ICT-2009-248730

Florence

Multi Purpose Mobile Robot for Ambient Assisted Living

STREP
Contract Nr: 248730

D2.6 Multi-purpose mobile robot implementation

Due date of deliverable: (30-09-2012)
Actual submission date: (31-10-2012)

Start date of Project: 01 February 2010 Duration: 36 months

Responsible WP: WP2 - TECNALIA

Revision: proposed

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
M		
m	Public	
PP	Restricted to other programme participants (including the Commission Service	
RE	Restricted to a group specified by the consortium (including the Commission Services)	X
CO	Confidential, only for members of the consortium (excluding the Commission Services)	

0 DOCUMENT INFO

0.1 Author

Author	Company	E-mail
Asier Fernandez	Tecnalia	asier.fernandez@tecnalia.com
Anthony Remazeilles	Tecnalia	anthony.remazeilles@tecnalia.com
Frank Van Heesch	Philips	frank.van.heesch@philps.com
Dietwig Lowet	Philips	dietwig.lowet@philips.com

0.2 Documents history

Document version #	Date	Change
V0.1	03/04/2011	Starting version, template
V0.2	17/09/2012	Definition of ToC
V0.3	02/10/2012	First complete draft
V0.4	11/10/2012	Integrated version (send to WP members)
V0.5	15/10/2012	Updated version (send to project internal reviewers)
V0.6	19/10/2012	First round of corrections and modifications
V0.7	25/10/2012	Second round of corrections and modifications
V0.8	29/10/2012	Third round of corrections and modifications
V1.0	31/10/2012	Final version

0.3 Document data

Keywords	Robot, elderly AAL, social robotics, tele-care, social connectedness, safety, coaching, collaborative gaming
Editor Address data	Name: Asier Fernandez Partner: TECNALIA Address: Miramon Pasealekua 1, 20009 DONOSTIA-SAN SEBASTIAN SPAIN Phone: +34 902.760.000 Fax: +34 901.706.009 E-mail: asier.fernandez.iribar@tecnalia.com
Delivery date	31-10-2012

0.4 Distribution list

Date	Issue	E-mailer
	Consortium members	al_florence_all@natlab.research.philips.com
	Project Officer	jan.komarek@ec.europa.eu
	EC Archive	INFSO-ICT-248730@ec.europa.eu

Table of Contents

0 DOCUMENT INFO 2
0.1 Author 2
0.2 Documents history 2
0.3 Document data 2
0.4 Distribution list 2
1 INTRODUCTION 5
1.1 Goal 5
1.2 Overview 5
2 SENSORS AND ACTUATORS ACCESS 12
2.1 Hokuyo laser 12
2.2 Kinect sensor 13
2.3 Turtlebot_node 16
3 ROBOT NAVIGATION STACK 18
3.1 Sensor sources 19
3.2 Frame transforms 20
3.3 Configuration of the navigation stack 22
3.4 How to use the navigation stack. 25
3.5 Laser Filter Component 30
3.6 Velocity Filter Component 32
3.7 Laser Combinator Component 35
3.8 The Florence Locator component 39
3.9 Detect Users Component 42
3.10 ApproachMe-Map Component 45
4 ROBOT ENABLERS 48
4.1 The FollowMe component 48
4.2 Filter User Tracker Component 52
4.3 The ApproachMe component 54
5 SUMMARY 58
6 REFERENCES 59
7 ANNEX A: USER MANUAL 60
7.1 Software installation 60
7.2 Calibration 61
7.3 Build a map 62

7.4 First time configuration 65
7.5 Starting the Florence system 65

1 Introduction

1.1 Goal

The goal of this deliverable is to describe the implementation of the software components developed in WP2 with the objective to extend and to detail the capabilities of the previous PekeeII robot within the context of the Florence project. The goal of the WP2 robotic software components is to provide the basic functionalities for controlling the Florence robot and for accessing sensor data. As such, these functionalities provide the middleware that is used by the WP3, WP4 and WP5 software components for building the Florence platform and services. Note that all software described in this document runs on the robot PC.

The development of the robot middleware required by Florence is strongly relying onto the Robot Operating System architecture (ROS, www.ros.org), which recently has reached sufficient maturity for reliable use. To ease and speed up the preparation of the robotic platform, we have decided to reuse appropriate ROS components when they exist, and to build dedicated ROS components to provide an API to other components.

Some changes from D2.3 are related to the Pekee-to-Turtlebot transition. As the robot has been changed, the hardware and software have been adapted.

The Turtlebot robot is well supported by the ROS framework. This means that most of the work regarding communication with this platform and sensor with the pc running ROS is available. Apart from that, several components have been added in order to improve navigation capabilities and some components have been modified to improve the robot's behaviour. This can be seen in the FlorenceLocator component described in Section 3.8, which now includes a better interaction with the user.

In short, this document presents how we use and adapt the ROS API to perform all the robot middleware services.

1.2 Overview

The development described in this document mainly provides motion capacities to the robot, as well as some basic human detection and interactions behaviours.

More specifically, in WP2 the following software components for the Robot are developed:

- Software components to access the Turtlebot sensors and motors (the turtlebot_florence package).
- Software components that are aimed at improving navigation as well as some aspects in the robot's behaviour
- Florence specific configuration of the ROS navigation stack.
- A FlorenceLocator component which makes use of the navigation stack (the move_base component). It is capable of sending navigation goals and controlling online navigation capabilities. Moreover, it can detect users and send the robot to them.
- A "follow-me" component that enables to make the robot follow the person while she is moving in the environment.
- An "Approach-me" component that provides the functionality of moving the robot close to the person to ease the interaction in between the robot and this person.

Note that FlorenceLocator, FollowMe and Approach_me describe software components that are aimed at being used by the WP3-5 software components. Among the components aforementioned, the Florence specific configuration, Florence Locator, follow me and approach me, describe software that can also be reused by higher level software adapted to specific circumstances. They can be considered to be proof of concepts and examples of how the Florence robot can be used.

An overview of these ROS and Hardware components and their relation is provided in Figure 1-1.

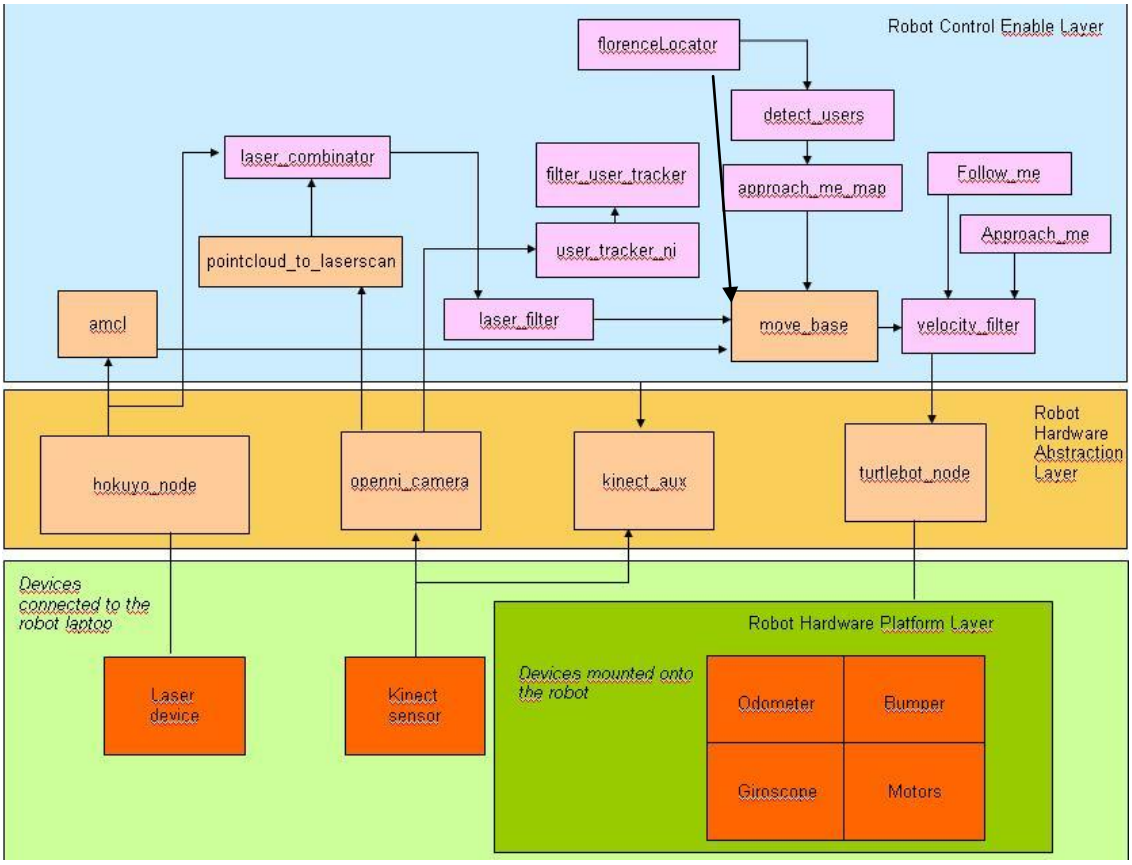

Figure 1-1 Overview of the robot components developed in WP2

In this figure, the red rectangles denote hardware; the brown boxes already existing ROS components and pink boxes Florence specific components that run on the robot PC. The arrows indicate the main flows of communication between the nodes. This can be either data flow or command flow.

As previously mentioned the development is realized using the ROS framework. We will thus start by summarizing the basic, key information with respect to this framework. We will then describe how the ROS framework can be installed onto the Turtlebot robot, and then how this is embedded within the developed framework.

1.2.1 A ROS-based implementation

Most of the multi-purpose mobile robot implementation has been realized within the ROS Framework [Quigley09], which is an open-source meta-operating system dedicated to robotic applications. This section will highlight some main concepts that

are broadly used for our implementation and that are thus frequently mentioned within the rest of the document.

The ROS framework addresses a set of requirements; the most important are to (i) make an abstraction of the hardware components (ii) ease the code reuse and (iii) facilitate the multi-process and multi-deployment design.

In this context, a *Node* is associated to a process performing computation. Since ROS is designed to be modular, a Node can be a sensor driver, a person detector within an image stream, or any other processing flow. Usually, an application is composed of several nodes, which interact between them through well-defined communication protocols.

ROS uses nodes that enable to realize communication through the definition of messages, as a structure composed of different typed fields. The selection of the type of communication relies on the underlying synchronization (or semantic operation) that is envisioned:

- *Topics* are used to transmit messages, following a “publish / subscribe” policy. The name of the topic is used to establish the connection. A subscriber only needs this name and the message type to get connected with any (one or several) nodes publishing to the mentioned topic. When the subscription is realized, the arrival of new messages triggers a call-back function defined by (each of) the subscriber(s).
- *Services* are used for implementing request/reply interactions. They are defined by two data structures, one for the request (input parameters) and one for the response (output parameters). Services are quite similar to classical function calls in that the calling node blocks until the response is received.
- *Actions* enable to launch asynchronous operations, unlike services which can be considered synchronous (i.e. blocking) communication tools. In this context, the result of the operation is not directly returned but is provided automatically once it is available. As an illustration, this kind of communication is well suited to navigation-like procedure, in which the first interaction (“go to the kitchen”) triggers a long-term operation, whose result will be available only at the end of the navigation process (“kitchen reached”). The interesting aspect is that such operation does not block the node requesting the operation. Specific tools enable to receive feedback data periodically as well to get informed once the goal has been reached.

Topics and services are embedded within the standard distribution of ROS. The Action is provided by the *actionlib* package.

In order to specify an *action*, the format of three messages needs to be defined: the *goal* message, which contains the input parameters to the action; the *feedback* message, which will be sent periodically by the provider of the action while it is running, to inform the caller about the action’s evolution; and the *result* message, that is sent only once, when the *action* finishes, and provides information about the final result.

As illustrated by the following figure, the execution of an action requires two components (the Client and the Server), communicating through different topics (represented by the different arrows in Figure 1-2).

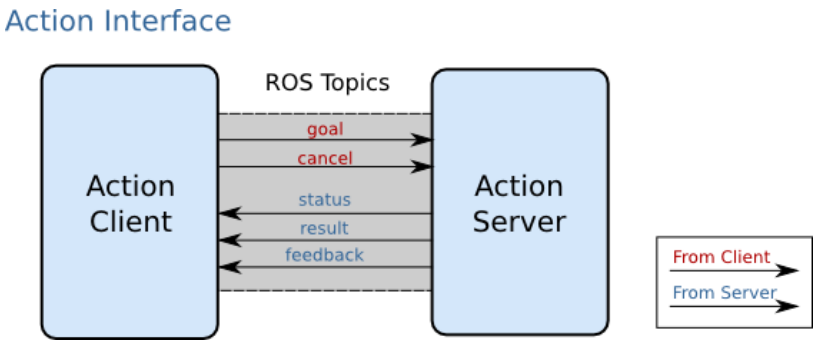

Figure 1-2: Client/Server Action communication flow (from the ROS online documentation [ROSaction])

The component willing to use a specific action creates an Action client instance, which is used to contact the Action server. The server then takes care of executing the steps required to perform the Action. The Action client sends to the server the details of the Action goal (by means of the *goal* message), and can continue doing "other work" while the action is asynchronously performed by the server. If needed, the action client can react to the *feedback* messages provided by the server and/or decide to stop the procedure at any time by sending a cancel message.

In addition to the *goal*, *feedback* and *result* messages, which are specified using a configuration file by the developer of the action, there are two other messages involved: the *cancel* message can be used by the client at any time to stop the Action server from continuing performing the action; the *status* message, informs the Action client of the status of the Action server by sending information about the state in which the internal state machine, which handles the server evolution, is..

The implementation of such paradigm is supported by the *actionlib* package that provides several implementations of the Client and Server classes [ROSaction]. One of the advantages of using these classes is that it enables to register specific functions as callback (like when a new feedback is received, or when the action is over). The specification of an action thus mainly relies on the specification of the data to be shared, and on the registration of related call-backs.

1.2.2 Turtlebot PC preparation

This section describes how to prepare the robot PC in order to launch the components described in the rest of the document. This installation mainly consists of putting the Linux environment onto this PC, and preparing the ROS framework. Note that this guideline can also be used to prepare any other computer to communicate with the mobile robot.

1. Ubuntu 11.10 installation
    a. Change boot order in BIOS to allow Booting from USB-CDROM
    b. Using a CD or a USB key with the Ubuntu installation program, install Ubuntu with the default settings.
    c. Create a user (selected to be `turtlebot` in the rest of the description)
    d. Follow the Ubuntu guidelines to get the latest packages.
2. ROS framework installation
    a. Setup ROS Ubuntu repository and key for apt. If the Ubuntu version is not 11.10, change 'oneiric' in the command bellow for the appropriate version name.

```
$ sudo sh -c 'echo "deb
http://code.ros.org/packages/ros/ubuntu oneiric main" >
/etc/apt/sources.list.d/ros-latest.list'

$ wget http://code.ros.org/packages/ros.key -O - | sudo apt-
key add –
$ sudo apt-get update
```

b. Install ROS *Electric* release and Turtlebot specific packages. This will install all ROS files in `/opt/ros/electric`

```
$ sudo apt-get install ros-electric-turtlebot-robot
```

c. Prepare the .bashrc file so that it sets up the ROS environment variable each time a terminal is opened, and source the file for the current session

```
$ echo "source /opt/ros/electric/setup.bash" >> ~/.bashrc
$ . ~/.bashrc
```

3. Install the additional required stacks that are not present in the installed ROS configuration

```
$ sudo apt-get install ros-electric-laser-drivers
```

After performing the steps above, the ROS installation is ready to run the software developed by the consortium. This software has to be installed by checking out a working copy from the consortium repository.

After installing the code locally, ROS has to be configured to be able to find the new packages. For this purpose, ROS provides an environment variable where the paths to the packages should be included. In this case a line should be added in the *.bashrc* configuration file with (all in one line)

```
export
ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:/path/to/florence/packages
```

Then, the *.bashrc* script has to be run again to update the environment variable by issuing the following command

```
$ . ~/.bashrc
```

The last step before being able to run the Florence system is to compile the packages developed by the consortium. This can be done by asking the ROS build toolchain to build the *florence_bringup* package, which has declared dependencies to every other packages in the project, using the command

```
$ rosmake --rosdep-install florence_bringup
```

The building process may ask for the admin password to install some additional ubuntu packages in case they're required.

After the building process has finished successfully, the whole system can be brought up by using one of the launch files (XML files with list of nodes to run) provided in the *florence_bringup* package. As an example, one of the launch files enables to run all the nodes:

```
$ roslaunch florence_bringup florence_bringup.launch
```

The different pieces of code described in the following sections correspond to the different ROS packages used or developed to upgrade the mobile robot with the requested functionalities.

The packages developed by the consortium are available at the SVN repository, within the WP2 directory. Each subdirectory corresponds to a package. Once checked out, the compilation of the sources is realized running "*rosmake* "within each of these directories.

1.2.3 Robot connection to ROS

Once ROS is installed on the robot, it is necessary to connect the robot to the ROS architecture. To do this we need to allow ROS to access the actuators and sensors located onto the Florence robot. Such operation will enable then to:

- **Control the actuators of the robot**: the ROS architecture assumes that it can send velocity commands using a message (geometry_msgs/Twist) defined in the base coordinate frame of the robot on a dedicated topic (the "cmd_vel"). Therefore, in order to make the robot "controllable" through the ROS framework, one just has to define a specific node that, through the subscription to the "cmd_vel" topic, is able to convert velocity information represented as a 6D twist vector into motor commands that can be sent to the mobile base. In the case of a differential drive platform like the Turtlebot robot, only the linear *x* component and the angular *z* component will be considered.
- **Read the odometry information**: odometry information is crucial for navigation. Such information is to be published as a nav_msgs/Odometry message and as a transform between the robot position and the fixed odometry frame using the tools provided by *tf* package. Using the *tf* package, the data can later be transformed to the desired reference frame (like sensor, mobile platform, or any other defined frame).
- **Read the rangefinder sensors information**: rangefinder sensors are frequently used to handle obstacles detection, avoidance and localization. Within the ROS architecture, it is assumed that these sensor data are publishing either through sensor_msgs/LaserScan or sensor_msgs/PointCloud messages.

In order to get the robotic platform fully supported by the ROS framework, we also have to specify the geometric model of the relative sensor positions and orientations. This is realized through the use of a URDF (Unified Robot Description Format) file. This file includes:

- **Robot shape description**: the ROS architecture needs to have one or more 3D geometric descriptions of the shape of the robot which is used for visualization and e.g. collision estimation. Typically a simple shape is used for collision detection while a more elaborate shape is used for visualization.
- **Coordinate frames description**: the ROS architecture requires that the robot is publishing information about the relationships between coordinate frames using the *tf* package. By parsing the URDF description, the *robot_state_publisher* node takes care of publishing all the frames of the robot using *tf.*

These configuration elements are used within the ROS framework and define the behaviour of the middleware. It is then possible to to build (our) services on top of this.

For example, the first useful API of this middleware is the navigation stack that takes in information from odometry, sensor streams, and a goal pose and outputs safe velocity

commands that are sent directly to the robot while the trajectory planner and obstacle avoidance of the robot are processing.

A dedicated ROS node, the Turtlebot_node communicates with the robot's firmware and also links to some of the standard ROS nodes of the navigation stack (e.g move_base). The Turtlebot's firmware manages the motors, the odometry, bumper sensors and analog and digital input and outputs.

The next section describes the ROS nodes that are used to get access to the different robot sensors and actuators. Section 3 provides information related to the navigation stack that is used to ease the robot navigation within its environment. In particular, all the shape and coordinate frame descriptions are described there. Finally, Section 4 will describe the other robot enablers that have been developed by the consortium to provide higher level control of the robot to the outside (as mentioned in Section 1.2 and Figure 1-1).

2 Sensors and actuators access

The robot is able to move in a 2D plane, so that the control is realized by specifying a 2D heading vector . Furthermore, the following sensors are available on the Florence robot: a Hokuyo laser scanner, a Microsoft Kinect 3D sensor, contacts sensor and odometry sensors. This section describes how all these sensors and actuators can be accessed through the ROS architecture.

2.1 Hokuyo laser

- The measurements from the laser scanner from Hokuyo are accessible through the ROS node named *hokuyo_node*[1]. When activated, it functions as a driver to the sensor. It publishes two topics, and provides one service. The interested reader can find more complete information within the ROS documentation. Description on how to use the Hokuyo laser scanners with the hokuyo node:

  http://www.ros.org/wiki/hokuyo_node/Tutorials/UsingTheHokuyoNode
- Description on how to dynamically reconfigure the hokuyo node:

  http://www.ros.org/wiki/hokuyo_node/Tutorials/UsingTheHokuyoNode

A brief summary follows here:

Table 1 The Hokuyo Node Interface

Hokuyo_node interface		
Published Topics		**Brief description**
Name	**Data type**	
/scan	/sensor_msgs/LaserScan	Header header float32 angle_min float32 angle_max float32 angle_increment float32 time_increment float32 scan_time float32 range_min float32 range_max float32[] ranges float32[] intensities
/diagnostics	/diagnostic_msgs/DiagnosticStatus	byte level string name string message string hardware_id keyvalue[] values
Published Service		

[1] http://www.ros.org/wiki/hokuyo_node

Name	Data type	Brief description
~self_test	diagnostic_msgs/SelfTest request	
	diagnostic_msgs/SelfTest response	string id byte passed DiagnosticStatus [] Status

The topic *scan* contains the sensing information. The following table provides a more extended description of the different parameters:

Table 2The scan message format

Scan message	
Entry	**description**
Header header	Header of the message, providing a timestamp for the first acquisition
float32 angle_min	start angle of the scan [rad] choosen between -2,08621rad and angle_max
float32 angle_max	end angle of the scan [rad] chosen between angle_min and 2.0862rad
float32 angle_increment	angular distance between measurements [rad]
float32 time_increment	time between measurements [seconds]
float32 scan_time	time between scans [seconds]
float32 range_min	minimum range value [m]
float32 range_max	maximum range value [m]
float32[] ranges	range data [m] (values < range_min or > range_ max should be discarded)
float32[] intensities	intensity data [device-specific units]

The second topic, *diagnostic_msg* describes the status of the sensor. It can be used to monitor if the sensor and its driver are working correctly. The service provided, *~self_test*, launches a set of more elaborated and specific tests that could be used to periodically check the laser sensor. This has not been used.

2.2 Kinect sensor

2.2.1 Image sensor

The *openni_camera*[2] package enables registering images (colour and/or depth images) acquired by the Kinect sensor. Its driver publishes more than 40 topics, so we won't list all of them. Only the most often used are mentioned.

[2] http://www.ros.org/wiki/openni_camera

The Kinect sensor can be seen as a collection of different sensors. It provides access to the colour image acquired by the RGB camera, as well as to the depth image provided by the depth sensor integrated. Furthermore, it enables accessing to the same depth data formatted as a point cloud directly. The topic */camera/rgb/points* illustrated in the following table enables associating each 3D point detected with the colour information observed within the colour image. The interested reader can find more complete information within the ROS documentation:

- Description of Camera_info:
  http://www.ros.org/doc/api/sensor_msgs/html/msg/CameraInfo.html
- Description of image format:
  http://www.ros.org/doc/api/sensor_msgs/html/msg/Image.html
- Description of the point cloud:
  http://www.ros.org/doc/api/sensor_msgs/html/msg/PointCloud2.html

A brief summary follows here:

Table 3The Openni camera interface

Openni_camera interface		
Published Topics		**Brief description**
Name	**Data type**	

`/camera/depth/camera_info`	`/sensor_msg/CameraInfo`	Provides a description of the different configuration parameters of the camera.
		`Header header`
		`uint32 seq`
		`time stamp`
		`string frame_id`
		`uint32 height`
		`uint32 width`
		`string distortion_model`
		`float64[] D`
		`float64[9] K`
		`float64[9] R`
		`float64[12] P`
		`uint32 binning_x`
		`uint32 binning_y`
		`RegionOfInterest roi`

`/camera/depth/image`	`/sensor_msgs/Image`	Contain the depth image acquired with some needed parameters to correctly use it.
		`Header header`
		`uint32 seq`
		`time stamp`
		`string frame_id`
		`uint32 height`
		`uint32 width`
		`string encoding`
		`uint8 is_bigendian`
		`uint32 step`

		uint8[] data
/camera/rgb/camera_info	/sensor_msg/CameraInfo	See topic /camera/depth/camera_info
/camera/rgb/image_color	/sensor_msgs/Image	If the image correspond to a classical color image, the topic presents the same structure as: /camera/depth/image
/camera/rgb/points	/sensor_msgs/PointCloud2	Enable to access to the cloud of 3D points with their respective color value. Header header Uint32 height Uint32 width PointField [] fields Bool is_bigendian Uint32 point_step Unint32 row_step Unint8[] data Bool is_dense

2.2.2 Kinect tilt motor and accelerometer sensor

The package *Kinect_aux*[3] enables to access the tilt motor of the Kinect, as well as its accelerometer. The interface is described in the following table created from http://www.ros.org/wiki/kinect_aux

Table 4The Kinect_aux interface

***Kinect_aux* interface**		
Subscribed Topics		**Brief description**
Name	**Data type**	
/tilt_angle	std_msgs/Float64	Used to set the tilt angle, in degree. The value must be within [-31;31]
/led_options	std_msgs/UInt16	Used to configure the kinect's LED (value within [0;7])
Published Topics		**Brief description**
Name	**Data type**	
/imu	Sensor_msgs/Imu	Provides the value measured by the embedded accelerometer
/cur_tilt_angle	std_msgs/Float64	Provide the current tilt angle, in degree

[3] http://www.ros.org/wiki/kinect_aux

/cur_tilt_status	std_msgs/UInt8	Provide a status information related to the tilt motor

2.3 Turtlebot_node

The same way that PekeeII robot had a main ROS driver to communicate with its interal microcontroller of the robot, Turtlebot also contains such a driver.

In order to control all the low level features of the robot's firmware, like the robot speed and steering control or like the access to the bumper measures or the odometry measures, a dedicated ROS Node, the turtlebot_node is used. This node is available within the turtlebot[4] ROS package.

Table 5 The turtlebot_node Node Interface

***Turtlebot_node Node* interface**		
Subscribed Topics		**Brief description**
Name	**Data type**	
cmd_vel	geometry_msgs/Twist	receives velocity commands to be transmited to the motors
Published Topics		**Brief description**
Name	**Data type**	
~sensor_state	turtlebot_node/TurtlebotSensorState	It updates sensor state at 10hz
odom	nav_msgs/Odometry	The odometry of the robot based on the gyro and sensor_state.
imu/data	sensor_msgs/Imu	The angular velocity and integrated position of the gryo.
/diagnostics	diagnostic_msgs/DiagnosticArray	The system diagnostic information published at 1Hz.
Published Service		**Brief description**
Name	**Data type**	
~set_digital_outputs	turtlebot_node/SetDigitalOutputs	Turtlebot internal microprocessor provides 3 digital outputs. uint8 digital_out_0 uint8 digital_out_1 uint8 digital_out_2
~set_operation_mode	turtlebot_node/SetTurtleb	It allows to set the robot in different operation modes

[4] http://ros.org/wiki/Robots/TurtleBot

	otMode	uint8 mode mode = 1 : Passive mode mode = 2 : Safe mode mode = 3 : Full mode bool valid_mode

3 Robot Navigation Stack

Navigation is one of the key features for a mobile robot. Even if it could appear very simple from a human point of view, robust navigation is a complex task using simultaneously:

- the control of the robot (with the command of the actuators),
- the odometers and telemeter's scanners data (to measure how the robot moves and how the environment evolves in real time),
- the knowledge and the memory of the navigation space (requiring a dynamic map) and
- the capacity to plan a complex trajectory with the ability to change this trajectory in case of obstacles, and if necessary to recover another equivalent trajectory after this obstacle is avoided.

We have decided to reuse the Navigation Stack provided within ROS (see http://www.ros.org/wiki/navigation).

This Navigation Stack serves to drive a mobile base from one location to another while safely avoiding obstacles. The Navigation Stack is fairly simple on a conceptual level. It takes in information from odometry and sensor streams and outputs velocity commands to be sent to a mobile base. As a pre-requisite to use this navigation stack, the robot must be running ROS, have a *tf* transform tree in place, and publish sensor data using the correct ROS message types. Also, the Navigation Stack needs to be configured for the shape and physical dynamics of the robot to perform at a high level.

In the following sections, we will explain how we have adapted the ROS navigation stack to be suited for the Florence robot. The navigation stack as described here will provide 2D like navigation by using the 2D laser scan data and Kinect 3D scan data. Despite a Kinect and 3D data is used, it is not a full 3D navigation because 3D information is converted to 2D and then a 2D map is used. This method decreases the computational load comparing to full 3D navigation. However, the stack provides support for 3D obstacle avoidance.

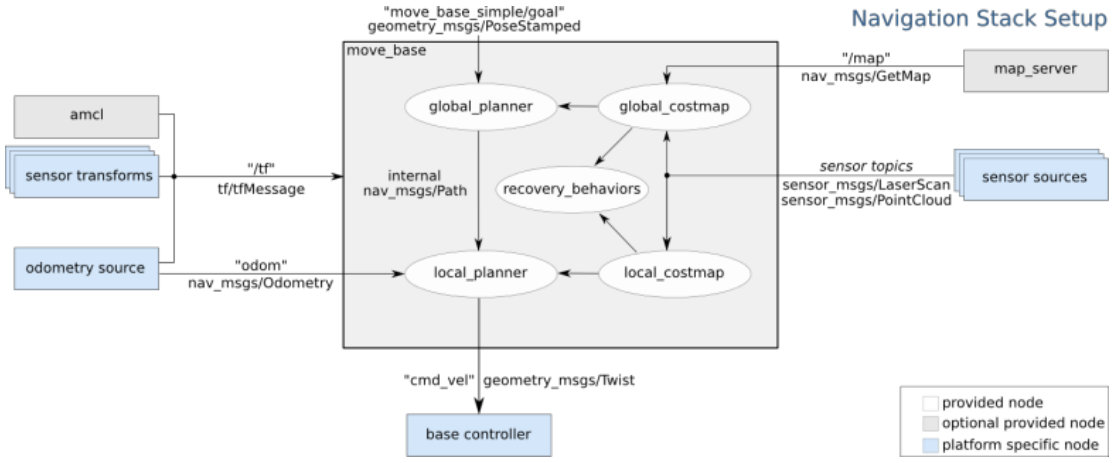

Figure 3-1 Overview of the components that are involved in the navigation stack

Figure 3-1 provides an overview of all the ROS components (nodes) that interact and are involved with the navigation stack. The main component in the navigation stack is the *move_base* node. This node is the one that provides the interface to the navigation

functionality via an *actionlib* action server or via messages on a topic. In any case, once a goal is received, the node will use the global planner it has loaded to try to find a trajectory that leads from the current position of the robot to the goal position. In the current state, the global planner is configured to use a static map to compute the path, and thus will not take into consideration the dynamic moving obstacles.

Once the global planner has obtained a trajectory, which consists of an array of poses, it is then handed to the local planner. The local planner will take the trajectory and calculate the appropriate motor commands that make the robot follow that trajectory while avoiding obstacles. Currently, the local planner is configured to run without a static map, and it will calculate the commands to send to the robot based on the global plan and on the obstacles it is able to see through the sensors. For each of the planners a costmap is maintained, consisting of a map, either static or dynamic depending on the case, which is used by each of the planners to compute the obstacle free paths. When the *move_base* node detects that the plan cannot be performed for some reason, then it also has a set of *recovery behaviours* that will move the robot in pre-defined sequences to try to clear the space around it.

In addition to the *move_base* node, the gray nodes in the figure above illustrate the *amcl* node and the *map_server* node. The *amcl* node is in charge of localizing the robot by reading the static map and comparing it to the sensor readings available. It then publishes its result by broadcasting its estimated transform between the odometry frame of the robot and the base frame of the map. The *map_server* node, in turn, is in charge of reading the previously created static map and publishing the data using ROS messages. These components are shown in gray because they are optional, and can be substituted for other equivalent nodes, like for example a node doing SLAM (Simultaneous Localization and Mapping).

The remaining components in the figure, the blue ones, are the nodes that are specific to the robot being used. In the case of the Florence project, the *base_controller* and *odometry_source* functionalities are both provided by the *turtlebot_node* node; the *sensor_sources* functionality is provided by *hokuyo_node* which publishes the laser rangefinder scans; and the *sensor_transforms* functionality is provided by the *robot_state_publisher* node, that publishes the position of the sensors on the robot by reading the URDF description.

In addition to providing the nodes depicted in blue in the figure, some navigation stack specific configuration files must be prepared.

The base controller functionality and the odometry source are implemented by the turtlebot_node already described in Section 2.3. Sections 3.1 and 3.2 will provide some additional details about the sensors data used and about publishing the appropriate frame transforms. Section 3.3 will describe the configuration and launch files required to setup the navigation system. Section 3.4 will then describe how this component can be used. Finally, a component that we developed and that is strictly based onto this Navigation stack, the florenceLocator component, will be defined in Section 3.8.

3.1 Sensor sources

The Navigation Stack as described in this report only uses the 2D laser scanner data. This means that the navigation step will not be able to avoid obstacles that are not visible in the 2D plane. In a later step, we will describe the use of the Infrared and ultrasound distance sensors in others 2D plans and also the Kinect in the navigation step to also enable 3D obstacle avoidance.

The laser scan data is provided by the Hokuyo laser driver node as described in Section 2.1.

3.2 Frame transforms

Many ROS packages (including the Navigation Stack) require the transform tree of a robot to be published using the *tf* software library. At an abstract level, a transform tree defines offsets in terms of both translation and rotation between different coordinate frames. To make this more specific, consider the example of a simple robot that has a mobile base with a single laser mounted on top of it. Two coordinate frames can be associated to the robot: one corresponding to the centre point of the base of the robot and one for the centre point of the laser that is mounted on top of the base, then *tf* can be used to publish the transform relating the position and orientation of those frames.

3.2.1 URDF model of the Florence robot

As it has already been mentioned, the URDF model of the robot contains the transforms that relate the position and orientation of several parts of the robot. Particularly, this includes the transform between the *base_link* frame, which is the frame in which the robot movement commands are received, and the *laser* frame, which is the frame in which the rangefinder measurements are given. The transform between those two frames is required by the Navigation Stack, and can be provided by running *robot_state_publisher*. The Navigation Stack may also need a 3D model of the robot to perform 3D obstacle avoidance. This 3D model can also be specified in URDF syntax and can also be used to visualize and/or simulate the Florence robot. ROS provides a URDF of the Turtlebot Robot. As the robot structure has been changed, a new URDF definition has been made. Instead of using original Turtlebot package a copy of all packages has been added to the svn repository within WP2 called turtlebot_control. This package contains a folder called urdf. It contains different files: original and modified files.

This changes have been made in the physical hardware: change the height between plates(10cm, 25cm and 50cm) and added new plates. Apart from that a laser scanner has been added positioned upside down. We have modified two files, and they described below:

- turtlebot_new_hardware.xacro

  Bellow an example of how to modify the height of the standoff with 0.25meters.

```
<macro name="turtlebot_standoff_25cm" params="parent number x_loc y_loc z_loc">
  <joint name="standoff_25cm_${number}_joint" type="fixed">
    <origin xyz="${x_loc} ${y_loc} ${z_loc}" rpy="0 0 0" />
    <parent link="${parent}"/>
    <child link="standoff_25cm_${number}_link" />
  </joint>

  <link name="standoff_25cm_${number}_link">
    <inertial>
      <mass value="0.00001" />
      <origin xyz="0 0 0" />
      <inertia ixx="1.0" ixy="0.0" ixz="0.0"
        iyy="1.0" iyz="0.0"
```

```
      izz="1.0" />
    </inertial>

    <visual>
      <origin xyz=" 0 0 0 " rpy="0 0 0" />
      <geometry>
        <cylinder length="0.25" radius="0.005"/>
      </geometry>
      <material name="Black1">
        <color rgba="0.4 0.4 0.4 1.0"/>
      </material>
    </visual>

    <collision>
      <origin xyz="0.0 0.0 0.0" rpy="0 0 0" />
      <geometry>
        <cylinder length="0.0635" radius="0.0381"/>
      </geometry>
    </collision>
  </link>
</macro>
```

- turtlebot_new_body.xacro
  Bellow an example of how to add a laser frame with an inverted frame

```
<!-- add laser-->
  <link name="base_laser_link"> // CREATE A FRAME
    <visual>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
      <box size="0.05 0.05 0.041"/>
      </geometry>
      <material name="Red">
      <color rgba="0.5 0.0 0.0 1.0"/>
      </material>
    </visual>
  </link>
  <joint name="base_laser_joint" type="fixed">
    <parent link="plate_1_link" /> // THE FRAME IS PLACED IN PLATE 1
    <child  link="base_laser_link" />
    <origin xyz="0.06 0 -0.02" rpy="3.14159265 0 0" />// INVERTED
  </joint>

  <link name="base_laser_round_link">
    <visual>
      <origin xyz="0 0 0" rpy="0 0 0" />
```

```
    <geometry>
      <cylinder radius="0.021" length="0.029" /> // DEFINE LASER RAY
    </geometry>
   <material name="Red">
    <color rgba="0.5 0.0 0.0 1.0"/>
    </material>
  </visual>
</link>
<joint name="base_laser_round_joint" type="fixed">
   <parent link="base_laser_link" />
   <child  link="base_laser_round_link" />
   <origin xyz="0 0 0.035" rpy="0 0 0" />
</joint>
```

Robot's visualization can be seen in the next figure.

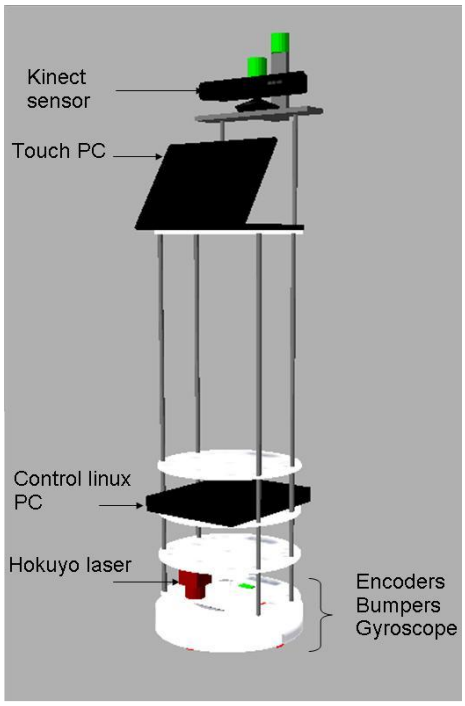

Figure 3-2 URDF model of the Florence robot

3.3 Configuration of the navigation stack

In this section, we describe and provide the configuration and launch file used to setup the robot and the Navigation Stack nodes required to run a static map-based navigation procedure. These configuration and launch files are included in the project repository, in the Turtlebot_control package.

A full description of the available parameters for the navigation costmaps is available at the *costmap_2d* package documentation in the ROS Wiki [ROScostmap].,

3.3.1 turtlebot_2dnav_control.launch

This launch file is a script that starts all the nodes required to be able to run the basic Navigation Stack.

```
<launch>

  <!--- minimal electric --> //Communication, Sensor, robot_state_publiser, filters….
  <include file="$(find turtlebot_control)/launch/minimal_control.launch" />

  <!--- Run the kinect --> // Run the kinect and get the fake laser
  <include file="$(find turtlebot_control)/launch/kinect_control.launch" />

  <!-- Run the map server --> // make the map available over  ROS
  <arg name="map_file" default="$(find turtlebot_control)/map/testlab.yaml"/>
  <node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)" />

  <!--- Run AMCL --> // Run Localization component
  <include file="$(find turtlebot_control)/config/amcl_turtlebot_control.launch" />

  <!--- Run Move Base --> // Run navigation Component
  <include file="$(find turtlebot_control)/config/move_base_turtlebot_control.launch" />

</launch>
```

3.3.2 costmap_common_params.yaml

This file sets the parameters that are common to the global and local costmaps, which are the maps used by both planners to find the best trajectories through free space.
One of the most critical parameters here are the robot footprint, representing the projection of the robot in the ground plane, which is specified as a set of *n* points forming a polygon.
Another important parameter is the observation sources parameter, which defines which sensors will be used to update the costmaps. For each sensor it requires specifying the frame in which the sensor data is provided, the type of sensor, the topic in which the data is published by the sensor and whether the sensor will be used to mark obstacles in the costmap, clear obstacles, or both. Currently the combination of the laser rangefinder and kinect's scan data is being used, but before passing to navigation stack it is filtered, and it is defined to look at **filtered_scan** topic.

```
obstacle_range: 2.5
raytrace_range: 3.0

footprint: [[-0.16, -0.16], [-0.16, 0.16], [0.19, 0.16], [0.24,
0.0], [0.19, -0.16]]
footprint_padding: 0.01
inflation_radius: 0.40

observation_sources: scan
scan: {data_type: LaserScan, topic: /filtered_scan, marking:
true, clearing: true}
```

3.3.3 global_costmap_params.yaml

This file defines the configuration parameters for the global costmap. The most important parameters are: *global_frame* that indicates which frame should be used by the costmap, which in case of using a static map should be set to the frame the map is being published in; *robot_base_frame*, which specifies the name of the robot base link; and *static_map*, which is a boolean that specifies whether a static map is being used.

```
global_costmap:
   global_frame: /map
   robot_base_frame: /base_link
   update_frequency: 5.0
   publish_frequency: 0.0
   static_map: true
   transform_tolerance: 0.5
```

3.3.4 local_costmap_params.yaml

This file stores the configuration options specific to the local costmap. In this case, the most important parameters, as in the case of the global costmap parameters are: *global_frame*, *robot_base_frame*, and *static_map*. In this case, since the local planner doesn't use a static map, *rolling_window* must be set to *true.*

```
local_costmap:
   global_frame: /odom
   robot_base_frame: /base_link
   update_frequency: 5.0
   publish_frequency: 5.0
   static_map: false
   rolling_window: true
   width: 10.0
   height: 10.0
   resolution: 0.1
   transform_tolerance: 0.5
```

3.3.5 base_local_planner_params.yaml

This is the configuration file for the local planner node, which includes some parameters that limit the acceptable velocities and accelerations that can be commanded to the robot. It also contains the *holonomic_robot* parameter, which defines which kind of drive the robot uses. For the Turtlebot robot it must set to true, since it's a holonomic drive robot (meaning that the robot cannot move in any direction at any given time, and it has to manoeuvre to move in certain directions instead). This parameter should be set to false only while using an omnidirectional drive robot (like one using mecanum wheels).

```
controller_frequency: 5.0
TrajectoryPlannerROS:
  max_vel_x: 0.50
  min_vel_x: 0.10
  max_rotational_vel: 1.5
  min_in_place_rotational_vel: 1.0
  acc_lim_th: 3.0
```

```
acc_lim_x: 0.50
acc_lim_y: 0.50

holonomic_robot: false
yaw_goal_tolerance: 0.3
xy_goal_tolerance: 0.25
goal_distance_bias: 0.7
path_distance_bias: 0.70
sim_time: 1.5
heading_lookahead: 0.325
oscillation_reset_dist: 0.05

vx_samples: 6
vtheta_samples: 20
dwa: false
```

3.3.6 move_base_turtlebot_control.launch

This is the launch file that loads all navigation parameters.

```
<launch>
  <node pkg="move_base" type="move_base" respawn="false" name="move_base"
output="screen">
    <rosparam file="$(find turtlebot_control)/config/costmap_common_params.yaml"
command="load" ns="global_costmap" />
    <rosparam file="$(find turtlebot_control)/config/costmap_common_params.yaml"
command="load" ns="local_costmap" />
    <rosparam file="$(find turtlebot_control)/config/local_costmap_params.yaml"
command="load" />
    <rosparam file="$(find turtlebot_control)/config/global_costmap_params.yaml"
command="load" />
    <rosparam file="$(find turtlebot_control)/config/base_local_planner_params.yaml"
command="load" />
  </node>
</launch>
```

3.4 How to use the navigation stack.

It is assumed that the “ros-diamondback-desktop-full” ROS configuration is installed on the robot, as well as the extra required stacks as described in Section 1.2.2. This stack contains already the “off-the-shelf” navigation components. It is also assumed that the code from the Florence project repository is installed onto the robot and compiled.

The adapted navigation stack can then be started by running (in separate terminals)

- `roscore`
- `roslaunch turtlebot_control turtlebot_2dnav_control.launch`

The navigation stack provides functionality that allows the robot to navigate to an arbitrary goal (x, y, angle) in the map.

The node responsible for providing the interface to this functionality is *move_base* node. This node runs an action server, but also accepts receiving commands by

sending a message to the *move_base_simple/goal* topic. Messages received in this topic will make *move_base* call its own action server to trigger the action.

Running the move_base node on a robot that is properly configured results in a robot that will attempt to navigate to a goal pose at (x, y, angle) within a user-specified tolerance. The move_base node will eventually get within this tolerance of its goal or signal failure to the user.

3.4.1 Interface

As mentioned, the interface for this node is composed of an action server which runs the *move_base* action, the *move_base_simple/goal* topic, and also several services. These are described in the following table.

Table 6 Overview of the ROS navigation stack interface

***move_base* interface**		
Subscribed Topics		**Brief description**
Name	**Data type**	
move_base/goal	move_base_msgs/MoveBaseActionGoal	A goal for move_base to pursue in the world. Header header Bool OnOff
move_base/cancel	actionlib_msgs/GoalID	A request to cancel a specific goal. Header header UInt16 RangeMin UInt16 RangeMax UInt16 StressValue Bool OnOff
move_base_simple/goal	geometry_msgs/PoseStamped	Provides a non-action interface to move_base for users that don't care about tracking the execution status of their goals.
Published Topics		**Brief description**
Name	**Data type**	
move_base/feedback	move_base_msgs/MoveBaseActionFeedback	Feedback contains the current position of the base in the world (/world_cu).
move_base/status	actionlib_msgs/GoalStatusArray	Provides status information on the goals that are sent to the move_base action.
move_base/result	move_base_msgs/MoveBaseActionResult	Result is empty for the move_base action.
move_base/cmd_vel	geometry_msgs/Twist	A stream of velocity commands meant for execution by a mobile base.
Published Service		**Brief description**
Name	**Data type**	
	nav_msgs/GetPlan	Allows an external user to ask for a plan to a given pose from

~make_plan		move_base without causing move_base to execute that plan.
	None	None
~clear_unknown_space	std_msgs/Empty	Allows an external user to tell move_base to clear unknown space in the area directly around the robot. This is useful when move_base has its costmaps stopped for a long period of time and then started again in a new location in the environment
	None	None
~clear_costmaps	std_srvs/Empty	Allows an external user to tell move_base to clear obstacles in the costmaps used by move_base. This could cause a robot to hit things and should be used with caution.
	None	None

Many parameters are also configurable to fine tune the performance of the planning and navigation algorithms used, to configure the map size and quality and to configure the period of the updating process of the map built. Among those parameters we can find:

- **base_global_planner** (string, default: "navfn/NavfnROS"): identification of the global planner to be used
- **base_local_planner** (string, default: "base_local_planner/TrajectoryPlannerROS"): identification of the local planner to be used
- **recovery_behaviors** (list, default: [{name: conservative_reset, type: clear_costmap_recovery/ClearCostmapRecovery}, {name: rotate_recovery, type: rotate_recovery/RotateRecovery},: A list of recovery behavior to use with move_base,. These behaviors will be run when move_base fails to find a valid plan in the order that they are specified. After each behavior completes, move_base will attempt to make a plan. If planning is successful, move_base will continue normal operation. Otherwise, the next recovery behavior in the list will be executed.
- **controller_frequency** (double, default: 20.0): The rate in Hz at which to run the control loop and send velocity commands to the base.
- **planner_patience** (double, default: 5.0): How long the planner will wait in seconds in an attempt to find a valid plan before space-clearing operations are performed.
- **controller_patience** (double, default: 15.0): How long the controller will wait in seconds without receiving a valid control before space-clearing operations are performed.
- **conservative_reset_dist** (double, default: 3.0): The distance away from the robot in meters at which obstacles will be cleared from the costmap when attempting to clear space in the map. Note, this parameter is only used when the default recovery behaviors are used for move_base.
- **recovery_behavior_enabled** (bool, default: true): Whether or not to enable the move_base recovery behaviors to attempt to clear out space.

- **clearing_rotation_allowed** (bool, default: true): Determines whether or not the robot will attempt an in-place rotation when attempting to clear out space.
- **shutdown_costmaps** (bool, default: false): Determines whether or not to shutdown the costmaps of the node when move_base is in an inactive state
- **oscillation_timeout** (double, default: 0.0): How long in seconds to allow for oscillation before executing recovery behaviors. A value of 0.0 corresponds to an infinite timeout.
- **oscillation_distance** (double, default: 0.5):How far in meters the robot must move to be considered not to be oscillating. Moving this far resets the timer counting up to the ~oscillation_timeout

Nota that the parameters above are read by the nodes in the navigation stack at initialization and that they need to be set either on the *move_base.launch* file or set using the *rosparam* command line tool before launching the *move_base.launch* file.

3.4.2 Implementation Details

The current ROS implementation of the navigation stack uses a grid-based global planner that assumes circular or polygonal robot footprint. This means that the global planner will produce waypoints for the robot that are optimistic for the actual robot raw footprint, and may in-fact be infeasible if the map is not correct or if some obstacles appear during the following of the trajectory.

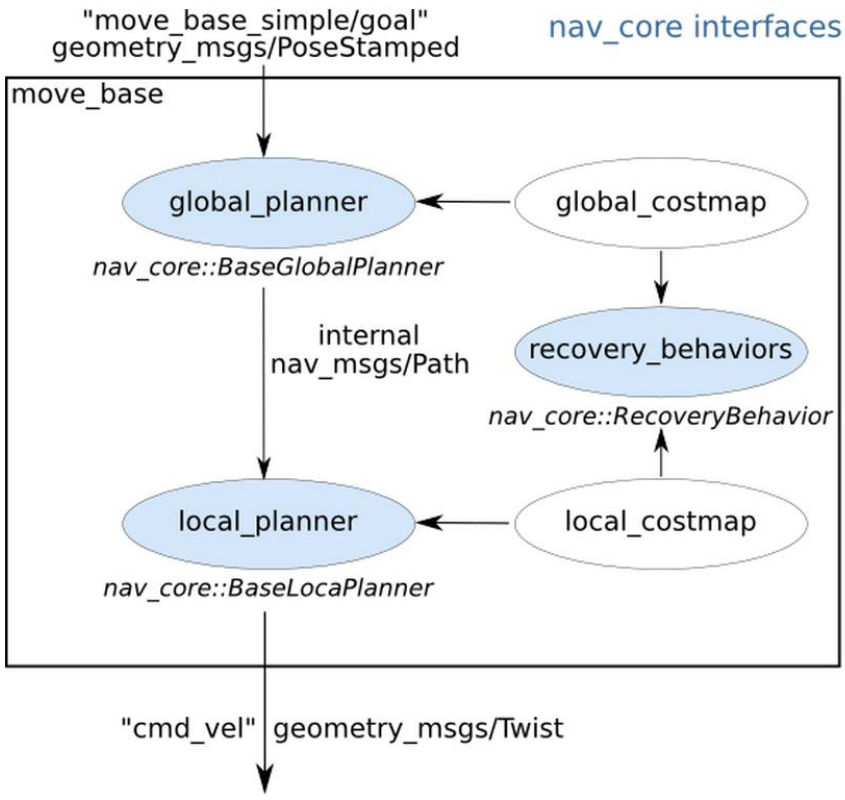

Figure 3-3 Overview of internal components of the move_base node

The move_base node uses a number of internal components, as shown in Figure 3-3, which have their own ROS APIs. These components may vary based on the values of the ~base_global_planner, ~base_local_planner, and ~recovery_behaviors respectively. The move_base node links together a global and local planner to accomplish its global navigation task. It supports any global planner adhering to the nav_core::BaseGlobalPlanner interface specified in the nav_core package and any local planner adhering to the nav_core::BaseLocalPlanner interface specified in the nav_core package. The move_base node also maintains two costmaps, one for the global planner, and one for a local planner (see the costmap_2d package) that are used to accomplish navigation tasks.

The nav_core::BaseGlobalPlanner provides an interface for global planners used in navigation. All global planners written as plugins for the move_base node must adhere to this interface. Current global planners using the nav_core::BaseGlobalPlanner interface are:

- navfn - A grid-based global planner that uses a navigation function to compute a path for a robot. This is the default global planner used if a different option is not specified.
- carrot_planner - A simple global planner that takes a user-specified goal point and attempts to move the robot as close to it as possible, even when that goal point is in an obstacle.

The nav_core::BaseLocalPlanner provides an interface for local planners used in navigation. All local planners written as plugins for the move_base node must adhere to this interface. Current local planners using the nav_core::BaseLocalPlanner interface are:

- base_local_planner - Provides implementations of the Dynamic Window and Trajectory Rollout approaches to local control

3.4.3 Developers Guide

Here an example is provided on how to use the `move_base` ROS node, in order to perform the Goto X-Y functionality. A client application can specify action goals in the form of Movebase messages as follows.

First an action client must be created to be able to interface with the move_base action server:

```
actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>
move_base_client("move_base", true);
```

Then a goal message must be filled and sent using the just created client

```
move_base_msgs::MoveBaseGoal goal;
goal.target_pose.header.frame_id = "map";
goal.target_pose.header.stamp = ros::Time::now();
goal.target_pose.pose.position.x = desired_x;
goal.target_pose.pose.position.y = desired_y;
goal.target_pose.pose.orientation =
tf::createQuaternionMsgFromRollPitchYaw(0, 0, desired_angle);

move_base_client.sendGoal(goal);
```

Then the node calling node can do other work, or otherwise sleep waiting for the navigation to finish by calling

```
move_base_client.waitForResult();
```

And check the result by requesting the status with.

```
Move_base_client.getState();
```

A more detailed tutorial on how to use the `move_base` ROS node is given at: http://www.ros.org/wiki/navigation/Tutorials/SendingSimpleGoals

3.5 Laser Filter Component

This component is used to clear obstacles from the costmap. The costmap automatically subscribes to sensors topics over ROS and updates itself accordingly. Each sensor is used to either mark (insert obstacle information into the costmap), clear (remove obstacle information from the costmap), or both. A marking operation is represented by an index into an array giving the change the cost of a cell. A clearing operation, however, consists of raytracing through a grid from the origin of the sensor outwards for each observation reported. If a three dimensional structure is used to store obstacle information, obstacle information from each column is projected down into two dimensions when put into the costmap.

When an obstacle is registered in the costmap and it moves and disappears, the information should be included in the costmap, but since this is not guaranteed, obstacle might not be cleared form the costmap if the scanner can't detect any obstacles within its maximum range. This leads to a cluttered navigation space because despite there is an open space, the robot considers that there is an area containing obstacles.

3.5.1 Interface

The following diagram shows how the laser_filter is used within the Florence system. The filter was developed to use only with Hokuyo laser scanner but when the laser_combinator component was developed, the same strategy to filter the data was chosen.

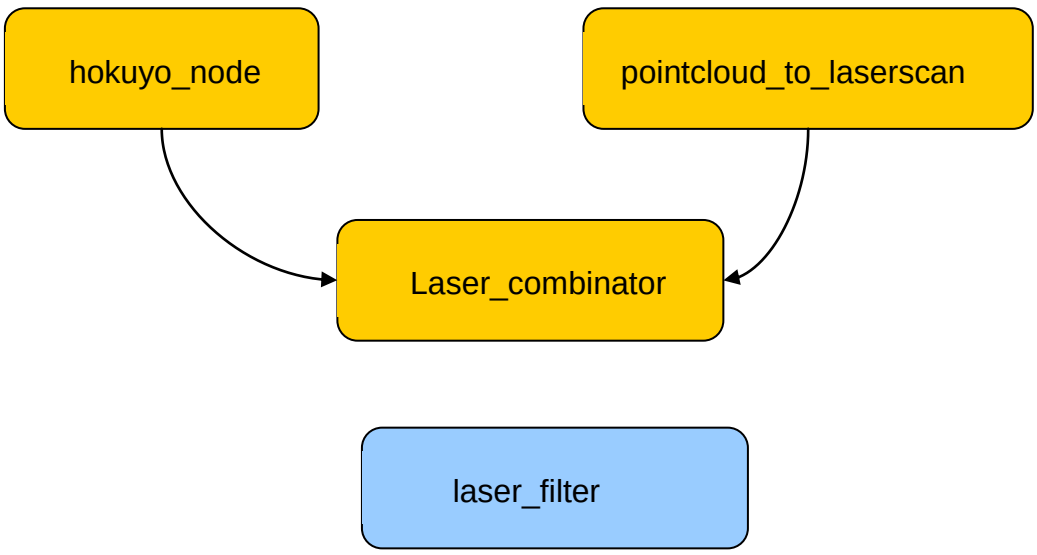

Figure 3-4. Laser Filter Component

The previous table shows that laser_filter component gets /combined_scan topic as an input, but the following table explains that it gets /scan topic. ROS framework allows to easily change input and output parameter names. This is done by remapping the names. This is shown in the following example. The following lines show how the node is launched in the main launch file:

```
<!-- Laser Filter -->
<node pkg="laser_filter" type="laser_filter_node" name="laser_filter_node">
   <remap from="scan" to="scan_combined"/>
</node>
```

Table 7. Interface for Laser Filter Interface

laser_filter interface		
Subscribed Topics		**Brief description**
Name	**Data type**	
/scan	sensor_msgs/LaserScan	Header header float32 angle_min float32 angle_max float32 angle_increment float32 time_increment float32 scan_time float32 range_min float32 range_max float32[] ranges float32[] intensities
Published Topics		**Brief description**
Name	**Data type**	
/filtered_scan	sensor_msgs/LaserScan	Header header float32 angle_min float32 angle_max float32 angle_increment float32 time_increment float32 scan_time float32 range_min float32 range_max float32[] ranges float32[] intensities

3.5.2 Implementation details

The Laser_filter component gets the reading of the /scan data and compares with the maximum range of the laser scanner. If an out of range value arrives, a slightly smaller value is assigned to the scan data.

Apart from that, as the Hokuyo laser detects Turtlebot's structure, and it is not interesting to include this information as obstacles, all laser information within robot's radius is removed.

3.5.3 Working hypothesis and current limitations

When the Hokuyo_node or the Kinect_fake_laser node get out of the range readings, they publish non-valid scan data. At this moment if an obstacle is detected and then disappears, it remains in the costmap. This means that the robot will consider that a obstacle remains. At this moment a non out-of-range scan reading is needed. adopted workaround for this requirement has been implemented by substituting out-of-range values by values slightly smaller than the maximum scan data value.

This workaround does not affect the localisation component because it always gets the original laser scan data. It can affect the navigation component because it could consider non existing obstacles at the maximum distance. However, the navigation component has been defined in such way that does not consider obstacles in this range. As an example, the navigation component takes into account obstacles in the range of 3 meters and the scan data provided by Hokuyo laser and Kinect have a range of 5.6 meters and 4 meters respectively.

3.5.4 Developers Guide

This component is developed under C++ programming language. As it is explained before, as soon as it receives a scan data and makes all calculations.

There is a callback function that manages this operation:

void laser_scanCB (const sensor_msgs::LaserScan::ConstPtr& msg)

This callback makes two things:

- If the laser gets a reading because it detects the robot's structure (a bar, a cable…), it filters assigning 0 value to the reading

```
if (filtered_scan.ranges[i] < ROBOT_RADIUS)
        filtered_scan.ranges.at(i) = 0.0;
```

- If a out-of-range value is read, assign a slightly smaller value.

```
if(filtered_scan.ranges[i] >= filtered_scan.range_max ||
filtered_scan.ranges[i] <= filtered_scan.range_min)
    filtered_scan.ranges[i] = filtered_scan.range_max - 1e-4;
```

3.6 Velocity Filter Component

It has been found that original Turtlebot ROS package does not allow a smooth control of the robot. The velocity commands that arrive to the turtlebot_node (main node in charge of driving the robot) are applied directly to the motors. As the robot's original structure has been adapted and the new structure is higher the robot has become more instable than the original design.

In order to minimize the instability, a velocity filter has been developed, and a smooth control of the robot has been achieved.

3.6.1 Interface

The communication interface for Velocity Filter Component is detailed in the following table.

Table 8. Interface for Velocity Filter Interface

velocity_filter interface		
Subscribed Topics		**Brief description**
Name	**Data type**	
/cmd_vel	geometry_msgs/Twist	Vector3 linear float64 x float64 y float64 z Vector3 angular float64 x float64 y float64 z
Published Topics		**Brief description**
Name	**Data type**	
/filtered_cmd_vel	geometry_msgs/Twist	Vector3 linear float64 x float64 y float64 z Vector3 angular float64 x float64 y float64 z
Parameters		**Brief description**
Name	**Data type**	
max_linear_acc	Float	It defines maximum linear acceleration

		DEFAULT_MAX_LINEAR_ACC: 1.0
max_angular_acc	Float	It defines maximum angular acceleration DEFAULT_MAX_ANGULAR_ACC: 1.0
publishing_rate	Float	It defines the rate that the output will be published DEFAULT_PUBLISHING_RATE: 50
command_timeout	Float	It defines a timeout for the incoming velocity command DEFAULT_COMMAND_TIMEOUT: 1.0

3.6.2 Implementation details

As soon as a velocity command is received, it is filtered. Filtering process consist on limiting the acceleration. This means that a variation in the velocity commands will be limited, and velocity could not increase or decrease immediately.

3.6.3 Working hypothesis and current limitations

This component needs a continuousstream of velocity commands. If a command does not arrive within a predefined interval, it gradually decreases the velocity to zero. This prevents from non-desirable situations where for an unknown reason the velocity command is not being published and the robot is taking the last value.

3.6.4 Developers Guide

This component is developed under C++ programming language. It receives a velocity command through a topic, and publishes the filtered velocity command.

The main function is called spin, and it is executed periodically:

void spin(void)

It checks how old is the last received velocity command, and in case it is older than the time-out, it send a null velocity command.

```
if((ros::Time::now() - last_velocity_cmd_vel_t_) >= command_timeout)
{
   desired_twist_ = geometry_msgs::Twist();
}
```

It calculates the desired accelerations (angular and linear) and if they are higher than the maximums, it limits them. Bellow an example for the linear:

```
float des_linear_acc = (desired_twist_.linear.x
filtered_twist_.linear.x)*publishing_rate_;
if(des_linear_acc >= 0)
{
    filtered_twist_.linear.x = (des_linear_acc > max_linear_acc_) ?
    filtered_twist_.linear.x + max_linear_acc_/float(publishing_rate_) :
    desired_twist_.linear.x;
}
else
{
    filtered_twist_.linear.x = (des_linear_acc < -max_linear_acc_) ?
    filtered_twist_.linear.x - max_linear_acc_/float(publishing_rate_) :
    desired_twist_.linear.x;
}
```

3.7 Laser Combinator Component

In order to improve navigation and detect obstacles in a different level than the laser scanner height, this component has been developed.

In order to achieve 3D like navigation, the Kinect sensor's depth information is used. The depth information is computed in order to get a fake laser scan data. Turtlebot's original stack contains a node called pointcloud_to_laserscan. It is in charge of transforming 3D point cloud data into a laser scan data.

3.7.1 Interface

The communication interface for Laser Combinator Component is detailed in the following table.

Table 9. Interface of the LaserCombinator interface.

LaserCombinator interface		
Subscribed Topics		Brief description
Name	Data type	
/scan	Sensor_msgs/LaserScan	The scan data is provided by Hokuyo Laser Header header float32 angle_min float32 angle_max float32 angle_increment float32 time_increment float32 scan_time float32 range_min float32 range_max

		float32[] ranges float32[] intensities
/kinect_scan	Sensor_msgs/LaserScan	The kienct_scan data is provided by pointcloud_to_laserscan node within the turtlebot stack Header header float32 angle_min float32 angle_max float32 angle_increment float32 time_increment float32 scan_time float32 range_min float32 range_max float32[] ranges float32[] intensities
Published Topics		**Brief description**
Name	**Data type**	
/scan_combined	Sensor_msgs/LaserScan	This is the main output of the component. It is a LaserScan type message and it contains combined information of hokuyo laser and the kinect Header header float32 angle_min float32 angle_max float32 angle_increment float32 time_increment float32 scan_time float32 range_min float32 range_max float32[] ranges float32[] intensities
Parameters		**Brief description**
Name	**Data type**	
frame_id	string	It allows to define a different frame name for the laser scanner DEFAULT_LASER_FRAME: "/base_laser_round_link"
kinect_frame_id	string	It allows to define a different

		frame name for the Kinect DEFAULT_KINECT_FRAME: "/camera_depth_frame"
hokuyo_node/max_ang	float	It is used to adjust combination range in according the range of the scanners. Automatically gets the hokuyo laser's maximum range. DEFAULT_LASER_RANGE: 1.570

Table 10. Interface for ScanCombined message.

Scan_combined message	
Entry	**description**
Header header	Similar to /scan topic. Header of the message, providing a timestamp for the first acquisition
float32 angle_min	Similar to /scan topic . start angle of the scan [rad] choosen between -2,08621rad and angle_max
float32 angle_max	Similar to /scan topic end angle of the scan [rad] chosen between angle_min and 2.0862rad
float32 angle_increment	Similar to /scan topic angular distance between measurements [rad]
float32 time_increment	time between measurements [seconds].
float32 scan_time	time between scans [seconds]
float32 range_min	Similar to /scan topic . minimum range value [m]
float32 range_max	Similar to /kinect_scan topic maximum range value [m]
float32[] ranges	Similar to /kinect_scan topic range data [m] (values < range_min or > range_ max should be discarded)
float32[] intensities	Similar to /scan topic. intensity data [device-specific units]

3.7.2 Implementation details

Laser Combinator node compares laser scanner's and Kinect's scan data, and it publishes the nearest value. As the Kinect_scan data is available at a higher

frequency, as soon as a new kinect_scan data arrives, both data are compared and published.

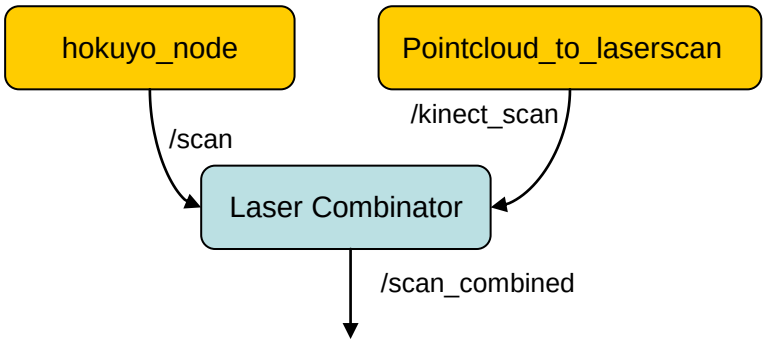

Figure 3-5. Laser Combinator component

3.7.3 Working hypothesis and current limitations

As the Kinect sensor has a small angle of view, 60 degrees, the robot cannot detect high obstacles in a wide range, it can only properly detect obstacles that are placed in front of it. This means that as soon as the robot rotates the obstacles will be lost.

Moreover, the Kinect can detect obstacles from 55cm-60cm on. If the robot gets very close to the obstacles, the Kinect will not be able to detect it. This could lead into a confusing behaviour, but safer than single laser scan navigation.

It has to be considered that this software is prepared to get laser scan when the laser scanner is placed into the robot upside down.

This component is implemented as a ROS package, written in C++. In order to work properly it needs two scans data. In this case it gets laser scan and kinect scan data.

3.7.4 Developers Guide

This component is implemented as a ROS package, written in C++. In order to work properly it needs two scans data. In this case it gets laser scan and kinect scan data.

As soon as the node is executed, it tries to get laser and kinect scan data. The node only publishes the resulting combined_scan data if two valid scan data (hokuyo and kinect) arrives within 1 second. If no valid data arrives the program stops publishing.

The hokuyo scan data and kinect data are received in the following callback functions, and they make a copy of their values in order to compare in the main function

void laser_scanCB (const sensor_msgs::LaserScan::ConstPtr& msg)

void kinect_scanCB (const sensor_msgs::LaserScan::ConstPtr& msg)

This component has a main loop that is executed periodically.

int spin(void)

The hokuyo scanner's data and Kinect scanner data provide different amount of data, with different resolutions.

It is known that hokuyo scanner's amount of data is higher than the kinect's so, a strategy to compare all hokuyo scanner's data has been chosen.

Both scanners frames are inverted 180º because the laser is upside down. Before comparing them, kinect's data is ordered.

The comparison between both scanners data is done by comparing angles. The angle to be incremented has the same resolution as the hokuyo scanner. Basically in each angle both scanners reading are compared, and the smallest one is saved.

3.8 The Florence Locator component

The ability to navigate a mobile platform, as described in the section above, is essential for navigation, but not sufficient to easily allow services and application to move the robot to meaningful and descriptive locations. For example, a service might want to navigate to "the kitchen" or to "the user".

To this end, the FlorenceLocator component provides an abstraction for the navigation component. It gets the robot's position by building upon the navigation functionality from ROS (i.e. the move_base and amcl nodes). From this the robot 's position it derives both in coordinates (x,y and orientation on a floor plan) and on a room level using room labels (e.g. "kitchen", "living room", "toilet"). Moreover, it implements functionality to send specific goals to the robot, again using either coordinates or room labels, and request the navigation status (identical to the move_base states). The FlorenceLocator component also implements functionality to navigate to a user, both on the room-level and the coordinate level, using the sensors of the home network (through the CMF) and the robot's Kinect. The functionality allows to navigate the robot to a 'polite' position close to the user.sThis is particularly, suitable when initiating user dialogs.

In the sections below, we will explain the implementation of this component in more detail.

3.8.1 Interface

The interface of the florenceLocator component comprises functions that services and applications can use for locating the robot and the user, functions for navigating the robot as well as functions for sensors to update the position of the user and functions for configuring the navigation behaviour. Below we will first provide the API and discuss its functionality in more details afterwards.

Table 11. Interface for FlorenceLocator Component.

***FlorenceLocator* interface**		
Subscribed Topics		**Brief description**
Name	**Data type**	
`/florenceLocator/manualOverride`	`std_msgs/Bool`	Enables/disables the control of the component of the robot. When a true is received the current navigation goal is aborted and this component does not produce any navigation goals.
Published Service		**Brief description**
Name	**Data type IN/OUT**	

/florenceLocator/ getUserPosRoom	None std_msgs/String	Returns the label of the room where the user has last been detected.
/florenceLocator/ getUserPosXYPhi	None std_msgs/Float32, std_msgs/Float32, std_msgs/Float32	Returns the location where the user has last been detected as a x,y,phi-coordinate.
/florenceLocator/ getRobotPosRoom	None std_msgs/String	Returns the label of the room where the robot's currently located.
/florenceLocator/ getRobotPosXYPhi	None std_msgs/Float32, std_msgs/Float32, std_msgs/Float32	Returns the robot's current location as a x,y,phi-coordinate.
/florenceLocator/ getGoalPosRoom	None std_msgs/String	Returns the room label of the robot's most recent navigation goal.
/florenceLocator/ getGoalPosXYPhi	None std_msgs/Float32, std_msgs/Float32, std_msgs/Float32	Returns the location of the robot's most recent navigation goal as a x,y,phi coordinate.
/florenceLocator/ getHotspot	std_msgs/String std_msgs/Float32, std_msgs/Float32, std_msgs/Float32	Requires a room label and returns a room's hotpot as a x,y,phi coordinate.
/florenceLocator/ setHotspot	std_msgs/String, std_msgs/Float32, std_msgs/Float32, std_msgs/Float32 None	Sets the location of a hotspot as a x,y,phi coordinate, corresponding to a room label.
/florenceLocator/ getStatus	None std_msgs/String	Returns the current navigation status. The syntax is copied from
/florenceLocator/ gotoUser	None None	Commands the robot to go to the current (last known) user position
/florenceLocator/ gotoRoom	std_msgs/String None	Commands the robot to go to the hotspot corresponding to the given room label.
/florenceLocator/	std_msgs/Float32,	Commands the robot to go to the

gotoXYPhi	std_msgs/Float32, std_msgs/Float32	location specified by the x,y,phi location.
	None	
/florenceLocator/ setUserPosRoom	std_msgs/String	Updates the coordinate of the current user location to the hotspot corresponding to the provided room label, in case the current location corresponds to a different room.
	-	
/florenceLocator/ setUserPosXYPhi	std_msgs/Float32, std_msgs/Float32, std_msgs/Float32	Updates the coordinate of the current user location
	-	

The Florence locator component simplifies the navigation of the robot and the locating of both robot and user by making use of three other components: The navigation stack of ROS (described above), the “goto user” component described below, and the CMF that provides user position updates.

The FlorenceLocator component maintains a list of hotspots. Hotspots are labeled (x,y,phi) coordinates, which are mainly used for translating room labels to specific navigation goals, although they are also used to specify the location of fixeds (e.g. a TV or blood pressure measure).

The FlorenceLocator component keeps track of the user’s location, by allowing any sensor to call setUserPosXYPhi() or setUserPosRoom(), depending on the accuracy of the sensor. When setUserPosRoom() is called it only updates the (x,y,phi) coordinate in case the current location corresponds to a different room. Typical user position updates are provided by PIR sensors through the CMF and the Kinect’s user tracker, but also the activation of a certain device (like a water cooker) could be used to update the user’s position.

Because many services include a dialog with the user, often the robot needs to navigate to the user. Instead of navigating to the exact position that the user has last been detected, the robot needs to detect the actual position of the user and approach it up to a certain polite distance. To this end, the gotoUser() command consists of navigating to the room that the user is last known, followed by a “detect user” and an “approach user” step. These are described in more detail in the following section.

3.8.2 User detection and approach Component

Section 3.9 and 3.10 describes in detail the description of these components. The main idea consist in the following process: the robot turns 45º, it stops, and it tries to detect a user. If no user is detected the process is repeated until a complete round is finished. Whenever a user is detected, the user position coordinate is updated.

When a user’s location is known, the robot is approached by calculating a safe navigation goal that takes a safety distance into account. This safety distance prevents colliding with the user.

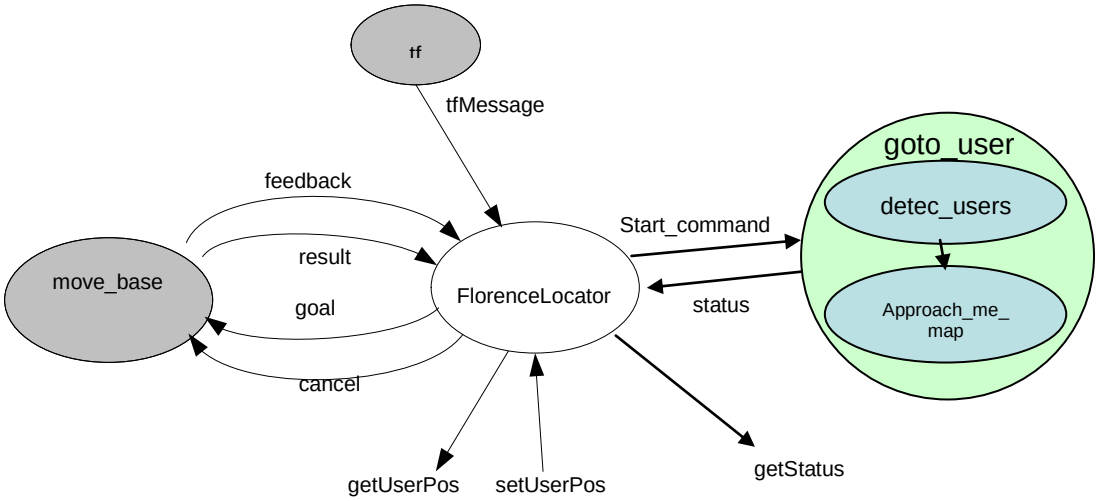

Figure 3-6. FlorenceLocator Component and goto_user component

3.8.3 Developers Guide

This component is developed under python programming language. It provides services for each functionality. As an example of that can be gotoRoom service:

In the initialisation of the class the service is created:

```
class FlorenceLocator:
  def __init__(self):
      self.gotoRoom_srv = rospy.Service('florenceLocator/gotoRoom',
      GotoRoom,      self.gotoRoom)
```

When a service is called, the following function is called and executed:

```
def gotoRoom(self, req):
```

3.9 Detect Users Component

This component is integrated with FlorenceLocator component. As the ROS framework allows to easily interconnect different components, it has been decided to develop components as modular as possible.

The FlorenceLocator component provides an interface to send the robot to the user. The first step consists on sending the robot to the room or area where the user is located. Once the robot has achieved this location it tries to detect the user. Detect User component is in charge of this task.

3.9.1 Interface

Table 12. Interface for detect_users Component.

detect_users interface		
Subscribed Topics		**Brief description**
Name	**Data type**	
/detect_user/cmd	std_msgs/Bool	It contains a command to start detecting any user. It is called by florenceLocator
/tracked_users	User_tracker_ni/TrackeUsers	This message contains information about detected users int32[] user_ids int32[] user_ids_ni int32 num_users_tracked
Published Topics		**Brief description**
Name	**Data type**	
/approach_user/status	std_msgs/String	This message is used tosend the feedback to the florenceLocator component Two different messages are defined: ACTIVE: the robot is running the action ABORTED: the robot could not achieve the user
/detected_user	detect_users/DetectedUser	This message contains the activation command and the position of detected user bool command float32 user_position_x float32 user_position_y
Subscribed Services		**Brief description**
Name	**Data type**	
/user_tracker_ni_node/Start	user_tracker_ni/Start	This service creates the class that tracks users
/user_tracker_ni_node/Stop	user_tracker_ni/Stop	This service destroys the class in charge of detecting users. This is done to delete previous information that openNI could keep.

3.9.2 Implementation details

Detect User component is triggered by previously described ROS topic. This topic is sent by FlorenceLocator component. Once it gets the command, the process of detecting the user starts.

Detect user component uses openNI framework and Kinect camera. It provides an interface to detect and get user's information. Despite it works well in static situations, its performance is lesson mobile platforms because it uses movement information to discriminate users (from a static background). When the Kinect is placed into the robot and it starts moving itusers false positive rate increases.

The strategy adopted to prevent false detections is to make stop and go iterations. More precisely, the robot starts turning itself 45 degrees and stops, then it tries to detect a user. This process is repeated until a full circle is made or when user is detected.

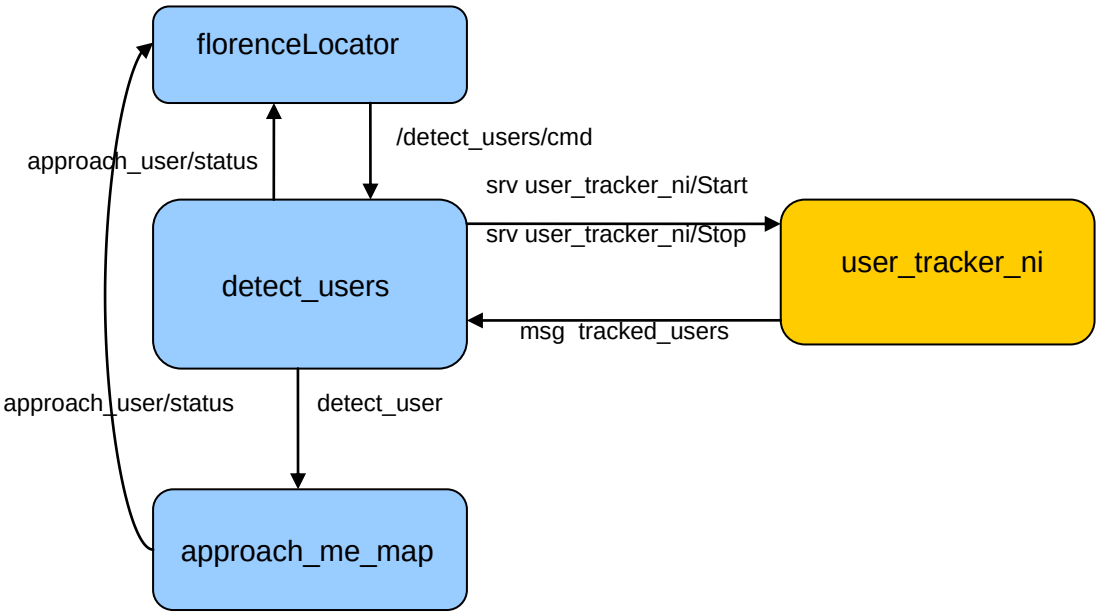

Figure 3-7. FlorenceLocator Detect Users and ApproachMe_Map components

3.9.3 Working hypothesis and current limitations

As it is described above, once the component gets the start commands it tries to detect a user. There is a specific ROS component is charge of detecting users, user_tracker_ni, which uses OpenNI. We have described that openNI needs to be stopped to detect real users, but there are still more things to do to prevent from getting false detections.

OpenNI stores all user's information for several seconds despite it is not tracking any new user. This is a problem in our case because once the robot is stopped and gets updated information about users, it gets several user's information (real and non-

real).This is implemented by erasing all openNI instances. To do that, a service provided by user_tracker_ni is used.

When the robot starts making 45º step, user_tracker_ni Stop service is called, and once the robot has made the step, user_tracker_ni Start service is called.

The Detect Users component is continuously sending a feedback message to the FlorenceLocator such that it can relay this status on request.

3.9.4 Developers Guide

This component is developed under C++ programming language. The most important functions are the following ones:

Call to the user_tracker_ni service in order to START tracking users

void call_user_tracker()

Callback to listen detect_user/cmd topic. It contains an activation command to start detecting users

void detect_user_cmdCB (const std_msgs::Bool::ConstPtr &msg)

Callback to listen tracked_users topic. It contains information regarding detected users

void tracked_usersCB(user_tracker_ni::TrackedUsers msg)

Main function

int detect_users(void)

Step by step, look if any user is detected

for (int i = 0; i < NUMBER_STEPS; i++)

Move until next step point is achieved

if(fabs(angle_to_compare_-robot_orientation_) <= ANGLE_PRECISION && !any_user_detected_)

Indentify Users

identify_users();

If a user is detected, send the information:

user_approach_position_.data.push_back(user_coordenate_x_);
user_approach_position_.data.push_back(user_coordenate_y_);
user_approach_pub_.publish(user_approach_position_);

3.10 ApproachMe-Map Component

In section 4.2, the ApproachMe Component will be described, despite its similar name this component is very different. This component uses a map for navigation while the other only uses odometry information to approach to the user.

The API of this component is described below. Basically, it continuously sends a topic containing the coordinate where the robot is required to move. Optionally, it is possible to specify a safe distance between robot and user, which the robot will take into account to stop safely.

3.10.1 Interface

Table 13. Interface for approach_me_map Component.

approach_me_map interface		
Subscribed Topics		**Brief description**
Name	**Data type**	
/detected_user	detect_users/DetectedUser	This message contains the activation command and the position of the detected user bool command float32 user_position_x float32 user_position_y
Published Topics		**Brief description**
Name	**Data type**	
/approach_user/status	std_msgs/String	This message is used to communicate with the florenceLocator component Three different messages are defined: ACTIVE: the robot is running the action SUCCEEDED: the robot achieved the user safely ABORTED: the robot could not achieve the user
Parameters		**Brief description**
Name	**Data type**	
safe_distance	Float	It is used to define the safe distance from the user where the robot has to step DEFAULT_SAFE_DISTANCE: 0.5

3.10.2 Implementation details

Figure 3.5 shows how it is implemented within the FlorenceLocator architecture.

3.10.3 Working hypothesis and current limitations

The approach_me_map component uses ROS navigation stack to navigate around the area. The behaviour is similar to sending a goal to the robot with the peculiarity that stops with a safe distance. In case that the robot cannot achieve the safe distance, it calculates two more new places. Eventually, if the robot cannot achieve the a safe distance in three trial, it sends an ABORTED command and stops.

3.10.4 Developers Guide

This component is developed under C++ programming language. The most important functions are described bellow:

Callback function that receives where the user has been detected, and a command to start the action.

void detected_userCB(detect_users::DetectedUser msg)

Calculate a safe distance to stop

bool calculate_pose(int angle_step)

Send the robot to the user. It tries to send the user to the goal_trial position. This point is a safe point where the robot could go. Sometimes this safe distance is not feasible to the robot to achieve and a new safe position is calculated. Maximun, three trials are done.

int start_approach_me(int goal_trial)

Main function. It is charge of calling previously mentioned function, and publishing the status of the action.

int spin (void)

4 Robot enablers

This section provides a description of enablers that have been developed to ease the use of the robot. The build on top of the components described in the previous chapter.

The components described are the following:

- The FollowMe component that enables to make the robot follow a person.
- The ApproachMe component that deals with the robot positioning with respect to a person location.

4.1 The FollowMe component

The interface for the robotic enabler "*FollowMe*" is implemented in the *follow_me* package and uses the concept of Action, as provided within the ROS framework (within the *actionlib* library), which has been previously described within the document.

4.1.1 Interface

The structure of the interface is imposed by the *actionlib* package. The specification of an action requires the definition of the parameters for the *goal*, *feedback* and *result* messages, as explained in section 1.2.1.

The table below describes the format of the messages specific to the *FollowMe* action.

Table 54. Interface of the FollowMe action 1.

FollowMe action interface		
Subscribed Topics		**Brief description**
Name	**Data type**	

`/follow_me/goal`	`follow_me/FollowMeActionGoal`	The FollowMe action goal specifies which user needs to be followed by providing the approximate coordinates of the user with respect to the robot. Optionally, the maximum linear and angular speeds for the robot's movement, as well as the desired distance at which the robot will follow the user may be specified.
		`float32 x`
		`float32 y`
		`float32 max_lin_speed`
		`float32 max_ang_speed`
		`float32 desired_follow_distance`

`/follow_me/cancel`	`Actionlib_msgs/GoalID`	Enable to cancel the action during its execution
		`Time stamp`
		`String id`

Published Topics		**Brief description**
Name	**Data type**	

/follow_me/feedback	follow_me/FollowMeActionFeedback	The feedback message contains the current position of the user being followed, expressed in the robot frame. float32 x float32 y
/follow_me/result	follow_me/FollowMeActionResult	The result message contains the position of the user at the moment the action finishes, expressed in the robot frame. float32 x float32 y
/follow_me/status	actionlib_msgs/GoalStatusArray	The status of the action is periodically sent by the action server. Note that the information provided is not specific to the action, as provided by the feedback topic. It describes more generally the action's state (pending, active, pre-empted…) Header header GoalStatus[] status_list

The ROS messages, including those used in actions, are handled via classes that are automatically generated by the ROS build toolchain by reading the definition provided in the message description text file. These automatically generated classes have the particularity of initialising all its members to null values (e.g. 0 value for numerical members, null vectors, null strings). This supposes that the default values that the action server will receive if the user leaves some of the parameters in the goal message unspecified will be all zeros. If zero is a feasible value for one of those parameters, then the default value for it will be zero. If otherwise zero is not a feasible value (e.g. *max_linear_speed*) then a non-zero default value will be set inside the *follow_me_node.*

This solution is not ideal and may be updated in future implementations to include ROS parameter interface to control those parameters, instead of sending them as part of the *goal* message.

4.1.2 Implementation Details

The previous section has illustrated the interface provided to request a follow-user action. The internal organization of the "*FollowMe*" enabler is now described with more detail.

The implementation of this enabler is done inside the *follow_me* package, in the *follow_me_node.cpp* source file. Here a ROS node is implemented which makes use of an instance of the SimpleActionServer class provided by the *actionlib* package to provide the interface. This node also depends on another external node, the *user_tracker*.

user_tracker: is a *generic* node in charge of the estimation of the user position with respect to the robot frame. It is defined as *generic* since the functionality it provides can be used for other enablers or services, not being something specific to the enabler being described.

The use of the *user_tracker* node is transparent to the user of the enabler, who should only interact with this enabler via the action server included in *follow_me_node.*

The following figure also displays some components that are not developed within the FollowMe enabler. They correspond to third-part elements used by our enablers. OpenNI is a third-party library (ROS only provides a wrapper that eases installation and dependency resolution) which is used by the *user_tracker* package. *turtlebot_node* is the low_level node that interfaces with the robot hardware. Finally, *hokuyo_node* is the ROS package enabling to connect to the laser rangefinder, and get access to the scans.

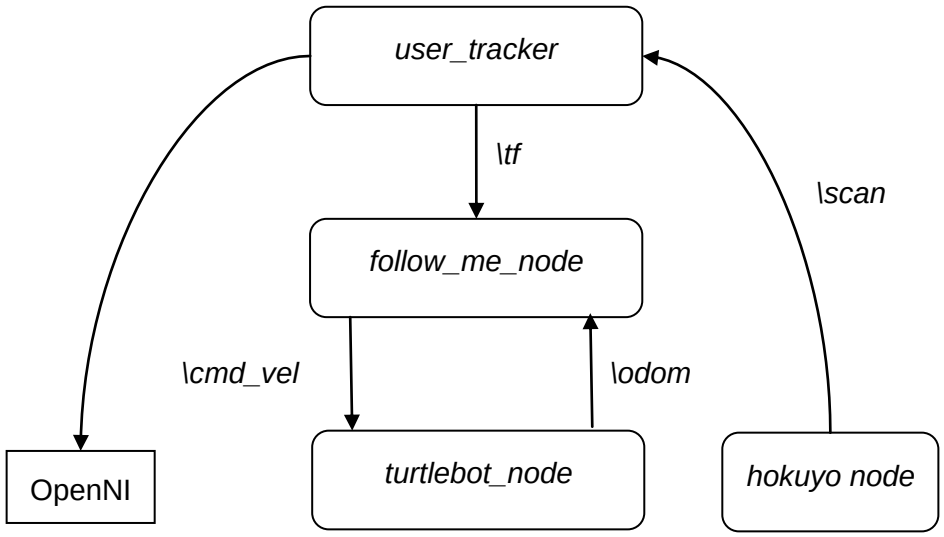

Figure 4-1: Internal implementation of the follow me robotic enabler

The OpenNI library enables to access the NITE middleware, which can grab the depth data provided by the Kinect sensor and provides user tracking functionality (such as person detection, skeleton pose and others).

The node *user-tracker* uses the person detection provided by OpenNI. When it detects a person in the sensor field of view, an ID is assigned, which is persistent as long as the user is being correctly tracked. Nevertheless, OpenNI does not ensure ID continuity when a person is lost and re-detected later on.

Internally, the *user_tracker* node monitors the person identification provided by OpenNI and overlays this with colour histogram based user descriptors, allowing to recognize a person that has been already tracked and to assign to this person a truly persistent ID.

The *user_tracker* node then takes the information provided by OpenNI and uses it to publish geometric transformations that define the user position relative to the robot using *tf* (which is the system used by ROS to represent the geometric representation of the world).

The *follow_me_node* node then retrieves the position of the user which the robot needs to follow using *tf* as well. Using the current person position relative to the robot, the node calculates the next control command to send to the robot using a spring-like proportional control law. Specifically, the linear v and angular ω velocities are computed as:

$$v = -k_v(d - d^*)$$
$$\omega = -k_\omega \theta$$

Where *d* is the distance in between the robot and the person, and θ the relative angle defining the person position with respect to the robot (in this case, it is supposed that the robot is asked to be aligned with the direction of the person, i.e. $\theta^* = 0$).

4.1.3 Working hypothesis and current limitations

Different assumptions upon which the correct behaviour of the *FollowMe* enabler depends can be identified:

- The person needs to be previously detected and visible within the robot sensor field of view before launching the action. The enabler, as a low-level service, does not have the capability to recognize the person to be followed. It is thus supposed that such operation is realized by the upper layers. When the action starts, the person position provided by the client is compared with the person detection information provided by OpenNI. If no person is detected, or if the error in location is considered too large, then the action directly finishes and returns an error message.
- The enabler is efficient as long as the person stays within the field of the view of the robot sensor. In case of having temporary tracking losses (during few sensor acquisitions), the enablers does not provide any active plan scheme to recover the visual contact with the person. If the perception fails to resume the person tracking within a given amount of time, the action will be finished with a specific return error.
- The obstacle avoidance is not handled within the FollowMe enabler. The obstacle avoidance enabler is supposed to run in parallel, controlling and correcting the motion commands sent by this enabler.

The first implementation of the *followMe* action presents furthermore some limitations that could be overcome within the next version delivery:

- The tracking process is currently based on the use of the Kinect sensor data through the OpenNI library. The information provided by the laser sensor, is so far not used. Its use would enable to enlarge the robot field of perception, and thus limit the risk of person loss.
- As previously mentioned, the basic action does not provide any strategy to search for the person that has been recently lost. Some simple procedures could be implemented to recover from this situation, similar to the use of recovery behaviours by the navigation stack.
- The use of the obstacle avoidance as an external process may produce more local temporary person loss. The execution of the obstacle avoidance within the FollowMe enabler could enable to better handle the situations in which the robot has to make a compromise in between these two objectives (follow the user, and avoid obstacles).
- The motion strategy is a purely reactive shadow-like behaviour. Depending on the user's feedback, the following strategy could be adapted. Furthermore, the control law gains are currently hardcoded and cannot be changed at runtime. Future implementations may include a mechanism to tune these gains.
- Finally, the control module could be embedded within the Navigation stack, to improve the overall navigation consistency.

4.1.4 Technology

This component is implemented using the actionlib ROS package, within C++. The vision-based person detection is provided by the components provided with the Kinect sensor.

4.1.5 Developers Guide

The use of this enabler through an action client is illustrated by a draft client provided with *the follow_me* ROS package in the *follow_me_client.cpp* file. Due to its length, the full code will not be included here, but it can be obtained by checking out a copy from the project repository. Only some particularly interesting snippets will be discussed in this section.

To start, after initializing the ROS node as usual, an instance of *SimpleActionClient* must be created with

```
actionlib::SimpleActionClient<follow_me::FollowMeAction> ac("follow_me",
true);
```

This call will create the client, specifying the name of the action and that it needs to create a thread to manage the callbacks.

Afterwards, when the position of the user that the robot needs to follow is acquired, the action can be triggered by filling a *goal* message and sending it with

```
follow_me::FollowMeGoal goal;
goal.x = user_position_x;
goal.y = user_position.y;
ac.sendGoal(goal, &doneCB, &activeCB, &feedbackCB);
```

The last three parameters in the *sendGoal()* function are optional references to callback functions that will be called during the action lifespan. The *doneCB*, and *activeCB* callbacks will be called once each, when the action finishes and starts respectively. The *feedbackCB* callback function will be called each time the action server publishes a feedback message, which in the case of the FollowMe server will be once every controller loop.

After requesting the action, the node that requested it can continue doing other tasks, and may cancel the action at any time by calling

```
ac.cancelGoal();.
```

4.2 Filter User Tracker Component

This component is used to improve follow_me functionality.

When a user is being tracked using openNI, an ID is assigned to the user. This ID is used to identify the user as long as it is detected. When the user is lost openNI keeps user's information for some seconds. This can be useful for some application but it is not useful for FollowMe component described in 4.1 section.

This component removes lost users as soon as they are lost.

4.2.1 Interface

The communication interface for Filter User Tracker Component is detailed in the following table.

Table 15. Interface for Filter_User_Tracker

filter_user_tracker interface		
Subscribed Topics		**Brief description**
Name	**Data type**	
/tracked_users	user_tracker_ni/ TrackedUsers	int32[] user_ids int32[] user_ids_ni int32 num_users_tracked
Published Topics		**Brief description**
Name	**Data type**	
/filtered_tracked_users	user_tracker_ni/ TrackedUsers	int32[] user_ids int32[] user_ids_ni int32 num_users_tracked

4.2.2 Implementation details

The component user_tracker_ni is in charge of tracking the user's position. It is similar to openni_tracker integrated within openNI framework, but it does not compute the user's skeleton and only publishes the centre of mass of the detected user.

Filter_user_tracker component gets the user's information and as soon as it gets it checks if the user's distances are within Kinect's range of vision. Otherwise the component is removed from the user's list.

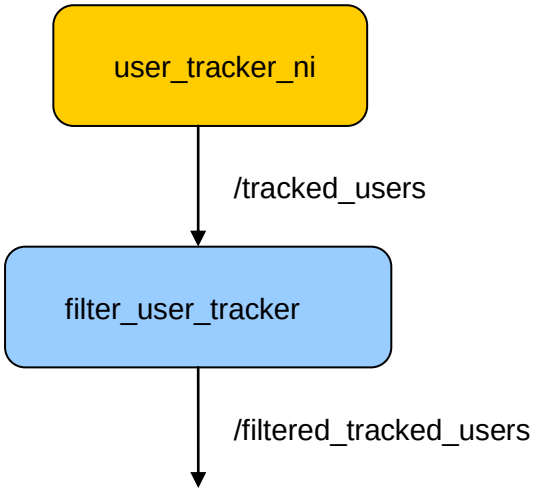

Figure 4-2. Filter User Tracker Component

4.2.3 Working hypothesis and current limitations

When a user is lost, openNI keeps the user ID available for some seconds. During this time despite the ID is accessible, its value is not desirable. It has been experimentally checked that the distance assigned to the ID is less than Kinect's minimum range.

At this moment, the only way to detect immediately if a user has been lost is checking if the user's distances provided by openNI correspond to the working range of the Kinect.

4.2.4 Developers Guide

This component is developed under C++ programming language. The most important functions are described bellow:

Callback function that is executed when a topic with tracked_users information arrives

void tracked_usersCB(const user_tracker_ni::TrackedUsersConstPtr &tracked_users_msg)

Filtering process is done in the following function. It is called by the previous callback function

void filter()

It compares all users information and if the users information is closer than the kinect's range ***ZERO_RADOIUS***, it filters.

4.3 The ApproachMe component

The robotic enabler *ApproachMe* takes care of bringing the robot closer to the person, suitable for having dialogs. This enabler is implemented using the Action paradigm provided by *actionlib* like it is done with the FollowMe. Nevertheless, two differences are to be highlighted:

- Contrary to the *FollowMe* enabler, the goal defined by *ApproachMe* corresponds to a final positioning with respect to the person. In this sense, the action can be concluded with a 'SUCCESS' status, when the robot reaches the desired relative position and orientation.
- In the *FollowMe* enabler, the goal is defined by the current position of the person to be followed, plus some optional parameters inclusing the robot-person distance. *ApproachMe* defines, in addition to this distance, the desired orientation of the person with respect to the robot and of the robot with respect to the person.

4.3.1 Interface

Following the description scheme provided for the *FollowMe,* the parameters involved for the communication with the *ApproachMe* server are defined in the following table.

Table 66. Interface of component ApproachMe.

FollowMe action interface		
Subscribed Topics		Brief description
Name	Data type	
`/approach_me/goal`	`approach_me/ApproachMeActionGoal`	The ApproachMe action goal specifies the position of the person to be approached, as well as the parameters that define the position with respect to the person that the robot has to reach (distance, and their respective orientations). Optionally, the maximum linear and angular speeds for the robot's movement may be specified. `float32 x`

		float32 y float32 desired_distance float32 desired_p2r_orientation float32 desired_r2p_orientation float32 max_lin_speed float32 max_ang_speed
/approach_me/cancel	Actionlib_msgs/GoalID	Enable to cancel the action during its execution Time stamp String id
Published Topics		**Brief description**
Name	**Data type**	
/approach_me/feedback	approach_me/ApproachMeActionFeedback	The feedback message contains the current position of the user being followed, expressed in the robot frame. float32 x float32 y
/approach_me/result	approach_me/ApproachMeActionResult	The result message contains the position of the user at the moment the action finishes, expressed in the robot frame. float32 x float32 y
/approach_me/status	actionlib_msgs/GoalStatusArray	The status of the action is periodically sent by the action server Header header GoalStatus[] status_list

As has already been explained in the description of the FollowMe enabler, the default values that the *goal* message will contain zeros for all arguments that are not explicitly assigned by the caller. This will be treated as a zero value in case this value is feasible (e.g. desired orientation), or will be substituted for a different than zero value where this value is not correct (e.g. max_linear_speed).

4.3.2 Implementation Details

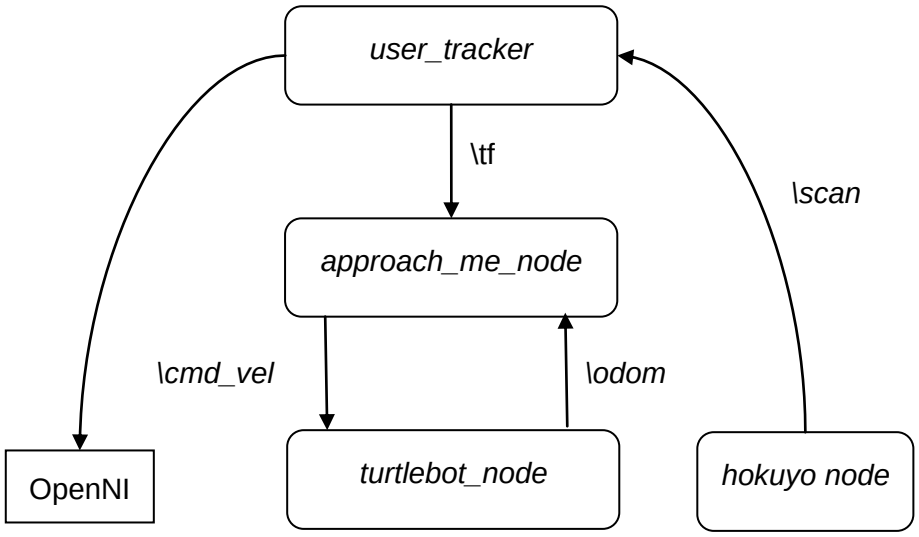

Figure 4-3: Internal implementation of the approach me robotic enabler

As illustrated within the Figure above, the implementation of the *ApproachMe* enabler is structured similarly to the *FollowMe* one. As a first implementation, the approach strategy will consist in (i) bringing the robot at the desired position with respect to the user, and then (ii) rotate to reach the specified orientation with respect to the person. During the first movement phase, the person is likely to get out of the field of view of the Kinect sensor, making thus impossible to use the OpenNI module to keep tracking the person. To avoid a temporary loss of the person position, the data provided by the Hokuyo sensor is also used to maintain the tracking, and to resume the visual contact when it is possible.

4.3.3 Working hypothesis and current limitations

The *ApproachMe* enabler presents the same assumptions that the *FollowMe* enabler:

- the person needs to be detected and visible within the robot sensor field of view before launching the action.
- The obstacle avoidance is not handled within the ApproachMe enabler.
- This enabler does not have an internal representation of the environment, and is not using such information. It is supposed that the surrounding space is free of obstacles.
- The orientation of the person is not verified nor updated during the operation. It is supposed that the orientation provided at the launch of the action is correct. This orientation (relative to the robot) is updated using only odometry information.

The first implementation of the ApproachMe will also have some limitations:

- It is supposed that the person is static during the whole motion procedure. This makes that the current state of the enabler is quite similar to the GoTo-X-Y enabler.

4.3.4 Technology

The interface to this component is implemented using the action paradigm provided by the *actionlib* ROS package. The vision-based person detection is provided by the components provided with the Kinect sensor and the node will fuse this visual detection with the information gathered from the laser scans. The positioning of the robot relies on the information provided by the odometry.

4.3.5 Developers Guide

The use of this enabler is similar to the one illustrated for the FollowMe service. A complete example is provided within the ApproachMe ROS package within the svn repository.

5 SUMMARY

In this deliverable, we have presented how to use the ROS architecture to develop and run all robot components needed in the Florence project. Some of these components come from the ROS framework and they have been just used.

The original Turtlebot robot's structure has been adapted and how the physical structure has been implemented has been explained.

Most of the components that have been reused are related to the navigation part. An adaption of their configuration was required in order to be able to work on top of those components. This means that for all the modules their working principles have been learned and they have been configured to work properly with the Florence robot. For some of them additional improvements have been made, like the addition of a laser to the only Kinect based navigation that came out of the box with the Turtlebot.

In addition to the configuration, some components have been designed to improve the general behaviour of the robot. An example of this is the smooth motion of the robot compared to the original ROS Turtlebot package.

Some other improvement for the robot enablers also has been detailed, making them more robust and getting safer behaviours, like improvements to the initialization and localization functions.

Finally, a user manual is described. It has been written to have a short and easy to read document where everything needed to run the Turtlebot base Florence platform is explained. The user manual can be found in ANNEX A.

6 References

[Martin2010] Martin, E. C. et al., ”Initial Florence System Architecture”, Florence Project Deliverable D2.2, 2010

[OpenCV] http://opencv.willowgarage.com/wiki/

[Quigley09] Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Wheeler, R. and Ng A. Y. ”ROS: an open-source Robot Operating System”, ICRA Workshop on Open Source Software, 2009

[ROSaction] ROS Actionlib package detailed description,

http://www.ros.org/wiki/actionlib/DetailedDescription

[ROScostmap] ROS costmap_2d package description,

http://www.ros.org/wiki/costmap_2d

7 ANNEX A: USER MANUAL

The aim of this manual is to explain shortly how to set-up everything including compilation of all florence ROS packages and execution of them. As an example a specific package created for the user test is explained

Some of the sections have been explained previously, but we consider it is worth to collect them again and provide a quick view of all steps.

7.1 Software installation

- **Ubuntu installation**

  Ubuntu 11.10 32bit version has to be installed

  Download web: http://releases.ubuntu.com/11.10/ubuntu-11.10-desktop-i386.iso

- **Turtlebot** specific software:

Installation instructions: http://ros.org/wiki/Robots/TurtleBot

Setup your sources.list

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu
oneiric main" > /etc/apt/sources.list.d/ros-latest.list'
```

Setup your keys

```
wget http://packages.ros.org/ros.key -O - | sudo apt-key
add -
```

Installation

```
sudo apt-get update
```

```
sudo apt-get install ros-electric-turtlebot-robot
```

Enviroment setup

```
echo "source /opt/ros/electric/setup.bash" >>
~/.bashrc
```

```
. ~/.bashrc
```

- **Florence** ROS packages installation instructions:

  Update the repository. There is a one ROS package containing all ROS Florence package dependency information: ..wp2/trunk/robot/**user_test**

  Compile packages:

```
rosmake --rosdep-install user_test
```

- **Laser** driver installation instruction

```
sudo apt-get install ros-electric-
laser-drivers
```

7.2 Calibration

There are different gyro versions on the market, and the range of them can vary, so its specifications have to be checked to determine the range. Before launching calibration process, a configuration file has to be edited.

..wp2/trunk/robot/turtlebot_control/config/**gyro_yourcompany.launch**

The calibration process consists in three steps:

1.- The calibration has to be done using the kinect as a laser

```
roslaunch turtlebot_control
calibration_control.launch
```

This launch file consists on two different launch files. It has been seen that sometimes there is a synchronization problem with both and it is necessary to launch them separately.

```
roslaunch minimal_calibration.launch
```

```
roslaunch turtlebot_calibration calibrate.launch
```

Once the turtlebot_calibration.launch has finished, look at the output in the screen:

[INFO] [WallTime: ...] Multiply the 'turtlebot_node/**gyro_scale_correction**' parameter with **1.171401**

[INFO] [WallTime: ... Multiply the 'turtlebot_node/**odom_angular_scale_correction**' parameter with **1.015922**

========== REQUIRED process [turtlebot_calibration-14] has died!

process has finished cleanly.

log file: /home/turtleflo32/.ros/log/44f8a89e-d49b-11e1-b9a0-7ce9d31428de/turtlebot_calibration-14*.log

Initiating shutdown!

- Copy the values into:
  ..wp2/trunk/robot/turtlebot_control/launch/**minimal_nvf_control.launch**
  And modify if necessary: **gyro_measurement_range**

2.- It is highly recommended to **repeat** the calibration process up to get a stable values, near to 1.00

3.- Make the values persistent the following file has to be edited:
..wp2/trunk/robot/turtlebot_control/config/gyro_yourcompany.launch

4.- It is still recommended to check the calibration parameters in the real scenario.

7.3 Build a map

Original ROS webpage tutorial:

http://ros.org/wiki/turtlebot_navigation/Tutorials/Build%20a%20map%20with%20SLAM

There are three ways to create a map, depending the hardware configuration: using the kinect, using the laser upside, and laser downside. These instructions are specific to the Florence robot configuration: **laser downside**

Step by step:

1.- On the TurtleBot or Workstation.

- As the laser is inverted, a patch in slam_gmapping has to be applied. A new patched package has been added to the repository. Compile the package

```
rosmake slam_gmapping_laser_inverted
```

2.- On the TurtleBot

```
roslaunch turtlebot_control minimal_control.launch
```

3.- On your Workstation

- Run the gmapping

```
rosrun gmapping_laser_inverted slam_gmapping
```

```
rosrun rviz rviz -d `rospack find turtlebot_navigation`/nav_rviz.vcg
```

4.- On the TurtleBot or Workstation

- Drive the robot around:

```
rosrun turtlebot_teleop turtlebot_teleop_key
```

- Some tips driving the robot

    - Drive the robot following walls
    - Once the robot has achieved a corner make in-place rotation of the robot
    - Avoid driving the robot randomly in the house

5.- Save the map TurtleBot or Workstation

- Save the map to file:

```
rosrun map_server map_saver -f /tmp/my_map
```

It is also possible to save the map in a different place. To do that change the "–f" location

Sometimes it takes some seconds to save the map. This is because ROS environment does not get accessible the map. If nothing happens, move the robot with the keyboard control

6.- Edit the map with a Image Editor

The objectives of editing are mainly two: **reduce the size** of the map and **clean**.

Meaning of the colors:

- White: open space
- Grey: unknown space
- Black: obstacles

Resize:

- Select the map
- Click in "Image/Crop to Selection"
- Optional: rotate the image

Clean

- paint open space areas with white color.
- Paint unknown areas with grey
- Make the walls and fixed obstacles well defined

Save the map

Save the new pgm file

- When the map is created two different files are automatically created:
  name_of_the_map.pgm
  name_of_the_map.yaml

After editing the map, if the name of the pgm file is changed, the yaml file needs to be edited. The name of the new map has to be defined into this file.

7.- Make the map persistent
- Copy the map and yaml file into: ..wp2/trunk/robot/turtlebot_control/map

7.4 First time configuration

1.- Add gyro information

edit ..wp2/trunk/robot/turtlebot_control/launch/**minimal_control.launch**

```
<!--- Gyro and Odometry correction -->
<include file="$(find turtlebot_control)/config/gyro_yourcompany.launch" />
```

2.- Add the map

edit ..wp2/trunk/robot/user_test/launch/**florence_user_test.launch**

```
<!-- Run the map server -->
<arg name="map_file" default="$(find turtlebot_control)/map/yourmap.yaml"/>
<node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)" />
```

7.5 Starting the Florence system

```
roslaunch user_test florence_user_test.launch
```