ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

Facultad de Ingeniería Eléctrica y Computación

"ROBOT OPERATING SYSTEM (ROS) COMO PLATAFORMA PARA EXTENDER LAS CAPACIDADES DE LEGOS NXT"

INFORME DE MATERIA DE GRADUACIÓN

Previa a la obtención del título de

INGENIERO EN TELEMÁTICA

Presentada por

María Fernanda Utreras Abad

Diana Carolina Decimavilla Alarcón

Guayaquil Ecuador

2013

AGRADECIMIENTO

A Dios, por estar a mi lado en cada momento y ayudarme a superar cada obstáculo, a mi madre por su apoyo y paciencia, a Héctor y Jonathan por su valiosa ayuda en el proyecto, a Diego y todas las personas que estuvieron pendientes de nuestros avances del Proyecto.

María Fernanda Utreras Abad

A Dios, por darme la fuerza necesaria para lograr dar este gran paso en mi vida; a mi familia por la paciencia y apoyo; a mis amigos por los momentos de felicidad en especial a Ivette, Eduardo, Andrés y Mafer. Por último a quien fue parte fundamental en mi vida, Gustavo por toda su paciencia, apoyo y amor para concluir este proyecto.

Diana Decimavilla Alarcón.

DEDICATORIA

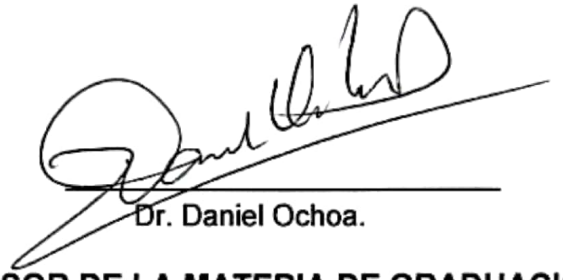
A mi Familia, quienes desearon este logro tanto como yo, a las personas que fueron participes de mi crecimiento personal y profesional, que con su apoyo y consejos, me ayudaron a terminar con esta etapa de mi vida.

María Fernanda Utreras Abad.

A todas las personas que estuvieron pendientes de mi desarrollo académico y personal, de los que cada día recibía sabias palabras para concluir con éxitos esta etapa de mi vida.

Diana Decimavilla Alarcón.

TRIBUNAL DE SUSTENTACIÓN



Dr. Daniel Ochoa.

PROFESOR DE LA MATERIA DE GRADUACIÓN

MSc.. Carlos Valdivieso

PROFESOR DELEGADO POR LA UNIDAD ACADÉMICA

DECLARACION EXPRESA

"La responsabilidad del contenido de este Informe de Graduación, nos corresponde exclusivamente; y el patrimonio intelectual de la misma, a la **Escuela Superior Politécnica del Litoral**"

(Reglamento de Graduación de la ESPOL)

Ma. Fernanda Utreras Abad.

Diana Decimavilla Alarcón

RESUMEN

La Industria robótica está avanzando cada vez más rápido y hoy en día con los cambios en los procesos de producción y la tendencia a hacer que todos los procesos automatizados, requiere que el manejo de los robots sea mucho más fácil, con mayor flexibilidad y rapidez y que la exactitud vaya aumentando. Para esto fue creado ROS. Se trata de un código abierto que proporciona los servicios que se esperan de un sistema operativo, incluyendo hardware abstracción, el control de dispositivos de bajo nivel, paso de mensajes entre procesos y gestión de paquetes para nuestro robot.

Hoy en día el uso de un Láser Scanner es fundamental para cualquier tipo de empresa que desee verificar la estructura de sus productos. Características como el tamaño y la forma son fundamentales para muchas empresas. De ahí el objetivo de este proyecto que es demostrar que podemos extender las capacidades de los Legos NXT para así dotar de funcionalidades tan específicas como la descrita en este Informe de la materia de graduación.

ÍNDICE GENERAL

Pag.

ANEXOS

ABREVIATURAS

ROS: Robot Operating System

O1v1: Objeto 1, vuelta 1

O1v2: Objeto 1, vuelta 2

O1v3: Objeto 1, vuelta 3

O1v4: Objeto 1, vuelta 4

O2v1: Objeto 2, vuelta 1

O2v2: Objeto 2, vuelta 2

O2v3: Objeto 2, vuelta 3

O2v4: Objeto 2, vuelta 4

GLOSARIO

API: Interfaz de programación de aplicaciones (API) especifica cómo algunos componentes de software deben interactuar entre sí.

En la práctica, en la mayoría de los casos una API es una biblioteca que por lo general incluye la especificación para las rutinas, estructuras de datos, clases de objetos y variables.

Framework: Un framework o infraestructura digital, es una estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de *software* concretos, que puede servir de base para la organización y desarrollo de *software*. Típicamente, puede incluir soporte de programas, bibliotecas, y un lenguaje interpretado, entre otras herramientas, para así ayudar a desarrollar y unir los diferentes componentes de un proyecto. En robótica existen algunos llamados: YARP, ORCOS, MOOS, CARMEN, entre otros.

Launch: Los launch son archivos de configuración que me permiten activar un conjunto de nodos, servicios, topics y establecer sus parámetros.

Manifiesto (manifest): Un manifiesto es una descripción de un paquete. Su función más importante es definir las dependencias entre paquetes.

Master (maestro): Servicio de nombres y de registro para todos los nodos de ROS, es decir, ayuda a los nodos de encontrarse unos a otros.

Mensajes (MSG): Tipo de datos que ROS utiliza para la suscripción o la publicación de un tema.

Microkernel: Es un tipo de núcleo de un sistema operativo que provee un conjunto de primitivas o llamadas mínimas al sistema para implementar servicios básicos como espacios de direcciones, comunicación entre procesos y planificación básica. Todos los otros servicios (gestión de memoria, sistema de archivos, operaciones de E/S, etc.), que en general son provistos por el núcleo, se ejecutan como procesos servidores en espacio de usuario.

Nodos: Un nodo es un archivo ejecutable que utiliza ROS para comunicarse con otros nodos.

Paquetes: Los paquetes son el nivel más bajo de la organización de software ROS. Pueden contener cualquier cosa: las bibliotecas, herramientas, ejecutables, etc. Un paquete es un directorio con un archivo manifest.xml.

Parameter (parametros): Los parámetros son llamados usando los nombres convencionales en ROS. Esto significa que los parámetros de ROS tienen una jerarquía que coincide con los espacios de nombres utilizados

para temas y nodos. Esta jerarquía está destinado a proteger a los nombres de los parámetros de colisión. El esquema jerárquico también permite que los parámetros para acceder de forma individual o como un árbol.

Parameter Server (Servidos de Parametros): Es un diccionario común, multi-variable que se puede acceder a través de las API de red. Los Nodos utilizan este servidor para almacenar y recuperar los parámetros en tiempo de ejecución. Dado que no está diseñado para un alto rendimiento, que es el más utilizado para los datos estáticos, no binarios, tales como los parámetros de configuración.

Pila Manifiesto (Stack Manifest): Describe a la pila e informa de sus dependencias.

Pilas (Stacks) : Las pilas son colecciones de paquetes que proveen funcionalidad, tal como una pila de navegación o una pila de manipulación. Forman una biblioteca de más alto nivel. Una pila es un directorio con un archivo stack.xml.

Recursos: En ROS un recurso puede ser un paquete, pila o nodo

roscore: Es una colección de nodos y programas que son pre-requisitos de un sistema basado en ROS. Necesariamente debe estar ejecutándose para que se comuniquen los nodos entre ellos.

roscpp: Biblioteca cliente de C++.

roslaunch: roslaunch es una herramienta de fácil enlance para nodos ROSde forma local o remota a través de SSH, así como establecer los parámetros en el servidor de parámetros. Cuenta con opciones para reaparecer de forma automática los procesos que ya han muerto.

rosmaster: implemente el ROS Master. La mayoría de los programas no tendrán que interactuar con este paquete directamente. El rosmaster se ejecuta automáticamente cada vez que roscore se ejecuta.

rosnode: muestra información sobre los nodos ROS que se están ejecutando en ese momento.

rosout: Este nodo siempre se está ejecutando, ya que recoge y registra la salida de depuración nodos. En ROS es equivalente a stdout / stderr.

rospy: Biblioteca cliente Python.

rosparam: contiene la herramienta de línea de comandos permite almacenar y manipular datos de diferentes tipos en el servidor de parámetros de ROS mediante archivos YAML codificados. Se puede invocar dentro de un roslaunch archivo.

Services (SRV): Los servicios me permiten enviar preguntas y recibir respuestas entre nodos, usando el programa rosservice. Cada tipo de topic tiene sus específicos servicios, estos se enlistan con list y se sabe su tipo con type el cual también especifica los parámetros.

SSH: es el nombre de un protocolo y del programa que lo implementa, y sirve para acceder a máquinas remotas a través de una red. Permite manejar por completo la computadora mediante un intérprete de comandos.

Topics (temas): Los topics son el medio por el cual los nodos pueden interactuar entre sí. El tipo de Topic se determina por el tipo de mensaje que se transmite.

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCION

Investigadores de robótica han creado diversos frameworks para poder sobrellevar la complejidad de las aplicaciones que actualmente se usan en centros de estudio y en la industria.

Los robots son complicados en lo que a software se refiere, debido a que en la práctica este software puede ser extenso y complejo requiriendo experiencia en programación. Además que cada robot utiliza un hardware que puede ser diferente.

Robot Operating System (ROS), es un framework de Robotica, con librerías y herramientas para ayudar a los desarrolladores de software a crear aplicaciones para Robots. ROS es similar a frameworks como “Player”, “YARP”, Open RObot Control Software (Orocos), “CARMEN”,

y “Microsoft Robotics Studio”, entre otros. ROS se puede utilizar con otros frameworks, ya ha sido integrado con Open cv, Orocos y Player.

ROS tiene como principal objetivo apoyar la reutilización de código para la robótica, habiendo un intercambio y colaboración de información por parte de los miles de expertos e investigadores alrededor del mundo.

Este documento describe que es ROS poniendo énfasis en una investigación de robótica. Se discutirán los objetivos del diseño de ros, cómo instalarlo, cómo utilizarlo, programarlo y demostrar cómo ROS nos puede ayudar en lo académico, específicamente en el proyecto que se desarrolló para esta Tesis que es el Laser Scanner.

CAPÍTULO 1

1. ESPECIFICACIONES GENERALES DE ROS Y LEGO NXT

En este capítulo se detalla de manera general los diferentes sistemas operativos en los que ROS puede ser instalado. Además de mencionar las plataformas robóticas donde se ha instalado ROS. Y por último se detalla especificaciones generales del LEGO NXT.

1.1. Plataforma de Desarrollo

1.1.1 Sistema operativo

La plataforma escogida para este proyecto es Ubuntu de Linux, debido a que la ROS esta soportada únicamente por esta distribución. Existen otras plataformas en las que se puede trabajar

como: Ubuntu ARM, OS X, Fedora, Gentoo, OpenSuse, Rasphian, Debian, Arch Linux, Windows, pero estas son solo experimentales.

1.2. Conmutación con Plataformas existentes

ROS se ha instalado y utilizado en plataformas Roboticas específicas como los Pioneer, Robotnik Cumbre, Gostai Jazz, Lego Mindstorms NXT. En la actualidad más de 50 Robots usan ROS.

Para este Proyecto, se investigó como programar un LEGO NXT con ROS, desde la configuración de ROS en la PC, el puente entre ROS y el Robot y las aplicaciones que puede tener como el Laser Scanner.

1.3. NXT

El Lego NXT utilizado en el proyecto, se suministro por el Centro de Visión y Robótica (CVR), este viene en KIT.

El kit contiene lo siguiente:

- 619 elementos para crear sus propios robots - LEGO TECHNIC elementos de construcción, engranajes, ruedas, tracks y neumáticos

- 1 NXT micro-ordenador (Brick)- que actúa como el cerebro del robot
- Sensores táctiles - que hace que el robot se sienten
- 1 Sensor ultrasónico - que hace que el robot "ver" - y detectar el movimiento
- 1 Color Sensor - que puede detectar diferentes colores de luz, ajustes y actúa como una lámpara
- 3 servomotores interactivos con sensores de rotación integrados. Ver Figura 1.2.
- 7 cables de conexión para enlazar motores y sensores al NXT
- Guía del usuario – con instrucciones de construcción para su primer robot y una introducción al hardware y software. Como muestra la Figura 1.1.

Figura 1.1 Elementos de construcción

Figura 1.2 Brick y sensores.

CAPÍTULO 2

2. ¿QUÉ ES ROS?

Robot Operating System (ROS) es un conjunto de programa mas no un sistema operativo, que posee diferentes herramientas de desarrollo así como herramientas de inspección y depuración [1]. Permite desarrollar, crear y ejecutar sistemas o aplicaciones robóticas sobre distribuciones del Sistema Operativo GNU-LINUX, contiene paquetes que incluyen librerías para varios tipos de robots. Podemos definir a ROS como un Framework de Desarrollo en Robótica RSF (siglas en ingles).

ROS proporciona los servicios que se esperan de un sistema operativo, como abstracción de hardware de bajo nivel de control del dispositivo, la implementación de la funcionalidad de uso común, paso de mensajes entre procesos y gestión de paquetes. También proporciona herramientas

de visualización y simulación además de bibliotecas para obtener, construir, escribir y ejecutar códigos a través de múltiples computadoras [2]. A diferencia de un Sistema Operativo no tiene Kernel, GUI, Sistemas de Archivos, Drivers.

2.1. Arquitectura de ROS

ROS gráficamente se comporta como una red peer-to-peer de procesos que están acoplados utilizando la infraestructura de comunicación ROS. ROS implementa varios estilos de comunicación, incluyendo el estilo sincrónico RPC de comunicación a través de services (servicios), transmisión asíncrona de datos a través de topics (temas), y el almacenamiento de datos en un Parameter Server (servidor de parámetros) [3]. Esto se explica con mayor detalle más adelante en la sección 2.2.

2.2. Niveles de ROS

ROS consta de tres niveles conceptuales que son: Nivel Gráfico, Nivel de Sistema de Archivos, y Nivel Comunitario. Los cuales son detallados a continuación.

2.2.1. Nivel Gráfico

Los conceptos básicos de este nivel grafico son: los Nodos, Master, mensajes, servicios, topics, servidor de parámetros. Todos estos proporcionan los datos a la gráfica de algunas maneras.

2.2.1.1. Nodos

Los nodos son programas ejecutables dentro de un paquete de ROS. El término nodo es intercambiable con “Módulo de Software”, ya que es un programa de computadora que debe realizar varias tareas para cumplir con una función u objetivo como se muestra en la Figura 2.1. Cada nodo que se ejecuta tiene un nombre único que los identifica del resto del sistema.

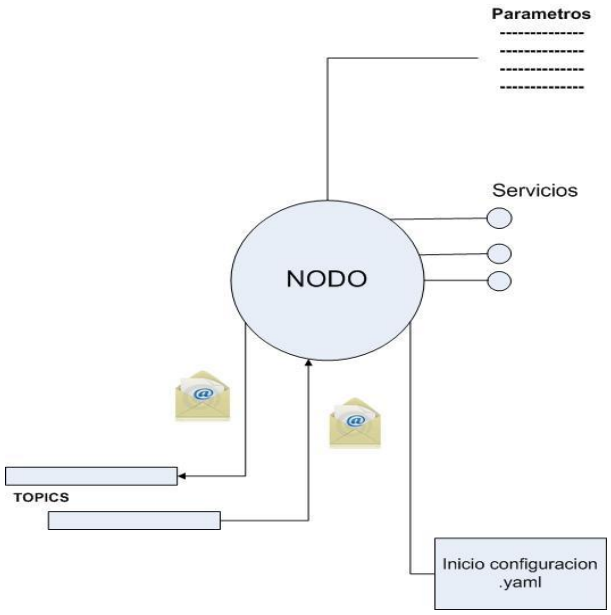


Figura 2.1 Ejemplo Nodo

Un sistema de control puede tener varios nodos, como por ejemplo:

- Un nodo que controla los motores de las ruedas.
- Un nodo que lleva a cabo la localización.
- Un nodo que controla un dispositivo láser.

El uso de nodos de ROS proporciona varios beneficios para el sistema. No hay tolerancia a errores, los nodos son individuales por lo tanto los accidentes o fallos que ocurran son aislados. La complejidad de código se reduce en comparación a los sistemas monolíticos, debido a que no existe un programa principal que invoque el procedimiento de un servicio, en ROS cada nodo se ejecuta de forma independiente.

El mecanismo de comunicaciones de los nodos se basa en que estos pueden publicar o suscribirse a ‘”Topics”, también pueden usar “Servicios”, enviar parámetros conceptos que explicaremos más adelante.

2.2.1.2. Mensajes

Los Nodos utilizan mensajes para comunicarse. Un mensaje se define como un archivo de texto en el que se describen los campos utilizados en el mensaje ROS. Los mensajes son enviados a través de los topics que los enruta con el sistema

Publicación/Suscripción. Los tipos de datos básicos que se usan son flotante, booleano, entero, etc. Como muestra la Figura 2.2

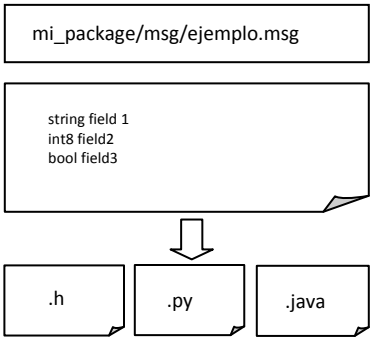


Figura 2.2 Contenido de un Mensaje.

2.2.1.3. Topics

Los Topics son llamados buses, debido a que este es el medio por el cual se intercambian mensajes. El nodo envía un mensaje de publicación en un determinado “Topic”. Cuando un nodo está interesado en un tipo de dato se suscribirá al topic correspondiente. Puede haber varios editores y suscriptores simultáneos para el mismo tópico y además un mismo nodo puede publicar o suscribirse en diferentes tópics. Los Topics están destinados a la comunicación unidireccional.

2.2.1.4. Servicios (SRV)

Los servicios se definen mediante archivos SRV, que se compilan en el código fuente de una biblioteca de ROS.

Los servicios están conformados por dos partes: un mensaje que es el que pregunta y otro mensaje que es el que responde. A diferencia de los topics, un nodo solo puede llamar a un servicio bajo un nombre en específico. La Figura 2.3 resume los dos métodos de comunicación entre nodos:

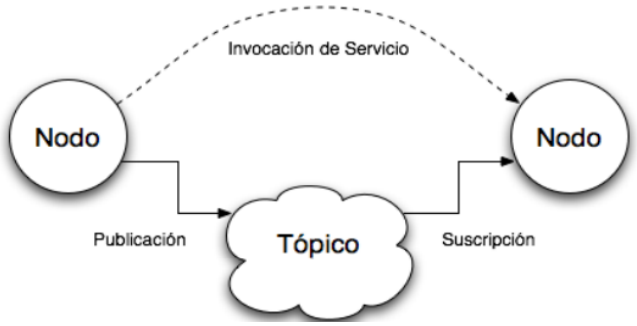


Figura 2.3 Métodos de Comunicación entre Nodos

De la Figura 2.3 se observa las dos maneras de comunicarse entre los nodos: La primera, a través de un topic de comunicación, donde un nodo publica información en el topic la cual es recibida por todo nodo que este suscrito al mismo topic. Por otra parte, es posible que un nodo se

comunique directamente con otro nodo a través de la invocación de un servicio.

2.2.1.5. Master

ROS Máster es el nodo principal de ROS, este proporciona servicios de nombres y registros para el resto de nodos en el sistema ROS. Sin él los otros nodos no podrían encontrarse, ni intercambiar mensajes o solicitar servicios.

Se inicializa a través de la instrucción roscore, es el primer comando que debemos ejecutar cuando usamos ROS, o en su defecto cuando usamos el comando roslaunch el cual se explicará más adelante. Y automáticamente inicializa ROS.

2.2.1.6. Servidor de Parámetros

Es parte del master, permite que los datos se almacenen por clave en una ubicación central. En ROS, existe un servidor de parámetros, llamado “Parameter Server”. Un servidor de parámetro es un diccionario común, multi-variable que se puede acceder a través de las API de red. Los nodos utilizan este servidor para almacenar y recuperar los parámetros en tiempo de ejecución. Dado que no está diseñado para un alto rendimiento, el cual es el más utilizado para los datos estáticos, no binarios, tales como los parámetros de

configuración. Está destinado a ser globalmente visible por lo que las herramientas pueden inspeccionar fácilmente el estado de configuración del sistema y modificarlo si es necesario [4].

2.2.1.7. Ejemplo de Funcionamiento del nodo máster

Un ejemplo sencillo es tener dos nodos: uno llamado Cámara y otro nodo llamado Image_viewer. Lo primero en suceder es: el nodo Cámara notifica que va a publicar imágenes sobre el topic “imagen”. El nodo Cámara publica las imágenes (datos), pero como no se ha configurado un nodo que se suscriba al topic “imagen” no se podrá enviar ningún dato por el momento [5]. Como se hace referencia en la Figura 2.4

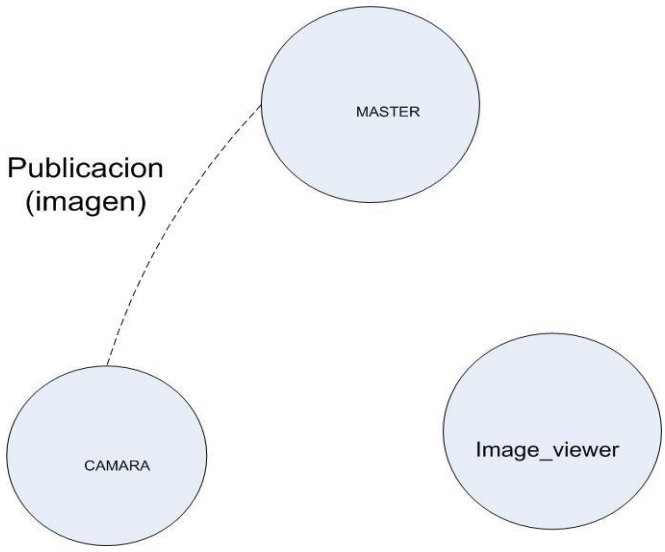


Figura 2.4 Nodo Cámara publica datos sobre el topic “imagen”

Luego se configura al nodo Image_viewer para que se suscriba al tópic “imágen” para lograr recibir las imágenes (datos). Como muestra la Figura 2.5

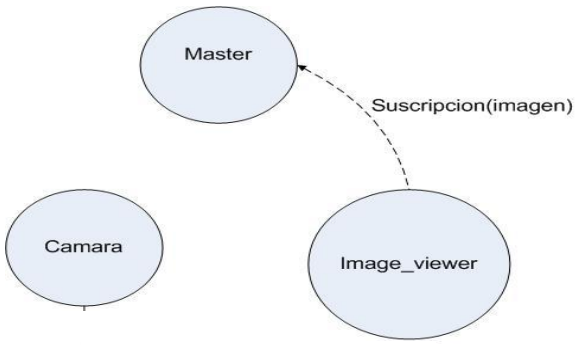


Figura 2.5 Nodo Image_viewer se suscribe al topic “imagen”

Por lo tanto el topic “imagen” tiene un publicador y un suscriptor. Para ilustración del ejemplo el topic “imagen” se lo representó con un rectángulo. Por último el nodo Máster notifica a los otros dos nodos que existe información para que se pueda iniciar la transferencia de imágenes entre sí. Como se ilustra en la Figura 2.6

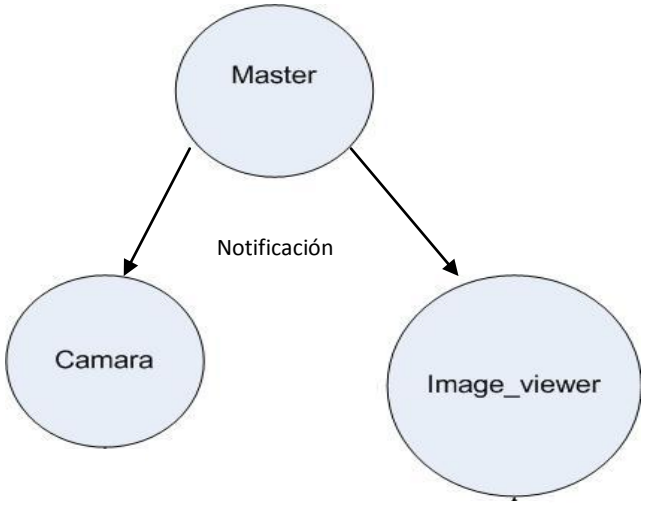


Figura 2.6 Topic "imagen" con su respectivo nodo publicador y suscriptor

2.2.2. Nivel de sistema de archivos

Los conceptos de nivel de sistema de archivos de ROS son recursos que se encuentran en el disco [6], herramientas para la gestión del código fuente, instrucciones de compilación y definiciones de mensajes. ROS es implementado de forma nativa en cada lenguaje y rápidamente define mensajes en un formato de lenguaje independiente como se muestra en la Figura 2.7.

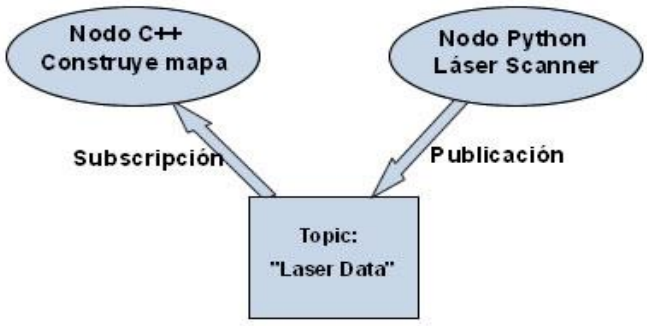


Figura 2.7 Lenguajes de Programación distintos pueden comunicarse en ROS

El nivel de Sistemas de archivos contiene herramientas para construcción de nodos, ejecución de nodos, visualización de la topología de red y supervisión el tráfico de red. Es una gran cantidad de pequeños procesos [7].

2.2.2.1. Recursos de ROS

- **PAQUETES:** Los paquetes son la unidad principal para organizar el software en ROS, un paquete puede contener procesos de ejecución de nodos, también contiene bibliotecas independientes, conjunto de datos, archivos de configuración. Estos están especificados en un archivo **manifest.xml.** El cual proporciona información sobre licencia y

dependencias, así como información específica del lenguaje de programación, tales como opciones del compilador [8].

El manifiesto (manifest.xml) es una especificación mínima sobre el paquete, un papel importante del manifest es la declaración de dependencias y del sistema operativo neutral. Una dependencia significa que incluye archivos de otros paquetes. Cualquier directorio que contiene un archivo manifest.xml se considera que es un paquete. Los archivos de manifiesto más comunes incluyen etiquetas <depend> y <export>, que ayudan a gestionar la instalación y uso de un paquete. La etiqueta <depend> apunta a otro paquete de ROS que se debe instalar.

Figura 2.8 Contenido de un Paquete

- **PILAS:** Los paquetes de ROS se organizan en pilas. Mientras que el objetivo de los paquetes es crear colecciones de código mínimo para una fácil reutilización, el objetivo de las pilas es simplificar el proceso de compartir código. Las pilas son el principal mecanismo de ROS para distribuir software. Cada pila tiene una versión asociada y puede declarar las dependencias de otras pilas. Estas dependencias también declaran un número de versión, lo que proporciona una mayor estabilidad en el desarrollo. A diferencia de una biblioteca de software tradicional que enlaza contra el tiempo de compilación, estas pilas también puede proporcionar esta funcionalidad en tiempo de ejecución a través de ROS temas y servicios [9].

  Una pila es simplemente un directorio que contiene un archivo stack.xml y proviene de las Variables de Entorno ROS_ROOT o ROS_PACKAGE_PATH de las cuales se hablará más adelante.

  Cualquier directorio dentro de la ruta del paquete ROS que contiene un archivo stack.xml es considerado como una pila, y cualquier paquete dentro de ella se consideran parte de dicha pila.

  El archivo stack.xml declara las dependencias de otras pilas. Los archivos stack.xml suelen incluir etiquetas <depend> que declaran las

pilas que se deben instalar como pre-requisitos. Además las pilas no pueden contener pilas. [10]

- **MSG:** ROS usa un lenguaje de descripción de mensajes simplificado para describir los valores de los datos que los Nodos de ROS publican. Esta descripción hace fácil para las herramientas de ROS generar el código fuente para el tipo de mensaje. Descripciones de los mensajes se almacenan en archivos **.msg** en el subdirectorio msg/ de un paquete de ROS. El cual consta de dos partes: campos y constantes. Los campos son datos que se envían dentro del mensaje. Y las constantes definen los valores útiles que se pueden utilizar para interpretar los campos [11]. Un archivo .msg tiene un contenido como el mostrado anteriormente en la Figura 2.2 y se debe llamar al archivo de la siguiente manera:

  nombre_paquete/msg/nombre_archivo.msg

  o directamente:

  nombre_paquete/nombre_archivo

- **SRV:** ROS utiliza un lenguaje de descripción de servicios simplificado para describir los tipos de servicio de ROS. Esto se basa directamente

en el formato **“msg”** de ROS para permitir la comunicación petición / respuesta entre los nodos. Las descripciones de servicios se almacenan en archivos .**srv** en el subdirectorio de un paquete **srv/.** Las descripciones de servicios se refiere a la utilización de nombres de paquetes de recursos.

Se debe llamar al archivo de la siguiente manera:

```
nombre_paquete/srv/nombre_archivo.msg
```

o directamente:

```
nombre_paquete/nombre_archivo
```

Un archivo de descripción de servicio consiste de una solicitud y un respuesta tipo **“msg”**, separados por '---' [12]. Cualquiera de los dos archivos .msg concatenados con un '---' son una descripción de servicio legal. Contenido de un archivo srv el cual solicita un string y entrega otro string:

```
string str
---
string str
```

2.2.3. Nivel Comunitario

Comprende los recursos disponibles a las comunidades que incluyen:

Distribución: Son colecciones de pilas con versiones que se pueden instalar. Es como una distribución de Linux que tratan de hacer fácil la instalación de programas informáticos y que logra mantener versiones consistentes de un software.

Repositorios: Gran cantidad de repositorios de código que forman parte de la red de donde las instituciones reutilizan código para luego lanzar sus propios software.

Wiki ROS: Aquí se concentra el más grande y principal foro donde se encuentra toda la información sobre ROS. Sólo creando una cuenta puedes contribuir con nueva documentación para ayudar a todos quienes están desarrollando alguna aplicación con ROS. Además de facilitar correcciones y actualizaciones, entre otros beneficios.

Listas de correo: La manera más fácil de comunicar sobre actualizaciones a los usuarios de ROS, además de tener un foro sobre el software ROS.

Blog: El blog de Willow Garage ofrece actualizaciones periódicas que incluyen fotos y videos.

Un programa de ROS realmente no es diferente de un programa tradicional, en lugar de redefinir por completo la "gramática" o "vocabulario" de la programación, ROS sólo incorpora funciones al tradicional programa C + +. En otras palabras, usted puede obtener:

- Abstracción jerárquica y la gestión de los programas en ejecución.
- La comunicación entre los programas.
- Una colección de programas de gran alcance y bibliotecas de código como extensión. Sin reescribir el código principal, sino simplemente utilizando algunas clases o llamadas a funciones.

2.3. Objetivos de ROS

- **Ligero:** ROS está diseñado para ser lo más ligero posible para que el código escrito por ROS se puede utilizar con otros frameworks. Algo importante de esto es que ROS es fácil de integrar con otros frameworks de software para robots: ROS ha sido integrado con OpenRAVE, Orocos y Player. [13]
- **Código reutilizable:** ROS reutiliza el código de muchos otros proyectos de código abierto, tales como los controladores, sistema de

navegación y simuladores del proyecto Player [14], a partir de algoritmos de visión OpenCV [15], y los algoritmos de planificación de OpenRAVE [16], entre otros. En cada caso, ROS se utiliza sólo para demostrar diversas opciones de configuración y para encaminar los datos hacia y desde el software correspondiente [17].

2.4. Características de ROS

- **Multi-Lenguaje:** El framework ROS es fácil de implementar en cualquier lenguaje de programación moderno. Ya ha sido implementado en Python, C++, y Lisp, y existen bibliotecas experimentales en Java y Lua [18].

  ROS es independiente del lenguaje de programación. Para apoyar el desarrollo de varios lenguajes, ROS utiliza un simple lenguaje neutral, Interface Definition Language (IDL) para describir los mensajes enviados entre los módulos. El IDL utiliza pequeños archivos de texto para describir los campos de cada mensaje, y permite la composición de los mensajes [19].

- **Libre y de código abierto:** El código fuente completo de ROS está disponible públicamente, esto es fundamental para facilitar la depuración en todos los niveles de la pila de software. ROS pasa datos entre módulos utilizando las comunicaciones entre procesos, y

no requiere que los módulos esten en el mismo ejecutable. Como tal, sistemas construidos alrededor de ROS puede utilizar diversos componentes. Los módulos individuales pueden incorporar software protegido por diferentes licencias que van desde la GPL a BSD a propietario [20].

- **Fácil verificación:** ROS tiene entre sus funciones una llamada "rostest" que permite hacer las pruebas de integración a través de múltiples nodos [21].
- **Basado en herramientas:** En un esfuerzo para gestionar la complejidad de ROS, se optó por un diseño microkernel para ROS (no se refiere al Kernel de Linux), donde un gran número de pequeñas herramientas se utilizan para generar y ejecutar los diversos componentes de ROS, en lugar de construir un desarrollo monolítico y entorno de ejecución. Estas herramientas realizan diversas tareas, por ejemplo, navegar por el código fuente, obtener y establecer los parámetros de configuración, visualizar la conexión peer-to-peer y generar automáticamente la documentación. A pesar de que podría haberse implementado servicios de núcleo, tales como un reloj mundial y un registrador dentro del módulo maestro, se ha intentado poner cada cosa en módulos separados para que la

pérdida de eficacia sea compensada por el aumento de la estabilidad y la complejidad de gestión [22].

- **Peer-to-peer:** Un sistema construido utilizando ROS se compone de una serie de procesos, potencialmente en un número de diferentes equipos, conectados en tiempo de ejecución en una topología pero-to-peer (punto a punto). Aunque los frameworks basados en un servidor central (por ejemplo, CARMEN) pueden también realizar los beneficios de diseño de multi-proceso y multi-equipos. Un servidor de datos central es problemático si los equipos están conectado en una red heterogénea.
- La topología peer-to-peer requiere algún tipo de mecanismo de búsqueda para permitir que los procesos se encuentren entre sí en tiempo de ejecución. A esto le llamamos name service o master que se describen más adelante [23].

Actualmente, ROS sólo se ejecuta en plataformas basadas en UNIX y ha sido probado en Ubuntu y Mac OS X.

ROS proporciona un sistema de comunicación entre procesos que permite el intercambio de datos entre varios procesos que se ejecutan en uno o más equipos físicos.

CAPITULO 3

3. Instalación y Configuración de ROS y NXT-ROS

En este capítulo se explica el proceso de instalación y configuración de ROS y NXT-ROS. Todo esto se realiza mediante la interfaz de comandos.

Antes de entrar en detalle la configuración, se debe tener claro que ROS tiene varias distribuciones. Una distribución de ROS es un conjunto de pilas, para las diferentes plataformas robóticas.

Las distribuciones disponibles en ROS son: ROS Groovy Galápagos, ROS Fuerte Turtle, ROS Electric Emys, ROS Diamondback, ROS C Turtle, ROS Box Turtle y la próxima en ser publicada ROS Hydro Medusa.

Para este proyecto de graduación se utilizó la distribución ROS Electric Emys. Debido a que es la única distribución de ROS que contiene el paquete de NXT-ROS.

Antes de comenzar con la instalación de ROS, se instaló Ubuntu Linux versión 11.10 (Oneiric).

3.1. Instalación de ROS

3.1.1. Configurar Plataforma Ubuntu Linux

Ubuntu debe aceptar los repositorios "restricted," "universe," y "multiverse." Estos repositorios facilitan la instalación de ROS, debido a que permite descargar software no gratuito (multiverse), software comunitario, es decir no oficialmente soportado (universe), software que no es completamente disponible bajo libre distribución (restricted).

Se debe configurar el computador para aceptar ROS, en este proyecto se utilizó el siguiente comando:

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

Este comando configura la lista de direcciones, de donde se descargan las fuentes de los programas y actualizaciones del sistema. Y se encuentra como lo muestra el comando en la ruta /etc/apt/source.list

3.1.2. Configuración de claves

Hay que configurar las claves (keys) con el siguiente comando:

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

Este comando permite la identificación de los paquetes de ROS, para ser aceptados como de confianza.

3.1.3. Instalación

Se debe asegurar que se ha incluido el servidor ros.org con el siguiente comando:

sudo apt-get update

En el tutorial de instalación de ROS se puede ver las diferentes configuraciones de ROS dependiendo de las librerías y herramientas que el usuario necesite. Para este proyecto se instaló Desktop-Full Install: (Recommended) con la siguiente instrucción:

sudo apt-get install ros-electric-desktop-full

3.1.4. Configuración de variables de entorno

Es conveniente que las variables de entorno se agreguen automáticamente al bash cada vez que un nuevo shell se inicia, el siguiente comando se utiliza para esto:

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

3.2. Configuración del Equipo para que trabaje con LEGO NXT

Antes de que se pueda establecer una comunicación con el brick NXT se debe establecer lo siguiente:

1. Se comienza agregando el grupo Lego

   sudo groupadd lego

2. Luego el usuario que va a trabajar con ROS debe adherirse al grupo lego

   sudo usermod -a –G lego <mafer utreras>

Luego de esto se debe crear un archivo de reglas udev para el grupo lego que se acaba de crear.

Udev es el gestor de archivos que usa el kernel de Linux [24]. Este detecta cuando se ha conectado o desconectado un dispositivo al

sistema. El archivos de reglas udev para este proyecto se lo crea con el siguiente comando:

echo "BUS==\"usb\", ATRÁS {idVendor]== \"0694\", GROUP=\"lego\", MODE=\"0660\""> /tmp/70-lego.rules && sudo mv /tmp/70-lego.rules /etc/udev/rules.d/70-lego.rules

Luego de esto se debe reiniciar el udev:

```
sudo restart udev
```

Una vez instalado ROS, se debe instalar los paquetes de NXT-ROS

```
sudo apt-get update
sudo apt-get install ros-electric-nxtall
```

3.3. Instalación y configuración del entorno de Ros para un Proyecto

Apenas se haya instalado ROS hay que tener el archivo setup.*sh en: '/opt/ros/<distro>/'

```
# source /opt/ros/<distro>/setup.bash
```

Que para este proyecto fue:

```
# source /opt/ros/electric/setup.bash
```

Cuando se quiere comenzar con un nuevo proyecto en ROS, debemos iniciar creando un “espacio de trabajo”, el cual administra todos los paquetes instalados en una ruta de ROS que el usuario que esté trabajando escoja.

3.3.1. Creación de un Espacio de Trabajo

Un espacio de trabajo es un conjunto de carpetas en el disco duro que se enlistan en un archivo especial llamado rosinstall [25]. El comando que se mostrará a continuación, crea un nuevo espacio de trabajo en: ~/electric_workspace el cual se extriende del conjunto de paquetes instalados en /opt/ros/electric

```
rosws init ~/electric_workspace /opt/ros/electric
```

El comando rosws genera un archivo llamado setup.sh, este agrega todas las carpetas enlistadas en el archivo especial rosinstall a una variable de entorno llamada ROS_PACKAGE_PATH. Esta variable puede ser usada como por ejemplo para encontrar carpetas con las librerías que el usuario desee compilar.

El directorio /electric_workspace es un nombre cualquiera que el usuario le da al espacio de trabajo en el que va a trabajar un proyecto. Por lo tanto siempre se debe utilizar el nombre del espacio de trabajo en el que

se vaya a trabajar. Se debe saber que el comando rosws es parte del paquete rosinstall, el cual no se instala por defecto, así que con la siguiente instrucción se podrá instalar:

```
sudo apt-get install python-rosinstall
```

3.3.2. Creación de un directorio para los nuevos paquetes

Los nuevos paquetes necesitan ser puestos en una ruta que está en la variable de entorno del ROS_PACKAGE_PATH.

Todos los directorios son administrados por rosws, es decir que utilizando rosws se agregan automáticamente a la ROS_PACKAGE_PATH, cuando un archivo setup.bash del area de trabajo correspondiente es obtenido.

Aunque los nuevos paquetes siempre deben ser puestos en repositorios que han sido instalados usando rosws, la creación de un directorio es muy conveniente debido a que aquí se pueden poner los paquetes creados para proyectos sin necesidad de utilizar comandos rosws adicionales. Por esto se debe crear un nuevo directorio y añadir el archivo .rosintall oculto con la instrucción rosws:

```
mkdir ~ / electric_workspace / nuevo_directorio
rosws set ~ / electric_workspace / nuevo_directorio
```

Cada vez que se desee ingresar al area de trabajo, hay que utilizar el siguiente comando:

```
source ~/electric_workspace/setup.bash
```

Source es un comando de Linux que permite afectar las variables de entorno desde un archivo. Aparentemente bash no las intepreta o ni genera un ambiente particular para los archivos .sh y por eso no se actualiza el global, usando el comando “source”, las variables de ambiente afectan desde que se las llama [26]. En ROS se tendrá que usar este comando para poder tener acceso a los comandos de ROS.

3.3.3. Creación de un Paquete

Los paquetes de ROS contienen archivos comunes: manifests, CMakeLists.txt, mainpage.dox, and Makefiles, la instrucción roscreate-pkg ayuda a crear todo esto automáticamente.

```
# roscreate-pkg [package_name]
```

Esta instrucción se la utiliza dentro de la ruta del directorio donde se desee crear el paquete.

También se le puede agregar a la instrucción las dependencias que se vayan a utilizar en el proyecto.

```
# roscreate-pkg [package_name] [depend1] [depend2] [depend3]
```

Los paquetes de ROS algunas veces requieren de bibliotecas externas y herramientas, que deben ser proporcionadas por el sistema operativo. Estas herramientas y bibliotecas se refieren comúnmente como dependencias del sistema.

Las dependencias del sistema comúnmente pueden ser: std_msgs rospy, roscpp, roslib, etc y así quedará la instrucción:

```
$ roscreate-pkg beginner_tutorials std_msgs rospy roscpp
```

3.3.4. Construir Paquete

Apenas se hayan creado las dependencias, se puede continuar construyendo el paquete creado con el siguiente comando:

```
rosmake [package]
```

El comando rosmake, es similar al comando make, cuando hacemos un rosmake [package], se construye el paquete con nombre[package] y sus respectivas dependencias como las ya mencionados antes: std_msgs, rospy y roscpp.

3.3.5. Creación del archivo launch

El archivo launch es un archivo escrito con el formato XML, este archivo describe los nodos que se van a ejecutar, los parámetros que deben ser establecidos y otros atributos de los nodos para que sean cargados y compilados con éxito. Un ejemplo del archivo launch es el siguiente:

```
1  <launch>
2  <node pkg="paquetenuevo" type="nuevo.py" name="laser"
       output="screen" respawn="false"/>
3  <node pkg="paquetenuevo" type="pruebacamara1" name="camara"
     output="screen" respawn="false"/>
4   </launch>
```

En la segunda línea donde especifica el nodo, se le está dando un nombre, el cual como lo hemos visto anteriormente tiene que ser único, en este caso pkg=”paquetenuevo” es el identificador. Siguiente a eso se tiene que encontrar el pkg que en este caso es el nuevo.py.

3.4. Creación de un Nodo Publicador y Nodo suscriptor en Python

ROS es independiente del lenguaje. En este momento, tres bibliotecas principales se han definido para ROS, lo que hace posible programar ROS en Python, Lisp o C++. Además de estas tres bibliotecas, dos bibliotecas experimentales se ofrecen, por lo que es posible programar ROS en Java o Lua [27].

Como se sabe en el capitulo anterior se explicó que es un nodo, el cual en términos sencillos es un ejecutable, el cual puede publicar o suscribirse a un topic.

El código del publicador o suscriptor puede estar escrito en c++ o en python y además ya están establecidos, lo único que hay que hacer es agregar el código a un proyecto deseado.

3.4.1 Código Publicador en Python

En términos generales el código del publicador es el siguiente:

```
1 #!/usr/bin/env python
2 import roslib; roslib.load_manifest('beginner_tutorials')
3 import rospy
```

```
4 from std_msgs.msg import String
5 def talker():
6  pub = rospy.Publisher('chatter', String)
7    rospy.init_node('talker')
8  while not rospy.is_shutdown():
9    str = "hello world %s" % rospy.get_time()
10   rospy.loginfo(str)
11     pub.publish(String(str))
12     rospy.sleep(1.0)
13 if __name__ == '__main__':
14 try:
15   talker()
16 except rospy.ROSInterruptException:
17   pass
```

La línea 1 explica que es un ejecutable de python, la 2da línea importa roslib para que lea el archivo manifests.xml y que añada todas las dependencias establecidas.

En la línea 3 se debe importar siempre rospy cuando se está trabajando con un nodo de ROS. En la línea 4 lo que explica es que se importa

String de std_msgs.msg lo cual simplemente hace que se pueda volver a utilizar un String para la publicación.

En la sección de la línea 7 a la 9 se define la interfaz del publicador en ROS, en la línea 8 declara que el nodo está publicado bajo el topic “chatter”. En la siguiente línea rospy.init_node es muy importante, debido a que este le dice el nombre del nodo a rospy y hasta que no reciba esta información, no se podrá iniciar la comunicación con el ROS Master.

La línea 10 muestra un lazo que se ejecuta mientras no haya una señal (como ctrl +C) para parar el lazo. En esta sección el caso es una llamada al pub.publish(String(str)), que publica a nuestro topic.

En la línea 14 rospy.sleep, es similar a time.sleep, este le pide información a rospy.loginfo(str) que debe hacer algunos trabajos como: imprimir mensajes en pantalla, escribir registro de los nodos y escribir a rosout. El mensaje std_msg.String es un tipo simple de mensaje.

En la última parte del código se encuentra el main en Python, este atrapa una excepción rospy.ROSInterruptException, que la puede lanzar rospy.sleep que son métodos para poder ingresar por teclado Ctrl-C para suspender el nodo. Esta excepción solo tiene una única razón que es

evitar que accidentalmente se continúe con la ejecución de un código después de la suspensión.

3.4.2. Código Suscriptor en Python

Ahora se explica el código del nodo suscriptor, que es el que recibe el o los mensajes.

```
1 #!/usr/bin/env python
2 import roslib; roslib.load_manifest('beginner_tutorials')
3 import rospy
4 from std_msgs.msg import String
5 def callback(data):
6     rospy.loginfo(rospy.get_name() + ": I heard %s" % data.data)
7 def listener():
8   rospy.init_node('listener', anonymous=True)
9    rospy.Subscriber("chatter", String, callback)
10    rospy.spin()
11 if __name__ == '__main__':
12  listener()
```

Este código es similar al código del publicador, la diferencia es que este código introduce un nuevo mecanismo de “callback” para suscribirse a los mensajes.

El nodo se suscribe al topic “”chatter” que es de tipo std_msgs.msgs.String como ya se lo ha mencionado antes. Cuando los mensajes son recibidos, el callback es invocado con el mensaje como el primer argumento.

También se puede destacar que se cambió la llamada a rospy.init_node(). Aquí se incrementó la palabra clave anonymous=true. ROS requiere que cada nodo tenga un nombre único. Si un nodo con el mismo nombre aparece, este tiene un conflicto con el anterior, esto es para que los nodos con mal funcionamiento fácilmente puedan ser expulsados de la red.

El anonymous=True le permite a rospy generar un único nombre para el nodo y asi se puede facilitar la ejecución de los nodos suscriptores.

Lo último por destacar es el rospy.spin(), este simplemente mantiene el nodo de salida hasta que este haya sido apagado.

3.5. Creación de un Nodo Publicador y Nodo suscriptor en C++

3.5.1. Código del Publicador

El código del publicador es el siguiente:

```
1  #include "ros/ros.h"
2  #include "std_msgs/String.h"
3  #include <sstream>
4  int main(int argc, char **argv)
   {
5   ros::init(argc, argv, "talker");
6  ros::NodeHandle n;
7                    ros::Publisher chatter_pub =
n.advertise<std_msgs::String>("chatter",1000);
8  ros::Rate loop_rate(10);
9  int count = 0;
10 while (ros::ok())
    {
11    std_msgs::String msg;
12   std::stringstream ss;
13    ss << "hello world " << count;
14   msg.data = ss.str();
15    ROS_INFO("%s", msg.data.c_str());
16   chatter_pub.publish(msg);
17   ros::spinOnce();
```

```
18   loop_rate.sleep();
19  ++count;
   }
20  return 0;
 }
```

En la línea 1 ros/ros.h se incluye, ya que conviene adherir todas las cabeceras necesarias para que puedan ser usadas las piezas más comunes de ROS.

En la línea 2 include "std_msgs/String.h, incluye los mensajes std_msgs/string, que se encuentra en el paquete de std_msgs. Este encabezado se genera automáticamente a partir del archivo std_msgs.

ros::init(argc, argv, "talker");

Esto inicializa el nodo, aquí también se especifica el nombre del nodo

ros::NodeHandle n;

Es una clase y tiene dos propósitos. En primer lugar, proporciona el arranque o inicialización del nodo que está dentro de un programa roscpp. En segundo lugar, proporciona una capa adicional de espacio de nombres, que puede hacer más fácil la escritura de los subcomponentes y en ese caso sería:

ros::NodeHandle nh("my_namespace");

En la creación, si el nodo interno no se ha iniciado aun, ros :: NodeHandle iniciará el nodo. Una vez que ros :: NodeHandle ha sido suspendido, el nodo se cerrará automáticamente.

ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);

Aquí le dice al Master que se va a publicar un mensaje de tipo std_msgs/String en el topic “chatter”. Esto deja que el Master diga a cualquier nodo que escuche en “chatter”, ya que se va a publicar datos en ese topic. El segundo argumento es el tamaño de la cola, en donde se explica que un máximo de 1000 mensajes se publicará antes de comenzar a desechar los antiguos.

ros::Rate loop_rate(10); esta instrucción permite especificar la frecuencia en el lazo. Esto también hará un seguimiento de cuánto tiempo ha pasado desde la última llamada a Rate::sleep(). En este caso se quiere que corra a 10Hz.

En la línea 9 y 10 roscpp por defecto instalará un SIGINT, que provee un manejo del Ctrl-C lo cual hara que ros::ok() retorne un falso, si esto pasa es porque: se recibió un Ctrl-C, cuando ha sido sacado de la red por otro

nodo del mismo nombre, cuando ros::shutdown() ha sido llamada por otra parte de la aplicación, cuando todos los ros::NodeHandles han sido destruidos, asi una vez que ros::ok devuelva un falso todas las llamadas de ROS fallaran.

De la línea 11 a la 14 se emite un mensaje en ROS usando una clase adaptador de mensaje, generalmente esta es generada desde un archivo msg. Hay tipos de datos mucho más complicados pero en este ejemplo es un dato estándar tipo String.

En la línea 16 es donde realmente se transmite el mensaje que cualquiera que esté conectado.

En la línea 15 ROS_INFO imprime en pantalla.

En la línea 17 esta instrucción es necesaria para que un nodo se suscriba a la aplicación, ya que sin esta, los callbacks nunca serian ejecutados.

En la línea 18 se manda a dormir durante el tiempo restante que deja la tasa de publicación de 10Hz. Lo que pasaría es que: inicializa ROS, anuncia los mensajes que se van a publicar (std_msgs / mensajes) en el topic “chatter” a el Master, por último el lazo publica los mensajes al topic “chatter” 10 veces por segundo. Ahora se debe escribir un nodo que reciba los mensajes.

3.5.2. Código del Suscriptor

El código del suscriptor es el siguiente:

```
1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3 void chatterCallback(const std_msgs::String::ConstPtr& msg)
 {
4   ROS_INFO("I heard: [%s]", msg->data.c_str());
 }
5 int main(int argc, char **argv)
  {
6   ros::init(argc, argv, "listener");
7   ros::NodeHandle n;
8   ros::Subscriber sub = n.subscribe("chatter", 1000,  chatterCallback);
9   ros::spin();
10   return 0;
  }
```

En la línea 3 y 4 aquí es donde se llama a la función callback siempre que un mensaje haya llegado al topic “chatter”.

En la línea 8, se suscribe al topic “chatter” con el Master. aqui el topic chatter se subscribe al nodo master de ROS. ROS llama a la función de callback cada vez que llega un nuevo mensaje. El segundo argumento es el tamaño de la cola, debido a que si los mensajes no se procesan lo

suficientemente rápido, cuando la cola llegue a mil mensajes, se comenzará a desechar mensajes antiguos cada vez que llegue uno nuevo.

ros::spin(); aquí el código retorna un falso, lo cual significa que el ros::shutdown() ha sido llamado, ya sea por teclado que se introduzca Ctrl-C.

3.6. Conectar NXT Lego

Vía USB se conecta el NXT Lego y se utiliza el comando:

```
$ roslaunch [nombredelpaquete] robot.launch
```

Este comando ejecuta los Nodos según se hayan programado, y se comunican entre sí dependiendo que tipo de nodos se hayan creado (suscriptor o publicador), se podrán ver los mensajes de comunicación entre uno y otro nodo. En el capítulo 5 se podrá ver un ejemplo con el proyecto Scanner Laser que se realizó para esta tesis.

CAPÍTULO 4

4. LENGUAJES DE PROGRAMACION EN ROS

Como se explicó anteriormente en ROS es posible programar en Python, Lisp o C++. No existe una regla en ROS que defina que un Lenguaje de Programación es utilizado para hacer algo específico. ROS permite usar cualquiera de los Lenguajes de Programación mencionados para las aplicaciones que se deseen hacer de acuerdo a sus propias habilidades.

Para el desarrollo del proyecto de graduación “Láser Scanner” del cual hablaremos en la siguiente sección, se usa Python para crear un código con la finalidad de controlar los motores del robot y C++ para adquirir imágenes de algún objeto.

4.1. Lenguaje de Programación Python

En esta sección se explicará el significado de las funciones, módulos y paquetes comúnmente usados en este proyecto en Python, además dar a conocer la estructura que debe tener un archivo Python listo para ser ejecutado desde la primera línea de código, hacer énfasis en el Paquete NXT-Python que se utiliza en este proyecto de graduación.

El Lenguaje de Programación Python tiene entre sus características más importantes:

- Claro y Sintaxis legible y clara.
- Fuerte capacidad de introspección.
- Orientación a objetos intuitivo.
- Expresión natural del código.
- Modularidad completa con el apoyo a los paquetes jerárquicos.
- Excepción basada en el manejo de errores.
- Extensas bibliotecas estándar y módulos de terceros para prácticamente todas las tareas.
- Extensiones y módulos fácilmente escritos en C, C + + (o Java para Jython, o. NET para IronPython).
- Integrable dentro de las aplicaciones como una interfaz de scripting [28]

Su biblioteca e intérprete son de código abierto, permite ampliarlo con nuevas funciones y tipos de datos implementados en C o C++. Python contiene distribuciones, módulos, herramientas y programas, además de documentación adicional. Python puede dividir un programa en módulos reutilizables. Los programas en Python son más cortos que los escritos en C o C++ por los siguientes motivos:

- En una sola sentencia donde se expresan operaciones con tipos de datos de alto nivel.
- Se agrupan las sentencias por las sangrías (indentación), no se usan llaves o begin/end.
- No es necesario declarar los argumentos ni las variables.

4.1.1. Python para NXT

El Lenguaje de Programación Python ha creado un controlador/interfaz exclusivamente para el Robot Lego Mindstorms NXT llamado nxt-python. NXT-Python es una API de Python para controlar el Robot a través del Lenguaje de Programación Python. El cual puede comunicarse mediante USB o Bluetooth.

Todo archivo Python tiene extensión .py y como todo script necesariamente debe tener escrito al inicio del archivo la línea:

```
#! /usr/bin/env python
```

Luego de la cual se escribe el código. Para su ejecución debe tener los permisos adecuados a través de consola con los comandos: **$ chmod +x nombre_de_script.py**. Dentro de un archivo Python se incluyen comentarios con el caractér "#" que tiene validez hasta el final de la línea física.

4.1.2. Funciones en Python

Una función es un fragmento de código con un nombre asociado que realiza una serie de tareas y devuelve un valor. A los fragmentos de código que tienen un nombre asociado y no devuelven valores se les suele llamar procedimientos. En Python no existen los procedimientos, ya que cuando el programador no especifica un valor de retorno la función devuelve el valor None (nada), equivalente al null de Java.

En Python las funciones se declaran de la siguiente forma:

```
def mi_funcion(param1, param2):
print param1
print param2
```

Para definir una función se escribe la palabra clave **def** seguida del nombre de la función y entre paréntesis los argumentos separados por comas. A continuación, en otra línea y después de los dos puntos las

líneas de código de la función. Para devolver valores, se utiliza la palabra clave **return** [29].

4.1.2.1 Funciones en NXT-Python

A continuación la Tabla I muestra las funciones más comunes para el desarrollo de este proyecto con el LEGO NXT.

Tabla I Funciones comunes.

find_brick	Busca un ladrillo y se conecta a él si puede.
get_touch_sample	Obtiene una muestra desde el puerto especificado.
close_brick:	Cierra la conexión con el ladrillo. `'close_brick'`
nxt_filer:	GUI simple para administrar archivos en un LEGO Mindstorms NXT.
filter(*función*, *secuencia*)	Filtra, devuelve una secuencia (del mismo tipo, si es posible) que contiene elementos de la secuencia de entrada para los que *función*(*elemento*) resulta verdadero.

map(*función*, *secuencia*)	Llama a *función*(*elemento*) para cada elemento de la secuencia, devuelve una lista de valores resultantes. Es posible pasar varias secuencias como parámetro pero la función debe tener tantos argumentos como secuencias y se llama a la función con el valor correspondiente de cada secuencia de entrada (o None cuando una secuencia es más corta que otra).
reduce(*func*, *secuencia*)	Reduce, devuelve un valor simple llamando a función binaria ***func*** con los dos primeros elementos de la secuencia, luego con el resultado y el siguiente elemento y así sucesivamente. Si sólo hay un elemento en la secuencia, se devuelve su valor; si está vacía, se lanza una excepción. Se puede pasar

	un tercer argumento para indicar valor inicia devolviendo este valor inicial para la secuencia vacía y la función se aplica al primer elemento, al segundo y así sucesivamente.
nxt_test:	Muestra información variada sobre todos los bricks que puede conectar.
nxt_sensor_report:	Guía al usuario a traves del proceso de reportar la identificación de sus sensores digitales para el tipo de comprobación de base de datos.
nxt_push:	Coloca un archivo a un brick del LEGO

4.1.2.2. Funciones principales para los Motores en NXT-Python

Tabla II Funciones para los motores

run(self, power=100, regulated=False)	Motor funcionando de forma continuas Si regulated=verdadero entonces la sincronización trabaja.

stop_motor:	Detiene el motor con el puerto y frenado especificado (bool).`'run_motor:a,1'`
update_motor	update_motor: Actualizaciones de el o los motores con el o los puertos, potencia. `'update_motor:a,100,300'`
turn(self, power, tacho_units, brake=True, timeout=1, emulate=True)	Mueve un motor a una velocidad 'power', un angulo de 'tacho_units' grados. "brake" es un parámetro opcional, y dice si debe o no mantener el motor después de las salidas de la función "timeout" es un parámetro opcional, y es el tiempo de espera si el motor está bloqueado para lanzar una excepción. "emulate" es un parámetro opcional, es un valor booleano si está en false el motor será consciente del valor del límite de grados, si es True, se

	utilizará una función equivalente llamada run().
turn(self, power, tacho_units, brake=True, timeout=1)	Mueve los motores sincronizados a velocidad 'power', un angulo de 'tacho_units' grados.
brake(self):	Mantiene el motor en un lugar.
get_accelerometer_sample():	Obtiene una muestra desde el puerto especificado.

Ejemplo 1:

```
#!/usr/bin/env python
# importamos las funciones de la API
import nxt.locator
from nxt.motor import *

# esta función trata de conectarse al brick
b = nxt.locator.find_one_brick()

# movemos el motor conectado en el puerto B del brick b
motor_izquierdo = Motor(b, PORT_B)

# con potencia 100 y que de una vuelta completa (360º)
motor_izquierdo.turn(100, 360)
```

4.1.2.3. Funciones principales para los Sensores en NXT-Python

Tabla III Funciones Sensor de Toque

Sensor de Toque	
__**init**__(self, brick, port):	Creador de un objeto de la clase Touch(self), en el parámetro brick se la pasa un objeto tipo brick anteriormente creado y en port el puerto en el cual estará el sensor.
get_sample():	Obtiene el valor instantáneo del sensor

Ejemplo 2: Sensor de Toque

```
#!/usr/bin/env python
import nxt.locator
from nxt.sensor import *
 # esta función trata de conectarse al brick
b = nxt.locator.find_one_brick()
 # usaremos el botón conectado en el puerto 1 del brick
boton = Touch(b, PORT_1)
 #obtenemos el valor del sensor
valor_boton = boton.get_sample()
print 'Botón: ', valor_boton
```

Tabla IV Funciones Sensor de Luz

Sensor de Luz	
__init__(self, brick, port, illuminated=True):	Crea un objeto de clase Ligth (self); al parámetro brick se le pasa un objeto tipo brick antes creado; port el puerto donde está el sensor, illuminated es de tipo booleano si está en True el sensor se ilumina si es False apagado.
set_illuminated(self, active):	Permite prender o apagar el led, pasándole como parámetro el objeto y un valor booleano (True para prenderlo, False para apagarlo).

Ejemplo 3: Sensor de Luz

```
#!/usr/bin/env python
import nxt.locator
from nxt.sensor import *
b = nxt.locator.find_one_brick()
luz = Light(b, PORT_3)
valor_luz = luz.get_sample()
print 'Luz: ', valor_luz
```

Tabla V Funciones Sensor de Sonido

Sensor de Sonido	
__init__(self, brick, port, adjusted=True):	Creador del un objeto de la clase Sound (self), en el parámetro brick se la pasa un objeto tipo brick anteriormente creado, en port el puerto en el cual estará el sensor, adjusted es del tipo booleano si está en True el sensor se ajustará para detectar sonidos iguales al rango audible del ser humano si es falso se ajustará para detectar todo tipo de sonidos.
set_adjusted(self, active):	Esta función permite reajustar el sensor según el parámetro booleano active.
play_tone():	Reproduce un tono de la frecuencia y la duración especificada ``'play_tone:500,100'``

Ejemplo 4: Sensor de Sonido

```
#!/usr/bin/env python
# importamos las funciones de la API
import nxt.locator
from nxt.sensor import *

# esta función trata de conectarse al brick
b = nxt.locator.find_one_brick()

# usaremos el sensor de sonido conectado en el puerto 2 del brick
sonido = Sound(b, PORT_2)

#obtenemos el valor del sensor
valor_sonido = sonido.get_sample()

# imprimimos en pantalla el val
print 'Sonido: ', valor_sonido
```

Tabla VI Funciones Sensor Ultrasónico

Sensor Ultrasónico	
__**init**__(self, brick, port, check_compatible=True):	Crea un objeto de la clase Ultrasonic; en el parámetro brick se pasa un objeto tipo birck ya creado; port el puerto donde estar el sensor, check_compatible es un parámetro booleano. [30]

Ejemplo 5: Sensor Ultrasónico

```
#!/usr/bin/env python
import nxt.locator
from nxt.sensor import *

# esta función trata de conectarse al brick
b = nxt.locator.find_one_brick()
 # usaremos el sensor ultrasónico conectado en el puerto 4 del brick
ultrasonico = Ultrasonic(b, PORT_4)
 #obtenemos el valor del sensor
valor_ultrasonico = ultrasonico.get_sample()

# imprimimos en pantalla el valor
print 'Ultrasónico: ', valor_ultrasonico
```

4.1.3. Módulos en Python

Para facilitar el mantenimiento y lectura los programas extensos pueden dividirse en módulos, agrupando elementos relacionados. Los módulos son entidades que permiten una organización y división lógica del código. Para utilizar la funcionalidad definida en este módulo se debe importarlo. Para importar un módulo se utiliza la palabra clave **import** seguida del nombre del módulo, que consiste en el nombre del archivo menos la extensión. Como ejemplo, se crea un archivo programa.py en el mismo directorio donde se guarda el archivo del módulo (esto es importante, porque si no se encuentra en el mismo directorio Python no podrá encontrarlo) con el siguiente contenido:

```
import modulo
modulo.mi_funcion()
```

El **import** no solo permite tener disponible todo lo definido dentro del módulo, sino que también ejecuta el código del módulo. La clausula **import** también permite importar varios módulos en la misma línea [29].

Un módulo puede *importar* a otros módulos o al módulo *principal* (la colección de variables accesible desde un guion ejecutado desde el nivel superior). El nombre del fichero es el nombre del módulo con el sufijo .py.

Estos pueden contener definiciones además de sentencias ejecutables que sirven para inicializar el módulo. Ejemplo:

import nxt.locator

Es costumbre no obligatoria colocar todas las sentencias import al principio del módulo. Existe además una manera de importar todos los nombres de un módulo. Ejemplo:

from nxt.motor import *

Esto importa todos los nombres, excepto los que empiezan por un guion bajo (_).

Al importar el módulo denominado **mimodulo**, el intérprete busca un fichero denominado mimodulo.py en el directorio actual, luego en la lista de directorios especificada por la variable de entorno PYTHONPATH, de la cual se trata más adelante. Si PYTHONPATH no tiene valor o no se encuentra el fichero, se continúa la búsqueda en un camino dependiente de la instalación. En UNIX, normalmente es: /usr/local/lib/python.

En realidad, se buscan los módulos en la lista de directorios dada por la variable **sys.path**, que se inicializa desde el directorio que contiene el guion de entrada (o el directorio actual), PYTHONPATH y el valor por omisión dependiente de la instalación. Esto permite que los programas

que saben lo que hacen modifiquen o reemplacen el camino de búsqueda de módulos [31].

4.1.3.1. Módulos en NXT-Python

El paquete NXT-Python del cual se habla en la siguiente sección contiene varios módulos, los más usados en este proyecto fueron: Modulo Locator y Modulo Motor los cuales se detallan a continuación en tablas:

Tabla VII Descripción del Módulo Locator

Modulo Locator	
Clases	
	BrickNotFoundError
	NoBackendError
Funciones	
	find_bricks(host=None, name=None)
	find_one_brick(host=None, name=None)
Variables	
	__package__ = **'**nxt**'**

Tabla VIII Descripción modulo motor

Módulo Motor	
Clases	
	Motor
Variables	
	PORT_A = 0
	PORT_B = 1
	PORT_C = 2
	PORT_ALL = 255
	MODE_IDLE = 0
	MODE_MOTOR_ON = 1
	MODE_BRAKE = 2
	MODE_REGULATED = 4
	REGULATION_IDLE = 0
	REGULATION_MOTOR_SPEED =1
	REGULATION_MOTOR_SYNC = 2
	RUN_STATE_IDLE = 0
	RUN_STATE_RAMP_UP = 16
	RUN_STATE_RUNNING = 32
	RUN_STATE_RAMP_DOWN =64
	LIMIT_RUN_FOREVER = 0

4.1.4. Paquetes en Python

Si los módulos sirven para organizar el código, los paquetes sirven para organizar los módulos. Los paquetes son tipos especiales de módulos (ambos son de tipo module) que permiten agrupar módulos relacionados. Mientras los módulos se corresponden a nivel físico con los archivos, los paquetes se representan mediante directorios [29].

Para hacer que Python trate a un directorio como un paquete es necesario crear un archivo __init__.py en dicha carpeta. En este archivo se define elementos que pertenezcan a dicho paquete, aunque habitualmente se trata de un archivo vacío. Para hacer que un cierto módulo se encuentre dentro de un paquete, basta con copiar el archivo que define el módulo al directorio del paquete. Como los módulos, para importar paquetes también se utiliza import y from-import y el caractér **.** para separar paquetes, subpaquetes y módulos [29].

```
import paq.subpaq.modulo
paq.subpaq.modulo.func()
```

Los paquetes se representan mediante directorios. Un ejemplo de cómo nombrar un módulo es: A.B donde se hace referencia a un submódulo denominado "B" de un paquete denominado "A". Ejemplo:

nxt.locator.find_one_brick()

Al importar el paquete, Python busca los directorios especificados en la variable **sys.path** buscando por el subdirectorio de paquetes. Los usuarios del paquete pueden importar módulos individuales del paquete, por ejemplo:

import test.moduloprueba.prueba1

De este modo se carga el submódulo test.moduloprueba.prueba1. Hay que hacer referencia a él por su nombre completo:

test.moduloprueba.prueba1.prueba1a(entrada, salida)

La sentencia **import** comprueba si el elemento está definido en el paquete. Si no, asume que es un módulo e intenta cargarlo. Si no lo consigue, se provoca una excepción como **ImportError**, de las cuales se describe más adelante.

Sin embargo, cuando se utiliza la sintaxis: **import** *elemento.subelemento.subsubelemento*, cada elemento menos el último debe ser un paquete. El último elemento puede ser un módulo o un paquete, pero no una clase, función o variable definida en el nivel superior.

También se importa elementos escribiendo: **from** nxt.motor **import** *

Aquí todo el sistema se rastrea para encontrar qué submódulos existen en el paquete y de esta manera importarlos todos.

4.1.4.1 Paquete NXT-Python

NXT_Python es *casi* como trabajar directamente en el brick, ya que no hay necesidad de compilar el programa ni cargarlo, solamente se ejecutan las instrucciones [32].

Ejemplo:

```
#!/usr/bin/env python
# importamos las funciones de la API
import nxt.locator
from nxt.sensor import *
sock = nxt.locator.find_one_brick()
if sock:
        brick = sock.connect()
        name, host, signal_strength, user_flash = brick.get_device_info()
        print 'NXT brick name: %s' % name
        print 'Host address: %s' % host
        print 'Bluetooth signal strength: %s' % signal_strength
        print 'Free user flash: %s' % user_flash
sock.close()
```

Lo que hace este código es buscar un brick, cuando lo encuentre obtendrá cierta información del mismo para mostrarla por pantalla.

NXT_Python permite controlar el Brick del NXT sin necesidad de modificar el firmware original de Lego. Por medio de una conexión Bluetooth o USB se puede comunicar con el brick por medio del PC y leer los valores de los sensores o manejar los servomotores.

De esta forma el código escrito es ejecutado directamente por el PC y se puede aprovechar toda la potencia de cálculo que este ofrece, así como la conexión con otros periféricos que complementan la labor del robot [33].

4.1.4.2. Contenido del Paquete NXT_Python

Esta sección es basada en su totalidad de la página oficial de ROS [34], contiene el API para la documentación del paquete nxt_python. Los objetos de Python que se definen en el proyecto se dividen en tablas separadas para cada paquete, módulo y clase. A continuación se describe el paquete nxt_python:

Tabla IX Contenido del Paquete NXT_PYTHON

Submódulos	Descripción
__init__.py	Se incluye para que un directorio sea considerado como paquete.

nxt.bluesock	Da origen a la comunicación Bluetooth con el NXT.
nxt.brick	Representa el brick del NXT.
nxt.direct	Usado para la comunicación directa con el NXT.
nxt.error	Declaraciones de errores.
nxt.locator	Encuentra un brick.
nxt.motor	Usado para controlar el motor.
nxt.sensor	Modulo para los sensores.
nxt.server	Se usa para un controlador de interfaz de socketNXT.
nxt.system	Se utiliza para las comunicaciones relativas al sistema de archivos NXT
nxt.telegram	Utilizado por nxt.system para enviar telegramas al NXT.
Variables	**Descripción**
__**package**__ = None	

4.1.5. Variables de Entorno

Hay muchas variables de entorno que se pueden establecer para modificar el comportamiento de ROS. De éstos, los más importante de entender son: ROS_MASTER_URI, ROS_ROOT y ROS_PACKAGE_PATH como comúnmente se usa en el sistema y frecuentemente se mencionan. Las variables de entorno cumplen diversos papeles en ROS:

- **Búsqueda de paquetes**: ROS_ROOT y ROS_PACKAGE_PATH habilitan a ROS para localizar paquetes y pilas en el sistema de archivos. También se debe establecer la PYTHONPATH para que el intérprete de Python puede encontrar bibliotecas de ROS.
- ***Da información en tiempo de ejecución:*** El ROS_MASTER_URI es una importante variable de entorno que le dice a un nodo donde el está el Maestro. ROS_IP y ROS_HOSTNAME hacen la dirección de red de un nodo y ROS_NAMESPACE le permite cambiar su espacio de nombres. ROS_LOG_DIR le permite establecer el directorio donde los archivos de registro se escriben. Muchos de estos pueden ser anulados por Reasignación de argumentos, así que tienen prioridad sobre las variables de entorno.

4.1.5.1. Variables de Entorno necesarias para ROS

La mayoría de los sistemas tienen que establecer ROS_PACKAGE_PATH, pero sólo las variables de entorno necesarias para ROS son ROS_ROOT, ROS_MASTER_URI y PYTHONPATH. Por defecto se ajusta automáticamente a través de la ruta: /opt/ros/fuerte/setup.bash.

- **ROS_ROOT:** establece la ubicación donde los paquetes básicos de ROS están instalados.
- **ROS_MASTER_URI:** Ajuste requerido que indica a los nodos donde se puede localizar al master.
- **PYTHONPATH:** ROS requiere que su PYTHONPATH sea actualizado, incluso si no se programa en Python. Muchas herramientas de infraestructura ROS confian en Python y necesitan tener acceso al paquete roslib para el arranque [35]. PYTHONPATH es una lista de nombres de directorios [36].

4.1.6. Excepciones

Las excepciones son errores detectados por Python durante la ejecución del programa. Cuando el intérprete se encuentra con una situación excepcional, como intentar dividir un número entre 0 o intentar acceder a un archivo que no existe, este genera o lanza una excepción,

informando al usuario de que existe algún problema. Si la excepción no se captura el flujo de ejecución se interrumpe y se muestra la información asociada a la excepción en la consola de forma que el programador pueda solucionar el problema.

A continuación en la Tabla X se listan a modo de referencia las excepciones disponibles por defecto, así como la clase de la que se deriva cada una de ellas entre paréntesis [29].

Tabla X Descripción de Excepciones

BaseException: Clase de la que heredan todas las excepciones.	
Exception(BaseException):	Super clase de todas las excepciones que no sean de salida.
GeneratorExit(Exception):	Se pide que se salga de un generador.
StandardError(Exception):	Clase base para todas las excepciones que no tengan que ver con salir del intérprete.
ArithmeticError(StandardError):	Clase base para los errores aritméticos.
	Error en una operación de coma

FloatingPointError(ArithmeticError):	flotante.
OverflowError(ArithmeticError):	Resultado demasiado grande para poder representarse.
ZeroDivisionError(ArithmeticError):	Lanzada cuando el segundo argumento de una operación de división o módulo era 0.
AssertionError(StandardError):	Falló la condición de un estamento assert.
AttributeError(StandardError):	No se encontró el atributo.
ImportError(StandardError):	No se encuentra el módulo o el elemento del módulo que se quería importar.
LookupError(StandardError):	Clase padre de los errores de acceso.
IndexError(LookupError):	El índice de la secuencia está fuera del rango posible.
KeyError(LookupError):	La clave no existe.
MemoryError(StandardError):	No queda memoria suficiente.
NameError(StandardError):	No se encontró ningún elemento con ese nombre.

4.2. Lenguaje de Programación C++

Para este proyecto en específico se uso el Lenguaje de Programación C++ para la creación de un nodo para el cual usamos la librería Video for Linux como herramienta para adquisición de imagen y video.

4.2.1. Video for Linux

Video4Linux es el nombre de la interface de software para la captura de video de Linux. Webcams, radio, tarjetas de captura y televisión, estas clases de dispositivos se agrupan en la categoría de captura de video por lo tanto es necesario usar una interface de programación escrita en lenguaje C para tener acceso a estos dispositivos conocida como v4l. Por la naturaleza de estos dispositivos, la abstracción de dispositivos de Unix/Linux tuvo que ser un poco extendida. Normalmente se puede acceder a estos dispositivos usando la función open() para luego escribir/ leer algunos bytes.

Los dispositivos de video que se usan en este proyecto necesitan leer/escribir datos en ellos para lo cual se utiliza la función **ioctl()** la cual puede tener muchos comandos de configuración para tratar con algunas

opciones. Existen diferentes métodos para leer/ escribir datos en un dispositivo:

Read/Write es el método clásico y seleccionado automáticamente usa las funciones read() y write(), si se quiere usar otros métodos deben ser negociados.

Memory Mapping o método mapeo de memoria se usa en esta aplicación que este consiste en el intercambio de punteros a buffers entre la aplicación y el driver, el dato en sí mismo no es copiado. Se asigna buffers de memoria del dispositivo al espacio de direcciones de la aplicación.

Ejemplo:

```
void init_mmap(void)
{
        struct v4l2_requestbuffers req;

        CLEAR(req);

        req.count = 4;
        req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        req.memory = V4L2_MEMORY_MMAP;

        if (-1 == xioctl(fd, VIDIOC_REQBUFS, &req)) {
                if (EINVAL == errno) {
```

```
                fprintf(stderr, "%s does not support "
                        "memory mapping\n", dev_name);
                exit(EXIT_FAILURE);
        } else {
                errno_exit("VIDIOC_REQBUFS");
        }
}

if (req.count < 2) {
        fprintf(stderr, "Insufficient buffer memory on %s\n",
                 dev_name);
        exit(EXIT_FAILURE);
}

buffers =(buffer*) calloc(req.count, sizeof(*buffers));

if (!buffers) {
        fprintf(stderr, "Out of memory\n");
        exit(EXIT_FAILURE);
}

for (n_buffers = 0; n_buffers < req.count; ++n_buffers) {
        struct v4l2_buffer buf;

        CLEAR(buf);

        buf.type        = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        buf.memory      = V4L2_MEMORY_MMAP;
        buf.index       = n_buffers;
```

```
if (-1 == xioctl(fd, VIDIOC_QUERYBUF, &buf))
        errno_exit("VIDIOC_QUERYBUF");

buffers[n_buffers].length = buf.length;
buffers[n_buffers].start =
        mmap(NULL /* start anywhere */,
             buf.length,
             PROT_READ | PROT_WRITE /* required */,
             MAP_SHARED /* recommended */,
             fd, buf.m.offset);
if (MAP_FAILED == buffers[n_buffers].start)
        errno_exit("mmap");
```

User pointers o punteros de usuario combina las ventajas de los métodos read/write y memory mapping. Los buffers son asignados por la aplicación misma y puede residir en memoria virtual o compartida. Solo los punteros a datos son intercambiados. Existen otros métodos pero estos o no están definidos o son experimentales tales como DMA buffers y asynchronous I/O.

4.2.1.1. Estructura general del algoritmo de captura de imágenes usando Video for Linux

Para extraer imágenes utilizando v4l organizamos el algoritmo de la siguiente forma:

- **Abrir el dispositivo.-** Consiste en utilizar la función open() para obtener el descriptor de archivo del dispositivo conectado.

Ejemplo:

```
void open_device(void)
{
        struct stat st;

        if (-1 == stat(dev_name, &st)) {
                fprintf(stderr, "Cannot identify '%s': %d, %s\n",
                        dev_name, errno, strerror(errno));
                exit(EXIT_FAILURE);
        }

}
```

- **Inicializar el dispositivo.-** Se asigna la configuración en el driver para poder obtener las imágenes en el formato que deseamos (debido a que se usa una cámara web el formato usado para el procesamiento de cada imagen es YUYV o YUV422). Aquí también se realiza el mapeo de memoria para leer/escribir en el dispositivo.

- **Iniciar la captura de imágenes.-** Esto inicia la transferencia de las imágenes desde el dispositivo hacia la memoria para el procesamiento posterior.

- **Extracción y procesamiento de cada imagen.-** Esta parte se explica en la siguiente sección.

- **Liberar memoria.-** En esta parte se usa la función munmap() para liberar toda la memoria requerida para el programa.

Ejemplo:

```
void uninit_device(void)
{
        unsigned int i;

         for (i = 0; i < n_buffers; ++i)
                        if (-1 == munmap(buffers[i].start, buffers[i].length))
                                errno_exit("munmap");

        free(buffers);
}
```

- **Cerrar dispositivo.-** Consiste en usar la función close() para liberar el descriptor de archivo del dispositivo conectado.

  Ejemplo:

```
void close_device(void)
{
        if (-1 == close(fd))
                errno_exit("close");

        fd = -1;
}
```

4.2.1.2. Extracción y procesamiento de cada imagen usando Video for Linux

La mayoría de las cámaras web tienen dos formatos de video MJPEG (Comprimido) y YUYV (Descomprimido conocido también como YUV422).

Puesto que YUV422 es un formato descomprimido facilita las operaciones de procesamiento de la imagen y transformación al formato RGB.

El propósito de procesamiento es extraer el canal rojo de cada imagen por lo tanto es necesaria la transformación al formato RGB como se explica a continuación.

YUV422 a RGB

En video4linux el formato YUV422 está conformado por 2 pixeles cada 4 bytes, en cada 4 bytes hay dos Y's, un Cb y un Cr, donde Y es la información para el brillo, Cb es la información para el azul y Cr es la información para el rojo, el canal que queremos extraer. A continuación un ejemplo de cómo están conformados los datos de una imagen de 4x4 pixeles, donde start es el origen de los datos.

start + 0: $Y'_{00}\ Cb_{00}\ Y'_{01}\ Cr_{00}\ Y'_{02}\ Cb_{01}\ Y'_{03}\ Cr_{01}$

start + 8: $Y'_{10}\ Cb_{10}\ Y'_{11}\ Cr_{10}\ Y'_{12}\ Cb_{11}\ Y'_{13}\ Cr_{11}$

start + 16: $Y'_{20}\ Cb_{20}\ Y'_{21}\ Cr_{20}\ Y'_{22}\ Cb_{21}\ Y'_{23}\ Cr_{21}$

start + 24: $Y'_{30}\ Cb_{30}\ Y'_{31}\ Cr_{30}\ Y'_{32}\ Cb_{31}\ Y'_{33}\ Cr_{31}$

Debido que el formato YUV422 tiene dos pixeles cada 4 bytes con dos posiciones para el brillo y el formato RGB tiene 1 pixel cada tres bytes (un byte por canal), para hacer una correcta transformación YUV-RGB se debe utilizar cada componente de brillo(Y) con los mismos componentes de Azul(Cb) y Rojo(Cr) de los 4 bytes(YUV) para obtener dos pixeles en RGB, así por cada 4 bytes en YUV se obtienen 6 bytes en RGB.

Para transformar un pixel YUV en un pixel RGB se implementó la función pix_yuv422_to_rgb24, esta función recibe como parámetros los componentes en YUV y retorna los componentes en RGB la firma de la función es la siguiente:

void pix_yuv422_to_rgb24(unsigned char y, unsigned char u, unsigned char v, unsigned char* r, unsigned char* g, unsigned char* b)

El Proceso de trasformación de cada componente es el siguiente:

$$R = 1.164 * (y - 16) + 1.596 * (Cr - 128)$$

$$G = 1.164 * (y - 16) - 0.813 * (Cr -128) - 0.391 * (Cb -128)$$

$$B = 1.164 * (y -16) + 2.018 * (Cb -128)$$

Esta transformación YUV-RGB es estándar y se la puede encontrar en cualquier texto de procesamiento de imágenes o en internet.

Así usando esta función se puede extraer solo el canal rojo de una imagen aprovechando el tamaño original de la imagen ya que el formato YUV solo tiene una componente rojo cada 2 pixeles.

CAPITULO 5

5. PROYECTO LASER SCANNER

En el presente capítulo se explica el proyecto "Láser Scanner". Además se describe el programa utilizado con imágenes que resultaron de la realización y estudio del proyecto usando ROS

5.1. Descripción General del Proyecto

Láser Scanner tiene como objetivo escanear un objeto el cual se encuentra en cierta posición para que un láser lo ilumine de arriba hacia abajo para poder tomar una foto del mismo. Cuando el laser llega a su punto más bajo, volverá a subir para hacer girar al objeto y repetir la secuencia de arriba hacia abajo hasta que el objeto haya girado por completo y vuelva a su posición inicial. Las fotos se guardan dentro de una carpeta en la ruta especificada dentro del código.

5.2. Objetivos del Proyecto

- Investigar la plataforma robótica ROS, para que pueda ser introducida en la materia de fundamentos de robótica que se dicta en la Facultad.
- Implementar un Laser Scanner con Lego NXT y la plataforma ROS.
- Desarrollar el proyecto de una forma eficiente mediante la aplicación de diferentes lenguajes de programación según convenga.
- Integrar en el proyecto, los conocimientos obtenidos a lo largo de nuestra carrera universitaria.
- Fomentar la plataforma robótica ROS en centros de educación y área industrial, debido a que es de libre distribución.

5.3. Configuración del Entorno ROS

A continuación pasos necesarios para crear en proyecto en ROS. Desde la creación del Espacio de Trabajo, paquetes, dependencias, nodos, etc.

5.3.1. Creación de un Espacio de Trabajo

Se crea un lugar dentro de ROS, para este proyecto lo llamaremos “mythesis”. Usamos el siguiente comando:

```
rosws init ~/mythesis /opt/ros/electric
```

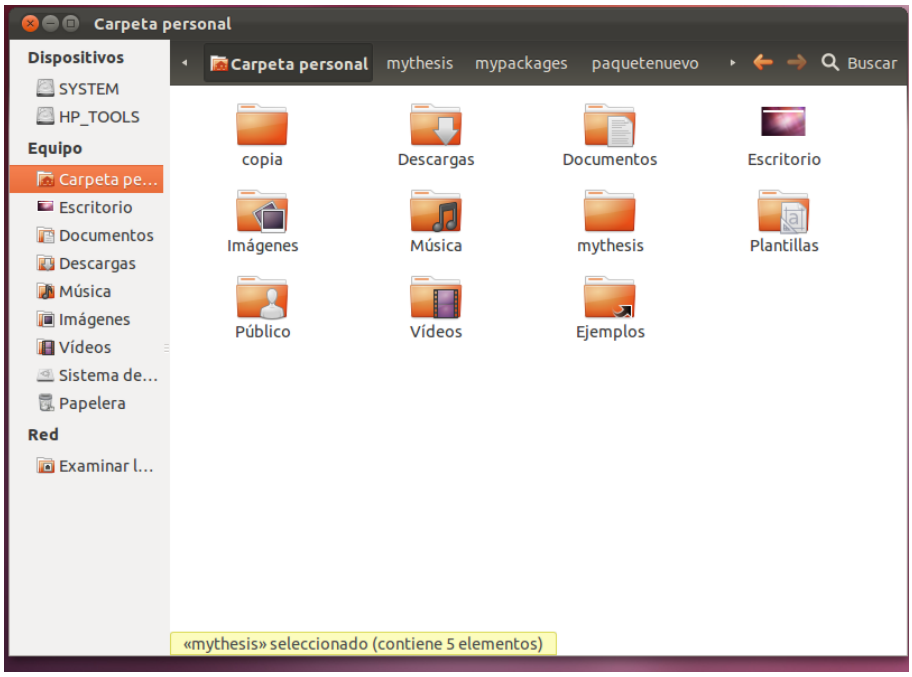


Figura 5.1 Creación del Espacio de Trabajo en ROS

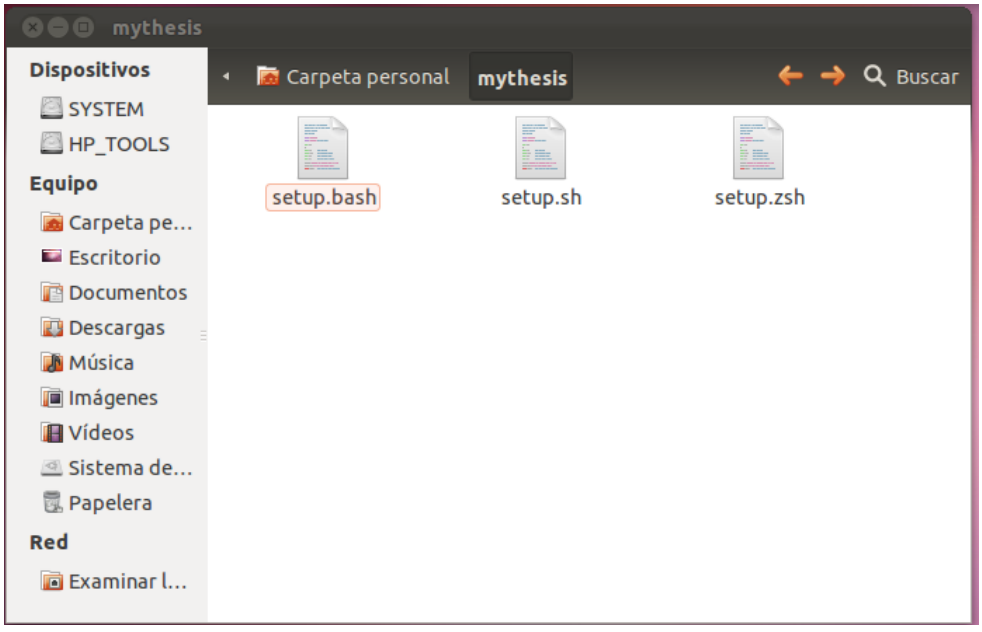


Figura 5.2 Contenido del Espacio de Trabajo

5.3.2. Creación de un directorio para los nuevos paquetes

Se usó el siguiente comando para crear un directorio donde se guardan los nuevos paquetes dentro del espacio de trabajo creado previamente.

```
mkdir ~ / mythesis / mypackages
rosws set ~ / mythesis / mypackages
```

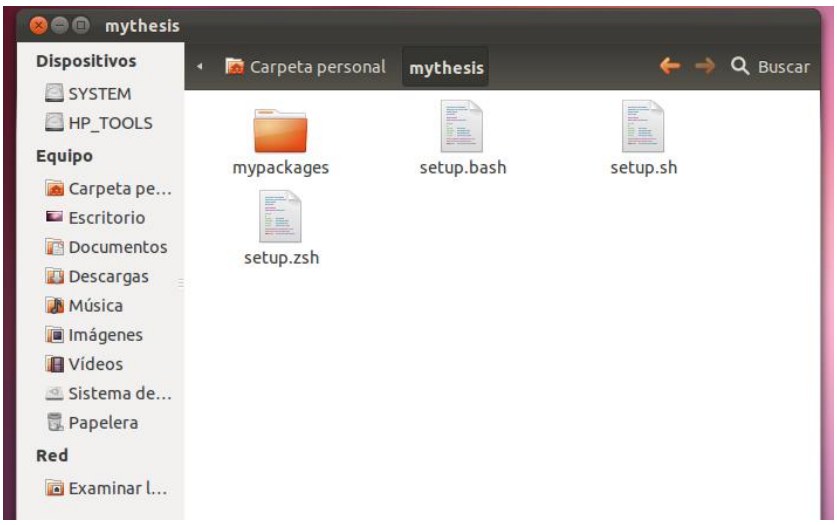


Figura 5.3 Creación de un directorio para los paquetes

No olvidar hacer el siguiente comando cada que se necesite crear algo dentro del espacio de trabajo:

```
#source /home/mafer/mythesis
```

5.3.3. Creación de un Paquete

Para crear el paquete debemos ingresar a la ruta destinada al paquete, que es la siguiente:

```
    cd ~ /mythesis/mypackages
# roscreate-pkg paquetenuevo
# roscreate-pkg paquetenuevo std_msgs rospy roscpp
rosmake paquetenuevo
```

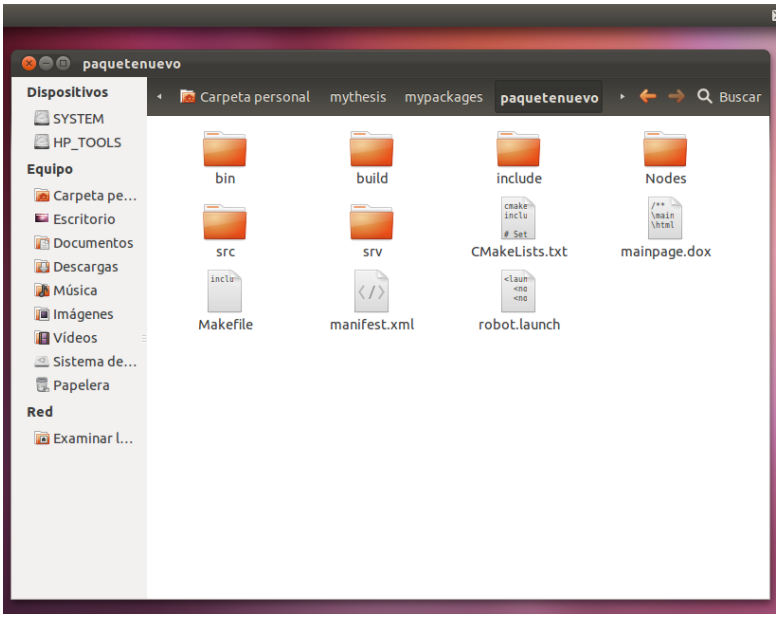


Figura 5.4 Creación del paquete en ROS

Para mejor organización del proyecto se creó una carpeta **Nodes**, la cual contiene los archivos ejecutables de los nodos necesarios para este proyecto.

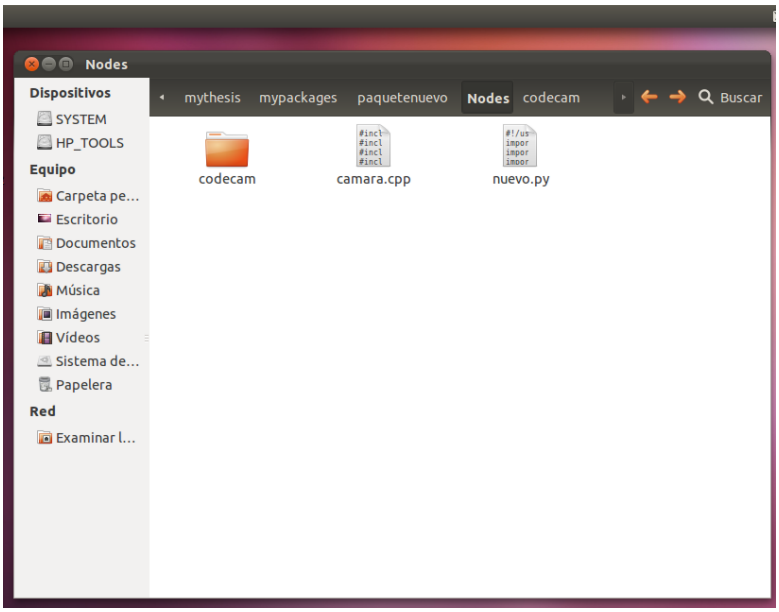


Figura 5.5 Creación de Carpeta para los Nodos

5.4. Descripción Técnica del Proyecto

Láser Scanner de acuerdo a la teoría de ROS consta de 3 Nodos los cuales se describen a continuación:

- **Nodo Motores**: Este nodo está encargado de controlar los motores que se usan en este proyecto, el cual usa como Lenguaje de Programación Python y tiene la característica de ser un Nodo Publicador. Para este proyecto el nodo publicador llamado “NodoMotores” da una orden a otro nodo para que tome una foto, al cual llamaremos “NodoCamara”.
- **Nodo Cámara:** Este nodo está encargado de controlar una cámara web. Usa C++ como Lenguaje de Programación y tiene como

característica principal ser un Nodo Subscriptor, dentro del código denominado “NodoCamara”. El cual ejecuta su código solo cuando recibe la orden del NodoMotores, para que tome una foto por cada vez que se recibe esta orden. Las imágenes se guardan en una carpeta dentro de nuestro espacio, la cual está definida y puede ser modificada dentro del código del “NodoCamara”.

- **Nodo rosout:** El Nodo rosout siempre se encuentra activo ya que este es el encargado de suscribir, registrar y publicar los mensajes. Este nodo se crea automáticamente al ejecutar el nodo principal roscore, que permite la comunicación entre los nodos de ROS, el cual se ejecuta al inicio de trabajar con ROS.

El NodoMotores está configurado para que controle dos motores que son:

- **Motor Base:** Motor Base está encargado de controlar el giro del objeto a escanear. Está programado para girar la base del objeto 90º cada vez que el Motor Láser haya concluido su ejecución de arriba hacia abajo. Dentro de la estructura del código se identifica al motor base bajo el nombre: m_base.
- **Motor Láser:** Motor Láser se encarga de controlar el láser. El cual está en una posición inicial y a partir de esta comienza a girar el motor para que el laser vaya iluminando de arriba hacia abajo el objeto. Para

este proyecto se programó al motor del láser para que gire cada 15º. Dentro del código se puede identificar el motor del láser bajo el nombre: m_ laser.

Los Nodos que intervienen en la realización del proyecto tienen características bien definidas y funcionan de la siguiente manera:

El NodoMotores envía una publicación definida por un “Topic” al cual denominaremos “foto”. Esto recibe el NodoCamara, el que ya tenemos programado para que solo se subscriba al Topic “foto” y ejecute el código escrito que toma una foto cada vez que reciba el “Topic”. Logrando con esto una comunicación sincronizada entre ambos nodos. El proceso concluye cuando el Motor Base haya vuelto a su posición inicial.

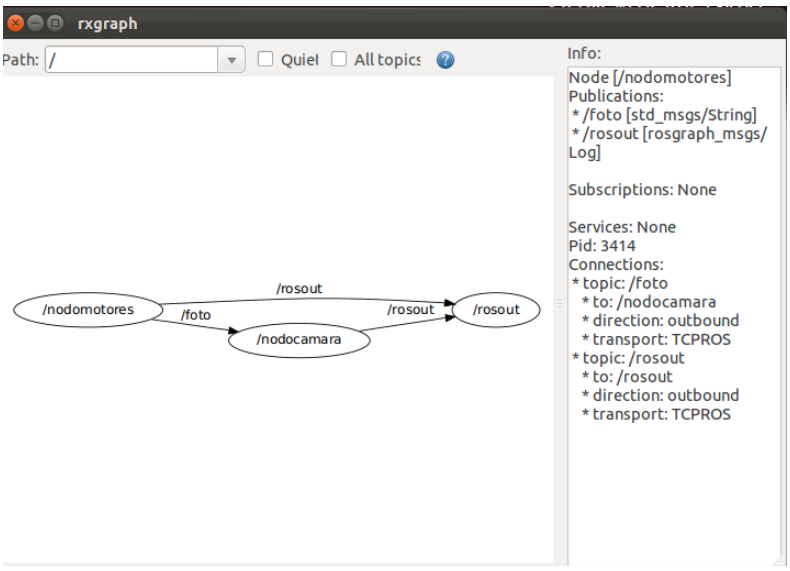


Figura 5.6 Comunicación de Nodos ejecutándose I.

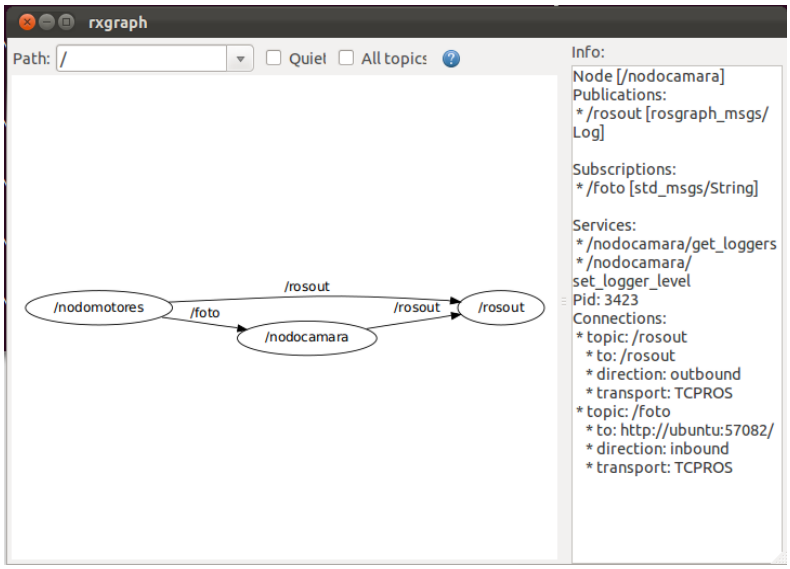


Figura 5.7 Comunicación de Nodos ejecutándose II.

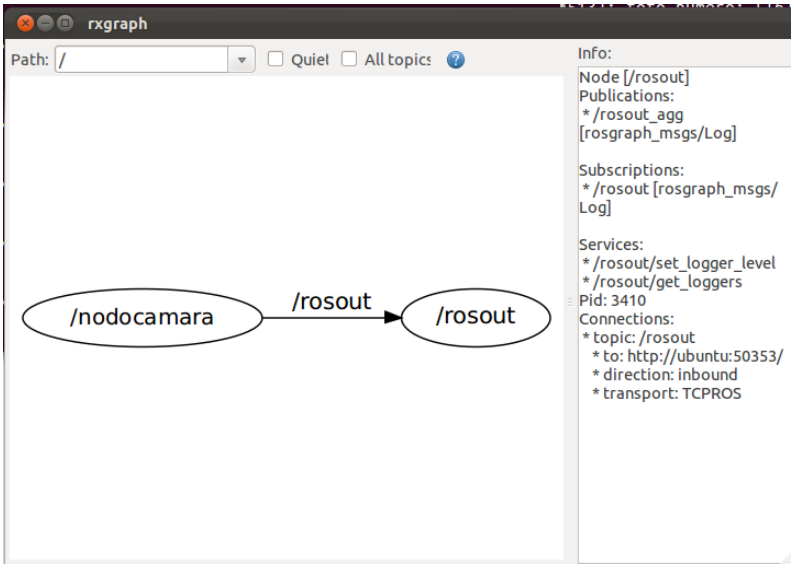


Figura 5.8 Comunicación de Nodos cuando no existe un Nodo Publicador

5.5. Recursos Físicos para Láser Scanner

Para la realización de este proyecto se usaron los siguientes elementos:

- 1 Brick
- 1 Cámara web
- 1 Láser lineal
- 2 Servo motores del Kit Lego Mindstorm NXT
- Piezas varias del Kit Lego Mindstorm NXT

Es necesario armar una infraestructura para ubicar el láser, la cámara y el objeto a escanear. Para nuestro proyecto la armamos de la siguiente manera aunque se la puede adaptar a las necesidades de cada proyecto. A continuación imágenes de la infraestructura:

Figura 5.9 Vista completa del Proyecto Láser Scanner

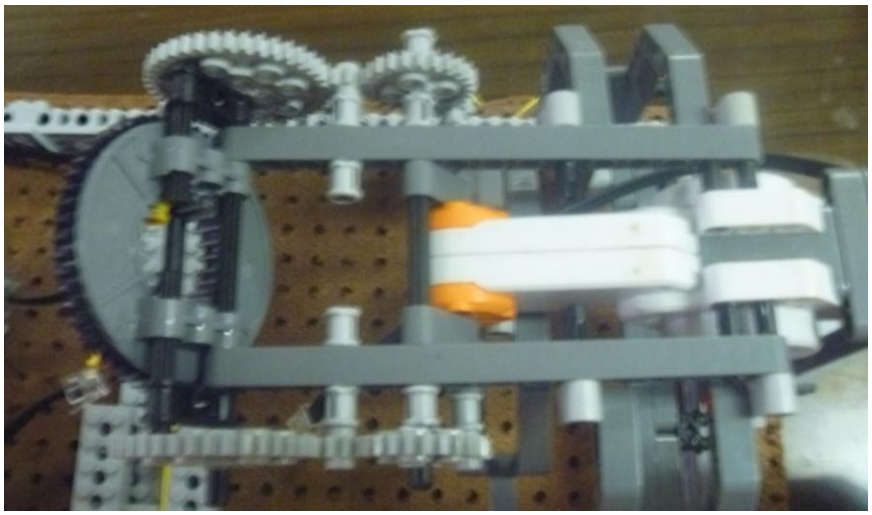


Figura 5.10 Vista superior del sujetador del Láser

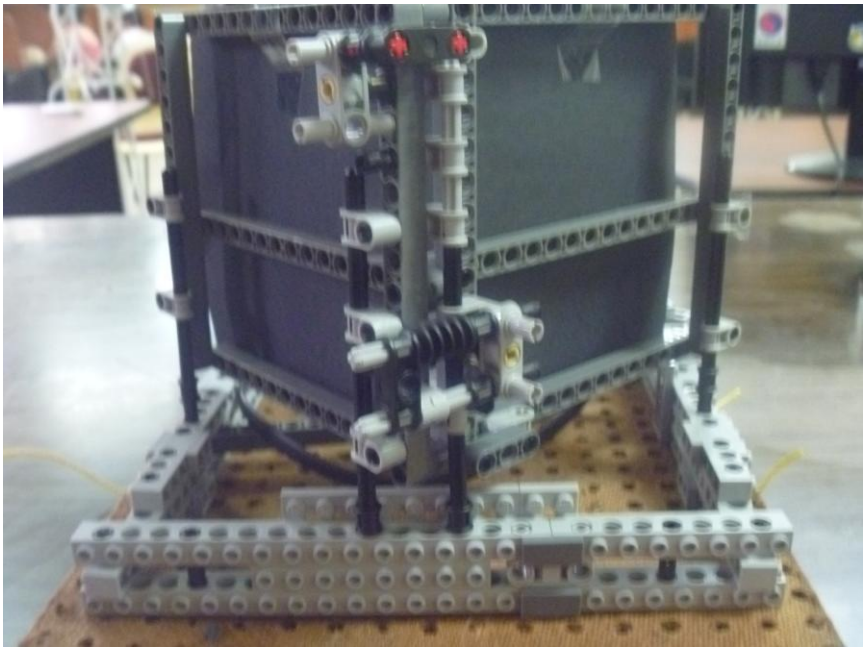


Figura 5.11 Vista trasera

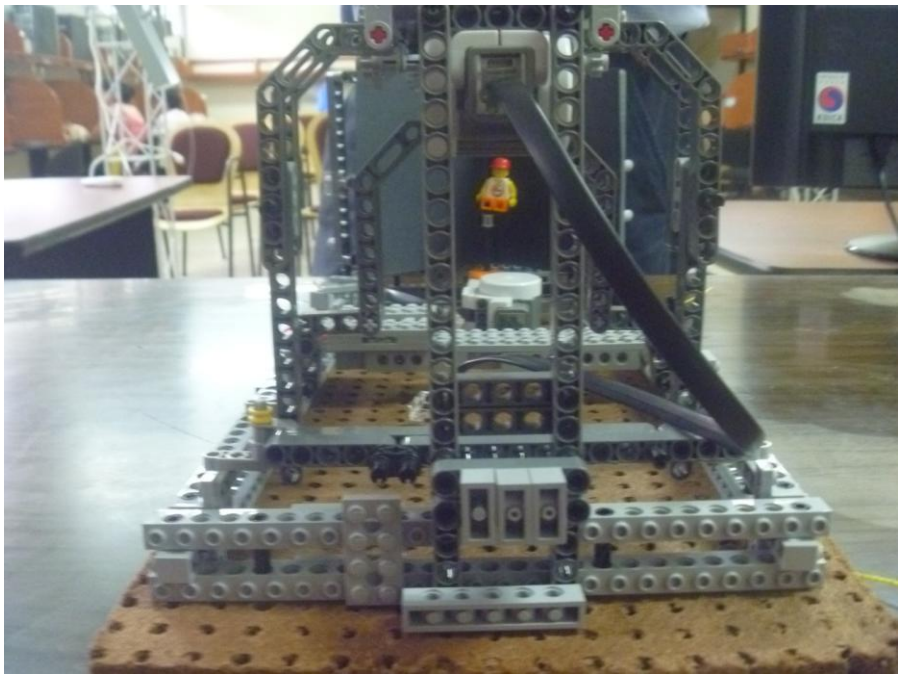


Figura 5.12 Vista Lateral

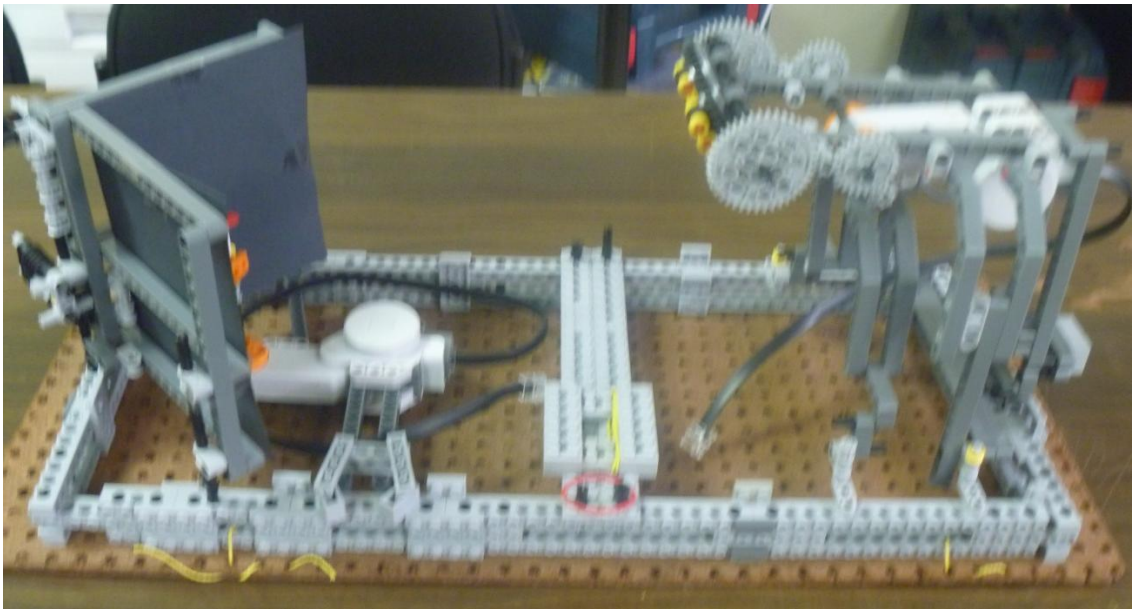


Figura 5.13 Vista Superior#1

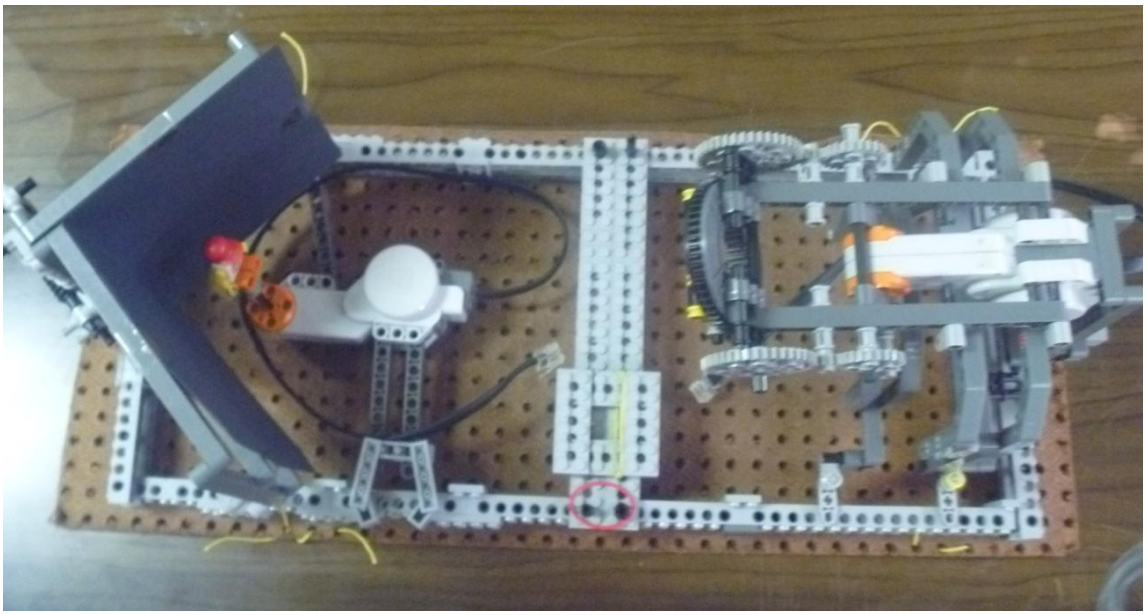


Figura 5.14 Vista Superior#2

5.6. Código del Nodo Publicador en Lenguaje Python

```
#!/usr/bin/env python
import roslib; roslib.load_manifest('paquetenuevo')
import rospy
import nxt.locator
from std_msgs.msg import String
from nxt.motor import *

def spin_around(b):
    rospy.init_node('NodoCamara')
    pub = rospy.Publisher('foto', String)
    m_laser = Motor(b, PORT_B)
    m_base = Motor(b, PORT_C)
```

```
        acu = 0
        cont = 0
        x = 15
        for s in range(0,4):
         for t in range(0,5):
            str = "l%s" % cont
            rospy.sleep(2.0)
            pub.publish(String(str))
            m_laser.turn(127,x,brake=True, timeout=1, emulate=True)
            acu = acu + x;
            if acu == 75:
             m_laser.turn(-127,acu-5,brake=True, timeout=1, emulate=True)
             acu = 0
            cont = cont+1
           m_base.turn(7,90)
    b = nxt.locator.find_one_brick()
    spin_around(b)
```

5.7. Código del Nodo Subscriptor en Lenguaje C++

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include<cstring>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <getopt.h>            /* getopt_long() */
```

```
#include <fcntl.h>              /* low-level i/o */
#include <unistd.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include <linux/videodev2.h>
#define CLEAR(x) memset(&(x), 0, sizeof(x))

struct buffer {
        void   *start;
        size_t length;
};

static char            dev_name[256];
static int             fd = -1;
struct buffer          *buffers;
static unsigned int     n_buffers;
static int             out_buf;
static int             frame_count = 5;

void errno_exit(const char *s);
int xioctl(int fh, int request, void *arg);
void pix_yuv422_to_rgb24(unsigned char y, unsigned char u, unsigned char v,
                         unsigned char* r, unsigned char* g, unsigned char* b);
void img_yuv422_to_rgb24(void *in_yuv422, unsigned char *out_rgb24, int width,int
height);
void img_yuv422_to_Rchannel(void *in_yuv422, unsigned char *out_Rchannel, int
```

```
width,int height);
int read_frame(char *out_name);
void start_capturing(void);
void uninit_device(void);
void init_read(unsigned int buffer_size);
void init_mmap(void);
void init_device(void);
void open_device(void);
void close_device(void);

void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  sprintf(dev_name, "/dev/video1");
          unsigned int count;
       count = frame_count;
          char     name[256];

       open_device();
       init_device();
       start_capturing();

                    fd_set fds;
                    struct timeval tv;
                    int r;
                    FD_ZERO(&fds);
                    FD_SET(fd, &fds);

                    /* Timeout. */
                    tv.tv_sec = 2;
                    tv.tv_usec = 0;
```

```
                r = select(fd + 1, &fds, NULL, NULL, &tv);
                if (-1 == r) {
                        fprintf(stderr, "select failure\n");
                        exit(EXIT_FAILURE);
                }
                if (0 == r) {
                        fprintf(stderr, "select timeout\n");
                        exit(EXIT_FAILURE);
                }
                          sprintf(name,
"/home/mafer/mythesis/mypackages/paquetenuevo/Nodes/codecam/outs%s",msg-
>data.c_str());

                read_frame(name);

 ROS_INFO("foto numero: [%s]", msg->data.c_str());

        uninit_device();
        close_device();

}

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

  ros::NodeHandle n;

  ros::Subscriber sub = n.subscribe("foto", 1000, chatterCallback);
```

```
  ros::spin();

  return 0;
}


void errno_exit(const char *s)
{
        fprintf(stderr, "%s error %d, %s\n", s, errno, strerror(errno));
        exit(EXIT_FAILURE);
}

int xioctl(int fh, int request, void *arg)
{
        int r;

        do {
                r = ioctl(fh, request, arg);
        } while (-1 == r && EINTR == errno);

        return r;
}
void pix_yuv422_to_rgb24(unsigned char y, unsigned char u, unsigned char v,
                                unsigned char* r, unsigned char* g, unsigned char* b)
{

           double R,G,B;

       B = 1.164 * (y -16)+ 2.018 * (u -128);
       G = 1.164 * (y - 16)  -  0.813 * (v -128)- 0.391 * (u -128);
```

```
R = 1.164 * (y - 16)  +  1.596 * (v - 128);

    //R, G and B must be in the range from 0 to 255
    if (R < 0)
        R=0;
    if (G < 0)
        G=0;
    if (B < 0)
        B=0;

    if (R > 255)
        R=255;
    if (G > 255)
        G=255;
    if (B > 255)
        B=255;

    *r=(unsigned char)(R);
    *g=(unsigned char)(G);
    *b=(unsigned char)(B);


}
void img_yuv422_to_rgb24(void *in_yuv422, unsigned char *out_rgb24, int width,int height){

    int i,yuvidx,rgbidx;
    unsigned char Y,U,V;
    unsigned char R,G,B;
    for(i=0; i<height*width/2; i++){
```

```
                yuvidx=i*4;
                rgbidx=i*6;

                Y=*((unsigned char*)in_yuv422+yuvidx);
                U=*((unsigned char*)in_yuv422+yuvidx+1);
                V=*((unsigned char*)in_yuv422+yuvidx+3);
                pix_yuv422_to_rgb24(Y, U, V, &R, &G, &B);

                *(out_rgb24+rgbidx)=R;
                *(out_rgb24+rgbidx+1)=G;
                *(out_rgb24+rgbidx+2)=B;



                Y=*((unsigned char*)in_yuv422+yuvidx+2);
                pix_yuv422_to_rgb24(Y, U, V, &R, &G, &B);
                *(out_rgb24+rgbidx+3)=R;
                *(out_rgb24+rgbidx+4)=G;
                *(out_rgb24+rgbidx+5)=B;

        }
}

void img_yuv422_to_Rchannel(void *in_yuv422, unsigned char *out_Rchannel, int
width,int height){

        int i,yuvidx;
        unsigned char Y,U,V;
        unsigned char R,G,B;
        for(i=0; i<height*width/2; i++){
                yuvidx=i*4;
```

```
                Y=*((unsigned char*)in_yuv422+yuvidx);
                U=*((unsigned char*)in_yuv422+yuvidx+1);
                V=*((unsigned char*)in_yuv422+yuvidx+3);
                pix_yuv422_to_rgb24(Y, U, V, &R, &G, &B);

                *(out_Rchannel+i*2)=R;

                Y=*((unsigned char*)in_yuv422+yuvidx+2);
                pix_yuv422_to_rgb24(Y, U, V, &R, &G, &B);
                *(out_Rchannel+i*2)=R;
        }
}

int read_frame(char *out_name)
{
        struct v4l2_buffer buf;
        unsigned int i;
           int width=640;
           int height=480;
           FILE    *fout_rgb,*fout_Rchannel;
           unsigned char *out_rgb,*out_Rchannel;

        CLEAR(buf);

              buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
              buf.memory = V4L2_MEMORY_MMAP;

              if (-1 == xioctl(fd, VIDIOC_DQBUF, &buf)) {
                    switch (errno) {
                    case EAGAIN:
```

```
                return 0;
            case EIO:
                /* Could ignore EIO, see spec. */

                /* fall through */

            default:
                errno_exit("VIDIOC_DQBUF");
            }
        }

        assert(buf.index < n_buffers);
            strcat(out_name,".pgm");
            fout_Rchannel = fopen(out_name, "w");
            if (!fout_Rchannel) {
                perror("Cannot open image");
                exit(EXIT_FAILURE);
            }
            out_Rchannel= (unsigned char *)malloc(width*height*sizeof(unsigned
char));
            fprintf(fout_Rchannel, "P5\n%d %d 255\n",width, height);
            img_yuv422_to_Rchannel(buffers[buf.index].start,out_Rchannel,
width,height);
            fwrite(out_Rchannel,1,width*height,fout_Rchannel);
            free(out_Rchannel);
            fclose(fout_Rchannel);

        if (-1 == xioctl(fd, VIDIOC_QBUF, &buf))
            errno_exit("VIDIOC_QBUF");
    return 1;
```

```
}

void start_capturing(void)
{
        unsigned int i;
        enum v4l2_buf_type type;

        for (i = 0; i < n_buffers; ++i) {
                struct v4l2_buffer buf;

                CLEAR(buf);
                buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                buf.memory = V4L2_MEMORY_MMAP;
                buf.index = i;

                if (-1 == xioctl(fd, VIDIOC_QBUF, &buf))
                        errno_exit("VIDIOC_QBUF");
        }
        type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        if (-1 == xioctl(fd, VIDIOC_STREAMON, &type))
                errno_exit("VIDIOC_STREAMON");
}

void uninit_device(void)
{
        unsigned int i;
         for (i = 0; i < n_buffers; ++i)
                        if (-1 == munmap(buffers[i].start, buffers[i].length))
                                errno_exit("munmap");
```

```
        free(buffers);
}

void init_read(unsigned int buffer_size)
{
        buffers = (buffer*)calloc(1, sizeof(*buffers));
        if (!buffers) {
                fprintf(stderr, "Out of memory\n");
                exit(EXIT_FAILURE);
        }

        buffers[0].length = buffer_size;
        buffers[0].start = malloc(buffer_size);
        if (!buffers[0].start) {
                fprintf(stderr, "Out of memory\n");
                exit(EXIT_FAILURE);
        }
}

void init_mmap(void)
{
        struct v4l2_requestbuffers req;

        CLEAR(req);

        req.count = 4;
        req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        req.memory = V4L2_MEMORY_MMAP;

        if (-1 == xioctl(fd, VIDIOC_REQBUFS, &req)) {
```

```
        if (EINVAL == errno) {
                fprintf(stderr, "%s does not support "
                        "memory mapping\n", dev_name);
                exit(EXIT_FAILURE);
        } else {
                errno_exit("VIDIOC_REQBUFS");
        }
}

if (req.count < 2) {
        fprintf(stderr, "Insufficient buffer memory on %s\n",
                 dev_name);
        exit(EXIT_FAILURE);
}

buffers =(buffer*) calloc(req.count, sizeof(*buffers));
if (!buffers) {
        fprintf(stderr, "Out of memory\n");
        exit(EXIT_FAILURE);
}

for (n_buffers = 0; n_buffers < req.count; ++n_buffers) {
        struct v4l2_buffer buf;

        CLEAR(buf);

        buf.type        = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        buf.memory      = V4L2_MEMORY_MMAP;
        buf.index       = n_buffers;
```

```
            if (-1 == xioctl(fd, VIDIOC_QUERYBUF, &buf))
                errno_exit("VIDIOC_QUERYBUF");
            buffers[n_buffers].length = buf.length;
            buffers[n_buffers].start =
                mmap(NULL /* start anywhere */,
                    buf.length,
                    PROT_READ | PROT_WRITE /* required */,
                    MAP_SHARED /* recommended */,
                    fd, buf.m.offset);

            if (MAP_FAILED == buffers[n_buffers].start)
                errno_exit("mmap");
        }
}



void init_device(void)
{
        struct v4l2_capability cap;
        struct v4l2_cropcap cropcap;
        struct v4l2_crop crop;
        struct v4l2_format fmt;
           struct v4l2_fmtdesc vid_fmtdesc;
        unsigned int min;
        if (-1 == xioctl(fd, VIDIOC_QUERYCAP, &cap)) {
            if (EINVAL == errno) {
                fprintf(stderr, "%s is no V4L2 device\n",
                        dev_name);
```

```
                exit(EXIT_FAILURE);
        } else {
                errno_exit("VIDIOC_QUERYCAP");
        }
}

if (!(cap.capabilities & V4L2_CAP_VIDEO_CAPTURE)) {
        fprintf(stderr, "%s is no video capture device\n",
                 dev_name);
        exit(EXIT_FAILURE);
}


/* Select video input, video standard and tune here. */
CLEAR(cropcap);

cropcap.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

if (0 == xioctl(fd, VIDIOC_CROPCAP, &cropcap)) {
        crop.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        crop.c = cropcap.defrect; /* reset to default */

        if (-1 == xioctl(fd, VIDIOC_S_CROP, &crop)) {
                switch (errno) {
                case EINVAL:
                        /* Cropping not supported. */
                        break;
                default:
                        /* Errors ignored. */
                        break;
```

```
                }
            }
        } else {
            /* Errors ignored. */
        }

        CLEAR(fmt);
           fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
           if (-1 == xioctl(fd, VIDIOC_G_FMT, &fmt))
               errno_exit("VIDIOC_G_FMT");

        fmt.fmt.pix.width       = 640;
           fmt.fmt.pix.height      = 480;
        fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;
        fmt.fmt.pix.field       = V4L2_FIELD_INTERLACED;

        if (-1 == xioctl(fd, VIDIOC_S_FMT, &fmt))
            errno_exit("VIDIOC_S_FMT");

        /* Buggy driver paranoia. */
        min = fmt.fmt.pix.width * 2;
        if (fmt.fmt.pix.bytesperline < min)
            fmt.fmt.pix.bytesperline = min;
        min = fmt.fmt.pix.bytesperline * fmt.fmt.pix.height;
        if (fmt.fmt.pix.sizeimage < min)
            fmt.fmt.pix.sizeimage = min;
        init_mmap();
}

void close_device(void)
```

```
{
        if (-1 == close(fd))
                errno_exit("close");

        fd = -1;
}

void open_device(void)
{
        struct stat st;
        if (-1 == stat(dev_name, &st)) {
                fprintf(stderr, "Cannot identify '%s': %d, %s\n",
                        dev_name, errno, strerror(errno));
                exit(EXIT_FAILURE);
        }

        if (!S_ISCHR(st.st_mode)) {
                fprintf(stderr, "%s is no device\n", dev_name);
                exit(EXIT_FAILURE);
        }

        fd = open(dev_name, O_RDWR /* required */ | O_NONBLOCK, 0);

        if (-1 == fd) {
                fprintf(stderr, "Cannot open '%s': %d, %s\n",
                        dev_name, errno, strerror(errno));
                exit(EXIT_FAILURE);
        }
}
```

5.8. Resultados del Proyecto

A continuación se muestra las imágenes que resultaron de la compilación del Proyecto Láser Scanner.

Figura 5.15 Vista Proyecto funcionando Objeto#1

Figura 5.16 Vista Proyecto funcionando Objeto#2

Imágenes resultantes de la comunicación entre Nodos con Objeto#1

Primera vuelta del motor de la base (Giro de 0º)

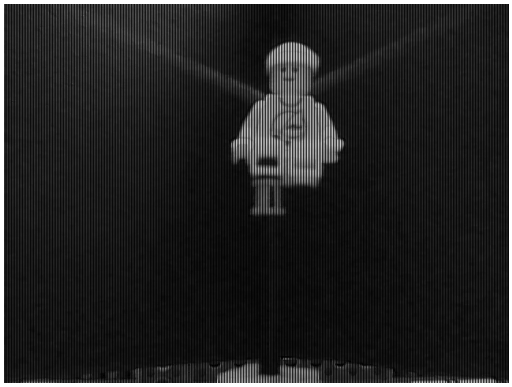


Figura 5.17 O1V1 m_laser 15º

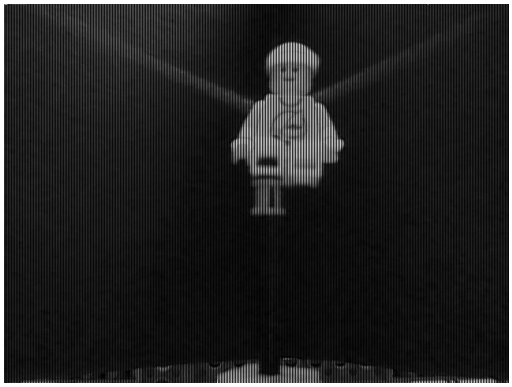


Figura 5.18 O1V1 m_laser 30º

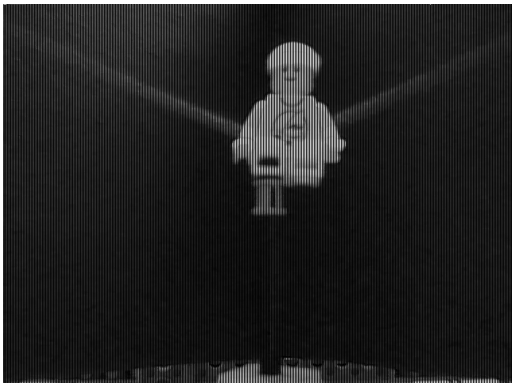


Figura 5.19 O1V1 m_laser 45º

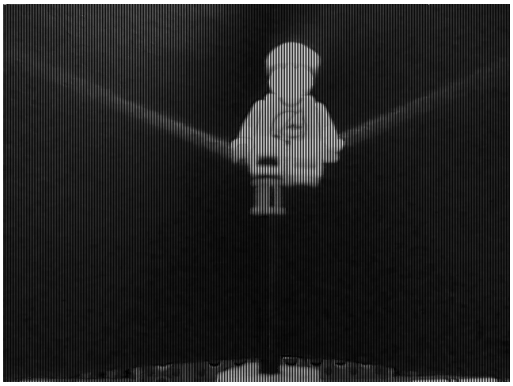


Figura 5.20 O1V1 m_laser 60º

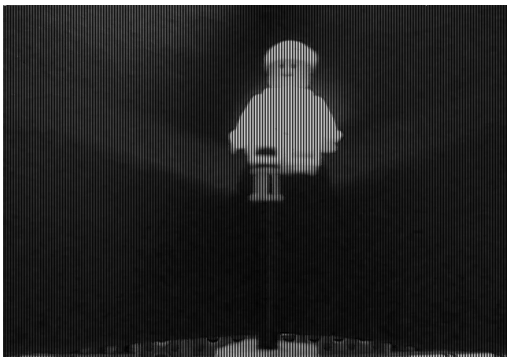


Figura 5.21 O1V1 m_laser 75º

Segunda vuelta del motor de la base (Giro de 90º)

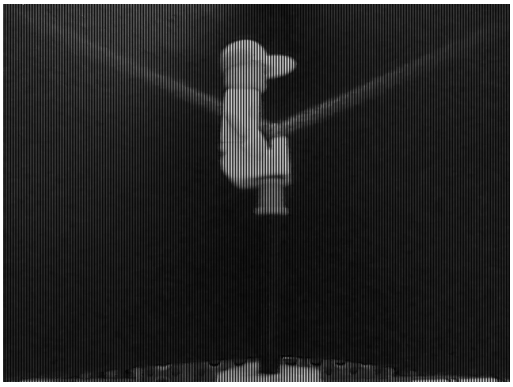


Figura 5.22 O1V2 m_laser 15º

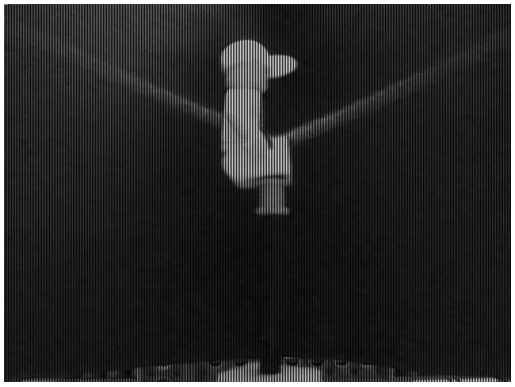


Figura 5.23 O1V2 m_laser 30º

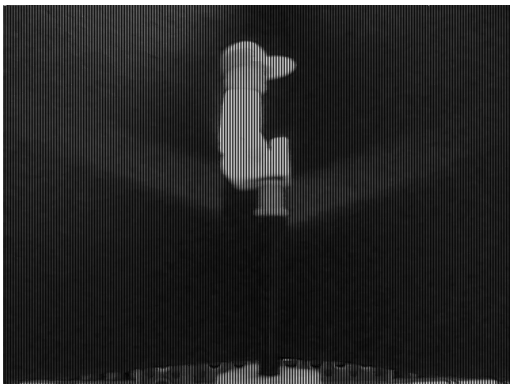


Figura 5.24 O1V2 m_laser 45º

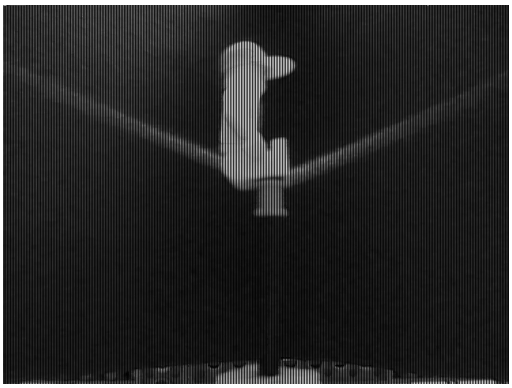


Figura 5.25 O1V2 m_laser 60º

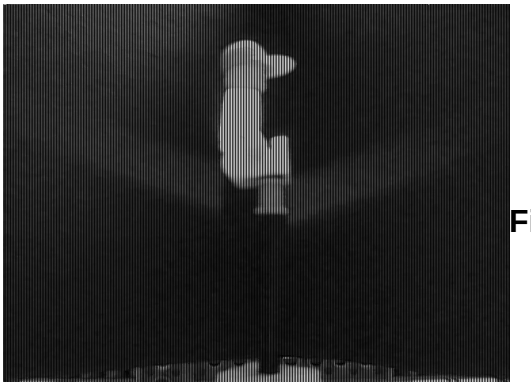


Figura 5.26 O1V2 m_laser 75º

Tercera vuelta del motor de la base (Giro de 180º)

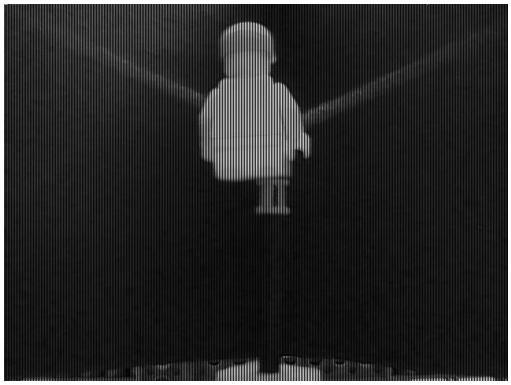


Figura 5.27 O1V3 m_laser 15º

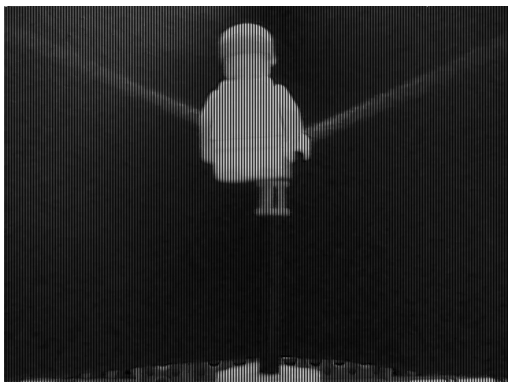


Figura 5.28 O1V3 m_laser 30º

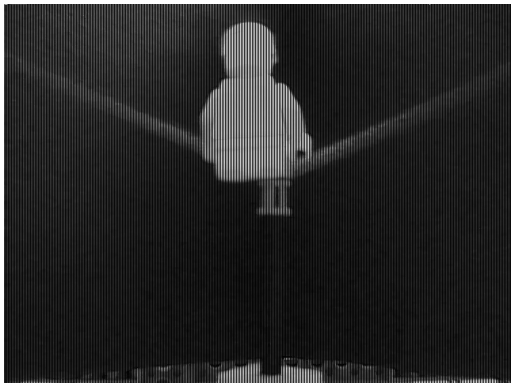


Figura 5.29 O1V3 m_laser 45º

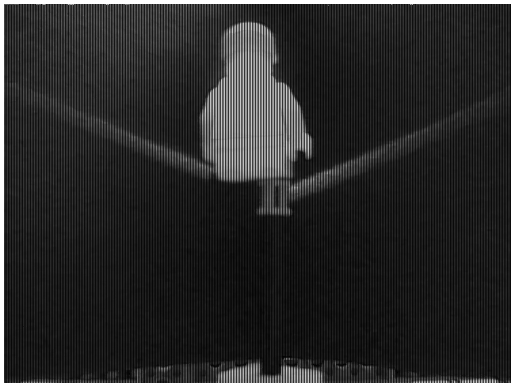


Figura 5.30 O1V3 m_laser 60º

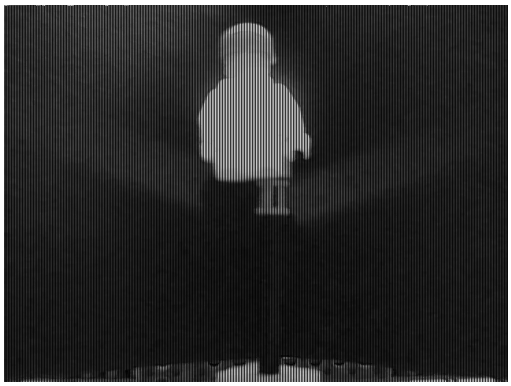


Figura 5.31 O1V3 m_laser 75º

Cuarta vuelta del motor de la base (Giro de 270º)

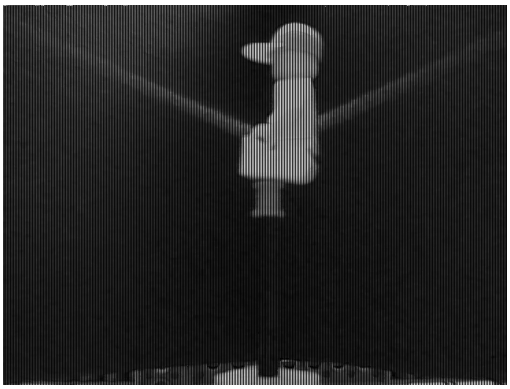


Figura 5.32 O1V4 m_laser 15º

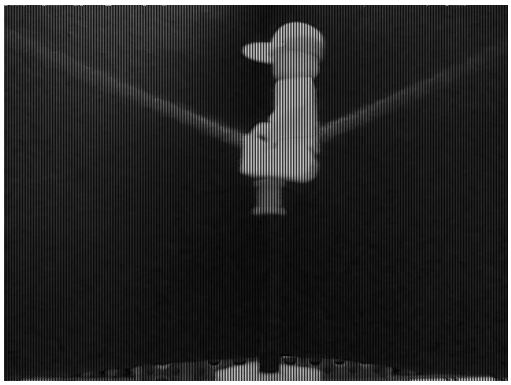


Figura 5.33 O1V4 m_laser 30º

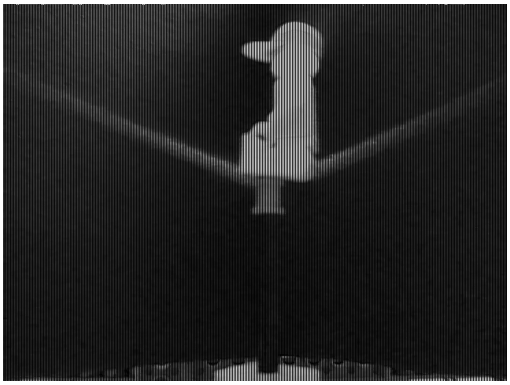


Figura 5.34 O1V4 m_laser 45º

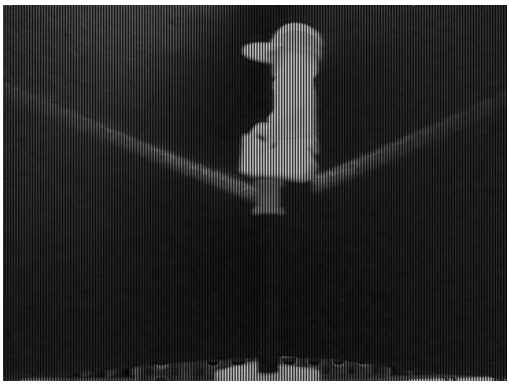


Figura 5.35 O1V4 m_laser 60º

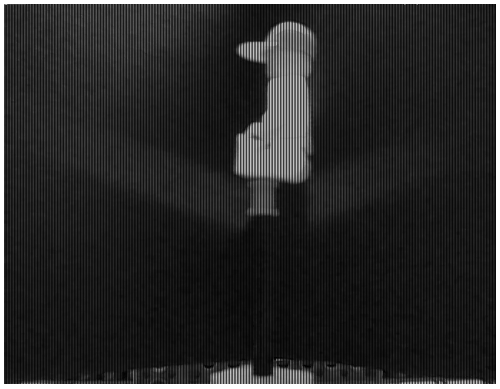


Figura 5.36 O1V4 m_laser 75º

Imágenes resultantes de la comunicación entre Nodos con Objeto#2

Primera vuelta del motor de la base (Giro de 0º)

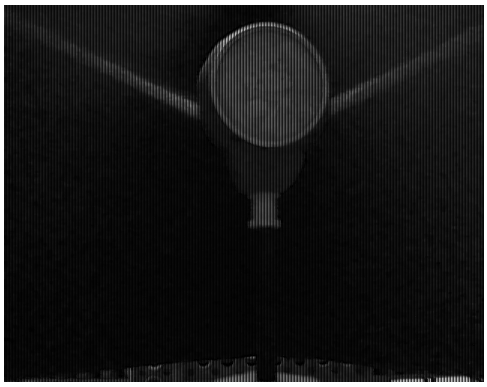


Figura 5.37 O2V1 m_laser 15º

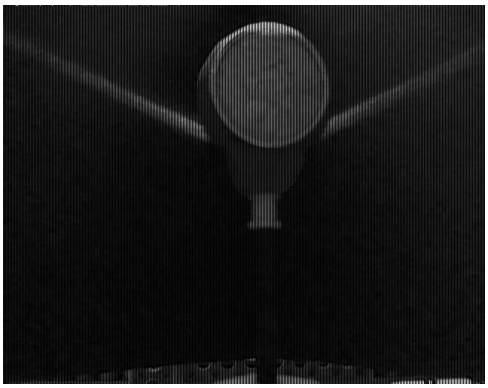


Figura 5.38 O2V1 m_laser 30º

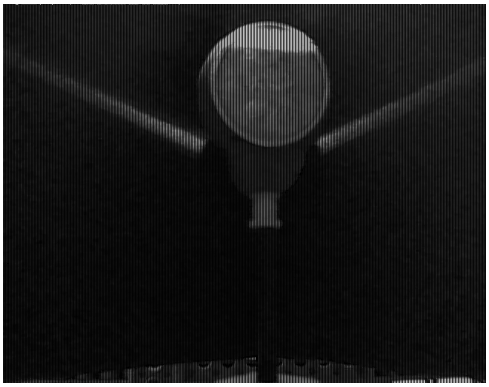


Figura 5.39 O2V1 m_laser 45º

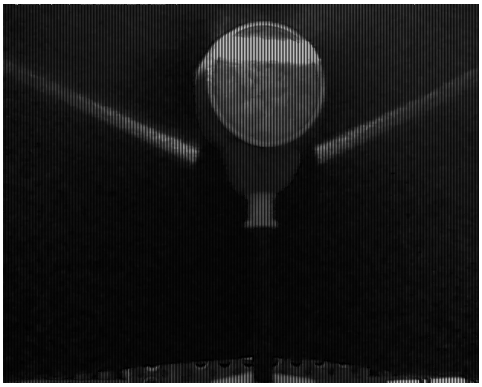


Figura 5.40 O2V1 m_laser 60º

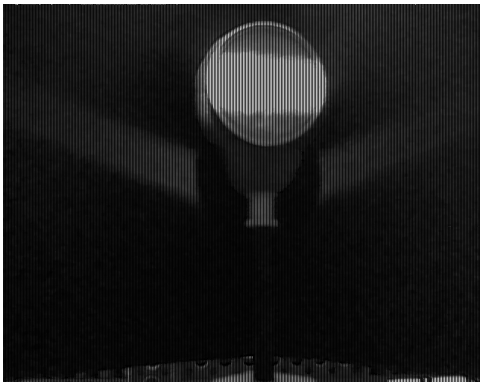


Figura 5.41 O2V1 m_laser 75º

Segunda vuelta del motor de la base (Giro de 90º)

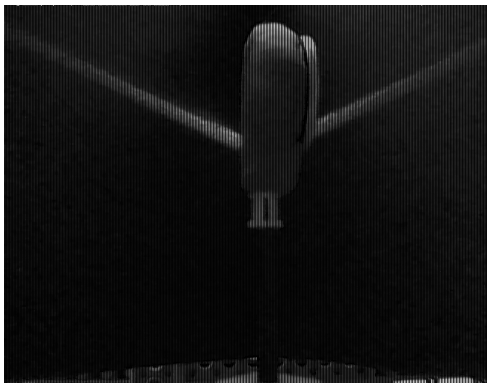


Figura 5.42 O2V2 m_laser 15º

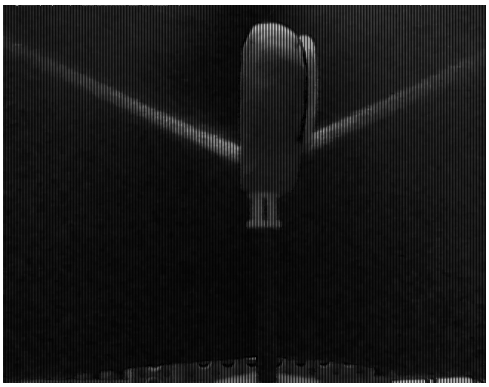


Figura 5.43 O2V2 m_laser 30º

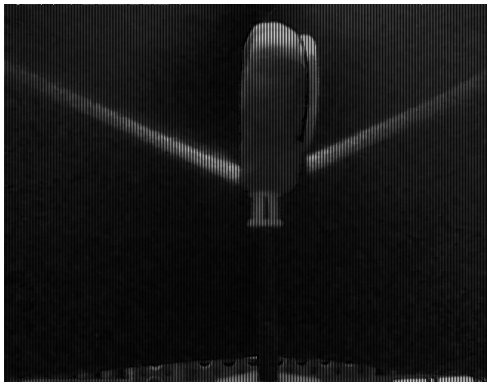


Figura 5.44 O2V2 m_laser 45º

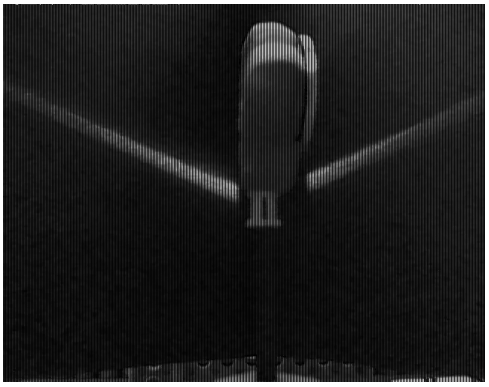


Figura 5.45 O2V2 m_laser 60º

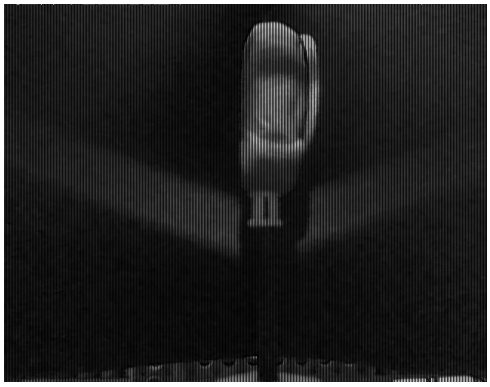


Figura 5.46 O2V2 m_laser 75º

Tercera vuelta del motor de la base (Giro de 180º)

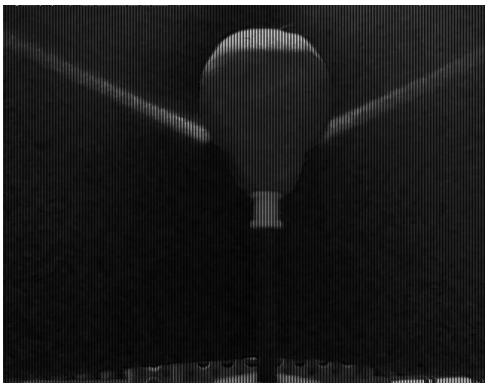


Figura 5.47 O2V3 m_laser 15º

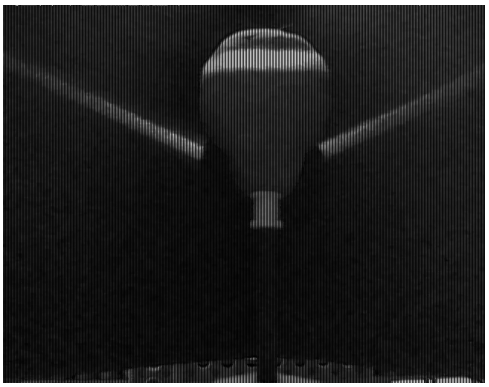


Figura 5.48 O2V3 m_laser 30º

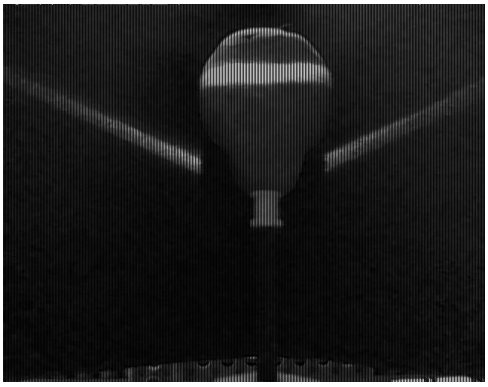


Figura 5.49 O2V3 m_laser 45º

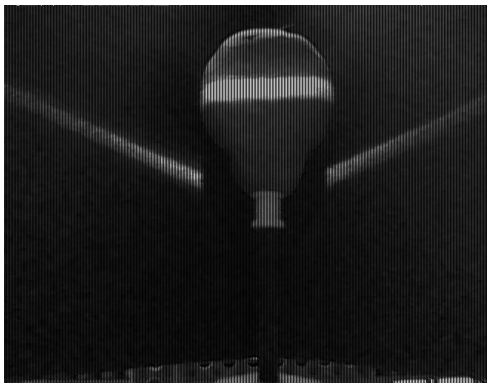


Figura 5.50 O2V3 m_laser 60º

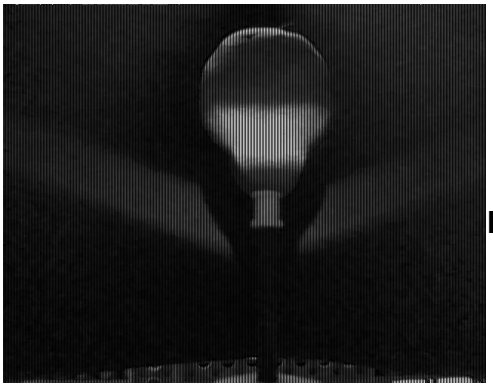


Figura 5.51 O2V3 m_laser 75º

Cuarta vuelta del motor de la base (Giro de 270º)

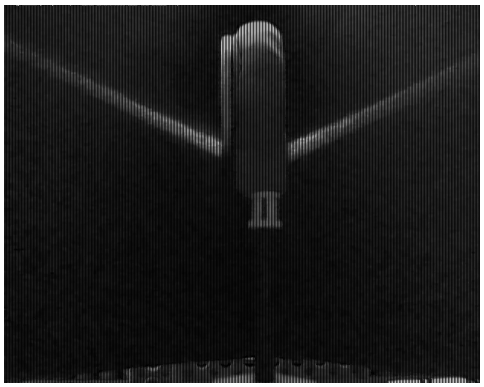


Figura 5.52 O2V4 m_laser 15º

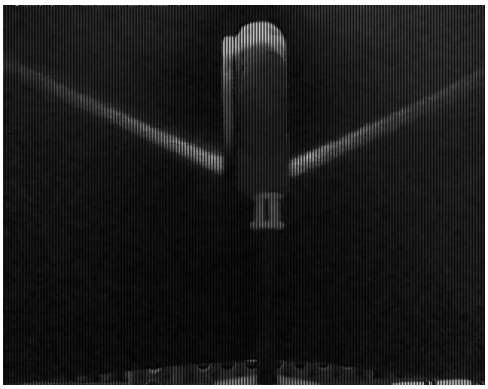


Figura 5.53 O2V4 m_laser 30º

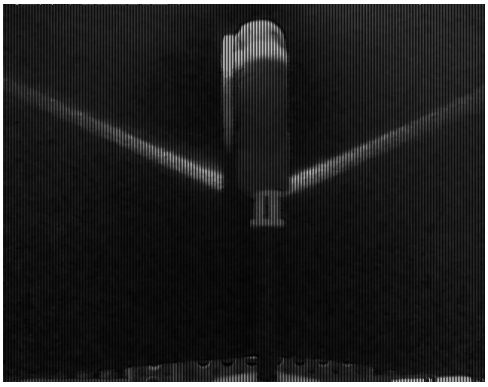


Figura 5.54 O2V4 m_laser 45º

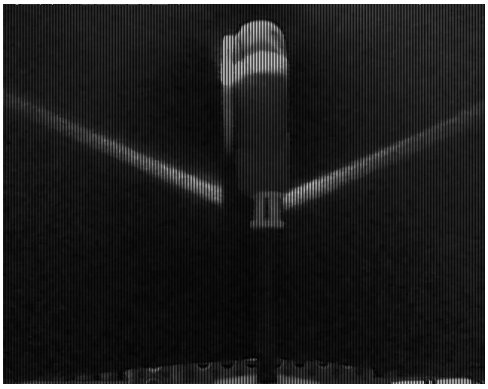


Figura 5.55 O2V4 m_laser 60º

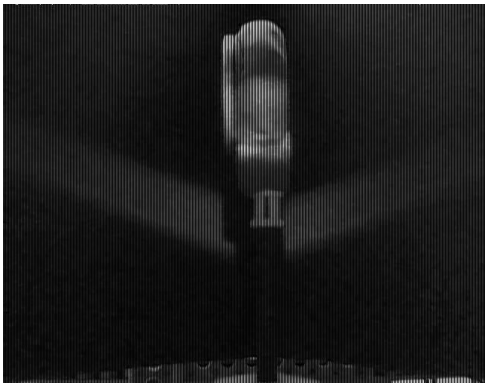


Figura 5.56 O2V4 m_laser 75

CONCLUSIONES Y RECOMENDACIONES

Conclusiones

1. El uso de ROS permite implementar soluciones de tipo académico e industrial según las necesidades actuales.

2. ROS brinda la facilidad de desarrollar aplicaciones menos costosas, debido a que es de libre distribución y gracias a esta característica cada día más personas acceden a su uso.

3. ROS proporciona una gran cantidad de recursos, que tienen la característica de ser de código abierto, lo que permite reutilizarlos y tener resultados más eficientes sin tener que utilizar grandes y complejos códigos.

4. En el proyecto Laser Scanner 3D, se comprobó que en ROS se pueden usar varios lenguajes de programación, dentro de una misma aplicación sin tener problemas de ejecución, ya que no se tuvo inconveniente al trabajar con Python y C++.

5. Se verifico que en ROS es posible solucionar errores de una manera más sencilla, debido a que las aplicaciones pueden ser divididas en nodos, los cuales cumplen diferentes y fundamentales funciones para las aplicaciones.
6. Los NXT Lego, son una herramienta fácil de manejar, sin embargo un proyecto como Laser Scanner 3D, sería imposible de realizar con el software original.
7. Laser Scanner 3D, es un proyecto ambicioso, que a pesar de ser creado con elementos de poca complejidad como NXT Lego, un laser lineal y una cámara web que no implicaron altos costos, puede ser aplicado en diversos y complicados campos como en Botánica, Robótica y demás fines académicos e investigativos debido a que su propósito aunque sencillo puede llegar a ser bien remunerado.

Recomendaciones

1. Es recomendable leer toda la documentación posible acerca de ROS y realizar los pequeños ejemplos de aplicación con los que la pagina de ROS propone comenzar, antes de alguna aplicación compleja, cabe recalcar que hay que tener un nivel avanzado de ingles, ya que un 95% de la información se encuentra en este idioma.
2. Seguir paso a paso los tutoriales de instalación de la página oficial de ROS, si se presenta un problema, se pueden buscar soluciones dentro de la misma

página ya que existe opciones de soporte a usuarios principiantes y avanzados [x1].

3. Se recomienda un nivel intermedio de conocimiento de Linux, debido a que todo se ejecuta por medio de la interfaz de línea de comandos.
4. Para el proyecto Laser Scanner 3D, es recomendable usar un laser linear lo más fino posible, para q así las imágenes puedan ser captadas con mayor precisión a la forma del objeto.
5. Es recomendable usar el lenguaje de programación Python para mover motores, ya que es mucho más fácil y no se necesitan tantas líneas de código como en otros lenguajes de programación.

BIBLIOGRAFÍA

[1] **Youscribe**, Que es ROS, http://es.youscribe.com, 03 de septiembre 2012

[2] **ROS.org**, Que es ROS, http://www.ros.org/wiki/ROS, 03 de septiembre 2012

[3] **ROS.org**, Introducción, http://www.ros.org/wiki/ROS/Introduction, 03 de septiembre 2012

[4] **ROS.org**, Server, http://www.ros.org/wiki/Parameter%20Server, 10 de septiembre 2012

[5] **ROS.org**, ROS Master, http://www.ros.org/wiki/Master, 10 de septiembre 2012

[6] **ROS.org**, Conceptos, http://www.ros.org/wiki/ROS/Concepts, 17 de septiembre 2012

[7] **Massachusetts Institute of Technology**, Herramientas de ROS, http://courses.csail.mit.edu/6.141/spring2012/pub/lectures/Lec06-ROS.pdf, 01 de octubre 2012

[8] **Massachusetts Institute of Technology**, Nivel de Sistema de Archivos de ROS, http://courses.csail.mit.edu/6.141/spring2012/pub/labs/lab4/docs/Visual-Servo-Lab-Procedure.pdf, 08 de octubre 2012

[9] **ROS.org**, Pilas, http://www.ros.org/wiki/Stacks, 15 de octubre 2012

[10] **ROS.org**, Pilas, http://www.ros.org/wiki/Stack%20Manifest, 22 de octubre 2012

[11] **ROS.org**, Pilas, http://www.ros.org/wiki/Stack%20Manifest,22 de octubre 2012

[12] **ROS.org**, SRV, http://www.ros.org/wiki/srv, 22 de octubre 2012

[13] **ROS.org**, Objetivos de ROS, http://www.ros.org/wiki/ROS/Introduction, 22 de octubre 2012

[14] **Airccse.org**, Proyecto Player, http://airccse.org/journal/ijaia/papers/1011ijaia10.pdf, 22 de octubre 2012

[15] **Cse.iitk.ac.in**, Open CV http://www.cse.iitk.ac.in/users/vision/dipakmj/papers/OReilly%20Learning%20OpenCV.pdf, 29 de octubre 2012

[16] **Openrave.org,** Openrave, http://openrave.org/docs/latest_stable/, 05 noviembre 2012

[17] **Stanford University,** ROS, http://pub1.willowgarage.com/~konolige/cs225B/docs/quigley-icra2009-ros.pdf, 05 de noviembre 2012

[18] **ROS.org**, Características de ROS , http://www.ros.org/wiki/ROS/Introduction, 12 de noviembre 2012

[19] **Stanford University,** Lenguaje de programación en ROS http://pub1.willowgarage.com/~konolige/cs225B/docs/quigley-icra2009-ros.pdf, 19 de noviembre 2012

[20] **Stanford University,** Características de ROS http://pub1.willowgarage.com/~konolige/cs225B/docs/quigley-icra2009-ros.pdf, 26 de noviembre 2012

[21] **ROS.org,** Características de ROS, http://ros.org/wiki/rostest, 03 de diciembre 2012

[22] **Stanford University,** Características de ROS http://pub1.willowgarage.com/~konolige/cs225B/docs/quigley-icra2009-ros.pdf, 03 de diciembre 2012

[23] **Stanford University,** Topologia http://pub1.willowgarage.com/~konolige/cs225B/docs/quigley-icra2009-ros.pdf, 03 de diciembre del 2013

[24] **Wikipedia**, Definición de udev, http://es.wikipedia.org/wiki/Udev, 24 de junio 2013

[25] **ROS.org**, Espacio de Trabajo

http://www.ros.org/doc/independent/api/rosinstall/html/rosws_tutorial.html, 24 de junio 2013

[26] **Nazca Sistemas**, Definición de Source

http://nazcasistemas.com/blogs/pacozarate/2010/07/24/linux-que-significa-el-comando-source/, 24 de junio 2013

[27] **Generation Robots**, Programacion con ROS

http://www.generationrobots.com/ros-robot-operating-system,us,8,74.cfm, 10 de diciembre 2012

[28] **Python.org,** Python, http://www.python.org/about/, 10 de diciembre 2012

[29] **Sergiopalay.files.wordpress.com,** Python

http://sergiopalay.files.wordpress.com/2012/03/python-para-todos.pdf, 10 de diciembre 2012

[30] **Fing.edu.uy,** API nxt-python

http://www.fing.edu.uy/inco/cursos/fpr/wiki/index.php/API_nxt-python, 17 de diciembre 2012

[31] **Pyspanishdoc.sourceforge.net,** Módulos

http://pyspanishdoc.sourceforge.net/tut/node8.html, 08 de enero 2013

[32] **Dcmembedded.wordpress.com,** NXT Python, Sistemas integrados, distribuidos y paralelos, http://dcmembedded.wordpress.com/2011/08/05/nxt_python/, 08 de enero 2013

[33] **Lrobotikas.net** Programación NXT. http://lrobotikas.net/wiki/index.php?title=Programaci%C3%B3n_NXT#Python, 15 de abril 2013

[34] **ROS.org,** Paquete NXT, http://ros.org/doc/electric/api/nxt_python/html/python/, 22 de abril 2013

[35] **ROS.org,** Variables de Entorno, http://www.ros.org/wiki/ROS/EnvironmentVariables, 06 de mayo 2013

[36] **Docs.python.org,** Módulos, http://docs.python.org/2/tutorial/modules.html#the-module-search-path, 12 de mayo 2013