

# Proyecto Final de Máster Universitario en Automática, Robótica y Telemática

Programación de robot móvil con manipulador para  
el sector del comercio en entorno ROS

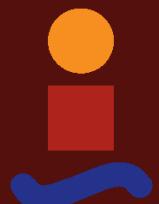
Autor: Francisco Javier Buenavida Durán

Tutores: Dr. Miguel Ángel Ridaó Carlini

Dr. Carlos Bordons Alba

**Dep. de Ingeniería de Sistemas y Automática  
Escuela Técnica Superior de Ingeniería  
Universidad de Sevilla**

Sevilla, 2017





Proyecto Fin de Máster Universitario  
en Automática, Robótica y Telemática

# **Programación de robot móvil con manipulador para el sector del comercio en entorno ROS**

Autor:

Francisco Javier Buenavida Durán

Tutores:

Dr. Miguel Ángel Ridaó Carlini

Dr. Carlos Bordons Alba

Dep. de Ingeniería de Sistemas y Automática  
Escuela Técnica Superior de Ingeniería  
Universidad de Sevilla  
Sevilla, 2017



Proyecto Fin de Máster: Programación de robot móvil con manipulador para el sector del comercio en entorno  
ROS

Autor: Francisco Javier Buenavida Durán

Tutores: Dr. Miguel Ángel Ridaó Carlini  
Dr. Carlos Bordons Alba

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

# Agradecimientos

---

Ha pasado más de un año desde que empecé este proyecto en los laboratorios del departamento de sistemas y automática. Todo comenzó cuando mi compañero del curso de máster, Diego López, que dio a conocer una vacante de becario en el departamento. Es por ello que en primer lugar me gustaría agradecerle a él, no solo que me diera a conocer este proyecto, sino por toda la ayuda que me brindó a lo largo del curso. Asimismo, aprovecho para agradecerle al resto de mis compañeros del master por las ayudas y los buenos momentos que pasamos durante este pasado año y medio.

Tampoco hubiera sido posible comenzar esta aventura sin la confianza que mis dos tutores, Miguel Ángel Rida y Carlos Bordons, depositaron en mí. Gracias a este trabajo pude incorporarme por primera vez a un grupo de desarrollo y trabajar para un cliente real como Comerzzia. Aprendí, junto a mis compañeros, cómo afrontar los problemas que se iban planteando gracias a los consejos que tanto Miguel Ángel como Carlos nos proporcionaba semana tras semana. Además, pude adquirir experiencia, un requisito tan necesario y demandado hoy en día para poder incorporarse al mercado laboral, gracias a ello puedo escribir estas palabras hoy desde Graz, Austria. Es por todo ello y más que me gustaría daros a ambos mi más sincero agradecimiento.

También tengo que agradecer a mis compañeros del proyecto, Gonzalo Hernández, Adrián Fernández y Adrián Cardona, toda la ayuda y apoyo que recibí, así como todos esos buenos momentos que tuvimos cada mañana en el laboratorio “peleándonos” con el robot. Fue un placer haber compartido este proyecto con vosotros y espero que os vaya bien en el futuro allí donde vayáis.

Por supuesto, como no puede ser de otra forma, quiero agradecerle a mi familia el apoyo incondicional que he recibido siempre por parte de todos, mis padres, mis hermanas, primos, tíos, abuelos, etc. Gracias por estar siempre ahí, tanto en los buenos como en los malos momentos. También quiero agradecerle a la otra familia, la familia que se elige, mis amigos, por todos los buenos momentos que he pasado durante el transcurso de este periodo, hubiera sido mucho más difícil sin ellos, seguro.

Por último, pero no menos importante, tengo que agradecer a mi amiga, compañera y consejera, mi novia María Fernández, por todas las ayudas, consejos y apoyo que me ha dado y me sigue dando desde que la conocí, tanto para el máster, como para la elaboración del presente documento o para la vida misma. Muchas gracias de corazón por todo y aunque la distancia nos separe hoy encontraremos la manera de estar juntos mañana.

*Francisco Javier Buenavida*

*Graz, 2017*



# Resumen

---

En la actualidad la investigación para el uso de robots en la vida cotidiana está a la orden del día. La utilización de tecnología para la asistencia de compras se está convirtiendo cada vez en un uso más habitual, como la utilización de aplicaciones para hacer la lista de la compra o para directamente comprar desde casa sin desplazarse a la tienda o supermercado. Es por ello que actualmente se está investigando en incorporar robots que asisten al cliente, mejorando así la experiencia de compra.

Dentro de este contexto se sitúa este trabajo, que forma parte del proyecto de I+D+i “*Smart Omnichannel Retail*” de Comerzzia realizado en colaboración con AICIA y financiado a través de la Corporación Tecnológica de Andalucía (CTA). En este proyecto se emplea un robot *Turtlebot2* que incorpora una cámara 3D, y al que se le añadió un brazo manipulador. Como plataforma software base donde desarrollar las diferentes tareas que realiza el robot, se ha utilizado el framework o infraestructura digital de código abierto y uso libre, *Robot Operating System* (ROS).

Este documento recoge el trabajo realizado en materia de visión y manipulación para el desarrollo de dicho robot en el sector *retail*, es decir, describe el software desarrollado para poder localizar los productos en la tienda, así como poder detectarlos y recogerlos con el brazo robótico. Estas etapas fueron desarrolladas en conjunto con otras series de etapas que están fuera del alcance de este documento pero que forman parte del proyecto I+D+i de Comerzzia. Los resultados obtenidos fueron satisfactorios respecto a la etapa de localización, mientras que, para la detección y manipulación de objetos, pese a lograrse su cometido, se precisa de mejores tecnologías e investigaciones futuras.



# **Abstract**

---

Nowadays, there are many research programs evaluating the use of robots in people's daily lives. With the increased use of technology for purchasing assistance, it is becoming more common to use technology in the retail sector. From applications that make grocery lists, to making purchases at home without having to set foot in the store, using technology in this way is becoming more of a part of our daily lives. Consequently, research is being done to incorporate robots that assist the customer in order to improve their shopping experience.

It is within this context that this thesis is set, and which form part of the Comerzzia's R&D project called "Smart Omnichannel Retail". This project was carried out in collaboration with AICIA and financed through the Technological Corporation of Andalusia (CTA). A Turtlebot2 robot, which has a 3D camera and an added manipulative arm, was used in order to investigate the possibilities of a robot in a retail environment. In addition, the open source framework called ROS was used to develop the different tasks that the robot can complete.

This document details all the work done in terms of vision and manipulation for a social robot in the retail sector. In other words, it describes the software that was developed in order to detect and pick up certain products within a shop. These stages were developed in conjunction with other series of stages that are outside the scope of this research, but which form part of the Comerzzia's R&D project. The results obtained from the location tests were satisfactory, whereas, for the detection and manipulation of objects, better technologies and future researches are needed.

# Índice

---

<b>Agradecimientos</b>	<b>vii</b>
<b>Resumen</b>	<b>ix</b>
<b>Abstract</b>	<b>xi</b>
<b>Índice</b>	<b>xii</b>
<b>Índice de Figuras</b>	<b>xiv</b>
<b>1 Introducción</b>	<b>1</b>
1.1 <i>Objetivo</i>	2
1.2 <i>Planteamiento del proyecto</i>	3
1.2.1 Interfaz y comunicaciones inalámbricas	4
1.2.2 Mapeado, localización y navegación	5
1.2.3 Visión y manipulación de productos	5
1.2.4 Interfaz cliente y comunicación con la plataforma Smart Omnichannel Retail	5
1.3 <i>Organización de la memoria</i>	5
<b>2 ROS</b>	<b>7</b>
2.1 <i>Historia</i>	8
2.2 <i>Arquitectura y conceptos</i>	8
2.2.1 Sistema de ficheros	9
2.2.2 Grafos de procesos	10
2.2.3 Comunidad de ROS	12
2.3 <i>Herramientas y librerías</i>	12
2.3.1 Gazebo	12
2.3.2 RVIZ	13
2.3.3 TF	14
2.3.4 URDF	15
<b>3 Turtlebot</b>	<b>17</b>
3.1 <i>Base móvil</i>	18
3.2 <i>Controlador</i>	19
3.3 <i>Escáner láser 2D</i>	20
3.4 <i>Manipulador</i>	21
3.5 <i>Sensor 3D</i>	22
<b>4 Localización de productos</b>	<b>25</b>
4.1 <i>Tecnologías empleadas</i>	26
4.1.1 Principio de funcionamiento de la Kinect	26
4.1.2 OpenCV	29
4.1.3 Point Coud Library	29
4.1.4 Zbar	30
4.2 <i>Arquitectura del software</i>	30
4.3 <i>Descripción del código</i>	31
4.3.1 Mensajes y servicios	31

4.3.2	Nodo barcode_detector	33
4.3.3	Nodo position_detector	38
<b>4.4</b>	<i>Resultados</i>	<b>49</b>
<b>5</b>	<b>Detección y manipulación de productos</b>	<b>53</b>
<b>5.1</b>	<i>Tecnologías empleadas</i>	<b>54</b>
5.1.1	ORK	54
5.1.2	Tabletop	55
5.1.3	Actionlib	57
5.1.4	Pxpincer	58
<b>5.2</b>	<i>Arquitectura del software</i>	<b>58</b>
<b>5.3</b>	<i>Descripción del código</i>	<b>60</b>
5.3.1	Mensajes y servicios	60
5.3.2	Nodo find_and_pick	62
5.3.3	Nodo move	74
<b>5.4</b>	<i>Resultados</i>	<b>83</b>
<b>6</b>	<b>Conclusiones</b>	<b>87</b>
<b>6.1</b>	<i>Líneas de investigación y desarrollos futuros</i>	<b>89</b>
<b>Referencias</b>		<b>91</b>

# ÍNDICE DE FIGURAS

---

Figura 1 – Robot NAO (izquierda) y Turtlebot2 (derecha) empleados por Comerzzia.	2
Figura 2 – Estructura empleada en ROS para controlar el Turtlebot.	4
Figura 3 – Interfaz desarrollada para la proyecto.	4
Figura 4 – Estructura del sistema de ficheros en ROS	9
Figura 5 – Arquitectura de los grafos de procesos en ROS	10
Figura 6 – Ejemplo del paradigma publicador/subscriptor que permite ROS.	11
Figura 7 – Simulación de un Rover lunar en Gazebo.	13
Figura 8 – Ejemplo de la interfaz de Rviz.	13
Figura 9 – Árbol de transformadas TF	14
Figura 10 – Esquema de los componentes de un fichero URDF.	15
Figura 11 – Hardware del Turtlebot 2.	17
Figura 12 – Base móvil iClebo Kobuki.	18
Figura 13 – Intel Nuc NUC5I5RYH.	20
Figura 14 – Láser LIDAR Hokuyo.	20
Figura 15 – PhantomX Pincher.	21
Figura 16 – Sensor Kinect. Primera versión.	22
Figura 17 – Sensor Kinect. Segunda versión.	23
Figura 18 – Patrón de puntos emitidos por la Kinect sobre una superficie plana.	26
Figura 19 – Triangulacion empleada por la Kinect para obtener la distancia a un punto.	27
Figura 20 – Nube de puntos obtenida con la Kinect.	28
Figura 21 – Grafo de procesos del paquete “Localización de productos”.	30
Figura 22 – Diagrama de flujo del servicio “Localiza productos”	39
Figura 23 – Imagen mostrada por barcode_detector tras detectar códigos.	49
Figura 24 – Ejemplo de imagen y nube de puntos superpuestas en Rviz.	50
Figura 25 – Ejemplo sobre entorno real de barcode_detector y position_detector.	52
Figura 26 – Reconocimiento de una superficie mediante segmentación.	56
Figura 27 – Modelo 3D de una lata de refresco en la base de datos.	56
Figura 28 – Representación del objeto detectado usando Tabletop.	57
Figura 29 – Simulación del brazo PhantomX Pincher mediante el paquete pxpincher_ros.	58
Figura 30 - Grafo de procesos de la etapa de “detección y manipulacion de objetos”.	59
Figura 31 – Diagrama de flujo del servidor “ <i>find_and_pick</i> ”	63
Figura 32 – Modelo 3D de un bolígrafo generado para la base de datos del paquete ORK.	83

Figura 33 – Escenario de pruebas de la detección y manipulación de objetos visto en Rviz.	84
Figura 34 – Imagen real del brazo incorporado en el Turtlebot2 atrapando un bolígrafo.	84
Figura 35 – Ejemplo de integración de una cámara RGB externa y una Kinect.	89
Figura 36 – Visualización en Rviz de un octomap.	90



# 1 INTRODUCCIÓN

---

*En el futuro, estoy seguro de que habrá muchos más robots en todos los aspectos de la vida. Si le dijeran a la gente en 1985 que en 25 años tendrían computadoras en su cocina, no tendría ningún sentido para ellos.*

- Rodney Brooks -

Como apoyo a la plataforma de fidelización omnicanal de tiendas desarrollada por Comerzzia, una de las empresas filiales de Tier1 y pioneras en tecnologías del comercio al por mayor o retail, se comenzó el desarrollo de un “robot social” que permitiera interactuar con los clientes de manera personalizada. Este proyecto de I+D+i, denominado en su conjunto “Smart Omnichannel Retail”, fue financiado a través de la Corporación Tecnológica de Andalucía (CTA) y en el que participaron dos grupos de investigación de la Universidad de Sevilla. [1]

El desarrollo del robot estuvo a cargo del departamento de ingeniería de sistemas y automática de la Escuela Técnica Superior de Ingenieros de Sevilla a través de la Asociación de Investigación y Cooperación Industrial de Andalucía (AICIA). El robot de Comerzzia consiste en dos unidades con funcionalidades claramente separadas, pero que se integran dentro del mismo sistema (Figura 1).

En primer lugar, el popular robot humanoide NAO de *Aldebaran Robotics* [2] se programó como una interfaz de usuario que permita reconocer al cliente, conectarse con el software desarrollado por Comerzzia, y ofrecer mediante voz y gestos, una experiencia personalizada y amigable al cliente en el momento que entre en la tienda. Por otro lado, una vez éste ha seleccionado su pedido e informado a la plataforma de Comerzzia, se le asigna un robot móvil que permita asistirle y guiarle por la tienda hacia los productos previamente seleccionados.



Figura 1 – Robot NAO (izquierda) y Turtlebot2 (derecha) empleados por Comerzzia.

## 1.1 Objetivo

El proyecto tiene como objetivo desarrollar y programar un robot móvil capaz de guiarnos por la tienda en función de la lista de productos que el cliente desee comprar. Este dispositivo permite realizar de manera mucho más sencilla, rápida y eficaz la tarea de realizar la compra. Para ello, el robot debe contar con una serie de características que le permita cumplir con las siguientes especificaciones:

- Mapeo: el robot será capaz de hacer un mapeo del entorno.
- Localización en interiores: para ser capaz de posicionar al robot con objeto de guiar al usuario por dentro de la tienda.
- Interacción simple con el usuario-cliente: para poder especificar la lista de la compra y para permitir un control con acciones simples sobre el robot.
- Planificación de trayectorias: calcula la ruta a seguir para llegar al producto en función de ciertos parámetros seleccionables.
- Velocidad de paseo de un humano.
- Comportamiento reactivo: para no colisionar con objetos móviles del entorno.
- Escalable a distintos escenarios.
- Comunicación inalámbrica vía WiFi.
- Lectura de etiquetas, tanto RFID como código de barras.
- Efecto final: debe disponer de un manipulador final para poder interactuar con distintos objetos o, al menos, se estudiarán las distintas posibilidades de su inclusión.
- Capacidad de carga, para poder asistir al operario en el transporte del pedido.

Sin embargo, estas especificaciones iniciales tenían como finalidad ser una guía de ruta para el proyecto, siendo alguna de ellas descartadas por mejores soluciones según avanzó el proyecto. Además, se introdujo finalmente un brazo manipulador con el fin de estudiar la posibilidad de ser utilizado en una tienda virtual o entorno cerrado al cliente (*Darkstore*).

Respecto al robot humanoide NAO, aunque éste consta de múltiples funciones de manera nativa desarrolladas por *Aldebaran* (como andar o reconocimiento facial), en este proyecto el robot se empleó básicamente como una interfaz humanoide para informar al cliente de aquellos datos relevantes para la tienda, mediante comando de voz y gesticulando como si de una persona real se tratara. La idea es mostrar información personalizada al usuario de manera amigable y divertida.

## 1.2 Planteamiento del proyecto

Para la realización del proyecto en primera instancia se investigó las posibles plataformas, tanto hardware como software, que permitieran desarrollar y cumplir con las especificaciones. Tras hacer un estudio del estado del arte de la robótica y el hardware disponible en el mercado, se decidió emplear como software base el *framework*, o infraestructura digital, ROS (*Robot Operating System*) ejecutado sobre un sistema operativo *Linux*, mientras que como robot móvil se empleó el “*Turtlebot 2*”.

Los motivos principales que llevaron a la elección de ROS como software fueron, entre otros, que se trata de una herramienta de uso libre, y que tiene detrás una amplia comunidad investigadora alrededor de todo el mundo que le da soporte. Dicha comunidad le nutre constantemente con nuevos paquetes software que contienen desde drivers para nuevas plataformas robóticas, hasta implementaciones de los algoritmos más avanzados. Así, por tanto, se evita tener que crear un sistema operativo robótico robusto desde cero, y poder apoyarse en una gran colección de herramientas, librerías y protocolos que facilitara el desarrollo. Respecto a la plataforma robótica se pensó en el *Turtlebot 2* ya que se trata de un robot de bajo coste y personalizable, que cuenta con un gran apoyo por parte de la comunidad de ROS.

Al tratarse de un software distribuido, ROS permite descentralizar los procesos dividiendo un problema tan complejo como es la programación de un robot en pequeños procesos o nodos que se encargan de una función específica. Además, no se limita a un solo sistema, pudiendo distribuir el software en diferentes plataformas. Por este motivo, para facilitar las cosas al usuario y evitar cargar de procesos al robot, se decidió dividir el sistema en un PC Central o “*Master*” y el *Turtlebot*, ambos conectado mediante una red inalámbrica (Figura 2). A su vez, el PC Master se conectará con los servidores de Comerzzia para poder comunicarse con el cliente. Aunque todos los procesos se encuentran dentro de la misma red, cada hardware se encargará de procesos específicos.

De esta manera, el *Turtlebot* ejecutará solo los procesos referidos a los sensores y actuadores que posee, y el PC master se encargará de ejecutar otros procesos como la interfaz, la gestión del sistema y los distintos algoritmos desarrollados para manejar el robot.

Una vez acotado el problema, desde el departamento de ingeniería de sistemas y automática se decidió dividir el proyecto en distintas áreas según el problema a tratar, de tal manera que diferentes personas trabajarán en diferentes partes del proyecto que a priori eran independientes, pero que en última instancia formarían un conjunto capaz de satisfacer las especificaciones. Así, por tanto, se puede distinguir cuatro áreas independientes:

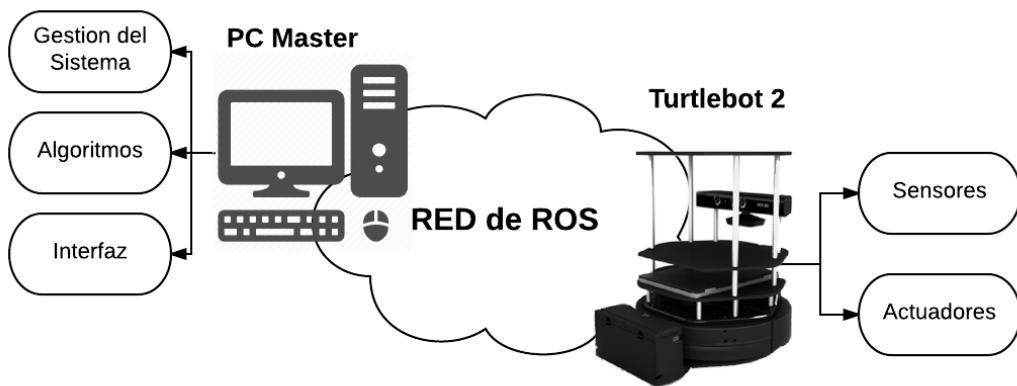


Figura 2 – Estructura empleada en ROS para controlar el Turtlebot.

### 1.2.1 Interfaz y comunicaciones inalámbricas

Para la interfaz de usuario, la idea era desarrollar una interfaz humana a través de un PC conectado remotamente al robot, en la que se mostrase el estado e información del robot móvil (mapa, lectura de los sensores, estado de las baterías ...), así como poder dar órdenes al mismo de manera inalámbrica. En la Figura 3 se muestra a la izquierda las posibles órdenes al robot, mientras que a la derecha se observa el mapa y el robot localizado en él.

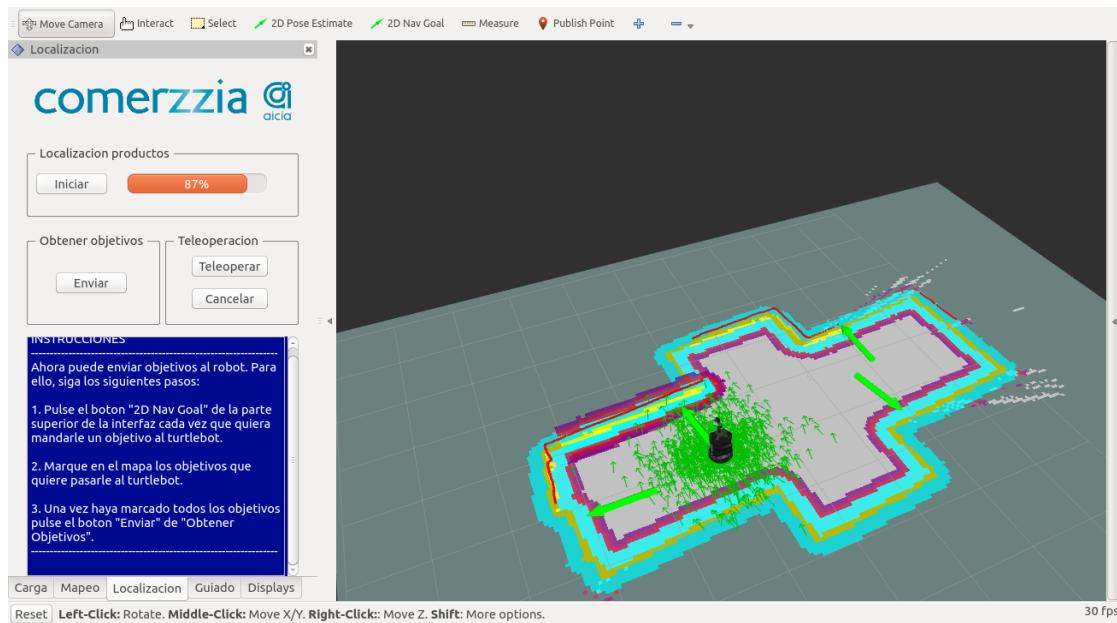


Figura 3 – Interfaz desarrollada para la proyecto.

Esta interfaz debe, pues, poder arrancar, dar órdenes y supervisar al robot en todo momento de manera intuitiva por un operario.

### 1.2.2 Mapeado, localización y navegación

En este caso, el objetivo era dotar al robot de autonomía a través de la técnica de mapeado y localización simultánea, también conocida por sus siglas en inglés SLAM (*Simultaneous Localization And Mapping*), utilizando distintos sensores que permitan tanto obtener una lectura del entorno (láser de barrido 2D, cámaras, etc.), como información interna del robot (odometría).

Una vez obtenido el mapa y estar localizado en él, el robot debe poder navegar a los distintos objetivos marcados por el usuario (Figura 3).

### 1.2.3 Visión y manipulación de productos

Gracias a la incorporación de una cámara 3D y un brazo robótico manipulador, el robot debe tener la posibilidad de identificar ciertos objetos o etiquetas, y localizar su ubicación entorno.

En primera instancia, el robot puede identificar etiquetas y su ubicación en el mapa como posibles objetivos para ser emplear en la navegación o guiado de los clientes. Asimismo, esta característica sirve además para detectar en el entorno, aquellos objetos que puedan ser manipulados mediante un brazo robótico.

### 1.2.4 Interfaz cliente y comunicación con la plataforma Smart Omnichannel Retail

Por último, aunque esta parte no está englobada en el contexto del Turtlebot 2 ni emplea ROS, forma parte del proyecto completo y se decidió desarrollar junto con el resto.

En esta ocasión, era necesario desarrollar una aplicación en el robot NAO que le permita identificar a los clientes mediante tarjetas de fidelización y conectarse remotamente a la plataforma “*Smart Omnichannel Retail*” obteniendo así información del usuario de manera personalizada. Dicha información se muestra mediante comandos de voz, y utilizando además la capacidad nativa del robot para gesticular.

## 1.3 Organización de la memoria

Aunque el proyecto en su totalidad engloba diferentes áreas de trabajo, el propósito principal de este documento de Trabajo Fin de Máster es documentar el trabajo realizado en materia de visión y manipulación del entorno, empleando para ello el sensor 3D *Kinect* y un brazo robótico, así como el sistema operativo ROS.

La memoria comenzará introduciendo los objetivos y describiendo el trabajo realizado, así como el contexto en el que se desarrolla, los problemas planteados y sus soluciones. A continuación, el segundo capítulo introduce al lector dentro del mundo que rodea al sistema operativo ROS. Aquí se explicará su filosofía, componentes y arquitectura, además de una breve explicación de su historia, de sus conceptos, estructuras, librerías y herramientas. Todo ello permitirá justificar su elección por encima de los otros sistemas.

Seguidamente, se describirá la parte del hardware utilizado, la plataforma robótica *Turtlebot2*. Se explicarán uno a uno los distintos componentes que lo forman, como también los periféricos añadidos que han sido empleados en el marco de este proyecto. En cambio, en los dos siguientes apartados, se pretende dar sentido al código utilizado, explicando cada parte del mismo y mostrando los resultados obtenidos.

Por último, se expondrán las conclusiones del proyecto. Además, se ha añadido un apartado de líneas futuras de investigación, para exponer las posibles mejoras a desarrollar para obtener unos resultados más satisfactorios, reduciendo las limitaciones y fallos encontrados.

## 2 ROS

---

*“Beethoven era un buen compositor porque utilizaba ideas nuevas en combinación con ideas antiguas. Nadie, ni siquiera Beethoven podría inventar la música desde cero. Es igual con la informática”*

- Richard Stallman -

**R**OS son las siglas en inglés de *Robot Operating System*. Se trata de un *framework* (o infraestructura digital) flexible de código abierto que consta de una colección de herramientas, bibliotecas y convenios, y que tienen como objetivo simplificar la tarea de desarrollar software para robots. Cuenta con una comunidad de expertos que proporciona algoritmos en estado de arte mediante una licencia permisiva BSD y que está siendo adaptado ampliamente por centros de investigación, educativos y por la industria. [3]

ROS proporciona los servicios típicos de un sistema operativo como la abstracción de hardware, control de dispositivos a bajo nivel, implementación de utilidades comunes, paso de mensajes y gestión de paquetes. Al mismo tiempo, incorpora una serie de herramientas y librerías para obtener, compilar, escribir y ejecutar código mediante varios ordenadores. [4]

Crear un software robusto y de propósito general es una tarea difícil y de mucha complejidad. Desde la perspectiva del robot, los problemas que parecen triviales para cualquier ser humano varían enormemente según la tarea a realizar y entorno donde se encuentre. Lidiar con todas las posibles variantes que se pueden llegar a dar es sumamente complejo para cualquier persona, laboratorio de investigación o institución que espere desarrollarlo por sí solo.

Como resultado, ROS fue construido desde cero para fomentar el desarrollo de software para robots de manera colaborativa. Por ejemplo, un laboratorio podría tener expertos en la cartografía de interiores, y podría contribuir con un sistema avanzado de primer nivel para la producción de mapas. En otro lugar, un grupo podría tener expertos en el uso de mapas para navegar, y a su vez, otro grupo podría haber descubierto un nuevo enfoque respecto a la visión por computador que funciona bien para reconocer objetos pequeños. Es por ello que ROS fue diseñado específicamente para grupos como estos, con el fin de poder colaborar y que todos puedan construir usando el trabajo de todos, evitando el coste que supone re-inventar contantemente el mismo software por parte de distintos grupos o personas. [5]

## 2.1 Historia

ROS es un gran proyecto que cuenta con diversos antecedentes y colaboradores. Fueron muchos los integrantes de la comunidad de investigación robótica que sintieron la necesidad de crear un *framework* de software libre.

A mediados de los años 2000, varios proyectos de la *Universidad de Stanford* crearon prototipos software de sistemas flexibles y dinámicos, para programas como el del *STanford AI Robot* (STAIR) y *Personal Robots* (PR). Pero no fue hasta 2008 cuando *Willow Garage*, una conocida incubadora de empresas y laboratorios de investigación robótica, proporcionó los recursos necesarios para ampliar el *framework* mucho más y crear un software mejor testeado y robusto. Tras ello, el proyecto fue impulsado a su vez por un sinnúmero de investigadores que contribuyeron con su tiempo y experiencia al núcleo de ROS y sus paquetes de software fundamentales. Desde febrero de 2013 hasta la actualidad, ROS está siendo mantenido por fundación robótica de código abierto (OSRF – *Open Source Robotics Foundation* [6]).

ROS, de principio a fin, fue desarrollado como software libre bajo la licencia permisiva de código abierto *BSD*, es por ello que poco a poco fue siendo ampliamente utilizado por la comunidad de investigación robótica. Aunque en un principio parecía más simple para todos sus creadores que ROS hubiera nacido en un mismo servidor donde añadir código poco a poco, debido a que fue desarrollado por múltiples instituciones para múltiples robots no pudo ser contemplado en un inicio. Pero irónicamente, con los años, esto ha significado una de las grandes fortalezas de ROS, ya que cualquier grupo o persona puede iniciar y desarrollar desde su propio repositorio código para ROS, manteniendo la propiedad y control total sobre el mismo. Si optan por hacer que su repositorio sea visible públicamente, pueden recibir el reconocimiento y el crédito que merecen por sus logros, así como poder beneficiarse de la retroalimentación o *feedback* de la comunidad, como en el resto de proyectos de software de código abierto. [5]

Actualmente el ecosistema de ROS consta de decenas de miles de usuarios en todo el mundo, trabajando en dominios que van desde proyectos personales como *hobby*, hasta grandes sistemas de automatización industrial. [5]

## 2.2 Arquitectura y conceptos

Para poder entender ROS completamente, es necesario distinguir entre tres secciones o niveles de conceptos [7]:

- **Sistema de ficheros:** Este nivel indica la estructura de carpetas, como está formado, y el número mínimo de archivos que ROS necesita para trabajar.
- **Grafo de procesos:** La estructura de grafos muestra la comunicación entre los distintos procesos del sistema. ROS está basado en una arquitectura de grafos, es decir, cuenta con un número de nodos independientes que puede comunicarse con el resto de nodos a través del modelo publicador / subscriptor.
- **Comunidad de ROS:** Por último, en este nivel se encuentran las distintas herramientas y conceptos utilizados por la comunidad para compartir conocimientos, algoritmos y código con cualquier desarrollador. Gracias a este nivel ROS ha podido y puede seguir creciendo rápidamente.

## 2.2.1 Sistema de ficheros

Similar a un sistema operativo, un programa en ROS se divide en carpetas que contienen ficheros dependiendo de su funcionalidad (Figura 4):

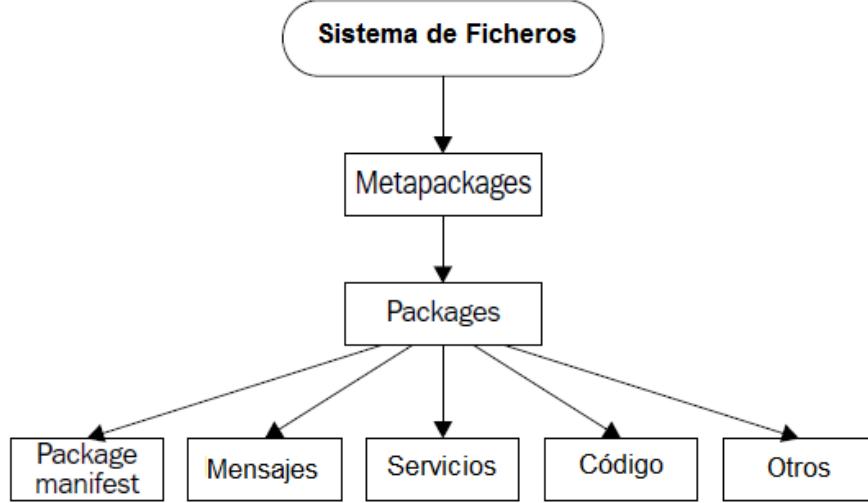


Figura 4 – Estructura del sistema de ficheros en ROS

Los conceptos más importantes de este nivel son [7]:

- **Packages:** Los paquetes forman el nivel atómico de ROS. Un paquete tiene la estructura y el contenido mínimos para crear un programa dentro de ROS. Estos paquetes pueden contener procesos (nodos), librerías, scripts, archivos de configuración (Makefiles), etc.
- **Package manifests:** Un manifiesto de paquete proporciona información sobre un paquete, licencias, dependencias, indicadores de compilación, etc. Estos se gestionan con un archivo denominado package.xml que está contenido dentro de un paquete.
- **Metapackages:** Los metapaquetes permiten simplificar ROS, agrupando varios tipos de paquetes en un solo grupo denominado “Stacks”. En ROS existen muchos ejemplos de Metapackages, por ejemplo, los de navegación.
- **Metapackages manifest:** En este caso, los manifiestos de metapaquetes (package.xml) son similares a los de los paquetes normales, pero con una etiqueta de exportación XML. Además, contiene ciertas restricciones en su estructura.
- **Mensajes (msg):** Un mensaje es la información o datos que un proceso (nodos) envía a otro proceso. Gracias a ello distintos procesos escritos en distintos lenguajes de programación pueden comunicarse. ROS tiene muchos tipos de mensajes estándar. Las descripciones de los mensajes se almacenan en “*my\_package/msg/MyMessageType.msg*”.
- **Servicios (srv):** los servicios definen las estructuras de datos o mensajes que se envían distintos procesos usando el modelo de petición y respuesta. Las descripciones de los servicios se almacenan en “*my\_package/srv/MyServiceType.msg*”.

- **Código (src):** Aquí es donde se encuentra el código fuente de los programas de ROS.

### 2.2.2 Grafos de procesos

ROS se basa en una arquitectura de grafos o red distribuida donde los distintos procesos o nodos pueden interactuar con otros procesos de la misma red, transmitiendo y/o recibiendo datos. Se trata de una infraestructura muy flexible y adaptable a diversas necesidades, pudiendo localizar los nodos en distintos sistemas según sea necesario (Figura 5).

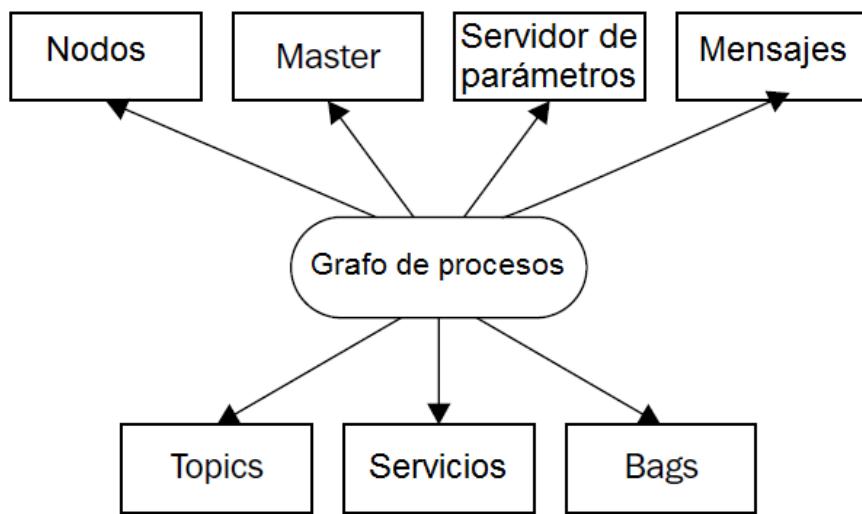


Figura 5 – Arquitectura de los grafos de procesos en ROS

Los conceptos más importantes de este nivel son [7]:

- **Master:** El ROS Master se encarga de proporcionar el registro de nombres y el servicio de búsqueda del resto de nodos, además se encargar de conectarlos. Su existencia es necesaria ya que, sin éste, los nodos no podrían encontrarse entre ellos, intercambiar mensajes o invocar servicios. En un sistema distribuido, el master se encontrará en un equipo pudiendo ejecutar nodos en este u otros equipos.
- **Nodos:** Son los procesos que realizan los cálculos. Debido a la filosofía modular de ROS, normalmente, un sistema tendrá muchos nodos para controlar diferentes funciones específicas del robot. Por lo general, es mejor tener muchos nodos que proporcionan una sola funcionalidad, en lugar de tener un gran nodo que se ocupe de todo el sistema.
- **Mensajes:** La comunicación entre nodos se realiza a través del uso de mensajes. Un mensaje contiene datos que envían información a otros nodos. ROS tiene definido muchos tipos de mensajes, aunque también se puede desarrollar un tipo de mensaje propio usando mensajes estándar.

- **Topics:** Es el nombre que se usa para identificar y poder encaminar los distintos mensajes dentro de la red ROS. Cuando un nodo envía un mensaje, se dice que el nodo está publicando en un topic. Dichos mensajes son encaminados mediante un sistema de transporte que se basa en el modelo publicador/subscriptor. Un nodo puede suscribirse a un topic y no es necesario que exista el nodo que está publicando en dicho topic, de esta manera se consigue desacoplar al publicador del subscriptor. Asimismo, múltiples nodos pueden hacer uso de un topic publicando o suscribiéndose a éste.
- **Servicios:** Cuando se publica en un topic, se están enviando datos de entre múltiples nodos sin ningún tipo de control. Sin embargo, cuando se necesita una solicitud o respuesta de un nodo es necesario emplear servicios. Los servicios, por tanto, proporcionan el modelo de petición y respuesta que a menudo se necesita en este tipo de sistemas distribuidos.
- **Servidor de parámetros:** Es un diccionario compartido multivariable que es accesible a través de la red. Los nodos hacen uso del servidor para almacenar y recibir parámetros en tiempo real, pudiendo incluso cambiar su funcionamiento. Actualmente es parte del Master.
- **Bags:** Se trata de un mecanismo para almacenar y reproducir los datos enviados a través de los mensajes en ROS, como datos de sensores que son difíciles de recopilar, pero que son especialmente útiles cuando se necesita desarrollar o probar algoritmos.

En la Figura 6 se muestra un ejemplo básico del funcionamiento de los topics y los nodos en ROS. En el ejemplo existen tres nodos, cada uno con una funcionalidad, y dos topics con diferentes tipos de datos. El primer nodo se trata de un sensor de odometría de la rueda, que publica en el topic llamado “*odom*” la posición en XYZ del robot. Esta información es recibida por el nodo planificador de caminos, que en base a dichos datos genera un mensaje en el topic “*cmd\_vel*”, que se trata del comando de velocidad y giro para seguir una trayectoria. Por último, el nodo encargado de controlar el motor del robot, procederá a mover las ruedas del robot según la información recibida por el planificador de caminos.

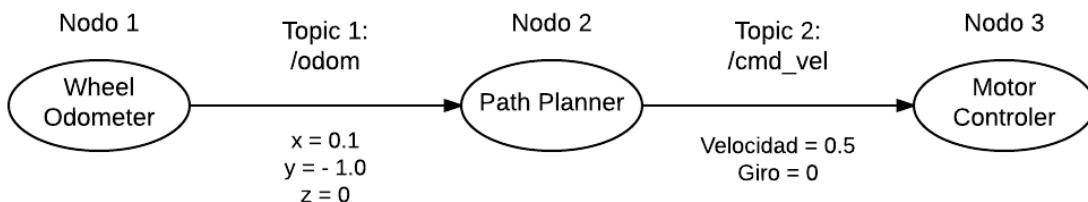


Figura 6 – Ejemplo del paradigma publicador/subscriptor que permite ROS.

Este tipo de grafos puede ser consultados en ROS en todo momento mediante la herramienta “*rqt\_graph*”, con la cual se pueden visualizar tanto los nodos que se están ejecutando como los topics que los unen.

### 2.2.3 Comunidad de ROS

En este último nivel se puede encontrar los recursos en ROS que permite a la comunidad intercambiar software y conocimientos. Estos recursos incluyen [7]:

- **Distribuciones:** Las distribuciones en ROS son colecciones de distintas versiones de metapaquetes que se pueden instalar. Las distribuciones ROS desempeñan un papel similar a las distribuciones de Linux, es decir, facilitan la instalación de una colección de software de una misma versión.
- **Repositorios:** ROS se basa en una red de repositorios donde diferentes instituciones pueden almacenar el código desarrollado en ROS.
- **La enciclopedia de ROS:** La enciclopedia de ROS o “ROS Wiki” es el principal foro de información sobre ROS. Es de libre acceso y por tanto cualquier persona puede formar parte de ésta aportando documentación, haciendo correcciones o actualizaciones, escribir tutoriales, etc.
- **Bug Tickets System:** Este recurso está pensado para comunicar problemas o bugs encontrados, así como proponer una nueva función a un determinado código o paquete.
- **Listas de correo:** La lista de correo de usuarios de ROS es el principal canal de comunicación sobre actualizaciones de ROS, así como un foro para hacer preguntas sobre ROS.
- **ROS Answer:** Este recurso es utilizado por los usuarios para hacer preguntas sobre ROS.
- **Blog:** En el blog de ROS [3] se encuentran las últimas actualizaciones sobre noticias de la comunidad, como eventos, nuevos paquetes o funcionalidades, videos, etc.

## 2.3 Herramientas y librerías

ROS cuenta con múltiples herramientas desde simuladores hasta herramientas que permiten facilitar la compresión, desarrollo y el manejo de los distintos procesos de ROS. A continuación, se mostrarán aquellas herramientas más útiles y utilizadas en ROS durante el desarrollo del proyecto.

### 2.3.1 Gazebo

La simulación es una herramienta esencial tanto en robótica como en otros campos. Gazebo es una herramienta que ofrece la posibilidad de simular en 3D con precisión y eficiencia robots tanto en entornos interiores como exteriores. También permite generar los datos de realimentación de los sensores y ofrecer interacciones físicas entre el robot y objetos del entorno gracias a su motor de físicas. Además, cuenta con una comunidad muy activa donde se puede obtener diversos modelos de robots, mundos virtuales y objetos [8].

Gazebo puede sincronizarse con ROS de manera que los nodos del robot emulado reciben y publican información en los distintos topics como si de un robot real se tratara. De esta manera permite probar rápidamente tanto algoritmos como robots diseñado, y realizar pruebas de regresión utilizando entornos realistas.

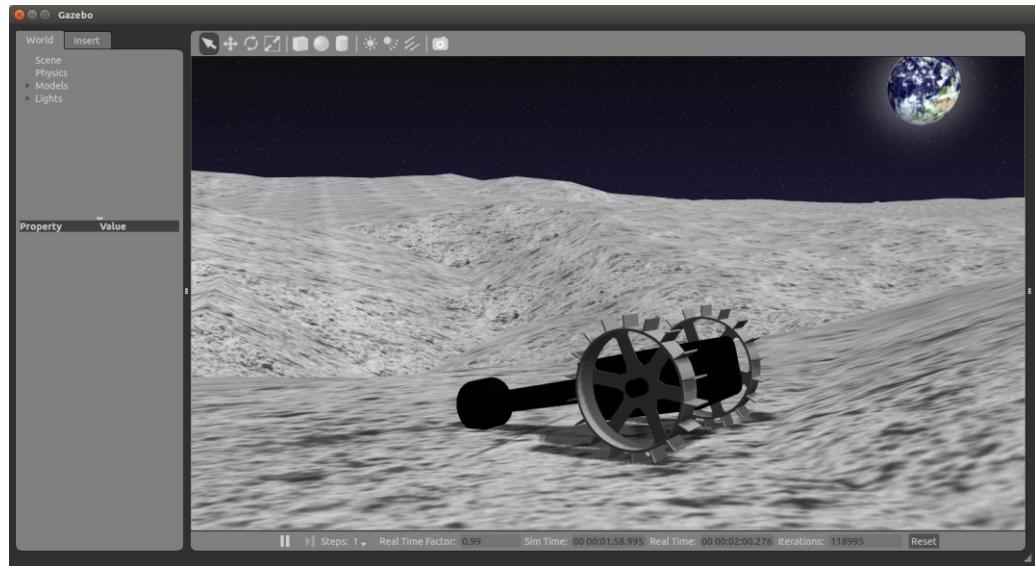


Figura 7 – Simulación de un Rover lunar en Gazebo.

### 2.3.2 RVIZ

Rviz es la abreviación de *ROS Visualization*, y se trata de una de las herramientas más completas y útiles de ROS ya que permite mostrar en 3D de manera gráfica, intuitiva y en tiempo real, el estado y la información de un sistema en ROS [9].

Entre sus múltiples funciones, se encuentra la posibilidad de ver la lectura de los sensores como láseres o cámaras de visión estereoscópica, visualizar la trayectoria que va a seguir dentro de un mapa, mostrar la configuración de los distintos marcos de referencia o coordenada del robot, etc. Además, esta herramienta cuenta con una interfaz modular y personalizable, pudiendo mover y programar paneles, así como crear nuevos plugins que permitan añadir nuevas funcionalidades.

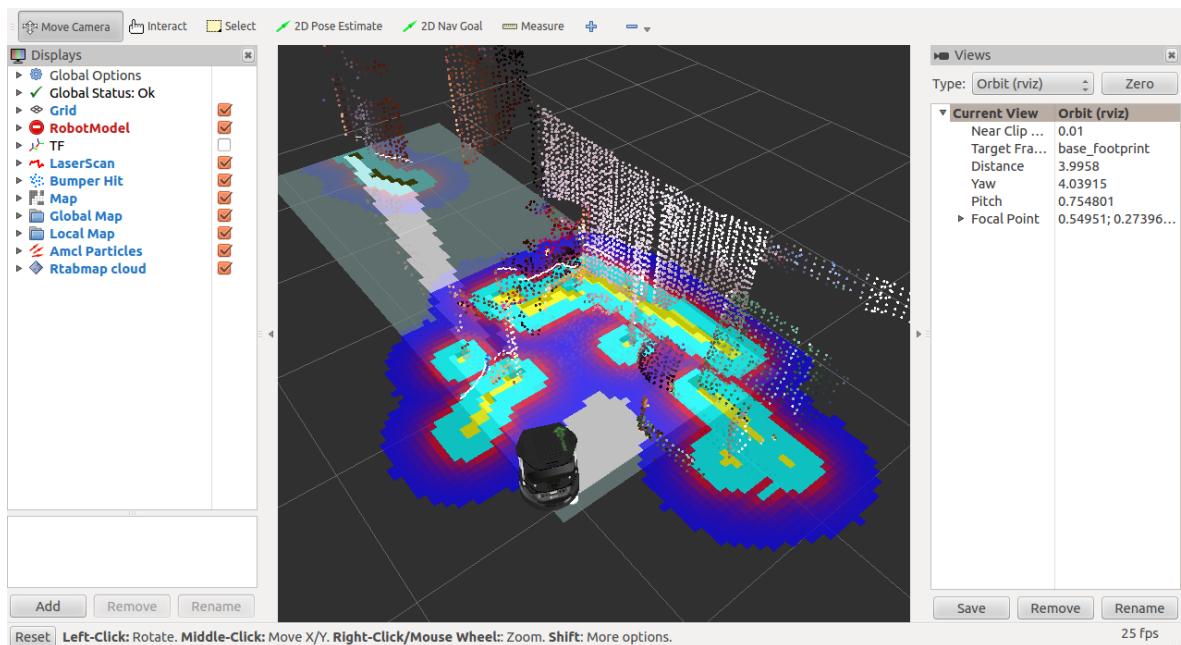


Figura 8 – Ejemplo de la interfaz de Rviz.

Rviz usa el sistema de transformadas (TF) para transformar los datos desde el marco o *frame* de coordenada recibido (por ejemplo, el del láser) a un marco de referencia global, como podría ser el mapa. Hay dos marcos de referencia que son importantes de conocer en el visualizador y que se puede modificar en todo momento para obtener la visualización adecuada:

- **Fixed frame:** Es el más importante de los dos. El *fixed frame* o marco fijo es el marco de referencia utilizado como *frame* global. Por norma general, este marco se suele asociar al mapa o “mundo” para poder visualizar los datos de manera correcta. En caso de no existir, también se suele utilizar el de odometría.
- **Target frame:** es el sistema de coordenadas que sirve como referencia a la cámara o vista del visualizador, pudiendo tener distintas vistas e ir cambiando dependiendo de la perspectiva que se desee visualizar. Este *frame* generalmente se suele situar en la base del robot.

### 2.3.3 TF

TF son las siglas en inglés de marco de transformadas (*Transform frames*) y es una de las librerías fundamentales de ROS. Permite coordinar y transformar los distintos marcos de referencia o ejes de coordenadas del robot respecto a un punto de referencia global y entre sí, a lo largo del tiempo. TF mantiene la relación entre los ejes mediante una estructura en forma de árbol almacenada en el tiempo, permitiendo al usuario transformar puntos o vectores entre dos marcos de referencia en cualquier instante de tiempo [10].

Además, TF ofrece una serie de herramientas o aplicaciones que facilitan al usuario la visualización del estado de las transformadas, como *tf monitor* o *tf echo* entre otras. En la Figura 9 se muestra un árbol de transformada haciendo uso de la herramienta *rqt\_tf\_tree*. En él se puede observar los distintos *frames* del sistema y sus relaciones.

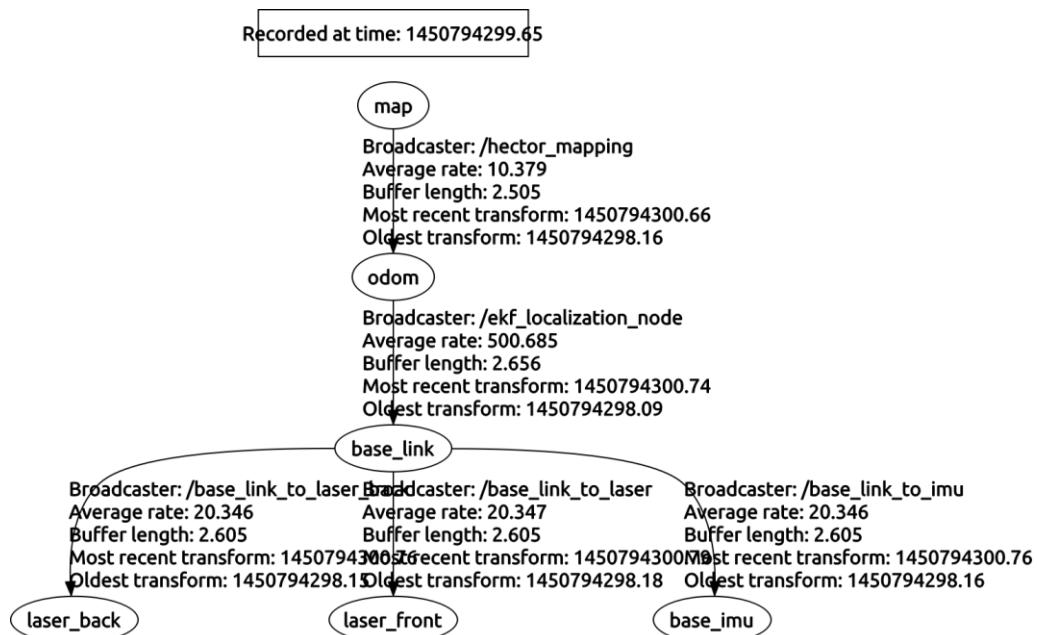


Figura 9 – Árbol de transformadas TF

### 2.3.4 URDF

URDF (*Unified Robot Description Format*) es una herramienta de modelado de robots. Es la encargada de especificar las propiedades del robot, como sus dimensiones, número de articulaciones, parámetros físicos, etc. [11]

Esta información viene descrita en forma de árbol en un archivo XML (Figura 10), distinguiéndose dos componentes principales necesarios para la construcción de la cadena cinemática de un robot, los eslabones (*Links*) y las articulaciones (*Joints*).

- **Link:** Los eslabones o enlaces describen la parte física rígida del robot, como la masa, geometría o la inercia, así como los componentes visuales necesarios para mostrar el robot en herramientas como Rviz
- **Joint:** Las articulaciones o uniones indican la relación entre los distintos eslabones del robot. Describen además la cinemática y dinámica de cada articulación, además de especificar los límites de colisión del robot.

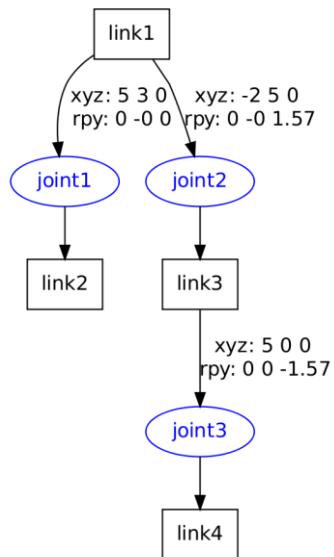


Figura 10 – Esquema de los componentes de un fichero URDF.



## 3 TURTLEBOT

*"¿Qué somos las personas sino máquinas muy evolucionadas?"*

- Marvin Minsky -

Como plataforma robótica se ha empleado la segunda versión del exitoso *Turtlebot* junto a una serie de dispositivos para aumentar sus capacidades, como un láser de barrido o un brazo robótico. Se trata de un robot de bajo coste, ideal para el área de la investigación y educación, que cuenta con múltiples sensores y actuadores, así como completa compatibilidad con ROS, pudiendo hacer uso de una gran cantidad de paquetes desarrollados por la comunidad. [12]



Figura 11 – Hardware del Turtlebot 2.

El *Turtlebot2* incorpora de serie una base *Kobuki*, una estación de carga, una cámara *Kinect*, un portátil con ROS y la estructura del propio *Turtlebot* formada por bandejas y soportes. Además, se le añadió posteriormente un láser y un brazo robótico de cinco grados de libertad, y se sustituyó el portátil por un *Intel NUC* de mayor rendimiento, así como la cámara *Kinect* por su segunda versión de mayor resolución, junto con una plataforma más elevada (Figura 11).

### 3.1 Base móvil

*iClebo Kobuki* es el nombre de la base móvil del *Turtlebot2*, y cuenta con cuatro ruedas fijas, dos de ellas en configuración diferencial (rueda izquierda y derecha).



Figura 12 – Base móvil iClebo Kobuki.

Aunque dispone de sensores, motores y suministradores de energía, esta base no puede hacer nada por sí sola. Por este motivo, para ser funcional, la base requiere estar conectada a un computador. Entre sus sensores principales destaca su odometría altamente precisa, y un giroscopio calibrado de fábrica que permite una navegación precisa. A continuación, se muestran las especificaciones de la base, tanto funcionales como de hardware [13].

Especificaciones funcionales:

- Velocidad lineal máxima de 0.7 m/s
- Velocidad angular máxima de 180 grados/s
- Capacidad de carga de 5kg en suelo duro, y de 4 Kg en alfombra o moqueta.
- Detección de pendientes verticales mayores de 5 cm de altura.
- Capaz de superar desniveles de 12 mm máximo.
- Tiempo de operación de hasta 3/7 horas (batería pequeña/grande).
- Tiempo de carga de 1.5/2.6 horas (batería pequeña/grande).
- Capacidad de acoplarse a la base de carga por sí sola en un área de  $2 \times 5 \text{ m}^2$ .
- Firmware actualizable vía USB.

Especificaciones hardware:

- Detección de sobrecarga del motor.
- Odometría (25715.16 ticks/vuelta o 11.7 ticks/mm).
- Giroscopio de 1 eje (100 grados/s).
- Tres bumpers (frontal, izquierda y derecha).
- Tres sensores de pendiente vertical (frontal, izquierda y derecha).
- Sensores de caída de rueda (rueda izquierda y derecha).
- Conectores de alimentación: 5V/1A, 12V/1.5A y 12V/5<sup>a</sup>.
- Conector de carga: 19V/2.1A.
- Pines de expansión: 3.3V/1A, 5V/1<sup>a</sup>, 4 entradas analógicas, 4 entradas digitales y 4 salidas digitales.
- Audio con diferentes secuencias programables usando bips o pitidos.
- Tres botones táctiles.
- Dos LEDs programables de colores.
- Un LED de estado del robot:
  - Intermitente: Cargando batería.
  - Verde: Batería nivel alto.
  - Naranja: Batería nivel bajo.
- Batería de Lition-Ion de 2200 mAh (pqueña) y 4400 mAh (grande).
- Frecuencia de datos de los sensores: 50 Hz.
- Recarga de baterías mediante estación de carga.
- Receptores IR de acoplamiento a la estación de carga (frontal, izquierda y derecha).

### 3.2 Controlador

El PC controlador es el encargado de ejecutar ROS y controlar el resto de dispositivos del robot. Esta tarea puede ser desempeñada por cualquier hardware con capacidad para ejecutar una distribución de Linux compatible con ROS.

Aunque al inicio del proyecto el hardware encargado de esta función era un portátil *netbook* de *Toshiba*, más adelante se decidió sustituirlo por un mini ordenador *Intel Nuc* de mayor capacidad de procesamiento. El motivo principal fue que conforme el proyecto fue avanzando se necesitó de mayor potencia para procesar los algoritmos de detección de objetos, además de obtener una mejora evidente en el rendimiento completo del robot.

Así pues, finalmente el PC Controlador empleado es un *Intel Nuc* NUC5I5RYH que monta en su interior un *Intel Core i5-5250U*, al que se le añadió un disco SSD de 120 GB y un módulo de memoria *RAM* de 4GB. [14]



Figura 13 – Intel Nuc NUC5I5RYH.

### 3.3 Escáner láser 2D

Aunque la cámara Kinect que incorpora de serie el robot permite una lectura de la distancia en 3D, y pese a que la comunidad de ROS ha desarrollado un software que permite transformar la lectura de la Kinect en un escáner 2D, se decidió añadir un láser con tecnología *LIDAR* para obtener un barrido más amplio y con mayor precisión, pudiendo dejar libre la cámara para otro tipo de aplicaciones.

La tecnología *LIDAR* (de los términos “light” y “radar”) permite determinar la distancia a la que se encuentra un objeto mediante la emisión de un rayo láser sobre el entorno. El receptor localizado dentro del propio láser calcula la distancia mediante el tiempo de vuelo o tiempo que tarda el rayo desde que se ha emitido hasta que el receptor lo recibe. Debido a la gran velocidad de la luz, el proceso completo se realiza casi instantáneamente, lo que permite hacer un barrido 2D rápidamente del entorno, apuntando a distintas direcciones mediante un espejo giratorio que desvía el láser emitido.

El láser utilizado es en concreto el *Hokuyo URG-04LX-UG01*. Se trata de uno de los láseres *LIDAR* más pequeños disponibles en el mercado, con un peso de solamente 160 gramos. Está diseñado especialmente para aplicaciones de interior, permitiendo medidas de hasta 4 metros de distancia, y un barrido de  $210^\circ$ . Igualmente, cuenta con unas medidas de alta precisión ( $\pm 30$  mm) y una resolución angular de  $0.382^\circ$ . [15]



Figura 14 – Láser LIDAR Hokuyo.

### 3.4 Manipulador

Respecto al manipulador, en la bandeja superior del robot se decidió colocar un brazo robótico modelo *PhantomX Pincher AX-12* del fabricante *TrossenRobotics*, ya que es ligero y compatible con ROS, ideal para integrarlo en el robot *Turtlebot2*.

El brazo completo tiene un peso de 550 gramos y un alcance horizontal de 28 centímetros, pudiendo levantar un peso máximo de 100 gramos si el objeto se encuentra cercano a la base, y de 40 gramos si el brazo está extendido, suficiente para su propósito como hardware para la investigación. [16]

Un robot manipulador está compuesto por una serie de elementos sólidos llamados eslabones, que están unidos mediante articulaciones que permiten un movimiento relativo entre eslabones consecutivos, también conocidos como grados de libertad. A su vez, típicamente se puede dividir un robot manipulador en tres partes diferentes, brazo, muñeca y efector final.

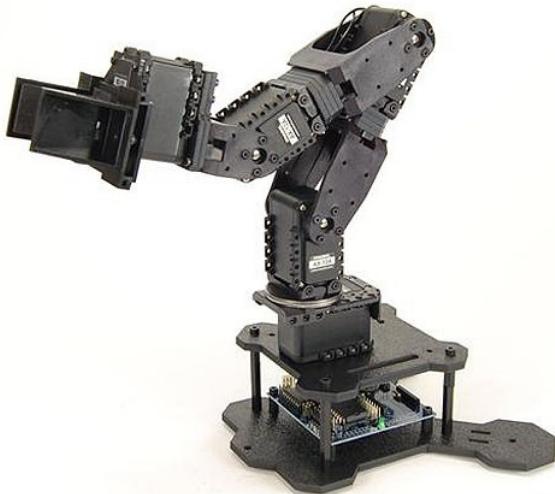


Figura 15 – PhantomX Pincher.

El *PhantomX Pincher* es robot manipulador de tipo antropomórfico que cuenta con cinco grados de libertad gracias a sus cinco actuadores *Dynamixel AX-12*, un microcontrolador *ArbotiX-M* y una garra personalizable como efector final. [16]

Los cinco grados de libertad del robot se reparten entre el brazo, la muñeca y el efector final. En primer lugar, el brazo cuenta con tres grados de libertad correspondiente a los tres primeros servos contando desde la base, que le permite colocar el efector final en cualquier punto de las tres direcciones del espacio. En segundo lugar, la muñeca posee únicamente un grado de libertad, pudiendo solo realizar el movimiento de cabeceo o *pitch* correspondiente a la orientación del efector final. Por último, el efector final cuenta con el último grado de libertad para poder realizar la apertura y cierre de la garra.

Todos los actuadores del *PhantomX Pincher* realizan un movimiento rotacional, excepto el último servo correspondiente con el efector final que, mediante un sencillo mecanismo, transforma el movimiento rotacional en un movimiento prismático para poder abrir y cerrar la garra.

### 3.5 Sensor 3D

La cámara o sensor 3D *Kinect* (Figura 16) es el nombre del dispositivo desarrollado por *PrimeSense* y que fue lanzado en noviembre de 2010 por *Microsoft* como periférico para su consola de videojuegos *Xbox 360*. Su principal función era poder ofrecer a los jugadores la posibilidad de interactuar con la consola sin necesidad de usar un controlador tradicional, sino que el propio usuario era el controlador mediante el uso del movimiento de su cuerpo y comandos de voz. [17]

En un principio la *Kinect* solo tenía como propósito su uso para videojuegos, pero la comunidad de usuario mediante ingeniería inversa lograron utilizarla en un PC convencional para hacer uso de sus características como sensor de profundidad. Para obtener dicha información, la *Kinect* emplea una técnica empleando luz infrarroja basada en la técnica de luz estructurada.

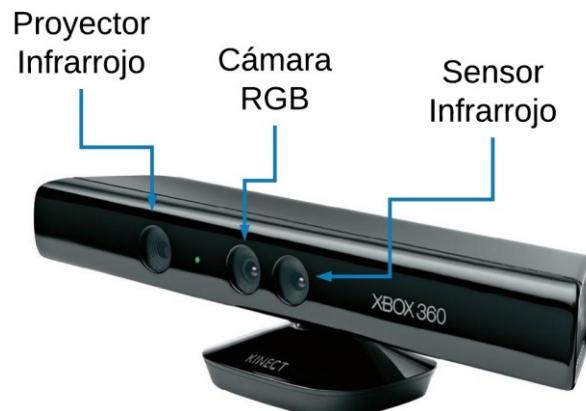


Figura 16 – Sensor Kinect. Primera versión.

Debido al entusiasmo que despertó este dispositivo en la comunidad de desarrolladores, *Microsoft* decidió liberar su SDK (*Software Development Kit*) para *Windows* en junio de 2011, permitiendo ser usada como cámara de visión 3D de bajo coste por toda la comunidad investigadora [18]. Esto llevó a que rápidamente se hicieran múltiples estudios y desarrollos de nuevos algoritmos de visión en todo el mundo apoyándose en las posibilidades que ofrecía *Kinect* [19]. Además, pronto se desarrolló los drivers necesarios para ser usada en ROS, y por consiguiente poder ser empleada en el *Turtlebot2*, el cual trae la cámara *Kinect* de serie.

La *Kinect* consta de una cámara RGB con posibilidad de obtener resoluciones de 640x480 a 30fps y 1280x960 pixeles a 12fps, un sensor de profundidad formados por un emisor y sensor de infrarrojos con resoluciones de 640x480, 320x240 y 80x60 pixeles, con un rango de visión horizontal de 57° y vertical de 43°, y un rango de profundidad de 1,2 a 3,5 metros. [20]



Figura 17 – Sensor Kinect. Segunda versión.

En 2013, *Microsoft* presentaría junto a su nueva consola *Xbox One*, la segunda versión del sensor *Kinect*, ofreciendo importantes mejoras respecto a la primera versión. Este sensor de profundidad, al igual que en la primera versión, está compuesto por una cámara y proyector de infrarrojos, además de incorporar la tecnología de tiempo de vuelo (TOF).

Entre las nuevas características del sensor, destacan una mayor resolución tanto en la cámara RGB (1920x1080 pixeles a 30fps) como de infrarrojos (512x424 pixeles), un aumento del campo de visión horizontal ( $70^\circ$ ) y vertical ( $60^\circ$ ), y una mejora del rango de profundidad de 0.5 a 4 metros. Además, se introdujo junto a la técnica de luz estructurada con luz infrarroja, la posibilidad de obtener la profundidad usando el método de tiempo de vuelo, permitiendo su uso en exteriores a diferencia de la primera versión. Es por ello, que según fue avanzando el proyecto, se decidió sustituir la *Kinect* original por su sucesora. [20]



## 4 LOCALIZACIÓN DE PRODUCTOS

---

*“La visión por computador y las técnicas de aprendizaje por una máquina realmente acaban de empezar a despegar, pero para mucha gente, la idea de lo que es una computadora viendo cuando está mirando a una imagen es relativamente desconocida.”*

- Mike Krieger -

Como parte esencial para poder navegar dentro de la tienda hacia los diferentes productos, se desarrolló un paquete que proporcionara al robot de la capacidad de detectar productos y obtener su ubicación en la tienda. Inicialmente, se pensó en usar etiquetas RFID como se detalla en las especificaciones al inicio del documento. Para ello, era necesario proveer a los productos de la tienda de etiquetas RFID, en caso de que no tuvieran, además de incluir un lector RFID en el robot para así poder detectarlas. Sin embargo, esta solución fue rápidamente descartada al estudiar las posibilidades que ofrece el sensor 3D *Kinect*, ya que además de poder usar procesamiento de imágenes para reconocer patrones, permite obtener la profundidad o posición de uno o varios puntos en el espacio que esté dentro de su campo de visión. El mecanismo empleado por la *Kinect* se explicará con más detalle posteriormente.

Como sustituto de las etiquetas RFID, se pensó en utilizar códigos de barras pues estos están presentes en todos los productos del mercado de manera universal. Sin embargo, estos códigos están pensados para ser leídos por escáneres de una dimensión a escasos centímetros, por lo que su lectura empleando una cámara de resolución normal y a una distancia de al menos un metro se antoja una tarea harta difícil. Por otro lado, los códigos de dos dimensiones sí que están diseñados para ser detectados mediante cámaras convencionales haciendo uso de algoritmos de visión, es por ello que finalmente se empleó códigos QRs al ser los códigos bidimensionales más famosos y extendidos. Aun así, el software utilizado permite la posibilidad de hacer uso de diferentes tipos de códigos, entre ellos los códigos de barra, aunque los resultados son mucho más pobres.

Antes de comenzar el desarrollo del software se buscó entre la comunidad de ROS un paquete similar. Aunque lamentablemente no se encontró nada que se adecuara a los requisitos deseados, sí que existían distintos paquetes que permitían detectar códigos de barra y QRs empleando una cámara de video. Entre estos paquetes se decidió utilizar el “*zbar\_detector*” del laboratorio *ViCoS* de la universidad de *Ljubljana* [21], el cual fue modificado y rebautizado en este proyecto como “*zbar\_ros*”. Seguidamente, se diseñó un nuevo paquete llamado “*position\_detector*” que utilizaba la información ofrecida por *zbar\_detector* para localizar los productos en el espacio 3D. Aunque inicialmente éste nació como otro nodo que enviaba un mensaje cada vez que detectara un código, al igual que *zbar\_detector*, conforme la aplicación fue creciendo se adaptó el código para que funcionara como un servicio.

De esta manera, cada vez que el robot se sitúe enfrente de una estantería o posible lugar donde pueda haber productos, el paquete de navegación llamará al servicio de “Localización de productos” y almacenará la información obtenida en una base de datos que se empleará en el futuro para guiar a los clientes por la tienda.

## 4.1 Tecnologías empleadas

Seguidamente, se describen las diferentes tecnologías empleadas y aquellos conceptos útiles necesarios para la elaboración de este paquete software.

### 4.1.1 Principio de funcionamiento de la Kinect

Aunque no se sabe de primera mano a través del fabricante el algoritmo empleado para obtener la profundidad, se conoce su funcionamiento de manera aproximada gracias al estudio de varios investigadores que han podido realizar ingeniería inversa sobre este dispositivo.

Como se ha mencionado en el capítulo anterior, la *Kinect* emplea la técnica de luz estructurada para obtener la distancia de los distintos puntos de la imagen. Para ello hace uso de un patrón de referencia como el que se muestra en la Figura 18. Este patrón se captura durante su etapa de fabricación sobre una superficie plana a una distancia del sensor conocida y se almacena en memoria. Por tanto, cuando se enciende el sensor, el láser o emisor de infrarrojo emite el patrón almacenado sobre la escena. Si en ese momento hubiera algún objeto o la superficie plana se encuentra a una distancia diferente de la conocida por el patrón de referencia, éste se distorsionará desde el punto de vista de la *Kinect*, existiendo ciertos desplazamientos de los puntos del patrón de referencia respecto al patrón obtenido por la cámara de infrarrojos.



Figura 18 – Patrón de puntos emitidos por la Kinect sobre una superficie plana.

Posteriormente, para obtener la distancia de los puntos del patrón reflejado, el dispositivo realiza una triangulación estableciendo una correlación entre los desplazamientos de los puntos de la imagen recibida y el patrón de referencia. Si bien se ha mencionado que se desconoce en detalle su funcionamiento, parece ser que se basa en una búsqueda y comparación por secciones de la imagen hasta encontrar puntos que sean coincidentes. En la Figura 18 también se puede apreciar las nueve regiones en las que está compuesta el patrón.

Cuando se obtiene la coincidencia, el dispositivo agranda la región observando los pixeles más próximos suponiendo que la diferencia en cada superficie no es muy grande. Dicha región se sigue agrandando hasta encontrar una variación grande en todas las direcciones que permitan deducir que se trate de una región distinta. Entonces, se abandona esa región y se busca otra coincidencia para formar nuevas regiones de la escena.

En la Figura 19 se puede observar la triangulación que realizaría el sensor para estimar la profundidad de un objeto  $k$ , donde  $d$  es la disparidad medida entre el plano de referencia y el plano del objeto obtenido. [22]

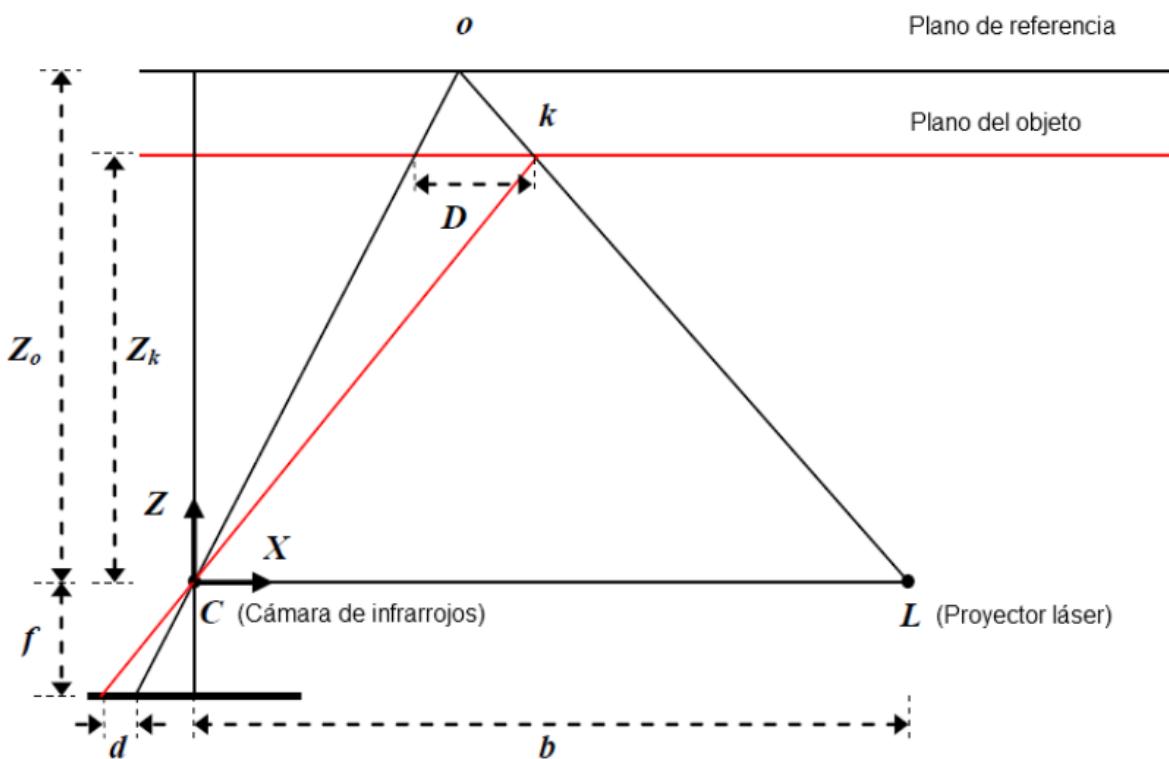


Figura 19 – Triangulación empleada por la Kinect para obtener la distancia a un punto.

En el ejemplo, el origen de coordenada se encuentra en el sensor infrarrojo, siendo  $Z$  ortogonal a la imagen y orientado al objeto y  $X$  en la dirección de la base  $b$  del dispositivo. Además, en el ejemplo se considera el objeto  $k$  a una distancia más pequeña ( $Z_k$ ) que el plano de referencia, produciéndose un desplazamiento  $D$  hacia la derecha entre el punto del patrón de referencia  $o$  y el punto obtenido en su lugar  $k$ . Si el objeto estuviera a una distancia mayor, el desplazamiento sería a la izquierda.

Para cada punto, el sensor no mide directamente  $D$  sino la disparidad  $d$ , obteniéndose así un mapa de disparidad. Observando los triángulos de la imagen se obtiene:

$$\frac{D}{b} = \frac{Z_o - Z_k}{Z_o} \quad ; \quad \frac{d}{f} = \frac{D}{Z_k}$$

Donde  $Z_k$  es la profundidad a la que se encuentra el objeto  $k$ ,  $f$  es la distancia focal del sensor infrarrojo,  $b$  la distancia entre el sensor y el emisor IR, y  $Z_o$  la distancia conocida entre el sensor y el patrón de referencia. Tanto  $Z_o$  como  $b$  y  $f$  son parámetros conocidos, por lo que es posible determinar la distancia del objeto  $k$  despejando en ambas ecuaciones obteniendo que:

$$Z_k = \frac{Z_o}{1 + \frac{Z_o}{f * b} d}$$

Una vez conocido  $Z_k$ , las coordenadas  $X_k$  e  $Y_k$  se obtienen de la siguiente manera:

$$X_k = -\frac{Z_k}{f} (x_k - x_o + \delta_x)$$

$$Y_k = -\frac{Z_k}{f} (y_k - y_o + \delta_y)$$

Donde  $x_k$  e  $y_k$  son las coordenadas en la imagen del objeto  $k$ ,  $x_o$  e  $y_o$  son las coordenadas del punto principal, es decir, del offset de la imagen, y  $\delta_x$ ,  $\delta_y$  son correcciones de la lente del sensor. Tanto los valores de offset como las correcciones se pueden obtener mediante la calibración de la cámara. [22]

Usando las expresiones anteriores se puede saber las coordenadas XYZ de cada punto del patrón recibido, formando así una imagen 3D o nube de puntos donde cada pixel de dicha imagen corresponde con un punto en el espacio respecto al sensor. Si el valor del pixel es cero o nulo significa que la cámara no ha podido estimar ese punto.

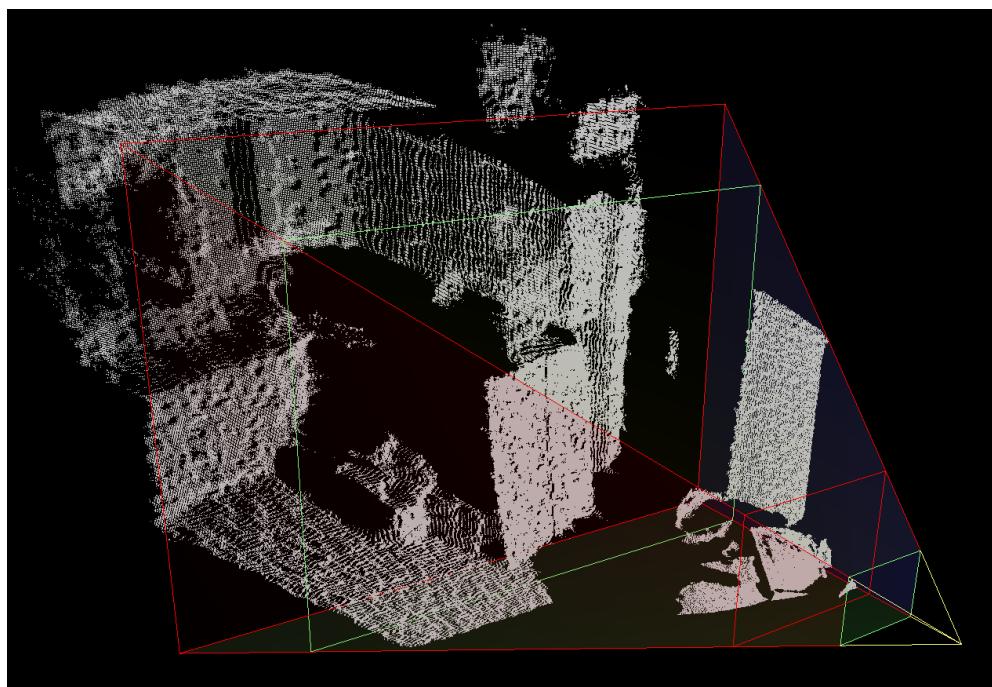


Figura 20 – Nube de puntos obtenida con la Kinect.

Por tanto, es posible que en ciertas ocasiones las medidas obtenidas por el sensor en algunos puntos no se pueden estimar o los datos no sean fiables. Esto sucede por una serie de limitaciones que se deben conocer antes de tratar con este tipo de sensores, y que vienen dadas por factores internos debido al propio diseño del dispositivo, como por factores externos debidos a la forma de la propia escena.

Como se puede ver en la Figura 20, el patrón de puntos no cubre de manera continua la superficie completa de los objetos por lo que algunos pixeles de la nube de puntos deben ser interpolados para obtener su valor. Esto implica un margen de error que será mayor cuanto más alejado se encuentre el objeto puesto que para una misma superficie los puntos obtenidos del patrón estarán más alejados. [23]

#### 4.1.2 OpenCV

OpenCV (*Open Source Computer Vision*) es una librería de tratamiento de imágenes desarrollada originalmente por *Intel*. Está disponible en los lenguajes de programación *C/C++*, *Python* y *Java*, y a su vez es compatible con *Windows*, *Linux*, *Android*, *iOS* y *Mac OS*. En 2006 fue liberada su primera versión estable, en octubre de 2009 apareció su segunda versión y en junio de 2015 su tercera y más reciente versión. [24]

Esta librería fue diseñada para ser eficiente computacionalmente con un especial enfoque en las aplicaciones en tiempo real, siendo su uso principal por tanto para aplicaciones de visión computacional en tiempo real. Su objetivo es proveer de una herramienta fácil de utilizar para los desarrolladores, poniendo a disposición de esto en torno a 500 funciones que permiten abarcar distintas áreas de la computación visual, incluyendo escaneo médico, automática, robótica, seguridad, etc. [25]

OpenCV se define como una estructura modular entre los que destacan los siguientes:

- **Core:** Es el módulo principal el cual incluye las estructuras de datos y funciones básicas.
- **Highgui:** Provee códecs de imagen y video, interfaz de usuario y la capacidad de capturar imagen y video.
- **Imgproc:** Está compuesto por algoritmos básicos como filtrado o transformado de imágenes.
- **Video:** Permite el análisis de videos como por ejemplo el seguimiento de objetos.
- **Objdetect:** Incluye algoritmos para la detección y reconocimiento de objetos.

#### 4.1.3 Point Coud Library

PCL son las siglas en inglés de “librería de nube de puntos” (*Point Cloud Library*). Se trata de un *framework* o infraestructura digital de código libre para percepción 3D escrito en *C++*. Esta librería fue desarrollada originalmente por *Willow Garage* como un paquete de ROS, pero debido a su utilidad como librería independiente rápidamente se desarrolló a parte. Actualmente está siendo desarrollada y mantenida por un consorcio de investigadores e ingenieros de todo el mundo, “*Open Perception Foundation*”. [26]

PLC permite procesar nubes de puntos mediante algoritmos de filtrado, reconstrucción de superficies, segmentación de regiones, búsqueda de características y puntos de interés, reconocimiento de objetos, etc. Está dividida en una serie de bibliotecas más pequeñas que proporcionan modularidad, reduciendo así limitaciones computacionales, que pueden ser compiladas y utilizadas de manera separada.

#### 4.1.4 Zbar

*Zbar* es una librería de código abierto que permite leer códigos de barras a través de imágenes o videos. Esta librería permite detectar diferentes tipos de símbolos incluidos *EAN-13/UPC-A*, *UPC-E*, *EAN-8*, *Code 39*, *Interleaved 2 of 5*, y códigos *QR*. [25]

## 4.2 Arquitectura del software

La arquitectura de este paquete software consta de dos nodos principales llamados “*barcode\_detector*” y “*position\_detector*”, además de otros nodos auxiliares como los drivers del robot y la *Kinect*, así como el nodo “*robot\_state\_publisher*”. Está diseñado como un servicio al que se le puede llamar desde otros nodos o desde el propio terminal mediante la siguiente instrucción:

- `rosservice call /localiza_productos`

La respuesta a dicha petición viene definida por el fichero “*BuscaProductos.srv*” que se detallará más adelante. En la Figura 21 viene representado los distintos nodos y conexiones o mensajes que participan en el proceso. Inicialmente, los drivers del robot y de la cámara *Kinect* se encargan de transformar las lecturas tanto del entorno como propias del robot en topics o mensajes que puedan ser de utilidad para otros nodos. Estos topics son enviados a la red de ROS constantemente, y son los nodos suscriptores de dichos topics los encargados de utilizar esa información según les convenga.

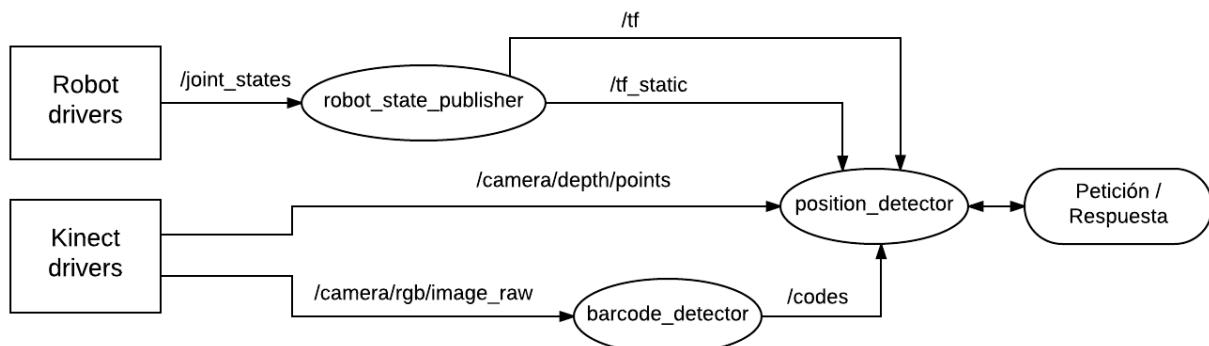


Figura 21 – Grafo de procesos del paquete “Localización de productos”.

En primer lugar, el nodo “*barcode\_detector*” perteneciente al paquete “*zbar\_ros*”, utiliza la información obtenida por la cámara RGB de la *Kinect* para detectar posibles códigos *QR* en la imagen. En caso afirmativo, se generará un mensaje llamado “*codes*” donde se incluirá la información sobre la posición en la imagen de los códigos detectados.

En segundo lugar, el nodo “*robot\_state\_publisher*”, gracias a la información obtenida del driver del robot sobre el estado de las articulaciones, genera dos topics indicando el estado de los distintos marcos de referencia tanto estáticos como no estáticos. Este nodo no pertenece al paquete de localización de productos propiamente dicho, pero es necesario ejecutarlo para conocer el estado de las “*tf*”.

Por último, el nodo “*position\_detector*” se suscribe a los topics “*codes*”, “*tf*” y “*tf\_static*” generado por los dos nodos anteriores, así como al topic de la nube de puntos ofrecido por la *Kinect*, empleando toda esa información para obtener la posición de los códigos respecto tanto a la base del robot como del mapa que éste emplee en el momento de la petición.

## 4.3 Descripción del código

En este punto se describirá el código empleado en este paquete haciendo hincapié en la funcionalidad de cada parte del código más que en las funciones empleadas en ellas. En primer lugar, se detallan los campos de los mensajes o servicios definidos, y posteriormente se explicará paso a paso los procesos que siguen los nodos *barcode\_detector* y *position\_detector* para detectar y obtener la distancia a un producto.

### 4.3.1 Mensajes y servicios

A continuación, se mostrará la definición de los distintos mensajes definidos para esta aplicación.

#### **Codes.msg**

Este mensaje es una versión modificada del original definido por el grupo *ViCoS* para su paquete *zbar\_detector* llamado “*Makers*” [21]. Su propósito es enviar los datos de los distintos códigos detectados por *barcode\_detector*. Algunos de los campos del mensaje son vectores debido a que en un mismo mensaje se puede enviar información de más de un código al mismo tiempo. Los diferentes campos del mensaje son:

- **header**

Se trata de una “cabecera” del tipo *Header* definido por la comunidad de ROS y que contiene diferente información que pueda ser útil sobre el estado del mensaje.

- **data**

Este campo contiene el valor almacenado en el código. El dato es guardado como una cadena de caracteres.

- **type**

Similar al campo anterior, contiene el nombre del tipo del código, por ejemplo, EAN-8 o QR.

- **center\_x**

Este campo indica la posición del centro de los códigos obtenidos en la imagen en el eje x. El valor está medido en pixeles y se almacena como un número de tipo entero.

- **center\_y**

Complementario al anterior, indica la posición del centro de los códigos obtenidos en la imagen en el eje y.

- **width**

Indica la anchura de los códigos obtenidos en la imagen medida en pixeles.

- **height**

En esta ocasión, la información ofrecida es la altura de los códigos obtenidos en la imagen medida en pixeles.

- **tam**

Este campo junto con la cabecera, son los únicos que no son vectores. Indica el tamaño de los vectores, o número de códigos obtenidos.

Por último, la definición de los campos dentro del fichero tiene el siguiente aspecto:

Header	header
string []	data
string []	type
int32 []	center_x
int32 []	center_y
int32 []	width
int32 []	height
int32	tam

### BuscaProductos.srv

Este fichero define la comunicación entre los nodos cliente y servidor del servicio ofrecido por “*position\_detector*”. En este caso, todos los parámetros del mensaje son de salida y deben ser generados por el servidor. Entre los datos a enviar se encuentran algunos procedentes del mensaje “*codes.msg*” sin alterar, como *data* y *type*. Al igual que en *codes*, algunos de los campos del mensaje son vectores debido a que en un mismo mensaje se puede enviar información de más de un código al mismo tiempo. Los diferentes campos del mensaje son:

- **header**

Se trata de una “cabecera” del tipo *Header* definido por la comunidad de ROS y que contiene diferente información que pueda ser útil sobre el estado del mensaje.

- **productos**

Indica el número de productos o códigos detectados.

- **error**

Se trata de un código numérico para informar al nodo cliente del servicio si ha ocurrido algún error. En caso de que el valor de este campo fuese cero, significa que no ha habido problemas. Los diferentes códigos de error son:

- 1. Motivo: TransformPoint.
- 2. Motivo: Zbar no está activo.
- 3. Motivo: Timeout PointCloud.
- 4. Motivo: El punto obtenido tiene un valor igual a NaN.
- 5. Motivo: Resolución incorrecta.

- **data**

Este campo contiene el valor almacenado en el código. El dato es guardado como una cadena de caracteres.

- **type**

Similar al campo anterior, contiene el nombre del tipo del código, por ejemplo, EAN-8 o QR.

- **map\_x**

Este campo indica la posición de los códigos respecto al mapa en la coordenada x. El valor está medido en metros y se almacena como un número de tipo flotante.

- **map\_y**

Igual al anterior, indica la posición de los códigos respecto al mapa en la coordenada y.

- **map\_z**

Similar al anterior, en esta ocasión indica la posición de los códigos respecto al mapa en la coordenada z.

- **base\_x**

Este campo indica la posición de los códigos respecto a la base del robot en la coordenada x. El valor está medido en metros y se almacena como un número de tipo flotante.

- **base\_y**

Igual al anterior, en esta ocasión indica la posición de los códigos respecto a la base del robot en la coordenada y.

- **base\_z**

Similar al anterior, en esta ocasión indica la posición de los códigos respecto a la base del robot en la coordenada z.

Por último, la definición de los campos dentro del fichero tiene el siguiente aspecto:

```
---
Header header
int32 productos
int32 error
string[] data
string[] type
float32[] map_x
float32[] map_y
float32[] map_z
float32[] base_x
float32[] base_y
float32[] base_z
```

Las tres líneas iniciales indican que es la respuesta del servicio. Los datos escritos por encima de estas tres líneas divisorias significarían datos de entrada durante la petición del servicio.

#### 4.3.2 Nodo barcode\_detector

Este nodo es el encargado de buscar en la imagen posibles códigos de barra o *QR*, y notificarlo a los nodos suscriptores mediante un mensaje. Este código, al igual que el mensaje *codes*, ha sido modificado del código original desarrollado por el grupo *ViCoS* en su paquete *zbar\_detector* [21]. Originalmente este nodo realizaba la detección de códigos y enviaba un mensaje con el primer código detectado sin importar que hubiera más de un código en la imagen. Además, la imagen mostrada consistía en una imagen en blanco y negro con un marco rodeando el código detectado.

Es por ello que según avanzó el proyecto y aumentaron los conocimientos adquiridos en ROS, se decidió modificar el código para que pudiera enviar en un solo mensaje todos los códigos detectados posibles, además de añadir mejoras visuales como mostrar una imagen a color junto con la información que contiene el código. Adicionalmente, se creó un pequeño servicio para comprobar si el nodo seguía ejecutado o no llamado “*amialive*”.

## Definiciones y dependencias

Al inicio del fichero *main.cpp* se encuentran las dependencias y definiciones de variables empleadas durante el código. Entre las dependencias, existen algunas conocidas ya como *Zbar* y *OpenCV*, así como necesarias para transformar la imagen recibida por la Kinect como “*cv\_bridge*”.

```
#include "ros/ros.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sstream>
#include <iostream>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <sensor_msgs/image_encodings.h>
#include <zbar.h>
#include "zbar_ros/codes.h"
#include <std_srvs/Trigger.h>
#include <vector>

using namespace std;
using namespace zbar;
using namespace cv;
namespace enc = sensor_msgs::image_encodings;

bool SHOW_CV_WINDOW;
string IMAGE_TOPIC;
cv_bridge::CvImagePtr bridge;
cv_bridge::CvImagePtr bridge2;

// Vectores
vector<std::string> datas;
vector<std::string> types;
vector<int> centers_x;
vector<int> centers_y;
vector<int> widths;
vector<int> heights;

std::stringstream sd,st;
IplImage * frame = 0;
uint message_sequence = 0;

// Lector
ImageScanner scanner;
// Publicador
ros::Publisher code;
```

Entre las variables definidas destacan los vectores, encargados de almacenar la información obtenida para construir el mensaje, el lector o *scanner*, necesario para detectar códigos en la imagen, y el publicador de ROS, el cual se encarga de transmitir el mensaje a la red.

También cabe destacar que se han creado dos variables del tipo *cv\_bridge* para poder transportar en una la imagen a color, y en la otra, la imagen codificada en blanco y negro para facilitar la detección de códigos al lector.

## Main

Esta es la función principal del nodo. Aquí se crea y ejecuta el nodo y se define entre otras los topics a los que se está suscrito, o el topic donde se publicará el mensaje. Se puede observar como *barcode\_detector* está suscrito al topic “*camera/rgb/image\_raw*” y publica en “*codes*”. Además, se configura el lector o el servicio “*amialive*”.

```
int main(int argc, char **argv)
{
    ros::init(argc, argv, "barcode_detector"); // Nodo barcode_detector
    ros::NodeHandle n,s; // Crea el NodeHandle para barcode_detector node
    para suscribirse y publicar topics

    code = n.advertise<zbar_ros::codes>("codes", 1000);
    ros::Subscriber sub = n.subscribe(IMAGE_TOPIC, 10, imageReceiver);

    //Definimos los parametros
    n.param("barcode_detector/show_cv_window", SHOW_CV_WINDOW, false);
    n.param("barcode_detector/image_topic", IMAGE_TOPIC,
    string("/camera/rgb/image_raw"));

    if(SHOW_CV_WINDOW)
        cv::namedWindow("Zbar_ros", CV_WINDOW_AUTOSIZE);

    // Configuración del lector
    scanner.set_config(ZBAR_NONE, ZBAR_CFG_ENABLE, 1);

    // Servicio para saber si el nodo ha iniciado correctamente
    ros::ServiceServer amIAlive;
    amIAlive = s.advertiseService("amialive", vivo);

    ros::spin();

    return 0;
}
```

Por último se ejecuta la instrucción *ros::spin()*. Esta instrucción es muy importante para el funcionamiento de un nodo, pues consiste en un bucle infinito que comprueba de manera periódica el estado de los topics a los que el nodo está suscrito. Si se recibe un mensaje por ese topic se ejecutará la función definida por el suscriptor, en este caso la función “*imageReceiver*”.

## Vivo

Se trata de una función muy simple cuya función es responder “*true*” si el servicio “*amialive*” es invocado. Es especialmente útil para que otros nodos conozcan de antemano si el nodo *barcode\_detector* está ejecutándose y disponible.

```
bool vivo(std_srvs::Trigger::Request &req, std_srvs::Trigger::Response &res)
{
    return true;
}
```

## ImageReceiver

Esta función es el cuerpo principal de este paquete y se ejecuta cada vez que se recibe una imagen a través del topic. Una vez ejecutada, obtiene la imagen en formato “raw” y la codifica en dos variables tipo *cv\_bridge*, una a color, empleada para mostrarla por pantalla posteriormente, y otra en blanco para facilitar la búsqueda de códigos.

Ambas imágenes a su vez, empleando la librería *OpenCV*, son almacenadas en matrices con el fin de facilitar su manipulación empleando dicha librería.

```
void imageReceiver(const sensor_msgs::ImageConstPtr &image) {

    // Imagen de color para mostrar
    try {
        bridge2 = cv_bridge::toCvCopy(image, enc::BGR8); //Encoder type ->
BGR8 : CV_8UC1, color image
    }
    catch (cv_bridge::Exception& e) {
        ROS_ERROR("cv_bridge exception: %s", e.what());
        return;
    }

    // ... Imagen en blanco y negro para proveresarla
    try {
        bridge = cv_bridge::toCvCopy(image, enc::MONO8); //Encoder type ->
MONO8 : CV_8UC1, grayscale image
    }
    catch (cv_bridge::Exception& e) {
        ROS_ERROR("cv_bridge exception: %s", e.what());
        return;
    }
    cv::Mat cv_matrix = bridge->image;
    cv::Mat cv_matrix2 = bridge2->image;

    int width = cv_matrix.cols;
    int height = cv_matrix.rows;
```

A continuación, se procede a la búsqueda de códigos en la imagen mediante la función “*scan*” proporcionada por la librería *Zbar*. Por cada código detectado se almacena en un vector los datos necesarios para construir el mensaje, como el tipo, el contenido del código, posición en la imagen, etc.

```
// ... encontrando códigos ...

uchar* raw = cv_matrix.ptr<uchar>(0);
Image scan_image(width, height, "Y800", raw, width * height);
int n = scanner.scan(scan_image);

if (n < 0) {
    ROS_ERROR("Error occurred while finding barcode");
    return;
}
```

```

// extrayendo resultados ...
for(SymbolIterator symbol = scan_image.symbol_begin();
    symbol != scan_image.symbol_end();
    ++symbol) {

    // Tomamos los datos y el tipo de cada código

    char name[256];

    sd << symbol->get_data();
    datas.push_back(sd.str());

    st << symbol->get_type_name();
    types.push_back(st.str());

    // Buscando la localización del código en la imagen
    int x1 = width, y1 = height, x2 = 0, y2 = 0;

    for (int i = 0; i < symbol->get_location_size(); i++) {
        x1 = MIN(x1, symbol->get_location_x(i));
        y1 = MIN(y1, symbol->get_location_y(i));
        x2 = MAX(x2, symbol->get_location_x(i));
        y2 = MAX(y2, symbol->get_location_y(i));
    }

    centers_x.push_back((x1 + x2)/2);
    centers_y.push_back((y1 + y2)/2);
    widths.push_back(x2 - x1);
    heights.push_back(y2 + y1);
}

```

Seguidamente, una vez obtenida la información de cada código se procede a “dibujarlos” en la imagen a color. Para ello en primer lugar se dibuja un cuadrado con un círculo en el centro, sabiendo la anchura y altura de cada código, así como la ubicación en la imagen. Además, se muestran el contenido del código por pantalla gracias a la función “*putText*”. En la Figura 23 se puede ver un ejemplo empleando un código QR y un código de barras.

```

int thickness = 2;

if(SHOW_CV_WINDOW) {

    // Dibujando los códigos en la imagen
    cv::rectangle(cv_matrix2, cv::Point(x1,y1),
    cv::Point(x2,y2),cv::Scalar(255, 0, 0),thickness);
    cv::circle(cv_matrix2, cv::Point((x1 + x2)/2, (y1 + y2)/2), 1,
    cv::Scalar(0,0,255),CV_FILLED);
    cv::putText(cv_matrix2, sd.str() , cv::Point(x1,(y1 + y2)/2),
    FONT_HERSHEY_SIMPLEX , 0.8, cv::Scalar(0,255,0), 2);

    // Inicializa los stringstream
    sd.str("");
    st.str("");

    // Mostrando la imagen en una ventana
    cv::imshow("Zbar_ros", cv_matrix2);
    cv::waitKey(1);

}

```

Por último, se crea y se almacena en el mensaje *codes* la información recopilada para cada uno de los códigos detectados, publicándolo finalmente en el topic que lleva el mismo nombre y esperando a una nueva llamada a la función tras recibir una nueva imagen.

```
// Creamos el mensaje
zbar_ros::codes msg;

msg.header.seq = message_sequence++;
msg.header.stamp = ros::Time::now();
msg.header.frame_id = image->header.frame_id;
msg.tam = datas.size();

if (msg.tam > 0){ // Nos aseguramos que solo publica si detecta un
// código o varios

    for(int i=0; i < tam; i++){

        msg.data.push_back(datas[i]);
        msg.type.push_back(types[i]);
        msg.center_x.push_back(centers_x[i]);
        msg.center_y.push_back(centers_y[i]);
        msg.width.push_back(widths[i]);
        msg.height.push_back(heights[i]);
    }

    //Publicamos el mensaje
    code.publish(msg);
}

// Se borra el contenido de los vectores
datas.clear();
types.clear();
centers_x.clear();
centers_y.clear();
widths.clear();
heights.clear();
}
```

#### 4.3.3 Nodo position\_detector

Este nodo se encarga de gestionar toda la información para obtener la posición de los productos. Está compuesto por un objeto llamado “*Detector*” que contiene a su vez las funciones encargadas de realizar las tareas oportunas. Entre estas funciones destaca el “servidor”, el cual se encarga de atender las peticiones.

El proceso se realiza en varias etapas como se describe en el diagrama de flujo de la Figura 22, comprobando que se ha podido realizar todas las tareas con éxito, enviando en caso contrario un código de error en el campo correspondiente del mensaje. Nótese que, el hecho de no recibir ningún código no implica un error en el proceso, sino que el número de códigos detectados es igual a cero.

Debido a que este nodo se sirve de *barcode\_detector* y de la información ofrecida por los drivers de la *Kinect*, el mensaje que activa para obtener la información necesaria para realizar la operación

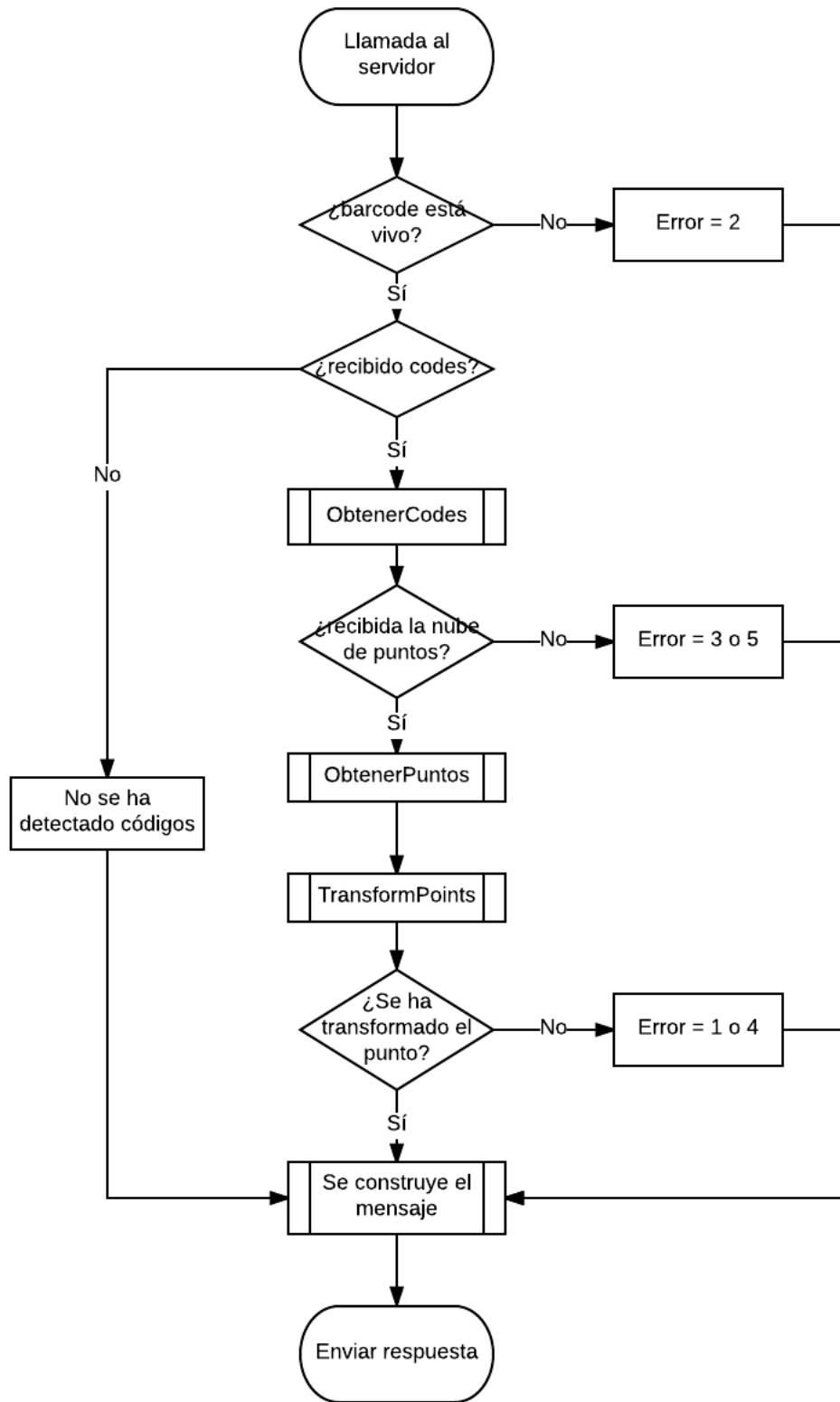


Figura 22 – Diagrama de flujo del servicio “Localiza productos”

## Definiciones y dependencias

En este caso, el fichero principal se llama “*position\_detector.cpp*”. Al inicio de dicho fichero se encuentran las dependencias y definiciones de variables empleadas durante el código. Entre las dependencias más importantes destacan “*point\_cloud*”, “*tf*” y los mensajes “*codes*” y “*BuscaProductos*” que será necesarios más adelante.

Además, se ha redefinido *pcl\_PointCloud<pcl\_PointXYZ>* como NubePuntos para simplificar el código.

```
#include <ros/ros.h>
#include <geometry_msgs/PointStamped.h>
#include <tf/transform_listener.h>
#include <vector>
#include <pcl_ros/point_cloud.h>
#include <iostream>
#include <sstream>
#include <boost/bind.hpp>
#include "zbar_ros/codes.h"
#include <position_detector/BuscaProductos.h>
#include <std_srvs/Trigger.h>
#include <cmath>

using namespace std;
typedef pcl::PointCloud<pcl::PointXYZ> NubePuntos;
```

## Main

El *main* es la función que se ejecuta al inicio y su función en este paquete es iniciar el nodo, crear la instancia del objeto *Detector* y ejecutar *ros::spin()*, el cual juega un papel fundamental en el funcionamiento del nodo dentro de la red de ROS, tal y como se describió en el caso de *barcode\_detector*.

```
int main(int argc, char **argv)
{
    ros::init(argc, argv, "position_detector"); // Inicia el nodo llamado
    "position_detector"

    Detector Detector; // se crea el objeto detector de la clase Detector

    ros::spin(); // ros::spin() entra en un bucle y llama a los callbacks
    de los mensajes tan rápido como puede.

    return 0;
}
```

## Clase Detector

Este paquete se decidió diseñarlo como un objeto ya que permite ordenar y simplificar el código, así como poder obtener las ventajas de la programación orientada a objetos. Este objeto, llamado *Detector*, contiene una serie de variables que son de acceso privado por el objeto, junto a una serie de métodos o funciones que son de acceso público y que se explicarán a continuación.

```
class Detector
{
public:
    //Constructor
    Detector();
    //Funciones
    void ObtenerCodigo(const zbar_ros::codes::ConstPtr& msg);
    void ObtenerPuntos(const NubePuntos::ConstPtr& msg);
    void transformPoint(const tf::TransformListener& listener);
    //Servidor
    bool servidor(position_detector::BuscaProductos::Request &req,
position_detector::BuscaProductos::Response &res);

private:
    ros::NodeHandle positionNode, sa;

    // Variables donde se guardan los parámetros
    int N;
    int WIDTH;
    int HEIGHT;
    double TIMEOUT0;
    double TIMEOUT1;
    double TIMEOUT2;
    std::string DEPTH_IMAGE_TOPIC;

    //Variables que compartirá el objeto para procesar los puntos
    vector<int> px_rgb;
    vector<int> py_rgb;
    vector<std::string> ID;
    vector<std::string> type;
    vector<float> mapx;
    vector<float> mapy;
    vector<float> mapz;
    vector<float> basex;
    vector<float> basey;
    vector<float> basez;

    int tam_vector; // Tamaño de los vectores
    int tam_msg; // Tamaño del mensaje de Zbar
    int cont; //Intentos hasta obtener todos los códigos
    int cont2; //Número de llamadas al servidor
    int error; // Bandera de error de transformación del punto

    geometry_msgs::PointStamped camera_point;
    geometry_msgs::PointStamped map_point;
    geometry_msgs::PointStamped base_link_point;

    tf::TransformListener listener;
    ros::ServiceServer server;
    ros::ServiceClient salive;
};
```

## Constructor

En primer lugar, es necesario definir el constructor del objeto. Este método especial, propio de los lenguajes orientados a objetos, es llamado cuando se instancia un objeto. Aunque el compilador crea por defecto un constructor sin parámetros si éste no está definido, en esta ocasión es útil definirlo pues se precisa inicializar ciertos parámetros como se muestra a continuación.

```
Detector::Detector():
    tam_vector(0),
    tam_msg(0),
    cont(0),
    cont2(0),
    error(0)
{
    // Se leen los parametros de un fichero. Si no se indican, se toman los
    // valores por defecto indicados
    positionNode.param("position_detector/N_intentos", N, 3);
    positionNode.param("position_detector/WIDTH", WIDTH, 640);
    positionNode.param("position_detector/HEIGHT", HEIGHT, 480);
    positionNode.param("position_detector/TIMEOUT0", TIMEOUT0, 2.0);
    positionNode.param("position_detector/TIMEOUT1", TIMEOUT1, 7.0);
    positionNode.param("position_detector/TIMEOUT2", TIMEOUT2, 15.0);
    positionNode.param("position_detector/DEPTH_IMAGE_TOPIC",
DEPTH_IMAGE_TOPIC, string("/camera/depth/points"));

    // Servicio
    server = positionNode.advertiseService("localiza_productos",
&Detector::servidor, this);
    salive = sa.serviceClient<std_srvs::Trigger>("amialive"); //Servicio
para concer si el nodo zbar está activo
}
```

Entre las inicializaciones destacan ciertos parámetros como los “*TimeOuts*” o el topic de la imagen de profundidad o nube de puntos que pueden ser modificados en un fichero dentro de la carpeta “*param*” incluida en el proyecto. Si estos parámetros no son indicados o el fichero no existe, por defecto se toman los valores que se indican en el código. Esta herramienta es muy útil puesto que permite cambiar el valor de ciertos parámetros sin necesidad de modificar y compilar el código.

Además, en el constructor también se definen el nombre del servicio, “*localiza\_productos*”, así como el servicio para comprobar si zbar está activo.

## Servidor

Este es el método que es llamado tras recibir una petición al servidor, y funciona como director de orquesta de todo el proceso. En primer lugar, inicializa las variables privadas del objeto para comenzar el proceso.

```

bool Detector::servidor(position_detector::BuscaProductos::Request &req,
position_detector::BuscaProductos::Response &res) {
    cont2++;
    cout << "Recibida petición del servidor..." << endl;
    cout << "Llamada al servidor número " << cont2 << "\n" << endl;

    double t_inicio = ros::Time::now().toSec(); // Tiempo de inicio
    double t_actual = ros::Time::now().toSec(); // Tiempo actual

    position_detector::BuscaProductos producto; // Se crea la variable o
mensaje que se enviará al cliente de la petición

    cout << "\tLa resolución de la cámara RGB es de " << WIDTH << "x" <<
HEIGHT << " pixeles" << endl;

    //Se inicializan todas las banderas y variables de nuevo por seguridad

    tam_msg = 0;
    tam_vector = 0;
    error = 0;
    cont = 0;
}

```

En segundo lugar, se realiza una espera activa para estabilizar la imagen. Esta instrucción fue necesaria incluirla puesto que para CPUs con poca capacidad de procesamiento como el portátil usado al inicio del proyecto, existe un pequeño retraso al recibir imágenes de la Kinect. Sin embargo, para procesadores con mayor capacidad es posible evitarlo definiendo *TIMEOUT0* como cero.

```

cout << "... esperando "<< TIMEOUT0 << " segundos de margen para que se
estabilice la imagen ... \n" << endl;
ros::Duration(TIMEOUT0).sleep();

cout << "\tBuscando códigos en la imagen RGB ... \n\n" << endl;

// Comprobación para saber si Zbar está activo
if(!ros::service::waitForService("amialive",ros::Duration(1)))
    error=2;

if(error==0){

    for(cont=0; cont < N; cont++){ // Se repite la búsqueda N veces

        cout<<"\nNúmero de intentos de lectura de Zbar " << cont+1 << " de "
<< N << " intento/s \n" << endl;

        zbar_ros::codes::ConstPtr msg =
ros::topic::waitForMessage<zbar_ros::codes>("/codes",ros::Duration(TIMEOUT1))
); // Se espera TIMEOUT1 segundos como máximo hasta recibir un mensaje de
zbar
        if (msg){
            cout<<"\tCódigo/s detectado/s" << endl;
            ObtenerCodigo(msg);
        }
        else{
            cout<<"\tNo se ha encontrado ningún código en " << TIMEOUT1 << "
segundos" << endl;
        }
    }
}

```

Tras ello se procede a escuchar el topic “codes” durante un tiempo definido por *TIMEOUT1*. Este proceso se repite *N* veces por robustez, siendo *N* un parámetro ajustable por el usuario. Esto se debe a que el procesamiento de imágenes es algo delicado, influyendo en exceso las condiciones de luz del entorno, por lo que al repetir el proceso varias veces se obtiene mayor probabilidad de obtener los códigos correctamente. Si se obtienen códigos se llama al método “*ObtenerCodigo*” y se le pasa el mensaje recibido como parámetro, guardando la información del mensaje en variables privadas. En caso contrario se asumirá que no hay códigos en esa posición y se procederá a enviar la respuesta.

```

if(tam_vector > 0){ //Si no se detecta productos no se procesa la nube

    cout << "\tBuscando puntos en la nube 3D ... \n\n" << endl;
    NubePuntos::ConstPtr msgcloud =
ros::topic::waitForMessage<NubePuntos>(DEPTH_IMAGE_TOPIC,ros::Duration(TIMEOUT2)); // Se espera TIMEOUT2 segundos como maximo

    if (msgcloud) {
        ObtenerPuntos(msgcloud);
    }
    else{
        cout<<"\tNo se ha recibido la nube de puntos en "<< TIMEOUT2 <<
" segundos \n" << endl;
        error = 3; // Si no se ha recibido mensaje es que hay error
    }
}

```

Posteriormente, en caso de existir códigos, se realiza un proceso parecido pero esta vez con la nube de puntos. En esta ocasión no se repite el proceso, sino que se espera un tiempo mayor. Si la espera se agota, se entiende que el driver de la cámara no está disponible o no funciona correctamente, y se tratará como un error. En caso contrario, se llama al método “*ProcesaPuntos*” que se encargará de obtener los puntos *XYZ* correspondiente con los códigos detectados.

```

cout << "\tAsignando datos para la respuesta del servidor \n\n" << endl;
}
if(error==0){

    if ( tam_vector > 0 ){

        for(int i=0; i < tam_vector; i++){

            res.productos = tam_vector; // productos encontrados
            res.header.stamp = ros::Time::now();
            res.error = 0;
            res.data.push_back(ID[i]);
            res.type.push_back(type[i]);
            //Puntos respecto al mapa
            res.map_x.push_back(mapx[i]);
            res.map_y.push_back(mapy[i]);
            res.map_z.push_back(mapz[i]);
            //Puntos respecto a la base
            res.base_x.push_back(base_x[i]);
            res.base_y.push_back(base_y[i]);
            res.base_z.push_back(base_z[i]);

            cout << "\nRespuesta del servidor construida \n" << endl;
        }
    }
}

```

Por último, se realiza la construcción del mensaje asignando a cada campo del mismo los valores obtenidos por los distintos métodos invocados por el servidor y que se verán a continuación. Como se dijo anteriormente, en caso de no detectar un código, se enviará como respuesta únicamente el valor del campo “*productos*” igual a 0.

```
    else{
        cout << "\tNingun productos encontrado ... \n" << endl;
        res.productos = 0; // Productos procesados --> 0
        res.error = 0;
        res.header.stamp = ros::Time::now();
    }
}
else{
    cout << "\nERROR: Imposible construir la respuesta del servidor
correctamente \n" << endl;
}

// Se borra el contenido de todos los vectores
ID.clear();
type.clear();
px_rgb.clear();
py_rgb.clear();
mapx.clear();
mapy.clear();
mapz.clear();
basex.clear();
basey.clear();
basez.clear();

cout << "Enviando respuesta ... \n" << endl;
t_actual = ros::Time::now().toSec();
cout << "El tiempo que se ha mantenido ocupado al servidor es de: " <<
t_actual-t_inicio << " segundos" << endl;

if (error != 0)
{
    cout << "\tRespuesta fallida [envía código error]" << endl;
    res.error = error;

    return true;
}
else{
    cout << "\tRespuesta valida" << endl;
    return true;
}
}
```

Finalmente se comprueba que no ha habido ningún otro error debido a otros factores como una mala conversión del punto o falsos positivos en la nube de puntos. En caso de existir, se envía una respuesta indicando en el campo de error, el tipo de error encontrado según el código de error definido en el apartado de mensajes y servicios.

## ObtenerCodigo

Mediante este método se obtiene los campos del mensaje *codes* enviado por *barcode\_detector* y es almacenado en las variables de tipo vector del objeto *Detector*. Este método será llamado tantas veces como se haya definido *N*, por lo que se necesita comprobar en cada llamada si el nuevo mensaje recibido contiene tantos productos como el vector almacenado. Inicialmente el tamaño del vector almacenado es cero.

```
void Detector::ObtenerCodigo(const zbar_ros::codes::ConstPtr& msg)
{
    tam_msg = msg->tam; // Tamaño del mensaje
    tam_vector = ID.size(); // Tamano del vector almacenado.

    cout << "Tam del mensaje recibido es " << tam_msg << endl;
    cout << "Tam del vector almacenado es " << tam_vector << endl;

    if (tam_vector < tam_msg){ // Si el vector ya almacenado es menor que
        el nuevo mensaje se sobreescribe ...

        if (tam_vector > 0){ // ... si existen datos anteriores
            cout << "Borrando antiguo vector y almacenando el nuevo vector
        recibido ... \n" << endl;

            ID.clear();
            type.clear();
            px_rgb.clear();
            py_rgb.clear();
            mapx.clear();
            mapy.clear();
            mapz.clear();
            basex.clear();
            basey.clear();
            basez.clear();
        }

        for(int i=0; i < tam_msg; i++){ // y se guardan los nuevos datos
            px_rgb.push_back(msg->center_x[i]);
            py_rgb.push_back(msg->center_y[i]);
            ID.push_back(msg->data[i].c_str());
            type.push_back(msg->type[i].c_str());
        }

        tam_vector = ID.size(); // Tras la asignacion, se calcula el nuevo
        Tam del vector
    }
    else{
        cout << "El vector almacenado es mayor o igual que el nuevo vector
        recibido\nSe desecha el mensaje recibido ... \n" << endl;
    }
}
```

Si el mensaje recibido contiene más códigos, se borra el vector anterior mediante el método “*clear*” y se almacena los nuevos datos en el vector mediante el método “*push\_back*”. Finalmente se actualiza el tamaño del vector para realizar la comprobación en una nueva llamada.

## ObtenerPuntos

Esta función es la encargada de realizar de obtener los puntos *XYZ* en el espacio 3D correspondiente con los códigos obtenidos. Esto se puede realizar gracias a que la imagen 2D obtenida por la cámara RGB y la nube de puntos son coincidentes, es decir, los pixeles de la imagen 2D corresponde con puntos dentro de la nube de puntos siempre que ambas imágenes tengan la misma resolución. En la Figura 24 se puede observar un ejemplo de ambas imágenes superpuestas en Rviz donde se aprecia dicha correspondencia.

```
void Detector::ObtenerPuntos(const NubePuntos::ConstPtr& msg) {

    int pixel = 0;
    for(int i=0; i < tam_msg; i++){
        // Conversion de resolucion a 640x480 necesarias en IR
        if((WIDTH==640)&&(HEIGHT==480)){
            pixel = py_rgb[i]*640 + px_rgb[i];
        }
        else if((WIDTH==1280)&&(HEIGHT==960)){
            cout << "Se procede a la conversion en 640x480 ..." << endl;

            int rgbx = ((px_rgb[i]*640)/WIDTH);
            int rgby = ((py_rgb[i]*480)/HEIGHT);
            pixel = rgby*640+rgbx;
        }
        else if((WIDTH==1920)&&(HEIGHT==1080)){
            cout << "Se procede a la conversion en 640x480 ..." << endl;

            int rgbx = ((px_rgb[i]*640)/WIDTH);
            int rgby = ((py_rgb[i]*480)/HEIGHT);

            pixel = rgby*640+rgbx;
        }
        else{
            cout << "\n\tLa resolucion no es correcta." << endl;
            error = 5;
        }
    }

    const pcl::PointXYZ& pt = msg->points[pixel];

    camera_point.header.frame_id = "camera_depth_optical_frame";
    camera_point.point.x = pt.x;
    camera_point.point.y = pt.y;
    camera_point.point.z = pt.z;

    // Llama a la funcion para transformar el punto obtenido
    transformPoint(boost::ref(listener));
}
}
```

Debido a que tanto las distintas versiones de la *Kinect* permiten mayores resoluciones de la imagen 2D, y puesto que se emplea mayor resolución para poder detectar mejor los códigos de barra o *QR*, es necesario realizar una conversión de los pixeles a la resolución de la nube de puntos que es de 640 por 480 pixeles. Seguidamente, mediante la librería *PointCloud* se obtienen los puntos correspondientes a los pixeles de la imagen 2D en la nube de puntos. Y, por último, tras obtener los puntos correspondientes, se llama al método “*transformPoint*”.

## TransformPoint

El método “*transformPoint*” transforma los puntos obtenidos referidos al sistema de referencia de la cámara tanto al sistema de referencia global del mapa como al sistema de referencia local situado en la base del robot.

```
void Detector::transformPoint(const tf::TransformListener& listener)
{
    try{
        listener.transformPoint("map", ros::Time(0), camera_point, "camera_depth_optical_frame", map_point);

        listener.transformPoint("base_link", ros::Time(0), camera_point, "camera_depth_optical_frame", base_link_point);

    }
    catch(tf::TransformException& ex){
        ROS_ERROR("Received an exception trying to transform a point from \
\"camera_point\" to \"map_point\": %s", ex.what());
        error = 1;
    }

    mapx.push_back(map_point.point.x);
    mapy.push_back(map_point.point.y);
    mapz.push_back(map_point.point.z);

    basex.push_back(base_link_point.point.x);
    basey.push_back(base_link_point.point.y);
    basez.push_back(base_link_point.point.z);

    // Comprobación para saber si el número obtenido es válido.
    // La función isnan devuelve true si es igual a NaN

    if((isnan(map_point.point.x))||(isnan(map_point.point.y))||(isnan(map_point.
    point.z))||(isnan(base_link_point.point.x))||(isnan(base_link_point.point.y)
    )||(isnan(base_link_point.point.z))){
        error = 4;
        ROS_ERROR("Se ha detectado uno de los puntos como NaN");
    }
}
```

Este método emplea para ello una clase llamada “*TransformListener*” incluida en la librería “*tf*”, que mantiene un seguimiento de la evolución de los distintos marcos de referencia del sistema y sus transformaciones a lo largo del tiempo. Además, posee a su vez una serie de métodos que permiten transformar puntos indicando el sistema de referencia origen y destino.

En caso de que la transformación se haya realizado con éxito, se comprueba que no se ha dado un falso positivo transformando un punto de valor nulo o “*NaN*”. En ese caso, se trataría como un error y se notificaría al servidor mediante la variable *error*.

## 4.4 Resultados

En primer lugar, tal y como se mencionó anteriormente, el nodo *barcode\_detector* permite obtener distintos tipos de códigos en una imagen y mostrar por pantalla la posición de dichos códigos y el contenido de estos, tal y como se muestra en el ejemplo de la Figura 23. Nótese que ambos códigos tienen un punto rojo señalado en el centro. Estos serán los puntos de la imagen que utilizará “*position\_detector*” para buscar sus correspondientes en la nube de puntos.



Figura 23 – Imagen mostrada por *barcode\_detector* tras detectar códigos.

Estos resultados han sido obtenidos a un metro de distancia empleando la segunda versión de la *Kinect*, con una resolución de imagen de 1920x1080, y utilizando códigos *QR* de un tamaño de 8 centímetros cuadrados, así como unos códigos de barra de 12x8 centímetros cuadrados, siendo estos los códigos más pequeños detectados con fiabilidad a esa distancia.

Sin embargo, la imagen de profundidad o nube de puntos tiene una resolución máxima de 640x480 por lo que esto implica que se pueden obtener puntos en la imagen 2D que no obtienen correspondencia en la nube de puntos. Para evitar este problema, dentro del nodo “*position\_detector*” se realiza una simple conversión del punto de la imagen de alta resolución a un punto aproximado en una imagen de menor resolución como se ha visto anteriormente. En principio puede parecer que se puede perder precisión, sin embargo, dado el tamaño y distancia a la que se pueden detectar los códigos, es extremadamente difícil que dos códigos coincidan con un mismo punto de la imagen de profundidad debido a la perdida de resolución.

En la Figura 24 se muestra un ejemplo visual en Rviz de una imagen de color obtenida por una cámara normal superpuesta a una nube de puntos. Conviene recordar que la nube de puntos no es más que una imagen donde cada pixel contiene la información de la posición en el espacio de dicho pixel. Por tanto, es fácil obtener la imagen de la figura simplemente “coloreando” cada punto del espacio definido en la nube de puntos según la información dada por la imagen de color.

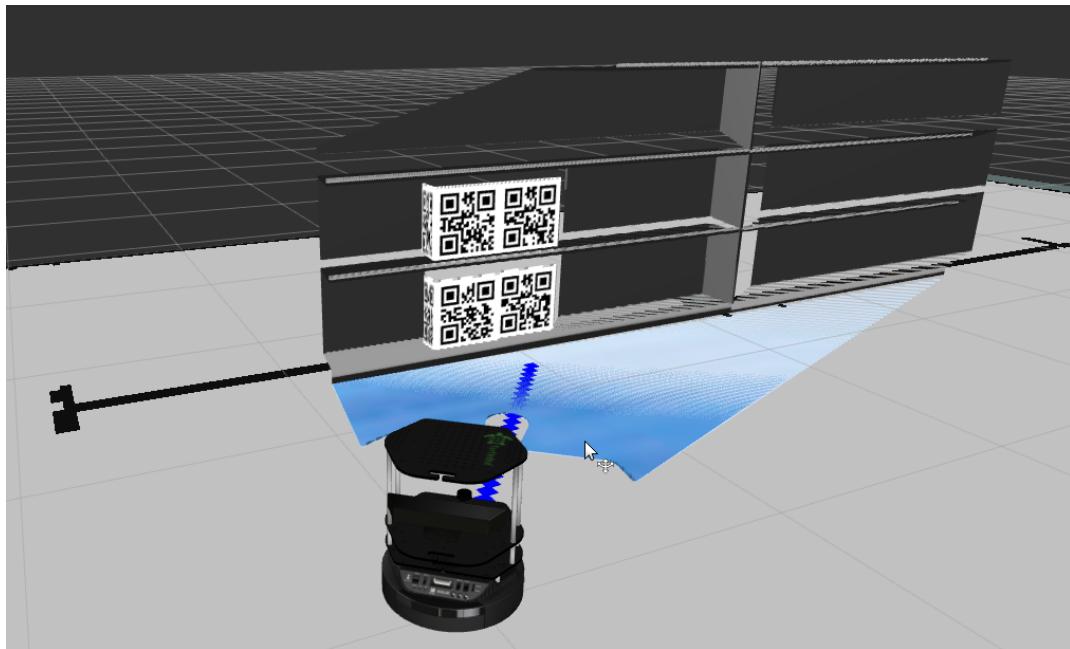


Figura 24 – Ejemplo de imagen y nube de puntos superpuestas en Rviz.

Para mostrar los resultados del servicio ofrecido por “*position\_detector*”, se ha tomado como escenario el simulador *Gazebo* utilizando el escenario de la Figura 24. En él se puede observar distintos códigos QR y el mapa en el que está localizado el robot *Turtlebot2*. En el momento de la llamada al servicio el robot estaba enfocando a dos de los cuatro códigos que se observan en la figura, obteniendo como respuesta el siguiente mensaje por terminal.

```
rosservice call /localiza_productos

header:
  seq: 0
  stamp:
    secs: 221
    nsecs: 760000000
  frame_id: ''
productos: 2
error: 0
data: ['Detergentes', 'Peras']
type: ['QR-Code', 'QR-Code']
map_x: [3.502, 3.521]
map_y: [-0.218, -0.210]
map_z: [0.205, 0.590]
base_x: [1.349, 1.368]
base_y: [-0.364, -0.372]
base_z: [0.205, 0.590]
```

En el mensaje se puede observar que se han detectado dos productos, “Detergentes” y “Peras”, junto a las coordenadas de ambos respecto al mapa y la base del robot. Debido a que la información esta almacenada en vectores, cada columna corresponde a un producto, siendo los puntos de la izquierda los correspondientes a “Detergentes”, y los de la derecha a “Peras”.

Además de esta información, el código de “*position\_detector*” cuenta con múltiples mensajes que se imprimen en el terminal, pudiendo hacer un seguimiento constante de los procesos que ocurren mientras se obtiene la respuesta del servidor. A continuación, se muestra un ejemplo de los mensajes que se imprimen por pantalla mientras se ejecuta el proceso.

```
#####
Recibida peticion del servidor...
Llamada al servidor numero 1

La resolucion de la camara RGB es de 640x480 pixeles
esperando 0 segundos de margen para que se estabilice la imagen ...

#####
Buscando codigos en la imagen RGB ...

Numero de intentos de lectura de Zbar 1 de 3 intento/s

Codigo/s detectado/s
Tam del mensaje recibido es 2
Tam del vector almacenado es 0
Codigo 0: [ Detergentes ]
Codigo 1: [ Peras ]

Numero de intentos de lectura de Zbar 2 de 3 intento/s

Codigo/s detectado/s
Tam del mensaje recibido es 1
Tam del vector almacenado es 2
El vector almacenado es mayor o igual que el nuevo vector recibido
Se desecha el mensaje recibido ...

Numero de intentos de lectura de Zbar 3 de 3 intento/s

Codigo/s detectado/s
Tam del mensaje recibido es 2
Tam del vector almacenado es 2
El vector almacenado es mayor o igual que el nuevo vector recibido
Se desecha el mensaje recibido ...

#####
Buscando puntos en la nube 3D ...

Punto obtenido de la nube 3D respecto a 'camera_depth_optical_frame':
( 0.377053 , 0.08163 , 1.43631 )
Punto procesado respecto a 'map':
( 3.50259 , -0.218505 , 0.205372 )
Punto procesado respecto a 'base_link':
( 1.34931 , -0.364551 , 0.205372 )
Punto obtenido de la nube 3D respecto a 'camera_depth_optical_frame':
( 0.384718 , -0.30331 , 1.45551 )
Punto procesado respecto a 'map':
( 3.52143 , -0.210013 , 0.590312 )
Punto procesado respecto a 'base_link':
( 1.36851 , -0.372217 , 0.590312 )

#####
```

```
#####
#####
```

Asignando datos para la respuesta del servidor

Código: Detergentes Tipo: QR-Code  
 [Map frame] x = 3.50259, y = -0.218505, z = 0.205372.  
 [Base\_link frame] x = 1.34931, y = -0.364551, z = 0.205372.

Código: Peras Tipo: QR-Code  
 [Map frame] x = 3.52143, y = -0.210013, z = 0.590312.  
 [Base\_link frame] x = 1.36851, y = -0.372217, z = 0.590312.

Respuesta del servidor construida  
 Enviando respuesta ...

El tiempo que se ha mantenido ocupado al servidor es de: 1.46 segundos  
 Respuesta válida [envía true]

```
#####
#####
```

Por último, en la Figura 25 se muestran los resultados en un entorno real con 5 códigos QR de 8 centímetros cuadrados a un metro de distancia.

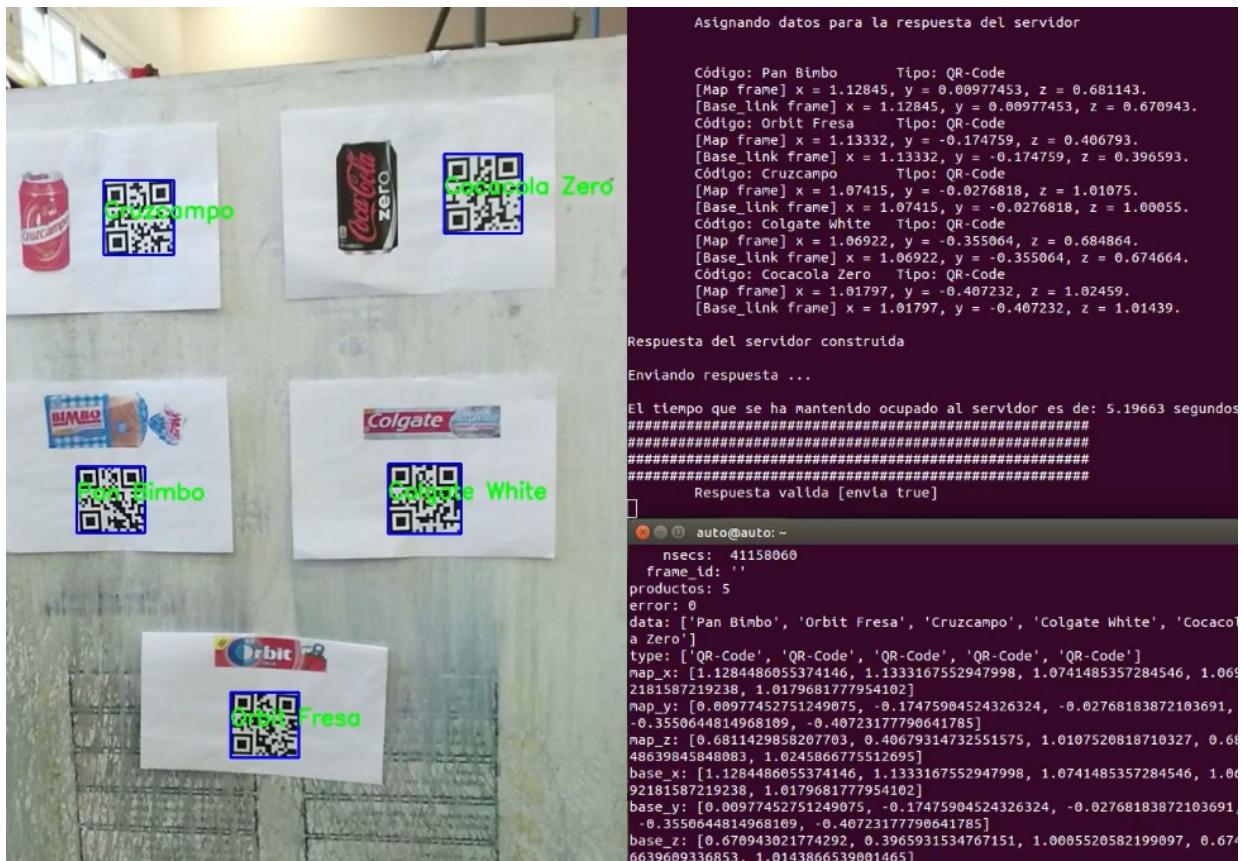


Figura 25 – Ejemplo sobre entorno real de barcode\_detector y position\_detector.

# 5 DETECCIÓN Y MANIPULACIÓN DE PRODUCTOS

---

*“Sólo podemos ver un poco del futuro, pero lo suficiente para saber que hay mucho por hacer”*

- Alan Turing -

Debido a los requisitos iniciales del proyecto, tras completar la localización de productos y gracias a la experiencia adquirida con la Kinect, se comenzó el estudio y desarrollo de un software que pudiera reconocer y manipular pequeños objetos con el brazo robótico incorporado en el *Turtlebot2*.

Inicialmente, el estudio consistió en la puesta a punto y programación del brazo robótico. Para ello, se empleó diferentes tipos de paquetes software desarrollados en ROS como “*turtlebot\_arm*” y “*moveit*” [28] [29]. Debido a la complejidad del software y los pobres resultados obtenidos con el *PhantomX Pincher*, se comenzó la búsqueda de paquetes alternativos que permitieran programar las trayectorias que debe seguir el brazo de manera más simple e intuitiva, como ocurre en el caso del paquete “*pypincher\_ros*” desarrollado por un grupo de la universidad técnica de *Dortmund* [30], el cual está especialmente desarrollado para el modelo *PhantomX Pincher*.

Este paquete permitió desarrollar diferentes rutinas para mover el brazo en distintas posiciones del espacio, así como controlar el nivel de apertura de la garra para poder manipular objetos en un nodo llamado “*move*”. Además, dicho nodo se implementó como un servidor *actionlib*, donde un nodo “*cliente*” sería el encargado de controlar el proceso de movimiento del brazo en todo momento.

Una vez que fue posible programar distintas rutinas para el brazo con éxito, el siguiente objetivo consistió en detectar y obtener la posición de un objeto que pudiera ser manipulado por el brazo. En primer lugar, se investigó las posibilidades de un paquete llamado “*find\_object\_2D*” [31] que tiene un funcionamiento parecido al paquete descripto en el capítulo anterior, sólo que no emplea códigos para detectar objetos, sino que realiza técnicas de reconocimiento de imágenes utilizando fotografías de una base de datos. Si en la imagen 2D percibida existe alguna coincidencia con las fotografías de los objetos almacenados, este paquete busca en la nube de puntos la posición de dicho objeto.

Sin embargo, este método cuenta con una serie de limitaciones por lo que se descartó su uso. Por un lado, la base de datos de imágenes implicaba hacer fotografías a un mismo objeto desde muchas perspectivas para que pudiera ser detectado con éxito. Mientras que, por otro lado, al emplear únicamente técnicas de reconocimiento 2D como *SIFT*, *SURF*, *FAST* y *BRIEF*, se pierde una de las características más importante que ofrece los sensores 3D como es la posibilidad de poder discriminar

entre objetos a distinta distancia con facilidad.

Aunque, dicho paquete suponía una solución al problema propuesto, se descubrió más adelante las bondades de otro paquete conocido como ORK (*Object Recognition Kitchen*) [32], el cual integra una serie de técnicas que permiten reconocer objetos con facilidad, empleando tanto técnicas de reconocimiento 2D como 3D. Por el contrario, el principal hándicap que implica la utilización de esta aplicación es la necesidad de crear una base de datos de modelos de objetos 3D para poder reconocer objetos. Aún así, con todo ello, los resultados obtenidos por ambos paquetes decantaban la balanza hacia el lado de ORK.

Finalmente, se adaptó el nodo “*cliente*” mencionado anteriormente para obtener las posiciones de los objetos mediante el paquete ORK. El nodo fue renombrado como “*find\_and\_pick*” y además se diseñó como otro servicio para poder ser invocado por otro nodo o manualmente tantas veces como fueran necesarias.

## 5.1 Tecnologías empleadas

Seguidamente, se describen las diferentes tecnologías empleadas y aquellos conceptos útiles necesarios para la elaboración de este paquete software. Además, se obviarán otros conceptos como nube de puntos o el software PCL que, si bien son empleados en este paquete también, ya han sido descritos en el capítulo anterior.

### 5.1.1 ORK

ORK son las siglas en inglés de “*Object Recognition Kitchen*”. Se trata de una herramienta desarrollada por *Willow Garage*, independiente de ROS, aunque posee una interfaz para trabajar con ROS. [32]

ORK está diseñado de tal manera que los algoritmos de reconocimiento de objetos se puedan dividir en distintos módulos. Implementa algunos métodos de reconocimiento de objetos diferentes y proporciona una interfaz, llamada “*pipeline*” o tubería, a la que se pueden adaptar nuevos algoritmos. Cada “*pipeline*” necesita de una entrada o “*source*” donde obtiene datos, un procesamiento de imagen adecuado, y un receptor donde se emiten los datos.

Al realizar la detección de objetos, el ORK puede ejecutar múltiples pipelines en paralelo para mejorar los resultados. Los pipelines incorporados son *LINE-MOD*, *Tabletop*, *TOD* y *Transparents Objects*.

- **LINE-MOD** es un pipeline que implementa el algoritmo del mismo nombre, para el reconocimiento genérico de objetos rígidos.
- **Tabletop** es una versión portada del paquete “*tabletop object detector*” de ROS, y el cual se detallará más adelante.
- **TOD** son las siglas de “*Textured Object Detection*” y como su nombre indica realiza la detección de objetos texturados, haciendo coincidir las superficies con una base de datos de texturas conocidas.
- **Transparents Objects** (objetos transparentes) es similar al “*Tabletop Object Detector*”, pero trabaja sobre objetos transparentes como tazas de plástico o vidrio.

De los cuatro pipelines de los que consta este paquete, sólo se ha empleado “*Tabletop*”, puesto que proporciona la posibilidad de detectar fácilmente objetos situados en superficies planas que puedan ser manipulados por un robot, siendo este el propósito principal de esta parte del proyecto.

### 5.1.2 Tabletop

*Tabletop Object Detector* (detector de objetos en la superficie de una mesa) o simplemente “*Tabletop*”, es una librería originalmente desarrollada por *Willow Garage* para su robot de investigación, PR2. El propósito de este paquete es proporcionar un medio para reconocer objetos domésticos sobre una superficie plana (como una mesa) que puedan ser manipulados eficazmente. [33]

La detección del objeto consta de dos partes claramente diferenciadas. Por un lado, se necesita segmentar el objeto, es decir, diferenciar que parte de los datos corresponde con el objeto y cuáles no. Por otro lado, es necesario reconocer de qué objeto se trata. [34]

Respecto a la segmentación, el software asume que:

- Los objetos se encuentran sobre una mesa, que será el plano dominante de la escena.
- La distancia mínima entre dos objetos será de 3 cm.

En el caso del reconocimiento de objetos, el software sólo puede manejar 2 grados de libertad a lo largo de los ejes X e Y (el eje Z se asume apuntando hacia "arriba" en relación con la mesa). Por lo tanto, para reconocer un objeto:

- Debe ser simétrico respecto al eje Z
- Debe tener una orientación conocida, tal como un vaso o un recipiente sentado "en posición vertical" sobre la mesa.

Hay que tener en cuenta que, aunque este paquete funciona bien para el robot PR2, para el cual fue desarrollado originalmente, tiene algunas limitaciones notables:

- No puede detectar objetos colocados en el suelo, porque espera la presencia de un plano de una mesa.
- Es muy sensible a los cambios de perspectiva de una tabla o superficie observada.

### Segmentación

En primer lugar, dada una nube de puntos de un sensor 3D como la *Kinect*, se realiza la detección de superficies planas buscando planos dominantes sobre la nube de puntos. Para ello se hace uso del algoritmo RANSAC (*Random sample consensus*) como se muestra en la Figura 26. Una vez obtenida la superficie, el algoritmo filtra la nube de puntos original para eliminar todos los puntos que no se encuentran directamente sobre dicha superficie. [34]

Los puntos obtenidos por encima de la superficie se consideran pertenecientes a objetos que puedan ser manipulados. Por tanto, dichos puntos se agrupan en distintos objetos discretos usando la agrupación de vecinos más cercanos con un árbol *Kd*. Este proceso produce una secuencia de nubes de puntos independientes para cada objeto, llamadas “*clusters*” o cúmulos.

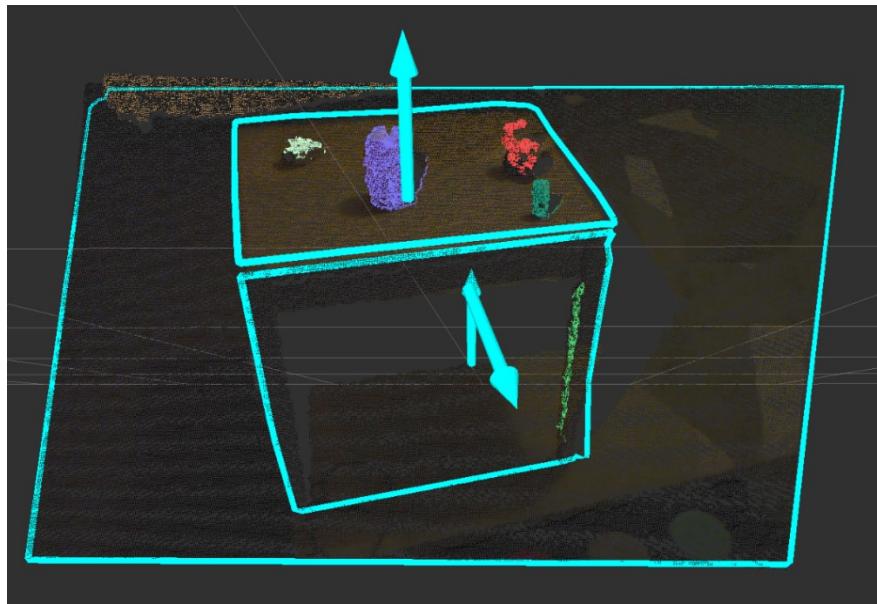


Figura 26 – Reconocimiento de una superficie mediante segmentación.

### Reconocimiento

Una vez se tiene segmentados los posibles objetos, se pasa al reconocimiento de los mismos. En este paso se utiliza la técnica de “adaptación iterativa simple” para cada cúmulo de puntos, el cual es un algoritmo cercano al algoritmo ICP (*Iterative Closest Point*). Dicho algoritmo compara la nube de puntos obtenida con un modelo 3D del objeto almacenado en una base de datos, como el que se muestra en la Figura 27. Esta base de datos se almacena de forma local en el PC que ejecute el nodo ORK. Para añadir nuevos modelos en la base de datos es necesario seguir una serie de pautas descritas en el tutorial indicado en la bibliografía [35].

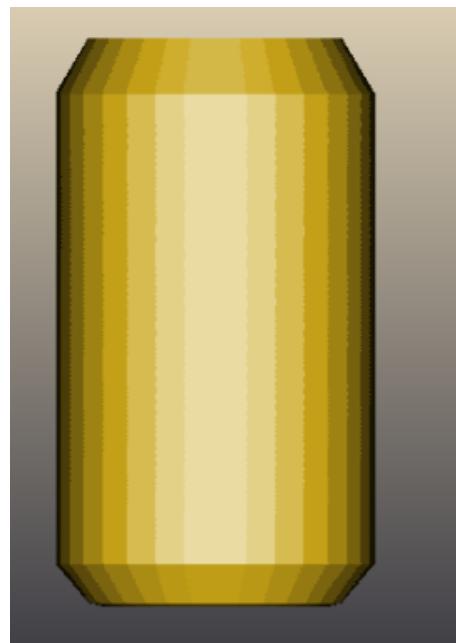


Figura 27 – Modelo 3D de una lata de refresco en la base de datos.

Si se obtiene una comparación satisfactoria del objeto, a modo de resultado se muestra en pantalla la ID del objeto y el punto de agarre junto con el “cluster” comparado. Dicho punto es traducido a las coordenadas del brazo para realizar la tarea de “*pick and place*”. En la Figura 28 se puede apreciar tanto la ID del producto, la nube de puntos del objeto resaltado y el punto de agarre, representado como unos ejes de coordenadas.



Figura 28 – Representación del objeto detectado usando Tabletop.

### 5.1.3 Actionlib

Los servicios en ROS permiten la interacción entre dos nodos mediante el modelo petición/respondida, pero tiene el inconveniente de que si la tarea se demora demasiado o el servidor no la ha finalizado se necesite esperar hasta que se complete. [36]

Para evitar este problema, en ROS se desarrolló otro método similar con la posibilidad de conocer el estado del servicio y cancelarlo si es necesario. Este método es conocido como “Actionlib” y permite implementar este tipo de tareas preventivas de manera estándar. Este paquete es muy utilizado sobretodo en la navegación de robots móviles.

Al igual que los servicios o los mensajes en ROS, en un actionlib se debe especificar el contenido o información que se intercambiarán los distintos nodos implicados. Esta información se almacena en un archivo de extensión “.action” que debe estar dentro de la carpeta “action”. Este archivo está dividido en distintas partes según su funcionalidad:

- **Goal:** En este campo se define la petición que el cliente le envía al servidor, similar a los servicios en ROS.
- **Feedback:** Cuando un cliente le envía una petición al servidor, este comenzará a ejecutar una función de devolución de llamadas. Este campo define por tanto la información intercambiada durante el proceso de retroalimentación del *actionlib*.
- **Result:** Tras completar el servicio, el servidor enviará el resultado de la operación que puede ser tanto un parámetro o mensaje, como un acuse de recibo.

### 5.1.4 Pxpincer

*Pxpincer\_ros* es un metapaqute desarrollado por el grupo de sistemas de control de la universidad técnica de *Dormunt*. Está pensado para simplificar el proceso de configuración de un brazo manipulador con fines educativos principalmente, por lo que utiliza una versión modificada del paquete “*turtlebot\_arm*”.

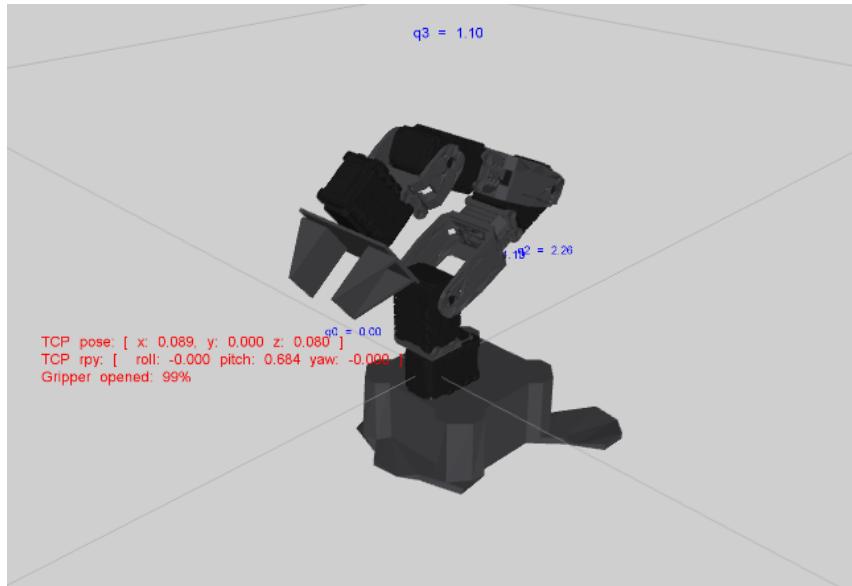


Figura 29 – Simulación del brazo PhantomX Pincher mediante el paquete *pxpincher\_ros*.

Este metapaqute proporciona los drivers y una serie de herramientas y librerías para controlar y simular el brazo *PhantomX Pincher*. La librería que ofrece *pxpincher\_ros* está programada en C++ y diseñada como servicios “*actionlib*” para controlar el brazo. [30]

## 5.2 Arquitectura del software

El paquete “detección y manipulación de productos” consta de dos etapas claramente diferenciadas en las que emplea dos dispositivos diferentes, el sensor 3D *Kinect* y el brazo manipulador, los cuales constan de diferentes nodos y drivers. Además, al igual que en el capítulo anterior, se ha desarrollado un nodo que trabaja como director y que controla las acciones de los distintos nodos para que la tarea se lleve a cabo en el orden apropiado. Este nodo conocido como “*find\_and\_pick*” se define como un servidor el cual será invocado con la siguiente instrucción:

- *rosservice call /find\_and\_pick*

La respuesta a dicho servicio está definida en el mensaje “*find\_and\_pick.srv*”. Las interacciones y conexiones entre los distintos nodos y dispositivos que componen este software vienen definida en la Figura 30.

En primer lugar, respecto a la detección de objetos, el nodo “*detection*” recibe del driver de la *Kinect* los topics correspondiente a la nube de puntos y la imagen RGB. Este nodo, perteneciente al paquete ORK, es el encargado de detectar los objetos haciendo uso de la base de datos. En caso de detectar objetos, este nodo envía por el topic “*recognized\_object\_array*” la información referida al tipo y posición de los objetos detectados.

Esta información es recibida y filtrada por el nodo “*object\_recognition\_kitchen*”, el cual envía por el topic “*object\_position*” la posición de un solo objeto en cada ocasión. Este nodo pertenece a un paquete conocido como “*pcl\_recognition*” [37] que inicialmente tenía como fin imprimir por pantalla la información de los objetos detectados. Sin embargo, se aprovechó y modificó en este proyecto para servir como puente entre ORK y el nodo servidor “*find\_and\_pick*”. Asimismo, se definió el mensaje que se iba a enviar por el topic en el fichero “*object\_position.msg*”.

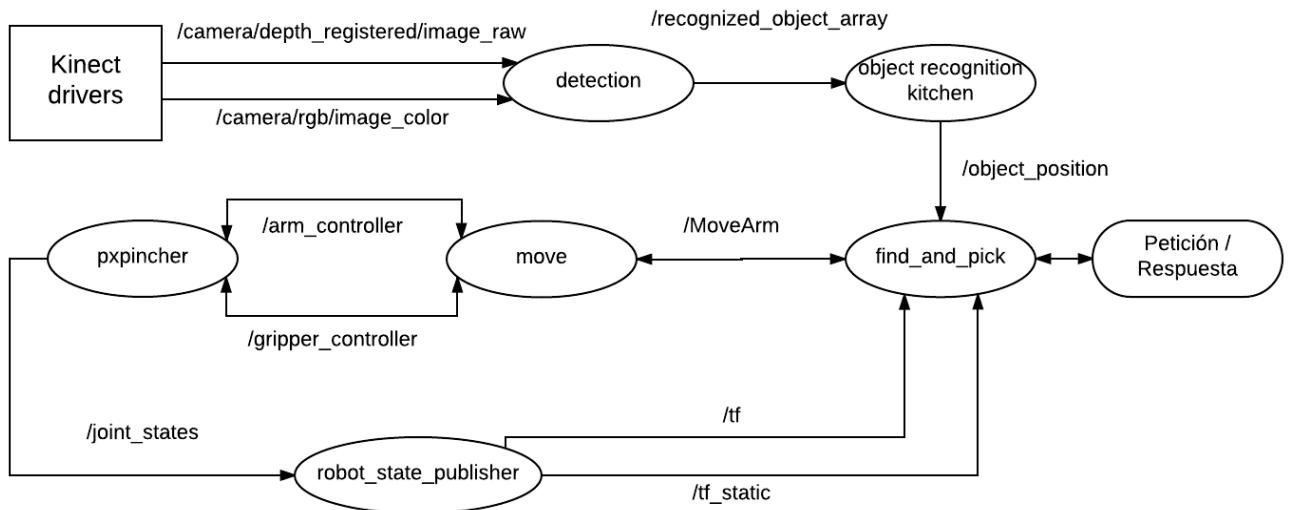


Figura 30 - Grafo de procesos de la etapa de “detección y manipulación de objetos”.

Este mensaje es recibido por el nodo principal, “*find\_object\_detector*”. Sabiendo la posición del objeto respecto a la cámara, este nodo transforma el punto respecto a la base del brazo e informa al nodo “*move*” para que se comunique con el driver del brazo y pueda recogerlo. El nodo “*move*” es el encargado de iniciar y mover el brazo mediante las distintas funciones disponibles en el paquete “*pxpincher\_ros*”. La comunicación entre “*move*” y el nodo principal se realiza mediante un *actionlib* llamado “*MoveArm*”.

Por el otro lado, respecto al brazo, “*move*” se comunica con el driver llamado “*pxpincher*” mediante dos *actionlib* llamados “*arm\_controller*” y “*gripper\_controller*”. Este driver implementa las funciones que permiten mover cada servo del robot, así como la planificación de trayectorias del brazo. Si un punto no puede ser alcanzar, el driver informará de vuelta a “*move*”, y éste a su vez al nodo principal. Simultáneamente, “*pxpincher*” publica en el topic “*joint\_state*” el estado de las articulaciones del brazo, haciendo que el nodo “*robot\_state\_publisher*” pueda publicar el estado de los marcos de referencia o “*tf*” del brazo junto con el resto del robot. Esta información es obtenida por el nodo principal previamente para transformar el punto obtenido por la *Kinect*.

Por último, cuando el proceso haya acabado, independientemente de si se consiguió realizar la tarea o no, el servidor “*find\_and\_pick*” envía la respuesta al nodo que le invocó inicialmente.

## 5.3 Descripción del código

En este punto se describirá el código empleado en este paquete haciendo hincapié en la funcionalidad de cada parte del código más que en las funciones empleadas en ellas. En primer lugar, se detallan los campos de los mensajes y servicios definidos, y posteriormente se explicará paso a paso los procesos que siguen los nodos *move* y *find\_and\_pick* para obtener la posición del objeto y mover el brazo manipulador.

### 5.3.1 Mensajes y servicios

A continuación, se mostrará la definición de los distintos mensajes definidos para esta aplicación.

#### **Find\_and\_pick.srv**

Este fichero define la respuesta enviada por el nodo “*find\_and\_pick*” cuando se recibe una petición. Este tipo de mensajes pertenecientes a un servicio viene separados por tres guiones indicando los campos de entrada y de salida que espera el servidor. En este caso, al solo contener campos por debajo de dichos separadores, los datos del mensaje son solo de respuesta. Los diferentes campos del mensaje son:

- **header**

Se trata de una “cabecera” del tipo *Header* definido por la comunidad de ROS y que contiene diferente información que pueda ser útil sobre el estado del mensaje.

- **status**

Indica el estado en el que finalizó el servicio mediante un numero entero.

- **description**

Contiene en un string la información correspondiente al estado del servicio.

La definición de los distintos campos del mensaje dentro del fichero es la siguiente:

```
---
```

Header	header
int32	status
string	description

#### **Object\_position.msg**

Este mensaje fue añadido al paquete “*pcl\_recognition*” para poder obtener la posición de los objetos por el nodo principal.

- **header**

Se trata de una “cabecera” del tipo *Header* definido por la comunidad de ROS y que contiene diferente información que pueda ser útil sobre el estado del mensaje.

- **name**

Este campo contiene el nombre almacenado en la base de datos del objeto reconocido.

- **id**

Similar al campo anterior, contiene el id almacenado en la base de datos del objeto.

- **pos\_x**

Indica la posición del objeto en el espacio en la coordenada x respecto al marco de referencia de la Kinect. El punto se almacena en una variable de tipo flotante medido en metros.

- **pos\_y**

Indica la posición del objeto en el espacio en la coordenada y respecto al marco de referencia de la Kinect. El punto se almacena en una variable de tipo flotante medido en metros.

- **pos\_z**

Indica la posición del objeto en el espacio en la coordenada z respecto al marco de referencia de la Kinect. El punto se almacena en una variable de tipo flotante medido en metros.

La definición de los distintos campos del mensaje dentro del fichero es la siguiente:

Header	header
string	name
string	id
float32	pos_x
float32	pos_y
float32	pos_z

### **MoveArm.action**

El fichero *MoveArm* contiene la información que se intercambian los nodos “move” y “find\_and\_pick” durante todo el proceso. En primer lugar, se define los parámetros necesarios para establecer el inicio de la operación, y a continuación, el mensaje resultante al finalizar. Por último, se definen aquellos campos pertenecientes a la retroalimentación del *actionlib*.

- **task**

Contiene el número de la tarea que se desea realizar.

- **N**

Según la tarea asignada, este campo indica un punto predefinido donde situar el brazo o el punto donde se ha detectado un objeto.

- **X**

Indica la posición del objeto en el espacio en la coordenada x respecto a la base del brazo manipulador. El punto se almacena en una variable de tipo flotante medido en metros.

- **Y**

Indica la posición del objeto en el espacio en la coordenada x respecto a la base del brazo manipulador. El punto se almacena en una variable de tipo flotante medido en metros.

- **Z**

Indica la posición del objeto en el espacio en la coordenada x respecto a la base del brazo manipulador. El punto se almacena en una variable de tipo flotante medido en metros.

- **Pitch**

Indica la orientación de *pitch* o “cabeceo” de la garra con la que se desea atacar al objeto. Esta variable es de tipo flotante y se almacena en radianes.

- **result**

Este campo indica mediante un número de tipo entero el resultado de la tarea.

- **status**

Informa del estado del servicio mediante un número entero mientras se realiza la operación.

La definición de los distintos campos del mensaje dentro del fichero es la siguiente:

```
# Define the goal
uint32      task
uint32      N
float32     X          # Punto X (en metros)
float32     Y          # Punto Y (en metros)
float32     Z          # Punto Z (en metros)
float32     Pitch       # Pitch   (en radianes)
---

# Define the result
uint32 result
---
```

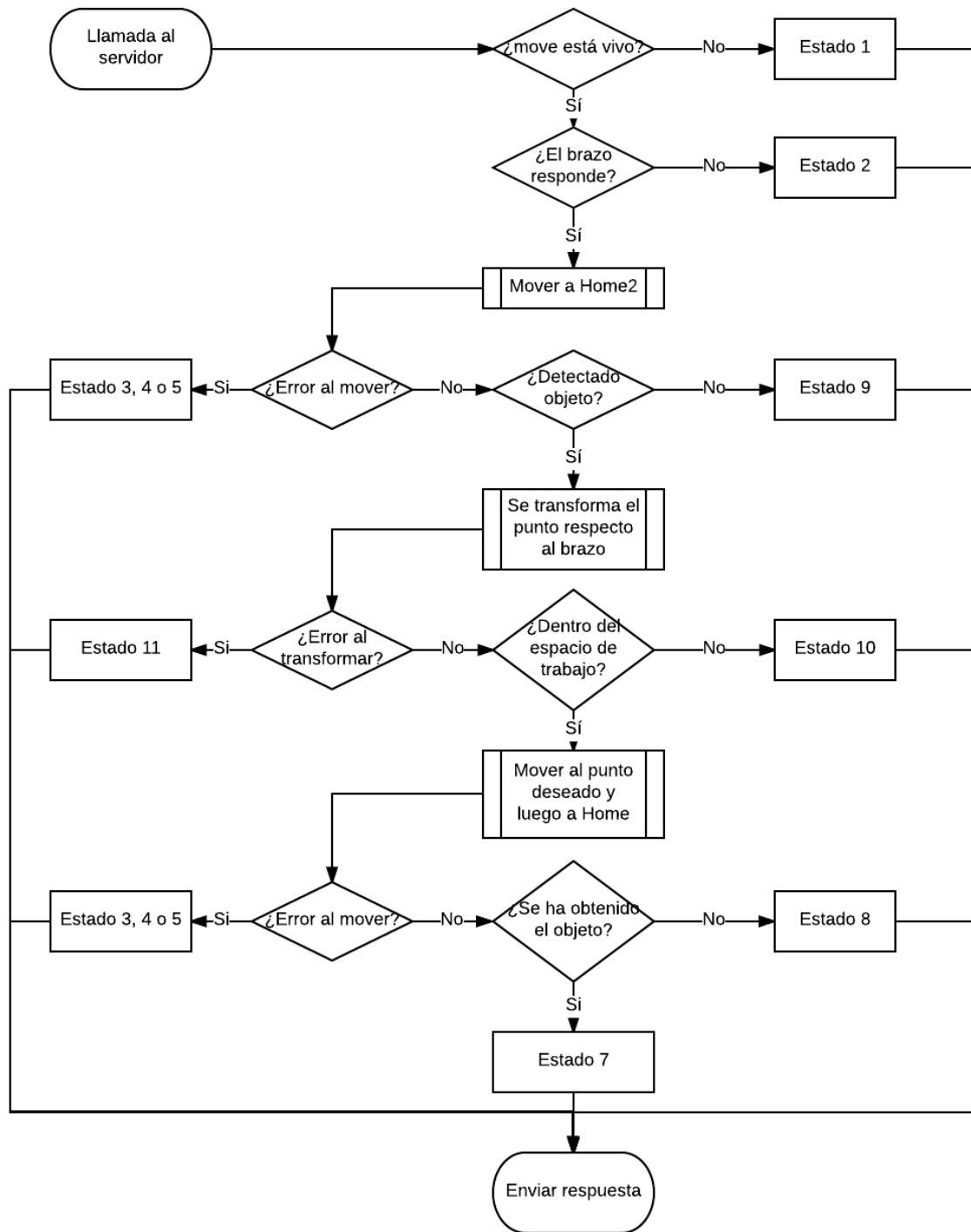
### 5.3.2 Nodo `find_and_pick`

El nodo `find_and_pick` actúa como servidor que arranca la etapa de detección y manipulación de objetos cuando es invocado. Este nodo es el encargado de gobernar al resto de nodos implicados en el proceso y de comprobar el estado de los mismos, respondiendo en un mensaje al cliente del servicio según se desarrolle la operación.

`Find_and_pick` fue diseñado como un objeto llamado “*Servicio*”, el cual incorpora una serie de funciones que se expondrán a continuación. La función principal de este objeto se llama “servidor” cuyo funcionamiento viene descrito en el diagrama de flujo de la Figura 31, y se ejecuta cuando es recibida la petición del servicio “`find_and_pick`”.

Al contrario que en el capítulo anterior, este servicio no ofrece información útil al cliente como parámetro de salida, sino que informa del estado en el que acabó la tarea por la que fue invocado, siendo esta última su principal función. Es por ello que el mensaje con el cual responde este servicio únicamente contiene los campos de estado y descripción del estado.

Además de funcionar como un servidor, este nodo incorpora a su vez un cliente *actionlib* para comunicarse con el nodo `move`, con el objetivo de poder ejecutar los comandos que le permitan mover el brazo a las distintas posiciones según sea necesario.

Figura 31 – Diagrama de flujo del servidor “*find\_and\_pick*”

## Definiciones y dependencias

Al inicio del fichero “*find\_and\_pick.cpp*” se encuentra definidas las dependencias del nodo, un *timeout* o tiempo de vencimiento de 15 segundos y la instrucción *typedef* para acortar el nombre del cliente *actionlib* a simplemente “*Client*”. Además, se incluye los *namespace std*” y *pxpincher\_lib*.

```
#include <ros/ros.h>
#include <tf/transform_listener.h>
#include <tf/transform_broadcaster.h>
#include <pxpincher_lib/MoveArmAction.h>
#include <actionlib/client/simple_action_client.h>
#include <pxpincher_lib/find_and_pick.h>
#include <pcl_recognition/object_position.h>
#include <Eigen/Geometry>

typedef actionlib::SimpleActionClient<pxpincher_lib::MoveArmAction> Client;

using namespace std;
using namespace pxpincher_lib;

#define TIMEOUT 15
```

## Main

La función *main* es la primera función que se invoca al lanzar el ejecutable de este fichero. En esta ocasión, la función main inicia el nodo “*find\_and\_pick*” y llama a la función *Run()*, la cual se encargará de crear el objeto Servicio y ejecutar la instrucción “*spin*” de ROS.

```
int main(int argc, char** argv)
{
    ros::init(argc, argv, "find_and_pick");

    Run();

    return 0;
}
```

## Run

Esta función se encarga de crear el bucle infinito que comprueba el estado de la red de ROS. De esta manera, tras crear el objeto “Servicio” se define la frecuencia con la que se comprobará el estado de la red referida en Hz. Para este caso ha sido definido como 10 Hz, es decir, cada 100 milisegundos.

Tras ello, se entra en un bucle que se ejecutará continuamente mientras la red de ROS funcione o se teclee *Ctrl-C* en un terminal. Este bucle comprueba la red con la instrucción *ros::spinOnce()*, y en caso de no haber recibido una petición para el servidor, éste se bloquea hasta que se cumpla el tiempo antes definido. Finalmente, en caso de que se salga del bucle, se moverá a el brazo a la posición de home y se cierra el nodo.

```

static void Run()
{
    Servicio servicio;
    ros::Rate r(10);

    while(ros::ok()){
        ros::spinOnce();
        r.sleep();
    }

    cout << "Moviendo a HOME para finalizar" << endl;
    servicio.enviaGoal(2,3);
    cout << "Cerrando el nodo." << endl;
    ros::shutdown();
}

```

La razón por la que se ha implementado de esta manera en vez de usar la instrucción `ros::spin()`, es debido a que se quería tener control del nodo antes de cerrarlo y para poder enviar el brazo a una posición segura.

### Clase Servicio

Debido a las ventajas que ofrece la programación orientada a objetos, en esta ocasión también se decidió programar este nodo como una clase a la que se le llamó “Servicio”. Esta clase contiene una serie de métodos declarados como públicos, entre las que destaca la función servidor encargada de gestionar todo el proceso.

```

class Servicio
{
public:
    Servicio();
    //Servidor
    bool servidor(pxpincer_lib::find_and_pick::Request &req,
pxpincher_lib::find_and_pick::Response &res);
    // Funciones
    void objectsDetected(const pcl_recognition::object_position::ConstPtr&
objects_msg);
    void enviaGoal(int task , int N);
    bool is_inside_workspace(double X, double Y, double Z);
    // Callbacks
    void doneCb(const actionlib::SimpleClientGoalState& state,
                const MoveArmResultConstPtr& result);
    void activeCb();
    void feedbackCb(const MoveArmFeedbackConstPtr& feedback);
}

```

También se define una serie de variables de acceso privado, es decir, solo son accesible desde los propios métodos de la clase Servicio. De esta manera todas las variables son visibles para todas las funciones de la clase, pero no así de manera externa.

```

private:
    ros::NodeHandle n,s;

    bool ready;
    int estado;
    int realim;
    bool finished_before_timeout;
    double t_inicio;
    double t_actual;

    geometry_msgs::PointStamped camera_point;
    geometry_msgs::PointStamped arm_base_point;

    tf::TransformListener listener;
    tf::TransformBroadcaster bpoint;
    tf::Transform transform;

    ros::ServiceServer server;
    pxpincher_lib::find_and_pick status;
    pxpincher_lib::MoveArmGoal goal;
    Client client;
};


```

## Constructor

El constructor es la primera función que se llama cuando se declara un objeto y permite definir los parámetros del objeto. En esta ocasión se definen las diferentes variables implicadas en el proceso, así como el servicio “*find\_and\_pick*” como el actionlib “*MoveArm*” encargado de la comunicación entre este nodo y el nodo *move*.

```

Servicio::Servicio():
    estado(0),
    ready(false),
    realim(0),
    finished_before_timeout(false),
    t_inicio(0),
    t_actual(0),
    client("MoveArm", true)
{
    server = s.advertiseService("find_and_pick", &Servicio::servidor,
this);
}

```

## Callbacks

Los “*callbacks*” son las funciones que permiten conocer el estado de los *actionlib* y se ejecutan siempre que el nodo *move* envíe información de vuelta al nodo *find\_and\_pick*. En este caso, se han definido tres callbacks según su objetivo. *ActiveCb* indica cuando un objetivo o *goal* del *actionlib* es activado, *feedbackCb* recibe información durante el desarrollo del proceso, y por último, *doneCb* se ejecuta una vez termina el *goal*.

```

// Es llamado cuando el objetivo finaliza
void Servicio::doneCb(const actionlib::SimpleClientGoalState& state,
                      const MoveArmResultConstPtr& result){
    ROS_INFO("Goal completed");
}

// Es llamado cuando el objetivo o goal se activa
void Servicio::activeCb() {
    ROS_INFO("Goal just went active");
}

// Es llamado cada vez que se recibe feedback
void Servicio::feedbackCb(const MoveArmFeedbackConstPtr& feedback) {
    realim = feedback->status;
    ROS_INFO("Got Feedback, realim: %d", realim);
    ROS_INFO("Status Feedback: %d", feedback->status);

    if(realim == 6) // Problemas con los servos
        estado = 6;

    ready = true; // Se ha recibido feedback
}

```

## Servidor

Esta función se encarga de orquestar las diferentes tareas necesarias para llevar acabo la detección y manipulación de objetos y es activada tras recibir una petición “*find\_and\_pick*”. Tan pronto como es ejecutada, hace una espera activa para comprobar que el *actionlib* del nodo *move* está activo.

```

bool Servicio::servidor(pxpincer_lib::find_and_pick::Request
&req,pxpincer_lib::find_and_pick::Response &res)
{
    cout << "\n\t#####\n" << endl;
    cout << "\t##### Recibida peticion #####\n" << endl;
    cout << "\t#####\n" << endl;

    t_inicio = ros::Time::now().toSec(); // obtiene el tiempo de inicio
    t_actual = ros::Time::now().toSec(); // obtiene el tiempo actual

    // ===== Llamadas action_lib ===== //
    ROS_INFO("Esperando al servidor ...");
    finished_before_timeout = client.waitForServer(ros::Duration(TIMEOUT));
    if (!finished_before_timeout)
    {
        ROS_INFO("Action did not finish before the time out.");
        estado = 1;
    }
}

```

A continuación, se comprueba que el brazo responde correstamente pidiendo que abra la garra mediante la función “enviaGoal” que se mostrará más adelante. Si ha respondido correctamente, en el paso siguiente, se envía el brazo a una posición llamada HOME2 para dejar visibilidad a la cámara y que pueda detectar objetos. Esto es debido a la posición en la que está colocado el brazo respecto a la cámara, pudiendo el brazo interferir en el campo de visión de la cámara. En la Figura 11 se puede ver al brazo en la posición de HOME2 junto a la cámara inclinada para detectar objetos cercanos al robot.

```
// === Task 1 --> Llamada para saber si el brazo esta inicializado

if(estado == 0)
    enviaGoal(1,0);

// === Task 2; Goal 4 --> Se lleva a la posicion HOME2 para no molestar
// a la visioón de la cámara

if(estado == 0)
    enviaGoal(2,4);

// ===== Busqueda de un objeto ===== //

pcl_recognition::object_position::ConstPtr msg =
ros::topic::waitForMessage<pcl_recognition::object_position>("/object_position",
ros::Duration(TIMEOUT));

if(msg){
    cout<<"\tObjeto/s detectado/s" << endl;
    // Conversión a los puntos XYZ para el mensaje actionlib
    objectsDetected(msg);
}
else{
    cout<<"\tNo se ha encontrado ningún objeto en "<< TIMEOUT << "
segundos" << endl;
    estado = 9;
}

// Se comprueba si el punto se encuentra dentro del espacio de trabajo
if(is_inside_workspace(goal.X,goal.Y,goal.Z) == false)
    estado = 10;

// === Task 2; Goal 2 --> Si el brazo esta inicializado se mueve
// primero al punto de aproximacion y luego al punto deseado

if(estado == 0) //Punto deseado
    enviaGoal(2,0);

// === Task 2; Goal 3 --> Se lleva a HOME si no se ha detecctado ningun
// objeto o ha habido error

if((estado == 3) || (estado == 5) || (estado == 9) || (estado == 10)){
    ROS_WARN("Error al intentar obtener posicion del objeto");
    enviaGoal(2,3);
}
```

A continuación, se hace una espera de “*Timeout*” segundos para recibir una respuesta del nodo *object\_recognition\_kitchen* con la posición del posible objeto detectado. En caso afirmativo, debido a que la posición viene indicada respecto al marco de referencia de la cámara, es necesario llamar a la función *objectsDetected* para convertir el punto recibido al marco de referencia de la base del brazo. Tras ello, se llama a una nueva función denominada *is\_inside\_workspace* que comprobará si el punto transformado forma parte del espacio de trabajo del brazo para poder alcanzarlo.

En el caso de que todo haya salido bien y la tarea se pueda realizar, se procede a enviar el punto detectado al nodo *move* para que coja el objeto y lo guarde. En caso contrario, se envía la orden de mover el brazo a la posición de HOME.

```
// Comprobación del feedback para saber si se ha obtenido un objeto

if(estado == 0){
    // Se espera al feedback en caso de que no se haya recibido aun
    if((realim!=4)&&(realim!=5)){
        ROS_INFO("Objeto atrapado? realim es %d y debe ser 4 o 5.", realim);
        ros::Rate r(10);
        int cont = 0;

        while (!ready){ ////
            ROS_INFO("Esperando al feedback ...");
            ros::spinOnce();
            r.sleep();
            cont++;

            if(cont == 3)
                ready = true;
        }
    }

    if(realim == 4)
        estado = 7;
    else if(realim == 5)
        estado = 8;
    else
        ROS_WARN("La variable realim es %d (Distinta de 4 o 5 como se esperaba). Por tanto no se ha recibido feedback a tiempo", realim);
}

// Se finaliza la tarea moviendo a la posición HOME3
enviaGoal(2,5);
```

Más adelante se comprueba el *feedback* o realimentación obtenido gracias al *actionlib* para ver si se ha obtenido el objeto o no, o incluso si ha ocurrido algún pequeño error. Tras ello, se mueve el brazo a la posición de HOME3, y se construye la respuesta del servicio indicando el estado en el que ha acabado la operación, así como la descripción de dicho estado.

Finalmente, se resetea el valor de la variable *estado* con el propósito de esperar una nueva llamada al servidor y repetir el proceso.

```

res.status = estado;
res.header.stamp = ros::Time::now();

switch(estado) {
case 0:
    res.description = "No se ha obtenido feedback del objeto.";
    break;
case 1:
    res.description = "El servidor no está inicializado o ha muerto.";
    break;
case 2:
    res.description = "El brazo no está inicializado.";
    break;
case 3:
    res.description = "El brazo no responde.";
    break;
case 4:
    res.description = "El brazo tarda demasiado en responder. Se cancela la operacion.";
    break;
case 5:
    res.description = "Error al computar la cinematica inversa. El punto no se puede alcanzar.";
    break;
case 6:
    res.description = "Existe un error en la lectura de la apertura del gripper/servos.";
    break;
case 7:
    res.description = "No ha habido errores. Se ha atrapado un objeto.";
    break;
case 8:
    res.description = "No ha habido errores. No se ha conseguido atrapar el objeto.";
    break;
case 9:
    res.description = "No se ha detectado ningun objeto.";
    break;
case 10:
    res.description = "La camara ha detectado un objeto pero no ha conseguido obtener el punto XYZ.";
    break;
case 11:
    res.description = "Se ha producido un error intentando transformar el punto del objeto detectado, al frame de la base del brazo.";
    break;
default:
    res.description = "La variable error ha tomado un valor no contemplado";
    break;
}

t_actual = ros::Time::now().toSec();
ROS_INFO("El tiempo que se ha mantenido ocupado al servidor ha sido %f segundos.", t_actual-t_inicio);

// Inicializamos estado a 0 de nuevo para una nueva llamada
estado = 0;
cout << "\t Enviando respuesta ... se espera nueva " << endl;
return true;
} // Fin del servidor

```

## EnviaGoal

Esta función se encarga de enviar una acción o goal al cliente del *actionlib*, en este caso el nodo *move*. Recibe como parámetros los campos *task* y *N*, que son enviados al cliente según la tarea que se desea realizar. *Task* indica el tipo de tarea, siendo 1 para comprobar si el brazo responde, y 2 para moverlo. En cambio, *N* indica el punto al que se desea mover el brazo, siendo todos ellos menos el 0, puntos predefinidos y almacenados en memoria, como los tres puntos de HOME. Cuando *N* es igual a 0 significa que el punto obtenido por la cámara se indica a través de los parámetros *X*, *Y*, *Z* y *Pitch*.

```
void Servicio::enviaGoal(int task , int N)
{
    goal.task = task;
    goal.N = N;
    client.sendGoal(goal,boost::bind(&Servicio::doneCb, this, _1,
    _2),Client::SimpleActiveCallback(),boost::bind(&Servicio::feedbackCb, this,
    _1));
    //wait for the action to return

    finished_before_timeout = client.waitForResult(ros::Duration(30.0));
    ready = false; // Se pone a "false" para que el feedback lo active

    if (finished_before_timeout)
    {
        ROS_INFO("Current State: %s.",
        client.getState().toString().c_str());
    }
    else{
        ROS_INFO("Action did not finish before the time out.");
        client.cancelAllGoals();
        ROS_INFO("Current State: %s.",
        client.getState().toString().c_str());

        // Comprobación de estado si esta inicializado o no
        if(task==1)
            estado = 2;
        else
            estado = 4;

        // Comprobación del feedback
        if(realm == 1)
            estado = 3;

        if(realm == 3)
            estado = 5;

        if(realm == 7)
            estado = 9;

        if(realm == 3)
            estado = 5;
    }
}
```

Por último, se comprueba que todo ha funcionado correctamente, y en caso contrario, teniendo en cuenta el valor de la variable *realm*, se asigna el valor correspondiente a la variable *estado*.

## ObjectsDetected

Esta función recoge el valor del punto del mensaje recibido por el nodo *object\_regonition\_kitchen* y lo transforma al marco de referencia de la base del brazo manipulador.

Debido a que la cámara se emplea tanto para la etapa del capítulo anterior como en esta, el fichero URDF del robot únicamente contiene el desplazamiento desde la base del brazo a la cámara Kinect, pero no así la inclinación necesaria para que la cámara pueda detectar objetos que puedan ser manipulados por el brazo. Es por esta razón que, después de la transformación del punto haciendo uso de la herramienta *tf* de ROS, se emplea otra librería llamada *Eigen* para rotar el punto de manera manual dependiendo de la inclinación de la cámara.

```

void Servicio::objectsDetected(const
pcl_recognition::object_position::ConstPtr& objects_msg)
{
    camera_point.header.frame_id = "camera_rgb_optical_frame";
    camera_point.point.x = objects_msg->pos_x;
    camera_point.point.y = objects_msg->pos_y;
    camera_point.point.z = objects_msg->pos_z;

    // Se transforma el punto del frame de la camara a la base del brazo
    try{
        listener.transformPoint("arm_base_link",ros::Time(0),camera_point,"camera_rg
        gb_optical_frame",arm_base_point);

    }
    catch(tf::TransformException& ex){
        ROS_ERROR("Received an exception trying to transform a point from
        \\\"camera_point\\\" to \\\"arm_base_point\\\": %s", ex.what());
        estado = 11;
    }

    // Rotación del punto de manera manual
    Eigen::Vector3d Point_aux1
    (arm_base_point.point.x,arm_base_point.point.y,arm_base_point.point.z);
    //float theta = 0.4887; // 28 grados en radianes para la Kinect1
    float theta = 0.5602; // 32.1 grados en radianes para la Kinect2

    Eigen::Affine3d ry =
        Eigen::Affine3d(Eigen::AngleAxisd(theta, Eigen::Vector3d(0,
1, 0)));
    Eigen::Vector3d Point_aux2 = ry*Point_aux1;

    goal.X = Point_aux2(0);
    goal.Y = Point_aux2(1);
    goal.Z = Point_aux2(2);
    goal.Pitch = 0;

    // Se crea tfBroadcaster para visualizar el punto transformado en Rviz
    transform.setOrigin( tf::Vector3(goal.X, goal.Y, goal.Z) );
    transform.setRotation( tf::Quaternion(0, 0, 0, 1) );
    bpoint.sendTransform(tf::StampedTransform(transform, ros::Time::now(),
"arm_base_link", "objetivo"));
}

```

Finalmente se guarda los valores del punto en el mensaje *goal* que posteriormente se enviará al nodo *move*. Además, se utiliza la herramienta *tfBroadcaster* para poder visualizar el punto en *Rviz*.

### Is\_inside\_workspace

El propósito de este método es comprobar que el punto transformado por la función anterior está dentro del espacio de trabajo del brazo. Para ello, se construye el vector que va desde el origen de coordenadas de la base del brazo hasta el punto que se recibe por parámetro.

Una vez construido, se calcula el módulo del vector y se comprueba si excede o no los límites máximos o mínimos requeridos. En caso de que el punto se encuentre dentro de los límites se devuelve *true*. En el caso contrario se devolvería un *false*.

```
bool Servicio::is_inside_workspace(double X, double Y, double Z)
{
    //Vector u = (u1,u2,u3) donde u1 es x1 - x0.
    //Como es respecto al origen, x0=y0=z0=0, por tanto u=(x1,y1,z1)

    float u1=X;
    float u2=Y;
    float u3=Z;

    float moduloVector = sqrt( pow(u1,2) + pow(u2,2) + pow(u3,2) );
    float moduloMax = 0.32;
    float moduloMin = 0.10;

    cout << "El modulo del punto es = " << moduloVector << endl;
    cout << "Debe ser menor que 0.32 y mayor que 0.10 ..." << endl;

    if (moduloVector > moduloMax){
        cout << "El punto se encuentra fuera del espacio de trabajo." <<
endl;
        return false;
    }
    else{

        if ( moduloVector < moduloMin){
            cout << "El punto se encuentra fuera del espacio de trabajo."
<< endl;
            return false;

        }else{
            cout << "El punto se encuentra dentro del espacio de trabajo"
<< endl;
            return true;
        }
    }
}
```

El valor máximo del módulo se debe a razones puramente estructurales del brazo ya que la longitud del mismo limita el alcance máximo. Sin embargo, el módulo mínimo se debe a que el brazo no posee suficientes grados de libertad como para alcanzar fácilmente puntos próximos a su propia base.

### 5.3.3 Nodo move

*Move* es el nombre que recibe el nodo encargado de hacer la tarea de “*pick and place*”, es decir, de coger y guardar un objeto con la ayuda del brazo manipulador que incorpora el Turtlebot2. Este nodo emplea la librería *pxpincher\_lib*, diseñada por un grupo de investigación de la universidad técnica de *Dortmund* [30], para poder mover el robot manipulador. En concreto, hace uso de la clase *PhantomXControl*, a la cual se le añade una serie de nuevas funcionalidades que se detallarán a continuación.

Además, este nodo incorpora un servidor actionlib que es utilizado por el nodo *find\_and\_pick* para realizar la comunicación entre ambos nodos.

#### Definiciones y dependencias

Como en el resto de archivos, al inicio se definen las diferentes dependencias y definiciones de las cuales se hará uso en el resto del código. Entre estas dependencias cabe destacar “*phantomx\_interface*”, que contiene la clase *PhantomXControl*, o *MoveArmAction*, el cual define el mensaje del actionlib definido entre *find\_and\_pick* y *move*.

Respecto a las definiciones, al igual que en el caso del cliente actionlib, en este fichero también se hace una simplificación de la expresión del servidor usando la instrucción *typedef* y renombrándolo como *Server*. También se define dos variables globales, y una variable local al *namespace pxpincher* donde están definida la clase *PhatomXControl*.

```
#include <ros/ros.h>
#include "tf/transform_datatypes.h"
#include <tf_conversions/tf_eigen.h>

#include <pxpincher_lib/phantomx_interface.h>

#include <termios.h>
#include <vector>
#include <stdexcept>

#include <pxpincher_lib/MoveArmAction.h>

#include <actionlib/server/simple_action_server.h>
#include "boost/bind.hpp"
#include <signal.h>
#include <math.h>

using namespace std;
using namespace pxpincher;
using namespace ros;

typedef actionlib::SimpleActionServer<pxpincher_lib::MoveArmAction> Server;

// Variables globales
sensor_msgs::PointCloud workspace;
pxpincher_lib::MoveArmFeedback feedback;

namespace pxpincher
{
// Variable comun al namespace pxpincher
bool objeto_atrapado = false;
}
```

## Main

Como es habitual en los paquetes de ros, la función main es la encargada de definir el nodo entre otras cosas.

```
int main( int argc, char** argv )
{
    ros::init(argc, argv, "move");
    ros::NodeHandle s;
    ros::NodeHandle n("~");

    pxpincher::PhantomXControl arm; // Creación del objeto arm
    ros::Publisher pub_workspace =
n.advertise<sensor_msgs::PointCloud>("workspace", 100);

    // Se inicializa el servidor
    ROS_INFO("Inicializando el servidor ...");

    Server server(s,"MoveArm", boost::bind(&execute, _1, &server,
&arm),false); // _1 significa que el primer argumento está vacío
server.start();
    ROS_INFO("... servidor inicializado");
    ROS_INFO("Inicializando el brazo ...");

    arm.initialize(); // Se inicializa el robot
    arm.move_home();
    arm.move_home3(); //Posición para no molestar a la camara cuando esté
en la etapa de localización de productos

    ROS_INFO("Sampling joint configurations to generate workspace
pointcloud ...");
    sensor_msgs::PointCloud workspace;
    arm.visualizeWorkSpace(workspace, 0.5);

    ros::Rate r(10);

    int apertura_estado_anterior = arm.getGripperJointPercentage();
    JointVector q = arm.getJointAngles();

    while (ros::ok()){

        // Visualización del workspace
        workspace.header.stamp = ros::Time::now();
        pub_workspace.publish(workspace);
        arm.publishInformationMarker();

        // Comprobación del valor de las articulaciones
        if (arm.getGripperJointPercentage() > 101 ){
            ROS_INFO("ERROR en el valor de GripperJoint");
            feedback.status = 6; // Problemas con los servos
        }
        else{
            // Se guarda los valores de los estados de las articulaciones
            apertura_estado_anterior = arm.getGripperJointPercentage();
            q = arm.getJointAngles();
        }
        ros::spinOnce();
        r.sleep();
    }
    return 0;
}
```

En esta ocasión, también se define el objeto *arm* de la clase *PhantomXControl* y el servidor *actionlib* pasándole por parámetros el objeto creado. Además, se le indica la función a ejecutar cada vez que se reciba una petición al servidor, “*execute*”.

Tras ello se inicializa el brazo, se mueve a las posiciones de HOME y HOME3 respectivamente y se define la variable *workspace* para poder ser empleada en *Rviz*. Por último, se crea el bucle infinito para comprobar el estado de la red de ROS haciendo uso de la instrucción *ros::spinOnce* y *ros::Rate*. Asimismo, se le ha añadido una comprobación en cada ciclo del estado de las articulaciones del brazo para informar al nodo principal de un posible error en el driver.

### Execute

Esta función se ejecuta siempre que se reciba un “*goal*” desde el cliente *actionlib*, y dependiendo del valor de *task* y *N* se seleccionará una acción diferente.

```
void execute(const pxpincher_lib::MoveArmGoalConstPtr& goal, Server* as,
pxpincher::PhantomXControl* robot){

    feedback.status = 0; // Estado inicial / Inicializacion del feedback
    as->publishFeedback(feedback);

    if(goal->task == 1){
        ROS_INFO("Brazo ya inicializado");
        feedback.status = 1; // El brazo esta inicializado
        as->publishFeedback(feedback);
        robot->open_gripper(); // Se abre el gripper para ver que responde
    }

    if(goal->task == 2){
        feedback.status = 2; // Moviendo brazo
        as->publishFeedback(feedback);
        ROS_INFO("Feedback publicado, empezamos a mover el brazo");

        switch(goal->N){
            case 0: //Punto XYZ dado
                try{
                    robot->custom_pick_and_place(goal->X, goal->Y, goal->Z,
goal->Pitch, 1, true);
                }catch( const std::invalid_argument& e )
                {
                    ROS_ERROR("%s, se cancela la operacion",e.what());
                    feedback.status = 3; // Error cinematica inversa
                    as->publishFeedback(feedback);
                    as->setAborted();
                    return;
                }
                break;
            case 1: //Punto 1 almacenado
                try{
                    robot->custom_pick_and_place(0.24, 0, 0, 0, 1, true);
                }catch( const std::invalid_argument& e )
                {
                    ROS_ERROR("%s, se cancela la operacion",e.what());
                    feedback.status = 3; // Error cinematica inversa
                    as->publishFeedback(feedback);
                    as->setAborted();
                    return;
                }
                break;
        }
    }
}
```

Por ejemplo, para *task* igual a 0 se abrirá la garra para comprobar que el brazo funciona, y para *task* igual a 1 se moverá el brazo a distintas posiciones predefinidas, existiendo 5 posiciones diferentes.

```
case 2:           //Punto 2 almacenado
    try{
        robot->custom_pick_and_place(0.24, 0, 0.172, 0, 1, true);
    }catch( const std::invalid_argument& e )
    {
        ROS_ERROR("%s, se cancela la operacion",e.what());
        feedback.status = 3; // Error cinematica inversa
        as->publishFeedback(feedback);
        as->setAborted();
        return;
    }
    break;
case 3:           //Mover a HOME
    try{
        robot->move_home();
        robot->open_gripper();
    }catch( const std::invalid_argument& e )
    {
        ROS_ERROR("%s, se cancela la operacion",e.what());
        feedback.status = 3; // Error cinematica inversa
        as->publishFeedback(feedback);
        as->setAborted();
        return;
    }
    break;
case 4:           //Mover a HOME2
    try{
        robot->move_home2();
    }catch( const std::invalid_argument& e )
    {
        ROS_ERROR("%s, se cancela la operacion",e.what());
        feedback.status = 3; // Error cinematica inversa
        as->publishFeedback(feedback);
        as->setAborted();
        return;
    }
    break;
case 5:           //Mover a HOME3
    try{
        robot->move_home3();
    }catch( const std::invalid_argument& e )
    {
        ROS_ERROR("%s, se cancela la operacion",e.what());
        feedback.status = 3; // Error cinematica inversa
        as->publishFeedback(feedback);
        as->setAborted();
        return;
    }
    break;
default:
    ROS_INFO("No existe el punto %d",goal->N);
    feedback.status = 7; // El punto indicado no existe
    as->publishFeedback(feedback);
    as->setAborted();
    return;
break;
}
```

Para el caso de *task* igual a 1 y *N* igual a 0, se obtiene los valores *X*, *Y*, *Z* y *Pitch* del mensaje “*goal*” y se emplea la función “*custom\_pick\_and\_place*” para intentar obtener el objeto dado en esa posición. El resto de valores de *N* son simplemente posiciones predefinidas. Por último, en caso de no recibir un valor de *N* correcto se responde indicando que el valor no es correcto.

En todos los casos se comprueba que se ha realizado la acción, enviando en caso contrario un mensaje de *feedback* y abortando la operación.

### Custom\_pick\_and\_place

Esta función contiene una serie de instrucciones para realizar la tarea de “*pick and place*”, es decir, coger y guardar un objeto. Para llevarlo acabo, recibe por parámetros el valor de las coordenadas *XYZ* en metros, el valor de la orientación del “*Pitch*” en radianes, la velocidad a la que se realiza la operación, y, por último, una variable booleana que indica si se desea alcanzar un punto de aproximación antes de alcanzar el punto final.

```
void PhantomXControl::custom_pick_and_place(double X, double Y, double Z,
double pitch, double speed, bool point_aux){

    cout << "Coordenadas del punto recibido respecto a arm_base_link (en
metros): X=" << X << ", Y=" << Y << ", Z=" << Z << ". " << endl;

    // Antes de acercarse a un punto se va a la posicion de seguridad HOME
    move_home();

    // Se calcula el punto de aproximacion
    geometry_msgs::Point p_aprox;

    if(point_aux == true) {
        p_aprox = gen_point_aprox(X,Y,Z);
        setEndeffectorPose({p_aprox.x , p_aprox.y , p_aprox.z }, pitch ,
speed, false, true);
    }
    // Abriendo la garra por seguridad en caso de que no estuviera abierta
    open_gripper();

    // Punto donde esta el objeto // Se considera gripper abierto
    setEndeffectorPose({X, Y , Z}, pitch , 0.7*speed, false, true);

    // Cerrando la garra
    close_gripper(100,33); // abertura inicial 100%; decremento = 33%

    // Punto de retirada
    if(point_aux == true)
        setEndeffectorPose({p_aprox.x , p_aprox.y , p_aprox.z }, pitch ,
0.5*speed, false, true);

    // Por seguridad, una vez atrapado (o no) un objeto se va a Home
    move_home();
}
```

Antes de realizar cualquier acción se mueve el brazo a HOME por seguridad, debido a que desde esa posición el brazo tiene mayor facilidad para calcular trayectorias ya que está en una posición central. Seguidamente, si está activada la opción del punto de aproximación se calcula y se mueve el brazo a dicho punto.

Posteriormente, se abre la garra, se procede a mover el brazo hasta la posición donde está el objeto y se cierra la garra. A continuación, se mueve el brazo a una posición de retirada que será HOME, o el punto de aproximación más HOME en caso de que esté activada dicha opción.

La función *close\_gripper* incorpora un algoritmo para comprobar si se ha obtenido un objeto o no, poniendo la variable *objeto\_atrapado* a *true* en caso afirmativo. En caso de ser *true* se almacena el objeto usando la función *store\_object()*. Por último, se informará al nodo principal de si se ha atrapado o no el objeto, utilizando el *feedback* del *actionlib*.

```
if(objeto_atrapado == true) {
    ROS_INFO("Guardando el objeto obtenido en la bandeja del robot");
    feedback.status = 4; // Objeto atrapado
    store_object(speed);
    move_home();
}
else{
    feedback.status = 5; // Objeto no atrapado
}
```

### Get\_point\_aprox

Esta función permite obtener un punto de aproximación dado un punto *XYZ* cualquiera. Para ello se parte del módulo del vector que forma el centro del marco de referencia de la base del brazo y el punto dado.

```
geometry_msgs::Point PhantomXControl::gen_point_aprox(double X, double Y,
double Z)
{
    //Punto Aproximacion
    geometry_msgs::Point point;

    // Vector u = (u1,u2,u3) donde por ejemplo u1 = x1 - x0.
    // Como es respecto al origen, x0=y0=z0=0, por tanto u=(x1,y1,z1)
    float u1=X;
    float u2=Y;
    float u3=Z;

    float moduloVector = sqrt( pow(u1,2) + pow(u2,2) + pow(u3,2) );
    float moduloPtoAprox = 0;
    float distanciaPto = 0.05;
```

Dada una distancia al punto de aproximación de 5 centímetros, se calcula el módulo del punto de aproximación en las tres componentes del espacio. Para ello se ha tenido en cuenta tanto el signo de las componentes del vector como un valor nulo de dicha componente.

Por último, se almacena cada componentes en una variable del tipo *geometry\_msgs::Point* y se devuelve su valor al finalizar la llamada.

```

    // Calculo componente x
    if(u1>0)
        point.x = moduloPtoAprox/sqrt(1 + (pow(u2,2)/pow(u1,2)) +
(pow(u3,2)/pow(u1,2)));
    else if (u1 == 0)
        point.x = 0;
    else
        point.x = -1*moduloPtoAprox/sqrt(1 + (pow(u2,2)/pow(u1,2)) +
(pow(u3,2)/pow(u1,2)));

    // Calculo componente y
    if(u2>0)
        point.y = moduloPtoAprox/sqrt(1 + (pow(u1,2)/pow(u2,2)) +
(pow(u3,2)/pow(u2,2)));
    else if (u2 == 0)
        point.y = 0;
    else
        point.y = -1*moduloPtoAprox/sqrt(1 + (pow(u1,2)/pow(u2,2)) +
(pow(u3,2)/pow(u2,2)));

    // Calculo componente z
    if(u3>0)
        point.z = moduloPtoAprox/sqrt(1 + (pow(u1,2)/pow(u3,2)) +
(pow(u2,2)/pow(u3,2)));
    else if (u3 == 0)
        point.z = 0;
    else
        point.z = -1*moduloPtoAprox/sqrt(1 + (pow(u1,2)/pow(u3,2)) +
(pow(u2,2)/pow(u3,2)));
    }
    else{
        cout << "El modulo es menor que la distancia al punto de
aproximacion que se desea calcular" << endl;
        point.x = 0;
        point.y = 0;
        point.z = 0;
    }
    return point;
}

```

## Open\_gripper

Esta función permite abrir la garra. Su funcionamiento se reduce a comprobar el valor de la apertura de la garra, si este está por debajo del 97% se deduce que no está abierta completamente y se procede a abrirlo mediante la función *setGripperJoint* de la clase *PhantomXControl*.

```

void PhantomXControl::open_gripper(){
    if (getGripperJointPercentage() < 97 ){
        cout << "Abriendo el gripper al 100%" << endl;
        setGripperJoint(99,true); // Garra abierta al 99% por seguridad
    }
    else{
        ROS_INFO("El gripper ya esta abierto completamente");
    }
}

```

### Close\_gripper

Se trata de una función recursiva que va cerrando la garra poco a poco hasta un valor dado por el parámetro *porcentaje*. A su vez, el decremento viene dado por la variable que lleva el mismo nombre y que también es recibida como un parámetro de la función.

```
void PhantomXControl::close_gripper(int porcentaje, int decremento)
{
    int abertura = porcentaje - decremento;

    cout << "Cerrando el gripper a " << abertura << "%." << endl;

    if(abertura < 0){
        cout << "El valor de la abertura es negativo, cerrando al 0%" <<
    endl;
        setGripperJoint(1,true); // Si la abertura es menor a 0 se satura a
1 por seguridad
        ROS_INFO_STREAM("No se ha cogido ningun objeto");
        objeto_atrapado = false;
    }
    else{
        setGripperJoint(abertura,true); // Garra abierta al percentage
recibido menos incremento%

        if (getGripperJointPercentage() > abertura+2 ){ // El 2 es un
minimo margen por fallos del encoder del gripper
            ROS_INFO_STREAM("Se ha cogido un objeto");
            objeto_atrapado = true;
        }
        else{
            close_gripper(abertura,decremento);
        }
    }
}
```

En primer lugar, se calcula la abertura a la que la garra debe cerrarse. Si dicha abertura es mayor que cero, se cierra la garra hasta alcanzar dicho valor tantas veces como sean necesarias hasta alcanzar un valor menor o igual que 0.

La razón por la que se decidió implementar esta función de manera recursiva fue para poder comprobar si se ha obtenido un objeto o no. De esta manera, cada vez que se intenta cerrar la garra un determinado valor, se comprueba si realmente se consiguió cerrarla hasta dicho valor. En caso negativo, significa que un objeto ha impedido cerrar la garra y por tanto se ha conseguido atraparlo. Para indicar si se ha obtenido o no un objeto se utiliza la variable *objeto\_atrapado*.

Nótese, además, que se utilizan ciertos márgenes a la hora de realizar comprobaciones. Esto se debe a que la lectura del encoder de la garra no es perfecta y frecuentemente se obtienen lecturas falsas que pueden llevar a errores como pensar que se ha obtenido un objeto cuando no ha sido así.

Hay que tener en cuenta que para determinados objetos pequeños se pueden obtener lecturas próximas a 0, haciendo fallar el algoritmo, y por tanto, haciendo creer al nodo principal que no se ha obtenido un objeto cuando si lo ha hecho.

### Move\_home

Las funciones `move_home()`, `move_home2()` y `move_home3()` permiten al brazo moverse a diferentes configuraciones de las articulaciones predefinidas haciendo uso de la función `setEndeffectorPose` de la clase `PhantomXControl`. Inicialmente se creó HOME como una posición central donde el brazo descansara y pudiera acceder a todas las posiciones de su espacio de trabajo, pero posteriormente se crearon HOME 2 y 3 a partir de estas con diferentes propósitos.

La posición HOME2 se definió para evitar molestar a la cámara cuando esté realizando la detección de objetos para ser manipulados como se muestra en la Figura 11. Además, dado que para la etapa anterior de localización de productos se puede emplear tanto la *Kinect* versión 1 como 2, se definió una tercera posición de HOME con el fin de evitar que el brazo apareciera en el campo de visión de la *Kinect* versión 2 durante dicha etapa. Esta posición es idéntica a la original bajando el brazo unos centímetros (Figura 29).

```
void PhantomXControl::move_home(){
    ROS_INFO("Moviendo a HOME");
    setEndeffectorPose({0.086, 0, 0.13}, M_PI/4, 1, false, true);
    if(objeto_atrapado == false)
        open_gripper();
}

void PhantomXControl::move_home2(){
    ROS_INFO("Moviendo a HOME2");
    setEndeffectorPose({-0.08, 0, 0.18}, -M_PI/4, 1, false, true);
    if(objeto_atrapado == false)
        open_gripper();
}

void PhantomXControl::move_home3(){
    ROS_INFO("Moviendo a HOME3");
    setJoints({ 0, -1.1, 2.26, 1.1}, 0.8, false, true);
    if(objeto_atrapado == false)
        open_gripper();
}
```

### Store\_object

Por último, la función `store_object()` define una serie de posiciones en coordenadas articulares para mover el brazo hasta la bandeja lateral derecha del robot y soltar la pieza suavemente.

```
void PhantomXControl::store_object(double speed){
    // Posicion lateral derecha en coordenadas articulares
    setJoints({ -2, -0.77, 1.6, 1.5}, speed, false, true); // Paux
    setJoints({ -2, 0.46, 1.8, 0.87}, speed, false, true); //P lateral
    open_gripper();
    setJoints({ -2, -0.77, 1.6, 1.5}, 0.5*speed, false, true); // Paux
}
```

## 5.4 Resultados

Los resultados obtenidos tras poner en práctica los diferentes paquetes utilizados y desarrollados, vienen limitados en gran medida por el hardware empleado. Un ejemplo de esta limitación es la longitud máxima que el brazo puede alcanzar. Además, existe una limitación respecto al tamaño mínimo y máximo de los objetos que viene dada por la resolución de la cámara y la apertura máxima de la garra respectivamente.

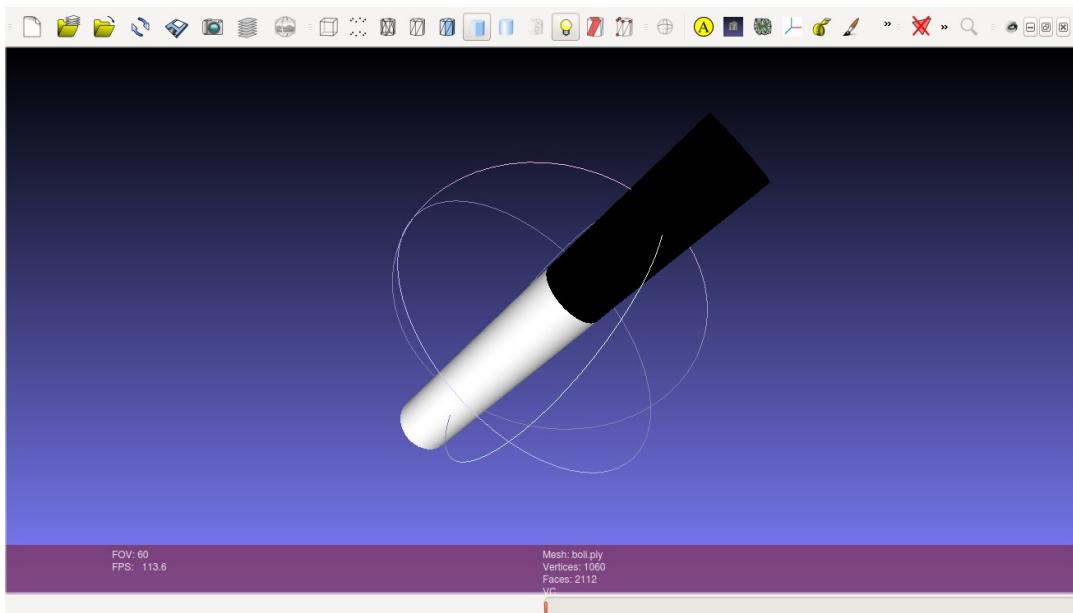


Figura 32 – Modelo 3D de un bolígrafo generado para la base de datos del paquete ORK.

Por ello, sabiendo que el objeto que debe atacar la garra debe ser lo suficiente pequeño para que pueda agarrarlo, se empleó un bolígrafo o rotulador. Por tanto, se modeló el bolígrafo en 3D empleando programas de modelado como *Blender*, como se muestra en la Figura 32.

Tras incluirlo en la base de datos [35], se probó el paquete de localización de objetos ORK obteniendo resultados favorables, aunque no exentos de ciertos problemas. Entre estos problemas se encuentra la posibilidad de que el software confunda en repetidas ocasiones partes del entorno con un bolígrafo, debido a la baja resolución del sensor y al pequeño tamaño del objeto. Aun así, en la mayoría de las ocasiones se consigue detectar con bastante fiabilidad.

En la Figura 33 se observa el escenario en el que se probó la detección y manipulación de objetos con éxito. En ésta se puede apreciar como el paquete ORK consigue detectar tanto el bolígrafo como la superficie donde éste está apoyado. Además, al ser una captura de *Rviz*, también se muestra el modelo del robot en la configuración en la que se encuentra en el momento de la detección, es decir, con el brazo en posición HOME2 para facilitar la detección de objetos.

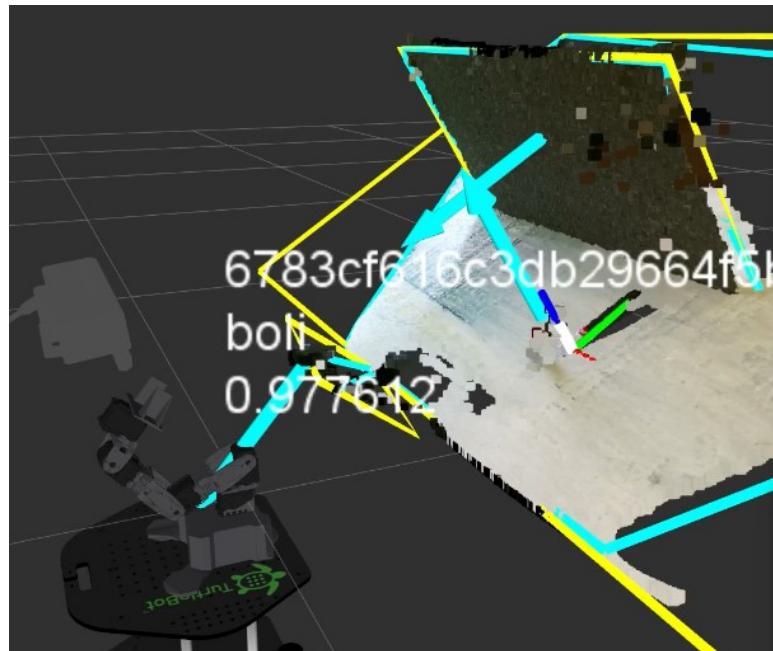


Figura 33 – Escenario de pruebas de la detección y manipulación de objetos visto en Rviz.

Nótese que el escenario donde se encuentra el bolígrafo está en una posición inclinada respecto al robot mientras que la cámara *Kinect* versión 2 se encuentra en posición horizontal. En realidad, el escenario se encuentra en horizontal respecto al suelo y es la cámara *Kinect* la que se encuentra inclinada como se observa en la Figura 34. Esto se debe a que, para permitir la compatibilidad entre esta característica y la anterior descrita en el capítulo 4, en el modelo del robot se configuró la cámara *Kinect* en horizontal respecto al suelo, realizando el cálculo de la inclinación por tanto manualmente tal y como se describe en el nodo *find\_and\_pick*.

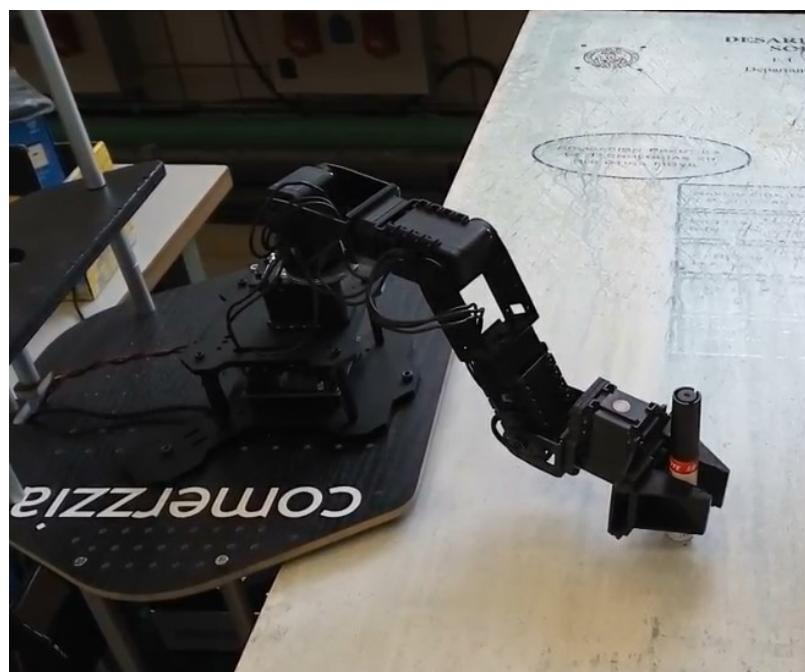


Figura 34 – Imagen real del brazo incorporado en el Turtlebot2 atrapando un bolígrafo.

Por último, en la Figura 34 se puede observar una foto del brazo robótico alcanzando y obteniendo el bolígrafo que se empleó para realizar las pruebas. Pese a obtener resultados satisfactorios en varias ocasiones, lo cierto es que la tasa de acierto es muy baja siendo muy difícil completar con éxito la tarea. Uno de los motivos es la imposibilidad de detectar el objeto, ya sea por condiciones de luz o de resolución de la cámara. Además, puede ocurrir que un objeto sea detectado en una posición incorrecta.

Aun así, en caso de que la parte de detección se haya realizado con éxito, existen otra serie de inconvenientes producidos por el propio brazo. Entre estos inconvenientes se encuentra la posibilidad de que el brazo deje de responder, que no encuentre una trayectoria válida para alcanzar el punto deseado, o simplemente que la garra no logre atrapar el objeto correctamente.

Es por todo ello que los resultados obtenidos no son satisfactorios, aunque sí que son esperanzadores de cara al futuro, ya que con un hardware mucho más fiable se lograría aumentar considerablemente la tasa de acierto, en especial con un brazo más robusto y sofisticado que tuviera mayores grados de libertad y alcance.



# 6 CONCLUSIONES

---

*Algunos hombres ven las cosas como son y dicen “¿por qué?”. Yo, en cambio, veo cosas que todavía no son y digo “¿por qué no?”*

*- Robert Francis Bobby Kennedy -*

A lo largo de este documento se ha presentado y desarrollado una parte de la solución obtenida para la división de robótica del proyecto de I+D+i de *Comerzzia* denominado “*Smart Omnichannel Retail*”, en concreto se ha abarcado la localización, detección y manipulación de productos con la plataforma robótica *Turtlebot2* utilizando el *framework* de código abierto ROS.

La elección de utilizar **ROS** ha sido clave, no solo para alcanzar los objetivos que se propusieron inicialmente en la definición del proyecto citado, sino que además ha permitido cumplir los objetivos iniciales e incluso ampliarlos, pudiendo desarrollar nuevas mejoras y características que fueron añadidas a posteriori. Esto se debe, principalmente, a que ROS cuenta con una enorme y diversa comunidad internacional que está en constante crecimiento, que abastece con nuevos contenidos asiduamente al resto de la comunidad que pueden emplearlos, modificarlos o mejorarlos para beneficio propio o por el bien de la comunidad. Esto permitió ahorrar mucho tiempo y esfuerzo en investigar y desarrollar software, ya que habían sido desarrollado y probado por otros grupos, y dedicar dicho tiempo a desarrollar aquellas características más concretas y propias del proyecto.

Por otra parte, el ***Turtlebot2*** resultó ser una plataforma ideal para poder desarrollar y poner en práctica todas las distintas ideas que se propusieron, con el fin de contemplar la validez y viabilidad de las mismas. Gracias a tratarse de una plataforma con total compatibilidad con ROS y a contar con gran apoyo de la comunidad, la incorporación del *Turtlebot2* en el proyecto simplificó y aceleró aún más el desarrollo del mismo. También, al tratarse de una plataforma abierta y totalmente configurable facilitó la incorporación de mejoras y periféricos, como la inclusión de la segunda versión de la *Kinect*, un *LIDAR* o el procesador de gran capacidad *Intel NUC*.

Respecto a la **localización de productos**, tras desechar la idea inicial de utilizar etiquetas RFID en beneficio del sensor 3D *Kinect*, se comprobó que el empleo de códigos que fueran fácilmente detectables mediante el procesamiento de imágenes constituía una solución razonable y viable respecto a la tarea que debía desempeñar el robot. Para ello se probó con diferentes tipos de códigos, desde códigos unidimensionales como códigos de barras hasta detección de caracteres mediante técnicas de reconocimiento óptico de caracteres, también conocidas como OCR (*Optical Character Recognition*). No obstante, finalmente se empleó códigos 2D como los populares códigos QR, empleando para ello la librería de código abierto *Zbar*, que ya había sido previamente incluida en ROS.

Por tanto, gracias a este paquete y a la utilización de la librería PCL para el manejo de nube de puntos, se desarrolló con éxito un paquete software que permite localizar en el espacio códigos QR que estén en el campo de visión del robot. Los resultados obtenidos fueron muy satisfactorios, logrando una fiabilidad y precisión extremadamente alta para códigos QR iguales o superiores a 8 centímetros de lado y una distancia de un metro.

En cuanto a la **detección y manipulación de objetos o productos**, tras probar los distintos paquetes disponibles en ROS para el manipulador *PhantomX Pincher* y comprobar su utilidad, se comenzó con la búsqueda de un software en ROS que permitiera detectar objetos que pudieran ser manipulados por el brazo. ORK fue la mejor opción y la solución más factible debido a los buenos resultados que ofrecía en cuanto al reconocimiento de pequeños objetos, y que además ofrecía la posibilidad de obtener su ubicación en el espacio. Una vez integrado este paquete, el desarrollo de un programa que gobernara tanto el driver del brazo como el paquete ORK no supuso una complicación añadida. Sin embargo, los resultados obtenidos distaron enormemente respecto al paquete anterior, siendo la tasa de error bastante alta, y su viabilidad en un entorno real está lejos aún del propósito real a largo plazo de este paquete. Ello se debe en gran parte a las limitaciones de hardware con las que se contaba, siendo el principal limitante y fuente de error el brazo empleado de 4 grados de libertad reales que obligó a simplificar la complejidad del software desarrollado.

Sin embargo, cabe recordar que su verdadero propósito consistía en estudiar y comprobar la posibilidad de incluir la opción de utilizar un brazo robótico para recoger objetos en un entorno de *retail*, o también para obtener el pedido de posibles clientes en un entorno de *darkstore*. En ese caso se concluye que, para llevar a cabo a tarea de detección y manipulación de objetos, las herramientas de las que se disponen son insuficientes y se necesita tanto una mejora en el hardware como en el software. Si bien se ha conseguido grandes avances en los últimos años, como es el caso del robot PR2 de *Willow Garage* en el cual se desarrolló el paquete ORK como se describió anteriormente, aún queda un largo camino por recorrer para alcanzar cotas aceptables en un entorno con clientes en el sector *retail*.

## 6.1 Líneas de investigación y desarrollos futuros

Tras haber abordado diferentes soluciones posibles y teniendo en cuenta los resultados obtenidos, así como el hardware disponible, se plantea a continuación una serie de mejoras y posibles líneas de investigación futuras que permitan ampliar y mejorar el trabajo que se ha expuesto a lo largo del presente documento.

- **Ampliar la resolución de la cámara RGB.**

Durante el desarrollo del paquete de “Localización de productos” se planteó como un objetivo añadido la posibilidad de localizar códigos *QR* y/o códigos de barra lo más pequeños posibles, es por ello que se optó por sustituir la primera versión de la Kinect por su segunda versión, la cual posee resolución HD.

Si bien los resultados fueron satisfactorios, se dejó como una posible mejora futura la posibilidad de incorporar una cámara de mayor resolución, por ejemplo, una resolución de 4K, que junto con la *Kinect* permitiera detectar códigos aún más pequeños. Puesto que la detección de los códigos y su posterior localización en el espacio son tareas separadas, existe la posibilidad de realizar dicha mejora reutilizando el software existente en casi su completa totalidad. La dificultad se encuentra en realizar una buena calibración de la cámara externa y el sensor 3D, aunque existe un tutorial en la wiki de ROS donde se explica el proceso de calibración donde se realiza una calibración de una cámara externa RGB con una Kinect, como se puede apreciar en la Figura 35. [38]



Figura 35 – Ejemplo de integración de una cámara RGB externa y una Kinect.

- **Utilizar un brazo manipulador más complejo de al menos 6 DOF.**

Una de las limitaciones más importantes que se ha encontrado en el desarrollo del paquete de “Detección y manipulación de objetos” ha sido el propio brazo manipulador. Si bien es cierto que logra su cometido, la tasa de error es muy elevada en gran medida debido a su corto alcance, su pequeña garra y no disponer de al menos 6 grados de libertad.

De las tres limitaciones enumeradas, tanto su alcance como su garra pueden ser fácilmente mejoradas aumentando el tamaño de ambas puesto que se trata de un brazo modular y ampliable. Sin embargo,

la falta de grados de libertad impide en ocasiones obtener una solución al problema de la cinemática inversa y, por tanto, impidiendo obtener una trayectoria válida que sitúe el efecto final en el punto deseado. Es por ello que con la sustitución del brazo actual por uno más robusto y versátil se lograría obtener mejores resultados.

Igualmente, el software desarrollado podría ser fácilmente reutilizable y mejorado sustituyendo el driver del brazo por el correspondiente desarrollado por el fabricante, y utilizando como planificador de trayectorias el software *MoveIt!* [29].

- **Mejora de la sensorización del entorno.**

Durante la investigación de paquetes que permitieran detectar objetos y su posterior manipulación, se estudió la posibilidad de incluir un *octomap* con el fin de mejorar la planificación de trayectorias del brazo. Un *octomap* no es más que una representación volumétrica del espacio en 3D basado en *octotrees* [38], empleado por ejemplo para localizar obstáculos de manera tridimensional por robots. Existe una librería del mismo nombre en ROS que permite obtener un *octomap* haciendo uso de sensores 3D como la Kinect [39].

Aunque se realizaron pruebas de manera satisfactoria con la Kinect, pronto se descartó su utilización debido a la importante cantidad de procesamiento que era necesario para generar un *octomap*, y al aumento de complejidad en la generación de trayectorias del brazo dado las limitaciones del hardware disponible. Sin embargo, gracias al *Intel Nuc* y a un posible nuevo manipulador más versátil, la integración y utilización de este paquete sería factible y bastante interesante, no solo en cuanto a la manipulación de objetos sino también para la creación de mapas en 3D o la navegación del propio robot por el mapa para evitar obstáculos a distintos niveles que el láser no pueda detectar. Además, el paquete *MoveIt!* integra el *octomap* de manera nativa como se puede observar en la Figura 36.

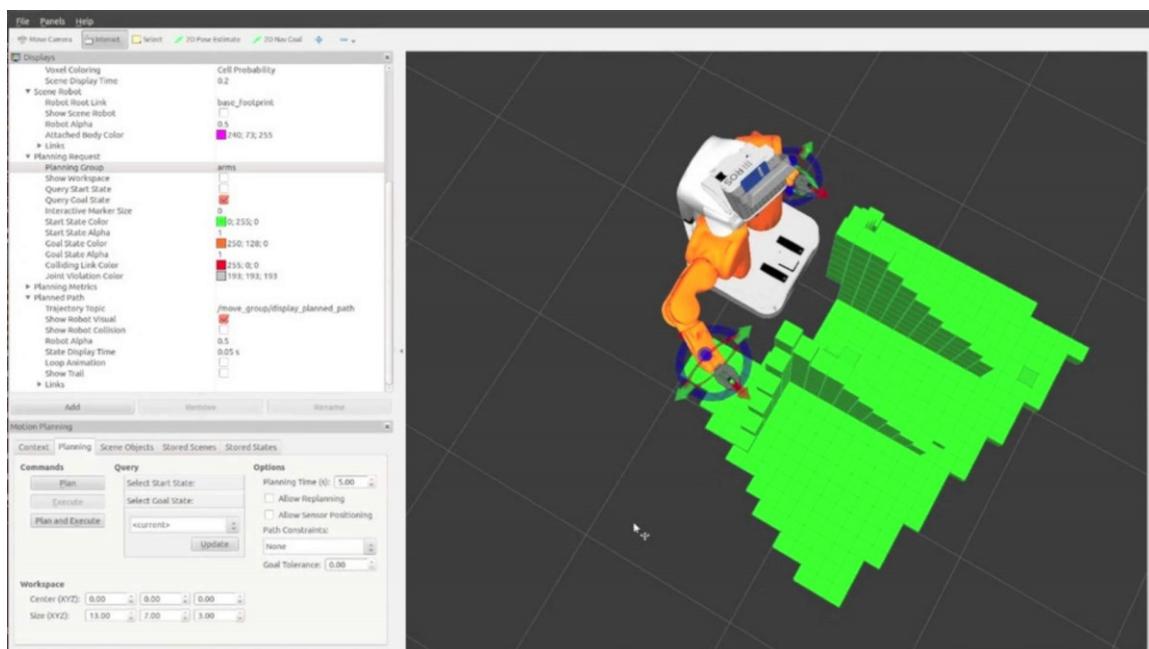


Figura 36 – Visualización en Rviz de un octomap.

# REFERENCIAS

---

- [1] CTA, «Proyecta - Boletín de noticias Nº 46,» Mayo 2016. [En línea]. Available: <http://www.corporaciontecnologica.com/export/sites/cta/.galleries/galeria-de-boletines/Boletin46.pdf>.
- [2] Aldebaran Robotics, «Nao,» [En línea]. Available: <https://www.ald.softbankrobotics.com/en/cool-robots/nao>.
- [3] Open Source Robotics Foundation, «ROS,» [En línea]. Available: <http://www.ros.org/>.
- [4] Erle Robotics, «Introducción de ROS,» [En línea]. Available: <http://erlerobotics.com/blog/ros-introduction-es/>.
- [5] B. G. W. D. S. Morgan Quigley, Programming Robots with ROS, O'Reilly Media, 2015.
- [6] «Open Source Robotics Foundation,» [En línea]. Available: <https://www.osrfoundation.org/>.
- [7] A. M. Romero, E. Fernández y L. S. Crespo, Learning ROS for Robotics Programming (Second Edition), 2015.
- [8] «Gazebo,» [En línea]. Available: <http://gazebosim.org/>.
- [9] D. G. J. F. Dave Hershberger, «RVIZ,» [En línea]. Available: <http://ros.org/wiki/rviz>.
- [10] E. M.-E. W. M. Tully Foote, «TF,» [En línea]. Available: <http://wiki.ros.org/tf>.
- [11] J. K. Ioan Sucan, «URDF,» [En línea]. Available: <http://wiki.ros.org/urdf>.
- [12] Robotnik, «Robot móvil Turtlebot 2,» [En línea]. Available: <http://www.robotnik.es/robots-moviles/turtlebot-2/>.
- [13] Yujinrobot, «Kobuki user guide,» [En línea]. Available: <http://kobuki.yujinrobot.com/wiki/online-user-guide/>.
- [14] Intel, «INTEL® NUC KIT NUC5i5RYH,» [En línea]. Available: <https://www-ssl.intel.com/content/www/us/en/products/boards-kits/nuc/kits/nuc5i5ryh.html>.
- [15] Hokuyo, «Hokuyo URG-04LX-UG01 Laser,» [En línea]. Available: <https://www.hokuyo-aut.jp/search/single.php?serial=166>.
- [16] Trossen Robotics, «PhantomX Pincher Robot Arm,» [En línea]. Available: <http://www.trossenrobotics.com/p/PhantomX-Pincher-Robot-Arm.aspx>.
- [17] J. Penalva, «Kinect: Precio, fecha de lanzamiento y juegos,» 2010. [En línea]. Available: <https://www.xataka.com/default/kinect-fecha-de-lanzamiento-y-juegos>.

- [18] M. Echezuria, «Microsoft libera el SDK de Kinect para Windows,» [En línea]. Available: <http://www.nolapeles.com/2011/06/20/microsoft-libera-el-sdk-de-kinect-para-windows/>.
- [19] J. Han, L. Shao, D. Xu y J. Shotton, «Enhanced Computer Vision with Kinect,» 2013.
- [20] E. Duque, «Diferencias entre Kinect V1 y Kinect V2,» [En línea]. Available: <https://edwinnui.wordpress.com/2015/02/05/diferencias-entre-kinect-v1-y-kinect-v2-2/>.
- [21] F. o. C. a. I. S. U. o. L. Visual Cognitive Systems Laboratory, «Zbar\_detector - ViCoS ROS Repository,» [En línea]. Available: [https://github.com/vicoslab/vicos\\_ros/tree/master/detection/zbar\\_detector](https://github.com/vicoslab/vicos_ros/tree/master/detection/zbar_detector).
- [22] K. Khoshelham y S. O. Elberink, «Accuracy and Resolution of Kinect Depth Data for Indoor Mapping Applications,» 2012.
- [23] F. A. C. Lucero, «Detección de robo/Abandono de objetos de interiores utilizando cámaras de profundidad,» 2012.
- [24] «OpenCV,» [En línea]. Available: [http://opencv.org/..](http://opencv.org/)
- [25] A. R. Bazaga, «OpenCV: Librería de Visión por Computador,» 18 Agosto 2015. [En línea]. Available: <https://osl.ull.es/software-libre/opencv-libreria-vision-computador/>.
- [26] «Point Cloud Library (PCL),» [En línea]. Available: <http://pointclouds.org/>.
- [27] «ZBar bar code reader,» [En línea]. Available: <http://zbar.sourceforge.net/>.
- [28] J. Santos, «Turtlebot Arm,» [En línea]. Available: [http://wiki.ros.org/turtlebot\\_arm](http://wiki.ros.org/turtlebot_arm).
- [29] I. A. Sucan y S. Chitta, «MoveIt!,» [En línea]. Available: <http://moveit.ros.org/>.
- [30] Rst-TU-Dortmund, «pxpincher\_ros Metapackage,» [En línea]. Available: [https://github.com/rst-tu-dortmund/pxpincher\\_ros/](https://github.com/rst-tu-dortmund/pxpincher_ros/).
- [31] M. Labbe, «Find object 2D,» [En línea]. Available: [http://wiki.ros.org/find\\_object\\_2d](http://wiki.ros.org/find_object_2d).
- [32] R. c. Willow Garage, «ORK: Object Recognition Kitchen,» [En línea]. Available: [https://wg-perception.github.io/object\\_recognition\\_core/](https://wg-perception.github.io/object_recognition_core/).
- [33] P. Malmsten, “Object Discovery with a Microsoft Kinect,” 2012.
- [34] M. C. Marius Muja, «Tabletop Object Detector,» [En línea]. Available: [http://wiki.ros.org/tabletop\\_object\\_detector](http://wiki.ros.org/tabletop_object_detector).
- [35] R. c. Willow Garage, «ORK Tutorials: Object Recognition Database,» [En línea]. Available: [https://wg-perception.github.io/ork\\_tutorials/tutorial01/tutorial.html](https://wg-perception.github.io/ork_tutorials/tutorial01/tutorial.html).
- [36] L. Joseph, Mastering ROS for Robotics Programming, 2015.
- [37] S. T. Ran Hao, «PCL Recognition,» [En línea]. Available: [https://github.com/TuZZiX/pcl\\_recognition](https://github.com/TuZZiX/pcl_recognition).
- [38] «Calibrating the Kinect depth camera to an external RGB camera,» [En línea]. Available:

[http://wiki.ros.org/openni\\_launch/Tutorials/ExtrinsicCalibrationExternal](http://wiki.ros.org/openni_launch/Tutorials/ExtrinsicCalibrationExternal).

[39] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss y W. Burgard, «OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees,» 2012.

[40] K. M. Wurm y A. Hornung, «octomap ROS,» [En línea]. Available: <http://wiki.ros.org/octomap>.