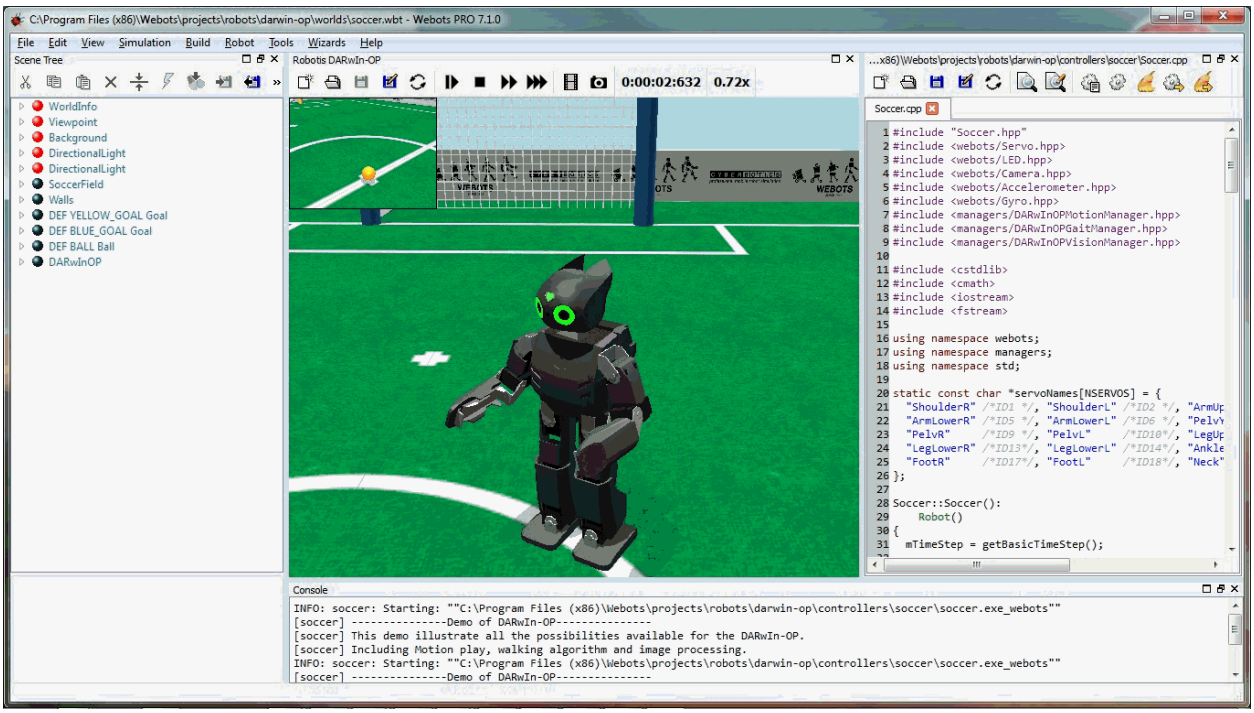


Figure 8 Webots Integrated Development Environment [6]

3.1.1.1 Advantages

Webots is a mature widely used application with a large community of users and lots of robot models and simulated real world environments to choose from out of the box.

3.1.1.2 Disadvantages

The main disadvantage of Webots is the fact that it is an expensive commercial software, offering a 30 days trial license, clearly insufficient for the timeline execution of this project. Also, integration with DARwin-OP was only achieved very recently by a project developed in 2013 for that purpose [6].

3.1.2 ROS

ROS stands for Robot Operating System and is a flexible framework for writing robot software, built from the ground up to encourage collaborative development. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms [7]. It is totally open source and a myriad of open-source high quality professional level tools, such as the Rviz and Gazebo simulators, are available for use out of the box. For all of these reasons its popularity has been growing rapidly.

In terms of communication infrastructure, ROS implements the publish/subscribe model. A process is called a node in ROS parlance and nodes communicate using topics. A topic is a name that is used to identify the

content of the message. A node sends out a message by publishing it to a given topic acting as the publisher. A node interested in a certain kind of data will subscribe to the appropriate topic playing the role of the subscriber. The general architecture of ROS is presented in Figure 9, the publish/subscribe model is depicted in Figure 10.

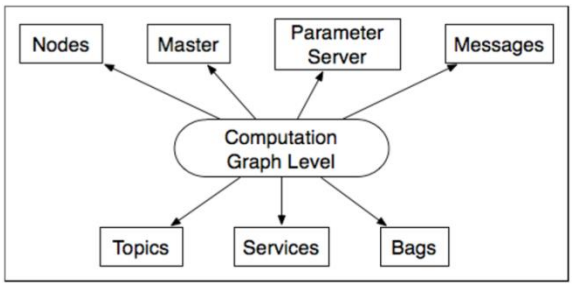


Figure 9 ROS architecture [7]

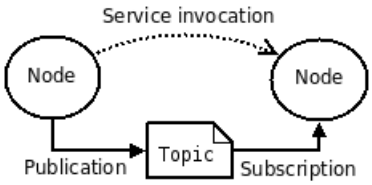


Figure 10 Publish/Subscribe Model [7]

3.1.2.1 Advantages

ROS is completely open source, very popular and with a huge community of users and developers worldwide. Several packages and tools have been developed to integrate with ROS and aid development considerably. ROS uses the publisher/subscriber model based on topics to abstract and simplify many of the low level details faced by other communication models.

3.1.2.2 Disadvantages

ROS needs some setup steps prior to its fully utilization, such as proper network publish/subscriber configurations and installing the appropriate packages. While most of these steps can be taken by simply following one of the many online tutorials devoted to this issue, some problems may still be encountered related to different system setups and version conflicts. Also a learning curve is to be expected.
Another disadvantage to consider is the fact that ROS is primarily targeted at Ubuntu and follows its release cycle, that is, each new release of ROS targets a specific release of the Ubuntu distribution.

3.1.3 Gazebo

Gazebo is an open source project simulator that integrates with ROS offering the ability to accurately and efficiently simulate populations of robots in complex indoor or outdoor environments. It provides a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces [9]. DARwin-OP is also supported as shown in Figure 11.

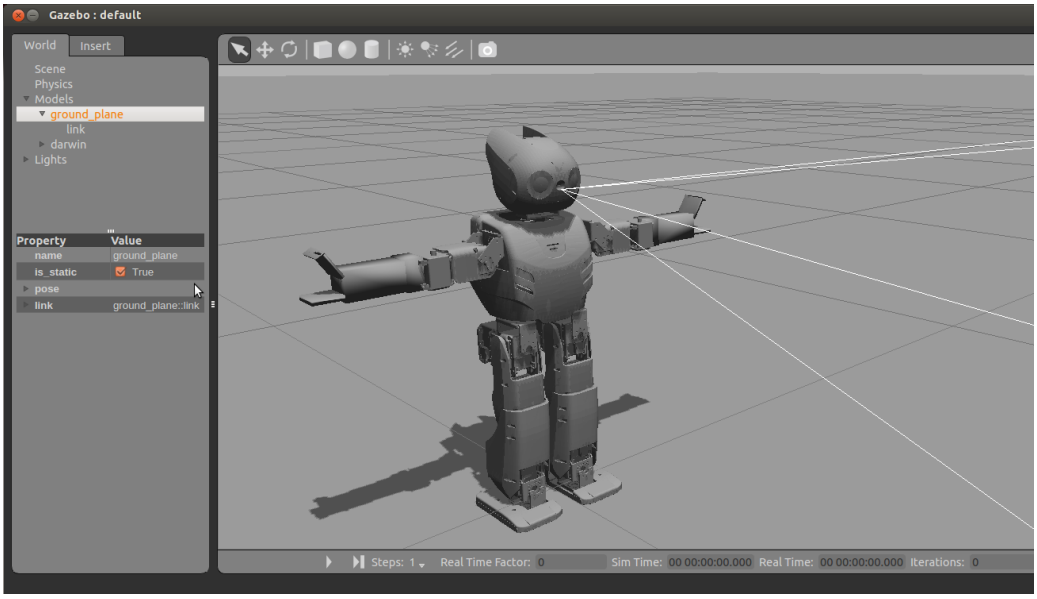


Figure 11 DARwin-OP simulated in Gazebo

3.1.3.1 Advantages

Gazebo is fully integrated with ROS and offers an open source professional quality tool for the purposes of simulation. Amongst its many features are a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces.

3.1.3.2 Disadvantages

To simulate a robot in Gazebo we need a Unified Robot Description Format (URDF) model of the robot. The construction of a URDF model of DARwin-OP is not necessarily a trivial exercise, at least for us, and we used an already existing URDF model of DARwin-OP [27].

Another issue to take into consideration is the fact that the simulated robot can move, at times, as if floating in thin air and can easily fall over, in some instances, just by moving its arms.

Gazebo was used in our project in the ROS integration phase.

3.1.4 Rviz

Rviz is a 3D visualization tool for ROS available as a ROS package. Using Rviz, you can visualize the robot's current configuration on a virtual model as well as live representations of sensor values coming over ROS Topics, including camera data and more.

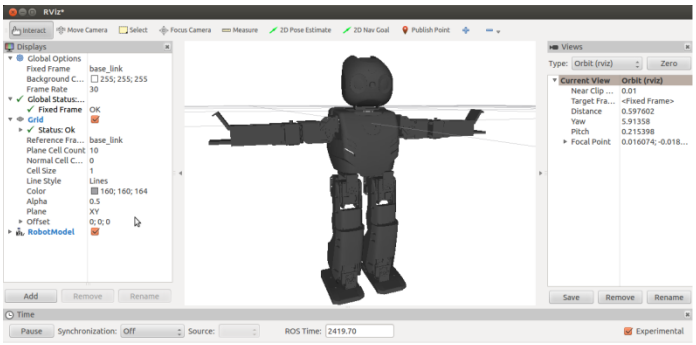


Figure 12 DARwin-OP simulated in Rviz

We did not use Rviz in our project because loading the URDF model of the robot, that we used, caused an error most of the time and it seemed to need Gazebo running, to work properly with the URDF.

3.1.4.1 Advantages

Rviz, similarly to Gazebo is completely open source and fully integrated with ROS. The robot can be easily simulated and access to the sensor's readings is also a trivial task.

3.1.4.2 Disadvantages

Rviz needs ROS to be installed on the system and like Gazebo needs an URDF file describing the robot's model.

3.1.5 Team UPennalizers

The University of Pennsylvania has a robotics team known has the UPennalizers associated with humanoid research, that often participates in the RoboCup competition with excellent results: namely on the Kid-sized Humanoid League using DARwin-OP [22]. The team has a wiki and shares its code base in GitHub at https://github.com/UPenn-RoboCup/UPennalizers. They offer an alternative approach to the DARwin-OP Framework supplied by ROBOTIS in that they have implemented their own framework independently.

3.1.5.1 Advantages

Team UPennalizers presents a full blown alternative to the ROBOTIS framework. Amongst many other features, this implementation offers forward and inverse kinematics solutions as well as cooperative communication between the robots.

3.1.5.2 Disadvantages

Team UPennalizers performs research on many humanoid platforms besides DARwin-OP, so their framework was designed to transparently accommodate the differences between them. Therefore their code base is very extensive and somewhat complex.

One other issue is ill documentation and the fact that in order for their most recent code releases to properly function we need to upgrade the robot OS to the Ubuntu 10.10 release.

Also, care should be taken to guarantee a safely usage of their code, to avoid the risk of damaging the robot. It is very important to pay attention to the robot firmware that they are using and to know what their code is doing and the context/conditions in which it is meant to be used.

3.1.6 FC Portugal

Although not directly related to DARwin-OP we mention team FC Portugal, namely the project implemented by Nuno Almeida [23] since some of its code base was used as the base for DARwin-OP's 2D model manipulation and visualization. FC Portugal is a joint project of University of Aveiro, University of Porto and University of Minho. The team participates in RoboCup competitions since 2000, mostly but not exclusively in the simulation category.

3.2 Technology

In this section we present a brief introduction to the various technologies involved or used in the development of the project.

3.2.1 Ubuntu

Ubuntu is an open source, freely available high-quality operating system built around the Debian Project [10], one of the most established versions of Linux. Since its inception Ubuntu has followed the motto "*Linux for humans*", that is, Ubuntu thrived to break out of the traditionally technically demanding world of open source and bring Linux to everyone instead, by providing an easy to use user interface and a powerful suite of office and internet software by default.

Ubuntu started mainly as a desktop solution and has been recently voted the most popular desktop Linux [29]. At present Ubuntu is everywhere and is considered the leading operating system for the PC, phone, tablet, server, the cloud and there is even a new exciting project for the TV [8]. DARwin-OP comes pre-installed with Ubuntu 9.10.

3.2.2 Qt

Qt ("*cute*") is a cross-platform application framework, available with both commercial and open source licenses, widely used for developing software that can be run on various platforms. It has a powerful built in Integrated Development Environment (IDE) and a graphical user interface designer that greatly aids the development of software applications [11]. It is mainly used to program in C++.

3.2.3 C++

C++ is a relatively mature general-purpose programming language developed by Bjarne Stroustrup at Bell Labs, starting in 1979 and initially standardized in 1998 (c++98). It is an efficient and flexible language for building fast, efficient, mission-critical systems. In the early '90s, C++ was the most popular object-oriented language and it has greatly influenced some of the most popular managed languages such as Java and C# [12]. DARwin-OP Software Development Kit (SDK) and samples are programmed in C++.

3.2.4 Qt Widgets for Technical Applications

Qt Widgets for Technical Applications (Qwt) is a library that contains GUI Components and utility classes which are primarily useful for programs with a technical background. Besides a framework for 2D plots, it provides scales, sliders, dials, compasses, thermometers, wheels and knobs to control or display values [13].

3.2.5 Boost

Boost is a free peer-reviewed bundle of portable C++ source libraries intended to be widely useful and usable across a broad spectrum of applications. The Boost license encourages both commercial and non-commercial use [14].

3.3 Conclusion

Being very mature and high quality solutions, the projects described here showed some disadvantages worth noting. As for Webots, being subject to a license purchase was considered a serious downside. On the other hand development with the real robot was also preferred at this stage in favor of the use of a simulator. Lastly, the integration with DARwin-OP is still recent and might present some residual issues.

ROS however presents a totally different matter. It is an open source project bundled with powerful professional level tools and very desirable to use and implement. In fact one of the goals of the project was to integrate with ROS via a wrapper. Unfortunately, the Ubuntu distribution that comes installed by default with DARwin-OP, Ubuntu 9.10 Karmic, being utterly outdated does not integrate with ROS and the upgrade process is ill documented and very error prone at this moment. Furthermore DARwin-OP ´s fully integration with ROS is being made as we speak, making the process rather laborious and difficult to achieve.

The use of other frameworks and or sample code for DARwin-OP made by third party individuals or teams, like the framework proposed by the UPennalizers [22], must be taken with some care and analysis.

The fact that we were working with an expensive and sensitive equipment was always a concern in our minds. Some of the projects mentioned here aim to deal with this issue by using simulation. This is of course an advantage.

Chapter 4

4. System Requirements and Architecture

4.1 System Requirements

This section presents the overall system requirements specification for the monitoring and control application. The requirements elicitation process is described first. The context, the actors and their respective use cases follow next. Finally, we address the non-functional requirements and the system's assumptions and dependencies. We also mention additions and or changes to the initial requirements that took place during the development phase due to the refinement of the problem domain.

4.1.1 Requirements Elicitation

The requirements elicitation underwent a phased process. A first informal interview with one of our project advisors/stakeholder Professor Nuno Lau was held in order to define the scope and overall goals of the application. During this meeting a working plan comprised of 3 main tasks was devised and is presented below:

- **Task 1** - Develop an intuitive graphical interface to control the DARwin-OP behavior with the option to parameterize the behavior in question. The application should also allow for the monitorization of the sensors integrated in the robot and any other internal state parameters considered relevant;

- **Task 2** - Integrate the application created in task 1 with ROS evaluating the advantages and disadvantages of this new implementation when compared with the initial one;

- **Task 3** – 2 possible routes could be taken: further develop the actuation ability of the robot by improving its pre-existing behaviors or developing and integrating new ones; or further develop the robot's perception capabilities by improving its vision and environment estimation algorithms, specifically self-location and reference points localization.

The next phase consisted mainly in an exhaustive research for similar projects, research studies and technologies comprising the state-of-the-art on this subject. The plethora of resources collected further improved our comprehension of the domain problem and served as an invaluable source of inspiration that ultimately led to the improvement and refinement of the requirements initially specified.

Given all these insights and after weighting all the possibilities, the following general decisions were made:

- A control and monitoring application was to be developed. This application should be distributed and comprised of 2 independent modules hereafter designated as the Server and the Client. Given the nature of the application no special log in or authentication process should take place;

- The Client module, running in some external machine, should consist of a graphical interface, allowing the user to easily and intuitively monitor and control the behavior of the robot. The user should be able to freely choose the relevant sensors to monitor at any time. The behaviors should be parameterized when applicable. For simplicity only the walking behavior, more specifically its duration, was to be subject of this parameterization;

- The Server module, running on the robot, should listen for client requests and act accordingly either by sending the requested sensors readings or by controlling the behavior of the robot as intended. To minimize bandwidth usage the Server should only send the requested readings and from these exclude the ones found unchanged. Given the resource restraints imposed by the robot itself, and the way that its framework was implemented, only one client at a time should be allowed to connect to the server;

- Being a distributed system, a communication protocol between the Server and Client modules had to be implemented. Sockets over TCP were the chosen technology for the task; the socket technology due to their platform independence, the TCP protocol due to its simplicity of deployment and reliability mechanisms. As for the data encoding, Javascript Object Notation (JSON), being a very lightweight, system independent, human readable data-interchange format, seemed appropriate. Finally, taking into account the different types of messages to be implemented and the need to properly encode image data, a suitable message format convention should be devised;

- Integration with ROS was to be delayed until the conclusion of the control and monitoring application. Further investigation on this subject should be resumed then;

- It became clear that the robot´s OS, Ubuntu 9.10, was outdated and should be upgraded to a newer version, Ubuntu 12.04. The lack of hard drive space was also a concern. After some thought it was decided not to upgrade the robot at this stage and delay the decision to a later more favorable moment. The possibility to deploy the system in an external drive should also be considered.

During the initial phase of the project, we explored the ROBOTIS framework provided samples and tools, we experimented with the UPennalizers [22] framework and we created several test programs using the ROBOTIS framework. All the source code for this project can be found in the project's web site at [28].

We realized that both frameworks were poorly documented and in the case of the UPennalizers [22] framework additional dependencies were needed. This meant that DARwin-OP OS should be upgraded. Furthermore, the UPennalizers [22] framework has extra levels of complexity (deals with team coordination, localization, supports several platforms, etc) and caused unexpected behavior on the real robot such as awkward joint positions and errors only cleared after a reset using the DXL Monitor tool. Therefore, we decided to delay the use and exploration of the UPennalizers [22] framework to a more convenient time frame during the project's life cycle.

4.1.2 Use Context description

The application is expected to be used in a variety of different scenarios: to carry out research in a calm controlled environment; for demonstration purposes at schools and other events taking place in uncontrolled noisy spaces; or to be used simply by curiosity, possibly under the guidance of an experienced user. Nevertheless the procedure is identical in all cases. No previous registration is needed, the user simply connects to the robot, the proper address and port must be provided on first use only or in case of change. From then on, usage is straightforward.

To start/stop monitoring any sensor the user just has to check/uncheck the proper option box provided at the top of each tab. To manipulate the behavior of the robot the user chooses from a list of predefined behaviors and presses the button play/stop.

With just a few clicks, and in an expectable short amount of time, any user should be able to effortlessly manipulate and monitor the robot.

4.1.3 Actors

The target user of the client module of the application is expected to have some degree of knowledge and experience in the field of robotics, however the user interface was designed in such a way to mitigate the possible lack of expertise in this field and with little effort and training just about anyone can use and benefit from the application. Next we enumerate a possible classification of the actors expected to use the application, but it is worth emphasizing that the distinction between them is merely semantic, given that all the functionalities are equally available to all of them.

- **Researcher** – Has a high level of expertise in the field of robotics and can interpret the readings of the sensors and easily understand the basics to manipulate the robot. Wishes to fully use the potential of the application.

- **Student/Graduate** – Might not have previous experience in robotics and is just curious. Will most likely use the robot's manipulation features of the application only.

- **Operator** – Similar to the previous actor but with a lower level of knowledge. The generic anonymous user. Will most likely need training before using the application and supervision during its usage.

4.1.4 Use Cases

In Figure 13 we present the use case model of the control and monitoring solution. Only the Client module is discussed since it is the only one directly used by the user. The Server module should run without any direct user intervention apart from the start procedure (the start procedure can be automated) and is therefore excluded from any further analysis. Also, for simplicity and since the differences between all the mentioned actors are purely semantic we have aggregated them in a general actor designated as the Operator.

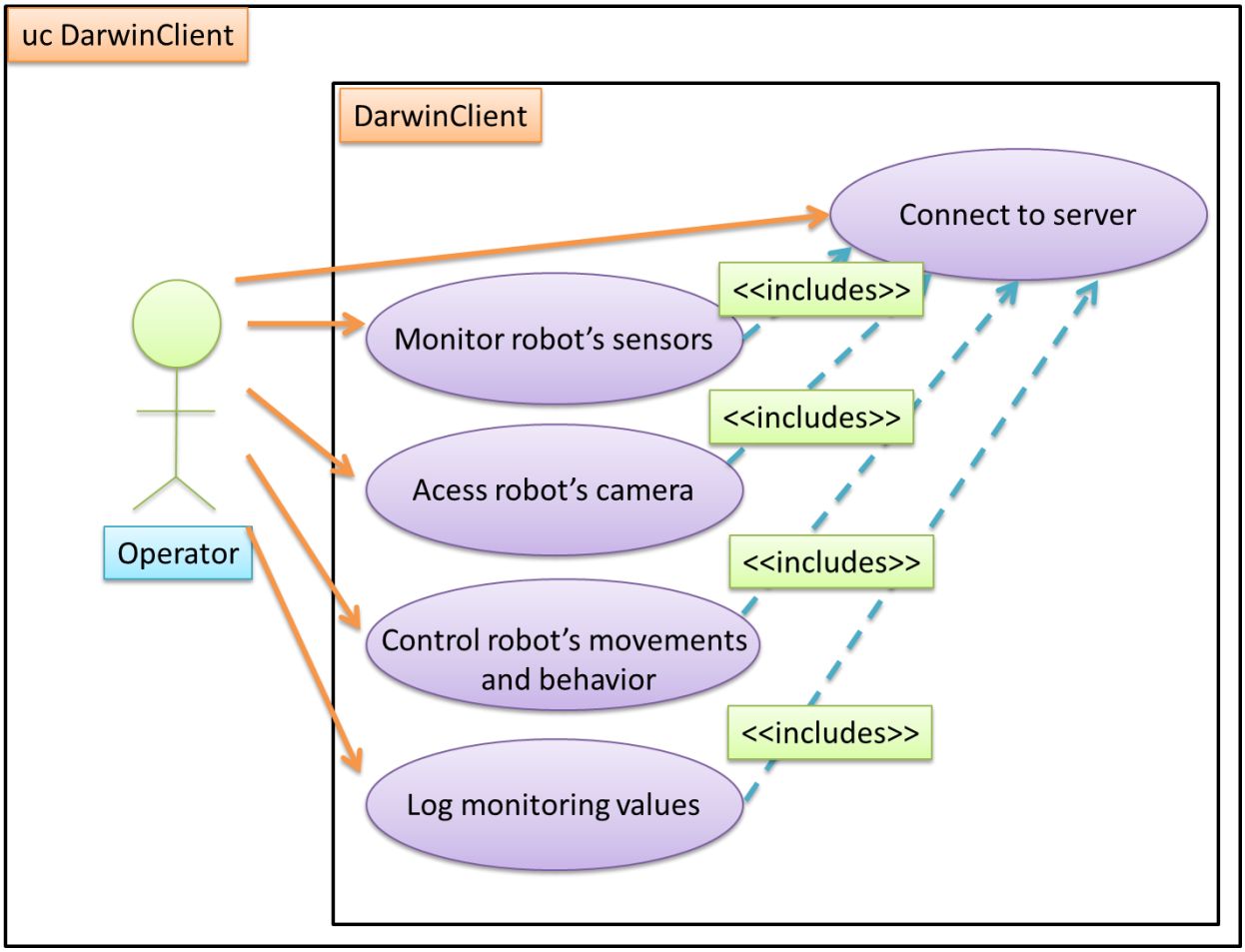


Figure 13 Use cases of the Client module

Use Case description:

- **Connect to server:** establishes communication with the Server module. All the other use cases presented depend on this action being taken prior to their execution.
  Priority: High.

- **Monitor robot's sensors:** allows the user to choose the relevant sensors to monitor in real time.
  Priority: High.

- **Access robot's camera:** allows the user to see what the robot is seeing.
  Priority: High.

- **Control robot's movement and behavior:** allows the user to manipulate the position of the joints of the robot individually or all at once. The user can also choose a sequence of robot behaviors from a list of pre-defined behaviors to build a choreography that the robot will perform.
  Priority: High.

- **Log robot's sensor:** allows the user to save the monitorization session to a local file for later analysis.
  Priority: Medium.

4.1.5 Non-functional Requirements

Below we present the non-functional requirements of the application.

- **Usability** – Given the wide spectrum of different users and the fact that no prior assumptions concerning their different levels of expertise are made, it is essential for the application to be intuitive and easy to use. Also, the system should be responsive at all times and show helpful messages to better guide the actions of the user in case of error.
  Priority: High.

- **Reliability** – Given the fact that the application manipulates a fairly expensive and fragile piece of hardware, the robot itself, the application should be robust and reliable enough not only to guarantee the safety of the robot during operation but also to instill a sense of confidence on the part of the user.
  Priority: High.

- **Interoperability** – The server module should be designed in a way that allows any type of client to use its services as long as that client implements the same communication protocol.
  Priority: High.

- **Efficiency** – The server module will operate on an environment with limited resources, the robot, and should therefore have the smallest footprint possible in order not to interfere with the robot's operations. Resources such as the camera should be carefully used because of their intensive resource consumption which can interfere with the robot operations such as walking causing it to walk erratically and possibly fall down.
  Priority: High.

- **Portability** – The Client module is not intended to be deployed in any other system besides Ubuntu.
  Priority: Low.

- **Internationalization** – The user interface should allow easy internationalization to many different languages.
  Priority: Medium.

4.1.6 Assumptions and Dependencies

In order for the application to work as expected, access to a network but not to the internet is required. Also, the Server module must be running on the robot at all times. The Client module was developed and tested in Ubuntu using a cross-platform framework, therefore theoretically the client could also run in other distributions of Linux, Windows and Mac. However it is assumed that the client will be used only in an Ubuntu distribution, preferably version 14.04 LTS or newer.

4.2 System Architecture

This section presents an overview of the system's architecture, its domain, physical and technological models and underlying relationships.

4.2.1 Domain Model

The domain model, depicted in Figure 14, describes the various entities, their attributes and relationships and also the constraints that govern the problem domain.

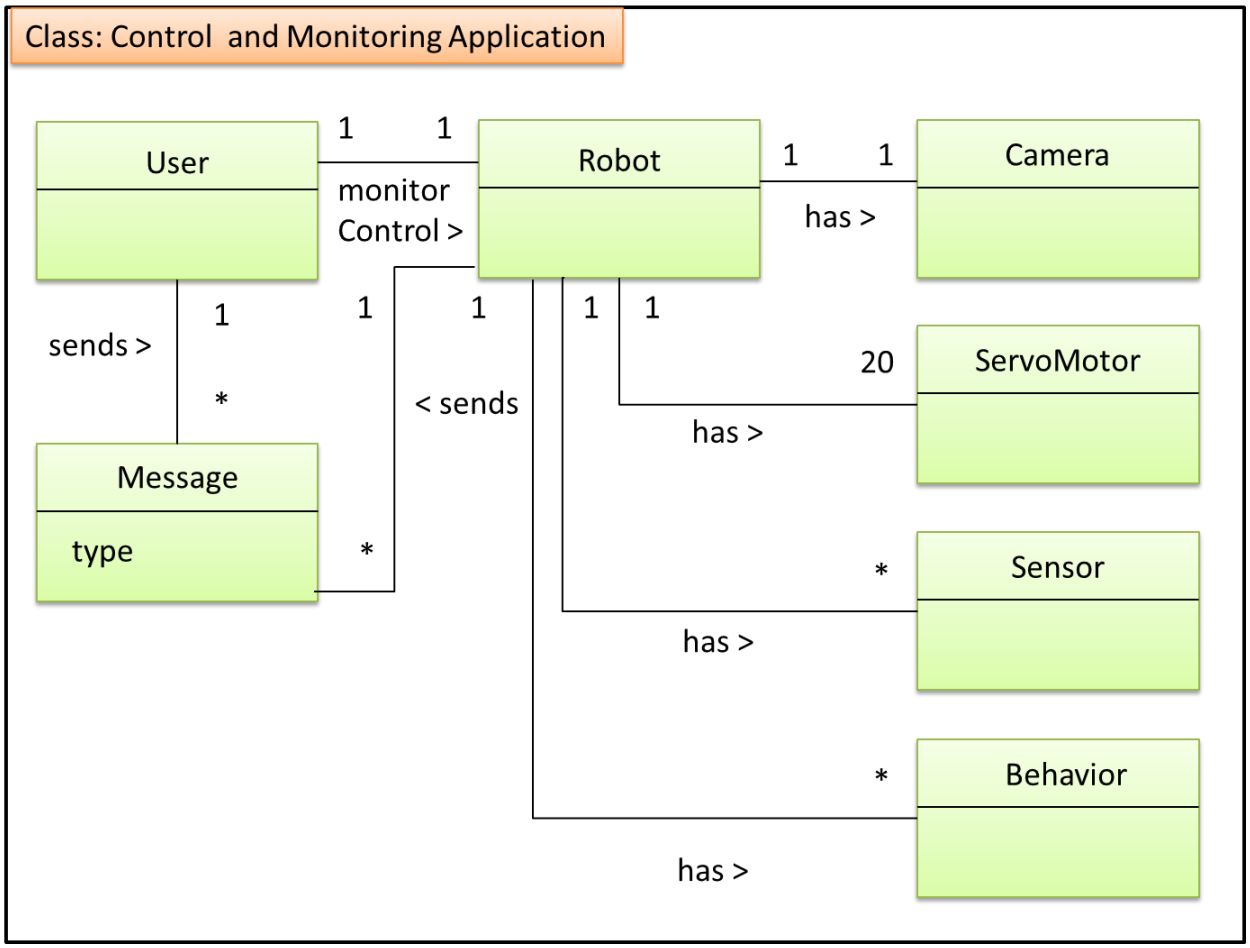


Figure 14 Control and Monitoring Application Domain Model

The User entity in this case represents the Client module of the application, while the Robot entity represents the robot itself as well as the Server module. As we can see, the user and the robot communicate through messages. These messages have an identifier type. Both the user and the robot can send more than one message but it is worth noting that only one user at a time might control and monitor the robot. Finally, we

can also see that the robot is composed of several elements; one camera, 20 servomotors and several sensors and pre-defined behaviors.

4.2.2 Physical Model

Figure 15 presents a high-level view of the physical architecture of the system: the components, their interaction and deployment.

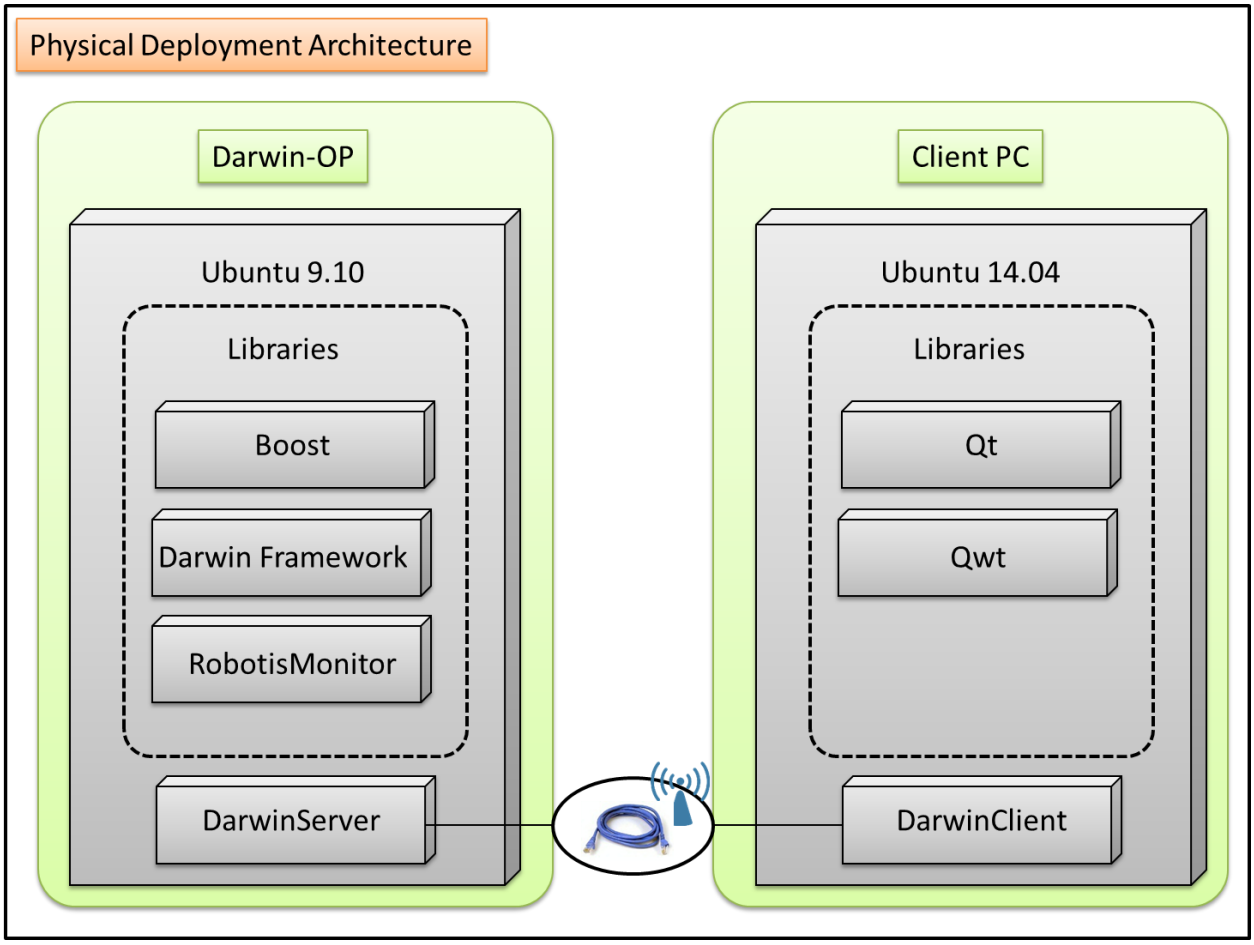


Figure 15 Physical Model

- **DARwin-OP** represents the real robot.
  - Is running Ubuntu 9.10 OS.
  - The Server module named as DarwinServer is installed and should be running all the time.
  - In order for the Server module to run properly, several dependencies, libraries in this case, must also be installed; Boost, the DARwin-OP Framework and RobotisMonitor, a wrapper specifically implemented for the ROBOTIS Framework and discussed in further detail in section 5.4.

- **Client PC** represents the user's computer.
    - The client can be any device running Ubuntu 14.04 OS or a newer version.
    - The Client module named DarwinClient must be installed.
    - In order for the Client module to run properly several dependencies must also be installed, Qt and Qwt in this case.

- The communication Client/Server is established over an Ethernet or wireless network.

4.2.3 Technological Model

The technological model is depicted in Figure 16 giving a high level overview of the technologies used by the system.

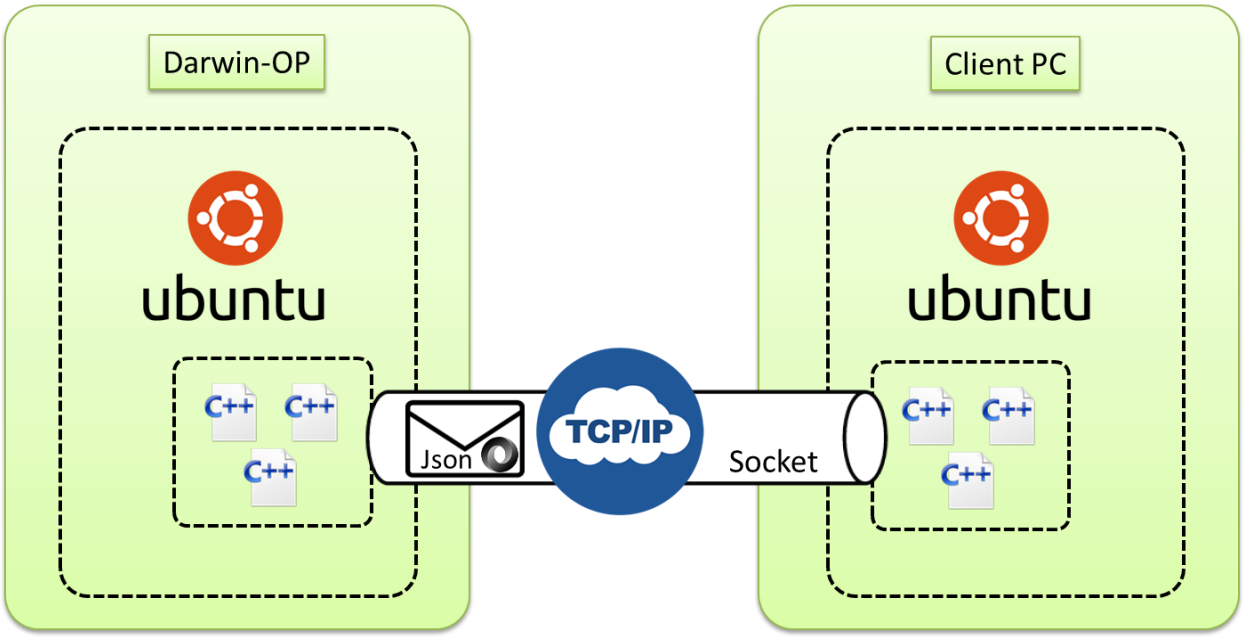


Figure 16 Technological Model

Both the robot, where the Server module is running, and the user's device, where the Client module is deployed, operate on Ubuntu. The modules communicate with each other through a TCP/IP Socket using a message format encoded in JSON.

Chapter 5

5. Implementation

The Control and Monitoring Application is comprised of two modules, outlined next in Figure 17.

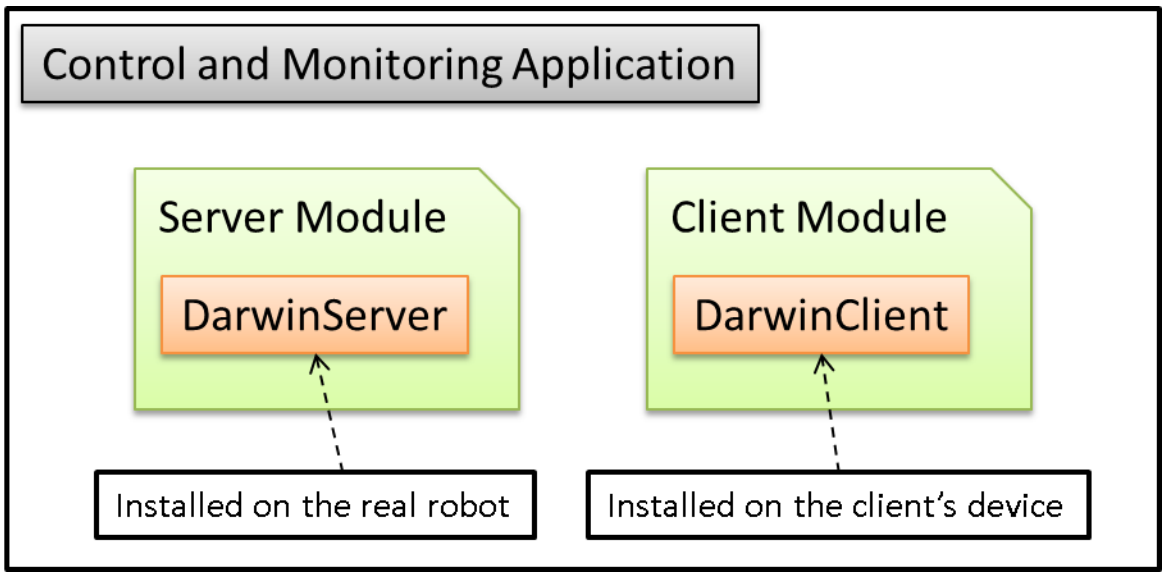


Figure 17 Application modules

Figure 17 presents the modules comprising the control and monitoring application.

- The Server module is comprised of the DarwinServer application installed on the real robot.
- The Client module is composed by the DarwinClient application installed on the user's device.

Next we further detail these modules.

5.1 Server Module

In this section we depict the Server module in more detail. We begin by enumerating the main composing classes, shown below in Figure 18.

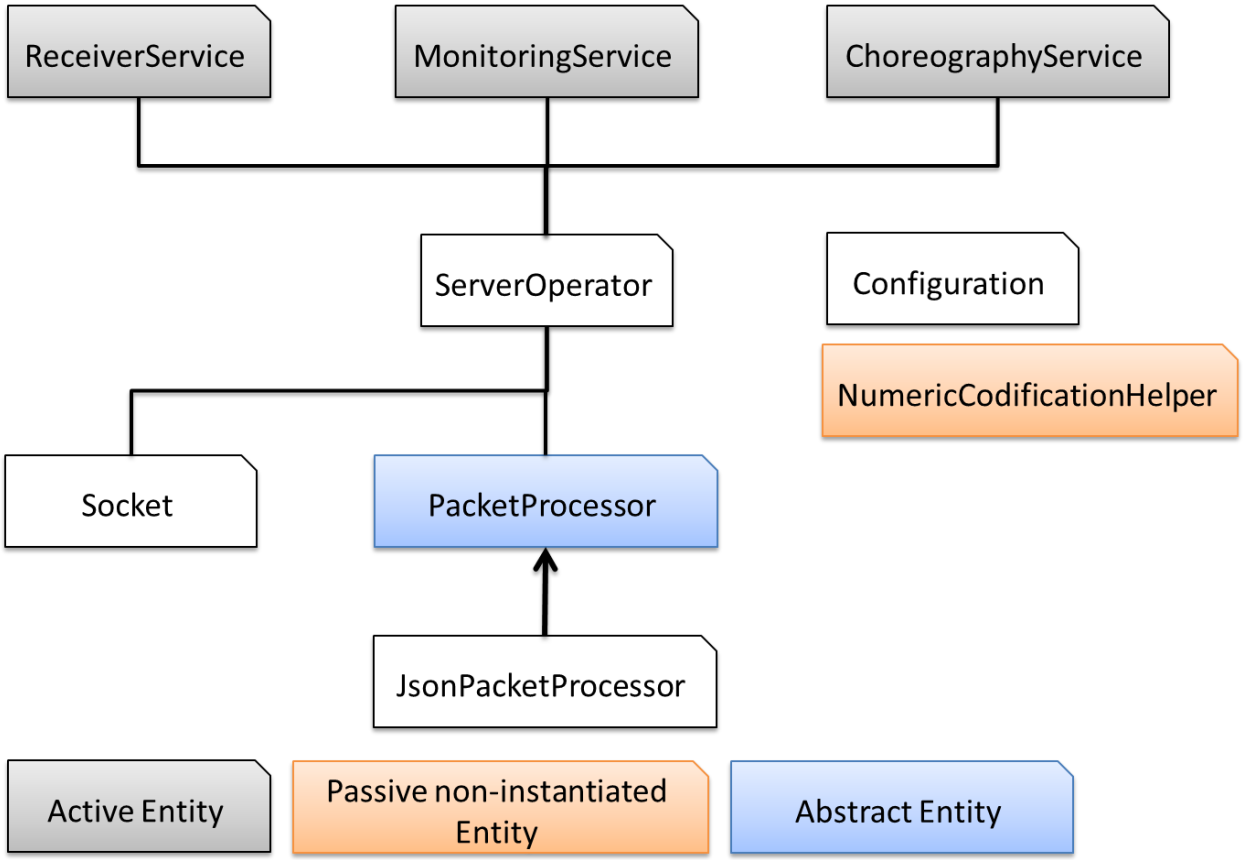


Figure 18 Server module classes

- **ServerOperator** class serves as a mediator between all classes except for Configuration and NumericCodificationHelper, allowing for their communication.

- **Socket** class establishes the Transmission Control Protocol (TCP) Socket communication channel and listens for incoming client requests. This same channel is used to send the appropriate response. This class encapsulates the use of Linux sockets.

- **JsonPacketProcessor** class is a specialization of the **PacketProcessor** class used to encode and decode respectively: the messages sent by the server in response to the client's request (encode) and the requests of the client (decode).

- **NumericCodificationHelper** class is a helper class used to create the header of the message. The format of the messages used will be detailed later on this chapter.

- **Configuration** class holds the configuration parameters of the Server module. The configurations are stored in a local file encoded in Extensible Markup Language (XML).

- **ReceiverService** class is responsible for redirecting the request of the client to the corresponding service that should handle it. In this case only two other services exist; the **MonitoringService** and the **ChoreographyService**.

- **MonitoringService** class handles monitoring requests.
- **ChoreographyService** class handles control and manipulation requests.

The following images depict the interaction diagrams between these classes.

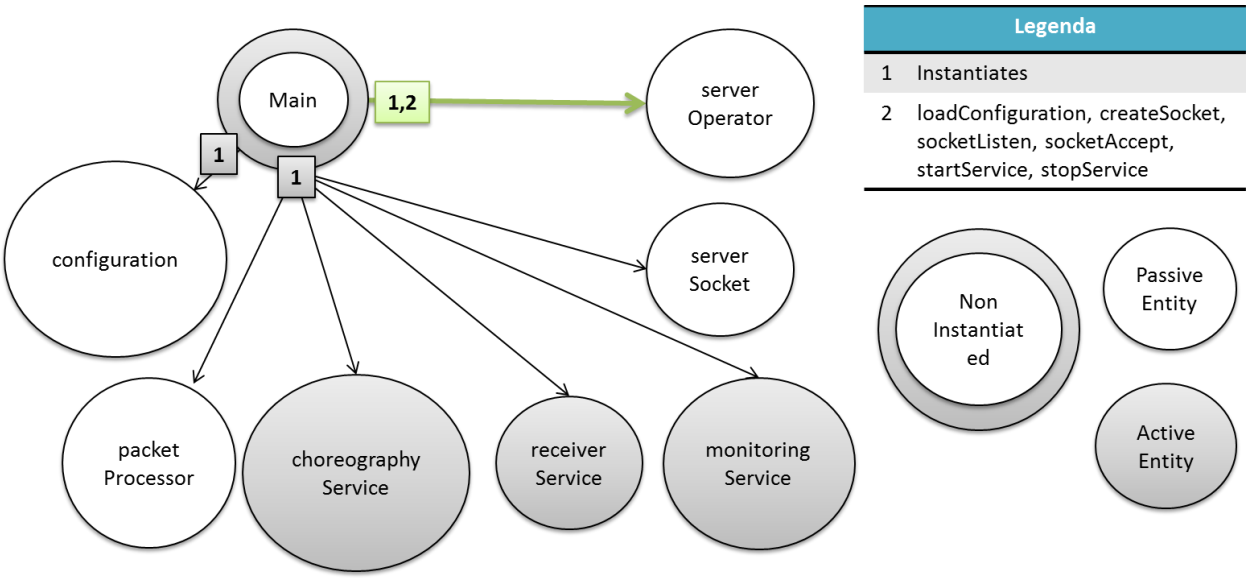
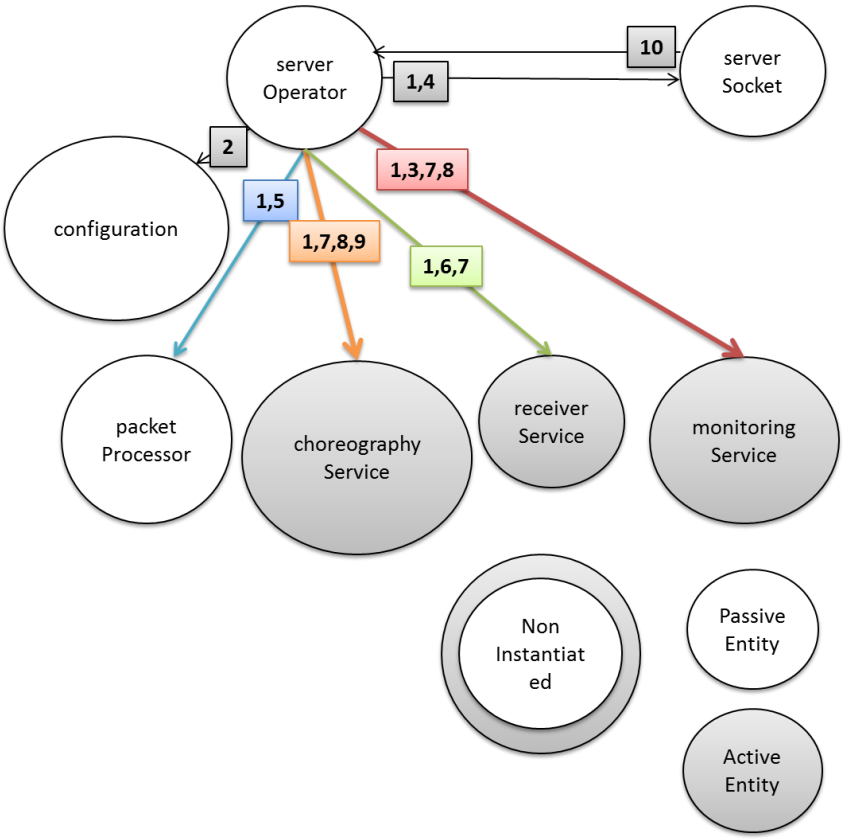
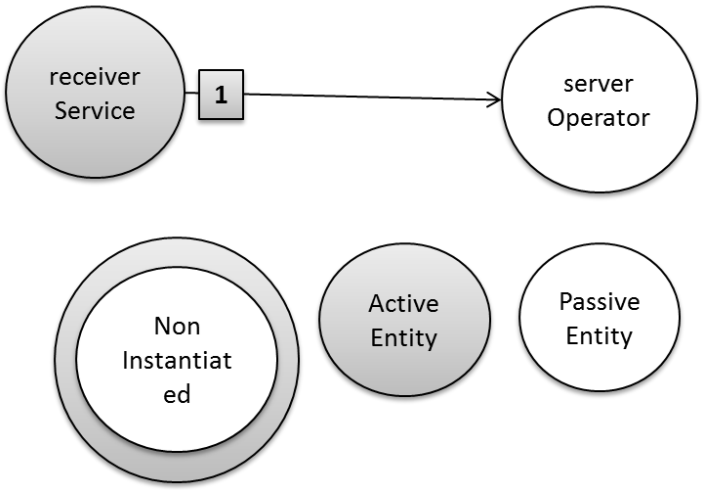


Figure 19 server main interaction diagram



Legenda	
1	setMediator
2	load
3	setTimeInterval
4	createSocket, socketListen, socketAccept, sendPacket
5	processPacket, buildMonitoringPacket, buildChoreographyResponsePacket
6	addMessage, resetServiceState
7	start, stop
8	pause, resume
9	setChoreography, setWalkOperation, setOrchestrationMode, isChoreographyRunning, stopChoreography, processChoreographyStatusPacket
10	clientDisconnected, receivedPacket

Figure 20 serverOperator interaction diagram



Legenda	
1	processPacket, isChoreographyRunning, processChoreographyStatusPacket, stopChoreography, processChoreographyStatusPacket, setChoreography, setOrchestrationMode, resumeService, setWalkOperation, getMonitoringTable

Figure 21 receiverService interaction diagram

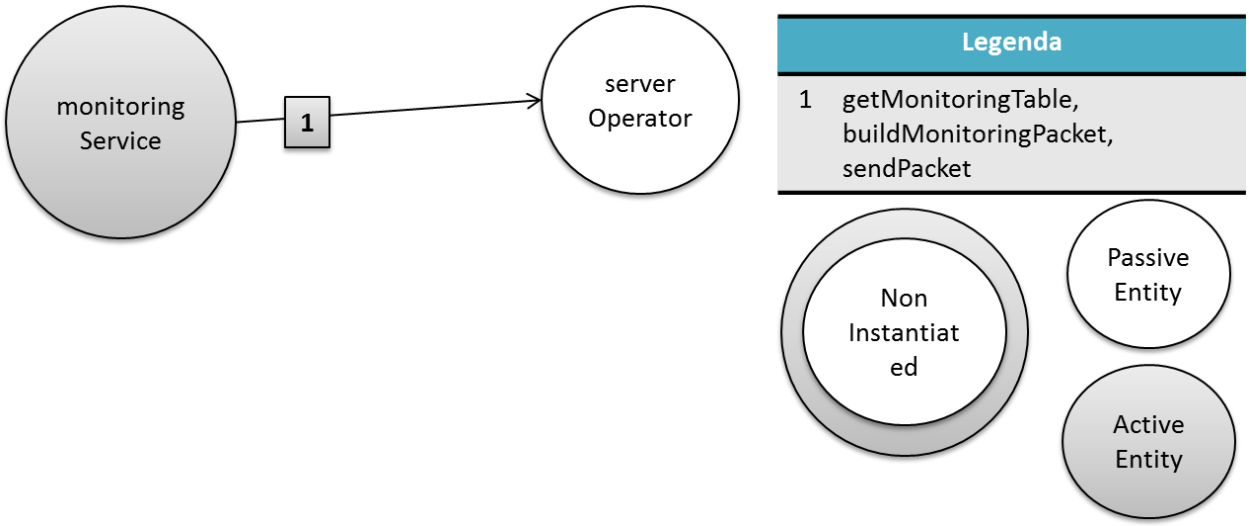


Figure 22 monitoringService interaction diagram

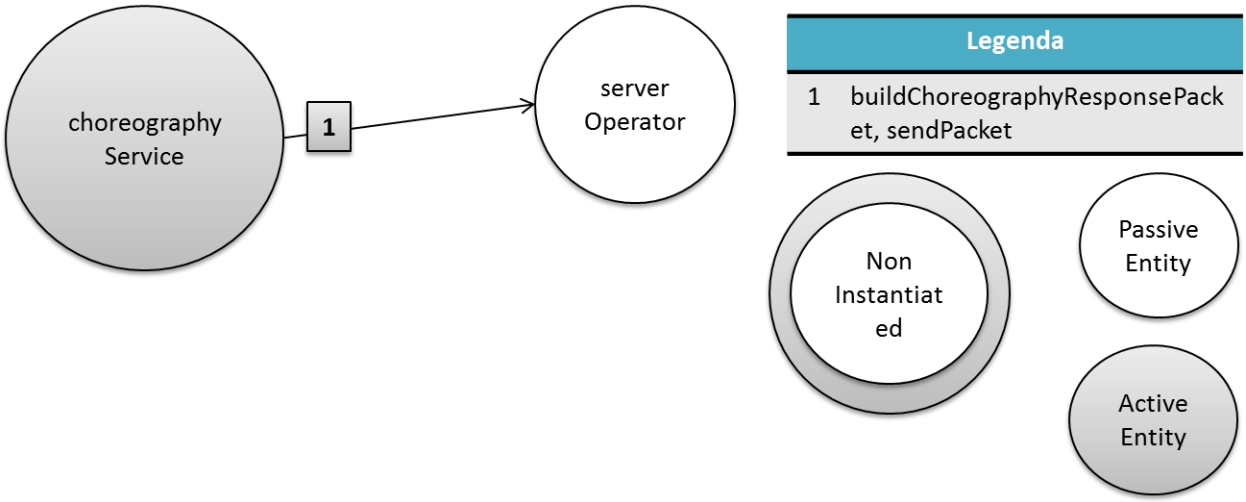


Figure 23 choreographyService interaction diagram

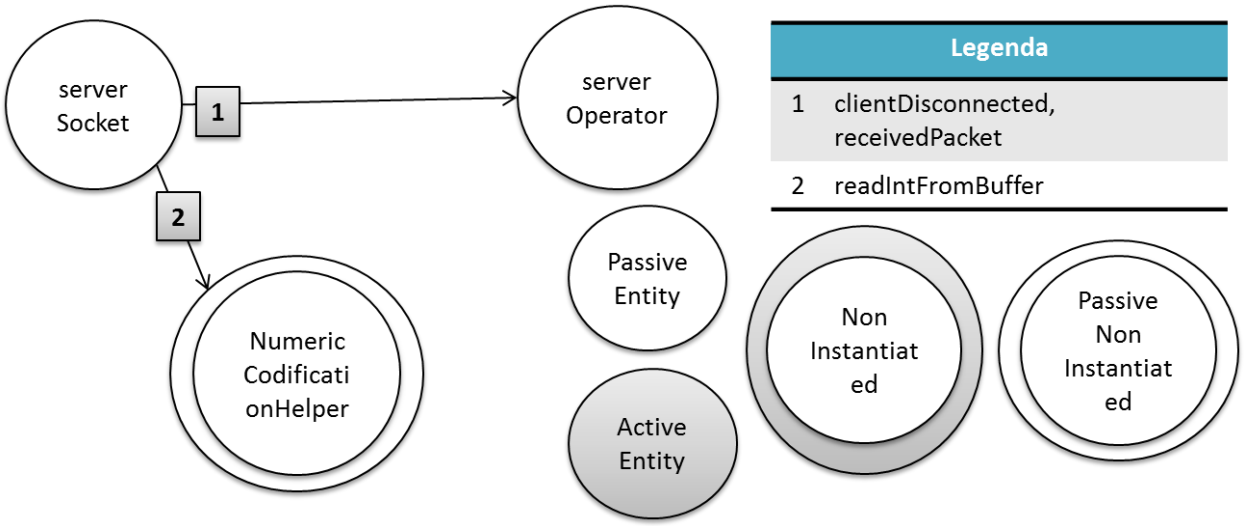


Figure 24 serverSocket interaction diagram

5.2 Client Module

In this section, we depict the Client module in more detail. We begin by enumerating the main composing classes shown below in Figure 25.

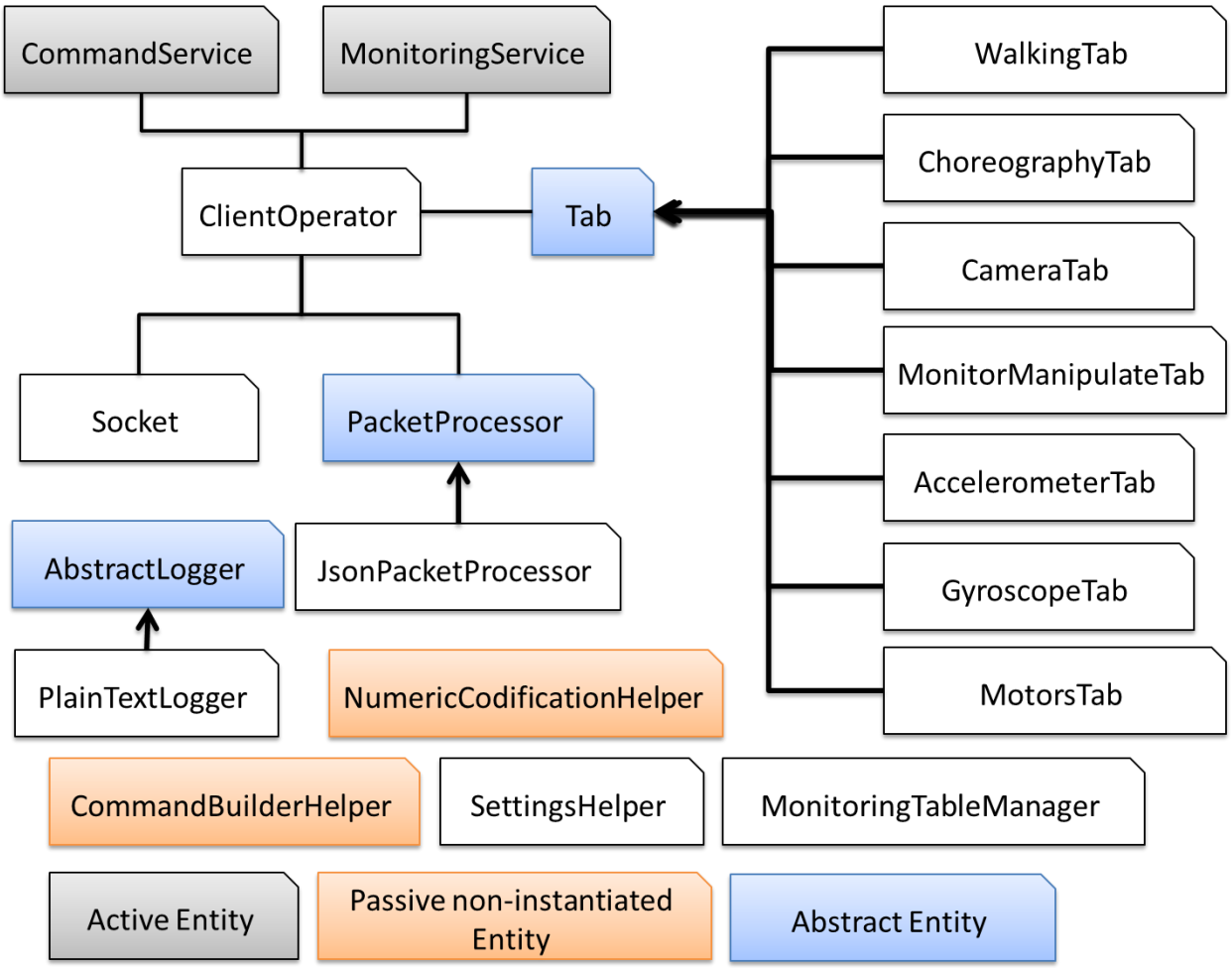


Figure 25 Client module classes

- **ClientOperator** class serves as the mediator between all other classes, except the Helpers, the PlainTextLogger and the MonitoringTableManager, allowing for their communication.

- **Socket** class establishes the Transmission Control Protocol (TCP) Socket communication channel to send the client's requests and receive the corresponding replies.

- **JsonPacketProcessor** class is a specialization of the **PacketProcessor** class used to encode and decode respectively; the requests of the client (encode) and the replies from the server (decode).

- **NumericCodificationHelper** class is a helper class used to create the header of the message. The format of the messages used will be detailed later on this chapter.

- **PlainTextLogger** class is used to generate a local file in plain text containing the readings monitored. This class is a specialization of the **AbstractLogger** class.

- **CommandBuilderHelper** class is a helper class used to translate the user's actions into a known internal structure.

- **SettingsHelper** class is a helper class used to manage the state of the application between different sessions.
- **MonitoringTableManager** class manages an internal instance of the sensor's readings.
- **MonitoringService** class handles all client/server communication.
- **CommandService** class handles the marshaling and translation from the command internal structure to a properly encoded request to be sent to the server side.
- **WalkingTab** class is a specialization of the **Tab** class responsible for the Walking Tab.
- **ChoreographyTab** class is a specialization of the **Tab** class responsible for the Choreography Tab.
- **CameraTab** class is a specialization of the **Tab** class responsible for the Camera Tab.
- **MonitorManipulateTab** class is a specialization of the **Tab** class responsible for the MonitorManipulate Tab.
- **AccelerometerTab** class is a specialization of the **Tab** class responsible for the Accelerometer Tab.
- **GyroscopeTab** class is a specialization of the **Tab** class responsible for the Gyroscope Tab.
- **MotorsTab** class is a specialization of the **Tab** class responsible for the Motors Tab.

The following images depict the interaction diagrams between these classes. Worth noting is the fact that a fair amount of communication between the different classes is achieved through the Qt signal/slot mechanism not presented here.

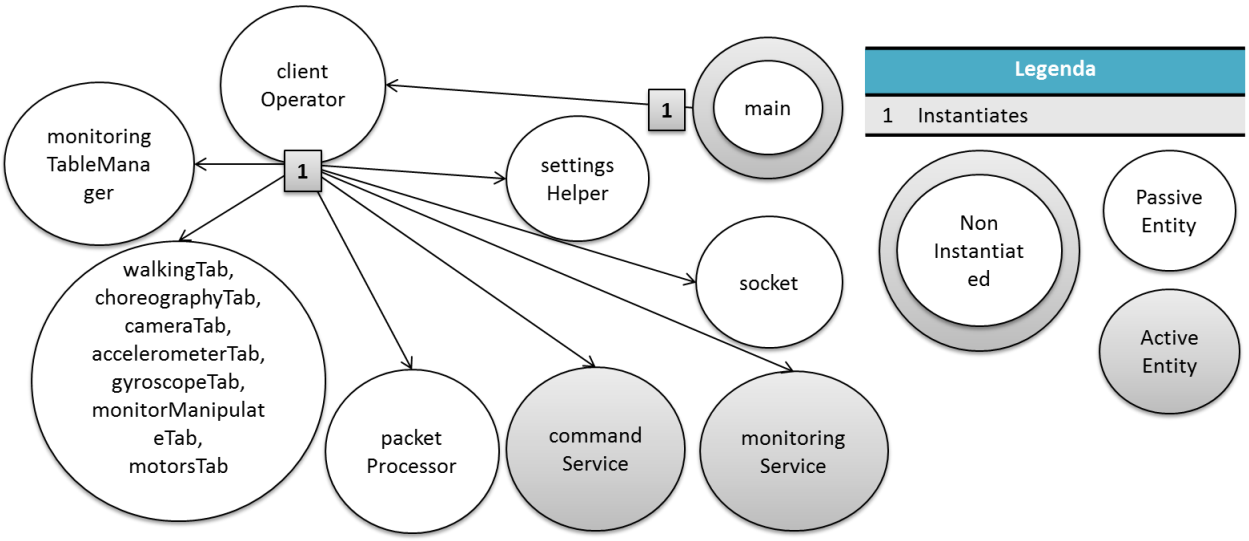
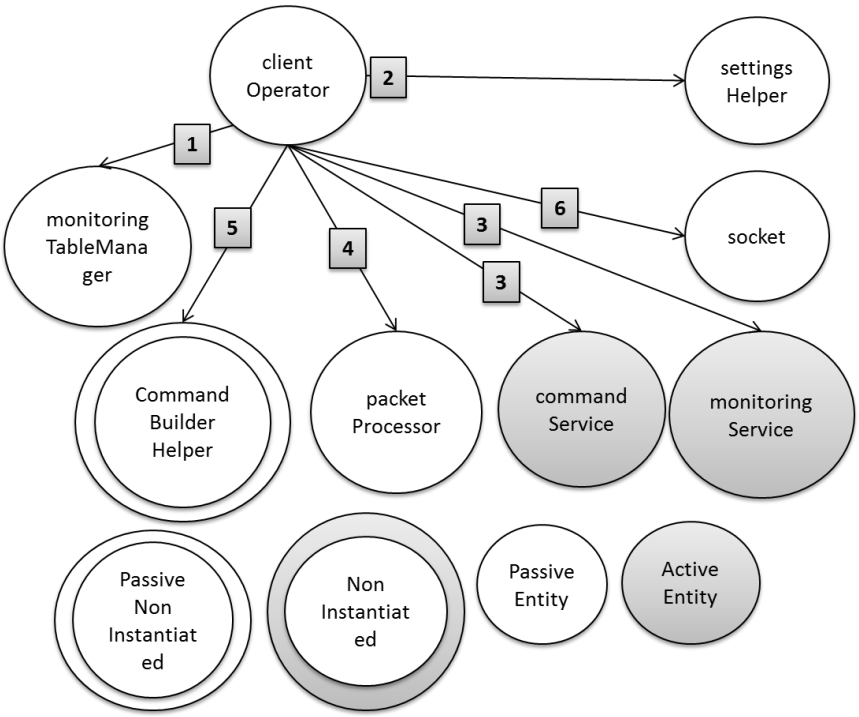
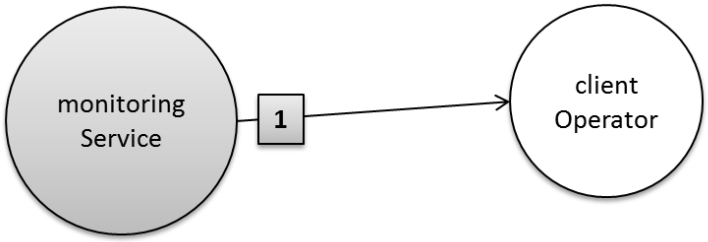


Figure 26 client main interaction diagram



Legenda	
1	getPlotSetupTableEntry, getMonitoringTable
2	restoreSettingsState, saveSettingsState, restoreConnectionSettings, saveConnectionSettings
3	Start, stop, wait
4	processCommand, processIncomingPacket
5	createCommand, createMonitoringCommand
6	sendPacket, disconnect

Figure 27 clientOperator interaction diagram



Legenda	
1	getMonitoringTable, processIncomingPacket,

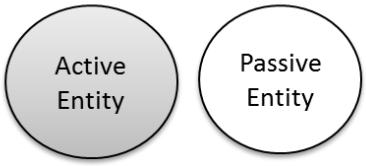


Figure 28 monitoringService interaction diagram

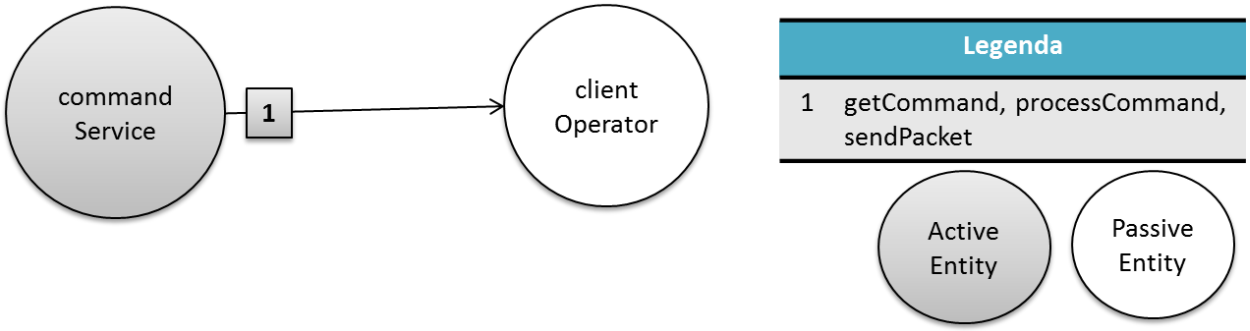


Figure 29 commandService interaction diagram

5.3 Use Cases

In this section we present the implementation of the use cases outlined during the requirements elicitation phase.

5.3.1 Connect to server

In order for the application to be of any use the Server module must be running on the robot and the Client module must be connected to the server via a socket connection. Therefore all of the following use cases presume that the client has successfully established a connection to the server, otherwise the user will be informed as shown in Figure 31. The User Interface (UI) makes it very easy to accomplish this task as depicted in Figure 30 below. To further aid the user, the connection information is saved and preserved between different sessions.

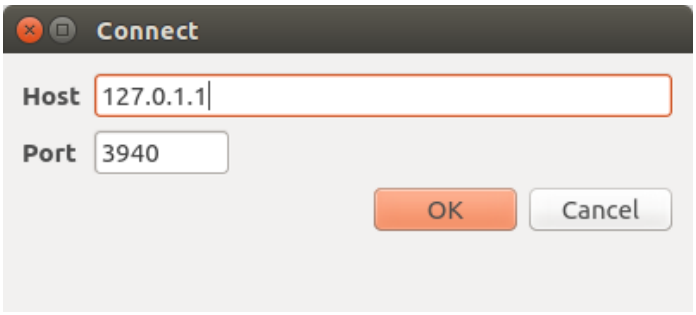


Figure 30 Connection box

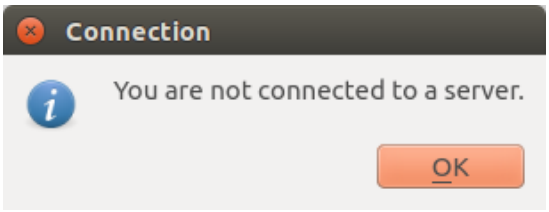


Figure 31 Client must connect to the server to perform intended action Information box

- To successfully connect to the server the user must simply insert the server's Internet Protocol (IP) address and the port number. This information will be saved and preserved between different sessions.

5.3.2 Monitor robot's sensors

The different sensors: Accelerometer, Gyroscope and Servomotors, were divided in four tabs in the UI in order to facilitate and better organize the monitorization procedure. Below we present a short explanation of each one of these tabs.

- The **Accelerometer Tab,** as shown in Figure 32, allows the user to easily start/stop monitoring the accelerometer by ticking/unticking the checkbox provided (top of the tab). The plot, also depicted on the image, shows the latest readings of the accelerometer as received from the server. The plot can also be zoomed in and out and the information shown to the user, in the form of lines on the plot, can be hidden/shown by toggling off/on the corresponding entry on the plot's legend. This is true for all plots presented on the other tabs.

- The **Gyroscope Tab,** depicted in Figure 33, is functionally identical to the Accelerometer Tab except that the readings concern the gyroscope.

- The **Motors Tab,** presented in Figure 34, serves two purposes; to control the position of the motors individually, which will not be detailed here, and to present the readings from the various parameters of the motors. Given that the robot has twenty motors a simpler way of showing this information, other than enumerating all twenty plots on the screen was chosen. Only two motors can be chosen at a time to be graphically monitored and a third plot below the other two shown in Figure 35 was introduced to enable the visualization of the selected motors side by side.

- The **Monitor Manipulate Tab,** shown in Figure 37, has a dual function: allowing for the monitorization of all the motors; providing a means to simulate via a Two Dimensional (2D) model the positioning of the motors and subsequently applying those values to the real robot. Only its monitoring aspect will be discussed here. In a sense, this tab complements the Motors Tab in that it allows the position of all the motors on the robot to be monitored at the same time.

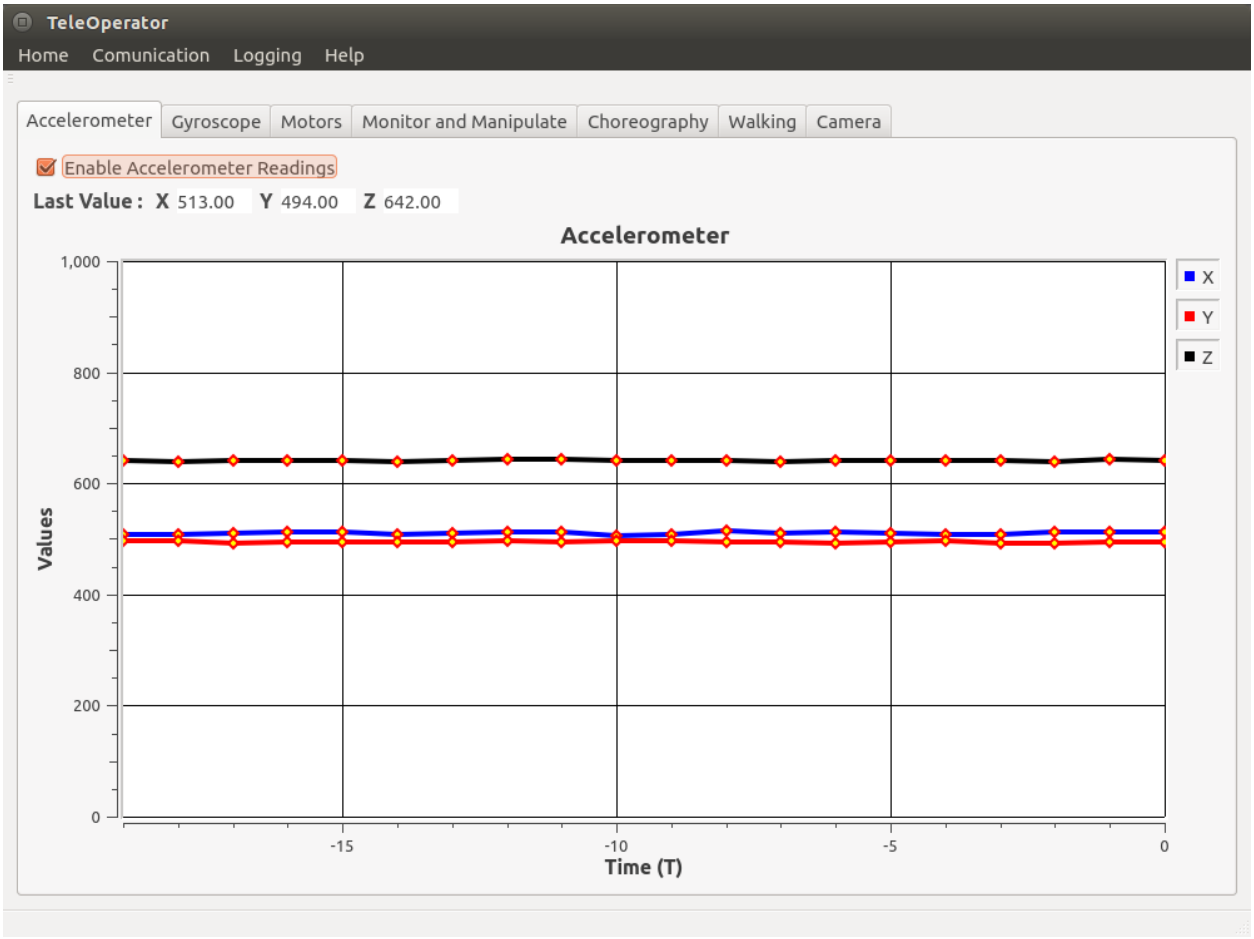


Figure 32 Accelerometer Tab

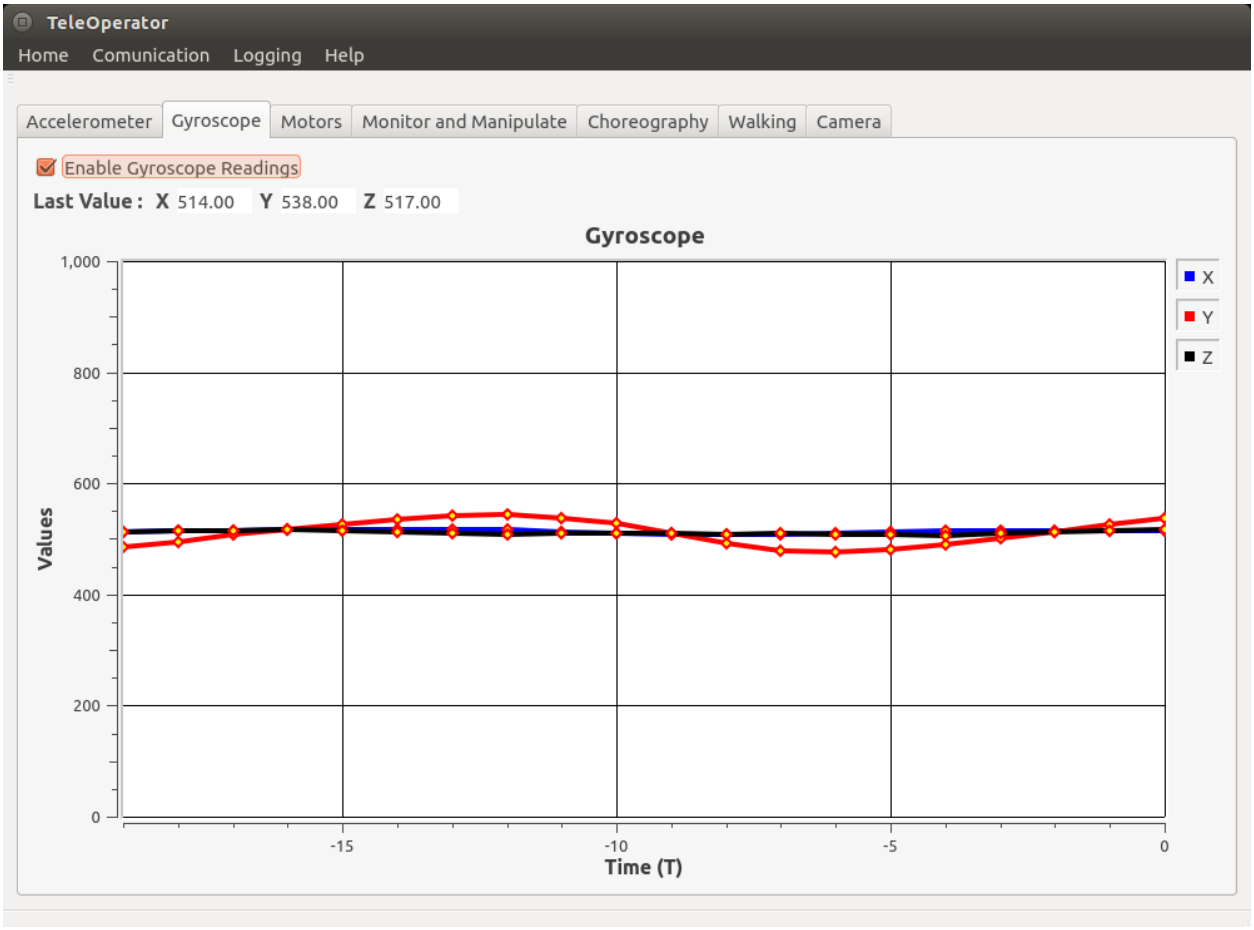


Figure 33 Gyroscope Tab

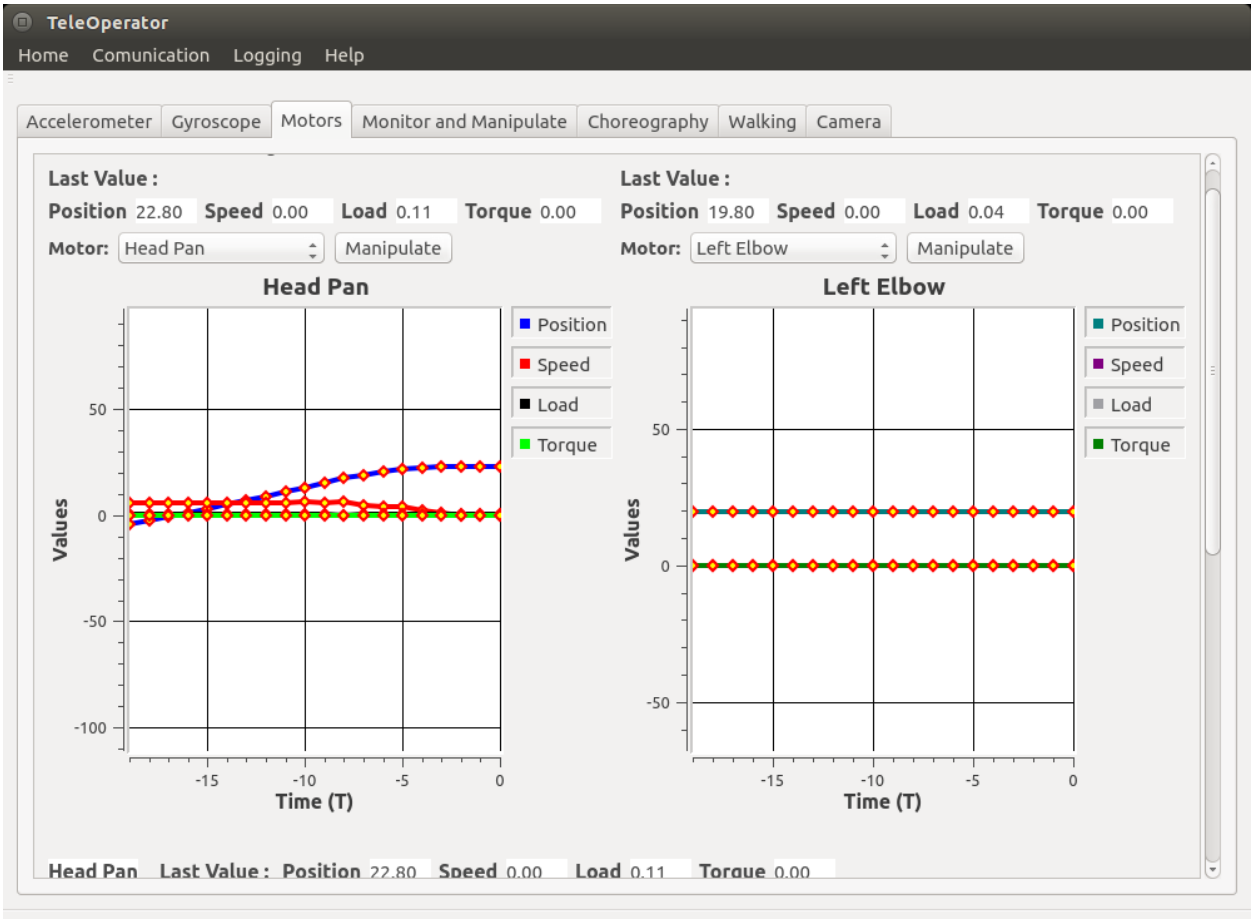


Figure 34 Motors Tab individual motor visualization

- The **Manipulate** button shown in the figure next to the selected motor, allows the user to open the motor's position manipulation box of the selected motor.

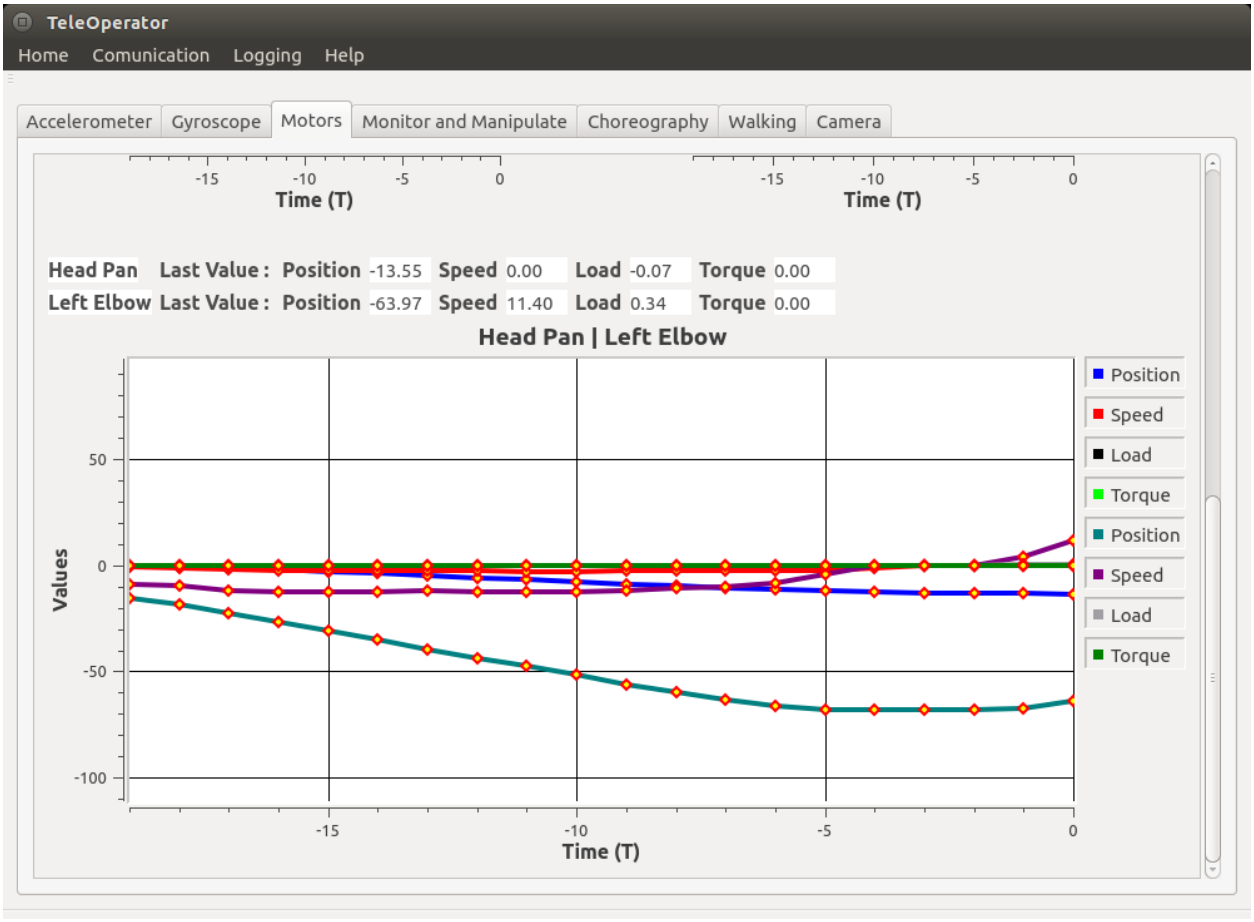


Figure 35 Motors Tab simultaneous motor visualization

It was not part of the initial requirements to control/monitor the robot through the use of a 2D representation. However, over time that became a desirable feature to have. The idea was to have a 2D skeleton representation of the robot that could be manipulated, using the principles of forward kinematics (FK), integrated with the real robot. FK is a fundamental concept in robotics but a new concept to us.

To implement this feature we had two options:

- Follow some existing works on the FK of DARwin-OP, these studies followed the Denavit–Hartenberg Parameters convention;

- Use the work of Nuno Filipe dos Reis Almeida [23] for the FC Portugal team as a reference point.

We decided to use Nuno Filipe dos Reis Almeida [23] work. We used the same convention, proposed in his work, to build an xml representation of the robot (set of body parts and joints). This representation uses an approximation of the real robot measures, taken from the ROBOTIS DARwin-OP _Kinematics charts.

The XML representation of the robot is built on the following conventions: the positive z axis is pointing up, the x positive axis is pointing right and the positive y axis is pointing forward, the torso is at the top of the body parts hierarchy.

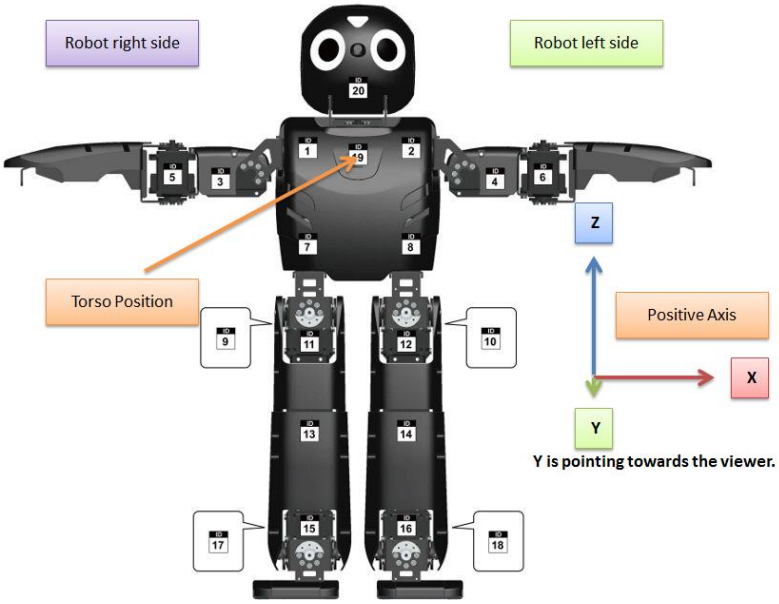


Figure 36 xml model representation convention used

We also used his code, mainly from the geometry and physics modules, as a starting point for our implementation. In the end and with some modifications we built a 2D skeleton representation of the robot, that can be manipulated, using FK, and that it is integrated with the real robot. The final result can be seen in the figure below.

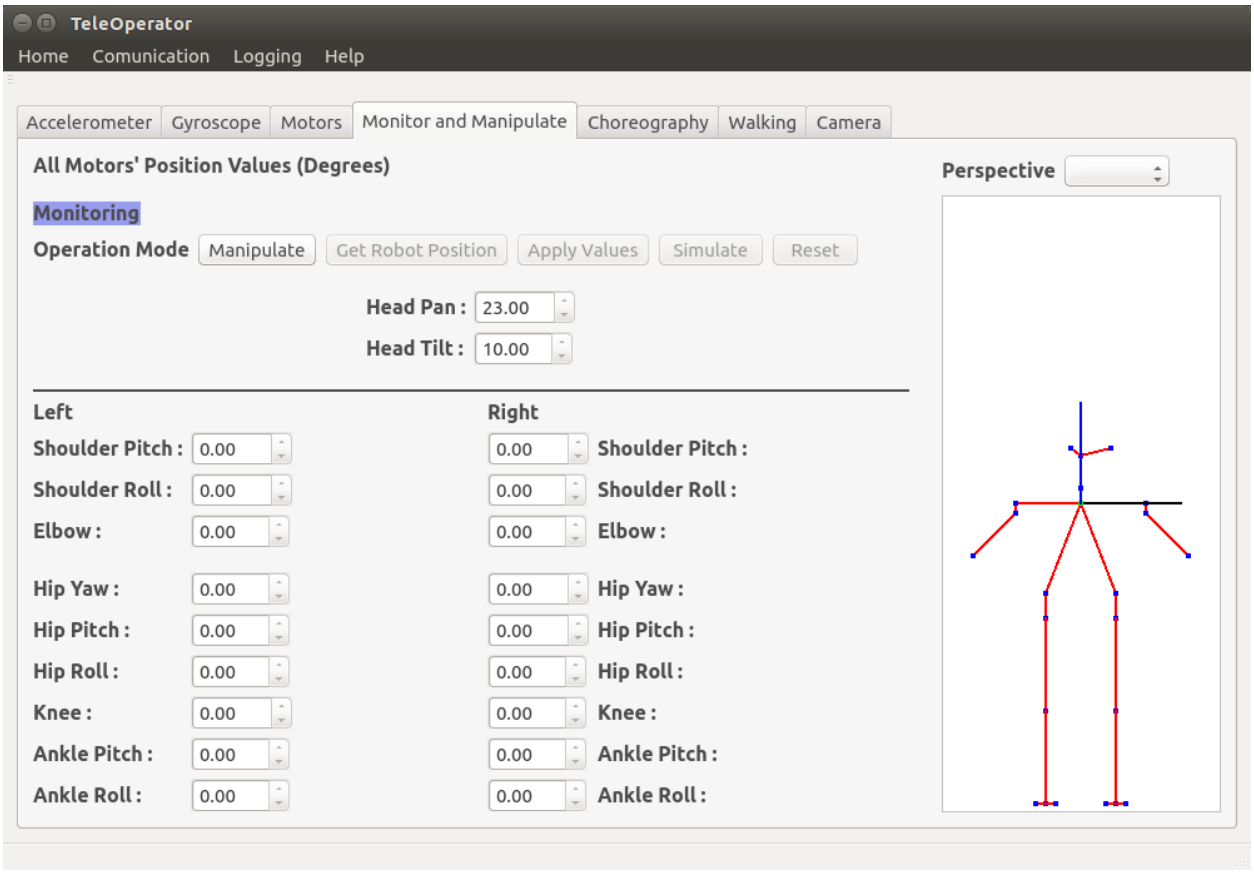


Figure 37 Monitor Manipulate Tab in monitorization mode

- The **Manipulate** button allows the user to change the tab mode to manipulation. The current mode is monitorization and the user cannot interact with the motor´s position values.

The sequence diagram of the monitoring action is depicted below Figure 38.

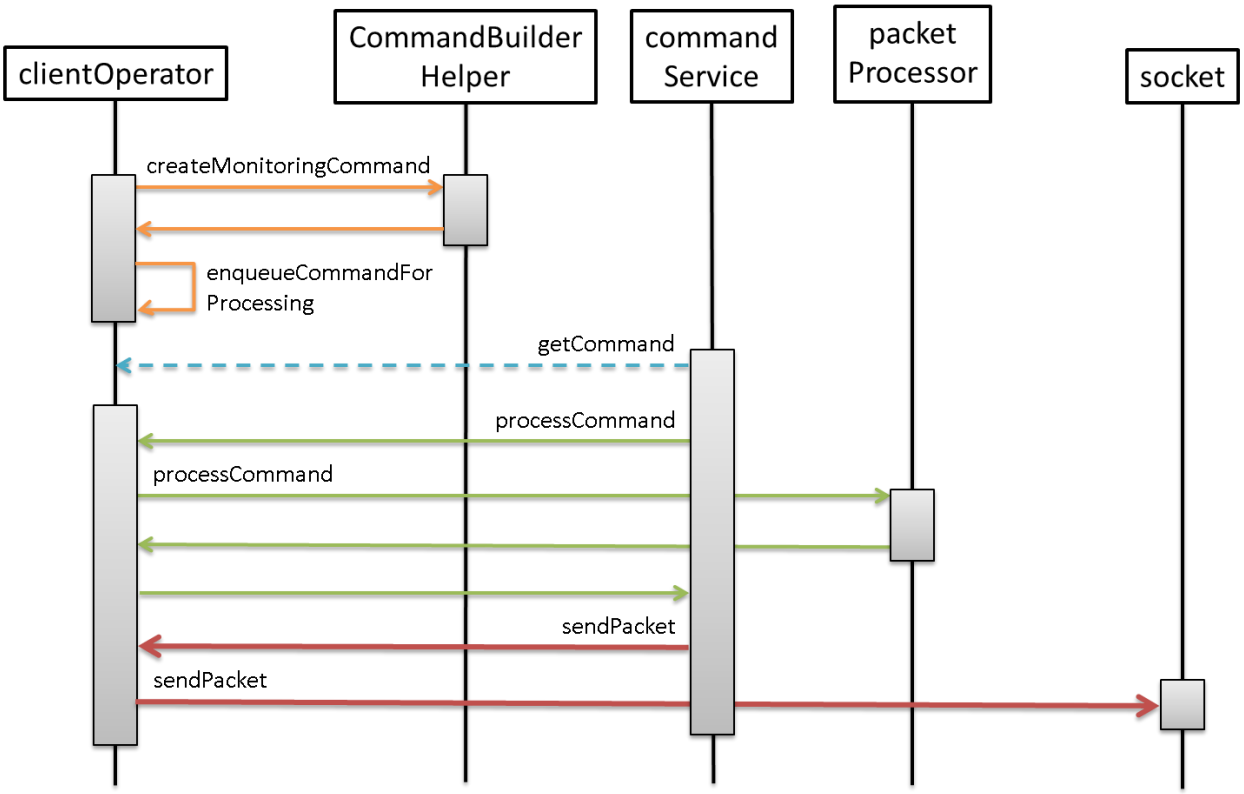


Figure 38 Client side flow of actions taken by the application after the user enables/disables monitoring a sensor

Figure 38 above is a general depiction of the way the Client module of the application handles the user actions and it's valid also for the manipulation cases.

- The user action represented by the **clientOperator** component is translated into an internal structure by a helper component, the **CommandBuilderHelper**, and stored in a queue, where it waits to be processed

- A dedicated component, the **commandService**, retrieves the command from the queue at some later unspecified time and sends it to be processed, that is translated and encoded into a properly formatted JSON message. Another dedicated component, the **packetProcessor**, is responsible for this task

- Finally the encoded message is sent to the server side by the **socket** component

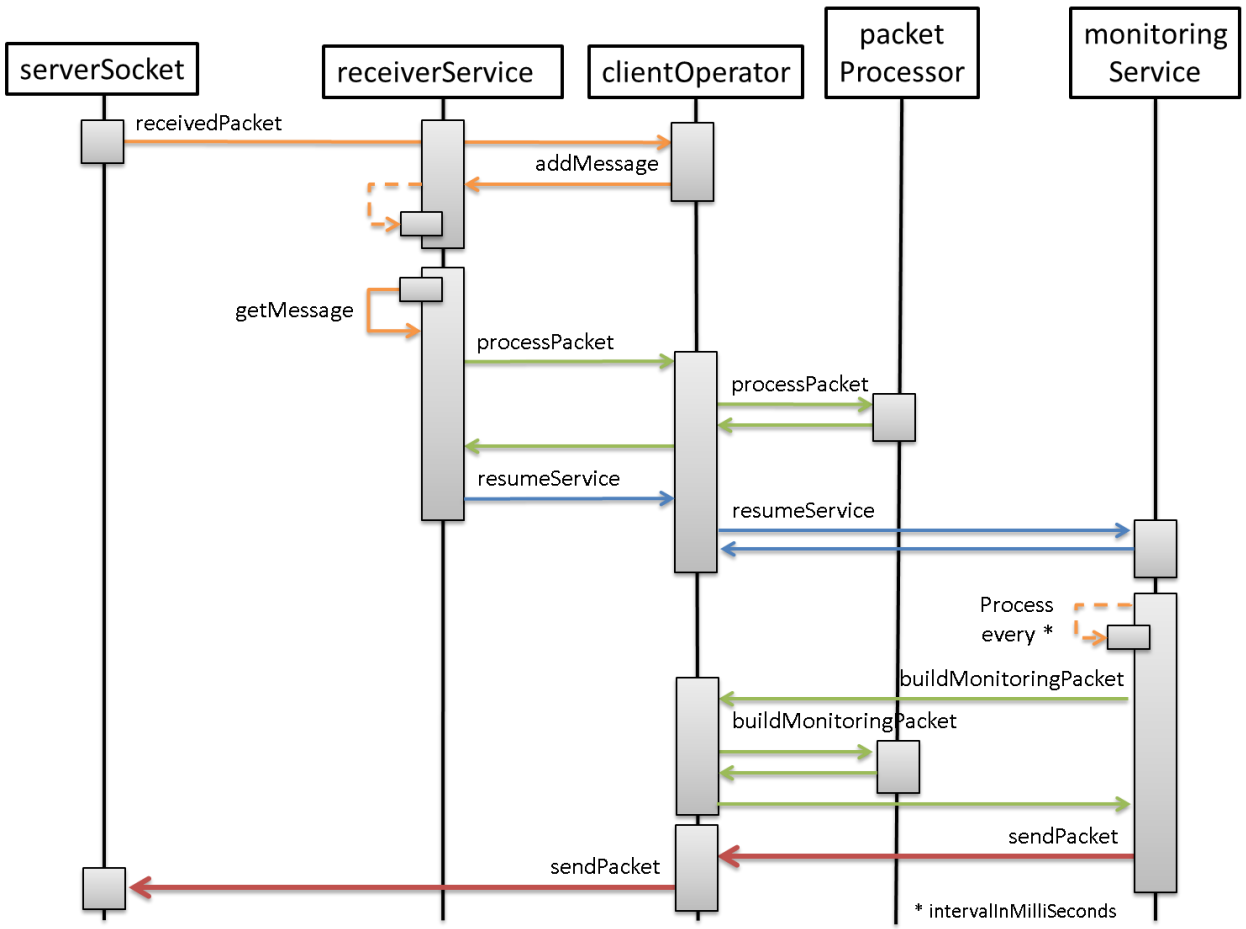


Figure 39 Server side flow of actions taken in response to a request from the client to monitor a sensor

Figure 39 outlines the flow of operations taking place on the server side upon receiving a request from the client. The first part of the process until and including the processing of the request is general to all requests received from the client, only the second part, after the request has been processed, is specific to the monitoring action. In the case shown the user has sent a request to monitor a sensor.

- The **serverSocket** component upon receiving a request forwards it to the **serverOperator** component. The **serverOperator** adds the request to an internal queue of a specialized service, the **receiverService**.

- The request is retrieved from this queue at a later time and sent to be decoded by the **packetProcessor** component.

- Once properly decoded the **receiverService** can now decide the appropriate handler for this request. In this case the **monitoringService** service is resumed.

- Once resumed the **monitoringService** service will periodically read the information from the requested sensors, send it to be encoded by the **packetProcessor** and marshaled to the client by the **serverSocket** component. This periodic task is configured via a local XML file and will occur until the service is paused.

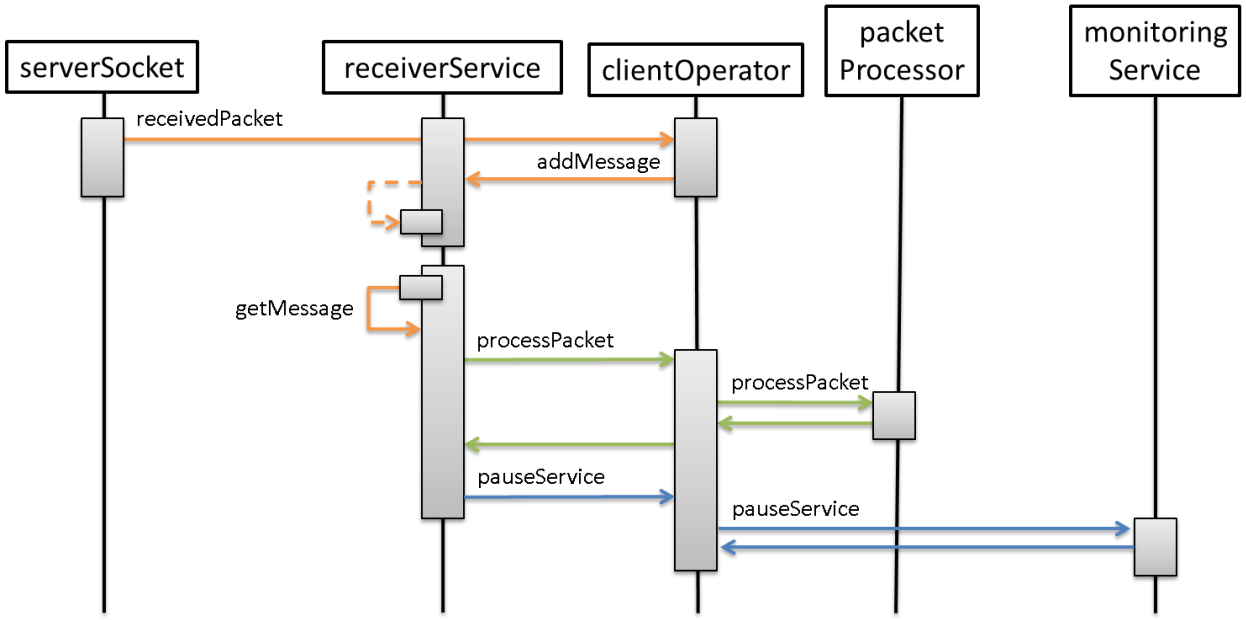


Figure 40 Server side flow of actions taken in response to a request to stop monitoring the last sensor that was still being monitored

Figure 40 shows the case where the user has chosen to stop monitoring a sensor. In this particular case the server has detected that this was the last sensor still being monitored and therefore pauses the **monitoringService**. No reply is sent back to the client.

5.3.3 Access robot's camera

The camera was given a separate tab, the **Camera Tab**, shown in Figure 41 that works similarly to the other monitoring tabs. The image shown corresponds to what the robot is seeing. The flow of actions taken by the application upon user interaction is as previously depicted.

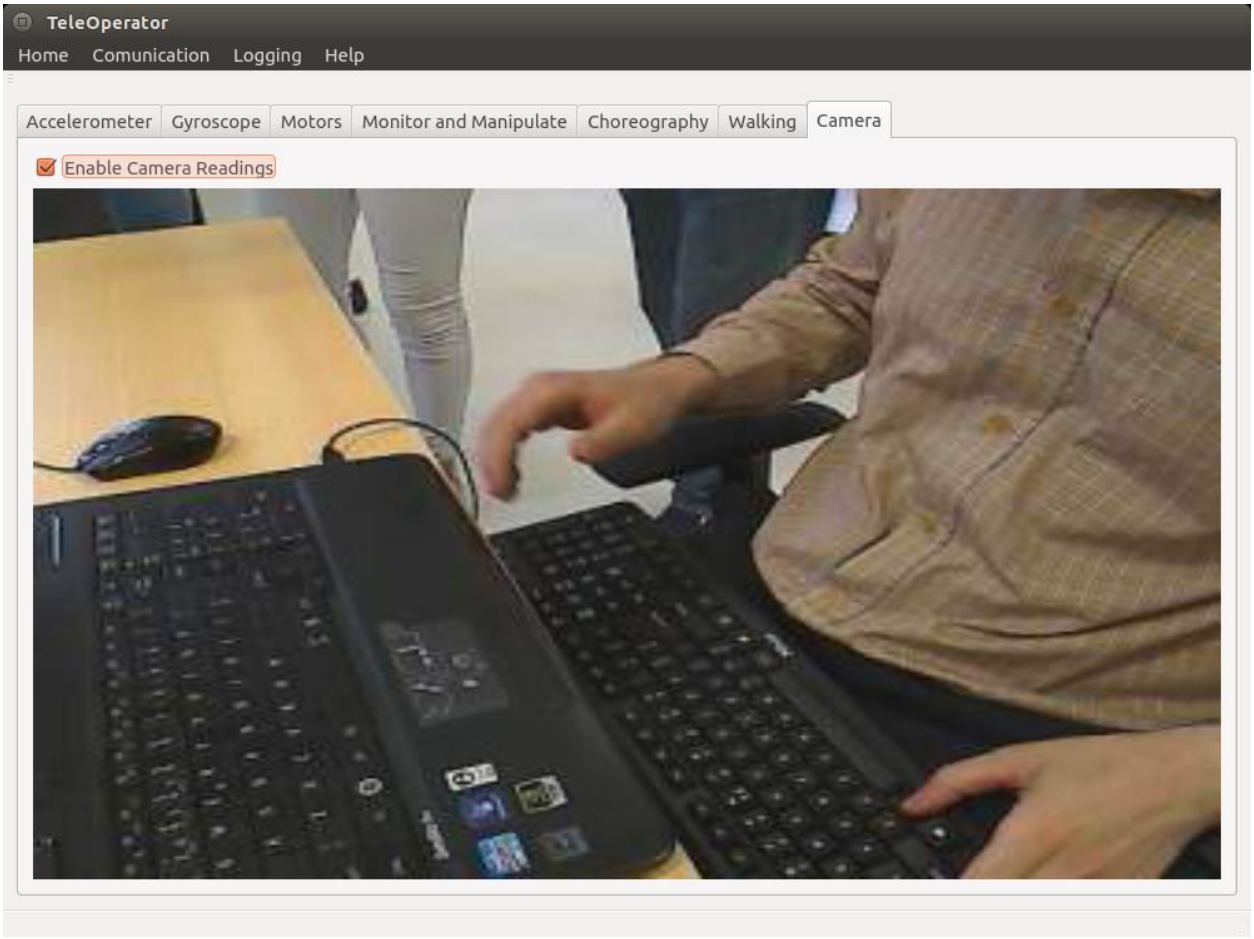


Figure 41 Camera Tab showing an image og what the robot is seeing

5.3.4 Control robot's movements and behavior

The control of the robot was divided into three different groups: manipulating the position of the motors individually or all at the same time; walking in all directions and playing with a ball; performing the pre-configured robot behaviors in a sequential way in any order and combination. To accommodate for this logical separation, four tabs were introduced in the UI and are described next.

- The **Motors Tab** depicted in Figure 35 has previously been addressed but now we detail its manipulation capabilities. In this tab the user can change the position of a single motor at a time, the selected motor in this case, using the Manipulate button provided. The manipulation box is shown in Figure 42.

- The **Monitor Manipulate Tab** as also been previously discussed and is presented in Figure 43 this time in manipulation mode. In this tab the user manipulates the position of all motors at the same time having the possibility to simulate all the wanted changes first prior to real execution.

- The **Walking Tab** in Figure 44 allows the user to make the robot walk in all directions and stop this action at any time. This tab allows also the execution of the pre-configured behavior where the robot follows and kicks a red ball.

- The **Choreography Tab** shown in Figure 45 contains a list of all the pre-configured behaviors of the robot. The user can create a list of these behaviors in any order and combination and make the robot play them in sequence. It is also possible to change the forehead and the eyes led's color.

Figure 42 Motor's position manipulation box

- The position of the motor can be changed by dragging the slider or by inputting the value directly in the numeric box. Upon any of these actions a message will be send automatically to the server ordering the motor to change its position accordingly.

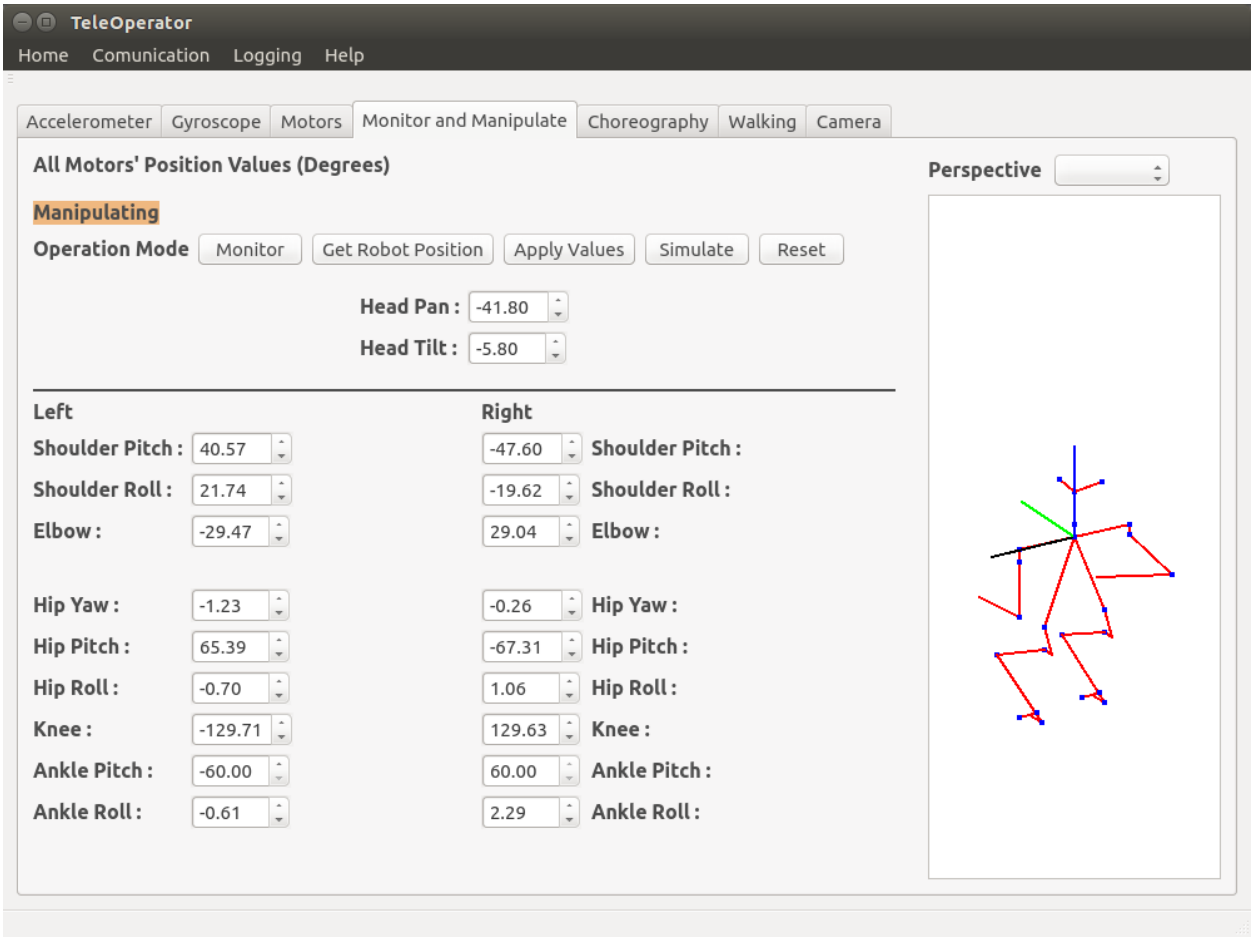


Figure 43 Monitor Manipulate Tab in manipulation mode

- The **Monitor** button allows the tab mode to be changed to monitorization.
- The **Get Robot Position** button allows the user to get the current positions of all the motors without the need to explicitly start monitorization.
- The **Apply Values** button sends the current motors' position values to the server. The real robot motors will move to the specified positions.
- The **Simulate** button will apply the current motor's position values to the 2D model acting as a preview to the real changes effects on the real robot.
- The **Reset** button resets all motor's position values to zero (the robot zero position). These changes apply to the 2D simulated model only.

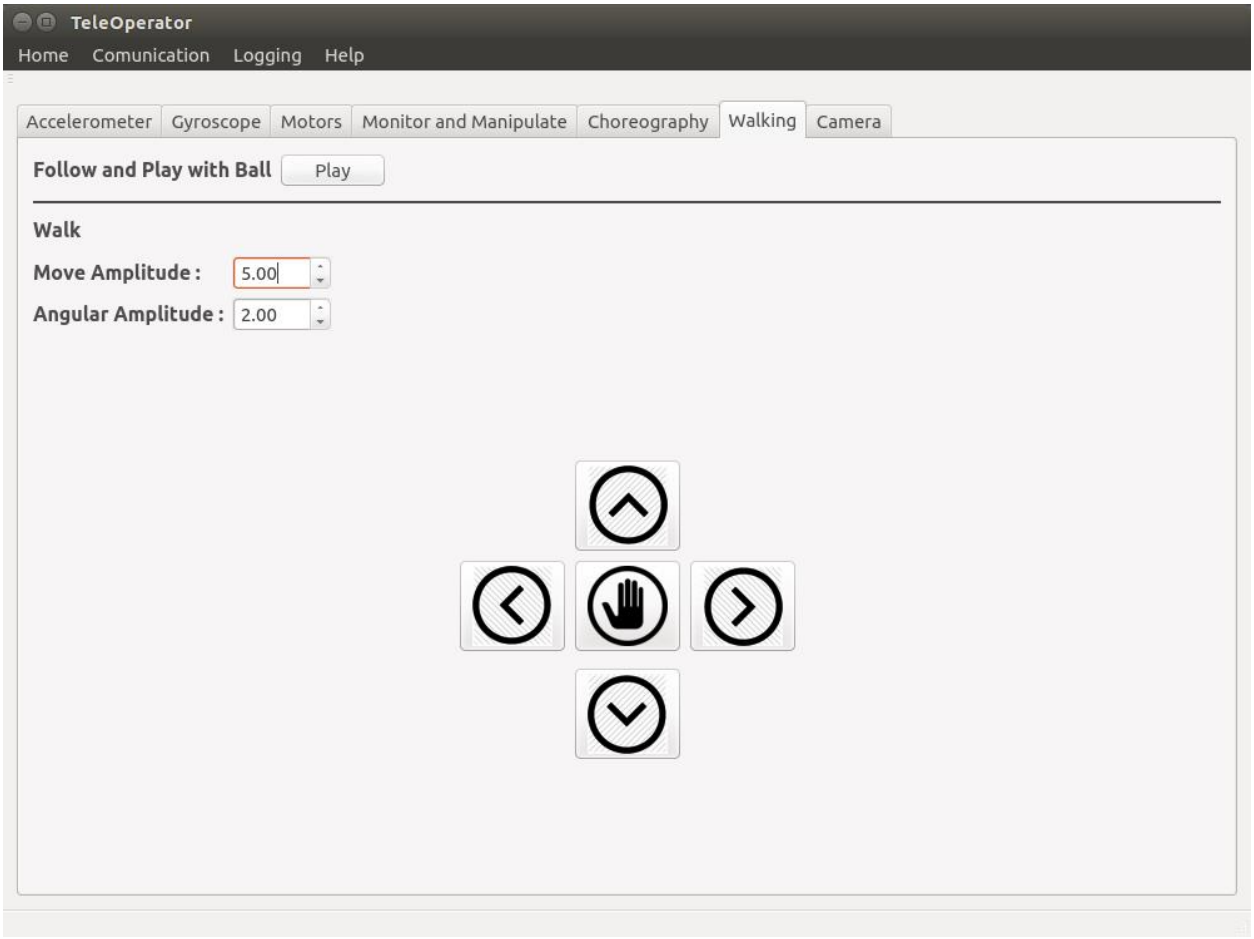


Figure 44 Walking Tab

- The **Play** button can be used to initiate the pre-configured behavior of the robot to follow and kick a red ball. This button can be used interchangeably to also stop this action at any time once started.

- The remaining buttons shown refer to the ability to order the robot to walk in all four directions, the buttons with the arrow images, and stop the walking behavior, the button with the hand.

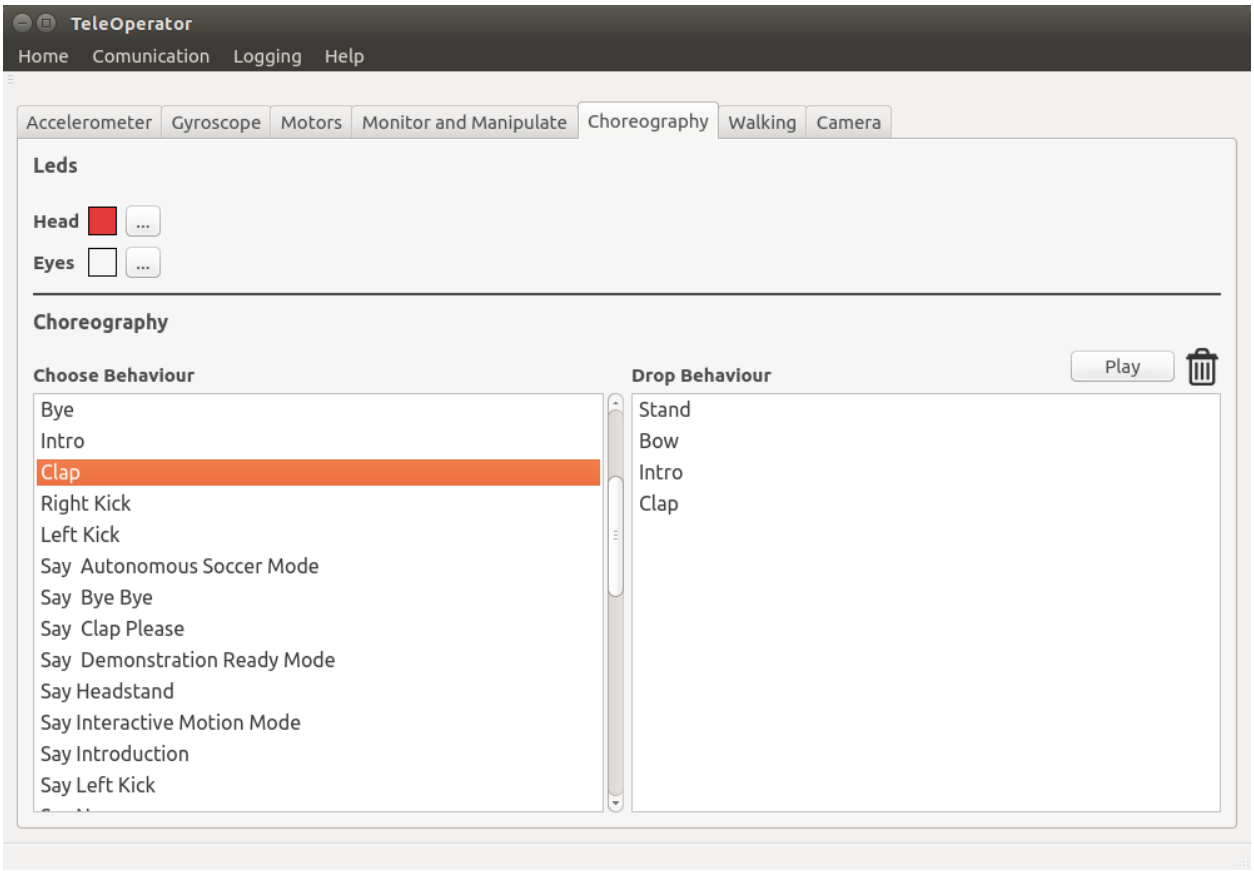


Figure 45 Choreography Tab

- The **ellipsis** buttons allow the user to choose a color for the robot's eyes and forehead leds. Figure 46 illustrates the color picker that will be presented to the user upon clicking the aforementioned buttons.

- Two lists are presented at the bottom of the tab. The left list contains all the preconfigured behaviors of the robot (animations and sounds). The user can drag and drop these behaviors in any order into the right list. The behaviors present in this list will be performed sequentially by the robot. When applicable the user as the option to configure the desired behavior. For simplicity only the walking behavior can be parameterized as depicted in Figure 47.

- The **Play** button can be used to start/stop the performance comprised by the list of selected behaviors.

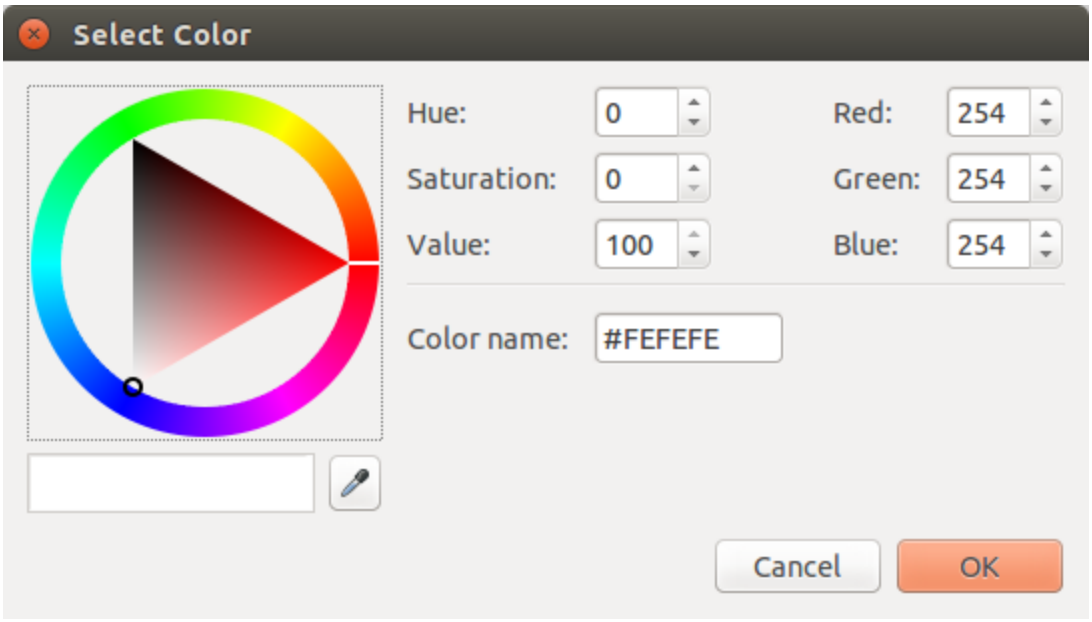


Figure 46 Color Picker for the Leds colors

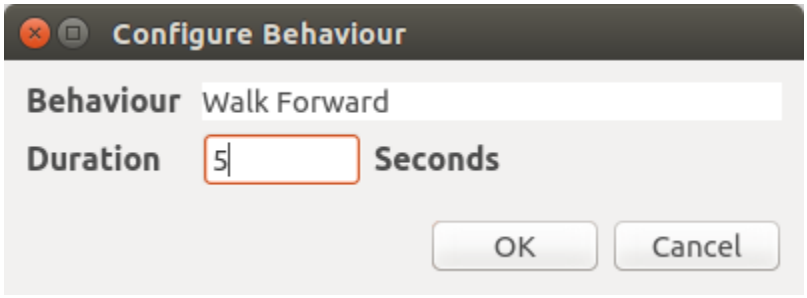


Figure 47 Walk behavior configuration dialog

Figure 48 presents the server side processing flow to handle a request to change a motor's position. The client side processing taking place is similar to that already depicted in Figure 40. A new component, **RobotisMonitor** is introduced here. This is the wrapper developed for the ROBOTIS framework and will be discussed later.

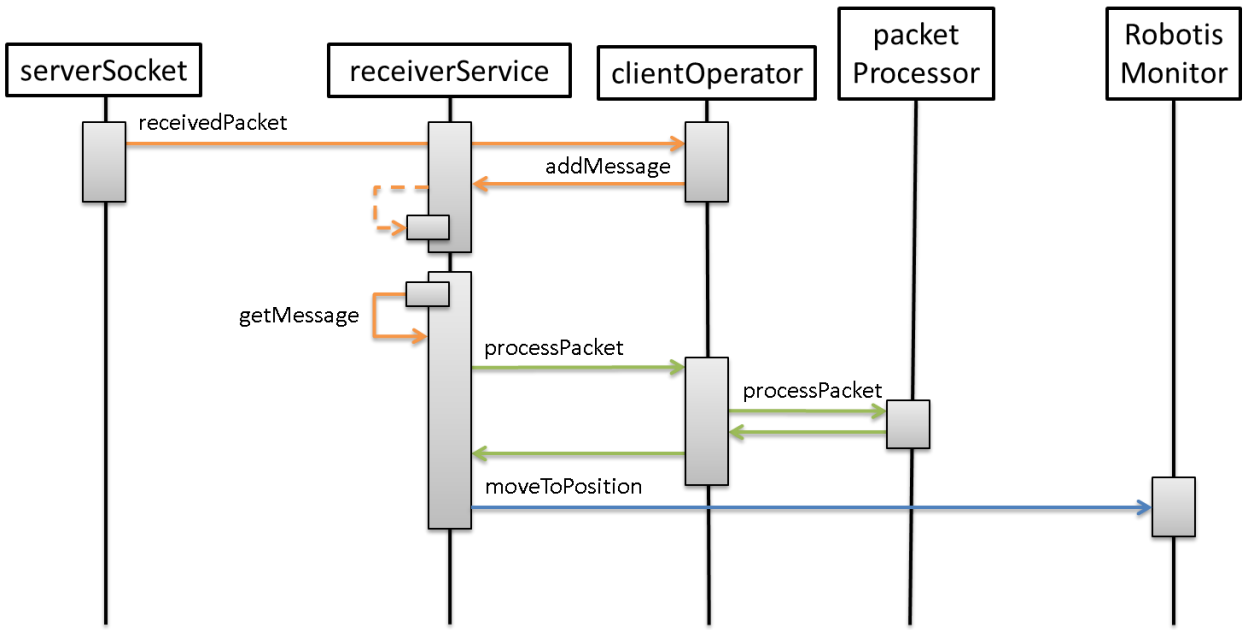


Figure 48 Server side processing flow to handle a request from the client to change a motor's position

5.3.5 Log monitoring values

The readings obtained from the accelerometer, gyroscope and all motors' positions are logged in a local file in plain text format so that the collected data can be later analyzed.

5.4 RobotisMonitor

The RobotisMonitor is the library we implemented as a wrapper to the ROBOTIS framework intended to act as an interface between the control and monitoring application and the real robot.

The RobotisMonitor wrapper was implemented in such a way that different frameworks, communicating directly with the robot, could be wrapped in a single interface. New frameworks could then be added or changed/enhanced without breaking the wrapper clients.

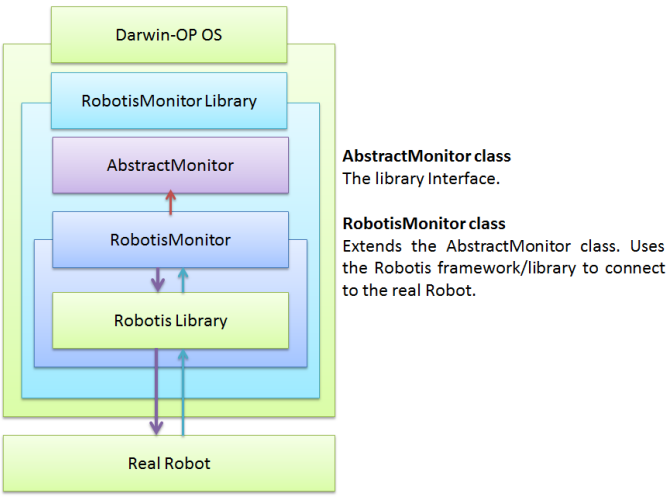


Figure 49 RobotisMonitor structure overview

The RobotisMonitor library has an abstract class, AbstractMonitor, that defines it´s interface to the world. This way, new frameworks can be added just by creating a new class that extends the AbstractMonitor class. In our project we only used the Robotis framework but we think that it would be possible to add the framework developed by the Pennalizers Team [22] to our library, probably with some extra work.

The interface provided by the RobotisMonitor library is the following:

bool connect()
Connects to the real robot.

getMotorValues(SensorReadings *sReadings)
Gets the motor information for all the robot motors.

bool changeLedColor(**int** genericLedID, **int** red, **int** green, **int** blue)
Changes the eyes and forehead leds color.

int getCameraImage(**unsigned char** **imageData, **int** imageQuality)
Gets the image from the robot camera in jpg format.

bool moveToPosition(**int** motorID, **double** goalPositionInAngles, **double** speed)
Moves a motor to a position.

bool playChoreography(**int** *listOfAnimations)
Plyas a choreography, that is, a list of animations in sequence.

bool walk(**double** x, **double** a, **double** y, **bool** stop)
Used to make the real robot walk.

bool executeAnimation(**int** animationCode)
Executes a single animation.

bool findAndPlayWithBall(**bool** stop)
Used to make the robot find and kick the red ball.

bool speak(Speech dialogue)
Used to play a sound.

bool moveJoints(std::map<**int**, JointInfo> *jointsGoalPosition)
Move a set of joints to the given positions.

5.5 Communication Protocol

A specific protocol was implemented to allow communication between the Server and Client modules. The encoding format chosen was JSON due to its lightweight and human readable characteristics. The general format of the message is shown in Figure 50.

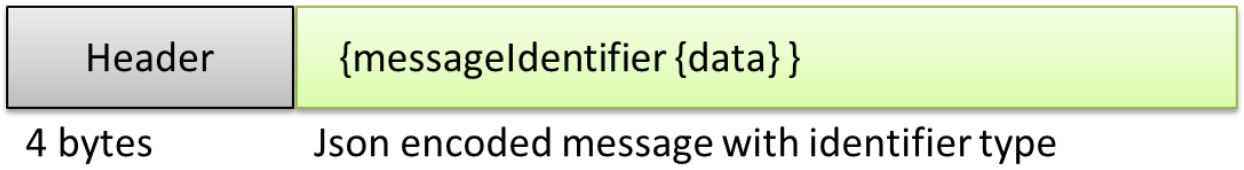


Figure 50 Message format

- The **Header** of the message is a 4 bytes integer that represents the total size of the message in bytes.
- The body of the message is encoded in JSON. Each message type has a specific message identifier, detailed next.

In order to differentiate the different types of client/server interaction, 6 groups of messages were created. Each group was therefore assigned a proper identifier. These identifiers are explained below.

- The **monitor** identifier identifies all messages concerning monitorization. In this case the enable/disable monitoring a sensor action.
- The **operate** identifier signals all messages intended to change the motor's position values.
- The **led** identifier is used to tag the messages sent to change the color of the forehead and eyes leds.
- The **orchestrate** identifier identifies the messages used to send the list of pre-configured robot behaviors to be performed sequentially by the robot.
- The **walk** identifier signals the messages that instruct the robot to walk in all directions.
- The **playball** identifier tags the message used to order the robot to perform the pre-configured robot behavior Play with Ball.

More information on the structure of each message type and actual message examples are presented in Appendix A.

5.6 Internationalization

The user interface of the client module was designed to support many different languages. For simplicity, only the Portuguese and English languages were bundled with the application by default. However any other language can be easily integrated.

Chapter 6

6. Integration with ROS

The third milestone objective was to port the monitoring and control application, developed in the previous milestone, to ROS. An alternative, that was considered later on, was to use ROS to communicate directly with the robot motors using already existing ROS packages. In this scenario the ROBOTIS framework would not be used.

The integration with ROS revealed two major problems: Darwin´s OS, Ubuntu 9.10, is outdated and does not support any version of ROS and there is not enough free disk space on Darwin´s flash disk to install ROS.

Our best options were:

- Free some disk space on Darwin´s flash disk and install ROS. Even if it was possible to free enough disk space to install ROS, the installation process requires extra work and it is not guaranteed that ROS will work on Ubuntu 9.10. After some analysis, we concluded that it was not possible to free the needed disk space for the installation of ROS.

- Upgrade Darwin´s OS to Ubuntu 12.04.5 LTS, without a graphical interface in order to save disk space. Upgrading Darwin´s OS is not well documented and involves some risk.

- Use an external USB disk to simulate and test the upgrade to Ubuntu 12.04.5 LTS and the integration with ROS.

It was decided that: first we would simulate the upgrade and the installation of ROS by using virtual machines. That way it would be possible to understand, with some degree of confidence, if the process could be done successfully on the real robot. Then, we would use an external USB disk with Ubuntu 12.04.5 LTS with ROS Hydro installed to boot Darwin. As part of this process we would also compile and test the ROBOTIS framework and samples and the monitoring application that we developed previously. This included installing all the necessary dependencies.

Finally, if the conclusion was that the process could be performed on the real robot, with some degree of confidence, then we would upgrade the real robot operating system, install ROS Hydro and start working with ROS according to the third milestone objectives.

Integrating DARwin-OP with ROS could be done in two different ways. We could use our wrapper library for the ROBOTIS framework or we could use ROS directly on the real robot. Because of time management and in order to reuse what we had already made, we decided to use our wrapper library to integrate DARwin-OP with ROS. We would also use 3 ROS packages made available at [27] to integrate Darwin with ROS.

The way we thought the integration with ROS using our wrapper library actually revealed another problem, based on the fact that the ROBOTIS framework, by construction, does not allow two users to connect to the sub-controller at the same time.

It was also decided that we would use ROS Hydro. At the time there was already another version of ROS available, ROS Indigo, but we decided in favor of Hydro, because of the OS version we would be using, 12.04.5 LTS, and because the available ROS packages that we would be using to integrate Darwin with ROS could eventually cause problems on Indigo.

We used our wrapper library in two ways: directly from the ROS nodes (a publisher and a subscriber) and with the ROS nodes communicating with the Server part of our application, using sockets and TCP/IP. A high view of both approaches can be seen in the next two pictures.

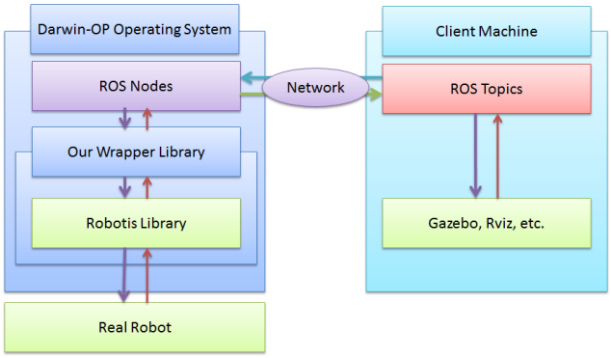


Figure 51 Overview of the structure using our wrapper library, directly from the ROS nodes, to integrate DARwin-OP with ROS

The ROS nodes running in DARwin-OP OS (a publisher and a subscriber), seen in the previous picture, can publish to or subscribe the ROS topics for the robot joint positions. Only one node can run at a time.

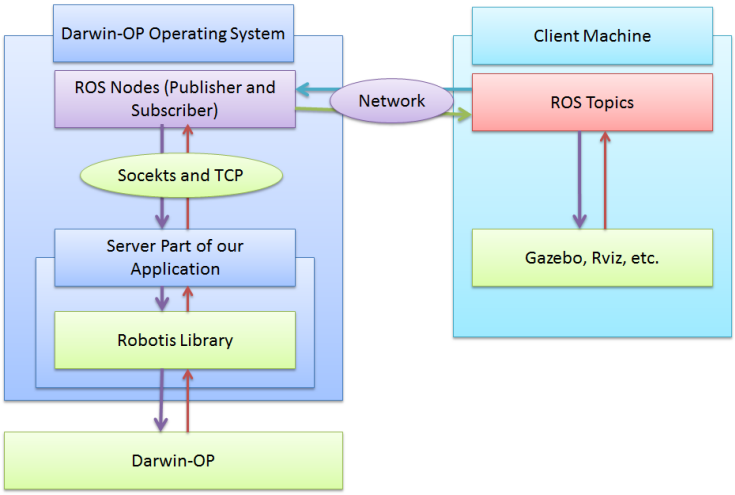


Figure 52 Overview of the structure using our wrapper library, by using the Server, to integrate DARwin-OP with ROS

This approach revealed some performance problems. However, in this scenario we can have clients, based on different technologies and or architectures, using our wrapper library to communicate with the robot.

6.1 Using the Virtual Machines

We used two virtual machines to simulate the real robot´s OS upgrade and the installation and use of ROS. One acted as a client machine, a machine that would be used by someone to communicate with the robot using ROS and the other as the real robot.

We followed the tutorial at [27] with minor changes to install and test the use of ROS. This tutorial also provides the 3 packages that we used to integrate the DARwin-OP robot with ROS.

At this stage we were able to compile the ROBOTIS framework and some samples, our wrapper library and the server part of our application. Because in this setup we do not have access to the real robot, we could not perform any tests. We did however simulate the communication of the two machines using ROS, by creating a ROS package with two nodes, a publisher and a subscriber, on the robot machine. With the client machine as the ROS master and the use of the 3 packages mentioned earlier, it was possible to get the joint position information of the simulated robot and to publish hardcoded joint information to the simulated robot, from the robot machine. Gazebo was used to see the simulated robot assuming the joint position information published to ROS.

To create our ROS package we had to install an additional ROS package, sensor_msgs, by using the following command: sudo apt-get install ros-hydro-common-msgs.

It is important to mention that the use of gazebo on virtual machines is not recommended. With a virtual box machine Gazebo was painfully slow and impractical to use. We had to use a vmware machine to run the client machine. Here, Gazebo performance is acceptable but there may be problems with OpenGL from time to time. Also, not all graphic cards allow us to run Gazebo on a virtual machine. We were able to run it using an NVidia Graphics Card. We also tried to use Rviz instead of Gazebo but Rviz issued some errors when loading the URDF model of the robot, provided with the 3 ROS packages at [27], so the use of Gazebo was preferred.

The setup was as follows:

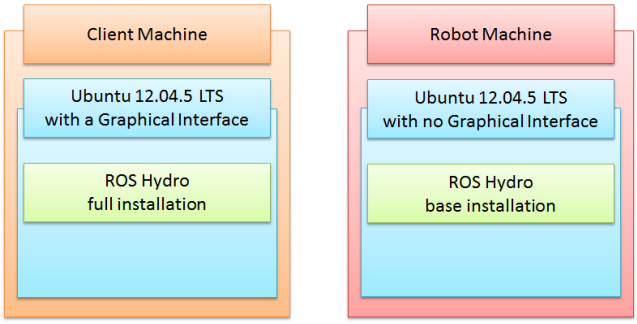


Figure 53 Virtual machines setup. The client machine is on a vmware machine and the robot machine is on a virtual box machine

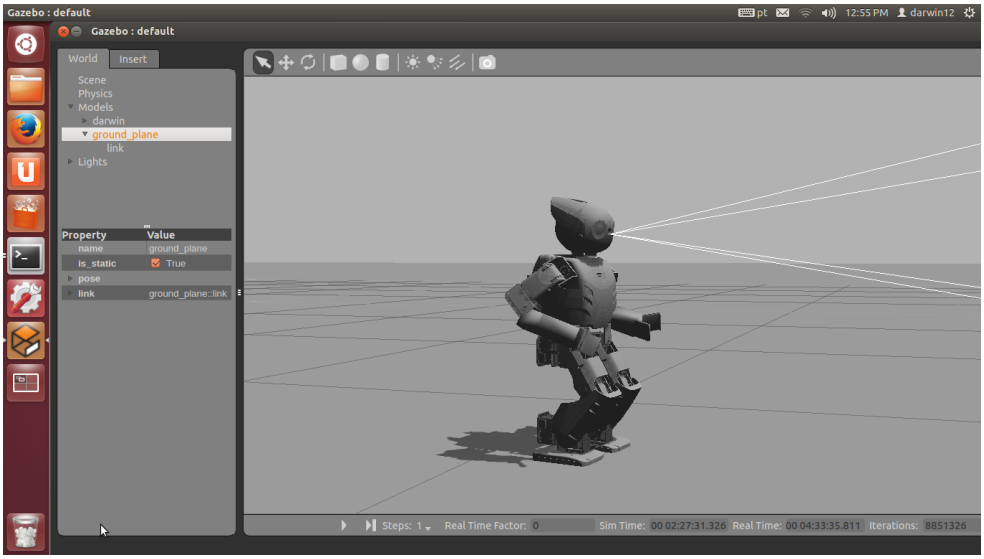


Figure 54 Client machine, screenshot. DARwin-OP on Gazebo

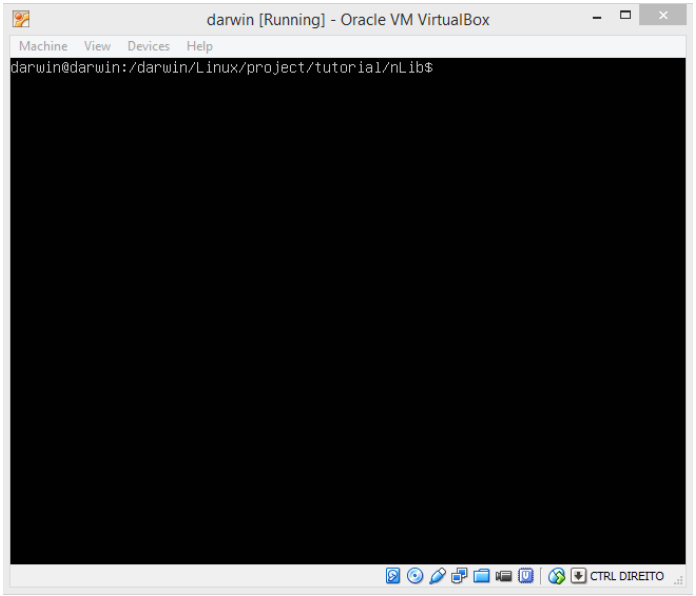


Figure 55 Machine simulating the real robot

6.2 Using a USB External Disk

We were able to install Ubuntu 12.04.5 on an external USD disk, with persistence capabilities. That enabled us to install: all the necessary dependencies, the ROBOTIS SDK and samples, the CM-730 firmware, our application (server module), the camera drivers and ROS Hydro. With this setup, it was possible to test the ROBOTIS SDK samples and our application, since we had physical access to the real robot.

We experienced some communication problems between the robot´s main and sub controllers. We think these problems are caused by the fact that we are running the robot operating system from an external USB disk.

At this stage we tested our ROS package with the real robot. During these tests we realized that: the real robot and the 3D model of the robot (URDF model) had different zero positions (position of the robot where all joint positions are 0) and the ROS packages, used to simulate DARwin-OP on Gazebo and integrate it with ROS, wore expecting joint positions given in radians whereas our wrapper library was expecting degrees.

To solve the first problem we had to change the URDF model, with the help of Professor Nuno Lau, and we had to compensate the zero position differences in the arms (between the real robot and the URDF model) by adding and subtracting angles.

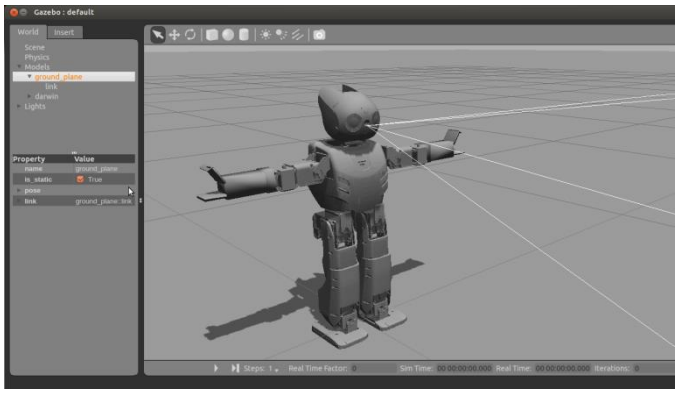


Figure 56 DARwin-OP zero position in Gazebo

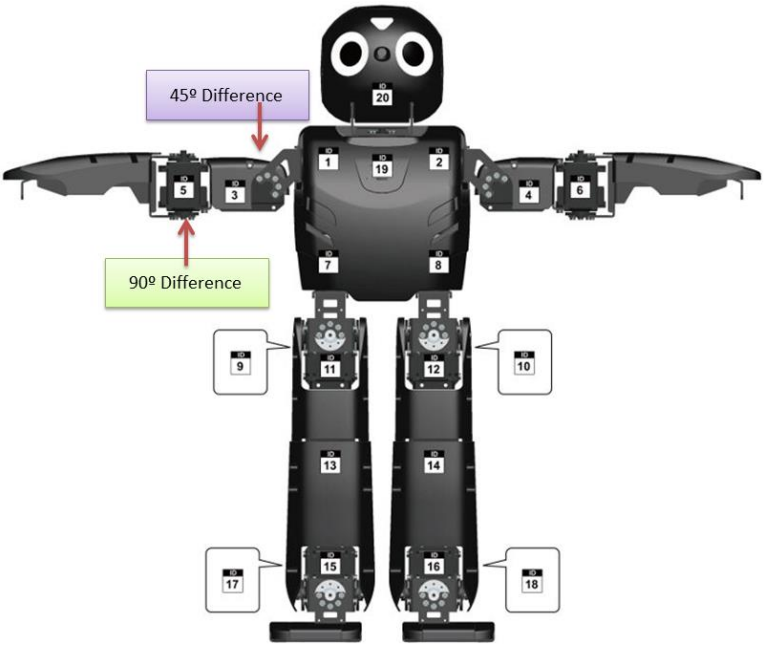


Figure 57 DARwin-OP angle compensation in arm joints relative to the real robot zero position: joints id 3 and 4 (45 degrees), joints id 5 and 6 (90 degrees)

The changes in the URDF model, the darwin.urdf file on darwin_description (the ROS package providing the mesh files and the URDF model of the DARwin-OP robot for use with the real robot or with Gazebo) were as follows:

```
<joint name="j_high_arm_r" type="revolute">
  <parent link="MP_SHOULDER_R"/>
  <child link="MP_ARM_HIGH_R"/>
  <origin xyz="0.016 0 0.02165" rpy="-1.5707963 1.5707963 0" />
  <axis xyz="0 0 -1" />
  <limit effort="2.8" velocity="5.648668" lower="-2.6179939" upper="2.6179939" />
 </joint>
```

<axis xyz="0 0 1" />, in the original version of the URDF model, was changed to <axis xyz="0 0 -1" />, as shown above.

At this stage we created additional ROS packages that could use our wrapper library: directly or by communicating with the Server to get to the real robot, as described earlier.

In the end we were able to put the simulated robot to walk in Gazebo, using a python script provided with the DARwin-OP simulating packages, and the real robot executing the same movements by subscribing to the ROS topic with the joint position information and we also published the real robot joint information to ROS (the result could be seen in Gazebo). However, in the current setup we are not able to move the real robot using our Client application while publishing it´s joint position data to ROS.

6.3 Using the Real Robot

We did not upgrade the real robot OS, therefore we did not install ROS in the real robot.

Chapter 7

7. Conclusion and Future Work

7.1 Summary

This report started out by presenting the DARwin-OP platform in terms of its hardware and software architecture. The report then proceeded on detailing the results on the research made around the topic of this project. Chapter 3 was totally devoted to this task. Throughout the referred chapter, the latest and most exciting projects and technologies related to the DARwin-OP platform were subject to thorough examination and careful thought concerning their advantages and disadvantages. It was then concluded that given the specific circumstances; the timeline to conclude the project and the robot setup, the projects studied were not suitable and the control and monitoring application should be implemented from scratch.

In chapter 4 the system requirements and overall architecture were specified after which followed the implementation of the proposed application. Two modules were implemented; the Server module to be deployed on the robot and the Client module to be installed on the client machine. A communication protocol between these modules was also implemented using JSON encoding format. The full details of the implementation phase were outlined in chapter 5.

With the control and monitoring application implemented it was time to test the possibility to fully integrate the DARwin-OP platform and the work already done to ROS. Chapter 6 is devoted to this issue.

7.2 Main Results

Given the wide spectrum comprised by this field of research and the paramount of possibilities, it is clear that our application is far from complete. Analyzing the initial goals of the project however we feel that most of the proposed tasks were achieved successfully. As specified, the client application is able to successfully monitor the robot and control its position and behaviors. These behaviors can also be combined and configured to some extent in ways other than the pre-configured in the robot for demonstration purposes.

Also, special attention was devoted to the usability and simplicity of the application concerning the different levels of expertise of the expected users.

Some issues remain however. We would like to better fine tune the access to the camera that still seems problematic in terms of performance. It would have been interesting to try a different message format specific to this sensor or a streaming solution to test different possibilities and try to correlate them to find the best one, performance wise.

7.3 Future Work

Mixing the pre-configured behaviors of the robot in other ways than the default proved challenging. An overall overview of the ROBOTIS framework is due in order to increase its flexibility to a whole new range of novel applications other than the hardcoded demonstration programs.

Also, an intensive performance benchmark should take place to better tune the integration of the robot with the underlying monitoring processes.

It would be interesting to use ROS to communicate directly with the real robot.

And it would also be interesting to be able to use ROS and our application at the same time. That way we could have a system that is not tied to a particular technology.

References

[1] RoboCup Humanoid League website https://www.robocuphumanoid.org/. Retrieved May 16th, 2015.

[2] DARwin-OP manual can be found at http://support.robotis.com/en/product/darwin-op.htm. Retrieved May 16th, 2015.

[3] Robert L. Williams II, Ph.D., DARWIN-OP HUMANOID ROBOT KINEMATICS, Mechanical Engineering, Ohio University, Athens, Ohio, USA.

[4] Mansolino David, Mobile Robot modeling, Simulation and Programming, Master project, 2012-13, section of microengineering Biorobotics Laboratory (BioRob), Cyberbotics

[5] Inyong Ha, Yusuke Tamura, and Hajime Asama; Department of Precision Engineering, The University of Tokyo, Tokyo, Japan, Jeakweon Han and Dennis W Hong; Department of Mechanical Engineering, Virginia Tech, Virginia, USA, Development of Open Humanoid Platform DARwIn, SICE Annual Conference 2011, September 13-18, 2011, Waseda University, Tokyo, Japan

[6] Webots website Http://www.cyberbotics.com. Retrieved May 16th, 2015.

[7] ROS website http://www.ros.org/. Retrieved May 16th, 2015.

[8] Ubuntu website http://www.ubuntu.com. Retrieved May 16th, 2015.

[9] Gazebo website. Retrieved May 17th 2015 from http://gazebosim.org/.

[10] Debian website http://www.debian.org. Retrieved May 16th, 2015.

[11] Qt website http://qt-project.org/. Retrieved May 16th, 2015.

[12] C++ website http://www.cplusplus.com/. Retrieved May 16th, 2015.

[13] Qwt website http://qwt.sourceforge.net/. Retrieved May 16th, 2015.

[14] Boost website http://www.boost.org/. Retrieved May 16th, 2015.

[15] IEEE-RAS Technical Committee on Humanoid Robotics website http://www.humanoid-robotics.org/. Retrieved May 16th, 2015.

[16] IEEE International Conference on Robotics and Automation (ICRA) 2015 Robot Challenges website http://icra2015.org/conference/robot-challenges#!amazon_pick_banner_robot. Retrieved May 16th, 2015.

[17] Federation of International Robot-soccer Association (FIRA) website http://www.fira.net/main/. Retrieved May 16th, 2015.

[18] Humabot Challenge website http://www.irs.uji.es/humabot/. Retrieved May 16th, 2015.

[19] ROBOTIS OP2 DARwin-OP R&D Collaboration Project website http://www.robotis.us/robotis-op2-us/. Retrieved May 16th, 2015.

[20] ROBOTIS e-Manual v1.25.00, Running the Demonstration Programs can be found at http://support.robotis.com/en/product/darwin-op/operating/running_the_demonstration_programs.htm. Retrieved May 16th, 2015.

[21] ROBOTIS e-Manual v1.25.00, Framework can be found at http://support.robotis.com/en/product/darwin-op/development/framework.htm. Retrieved May 16th, 2015.

[22] Pennalizers Team website https://fling.seas.upenn.edu/~robocup/wiki/. Retrieved May 16th, 2015.

[23] Nuno Filipe dos Reis Almeida, Control Agent Architecture of a Simulated Humanoid Robot, Dissertation, Departamento de Electrónica, Telecomunicações e Informática, Universidade de Aveiro, 2008.

[24] ROS Hydro installation http://wiki.ros.org/hydro/Installation/Ubuntu. Retrieved June 8th, 2015.

[25] Gazebo installation http://gazebosim.org/tutorials?tut=install&ver=1.9&cat=install. Retrieved June 8th, 2015.

[26] Simulating DARwin-OP on Gazebo http://www.generationrobots.com/en/content/83-carry-out-simulations-and-make-your-darwin-op-walk-with-gazebo-and-ros. Retrieved June 8th, 2015.

[27] Simulating DARwin-OP on Gazebo main article http://www.ros.org/news/2015/02/darwin-op-package-for-rosgazebo-available.html. Retrieved June 8th, 2015.

[28] http://code.ua.pt/projects/darwin-op/repository/

[29] Ubuntu website http://www.ubuntu.com/news/eWeek_and_DesktopLinux_declare_Ubuntu_the_Champ. Retrieved May 16th, 2015.

Appendix A

1 Client Monitor Message

The monitor message is sent when the user enables/disables the monitorization of a sensor and is identified by the **monitor** identifier tag. No response is sent back from the server. The format of the message is outlined in Figure 58.

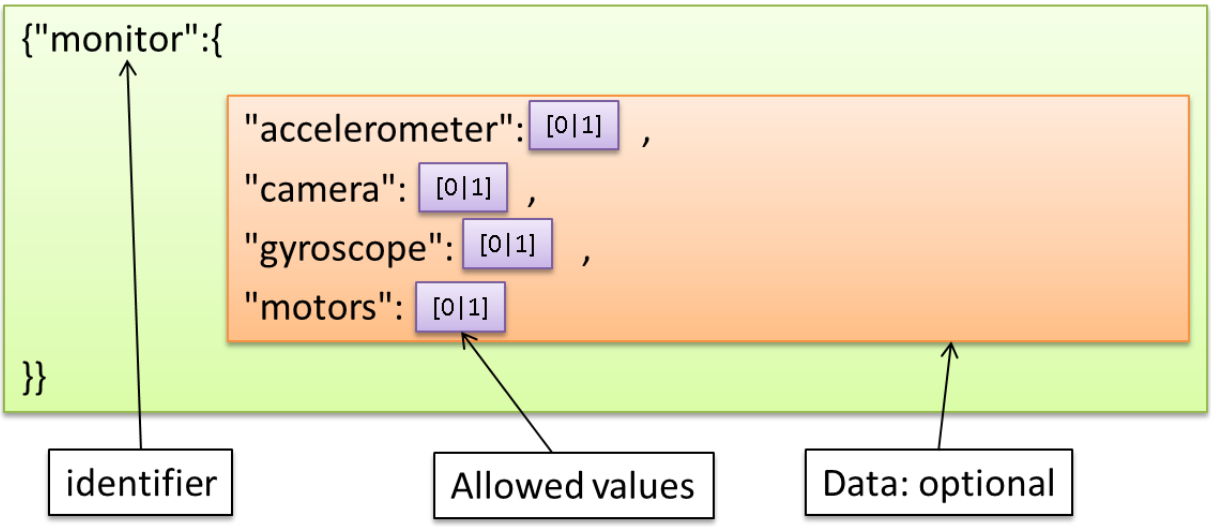


Figure 58 Monitor message general structure

- The message is identified by the **monitor** tag.
- The **accelerometer** optional field is used to signal if the accelerometer sensor should be monitored or not.
- The **camera** optional field is used to signal if the images from camera should be retrieved or not.
- The **gyroscope** optional field is used to signal if the gyroscope sensor should be monitored or not.
- The **motors** optional field is used to signal if the motors should be monitored or not.
- The allowed values are 0 or 1 meaning respectively don't monitor, monitor sensor.

Figure 59 depicts 2 examples of this message. The top message indicates that no sensors should be monitored. The bottom message is asking the server to monitor the accelerometer sensor.

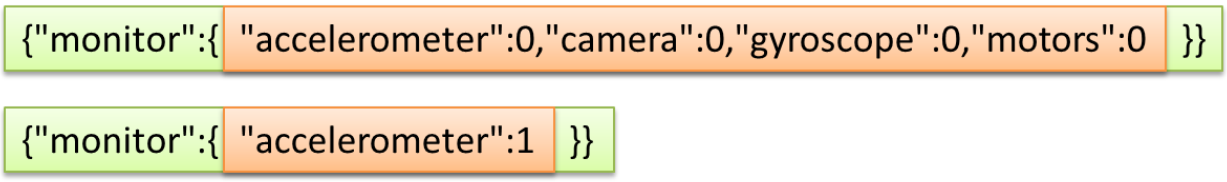


Figure 59 Monitor message examples

2 Operate Message

The operate message is sent when the user wants to change the position of a single motor and is identified by the **operate** identifier tag. No response is sent back from the server. The format of the message is presented in Figure 60.

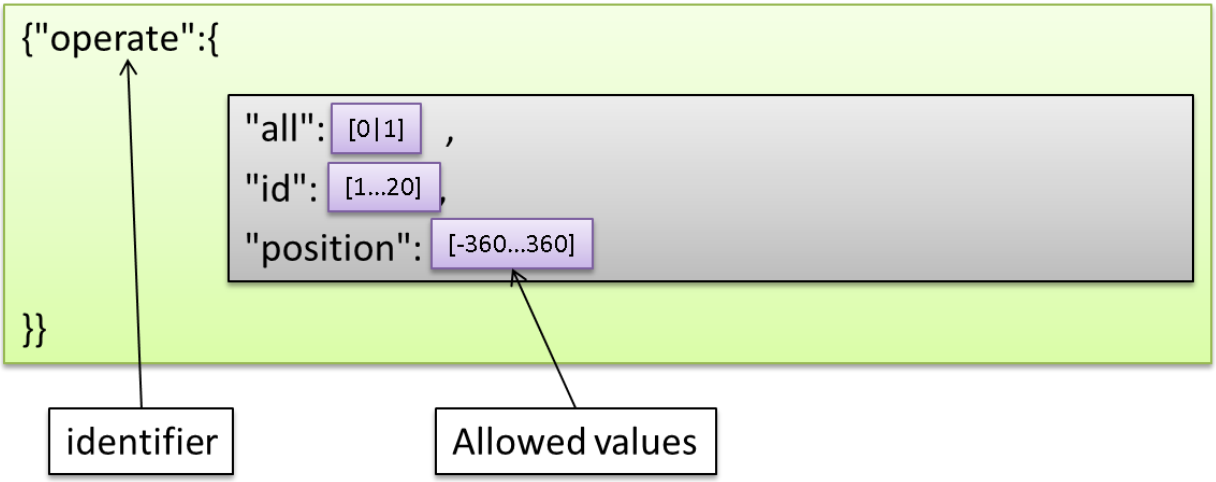


Figure 60 Operate message general structure

- The message is identified by the **operate** tag.
- The **all** non-optional field is used to signal if the change should be propagated to all motors or just the one identified by de id field. Currently only one motor is allowed to be manipulated by this message and this field remains to be fully used in future releases. The allowed values for this field are 0 to indicate only one motor and 1 to indicate all motors.
- The **id** non-optional field represents the id of the motor whose position should be changed. The allowed values are the ids of the motors in the range from 1 to 20.
- The **position** non-optional field represents the new position in degrees that the motor should assume. The allowed values range from -360 to 360 degrees.

In Figure 61 we can see an example of this message.

Figure 61 Operate message example

3 Led Message

The led message is sent when the user wants to change the color of the eyes or forehead leds and is identified by the **led** identifier tag. No response is sent back from the server. The format of the message is shown in Figure 62.

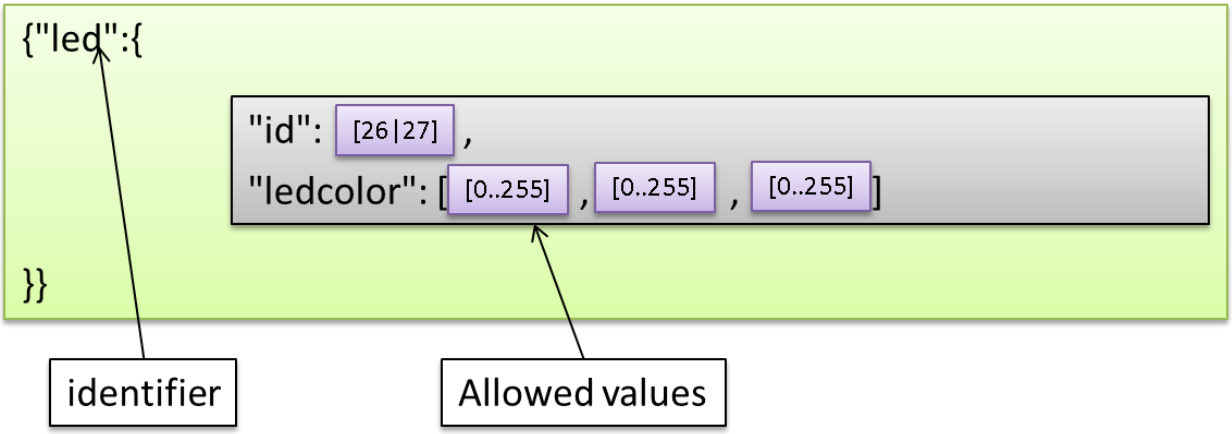


Figure 62 Led message general structure

- The message is identified by the **led** tag.
- The **id** non-optional field represents the id of the led whose color should be changed. The allowed values are the ids of the leds 26 and 27.
- The **ledcolor** non-optional field holds the color components of the new color. The allowed values to each of the color components Red Green and Blue (RGB) ranges from 0 to 255.

Figure 63 depicts an example message.

Figure 63 Led message example

4 Orchestrate Message

The orchestrate message as in Figure 64 is sent when the user wants the robot to perform a list of pre-configured behaviors in a sequential way and is identified by the **orchestrate** identifier tag. The server responds back. The format of the response message is shown in Figure 65.

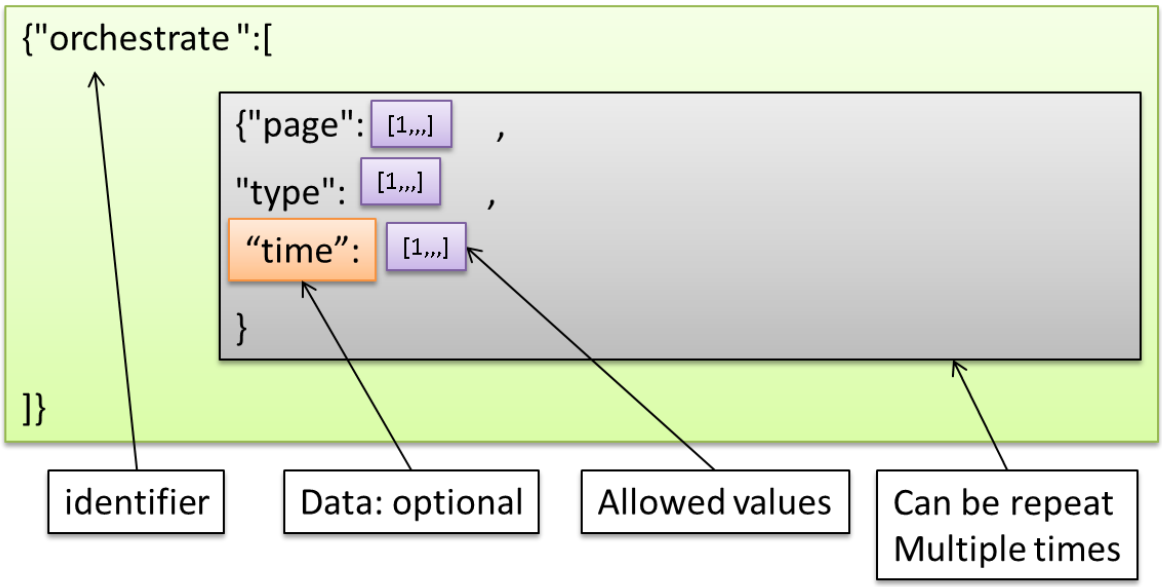


Figure 64 Orchestrate message general structure

- The message is identified by the **orchestrate** tag.
- The non-optional field **page** represents the behavior on the list and is a numeric number greater or equal to 1. As many behaviors can be added to the list the page element can be repeated several times.
- The non-optional field **type** represents the type of the behavior and ranges from 1 onwards.
- The optional field **time** is only used for the walking behavior and represents the time in seconds the robot should walk. It accepts any integer value greater than 1.

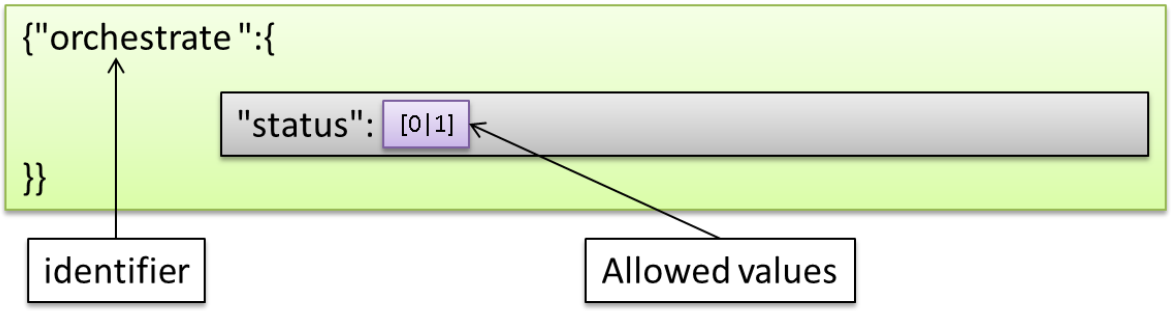


Figure 65 Orchestrate message response/stop general structure

Figure 65 above depicts the alternative structure of the orchestrate message. This alternate form can be sent in 2 different situations:

- The response sent by the server. In this case the non-optional field **status** will hold the value 0 in case the server reports the robot as stopped playing the list of behaviors sent by the client and the value 1 if the server wants to report that the robot as started playing the desired list.

- A request from the client telling the robot to stop as soon as possible and ignore the remaining items on the list. In this case the non-optional field **status** will hold the value 0 indicating to the server the desire to interrupt the robot performance.

Figure 66 and Figure 67 below present 2 examples of the orchestrate message. The first example represents a request from the client in the form of a list of pre-configured behaviors that the robot should perform in sequence. The second example depicts the response sent back by the server indicating, in this case, that the robot as started playing the list just received from the client.

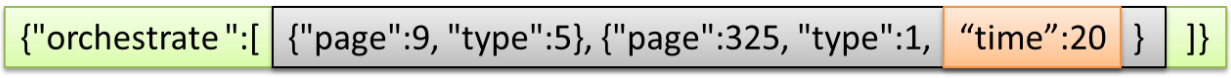


Figure 66 Orchestrate message example

Figure 67 Orchestrate message in alternative form. Response from the server informing that the robot as started playing the list of behaviors sent by the client

It should be noted that once the robot as played the full list, the server will inform the client of this fact by sending the message shown in Figure 68.

Figure 68 Orchestrate message sent by the server once the robot as finished playing the behaviors list

5 Walk Message

The walk message shown below in Figure 69 is sent when the user wants the robot to walk in one of four directions; forward, backwards, right and left, and is identified by the **walk** identifier tag. No response from the server is sent back.

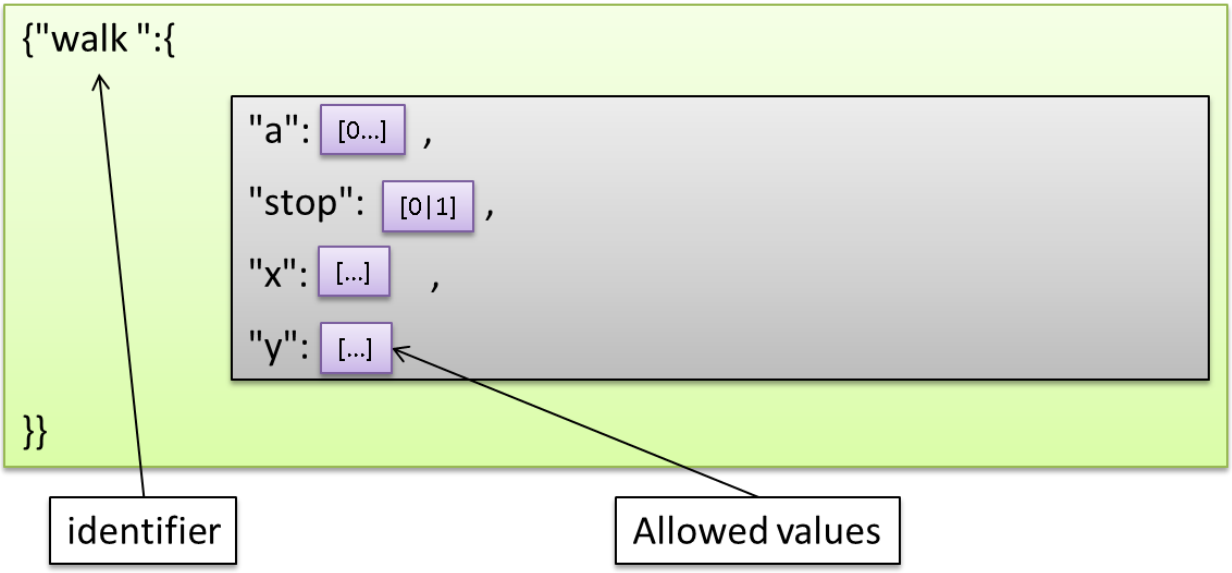


Figure 69 Walk message general structure

- The message is identified by the **walk** tag.
- The non-optional field **a** represents the amount of angular turn. Can hold any positive numeric value.
- The non-optional field **stop** can hold the values 0 indicating that the robot should start walking and 1 indicating that the robot should stop walking.
- The non-optional field **x** can hold numeric values. Negative values order the robot to walk backwards while positive ones instruct the robot to walk forward.
- The non-optional field **y** acts similarly to the x field. Negative values order the robot to walk to the right while positive ones instruct the robot to walk to the left.

Figure 70 shows an example message. In this case the robot is being ordered to walk forward as denoted by the positive value of the **x** field.

Figure 70 Walk message example

6 PlayBall Message

The playball message, identified by the **playball** identifier tag and depicted in Figure 71 is sent when the user wants the robot to perform the pre-configured behavior were the robot follows and kicks a red ball. The server responds to this request with a response informing the client if the actions as started or not. This message as 2 possible usages discussed shortly.

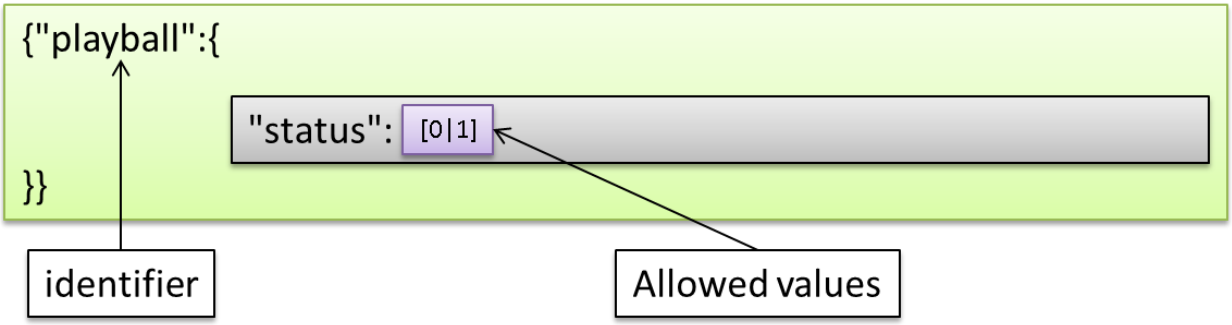


Figure 71 Playball message general structure

- The message is identified by the **playball** tag.
- The non-optional field **status** holds the values 0 or 1. The value 0 indicates that the robot should start the action while the value 1 instructs otherwise, that is the robot should stop.

An example of this message is shown in Figure 72. The example message depicts the response from the server informing the client that the action started. Alternatively the client can stop the action at any given time by sending this same message to the server.

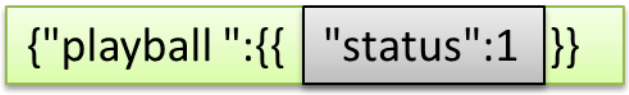


Figure 72 Playball message response example

7 Server Monitoring Message

Having received a monitor message from the client and as long as at least one sensor is being monitored, the server will send periodic messages to the client containing the values read from the aforementioned sensor(s). The general format of this message is presented in Figure 73.

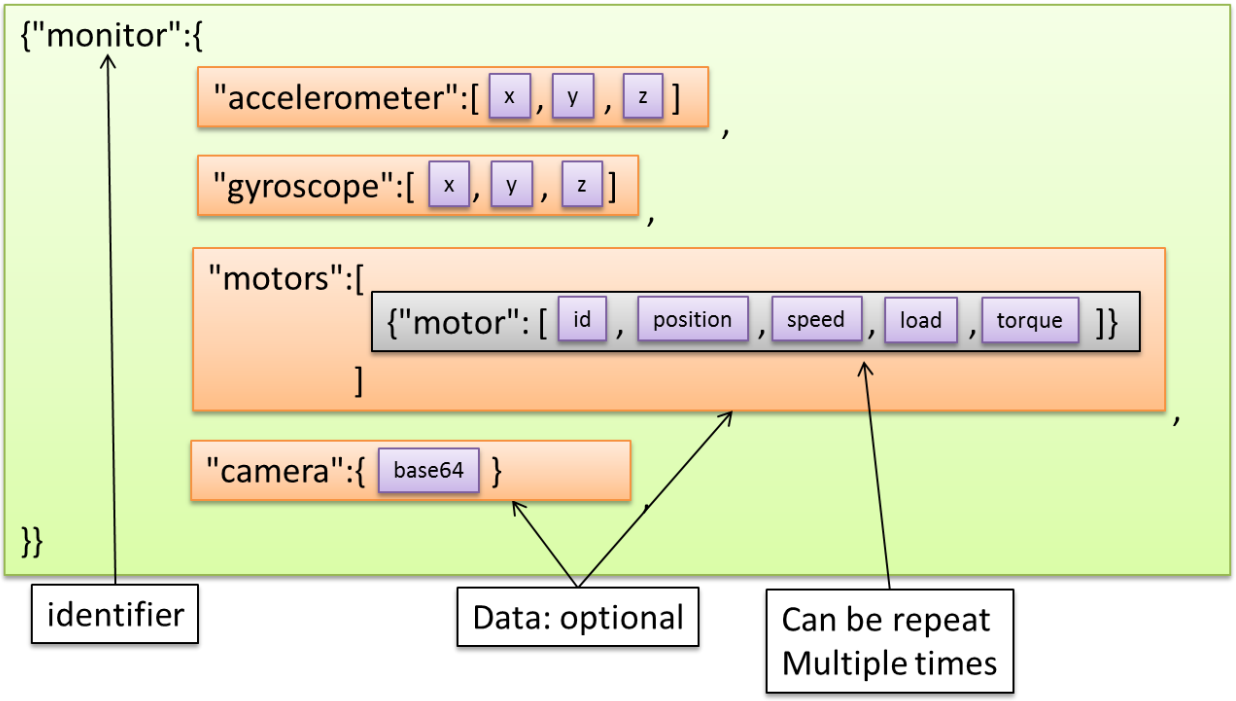


Figure 73 Server monitoring message general structure

- The message is identified by the **monitor** tag.
- The optional field **accelerometer** holds the values as read from the 3 axis of the accelerometer.
- The optional field **gyroscope** holds the values as read from the 3 axis of the gyroscope.
- The optional field **motors** is an array of **motor** tags.
- The non-optional field **motor** can be repeated multiple times, once per motor. Each occurrence of this tag includes the motor's id, position, speed, load and torque.
- The optional field **camera** holds one image at a time taken from the robot's camera encoded in base64 format.

An example of this message is shown in Figure 74. In the example presented only the motors are being monitored by the client.

Figure 74 Server monitoring message example

Appendix B

1 ROS Installation Procedure on the Robot Machine

We followed the tutorial at [24] with minor changes to install ROS Hydro on Ubuntu 12.04.5 LTS.

1 - Set links for Ubuntu 12.04 ROS packages
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu precise main" > /etc/apt/sources.list.d/ros-latest.list'

2 - Set up needed keys
wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -O - | sudo apt-key add -
sudo apt-get update

3 - Install ROS Base (ROS package, build, and communication libraries). No GUI tools.
sudo apt-get install ros-hydro-ros-base

4 - Before we can use ROS, we need to initialize rosdep.
sudo rosdep init
rosdep update

5 - Environment setup
echo "source /opt/ros/hydro/setup.bash" >> ~/.bashrc
source ~/.bashrc

6 – Get rosinstall, optional.
sudo apt-get install python-rosinstall

7 – Test ROS installation
roscore

2 ROS Installation Procedure on the Client Machine

We followed the tutorial at [25] to install gazebo.

Then we followed the tutorial at [24] to install ROS Hydro. The procedure is similar to the one used on the robot machine with the difference that we did a ROS full installation on the client machine (sudo apt-get install ros-hydro-desktop-full).

Finally we followed the tutorial at [26] to install and test the 3 packages for simulating the DARwin-OP robot on Gazebo.

The main article for the 3 packages for simulating the DARwin-OP robot on Gazebo can be found at [27].