



**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA**

**Grado en Ingeniería Informática - Tecnologías Informáticas**

**Análisis de la eficiencia en las comunicaciones del robot TurtleBot  
con visión procesada en un servidor local**

**Realizado por  
Juan José Requena Lama**

**Dirigido por  
Fernando Díaz del Río**

**Departamento  
Arquitectura y Tecnología de Computadores**

**Sevilla, (06,2021)**

---

# Resumen

---

ROS, siglas de Robot Operating System, se trata de un framework para el desarrollo de software para robots. Debido a su creciente popularidad, se está empezando a utilizar junto con el uso de computación en la nube para la obtención de off-loading computacional en robots.

Sin embargo, dentro de una red local inalámbrica, las necesarias conexiones TCP que el sistema exige para la comunicación entre los llamados nodos de ROS, presentan diferentes inconvenientes en cuanto a límite de ancho de banda, retardos y jitter. En el presente trabajo, se analizan las dificultades encontradas en un robot TurtleBot que transmite las imágenes tomadas por su cámara a un servidor local. Se proponen también posibles soluciones, como la compresión de la imagen previamente a su transferencia o la reducción de la frecuencia de su publicación para conseguir una mejora en el rendimiento de dicha comunicación.

---

# Abstract

---

The Robot Operating System (ROS) is a framework used to develop software for robots. Due to its growing popularity, it has started to be used alongside Cloud Computing to achieve computational off-loading in robots.

However, using a local wireless network, the mandatory TCP connections that the system requires for connecting the so-called ROS nodes, present various concerns in relation to limited-bandwidth, delays and jitter. In the present work, the difficulties found in a TurtleBot robot that sends images taken from its camera to a local server are analyzed. Furthermore, different possible solutions are suggested, such as compressing the image before transmission or reducing the frequency of its publication in order to improve the performance of this communication.

---

# Índice general

---

<b>1. Introducción</b>	<b>1</b>
1.1. Motivación . . . . .	1
1.2. Objetivos . . . . .	1
1.3. Estructura de la memoria . . . . .	1
<b>2. Planificación</b>	<b>3</b>
2.1. Análisis temporal y costes de desarrollo . . . . .	3
2.1.1. Actividades, tareas e hitos . . . . .	3
2.1.2. Cronograma . . . . .	6
2.1.3. Estimación de costes . . . . .	6
2.2. Plan de riesgos . . . . .	7
2.2.1. Identificación de los riesgos . . . . .	7
2.2.2. Planes de contingencia . . . . .	8
2.3. Plan de calidad . . . . .	8
2.3.1. Indicadores . . . . .	8
2.3.2. Plan de mejora . . . . .	8
2.4. Plan de comunicaciones . . . . .	8
<b>3. Tecnologías utilizadas</b>	<b>9</b>
3.1. Introducción . . . . .	9
3.2. ROS . . . . .	9
3.2.1. ¿Qué es ROS? . . . . .	9
3.2.2. Conceptos fundamentales . . . . .	10
3.2.3. Funcionamiento de ROS . . . . .	11
3.2.4. Comandos de ROS . . . . .	13
3.3. TurtleBot . . . . .	13
3.4. Gazebo . . . . .	14
3.5. OpenCV . . . . .	15
<b>4. Implementación</b>	<b>16</b>
4.1. Introducción . . . . .	16
4.2. Configuración de red . . . . .	16
4.3. Diagrama TurtleBot-Servidor . . . . .	17
4.4. frameAnalyzer.py y la clase ROS_Timer . . . . .	20
4.4.1. Diagrama de flujo del nodo . . . . .	20
4.4.2. La clase ROS_Timer . . . . .	21
4.4.3. OpenCV y filtrado de imágenes . . . . .	23
4.5. Compresión de imágenes y frecuencia de publicación . . . . .	25
4.5.1. Compresión de imágenes . . . . .	26
4.5.2. Frecuencia de publicación . . . . .	27
4.6. Aplicaciones reales y follower.py . . . . .	28
<b>5. Resultados</b>	<b>31</b>

5.1. Introducción . . . . .	31
5.2. Evaluación experimental . . . . .	31
5.3. Pérdida de paquetes y Wireshark . . . . .	33
5.4. Aplicación real y calidad de imagen . . . . .	35
5.5. Conclusiones . . . . .	36
<b>6. Conclusiones y trabajo futuro</b>	<b>37</b>
<b>7. Bibliografía</b>	<b>38</b>
<b>Anexos</b>	<b>39</b>
<b>A. Anexo I: Manual de instalación del entorno</b>	<b>39</b>
<b>B. Anexo II: Scripts dentro de turtlebot_vision</b>	<b>42</b>

---

# Índice de figuras

---

2.1. Cronograma del proyecto . . . . .	6
3.1. Logo de ROS . . . . .	9
3.2. Organización del sistema de ficheros de ROS . . . . .	10
3.3. Diagrama de comunicación entre nodos . . . . .	12
3.4. TurtleBot 2 . . . . .	14
3.5. Logo de Gazebo . . . . .	14
3.6. Simulación del robot TurtleBot dentro de Gazebo . . . . .	15
3.7. Logo corporativo de OpenCV . . . . .	15
4.1. Configuración utilizada haciendo uso de simulación . . . . .	16
4.2. Configuración utilizada en el laboratorio . . . . .	17
4.3. Diagrama conceptual TurtleBot-Servidor . . . . .	18
4.4. Contenido dentro de un mensaje sensor_msgs/Image . . . . .	19
4.5. Diagrama conceptual con frameAnalyzer . . . . .	20
4.6. Diagrama de los instantes de las mediciones . . . . .	20
4.7. Diagrama de flujo de frameAnalyzer.py . . . . .	22
4.8. Diagrama de flujo de frameAnalyzer.py . . . . .	24
4.9. Simulación del TurtleBot frente a un contenedor . . . . .	25
4.10. Imágenes obtenidas al ejecutar frameAnalyzer.py . . . . .	25
4.11. Parámetros configurables en /camera/driver . . . . .	27
4.12. yellowWalled.world . . . . .	28
4.13. Imágenes obtenidas al ejecutar follower.py . . . . .	29
4.14. Circuito en el laboratorio . . . . .	30
5.1. Comienzo de la inestabilidad en la transmisión . . . . .	34
5.2. Continua transmisión de TCP Dup ACK . . . . .	34
5.3. Resolución mediante TCP Fast Retransmission . . . . .	34
5.4. Imágenes obtenidas al ejecutar frameAnalyzer.py con compresión JPEG <sub>1</sub>	36

---

# Índice de extractos de código

---

B.1.	frameAnalyzer.py	42
B.2.	ROS_Timer.py	45
B.3.	follower.py	51
B.4.	imageRepubisher.py	54

---

# **1. Introducción**

---

En esta primera sección de la memoria se introducirá al proyecto en cuestión, así como los diferentes motivos por los que se ha decidido trabajarla. Asimismo, se comentarán los objetivos que se esperan alcanzar al final del trabajo, junto con una breve descripción de la estructura del documento.

## **1.1. Motivación**

La aplicación de computación en la nube en el campo de la robótica se trata de un estudio que en los últimos años ha estado en numerosas investigaciones. Los robots presentan la necesidad de tomar grandes cantidades de información sobre su entorno para así procesarla y poder interactuar con este. Liberar al robot de dicha carga resultaría en un cúmulo de ventajas.

Motivado por este paradigma y dado los límites que supone un Trabajo de Fin de Grado, este proyecto tratará únicamente de afrontar un problema concreto, la transmisión de imágenes entre un robot TurtleBot y un ordenador en su misma red local.

Se estudiará el funcionamiento de ROS para conseguir el despliegue y la comunicación entre ambos. También se desarrollará el software necesario para analizar la transmisión de imágenes entre los dispositivos, tanto en un entorno con un robot simulado como con uno real. Por último, se propondrán e implementarán soluciones, como la compresión de la imagen antes de su transmisión o la reducción de la frecuencia de su publicación para intentar resolver los posibles problemas de transferencia que encontremos. Para finalizar, analizaremos los resultados obtenidos.

## **1.2. Objetivos**

El objetivo de este trabajo es, dado un robot móvil con ROS (y como ejemplo de uso, un TurtleBot), que envía las imágenes tomadas por su cámara a un servidor local, identificar los problemas de comunicación existentes al enviar dichas imágenes, proponer una mejora a los problemas detectados y analizar los resultados.

## **1.3. Estructura de la memoria**

A continuación se detalla como está distribuida la presentación de la memoria:

- Capítulo 2: Se determina la planificación seguida durante el desarrollo del proyecto.

- Capítulo 3: Se definen cuales han sido las principales tecnologías y herramientas utilizadas. Se adjuntan explicaciones en cuanto a cómo se utilizarán dichas herramientas.
- Capítulo 4: Se detallan los pasos que se han seguido durante la implementación.
- Capítulo 5: Se analizan y comparan los resultados obtenidos.
- Capítulo 6: Conclusiones y trabajo futuro.
- Apéndices: Incluye el manual de instalación y los scripts principales implementados durante el proyecto.

---

## 2. Planificación

---

En esta sección serán concretadas cuáles son las **actividades** a realizar con sus correspondientes **tareas**, comentando los diferentes **hitos** que resulten de interés y que, seguramente, darán lugar a los diferentes **entregables** con los que podremos valorar el desempeño del proyecto. Además, tendremos que nombrar los conjuntos de personas que se encargarán de realizar las diferentes tareas mediante la **asignación** de roles. Situaremos en una línea temporal dichas tareas mediante un cronograma. Este **cronograma** nos permitirá la visualización del camino crítico así como el menor tiempo requerido para terminar el proyecto.

La estimación de costes del proyecto vendrá dada por la asignación de tareas y roles, añadiendo las diferentes **adquisiciones** que estén previstas.

Además, en este mismo apartado se incluirán los planes de gestión: de riesgos, de calidad y de comunicaciones.

### 2.1. Análisis temporal y costes de desarrollo

Identificaremos en primer lugar cuáles son las actividades, tareas a hitos, las situaremos en un cronograma y, tras esto, detallaremos los costes de desarrollo del proyecto.

#### 2.1.1. Actividades, tareas e hitos

Presentamos en el cuadro 2.1 las diferentes actividades planteadas junto a las tareas a realizar por actividad y el rol del responsable de dicha tarea. En negrita, se encontrarán los diferentes hitos de seguimiento y control.

Se incluye también el cuadro 2.2 que contiene las horas de trabajo aproximadas para cada tarea según el rol del integrante.

Actividad	Rol
<b>Iniciación del proyecto</b>	
Identificar objetivos, alcance, interesados y riesgos	Jefe de proyecto
Acta de constitución del proyecto	Jefe de proyecto
<b>Elaboración de planes</b>	
Plan de riesgos	Analista
Plan de calidad	Analista
Plan de comunicaciones	Analista
<b>Seguimiento y control.</b> Entregable: versión alfa de los planes	
<b>Configuración del robot simulado y servidor en área local</b>	
Instalación del SO en robot y servidor	Administrador
Diseño de la comunicación	Programador
Programación del código y despliegue	Programador
Seguimiento y control. Entregable: primera versión en entorno simulado	Jefe de proyecto
<b>Despliegue con robot real</b>	
Configuración del robot del laboratorio	Administrador
Despliegue del servicio junto al robot	Administrador
<b>Seguimiento y control.</b> Entregable: segunda versión haciendo uso de un robot real	Jefe de proyecto
<b>Análisis</b>	
Pruebas de comunicación	Administrador
Estudio de posibles mejoras	Administrador
Implementación de las mejoras	Programador
Validación de la comunicación	Administrador
<b>Seguimiento y control.</b> Entregable: versión final con las mejoras integradas	Jefe de proyecto
<b>Redacción de la memoria</b>	
Redacción de la memoria	Jefe de proyecto
<b>Cierre del proyecto</b>	
Defensa del trabajo fin de grado	Jefe de proyecto

Cuadro 2.1: Actividades a realizar por rol

Tarea asignada	Horas de trabajo
<b>Jefe de proyecto</b>	<b>97 horas</b>
Identificar objetivos, alcance, interesados y riesgos	20 horas
Acta de constitución del proyecto	8 horas
Entregable: versión alfa de los planes	0 horas
Entregable: primera versión en entorno simulado	0 horas
Entregable: segunda versión haciendo uso de un robot real	0 horas
Entregable: versión final con las mejoras integradas	0 horas
Redacción de la memoria	67 horas
Defensa del trabajo fin de grado	2 horas
<b>Analista</b>	<b>16 horas</b>
Plan de riesgos	6 horas
Plan de calidad	6 horas
Plan de comunicaciones	4 horas
<b>Programador</b>	<b>41 horas</b>
Diseño de la comunicación	6 horas
Programación del código y despliegue	20 horas
Implementación de las mejoras	15 horas
<b>Administrador</b>	<b>46 horas</b>
Instalación del SO en robot y servidor	10 horas
Configuración del robot del laboratorio	12 horas
Despliegue del servicio junto al robot	8 horas
Pruebas de comunicación	5 horas
Estudio de posibles mejoras	7 horas
Validación de la comunicación	4 horas
<b>TOTAL</b>	<b>200 horas</b>

Cuadro 2.2: Horas necesarias para la finalización de cada actividad

## 2.1.2. Cronograma

En el cronograma de la figura 2.1 se puede comprobar las tareas que se solapan en el tiempo. La fecha de inicio coincidirá con el comienzo del segundo cuatrimestre. La

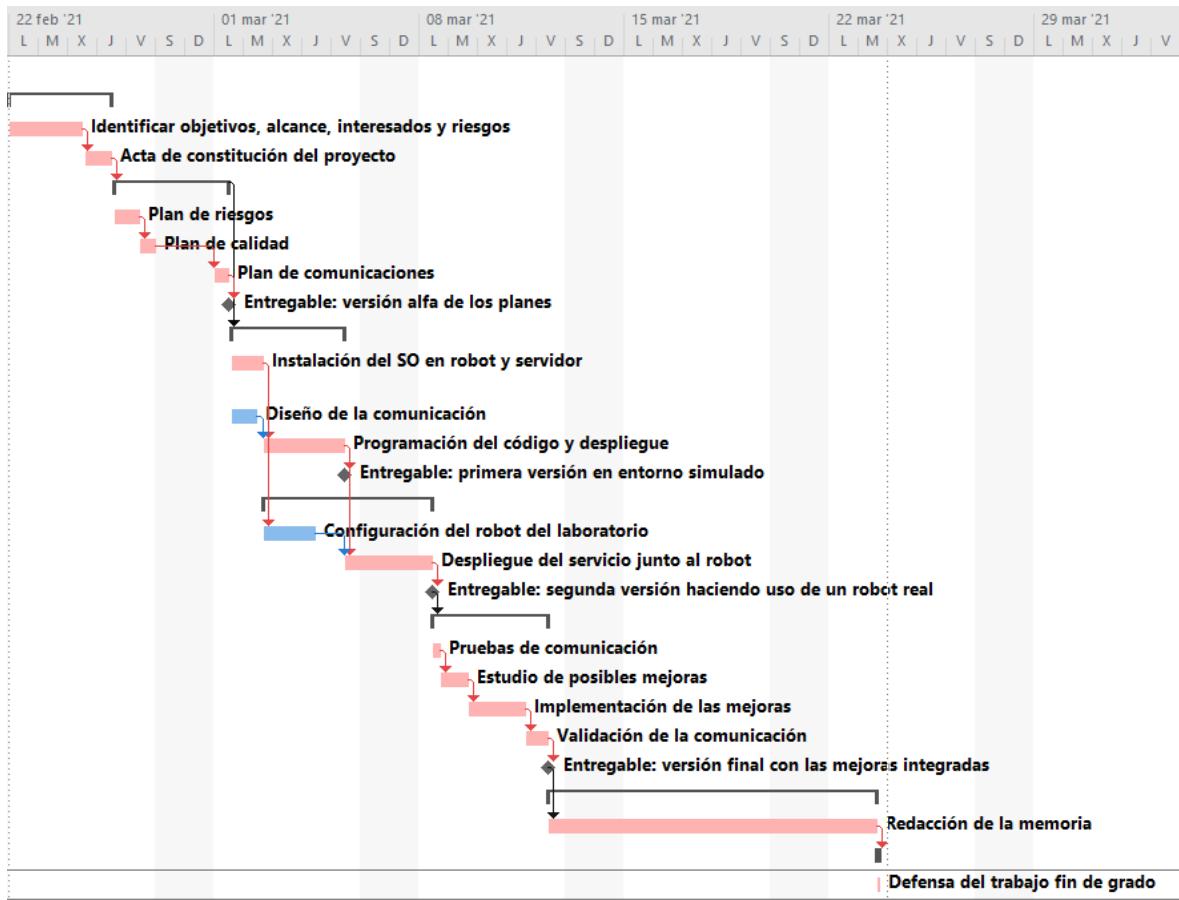


Figura 2.1: Cronograma del proyecto

fecha de finalización se trata de una estimación, han sido asignadas 8 horas disponibles de trabajo por día para los diferentes roles determinados, trabajando de lunes a viernes.

El proyecto requiere de 200 horas de trabajo. En total serían 21,88 días para completarlo según las restricciones y horarios dispuestos. El **caminio crítico** está marcado en rojo y es el que ha determinado la duración en días de este proyecto.

La defensa del proyecto ha sido añadida como una tarea más a realizar después de la redacción de la memoria, aunque, en realidad, su ejecución se haga sobre julio de este mismo año. Se ha hecho así únicamente para facilitar el cálculo del cronograma y duración del proyecto.

## 2.1.3. Estimación de costes

Este proyecto no va a presentar costes económicos al tratarse de un proyecto realizado para la asignatura de Trabajo de Fin de Grado, aun así, se ha decidido

adjuntar tablas haciendo referencia a los costes de personal que supondría la realización de este proyecto en un escenario laboral.

El siguiente cuadro indica la lista de precios/hora estimada en el informe de la Junta de Andalucía **Informe final sobre la consulta preliminar del mercado “perfiles profesionales ámbito informático”** realizado en los años 2018-2019.

Rol	Precio (€/hora)
Jefe de proyecto	48,75
Analista	39,98
Administrador	43,11
Programador	33,77

Cuadro 2.3: Lista de precios/hora según rol

Esto, sumado al cálculo de horas por tarea realizado anteriormente, nos da como resultado un coste final de **8736.06€** reflejado en la siguiente tabla:

Rol	Precio (€/hora)	Horas de trabajo	Precio total
Jefe de proyecto	48,75	97 horas	4728.75€
Analista	39,98	16 horas	639.68€
Administrador	43,11	46 horas	1983.06€
Programador	33,77	41 horas	1384.57€
		<b>200 horas</b>	<b>8736.06€</b>

Cuadro 2.4: Costes totales del proyecto

## 2.2. Plan de riesgos

### 2.2.1. Identificación de los riesgos

Posibles riesgos:

- Los retrasos que pueden suponer compatibilizar la realización del proyecto con los estudios del alumno.
- La curva de aprendizaje que presenta ROS para comenzar a hacer uso de las funcionalidades que presenta.
- Dificultad de comunicación entre tutor y alumno debido a la imposibilidad de recibir tutorías físicas.
- El uso de un robot con una versión de ROS desactualizada y cuyos paquetes ya no estén soportados.

## **2.2.2. Planes de contingencia**

Para resolver los diferentes problemas a los que nos podamos encontrar, podríamos recurrir, en cada caso, a:

- Reorganizar el cronograma, si se dispone de margen para ello hasta la fecha límite de entrega.
- Buscar material de apoyo e información en los foros de discusión de ROS que proporcionarán soluciones a muchos de los problemas que nos podamos encontrar.
- Hacer uso de otro programa de programación diferente a C como Python aunque sea menos favorable para que el programador se sienta más cómodo con la implementación del código.

## **2.3. Plan de calidad**

### **2.3.1. Indicadores**

- Limpieza y comentarios introducidos en el código para su mejor entendimiento.
- Experiencia a nivel de usuario a la hora de interactuar con el robot.
- Satisfacción de los tiempos de respuesta requeridos.
- Validación de los requerimientos iniciales.

### **2.3.2. Plan de mejora**

En cada control que se realice durante el proyecto, existirán una serie de comentarios, sugerencias y valoraciones realizadas por todas las partes implicadas en el proyecto. Se atenderán a dichas observaciones en la medida de lo posible.

## **2.4. Plan de comunicaciones**

Se realizarán tutorías online entre el tutor (cliente) y el alumno (jefe de proyecto) cuando se alcancen los hitos del cronograma. También se realizarán tutorías no planificadas entre alumno y tutor para la resolución de conflictos o dudas planteadas. Se comunicarán los resultados a los diferentes interesados del proyecto, así como a la comunidad de usuarios de ROS para la obtención posterior de feedback.

---

## 3. Tecnologías utilizadas

---

### 3.1. Introducción

Para este proyecto, bien se podría diseñar y poner en marcha un robot desde cero o, utilizar un framework ya existente hasta conseguir el entorno deseado. Dada la dificultad que conlleva desarrollar un entorno en su totalidad para un solo robot y, debido a que esto se aleja de los objetivos del proyecto, se ha decidido hacer uso de la última de estas dos opciones. Utilizaremos un framework que disponga de herramientas que ofrezcan una capa de abstracción al ingeniero.

En este capítulo se introducirán las diferentes tecnologías e instrumentos utilizados para la realización del trabajo. Explicaremos únicamente aquellos apartados que consideremos clave para la comprensión del desarrollo del proyecto.

### 3.2. ROS

#### 3.2.1. ¿Qué es ROS?

ROS (Robot Operating System) [1] es el framework con el que desplegaremos software para el robot en este proyecto. No tratándose de un sistema operativo, provee los servicios que se esperarían de uno. Tales como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos o mantenimiento de paquetes.

ROS, cuyo logo se muestra en la figura 3.1<sup>1</sup>, se desarrolló originalmente en 2007 bajo el nombre de switchyard por el Laboratorio de Inteligencia Artificial de Stanford y se ha convertido en un proyecto de código abierto, consiguiendo una gran comunidad activa y colaborativa.



Figura 3.1: Logo de ROS

---

<sup>1</sup>Fuente: [www.ros.org](http://www.ros.org)

### 3.2.2. Conceptos fundamentales

ROS utiliza una arquitectura de grafos donde el procesamiento se realiza en los nodos, que pueden recibir, mandar y multiplexar mensajes de sensores, actuadores y otros nodos.

#### Sistema de ficheros

Como en un sistema operativo real, los ficheros dentro de ROS están organizados en una manera específica dentro del disco duro, tal y como se indica en la siguiente figura:

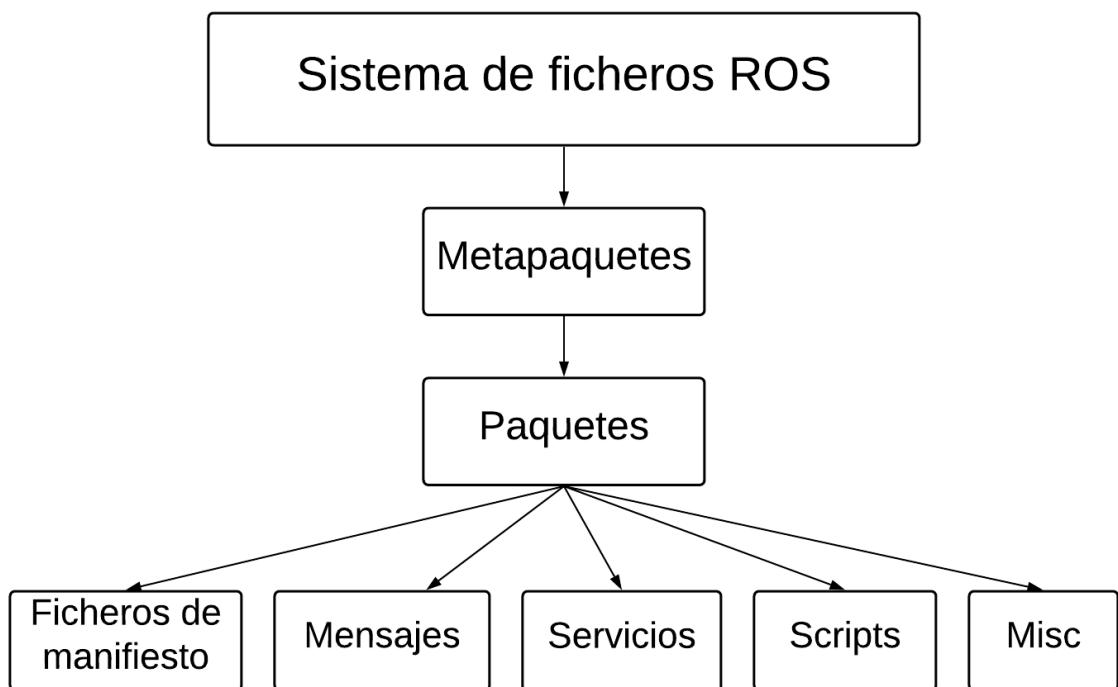


Figura 3.2: Organización del sistema de ficheros de ROS

En el sistema de archivos de ROS, se pueden diferenciar:

- Paquetes (Packages): Son la unidad de organización principal y más básica de software en ROS. Un paquete contiene uno o más programas (nodos), librerías y archivos de configuración, todo organizado como una única unidad.
- Metapaquetes (Metapackages): El término metapaquete se refiere a una colección de uno o más paquetes que se pueden destinar a un proceso común. Se trata de una capa de abstracción para agrupar varios paquetes en una misma tarea.
- Ficheros de manifiesto (Manifest files): Son ficheros dentro de paquetes o metapaquetes que proporcionan información relevante sobre ellos como puede ser el autor, licencia, dependencias del paquete y demás.

- Mensajes (Messages): Los mensajes dentro de ROS contienen la información que es enviada de un proceso a otro. Los ficheros de este tipo definen la estructura de dichos mensajes en ROS.
- Servicios (Services): Un servicio en ROS se trata de una especie de interacción petición-respuesta entre procesos. Estos archivos detallan el tipo de petición o respuesta dentro de una comunicación cliente-servidor en ROS.

### **Red de computación**

A continuación se describirán los diferentes elementos que forman parte del proceso de comunicación entre nodos:

- Nodo: Los nodos son los procesos que se están ejecutando. Un nodo es un archivo ejecutable programado en C++ o Python dentro de un paquete de ROS. Estos nodos dan una forma modular al sistema al ejecutarse individualmente. Aunque estarán comunicados por los llamados topics que veremos más adelante. Un sistema de control del robot se compone, por lo general, de muchos nodos.
- Master (Maestro): Es el proceso principal encargado de organizar la ejecución de todos los nodos. Para la realización de esta tarea, hace uso de servicios mediante los cuales permite a los nodos comunicarse entre sí.
- Servidor de parámetros: Pila del Master que contiene todos los parámetros que pueden ser consultados y actualizados por los nodos.
- Mensajes: Es la forma de comunicación entre nodos. Un mensaje es una estructura simple de datos con campos tipados.
- Topics (Temas): El topic es un canal de comunicación entre nodos. Los topics son buses de datos donde los nodos pueden intercambiar mensajes entre ellos. El nombre del topic se usa para identificar el contenido del mensaje. Los topics implementan un mecanismo de comunicación publish/subscribe donde los nodos podrán publicar mensajes sobre un topic (Publisher) y otros nodos podrán suscribirse a este mismo topic para recibir los mensajes publicados (Subscriber).
- Bags (Bolsas): Archivos encargados de guardar y reproducir los diferentes mensajes que se han enviado.
- Servicios: Los servicios son otra manera de enviar datos entre nodos de ROS. No son más que llamadas síncronas a procedimientos remotos; permiten que un nodo llame a una función que se ejecuta en otro nodo y obtener una respuesta concreta.

#### **3.2.3. Funcionamiento de ROS**

Un sistema ROS está formado por muchos nodos que son ejecutados simultáneamente y que se comunican los unos con los otros a través de mensajes, todo ello con la ayuda del Master.

La forma mediante la cual opera ROS es la siguiente. El Master es iniciado y le da un nombre al sistema. Durante el arranque, cada nodo de ROS se registrará y pedirá encontrar al Master los diferentes nodos y streams de datos solicitados. Tras ello, cada nodo le indicará al Master qué tipo de mensaje provee y a qué nodo le gustaría suscribirse. Finalmente, el Master proporcionará las respectivas direcciones de los publicadores y suscriptores, consiguiendo así que los nodos se conecten directamente entre ellos. El protocolo de conexión habitual que se utilizará entre los nodos dentro de ROS será TCP.

En la figura 3.1, se introduce un ejemplo simple de comunicación entre dos nodos dentro de ROS.

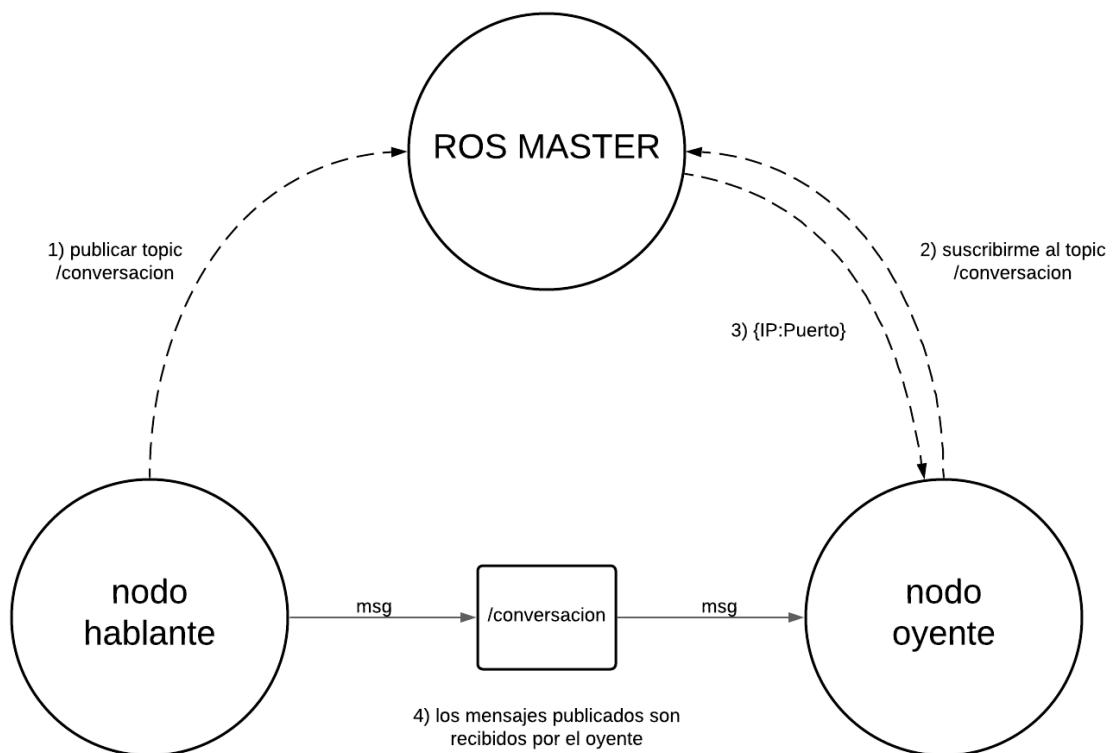


Figura 3.3: Diagrama de comunicación entre nodos

1. El nodo hablante informa al Master sobre su intención de publicar mensajes en el topic /conversacion.
2. El nodo oyente le pide al Master la información para conectarse al topic.
3. El Master le indica la dirección IP y puerto al oyente.
4. Se realizarán las conexiones TCP correspondientes para establecer conexión entre los nodos y, tras ello, los mensajes publicados por el nodo hablante llegarán al nodo oyente a través del topic.

Este desacoplamiento entre los diferentes componentes del sistema se ha conseguido mediante el uso de topics como medio de comunicación entre nodos. Un nodo podrá suscribirse a otro sin saber si ese otro está publicando o, por el contrario, un

nodo podrá publicar mensajes sin saber si otro está suscrito a él. Esto posibilita que los nodos puedan ser iniciados, reiniciados o eliminados sin necesidad de generar conflictos.

### 3.2.4. Comandos de ROS

Algunos de los comandos o líneas de comandos más importantes que se mencionarán en este trabajo serán:

- `rosrun <paquete> <ejecutable>`: este comando permite correr un ejecutable dentro de un paquete desde cualquier directorio sin tener que dar su ruta completa o encontrarse en el mismo directorio que el ejecutable.
- `roscore`: se trata de una colección de nodos y programas requeridos previamente para iniciar un sistema ROS. Debe de existir un `roscore` en ejecución para que los nodos dentro de ROS se comuniquen. Todo esto se lanza usando el comando `roscore`.
- `roslaunch <paquete> <fichero.launch>`: es una herramienta para lanzar fácilmente múltiples nodos dentro de ROS, tanto de forma local como de forma remota a través de SSH. También permite establecer parámetros en el servidor de parámetros. Una ejecución de `roslaunch` iniciará automáticamente `roscore` si detecta que no hay ninguno activo.
- `rostopic <opcion> <topic>`: `rostopic` se trata de una línea de comandos que muestra información sobre los diferentes topics de ROS. Entre las diferentes opciones utilizaremos:
  - `rostopic bw`: muestra el ancho de banda consumido por un topic.
  - `rostopic hz`: muestra la frecuencia de publicación de un topic.
  - `rostopic list`: imprime información sobre los topics actualmente activos.

## 3.3. TurtleBot

TurtleBot [2] es un kit para tener un robot personal de bajo coste que hace uso de ROS como si de un sistema operativo se tratase. TurtleBot fue creado en Willow Garage por Melonee Wise y Tully Foote en noviembre de 2010.

Dada su disponibilidad en el departamento, será el robot TurtleBot 2, mostrado en la figura 3.4, el que utilizaremos para realizar las pruebas en el laboratorio. En las simulaciones también haremos uso de un TurtleBot simulado.

TurtleBot 2 cuenta con las siguientes características:

- Base móvil Yujin Kobuki.
- Sensor 3D Microsoft Kinect.
- Portatil Asus 1215N con procesador de doble núcleo.

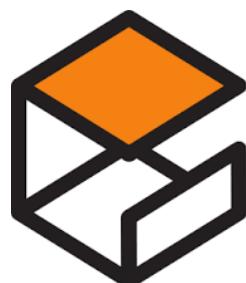
- Batería de 2200 mAh.
- Cargador de carga rápida.
- Kit de montaje de hardware para unir todos los componentes y añadir en un futuro otros sensores.



Figura 3.4: TurtleBot 2

### 3.4. Gazebo

Gazebo [3] es un simulador de entornos 3D de código abierto que permite al usuario analizar el comportamiento de un robot en un entorno simulado. Gazebo, cuyo logo es mostrado en la figura 3.5<sup>2</sup>, incluye un motor de físicas para simular la gravedad, inercias o iluminación para no tener la necesidad de construir un escenario físico.



GAZEBO

Figura 3.5: Logo de Gazebo

Respalgado por la Open Source Robotics Foundation (OSRF), el Gazebo project realiza un gran trabajo de adaptación a ROS, de hecho, la aplicación ya es instalada

---

<sup>2</sup>Fuente: <http://gazebosim.org/>

conjuntamente cuando uno instala ROS. Se muestra en la figura 3.6 una simulación sencilla del robot TurtleBot realizada mediante Gazebo.

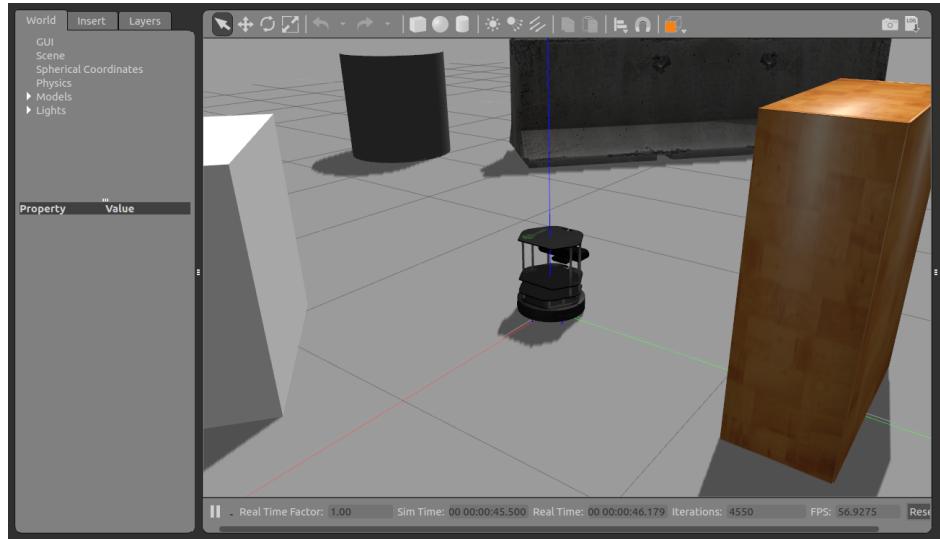


Figura 3.6: Simulación del robot TurtleBot dentro de Gazebo

### 3.5. OpenCV

OpenCV (Open Source Computer Vision Library) [4] es una biblioteca de software de visión artificial y aprendizaje automático de código abierto. OpenCV se construyó para proporcionar una infraestructura común para las aplicaciones de visión por ordenador y para acelerar el uso de la percepción de las máquinas en productos comerciales.

Al ser un producto con licencia BSD, OpenCV facilita a las empresas el uso y modificación del código. Tiene interfaces C++, Python, Java y MATLAB y es compatible con Windows, Linux, Android y Mac OS. En la figura 3.7<sup>3</sup> se muestra su logo corporativo.

En este proyecto, haremos uso de la versión 3.3.1 de OpenCV para el procesamiento de las imágenes que obtengamos mediante la Kinect de nuestro TurtleBot.



Figura 3.7: Logo corporativo de OpenCV

<sup>3</sup>Fuente: <https://opencv.org/>

---

# 4. Implementación

---

## 4.1. Introducción

En el anterior capítulo hemos comentado las diferentes tecnologías que vamos a utilizar. Ahora, pasaremos a detallar qué pasos hemos seguidos para la elaboración del proyecto en su totalidad. Explicaremos también los fundamentos teóricos y prácticos necesarios para una completa comprensión de lo que estamos haciendo.

Como ya se indicó en la sección 1.2, queremos enviar las imágenes capturadas por un robot móvil con ROS a un servidor que se encuentra en su misma red local y analizar dichas comunicaciones. Haremos uso de dos escenarios: uno simulando el robot TurtleBot con Gazebo y otro haciendo uso del robot real.

Cada escenario presentará características diferentes, tanto de configuración de red como de funcionamiento dentro de los scripts diseñados, los detalles más importantes serán remarcados a lo largo del capítulo.

Cabe mencionar que, en el anexo A se encuentra un manual de instalación del entorno sobre del que partiremos en este capítulo. Además, en el anexo B se encuentran todos los scripts que posteriormente serán mencionados.

## 4.2. Configuración de red

En primer lugar, mencionaremos a grandes rasgos la configuración de red de la que partiremos en ambos escenarios.

En el primero de los casos, como es mostrado en la figura 4.1, dispondremos de un router encargado de comunicar el servidor con el portátil que ejecuta la simulación de Gazebo. Se comunicarán router y portátil de manera inalámbrica, mientras que el

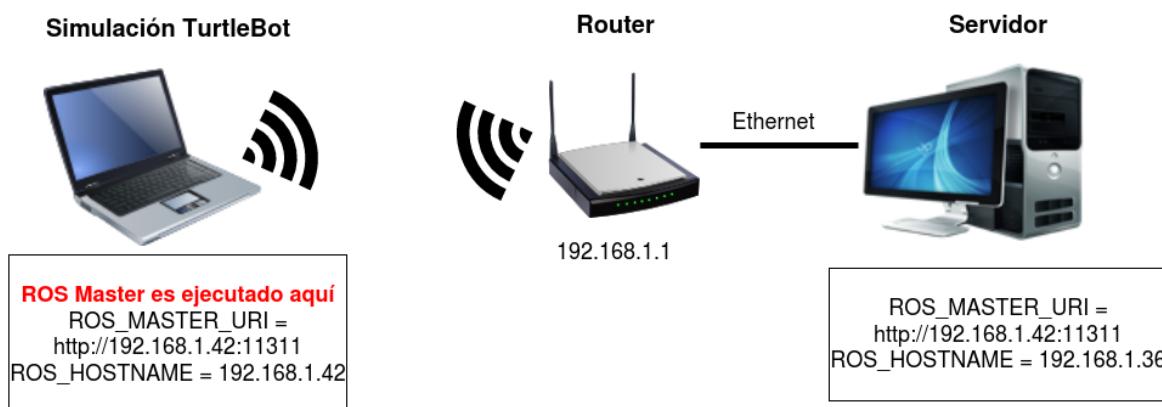


Figura 4.1: Configuración utilizada haciendo uso de simulación

router y el servidor lo harán mediante un cable de red. Es importante remarcar que, el ROS Master será ejecutado en el portátil que lleva la simulación del TurtleBot.

En cambio, puede observarse en la figura 4.2 cómo fue configurado el escenario donde tenemos un TurtleBot real.

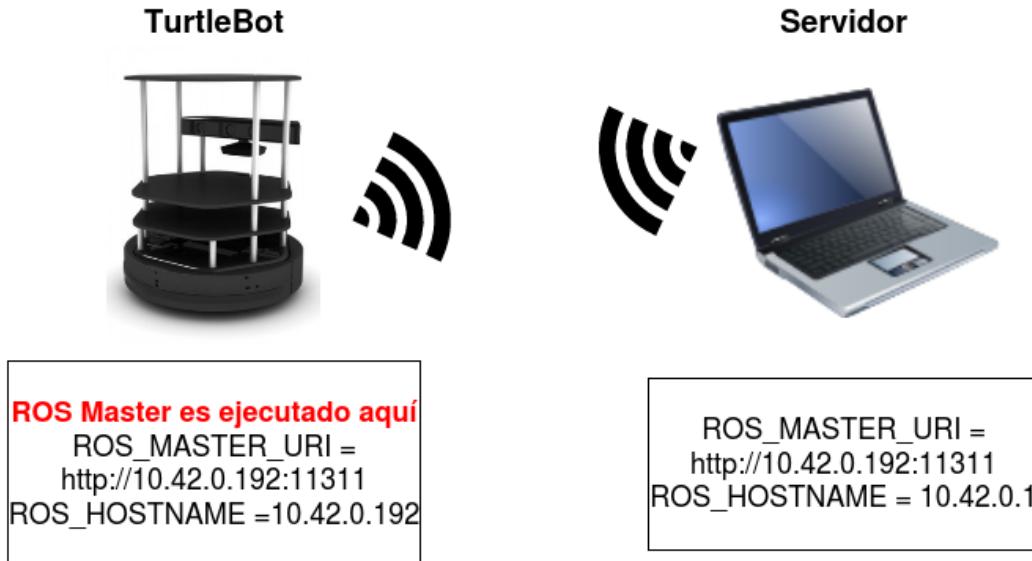


Figura 4.2: Configuración utilizada en el laboratorio

Aquí, el portátil que hará de servidor, creará un punto de acceso WiFi portátil al que se conectaría el robot, siendo la conexión entre ellos inalámbrica. De nuevo, remarcamos que el ROS Master se iniciará en el TurtleBot.

Ninguno de los dos escenarios lo consideramos usual en robots, estos presentan con frecuencia mayor autonomía, sin estimarse adecuada la necesidad de comunicarse con un servidor.

Sí pudiendo residir mayor interés al ligar la similitud en requisitos de red que podría compartir con sistemas multirobot. En estas redes, los datos intercambiados entre robots deben transferirse dentro de ventanas de tiempo fijas para permitir que las tareas se completen con éxito. Nuestros escenarios, a diferencia del anterior, nos permiten obtener resultados más precisos de las mediciones que realicemos, evitando así posible ruido introducido por la habitual presencia de redes multisalto en sistemas multirobot.

A partir de la siguiente sección en adelante, utilizaremos la imagen del TurtleBot para representar de manera indiferente tanto al robot real como al portátil con la simulación.

### 4.3. Diagrama TurtleBot-Servidor

Ahora que se ha aclarado la configuración utilizada, comenzaremos a explicar qué estrategia se ha utilizado para afrontar el problema. En primer lugar, debemos de ave-

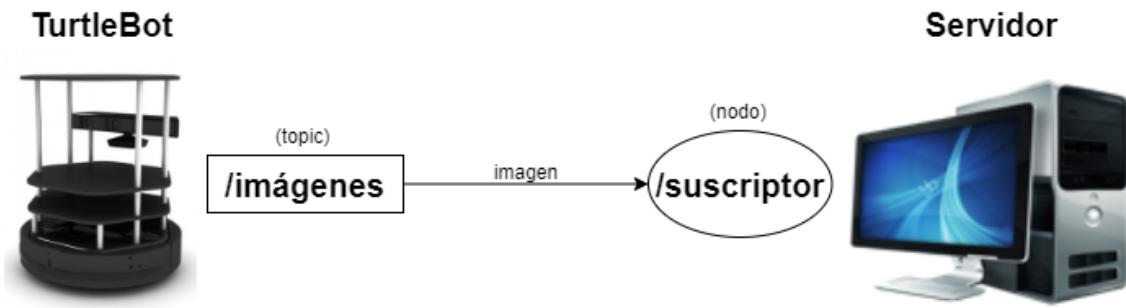


Figura 4.3: Diagrama conceptual TurtleBot-Servidor

riguar en qué topic son publicadas las imágenes tomadas por la Kinect del TurtleBot. Partiremos del diagrama mostrado en la figura 4.3, y a partir de ahí iremos profundizando.

Tras ejecutar una simulación del TurtleBot mediante el comando:

```
$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

Podremos ver todos los topics publicados por el TurtleBot ejecutando rostopic list en un terminal diferente:

```
$ rostopic list
/camera/depth/camera_info
/camera/depth/image_raw
/camera/depth/points
/camera/parameter_descriptions
/camera/parameter_updates
/camera/rgb/camera_info
/camera/rgb/image_raw
/camera/rgb/image_raw/compressed
/camera/rgb/image_raw/compressed/parameter_descriptions
/camera/rgb/image_raw/compressed/parameter_updates
/camera/rgb/image_raw/compressedDepth
/camera/rgb/image_raw/compressedDepth/parameter_descriptions
/camera/rgb/image_raw/compressedDepth/parameter_updates
--More--
```

Es `/camera/rgb/image_raw` el topic que nos interesa para este trabajo. Aquí es donde la Kinect publica las imágenes que toma sin realizar transformación alguna. En el TurtleBot real, el topic donde publicará las imágenes es `/camera/rgb/image_color`. Para saber el tipo de mensaje que es transmitido en este topic, ejecutaremos el comando rostopic info sobre él:

```
$ rostopic info /camera/rgb/image_raw
Type: sensor_msgs/Image

Publishers:
 * /gazebo (http://localhost:41293/)
```

**Subscribers:** None

Así es cómo sabremos que los mensajes publicados son del tipo **sensor\_msgs/Image**. El contenido incluido dentro de cada mensaje se muestra en la figura 4.4<sup>1</sup>, tomada de la documentación oficial de ROS.

## **sensor\_msgs/Image Message**

**File:** `sensor_msgs/Image.msg`

### **Raw Message Definition**

```
# This message contains an uncompressed image
# (0, 0) is at top-left corner of image
#
Header header          # Header timestamp should be acquisition time of image
# Header frame_id should be optical frame of camera
# origin of frame should be optical center of camera
# +x should point to the right in the image
# +y should point down in the image
# +z should point into to plane of the image
# If the frame_id here and the frame_id of the CameraInfo
# message associated with the image conflict
# the behavior is undefined
uint32 height           # image height, that is, number of rows
uint32 width             # image width, that is, number of columns
#
# The legal values for encoding are in file src/image_encodings.cpp
# If you want to standardize a new string format, join
# ros-users@lists.sourceforge.net and send an email proposing a new encoding.
string encoding          # Encoding of pixels -- channel meaning, ordering, size
# taken from the list of strings in include/sensor_msgs/image_encodings.h
uint8 is_bigendian        # is this data big endian?
uint32 step               # Full row length in bytes
uint8[] data              # actual matrix data, size is (step * rows)
```

### **Compact Message Definition**

```
std_msgs/Header header
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

Figura 4.4: Contenido dentro de un mensaje `sensor_msgs/Image`

Esta información y, en concreto, la incluida en la cabecera o *header* será muy valiosa en el apartado siguiente donde veremos cómo calcular la latencia mediante los popularmente conocidos *timestamps* o marcas temporales.

<sup>1</sup>Fuente: [https://docs.ros.org/en/melodic/api/sensor\\_msgs/html/msg/Image.html](https://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/Image.html)

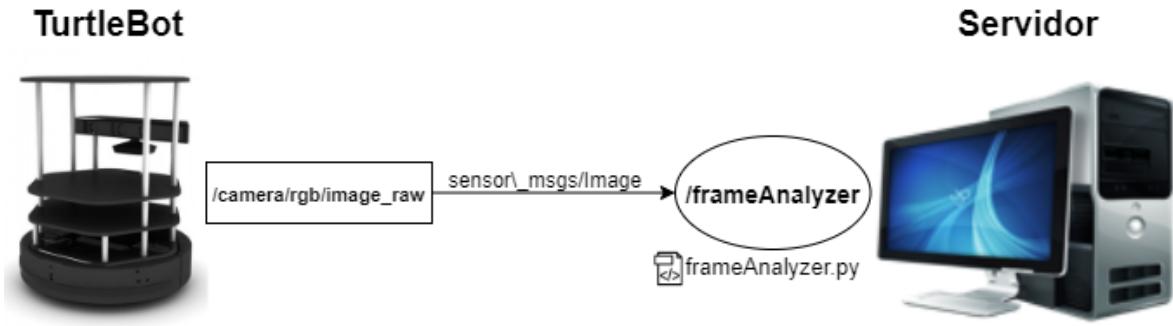


Figura 4.5: Diagrama conceptual con frameAnalyzer

Para capturar las imágenes enviadas por el TurtleBot, se creará un nodo suscriptor llamado frameAnalyzer que será ejecutado en el servidor. Este mismo nodo será el encargado de medir los retardos de red y otras estadísticas que necesitaremos para estudiar el rendimiento de la comunicación TurtleBot-Servidor. Para elaborar este nodo, tendremos que crear un script con el mismo nombre, frameAnalyzer.py, quedando el diagrama como muestra la figura 4.5.

## 4.4. frameAnalyzer.py y la clase ROS\_Timer

Si bien en las anteriores secciones hemos visto las características del entorno en el que trabajamos y, cómo vamos a operar sobre este, ahora veremos en detalle qué es lo que nos ofrece el *package* que se ha desarrollado, **turtlebot\_vision**. En concreto, veremos el script frameAnalyzer.py y la clase en la que se apoya, ROS\_Timer.

### 4.4.1. Diagrama de flujo del nodo

Para evaluar las comunicaciones entre TurtleBot y servidor, hay precisos momentos en la transmisión de la imagen cuyos timestamps debemos conocer. Estos momentos son: (Ver figura 4.6)

1. Al capturar el sensor Kinect la imagen.
2. Cuando el nodo frameAnalyzer recibe el frame.
3. Tras procesar la imagen.

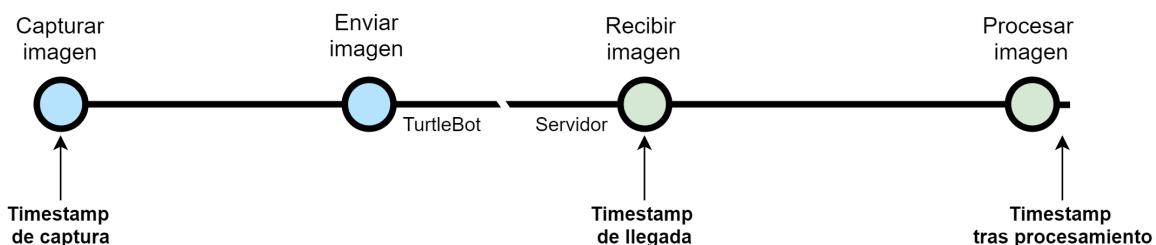


Figura 4.6: Diagrama de los instantes de las mediciones

Los timestamps de dichos instantes van a presentar gran valor para nosotros pues la diferencia entre el primero y el segundo nos indicará la latencia de red aproximada y el resto del tercero con el segundo timestamp mostrará el tiempo que nuestro nodo puede tardar en procesar la imagen.

Siendo conocedores de esta información, podemos ya introducir el diagrama de flujo creado para nuestro script. Dicho diagrama se presenta en la figura 4.7. Más tarde en esta misma sección, especificaremos las diferentes actividades aún no explicadas como la conversión de la imagen a un objeto de OpenCV y su filtrado o la escritura de la información recopilada.

#### 4.4.2. La clase ROS\_Timer

Para recoger toda la información que necesitamos y hacer uso de ella de manera despreocupada, hemos creado una clase llamada ROS\_Timer. Comentaremos únicamente algunas de sus funcionalidades y estructuras de datos. Si se desea entrar en detalle sobre la implementación de sus diferentes funciones, puede dirigirse al anexo (B.2). Mediante las apropiadas llamadas, esta clase se encargará de:

- Guardar información relevante de cada imagen y escribirla en un fichero llamado **data\_collected.csv**. Cada frame incluirá:
  - Número de secuencia del frame.
  - Timestamp al ser tomada la imagen.
  - Timestamp al recibir la imagen.
  - Timestamp tras procesar la imagen.
  - Número de frames perdidos entre la anterior imagen y esta.
  - Latencia de red (s).
  - Retardo de procesamiento de la imagen (s).
  - Retardo total (s).
- Analizar dicha información, escribiendo los resultados en el fichero **useful\_info.txt**. Obtendrá:
  - Tiempo de simulación (s).
  - Número total de imágenes recibidas.
  - Número total de imágenes perdidas.
  - Porcentaje de imágenes perdidas.
  - Latencia de red media (ms).
  - Máxima latencia de red registrada (ms).
  - Mínima latencia de red registrada (ms).
  - Jitter medio (ms).

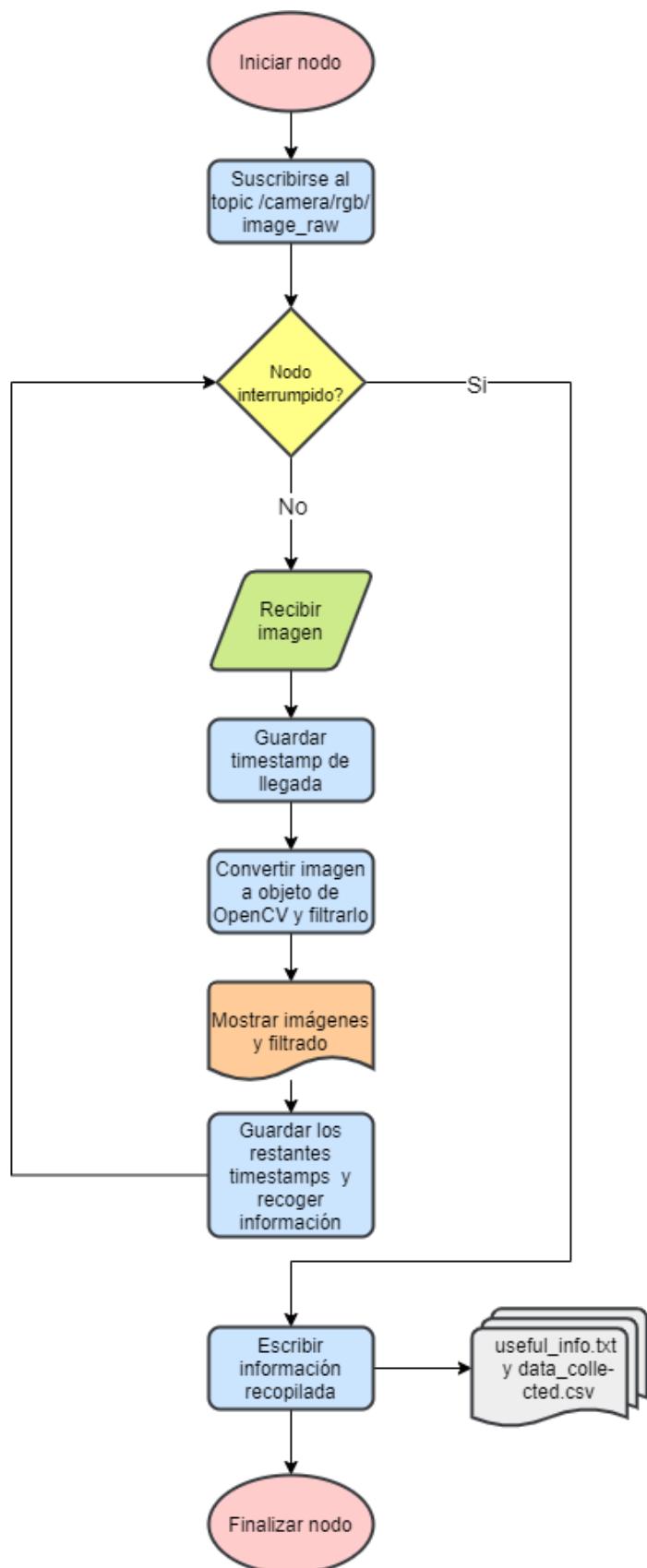


Figura 4.7: Diagrama de flujo de `frameAnalyzer.py`

- Frecuencia aproximada de frames recibidos (FPS).
- Consumo medio de ancho de banda (Mb/s).

Serán entonces los diferentes ficheros `data_collected.csv` y `useful_info.txt` que obtengamos en los múltiples escenarios los que utilizaremos para analizar los resultados en el capítulo 5.

En las tablas 4.1 y 4.2 se encuentra contenido de ejemplo tal y como quedaría recogido en cada uno de los archivos. Esta información se ha conseguido al ejecutar una simulación de TurtleBot y recogiendo los datos en el mismo ordenador. De ahí que las estadísticas obtenidas sean muy diferente a las que más tarde veremos.

Seq. No.	Camera stamp	Reception stamp	Post- processing stamp	Network delay (s)	Processing delay (s)	Total delay (s)
1	46.13	46.58	46.58	0.45	0.0	0.45
2	46.23	46.65	46.66	0.42	0.01	0.43
3	46.33	46.7	46.71	0.37	0.01	0.38
4	46.43	46.74	46.74	0.31	0.0	0.31
5	46.53	46.75	46.77	0.22	0.02	0.24
6	46.63	46.79	46.8	0.16	0.01	0.17
7	46.73	46.83	46.85	0.1	0.02	0.12
8	46.84	46.88	46.89	0.04	0.01	0.05
9	46.94	46.97	46.98	0.03	0.01	0.04
10	47.03	47.05	47.05	0.02	0.0	0.02

Cuadro 4.1: `data_collected.csv`

<b>TOTAL TIME SPENT ON SIMULATION</b>	33.3847680092 s
<b>TOTAL NUMBER OF IMAGES RECEIVED</b>	355 images
<b>TOTAL NUMBER OF IMAGES LOST</b>	0 images
<b>% OF IMAGES LOST</b>	0.0 %
<b>MEAN LATENCY ACHIEVED</b>	23.4366197183 ms
<b>MAX LATENCY ACHIEVED</b>	450.0 ms
<b>MIN LATENCY ACHIEVED</b>	9.9999999999 ms
<b>AVERAGE JITTER</b>	12.4011299435 ms
<b>APROX FREQUENCY</b>	10.6335919394 FPS
<b>AVERAGE BANDWIDTH CONSUMPTION</b>	74.9780558401 Mb/s

Cuadro 4.2: `useful.info.txt`

#### 4.4.3. OpenCV y filtrado de imágenes

Ahora que ya sabemos cómo funciona la clase `ROS_Timer` y lo que nos puede ofrecer, solo nos queda por ver cómo vamos a tratar las imágenes cuando sean recibidas

por el nodo frameAnalyzer.

Al llegar una imagen a nuestro nodo, esta se encuentra en el formato proporcionado por ROS. Si queremos hacer uso de ella, tendremos que convertirla a un objeto manipulable por OpenCV. Para ello, vamos a apoyarnos en la librería de ROS llamada **CvBridge** [5]. Mostramos en la figura 4.8<sup>2</sup> cómo esta librería va a hacer de puente entre ambos formatos de imagen.

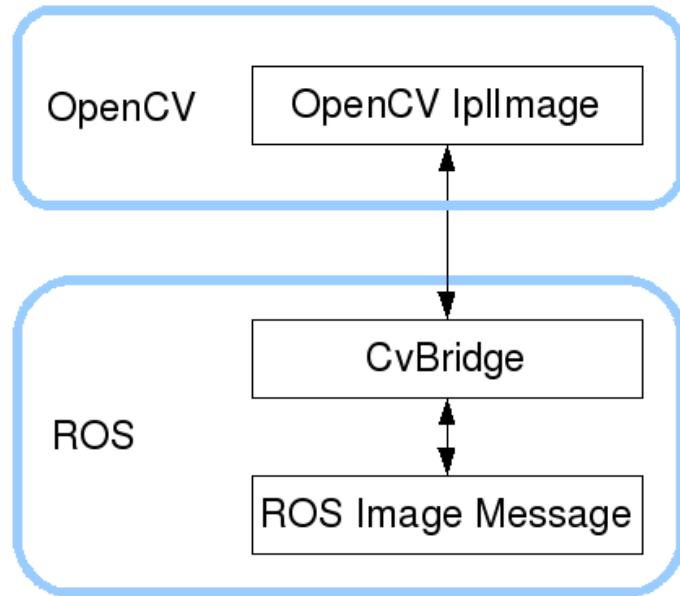


Figura 4.8: Diagrama de flujo de frameAnalyzer.py

Para convertir una imagen de ROS en una para OpenCV, hemos incluido el siguiente código en nuestro script frameAnalyzer.py:

```
1 from cv_bridge import CvBridge  
2 bridge = CvBridge()  
3 cv_image = bridge.imgmsg_to_cv2(image_message,  
        desired_encoding='passthrough')
```

Tras haber conseguido la imagen en el formato deseado, solo nos queda procesarla. En nuestro caso, hemos optado por una simple operación: filtrar el color verde. Es decir, hemos creado una función llamada **bgr\_to\_hsv(imagen)** que, además de transformar la imagen de BGR a HSV, va a filtrar todo lo que sea verde, ocultando todo lo demás.

Presentamos a continuación como quedaría el resultado. La figura 4.9 muestra la posición del TurtleBot dentro de una simulación.

<sup>2</sup>Fuente: [http://wiki.ros.org/cv\\_bridge](http://wiki.ros.org/cv_bridge)

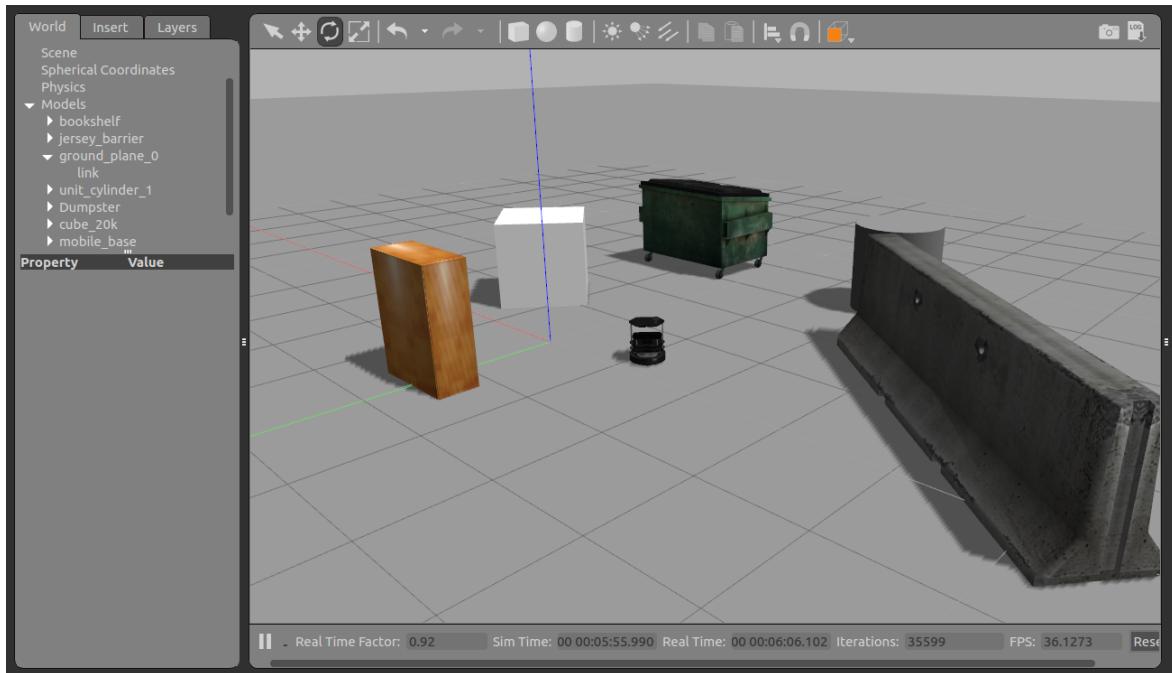


Figura 4.9: Simulación del TurtleBot frente a un contenedor

Tras ejecutar el código de nuestro script, se van a mostrar 3 imágenes tal y como aparecen en la figura 4.10. Estas imágenes son:

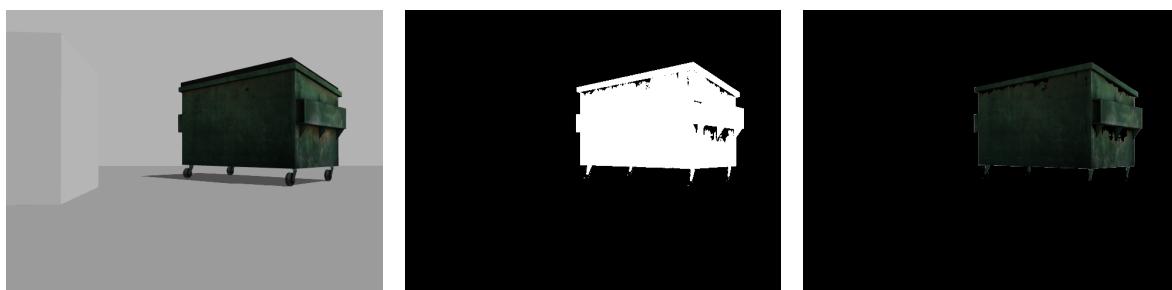


Figura 4.10: Imágenes obtenidas al ejecutar frameAnalyzer.py

La figura 4.10b, muestra el filtro que ha sido aplicado sobre la imagen original, revelando de color blanco todo aquello que finalmente se incluirá. El resultado de este filtro y la imagen original, es mostrado en la figura 4.10c.

#### 4.5. Compresión de imágenes y frecuencia de publicación

Las principales funcionalidades que necesitábamos para este proyecto ya han sido detalladas en las secciones anteriores. No obstante, a la vista de los resultados que obtendremos en el capítulo 5, donde se presenta tanto una gran latencia de red como

un gran porcentaje de paquetes perdidos en su transmisión, tenemos aún que aclarar un par de consideraciones que hemos tenido para mejorar estos parámetros y que nos ha llevado al resto de scripts implementados.

En esta sección trataremos la compresión de imágenes y la frecuencia de publicación. El uso de aplicaciones reales será tratado en la sección [4.6](#).

#### 4.5.1. Compresión de imágenes

Nuestro fichero frameAnalyzer.py implementa un nodo que se suscribe al topic `/camera/rgb/image_raw`. Dado que las imágenes originales presentan un gran tamaño (en torno a 0.8 MB en simulaciones), se nos van a presentar complicaciones a la hora de transmitir dichos frames.

Es por eso que, haremos uso de la compresión de imágenes previa a su envío para obtener mejores resultados. Existe un topic llamado `/camera/rgb/image_raw/compressed` que ya vimos tras ejecutar el comando `rostopic list` en la sección [4.3](#).

Mediante el comando:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

Podremos acceder a los parámetros de configuración del topic, consiguiendo así generar cuatro configuraciones:

- Compresion jpeg (quality = 80)
- Compresion jpeg (quality = 1)
- Compresion png (level = 9)
- Compresion png (level = 1)

Concretamos, un valor de **jpeg quality** alto va a indicar que la imagen apenas ha sido modificada, un valor bajo considerará que la compresión será máxima pero también disminuirá drásticamente la calidad de la imagen. No siendo lo mismo para el **png level**, donde un valor alto indicará que menos tiempo en comprimir ha empleado y más pesará la imagen. En cambio, un valor bajo tardará más tiempo y conseguirá una mejor compresión.

Además, hemos implementado un script llamado **imageRepubliser.py** contenido en el anexo [B.4](#). Este script es lanzado en el robot, ejecutando lo siguiente:

1. Se suscribe al topic `/camera/rgb/image_raw`.
2. Cada imagen que le llega es comprimida mediante OpenCV.
3. Publica la imagen comprimida en el topic `/image_half_res`.

El paso 2 puede ser conseguido convirtiendo primeramente cada imagen recibida a formato OpenCV y luego, aplicando la técnica de compresión deseada, ya sea con o sin pérdida de información, como podría ser la codificación por transformación, el *delta encoding* o la codificación entrópica.

Debido a su sencillez, se ha optado por reducir el número de píxeles de la imagen a la mitad, tanto verticalmente como horizontalmente. La librería OpenCV presenta la función `resize` que utiliza interpolación bilineal para llevar a cabo esto:

```
cv_modified = cv2.resize(cv_image, (0,0), fx = 0.5, fy = 0.5)
```

Consiguiendo así que la imagen resultante tenga una resolución de 1/4 de la original.

#### 4.5.2. Frecuencia de publicación

Sabemos que por defecto, el sensor Microsoft Kinect tiene una tasa de fotogramas o *framerate* igual a 30 FPS, es decir, captura 30 imágenes por segundo. Con esta misma frecuencia las publicará en el topic `/camera/rgb/image_raw` o `/camera/rgb/image_color` según se utilice simulación o el robot real.

Debido a que 30 hz se trata de una frecuencia muy alta de publicación para unas imágenes que, además se tienen que enviar a través de una red inalámbrica, probaremos a disminuir el valor de dicha frecuencia.

Haciendo uso del robot real, podremos cambiar el valor de este parámetro mediante el uso nuevamente de `rqt_reconfigure`. Desde la interfaz gráfica que se despliega, tendremos que buscar el nodo `/camera/driver` junto con el parámetro `data_skip`. En la figura 4.11 se le asigna un valor igual a 2 para conseguir una frecuencia de publicación de 10 hz.

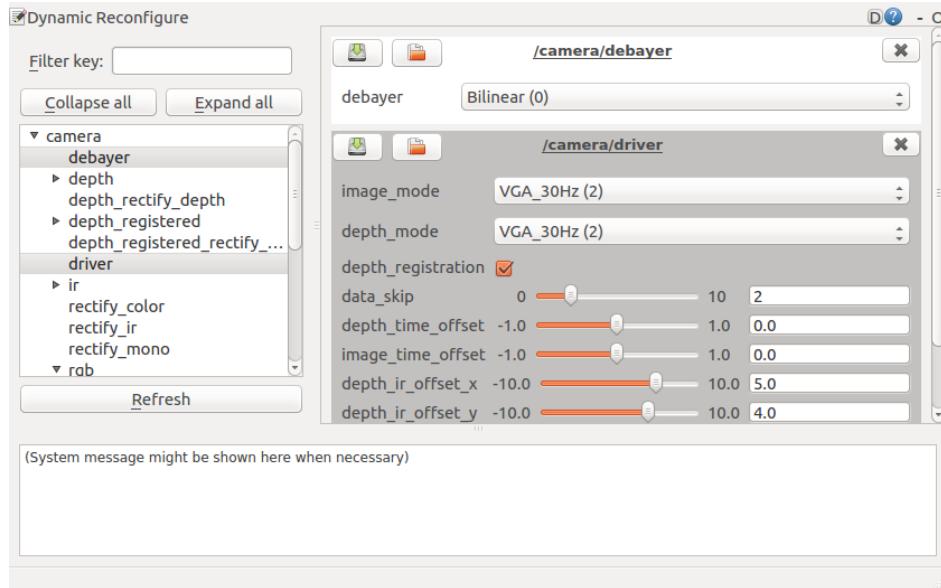


Figura 4.11: Parámetros configurables en `/camera/driver`

Utilizando la simulación, los pasos para realizar esta configuración variarán. Para cambiar la frecuencia de publicación de imágenes del TurtleBot tendremos que dirigirnos al directorio `/opt/ros/kinect/share/turtlebot_description/urdf/` y editar el fichero `turtlebot_gazebo.urdf.xacro`. Dentro de él tendremos que dirigirnos al parámetro `updateRate` del driver de la cámara y asignarle un valor de 10.0 para conseguir dicha frecuencia.

## 4.6. Aplicaciones reales y follower.py

Para finalizar este capítulo, vamos a comentar el uso de una aplicación con utilidad real en este proyecto, **follower.py**.

Como ya sabemos, los scripts realizados nos permiten analizar la transmisión de imágenes entre el TurtleBot y el servidor local. Ahora bien, el objetivo de mandar estos datos a un servidor es para que dicho servidor le mande de vuelta un resultado o respuesta al TurtleBot.

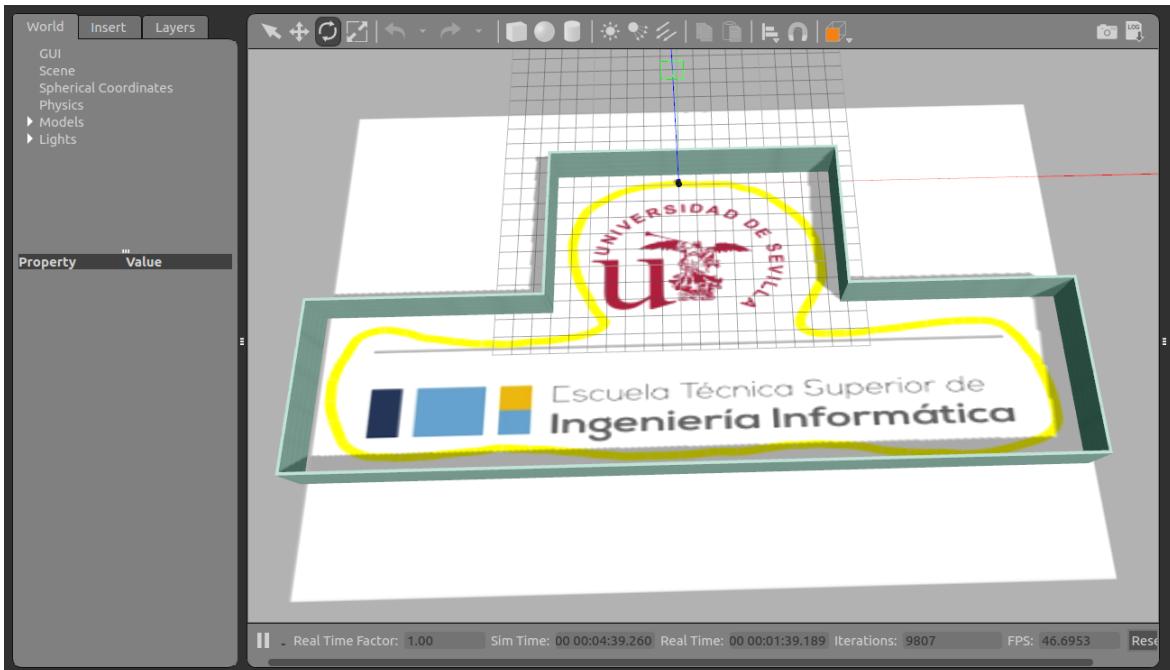


Figura 4.12: yellowWalled.world

El programa follower.py incluido en el anexo B.3, se encarga de cumplir dicha función. Este script está basado en el follower incluido en el libro Programming Robots with ROS: a practical introduction to the Robot Operating System [6]. Mediante su ejecución, conseguirá que el robot TurtleBot sea capaz de seguir un circuito amarillo dibujado en el suelo. Procesará las imágenes desde el servidor y mandará los comandos al TurtleBot. Realizará las siguientes funciones en orden:

1. Suscribirse a un topic de imágenes.
2. Filtrar el color amarillo en cada imagen que recibe.
3. Obtiene la sección de la imagen que se encuentra a 1 metro aproximado del robot.
4. Calcula el centro de masas de dicha sección y traza un círculo rojo sobre este.
5. Computa la velocidad angular.
6. Publica la velocidad en un topic al que está suscrito el TurtleBot.

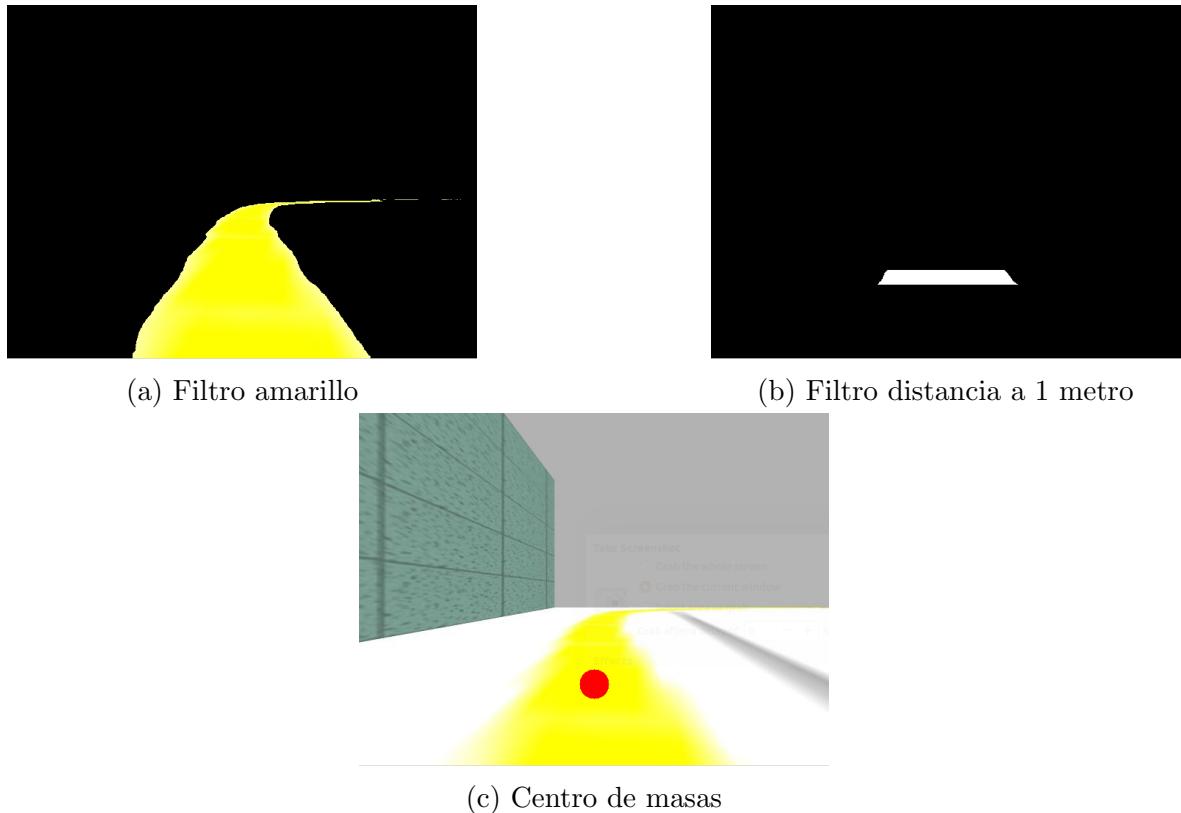


Figura 4.13: Imágenes obtenidas al ejecutar follower.py

El propio programa nos mostrara por pantalla el resultado de los pasos 2, 3 y 4 como se expone en la figura 4.13.

Para probar el programa, hemos tenido que crear un archivo .world personalizado en el que hubiese dibujado en el suelo un circuito trazado con lineas amarillas. El resultado es el archivo **yellowWalled.world**, mostrado en la figura 4.12, que se encuentra en la carpeta *worlds* de nuestro repositorio<sup>3</sup>. Para el escenario real, hemos dibujado un circuito en el laboratorio también formado por líneas amarillas. El resultado se muestra en la figura 4.14.

Con esto, hemos terminado de explicar toda la configuración realizada y software implementado para el proyecto. En el siguiente capítulo veremos los resultados obtenidos en los diferentes escenarios y examinaremos el rendimiento utilizando las diferentes configuraciones.

---

<sup>3</sup><https://github.com/wuaho/turtlebot-vision>

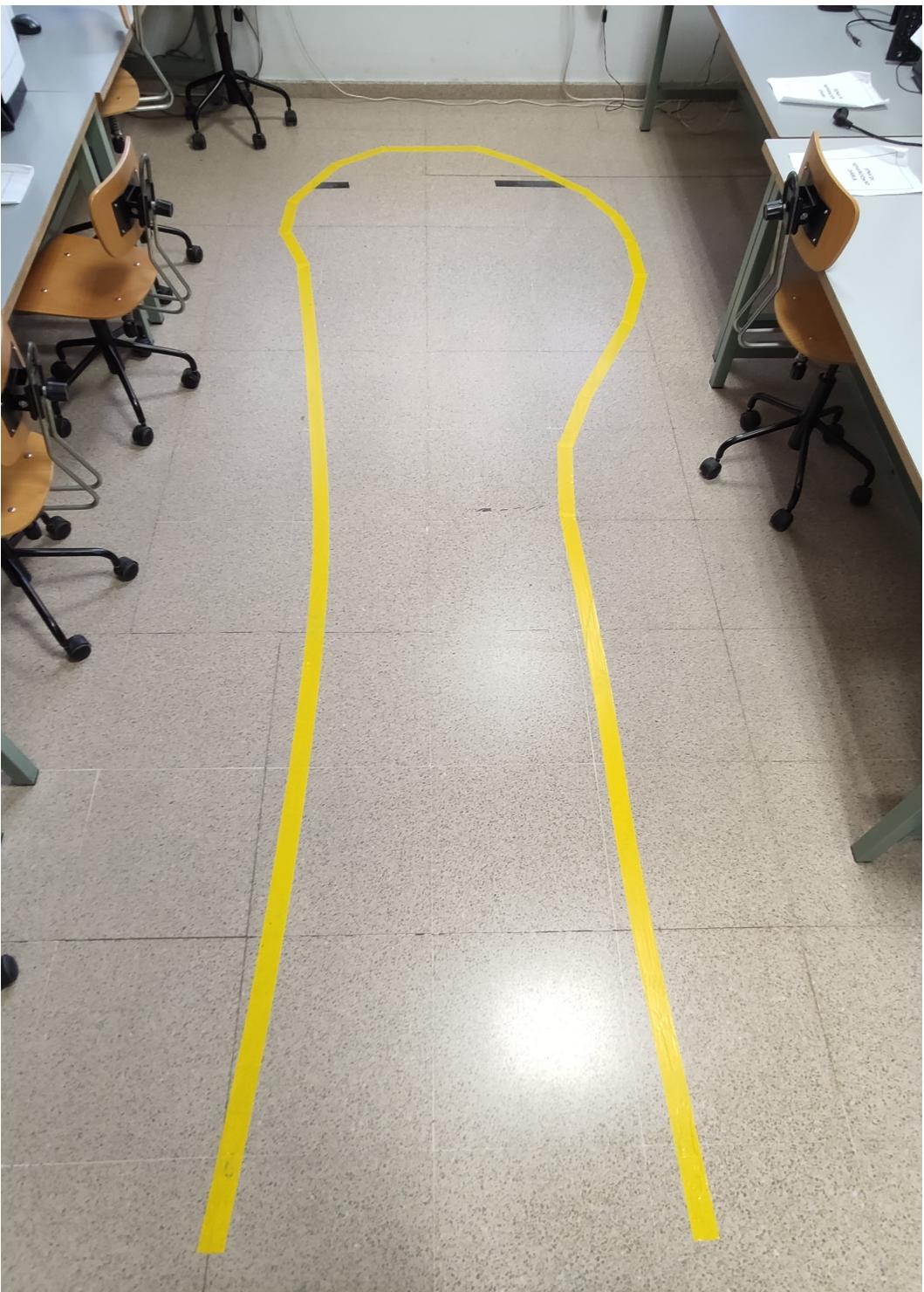


Figura 4.14: Circuito en el laboratorio

---

# 5. Resultados

---

## 5.1. Introducción

En el presente capítulo se desarrolla el análisis de los resultados obtenidos mediante las dos configuraciones de TurtleBot comentadas anteriormente: simulando el robot en un portátil mediante Gazebo y haciendo uso de un TurtleBot 2 real disponible en el laboratorio.

Para ello, se hará uso de los scripts desarrollados y se llevarán a cabo las simulaciones pertinentes. Una vez analizados ambos escenarios en conjunto, las conclusiones sacadas serán recogidas en la sección 5.5.

## 5.2. Evaluación experimental

Como ya se anticipó anteriormente, en este experimento hemos considerado dos robots, simulado y real, para probar la transmisión inalámbrica de imágenes a un servidor de nuestra red local. El sensor Kinect ha sido configurado para que publique por defecto a 30 hz tanto en simulación como en el robot real. Los datos han sido recogidos de los ficheros useful\_info.txt obtenidos mediante la clase ROS\_Timer tras 300 segundos de ejecución. Las conexiones entre los nodos de ROS se realizan a través de TCP.

Los métodos de envío de imágenes utilizados en este trabajo junto a sus identificadores son:

- Orig: imagen original, sin modificación alguna.
- Orig<sub>10</sub>: imagen original pero la frecuencia de publicación de imágenes por el sensor Kinect es de 10 hz.
- JPEG<sub>80</sub>: imagen comprimida utilizando codificación JPEG donde se ha especificado un valor igual a 80 en el parámetro quality.
- JPEG<sub>1</sub>: imagen comprimida utilizando codificación JPEG donde se ha especificado un valor igual a 1 en el parámetro quality.
- PNG<sub>9</sub>: imagen comprimida haciendo uso de compresión PNG donde se ha otorgado un valor igual a 9 en el parámetro level.
- PNG<sub>1</sub>: imagen comprimida a mediante compresión PNG donde se ha otorgado un valor igual a 1 en el parámetro level.
- Res: imagen con 1/4 de la resolución de la imagen original. Se ha conseguido utilizado el script imageRepubliser.py.

Dos diferentes sets de experimentos han sido realizados para considerar las dos configuraciones ya detalladas en la sección 4.2. Los experimentos tenían como objetivo analizar el jitter y la latencia de red en el flujo TurtleBot → Servidor. El retardo fue medido calculando la diferencia entre el timestamp del robot al capturar la imagen y el timestamp recogido en el servidor al recibir el mensaje. Recogeremos el retardo medio ( $R$ ), el mínimo retardo ( $R_{\text{Min}}$ ) y el máximo retardo observado ( $R_{\text{Max}}$ ).

Entendemos por jitter a la fluctuación o variación en el tiempo de la entrega de dos imágenes consecutivas. Caracteriza la variación del retardo de la red. Calcularemos el jitter ( $J_d$ ) como la media de la diferencia de la latencia observada entre una imagen y la siguiente.

Recopilaremos otros valores como la cantidad de imágenes recibidas por el servidor (IMG). El porcentaje de imágenes perdidas en su envío lo llamaremos PIP. Se utilizará  $T_{\text{IMG}}$  para describir el tamaño medio por imagen, medida que luego nos servirá para calcular el ancho de banda medio consumido (AB). Sabiendo que se utiliza el estándar IEEE 802.11ac en las conexiones inalámbricas dentro de nuestra red, se debería poder alcanzar tasas de transferencia de hasta 433 Mbit/s. Por último, también almacenaremos la tasa de refresco (TR) percibida por el servidor.

El cuadro 5.1 muestra los resultados obtenidos para el escenario donde utilizamos el TurtleBot simulado en el portátil, mientras que la tabla 5.2 presenta los resultados recogidos haciendo uso del TurtleBot real.

	IMG (imágenes)	PIP (%)	R (ms)	$R_{\text{Max}}$ (ms)	$R_{\text{Min}}$ (ms)	$J_r$ (ms)	TR (FPS)	$T_{\text{IMG}}$ (KB)	AB (Mbps)
Orig	2900	74.9	234.7	630.0	170.0	30.9	9.7	900.31	67.95
Orig <sub>10</sub>	2288	38.8	361.2	1430.0	190.0	65.7	7.6	900.39	53.62
JPEG <sub>80</sub>	9187	0.3	19.1	380.0	10.0	8.8	30.6	16.43	3.93
JPEG <sub>1</sub>	9347	0.0	14.5	280.0	10.0	8.3	31.1	5.63	1.37
PNG <sub>9</sub>	2056	0.0	150.5	260.0	50.0	8.0	6.9	84.68	4.53
PNG <sub>1</sub>	7498	0.0	28.1	420.0	10.0	6.4	25.0	83.06	16.21
Res	9697	0.0	237.4	1470.0	10.0	14.7	32.3	225.02	56.80

Cuadro 5.1: Resultados para el TurtleBot simulado

	IMG (imágenes)	PIP (%)	R (ms)	$R_{\text{Max}}$ (ms)	$R_{\text{Min}}$ (ms)	$J_r$ (ms)	TR (FPS)	$T_{\text{IMG}}$ (KB)	AB (Mbps)
Orig	587	93.4	579.6	916.6	227.5	128.7	2.0	901.53	13.78
Orig <sub>10</sub>	646	78.3	567.0	1168.7	237.4	130.2	2.2	901.39	15.16
JPEG <sub>80</sub>	8955	0.17	25.5	301.2	11.0	4.9	29.8	26.09	5.70
JPEG <sub>1</sub>	8966	0.0	18.0	187.4	7.2	3.9	29.9	5.82	1.37
PNG <sub>9</sub>	1618	64.1	357.1	671.9	132.0	117.8	5.4	329.64	13.99
PNG <sub>1</sub>	1902	57.6	323.0	648.5	135.0	92.9	6.3	331.00	16.39
Res	2148	76.1	203.8	798.9	47.9	201.7	7.2	225.10	7.16

Cuadro 5.2: Resultados para el TurtleBot real

Destacamos en los resultados obtenidos, el alto porcentaje de imágenes perdidas

en las transmisiones Orig mediante simulación. Valores que se repetirán en transferencias PNG y Res utilizando el escenario del laboratorio.

La latencia media supera con creces los 100 ms en ambos escenarios, no siendo así cuando el formato de imagen se trata de JPEG. Difieren los resultados de ambos escenarios en este parámetro con la configuración PNG<sub>1</sub>, donde la simulación va a dar una menor latencia. Los valores de latencia máxima alcanzan los 1000 ms.

Como era de esperarse, el jitter se disparará en el escenario real. Además, sobresale el gran trabajo de compresión que realiza la codificación JPEG en el TurtleBot real, siendo el tamaño de la imagen resultante menor al 3 % del tamaño de la imagen original.

### 5.3. Pérdida de paquetes y Wireshark

Como hemos visto en los resultados de la sección anterior, cuando se transmiten las imágenes en su formato original, Orig, llegamos a un porcentaje muy alto de frames perdidos. Tanto en simulación como en el escenario del laboratorio. En esta sección inspeccionaremos a bajo nivel a qué se debe esta inconsistencia en la transmisión de paquetes.

Para ello, hemos hecho uso de Wireshark<sup>1</sup>, una muy popular aplicación en el análisis de paquetes de red. Mientras se ejecutaba una simulación, hemos activado la captura de paquetes desde el servidor para luego buscar aquellos relacionados con ROS.

Para su búsqueda, hemos filtrado todos los paquetes que tuviesen en la IP de origen o destino la del TurtleBot. Siendo **192.168.1.42** la IP del portátil con la simulación del TurtleBot y **192.158.1.36** la IP del ordenador servidor.

Tras filtrar dichos paquetes, hemos encontrado la siguiente traza:

Comienza con la ausencia de confirmación de recepción de un paquete, *TCP Previous Segment Not Captured* por parte del portátil que ejecuta la simulación. Como respuesta, el servidor empezará a enviar *TCP Dup ACK* hasta que le reenvíen el paquete que se perdió. Mientras tanto, el portátil seguirá enviando paquetes que no tendrán respuesta por el servidor. Ver figura 5.1.

Tras esto, el TurtleBot seguirá enviando nuevos paquetes mientras el servidor continuará pidiendo el paquete que se perdió como se muestra en la figura 5.2

Por último, para volver a enviarle el paquete perdido al servidor, el portátil realizará un *TCP Fast Retransmission* (ver figura 5.3). El servidor recibirá este mensaje y posteriormente empezará a responder a todos los paquetes restantes en su pila. El portátil le comunicará que los paquetes están en un orden diferente al original. Justo antes de realizar el Fast Retransmission, el portátil envía un paquete con etiqueta PSH al servidor que rápidamente este responderá debido a la naturaleza de dicha etiqueta. Tras responder a todos los paquetes en su pila, el servidor llegará al último paquete que le envió el portátil antes del Fast Retransmission y a partir de ahí la comunicación volverá a ser estable.

---

<sup>1</sup><https://www.wireshark.org/>

No.	Time	Source	Destination	Protocol	Length	Info
74276	63.639843188	192.168.1.36	192.168.1.42	TCP	66	49856 - 41353 [ACK] Seq=2229 Ack=197204899 Win=3145728 Len=0 TsvaI=3990236440 Tscr=766928225
74277	63.640956724	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=197204899 Ack=2229 Win=64128 Len=2896 TsvaI=766928225 Tscr=3990236421
74272	63.640980415	192.168.1.36	192.168.1.42	TCP	66	49856 - 41353 [ACK] Seq=2229 Ack=197207786 Win=3145729 Len=0 TsvaI=3990236444 Tscr=766928225
74273	63.6403606492	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=197207786 Ack=2229 Win=64128 Len=2896 TsvaI=766928225 Tscr=3990236421
74274	63.640343444	192.168.1.36	192.168.1.42	TCP	66	49856 - 41353 [ACK] Seq=2229 Ack=197210682 Win=3145728 Len=0 TsvaI=3990236444 Tscr=766928225
74275	63.640578139	192.168.1.42	192.168.1.36	TCP	1514	41353 - 49856 [ACK] Seq=197210682 Ack=2229 Win=64128 Len=1448 TsvaI=766928225 Tscr=3990236421
74276	63.640566393	192.168.1.42	192.168.1.36	TCP	1514	[TCP Previous segment not captured] 41353 - 49856 [ACK] Seq=197215029 Ack=2229 Win=64128 Len=1448
74277	63.640536933	192.168.1.36	192.168.1.42	TCP	78	49856 - 41353 [ACK] Seq=2229 Ack=19721230 Win=3144320 Len=0 TsvaI=3990236444 Tscr=766928225 SLE=10
74278	63.64085637	192.168.1.42	192.168.1.36	TCP	1514	[TCP Previous segment not captured] 41353 - 49856 [ACK] Seq=197228618 Ack=2229 Win=64128 Len=1448
74279	63.640831653	192.168.1.36	192.168.1.42	TCP	86	[TCP Dup ACK 74277#1] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236441
74280	63.640831653	192.168.1.42	192.168.1.36	TCP	78	41353 - 49854 [PSH, ACK] Seq=15845 Ack=352 Win=64849 Len=12 TsvaI=766928225 Tscr=3990236421
74281	63.640840732	192.168.1.42	192.168.1.36	TCP	1514	[TCP Previous segment not captured] 41353 - 49856 [ACK] Seq=197228619 Ack=2229 Win=64128 Len=1448
74282	63.640841441	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#2] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236441
74283	63.640861796	192.168.1.36	192.168.1.42	TCP	66	49854 - 41353 [ACK] Seq=352 Ack=15857 Win=64128 Len=0 TsvaI=3990236441 Tscr=766928225
74284	63.641976316	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=766928225 Tscr=3990236421
74285	63.641081795	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#3] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236442
74286	63.641361268	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=1072336954 Ack=2229 Win=64128 Len=2896 TsvaI=766928227 Tscr=3990236426
74287	63.641365524	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#4] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236442
74288	63.641551212	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=107233850 Ack=2229 Win=64128 Len=2896 TsvaI=766928227 Tscr=3990236422
74289	63.641555818	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#5] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236442
74290	63.641824553	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=107236746 Ack=2229 Win=64128 Len=2896 TsvaI=766928227 Tscr=3990236422
74291	63.641830966	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#6] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236442
74292	63.642077397	192.168.1.36	192.168.1.42	TCP	1514	41353 - 49856 [ACK] Seq=107239642 Ack=2229 Win=64128 Len=1448 TsvaI=766928227 Tscr=3990236422
74293	63.642077937	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#7] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236443
74294	63.642089066	192.168.1.42	192.168.1.36	TCP	1514	41353 - 49856 [ACK] Seq=107241099 Ack=2229 Win=64128 Len=1448 TsvaI=766928229 Tscr=3990236428

Figura 5.1: Comienzo de la inestabilidad en la transmisión

No.	Time	Source	Destination	Protocol	Length	Info
74309	63.648961143	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#15] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236450
74310	63.649264572	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=107261362 Ack=2229 Win=64128 Len=2896 TsvaI=766928235 Tscr=3990236439
74311	63.649210909	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#16] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236450
74312	63.649445989	192.168.1.42	192.168.1.36	TCP	1514	41353 - 49856 [ACK] Seq=107262448 Ack=2229 Win=64128 Len=1448 TsvaI=766928232 Tscr=3990236439
74313	63.649455939	192.168.1.36	192.168.1.42	TCP	2962	41353 - 49856 [ACK] Seq=107267559 Ack=2229 Win=64128 Len=2896 TsvaI=766928235 Tscr=3990236432
74314	63.649682939	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=107268869 Ack=2229 Win=64128 Len=2896 TsvaI=766928235 Tscr=3990236432
74315	63.649682626	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#18] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236450
74316	63.649953146	192.168.1.42	192.168.1.36	TCP	4419	41353 - 49856 [ACK] Seq=107286869 Ack=2229 Win=64128 Len=1448 TsvaI=766928234 Tscr=3990236432
74317	63.649953595	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#19] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236451
74318	63.650203192	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=107272946 Ack=2229 Win=64128 Len=2896 TsvaI=766928235 Tscr=3990236432
74319	63.650227708	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#20] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236451
74320	63.650227193	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=107275842 Ack=2229 Win=64128 Len=2896 TsvaI=766928235 Tscr=3990236432
74321	63.650448855	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#21] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236451
74322	63.650476003	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=107278738 Ack=2229 Win=64128 Len=2896 TsvaI=766928235 Tscr=3990236432
74323	63.650579579	192.168.1.36	192.168.1.42	TCP	1514	41353 - 49856 [ACK] Seq=107281634 Ack=2229 Win=64128 Len=1448 TsvaI=766928235 Tscr=3990236432
74324	63.650595389	192.168.1.42	192.168.1.36	TCP	1514	41353 - 49856 [ACK] Seq=107282944 Ack=2229 Win=64128 Len=1448 TsvaI=766928235 Tscr=3990236452
74325	63.650595176	192.168.1.36	192.168.1.42	TCP	1514	41353 - 49856 [ACK] Seq=107283456 Ack=2229 Win=64128 Len=2896 TsvaI=766928235 Tscr=3990236432
74326	63.650595176	192.168.1.42	192.168.1.36	TCP	1514	41353 - 49856 [ACK] Seq=107284244 Ack=2229 Win=64128 Len=1448 TsvaI=766928235 Tscr=3990236452
74327	63.650595176	192.168.1.36	192.168.1.42	TCP	1514	41353 - 49856 [ACK] Seq=107284244 Ack=2229 Win=64128 Len=1448 TsvaI=766928235 Tscr=3990236452
74328	63.651199799	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=107284538 Ack=2229 Win=64128 Len=2896 TsvaI=766928235 Tscr=3990236433
74329	63.651204862	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#25] 49856 - 41353 [ACK] Seq=2229 Ack=107212130 Win=3144320 Len=0 TsvaI=3990236452
74330	63.651439183	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=107287462 Ack=2229 Win=64128 Len=2896 TsvaI=766928235 Tscr=3990236433

Figura 5.2: Continua transmisión de TCP Dup ACK

No.	Time	Source	Destination	Protocol	Length	Info
74369	63.655670558	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#45] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236456
74370	63.655915192	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=107339554 Ack=2229 Win=64128 Len=2896 TsvaI=766928243 Tscr=3990236439
74371	63.655920956	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#46] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236457
74372	63.656171519	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=107342459 Ack=2229 Win=64128 Len=2896 TsvaI=766928243 Tscr=3990236439
74373	63.656183948	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#47] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236457
74374	63.656426883	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=107345456 Ack=2229 Win=64128 Len=2896 TsvaI=766928243 Tscr=3990236439
74375	63.656433237	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#48] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236457
74376	63.656567890	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=107348242 Ack=2229 Win=64128 Len=2896 TsvaI=766928243 Tscr=3990236439
74377	63.656618424	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#49] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236457
74378	63.656626448	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=107351158 Ack=2229 Win=64128 Len=2896 TsvaI=766928247 Tscr=3990236446
74379	63.656931534	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#50] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236456
74380	63.657151958	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=107354534 Ack=2229 Win=64128 Len=2896 TsvaI=766928247 Tscr=3990236446
74381	63.657181136	192.168.1.36	192.168.1.42	TCP	94	[TCP Dup ACK 74277#51] 49856 - 41353 [ACK] Seq=2229 Ack=197212130 Win=3144320 Len=0 TsvaI=3990236456
74382	63.657415747	192.168.1.42	192.168.1.36	TCP	2962	41353 - 49856 [ACK] Seq=107356923 Ack=2229 Win=64128 Len=2896 TsvaI=766928247 Tscr=3990236446
74383	63.657642156	192.168.1.36	192.168.1.42	TCP	98	41353 - 49854 [PSH, ACK] Seq=15957 Ack=448 Win=64128 Len=2896 TsvaI=766928247 Tscr=3990236441
74385	63.657658543	192.168.1.42	192.168.1.36	TCP	1514	41353 - 49853 [ACK] Seq=107359822 Ack=2229 Win=64128 Len=1448 TsvaI=766928247 Tscr=3990236440
74386	63.657685532	192.168.1.42	192.168.1.36	TCP	1514	41353 - 49853 [ACK] Seq=107361158 Ack=2229 Win=64128 Len=1448 TsvaI=766928247 Tscr=3990236445
74387	63.657693255	192.168.1.36	192.168.1.42	TCP	78	49856 - 41353 [ACK] Seq=2229 Ack=1072123578 Win=3142912 Len=0 TsvaI=3990236459 Tscr=766928247
74388	63.657694487	192.168.1.42				

Este patrón se repetirá numerosas veces en la misma simulación, mostrando así por qué al enviar las imágenes Orig, la comunicación es inestable entre servidor y TurtleBot y de ahí, la alta tasa de paquetes perdidos en los resultados.

## 5.4. Aplicación real y calidad de imagen

Hemos visto por qué la transmisión de imágenes en su formato original mediante ROS no es viable en una red local con las características presentadas. Veamos ahora la ejecución de una aplicación real como puede ser follower.py, el seguidor de líneas amarillas. Recordamos que, el funcionamiento de este script está recogido en la sección [4.6](#) del capítulo anterior.

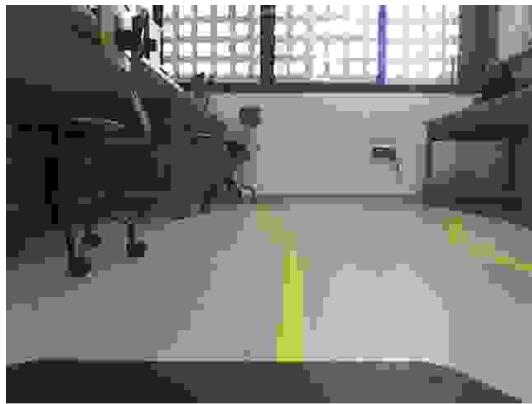
Hemos ejecutado el script follower.py usando cada uno de los métodos de transmisión y hemos observado si el robot consigue completar el circuito. El desenlace ha sido recogido en la tabla [5.3](#). Los resultados han sido idénticos tanto para el robot simulado como para el real. La compresión mediante JPEG<sub>80</sub> es la configuración que nos ha asegurado mejores resultados, consiguiendo terminar el circuito sin problemas.

Exitoso	
Orig	No
Orig <sub>10</sub>	No
JPEG <sub>80</sub>	Si
JPEG <sub>1</sub>	Si
PNG <sub>9</sub>	No
PNG <sub>1</sub>	No
Res	No

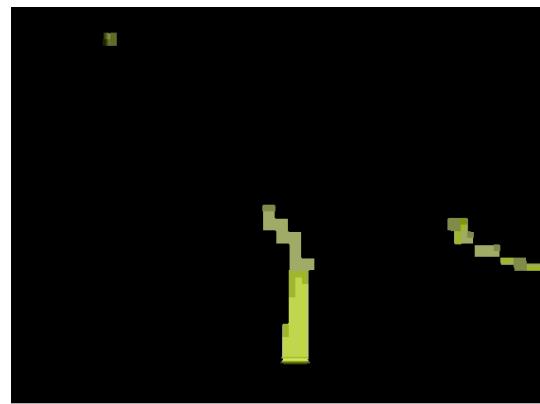
Cuadro 5.3: Éxito al ejecutar follower.py

Sorprendentemente, la compresión JPEG<sub>1</sub> también consigue terminar el circuito. A pesar de la baja calidad de imagen que ofrece tras ejecutarse la compresión. En la figura [5.4](#) puede apreciarse los problemas que pueden surgir al utilizar una imagen de tan baja calidad, demasiada información ha sido perdida y, consecuentemente, el resultado carece de detalle.

Los demás medios no han conseguido terminar el circuito, saliéndose rápidamente de este y siendo inmovilizados al no reconocer el color amarillo en las imágenes. Podríamos remarcar que, a diferencia de los dos casos exitosos, los demás métodos presentan una mayor latencia media. Es decir, dado que el tiempo entre que llega una imagen y la siguiente es tan extenso, el robot continuará moviéndose con el mismo vector velocidad. Cuando llegue la siguiente imagen, ya se habrá descarrilado fuera del circuito, no permitiéndole continuar. El máximo valor de latencia media con el que el TurtleBot ha conseguido terminar el circuito, ha sido aproximadamente 27 ms.



(a) Imagen sin filtrar



(b) Filtro sobre la imagen original

Figura 5.4: Imágenes obtenidas al ejecutar frameAnalyzer.py con compresión JPEG<sub>1</sub>

## 5.5. Conclusiones

Como ya fue comentado en secciones anteriores, las conexiones TCP entre nodos de ROS en redes inalámbricas puede llegar a suponer grandes retrasos en el envío de mensajes. Esto, sumado al gran tamaño de los paquetes que contienen imágenes, resultará en una comunicación insatisfactoria entre TurtleBot y servidor. Hemos probado diferentes tipos de compresión y reducido la frecuencia de publicación para conseguir un mejor rendimiento, no obteniendo muy buenos resultados mediante la última de estas dos opciones. Pero sí consiguiendo un excelente rendimiento a través de las imágenes JPEG<sub>80</sub>. Podríamos concluir con que, los protocolos de comunicación estándar de ROS no están optimizados para redes inalámbricas, donde aspectos como un ancho de banda reducido (sobre todo en redes *multi hop*) y una mayor latencia y jitter deben de tenerse en consideración.

---

## 6. Conclusiones y trabajo futuro

---

En este trabajo hemos aprendido a utilizar un framework con una lenta curva de aprendizaje como se trata de ROS y el manejo de entornos simulados con Gazebo. Tras estudiar las herramientas y funcionalidades que nos ofrece TurtleBot, hemos conseguido implementar nodos de ROS, tanto para analizar las comunicaciones entre robot y servidor, como para hacer programas funcionales como el seguidor de líneas y así ampliar nuestros conocimientos en visión artificial y OpenCV.

Con dichos nodos, hemos evaluado las comunicaciones entre el robot TurtleBot y un servidor local, tanto en un entorno simulado como en uno real. Además, hemos observado las complicaciones que surgen al enviar imágenes en una red inalámbrica y, propuesto soluciones como: la compresión de los frames antes de su transmisión y la bajada de frecuencia de publicación de imágenes por el robot. Tras recoger los datos de la evaluación experimental hemos examinado a más bajo nivel los resultados obtenidos, aportando nuestras propias conclusiones.

Para finalizar, recogemos futuras mejoras para este trabajo que no dieron tiempo a desarrollar:

1. Transformar la clase ROS\_Timer en un paquete de ROS disponible para la comunidad.
2. Empleo de UDP como protocolo de transporte para la transmisión de imágenes. Por defecto, se utiliza TCP. Pese a que ROS no proporciona ningún control para que los paquetes lleguen en orden, esto no supondría mucha dificultad debido a la naturaleza del problema.
3. Hacer uso de un servidor en la nube como ordenador que procesa las imágenes. Así podríamos probar la posibilidad de computación en la nube con robots y evaluar los problemas que supone tener un ancho de banda limitado en redes multi hop.
4. Implementar un nodo o servicio de ROS para acordar una frecuencia de publicación de imágenes entre TurtleBot y así alcanzar el mínimo porcentaje de frames perdidos en su envío.
5. Implementar un nodo suscriptor en C++ para evaluar las nubes de puntos ofrecidas por el sensor Kinect y crear una aplicación funcional con ellas. Debido a su gran tamaño, deberían ser comprimidas antes de su envío.

---

## 7. Bibliografía

---

- [1] Open Robotics. ROS.org — Powering the world’s robots, 2021. URL <https://www.ros.org>.
- [2] Open Source Robotics Foundation. Turtlebot.com, 2013. URL <https://www.turtlebot.com/>.
- [3] OSRF. Gazebosim.org, 2019. URL <http://gazebosim.org/>.
- [4] OpenCV team. Opencv, 2021. URL <https://opencv.org/>.
- [5] GitHub. Cvbridge, 2021. URL [https://github.com/ros-perception/vision\\_opencv](https://github.com/ros-perception/vision_opencv).
- [6] Morgan Quigley, Brian Gerkey, and William D Smart. *Programming Robots with ROS: a practical introduction to the Robot Operating System.* .”Reilly Media, Inc.”, 2015.

---

# A. Anexo I: Manual de instalación del entorno

---

En este manual de instalación se procederá a describir cómo instalar las diferentes herramientas para conseguir el mismo entorno que fue utilizado durante el proyecto. Partiremos de una instalación limpia de Ubuntu 16.04.7 LTS (Xenial Xerus)<sup>1</sup> pues fue la utilizada para este proyecto y es la única compatible con la versión de ROS utilizada.

1. Configuramos el ordenador para que acepte software que provenga de packages.ros.org:

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu\n$lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

2. Establecemos las claves correspondientes:

```
$ sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

3. Nos aseguramos de que el índice de nuestros paquetes Debian esté actualizado:

```
$ sudo apt-get update
```

4. Instalamos la versión completa de ROS Kinetic:

```
$ sudo apt-get install ros-kinetic-desktop-full
```

5. Procedemos ahora a instalar los paquetes de ROS para TurtleBot:

```
$ sudo apt-get install ros-kinetic-turtlebot ros-kinetic-turtlebot-apps ros-kinetic-turtlebot-interactions ros-kinetic-turtlebot-simulator ros-kinetic-kobuki-ftdi ros-kinetic-rocon-remocon ros-kinetic-rocon-qt-library ros-kinetic-ar-track-alvar-msgs
```

6. Por comodidad, añadiremos la siguiente linea a nuestro archivo bash para que se inicien automáticamente las variables de ROS:

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

7. Si el lector quisiera utilizar las diferentes herramientas que ROS ofrece y poder compilar los diferentes componentes de los paquetes de ROS, deberá de ejecutar las siguientes líneas:

```
$ sudo apt install python-rosdep
```

---

<sup>1</sup>Fuente: <https://releases.ubuntu.com/>

```
$ sudo rosdep init  
$ rosdep update
```

8. Se procederá ahora a clonar el repositorio del proyecto nombrado **turtlebot-vision**<sup>2</sup>. Se trata de un repositorio alojado en GitHub que contiene los siguientes directorios:

- config: contiene un script de configuración llamado scriptVariables.sh si no se llegase a entender la siguiente sección.
- stats: carpeta donde se van a guardar las estadísticas generadas mediante los scripts.
- vision\_ws: espacio de trabajo o *workspace* de ROS, contiene los scripts del proyecto.
- worlds: directorio que contiene los ficheros .world utilizados para las simulaciones con Gazebo.

Para clonar este repositorio necesitaremos la herramienta Git, si no contásemos con ella, la instalamos mediante:

```
$ sudo apt install git
```

Tras ello, podremos clonar el repositorio ejecutando el comando a continuación:

```
$ git clone https://github.com/wuaho/turtlebot-vision.git
```

Si se desea ejecutar alguno de los scripts descargados del repositorio, tendremos antes que darle permisos de ejecución, esto lo podemos hacer situándonos en el directorio donde se encuentre el script y ejecutando:

```
$ sudo chmod +x NOMBRE_DEL_SCRIPT
```

9. Ejecutaremos ahora los comandos que permiten la comunicación entre robot y servidor. Entendemos también por robot a un ordenador ejecutando una simulación del TurtleBot con Gazebo. La cadena IP\_DE\_ESTA\_MAQUINA deberá ser sustituida por la dirección IP que este utilizando dicha maquina. Los comandos son los siguientes:

```
$ echo export ROS_MASTER_URI=http://IP_DE_ESTA_MAQUINA:11311 >> ~/.bashrc  
$ echo export ROS_HOSTNAME=IP_DE_ESTA_MAQUINA >> ~/.bashrc
```

10. Por último, solo nos quedará ejecutar comandos que inicien variables de entorno en la bash de Linux. Deberemos ejecutar finalmente:

<sup>2</sup>Fuente: <https://github.com/wuaho/turtlebot-vision>

```
$ echo export DISPLAY=:0 >> ~/.bashrc
$ echo source $HOME/turtlebot-vision/vision_ws/devel/setup.bash >> ~/.bashrc
$ echo export TURTLEBOT_3D_SENSOR=kinect >> ~/.bashrc
```

Tras esto, habremos finalizado adecuadamente la instalación del entorno. Seremos capaces a partir de ahora de hacer uso de todas las herramientas y tecnologías mencionadas en el proyecto.

---

## B. Anexo II: Scripts dentro de turtlebot\_vision

---

Extracto de código B.1: frameAnalyzer.py

```
1 #!/usr/bin/env python
2 # encoding: utf-8
3
4 import rospy
5 from sensor_msgs.msg import Image
6 from cv_bridge import CvBridge
7 import cv2
8 import csv
9 import numpy as np
10 from ROS_Timer import ROS_Timer
11
12 ros_timer = ROS_Timer()
13
14 class Camera_Frames_Analysis:
15
16     def __init__(self):
17         #Start of the timer
18         ros_timer.start()
19         self.image_sub = rospy.Subscriber('camera/rgb/
20             image_raw/', Image, self.image_callback)
21
22     def image_callback(self, msg):
23         #The first thing we do is to tell ros_timer to set
24             #the timestamp for the receipt time
25         ros_timer.set_receiptStamp()
26
27     try:
28         #Making use of bridge, we transform the msg to
29             #something that OpenCV can handle
30         cv2_img = CvBridge().imgmsg_to_cv2(msg,
31             desired_encoding='passthrough')
32
33     except CvBridgeError as e:
34         #Code executed when an error is encountered
35         print(e)
36
37     else:
38         #Vision processing code that we want to execute
39         #RGB to HSV and green filter
```

```

36         filtered_with_green,green_mask = bgr_to_hsv(
37             cv2_img)
38
39     # We tell ros_timer to get the post processing stamp
40     # right after we manipulate the frame
41     ros_timer.set_postProcessingStamp()
42
43     cv2.imshow('filter',green_mask)
44     cv2.imshow('filtered',filtered_with_green)
45     cv2.imshow('original',cv2_img)
46     cv2.waitKey(1)
47
48     #Getting the timestamp stored in the message sent by
49     # the camera and make ros_timer handle it
50     # and also getting the number sequence inside the
51     # image taken by the camera
52     ros_timer.set_containedMessageInfo(msg)
53
54     #We add the information we gathered to the data
55     # container
56     ros_timer.add_frame_to_data()
57
58 def bgr_to_hsv(imagen):
59     # Convert BGR to HSV
60     hsv = cv2.cvtColor(imagen, cv2.COLOR_BGR2HSV)
61     # define range of green color in HSV
62     lower_green = np.array([36,0,0])
63     upper_green = np.array([86,255,255])
64     # Threshold the HSV image to get only green colors
65     mask = cv2.inRange(hsv, lower_green, upper_green)
66     # Bitwise-AND mask and original image
67     res = cv2.bitwise_and(imagen,imagen, mask= mask)
68
69     return res,mask
70
71 def main():
72
73     #Our node, camera_frames_analysis, is started
74     rospy.init_node('camera_frames_analysis')
75     Analyzer=Camera_Frames_Analysis()
76

```

```
77  
78 #Rospy.spin will detect when the node is stopped  
79 rospy.spin()  
80  
81 #If you want it to stop after a duration of time use this  
     line instead  
82 #rospy.sleep(300)  
83  
84 #Stop of the timer  
85 ros_timer.stop()  
86  
87 #We write the data stored in a csv file  
88 ros_timer.write_data()  
89 ros_timer.write_usefulInfo()  
90  
91  
92 if __name__ == "__main__":  
93     main()
```

Extracto de código B.2: ROS\_Timer.py

```
1 #!/usr/bin/env python
2 # encoding: utf-8
3 import rospy
4 import time
5 import csv
6 from datetime import datetime
7
8
9 class ROS_Timer:
10
11     def __init__(self):
12         self.startTime = None
13         self.stopTime = None
14
15         self.firstImage = True
16         self.imageCounter = 0
17         self.frameSeqNumber = None
18         self.expectedFrameSeqNumber = None
19
20
21         self.cameraStamp = None
22         self.receiptStamp = None
23         self.postProcessingStamp = None
24
25
26         self.lostPackets = 0
27         self.totalLostPackets = 0
28         self.sumImageSize = float(0)
29         self.networkDelay = float(0)
30         self.totalNetworkDelay = float(0)
31         self.maxTime = float('-inf')
32         self.minTime = float('inf')
33
34
35     ###DATA STRUCTURE: LIST OF TUPLES. Each tuple: (
36         frameSeqNumber, cameraStamp, receiptStamp,
37         postProcessingStamp,
38         # lostPackets, network delay, processing delay, total
39         # delay)
40         self.storedData = []
41
42     def __increaseCounter(self):
43         self.imageCounter += 1
44
45     def __addPacketsLost(self):
46         """Sets the number of packets lost before the last
```

```

        one and sets the counter for the next expected
        image sequence number"""
44    if self.expectedFrameSeqNumber is not None:
45        self.lostPackets = self.frameSeqNumber - self.
46            expectedFrameSeqNumber
47        self.totalLostPackets += self.lostPackets
48
49    self.expectedFrameSeqNumber = self.frameSeqNumber + 1
50
51
52    def __calculateDelays(self):
53        self.networkDelay = float(self.receiptStamp -self.
54            cameraStamp)
55        self.totalNetworkDelay += self.networkDelay
56
57        if(self.networkDelay > self.maxTime):
58            self.maxTime = self.networkDelay
59
60        if(self.networkDelay < self.minTime and self.
61            networkDelay >0):
62            self.minTime = self.networkDelay
63
64        processingDelay = float(self.postProcessingStamp -
65            self.receiptStamp)
66        totalDelay = float(self.postProcessingStamp - self.
67            cameraStamp)
68
69        return processingDelay, totalDelay
70
71
72    def set_cameraStamp(self,timestamp):
73        """Takes the timestamp when the robot took the frame
74        and saves it as a float"""
75
76        self.cameraStamp = float(timestamp.secs + timestamp.
77            nsecs *10**-9)
78
79    def set_receiptStamp(self):
80        """Call this function after the frame was received
81            by the subscriber to set the receipt stamp"""
82        self.receiptStamp = rospy.get_time()
83
84    def set_postProcessingStamp(self):
85        """Call this function after the frame was processed
86            to set the post processing frame stamp"""
87        self.postProcessingStamp = rospy.get_time()
88
89    def set_frameSeqNumber(self,numberOfframe):
90        """Sets the sequence number for the frame took by the

```

```

    camera """
80     self.frameSeqNumber = numberOfframe
81
82     def set_imageSize(self, array_size_in_bytes):
83         """Sets the size of the image contained in the packet
84             """
85         self.sumImageSize += float(array_size_in_bytes)
86
87     def set_containedMessageInfo(self, msg):
88         """Calls the functions set_cameraStamp and
89             set_frameSeqNumber which save important
90                 information about when
91                 was the image taken by the camera and the sequence
92                 number of the frame"""
93         self.set_cameraStamp(msg.header.stamp)
94         self.set_frameSeqNumber(msg.header.seq)
95         self.set_imageSize(len(msg.data))
96
97     def get_totalReceivedImages(self):
98         return self.imageCounter
99
100    def get_totalLostImages(self):
101        return self.totalLostPackets
102
103    def get_percentageOfLostImages(self):
104        suma = float(self.get_totalReceivedImages() + self.
105                    get_totalLostImages())
106        pctg = self.get_totalLostImages() / suma *100
107
108        return pctg
109
110    def get_minTime(self):
111        mintime = (self.minTime * 1000)
112        return mintime
113
114    def get_meanTime(self):
115        mean = ((self.totalNetworkDelay / self.
116                  get_totalReceivedImages()) * 1000)
117        return mean
118
119
120    def get_maxTime(self):
121        maxtime = (self.maxTime * 1000)
122        return maxtime
123
124    def get_runtime(self):
125        """Returns the number of seconds since the timer was
126            started to when it was stopped
127            """
128
129        return self.stopTime - self.startTime
130
131    def get_time_passed(self):
132        """Returns the number of seconds since the timer was

```

```

        started to the actual time
"""
119
120     return time.time() - self.stopTime
121
122 def get_receiptStamp(self):
123     """Call this function after the frame was received by
124         the subscriber to set the receipt stamp"""
125     return self.receiptStamp
126
127 def calculate_fps(self):
128     """Returns the fps that has been achieved after
129         stopping the timer"""
130     fps = self.get_totalReceivedImages() / self.
131         get_runtime()
132     return fps
133 def calculate_jitter(self):
134     """Returns the jitter that has been achieved after
135         stopping the timer. That is the mean difference of
136         delay
137         between consecutive packets"""
138     diff = float(0)
139     for i in range(len(self.storedData)-1):
140         diff += abs(self.storedData[i][5] - self.
141             storedData[i+1][5])
142
143     jitter = diff / ( len(self.storedData) -1)
144     jitter = (jitter * 1000)
145     return jitter
146
147 def calculate_bw(self):
148     """Returns the average bandwidth consumption in Mb/s
149         """
150     sum_of_bytes = self.sumImageSize
151     average_bytes_per_image = sum_of_bytes / self.
152         get_totalReceivedImages()
153     FPS = self.calculate_fps()
154     bw = (((average_bytes_per_image / 1024) / 1024) * FPS
155         ) * 8
156     return bw
157
158 def print_average_bytes_per_image(self):
159     ###BORRAR MAS ADELANTE, ES SOLO UN METODO AUXILIAR
160     sum_of_bytes = self.sumImageSize
161     average_MB_bytes_per_image = sum_of_bytes / self.
162         get_totalReceivedImages() /1024/1024
163     print(average_MB_bytes_per_image, "MB")
164

```

```

155
156     def start(self):
157         """Starts the timer"""
158         self.startTime = time.time()
159
160     def stop(self):
161         """Stops the timer"""
162         self.stopTime = time.time()
163
164
165     def add_frame_to_data(self):
166         """Call this function before exiting finishing the
167             code for each image """
168         if(not(self.firstImage)):
169             self.__increaseCounter()
170             self.__addPacketsLost()
171
172
173
174         self.storedData.append((self.frameSeqNumber, self.
175             cameraStamp, self.receiptStamp, self.
176             postProcessingStamp,
177                     self.lostPackets, self.
178                     networkDelay,
179                     processingDelay,
180                     totalDelay))
181
182     else:
183         self.firstImage = False
184
185     def write_data(self):
186         """Writes a file in the same directory named "
187             data_collected.csv" that contains all the appended
188             information
189
190         """
191
192         ###DATA STRUCTURE: LIST OF TUPLES. Each tuple: (
193             frameSeqNumber, cameraStamp, receiptStamp,
194             postProcessingStamp,
195             # lostPackets, network delay,processing delay, total
196             # delay)
197
198
199         with open('stats/data_collected.csv', 'w') as csvfile:
200             csvfile.write("frameSeqNumber, cameraStamp,
201                 receiptStamp, postProcessingStamp, lostPackets,
202                 network delay, processing delay, total delay")
203             for stamp in self.storedData:

```

```

188         line = ",".join(map(str,stamp))
189         csvfile.write("\n")
190         csvfile.write(line)
191
192     def write_usefulInfo(self):
193         """Writes a file in the same directory named 'useful_info.txt' that contains some performance information about the simulation"""
194         with open('stats/useful_info.txt','w') as txtfile:
195             txtfile.write("TOTAL TIME SPENT ON SIMULATION: "+str(self.get_runtime())+" s\n")
196             txtfile.write("TOTAL NUMBER OF IMAGES RECEIVED: "+str(self.get_totalReceivedImages())+" images\n")
197             txtfile.write("TOTAL NUMBER OF IMAGES LOST: "+str(self.get_totalLostImages())+" images\n")
198             txtfile.write("% OF IMAGES LOST: "+str(self.get_percentageOfLostImages())+" %\n")
199             txtfile.write("MEAN LATENCY ACHIEVED: "+ str(self.get_meanTime())+" ms\n")
200             txtfile.write("MAX LATENCY ACHIEVED: "+ str(self.get_maxTime())+" ms\n")
201             txtfile.write("MIN LATENCY ACHIEVED: "+ str(self.get_minTime())+" ms\n")
202             txtfile.write("AVERAGE JITTER: "+ str(self.calculate_jitter())+" ms\n")
203             txtfile.write("APROX FREQUENCY: "+ str(self.calculate_fps())+" FPS\n")
204             txtfile.write("AVERAGE BANDWIDTH CONSUMPTION: " + str(self.calculate_bw()) +" Mb/s")
205
206
207
208
209 if __name__ == "__main__":
210     print("Start")

```

Extracto de código B.3: follower.py

```
1 #!/usr/bin/env python
2 # encoding: utf-8
3
4 import rospy, cv2, csv
5 from sensor_msgs.msg import Image, CompressedImage
6 from cv_bridge import CvBridge
7 from geometry_msgs.msg import Twist
8 import numpy as np
9
10
11 class Follower:
12
13     def __init__(self):
14         #Raw image version
15         #self.image_sub = rospy.Subscriber('camera/rgb/
16             #image_raw/', Image, self.image_callback)
17
18         #Compressed image version
19         self.image_sub = rospy.Subscriber('camera/rgb/
20             #image_raw/compressed', CompressedImage, self.
21             #image_callback)
22         #Half res version
23         #self.image_sub = rospy.Subscriber('image_half_res',
24             Image, self.image_callback)
25
26         self.cmd_vel_pub = rospy.Publisher('cmd_vel_mux/input
27             /teleop', Twist, queue_size=0)
28         self.twist = Twist()
29
30     def image_callback(self, msg):
31         #Making use of bridge, we transform the msg to
32             #something that OpenCV can handle
33             #image = CvBridge().imgmsg_to_cv2(msg,
34                 desired_encoding='passthrough')
35
36             #For compressed images we would use this one:
37             image = CvBridge().compressed_imgmsg_to_cv2(msg,
38                 desired_encoding='passthrough')
39
40
41             #Vision processing code that we want to execute
42             #RGB to HSV and green filter
43             filtered_with_yellow,yellow_mask = bgr_to_hsv(image)
44
45
46             #Since we dont need the whole line on the picture for
47                 #the robot, we will slice everything that is not
```

```

    in the
38   #20-row portion of the image. Doing this, we get the
39   part that is one-meter distance from the robot.
40   h, w, d = image.shape
41   search_top = 3*h/4
42   search_bot = search_top + 20

43   yellow_mask[0:search_top, 0:w] = 0
44   yellow_mask[search_bot:h, 0:w] = 0

45
46   #Here we calculate the centroid or center of mass of
47   the remaining masked picture
48   M = cv2.moments(yellow_mask)
49   if( M['m00'] > 0 ):
50       cx = int(M['m10']/M['m00'])
51       cy = int(M['m01']/M['m00'])
52       cv2.circle(image, (cx, cy), 20, (0,0,255), -1)
53       #Movement of the robot, first line calculates the
54       #error between the center column of the image
55       #and the center
56       #of the line
57       err = cx - w/2
58       self.twist.linear.x = 0.6
59       self.twist.angular.z = -float(err) / 100
60       self.cmd_vel_pub.publish(self.twist)

61
62
63   cv2.imshow('filter',yellow_mask)
64   cv2.imshow('filtered',filtered_with_yellow)
65   cv2.imshow('original',image)
66   cv2.waitKey(1)

67
68 def bgr_to_hsv(imagen):
69     # Convert BGR to HSV
70     hsv = cv2.cvtColor(imagen, cv2.COLOR_BGR2HSV)
71     # define range of yellow color in HSV
72     lower_yellow = np.array([22, 93, 0])
73     upper_yellow = np.array([45, 255, 255])
74     # Threshold the HSV image to get only yellow colors
75     mask = cv2.inRange(hsv, lower_yellow, upper_yellow)
76     # Bitwise-AND mask and original image
77     res = cv2.bitwise_and(imagen,imagen, mask= mask)

78
79 return res,mask

```

```
80
81
82 def main():
83
84     rospy.init_node('follower')
85     follower=Follower()
86
87     #Rospy.spin will detect when the node is stopped
88     rospy.spin()
89
90
91 if __name__ == "__main__":
92     main()
```

Extracto de código B.4: imageRepubliser.py

```
1 #!/usr/bin/env python
2 # encoding: utf-8
3
4
5 import sys
6 import rospy
7 import cv2
8 from sensor_msgs.msg import Image
9 from cv_bridge import CvBridge, CvBridgeError
10
11 class image_converter:
12     def __init__(self):
13         self.image_pub = rospy.Publisher("image_half_res",
14                                         Image, queue_size=0)
15
16         self.bridge = CvBridge()
17         self.image_sub = rospy.Subscriber('camera/rgb/
18                                         image_raw/', Image, self.callback)
19
20     def callback(self, data):
21
22         try:
23             cv_image = self.bridge.imgmsg_to_cv2(data,
24                                                 desired_encoding='passthrough')
25         except CvBridgeError as e:
26             print(e)
27
28         #We make the size a 1/4 of the original one
29         cv_modified = cv2.resize(cv_image, (0,0), fx = 0.5,
30                                fy = 0.5)
31
32         try:
33             ros_image = self.bridge.cv2_to_imgmsg(cv_modified
34                                                 , "bgr8")
35             #The stamp inside the header is not saved after
36             #converting the image so we assign the same one
37             #that
38             #original message had
39             ros_image.header.stamp = data.header.stamp
40             self.image_pub.publish(ros_image)
41         except CvBridgeError as e:
42             print(e)
43
44     def main(args):
45         rospy.init_node("image_converter")
46         ic = image_converter()
```

```
40     try:
41         rospy.spin()
42
43     except KeyboardInterrupt:
44         print("Shutting down")
45
46 if __name__ == '__main__':
47     main(sys.argv)
```