ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

Facultad de Ingeniería en Electricidad y Computación

"ESTUDIO DE UNA METODOLOGÍA PARA LA INTEGRACIÓN DE SENSORES Y ACTUADORES EN SISTEMAS ROBÓTICOS HÍBRIDOS"

INFORME DE PROYECTO DE GRADUACIÓN

Previo a la obtención del Título de:

INGENIERO EN TELEMÁTICA

INGENIERO EN CIENCIAS COMPUTACIONALES SISTEMAS TECNOLÓGICOS

Presentado por:

NELSON FERNANDO MONCAYO LUNA

DANIEL ROBERTO CHIANG GUERRERO

Guayaquil – Ecuador

AÑO: 2015

AGRADECIMIENTO

A Dios, por estar siempre a mi lado todo el tiempo y ayudarme a dar este gran paso en mi vida, a mi madre por su paciencia y apoyo, a mis hermanas por el apoyo, a mi papa por siempre darme consejos, y al Dr. Daniel Ochoa por su aporte, paciencia y confianza en la dirección de esta tesis.

Nelson Fernando Moncayo Luna

A mi familia por haberme dado los estudios, a mis amigos por hacer de mi vida académica una aventura, a la comunidad de software libre KOKOA de ESPOL porque en sus actividades aprendí a no conformarme con lo aprendido en las aulas, y a mi mujer María Eugenia por ser mi fuente de inspiración y principal apoyo ante las diversas situaciones que nos trae la vida.

Daniel Roberto Chiang Guerrero

DEDICATORIA

A mi familia, que estuvieron pendientes de mi vida académica y personal, que con su apoyo y consejos, me ayudaron a terminar con esta etapa de mi vida.

Nelson Fernando Moncayo Luna

A todas las personas que han sido parte de mi vida académica, y a todas aquellas personas que me han alentado y apoyado a conseguir esta meta.

Daniel Roberto Chiang Guerrero.

TRIBUNAL DE SUSTENTACIÓN

Ph.D. Sixto García A.

SUBDECANO
SUBROGANTE DE LA
FIEC

Ph.D. Daniel Ochoa D.

DIRECTOR DEL PROYECTO

DE GRADUACIÓN

M.Sc. Víctor Asanza A.

MIEMBRO DEL TRIBUNAL

DECLARACIÓN EXPRESA

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

(Reglamento de Graduación de la ESPOL)

_____

Nelson Fernando Moncayo Luna

_____

Daniel Roberto Chiang Guerrero

RESUMEN

Este trabajo tiene como propósito definir una metodología para el desarrollo de sistema robóticos híbridos. Para esto, estudiamos diferentes frameworks usados para el diseño e implementación de sistemas robóticos, además de otras herramientas de software que los complementan para extender sus capacidades. Esto nos ha permitido seleccionar un framework como el más apto. Al final implementamos un sistema robótico híbrido como ejemplo de uso de nuestra metodología.

El documento está dividido en 5 capítulos que comprenden: la información general, frameworks, herramientas de software, arquitectura del framework seleccionado, implementación de un sistema robótico híbrido y posteriormente las conclusiones y recomendaciones.

En el capítulo 1 se expone la descripción del problema, los objetivos de nuestra tesis, su justificación, la forma en que se desarrolla esta tesis y los resultados que se espera obtener.

En el capítulo 2 indicamos que es un framework y su papel en el mundo de la robótica. Además se describen diferentes frameworks para la implementación de sistemas robóticos. También se analizan diferentes características que son importantes en este tipo de software y como conclusión se elige a aquel que consideramos más apto para el desarrollo de nuestra metodología.

En el capítulo 3 mencionamos algunas herramientas que son frecuentemente utilizadas en la implementación de sistemas robóticos, ya que pueden extender la capacidad de los frameworks añadiendo funcionalidades como visión por computadora, procesamiento de nubes de de puntos, y simulación de robots.

En el capítulo 4 se divide en dos partes. En la primera describimos la arquitectura del framework seleccionado, enfocándonos en aquellos componentes cuya comprensión consideramos es fundamental para poder hacer un correcto uso del mismo. En la segunda parte definimos y detallamos nuestra metodología que abarca temas tanto de diseño, desarrollo e incluso reutilización componentes para poder llegar a implementar un sistema robótico híbrido.

En el capítulo 5 se muestra un ejemplo de aplicación de nuestra metodología. Empezamos por describir el sistema robótico híbrido a implementar y su diseño, luego se describen los componentes que serán reutilizados y aquellos que es necesario desarrollar. También se mencionan los componentes físicos que forman parte del robot. Al finalizar del capítulo se presentan los resultados obtenidos en el mismo.

Para finalizar esta tesis se presentan las conclusiones obtenidas y nuestras recomendaciones. Además se incluyen como anexo los fuentes desarrollados para la implementación del sistema robótico híbrido descrito en el capítulo 5.

ÍNDICE GENERAL

PAG.

CAPÍTULO 2

CAPÍTULO 3

CAPÍTULO 4

ÍNDICE DE FIGURAS

PÁG.

ÍNDICE DE TABLAS

PÁG.

INTRODUCCIÓN

“Un robot es un sistema autónomo que existe en el mundo físico, con la capacidad de percibir su entorno e interactuar en él para lograr un objetivo.” [1]

Hay indicios que demuestran que desde la época de la antigua Grecia (si es que no es antes), siempre ha existido el anhelo de poder crear seres autónomos con apariencia humana que realicen tareas de la misma forma que los humanos. Esta es una idea que ha rondado la cabeza tanto de escritores, ingenieros e inventores. Todos refiriéndose a lo mismo sin saberlo ya que no se contaba con un término que defina a estos seres autónomos.

La palabra “robot” fue usada por primera vez en 1920 por el escritor Karel Čapek en su obra *R.U.R.* (Rossum's Universal Robots) para denotar a trabajadores artificiales. Deriva del checo “robota” que significa trabajo servil [2]. En 1941 nace la robótica enfocada en el estudio, desarrollo y uso de robots.

En la actualidad existen gran cantidad de robots tanto industriales como de servicio, que son utilizados para realizar una amplia gama de tareas con mayor precisión y fiabilidad que los humanos. Una de las grandes ventajas de emplear robots en sitios de trabajo es que ellos pueden desempeñar aquellas tareas que para nosotros pueden resultar desagradables o incluso peligrosas. Su utilidad es diversa, en la actualidad se usan mayormente en las líneas de fabricación, montaje y embalaje, transporte, exploración de la tierra y el espacio, y la producción en masa de bienes de consumo e industriales.

En algunos casos, existen investigadores en robótica que no disponen de muchos recursos económicos lo que dificulta la adquisición de componentes necesarios para la elaboración de un robot. Sin embargo existen alternativas a nivel de software que les permiten reutilizar componentes independientemente de su fabricante obteniendo como producto final a lo que llamamos un robot híbrido, y de esa forma socavar dicha limitante.

El presente trabajo se enfoca en la definición de una metodología para el desarrollo de sistemas robóticos híbridos utilizando herramientas de software para el desarrollo robótico a las que nos referiremos como frameworks y que deban poder ejecutarse sobre un sistema operativo libre. Como parte de esto se analiza previamente las capacidades de los diferentes frameworks. Y se realiza además un ejemplo práctico de la implementación.

CAPÍTULO 1

1. INFORMACIÓN GENERAL

En este capítulo se presenta una descripción global del proyecto, así como la importancia que puede tener su aplicación en el campo de la robótica. Además se exponen las bases sobre las cuales se justifica el proyecto y se describe los objetivos generales y específicos del proyecto. Al final se describe los resultados esperados al finalizar el proyecto.

1.1. DESCRIPCIÓN DEL PROBLEMA

Hoy en día la utilización de sistemas robóticos es cada vez más frecuente.

Cada día van aumentando las diferentes tareas que estos pueden realizar, ya sea en la industria, en lo académico o incluso en el hogar.

Los sistemas robóticos desempeñan funciones como trasladar objetos de un lugar a otro, podar el césped, desminar de campos, cuidar de cultivos, entre otras. Todas estas funciones son posibles gracias a la utilización de sensores (giroscopios, acelerómetros, ultrasónicos, etc.) y actuadores (motores) que permiten al robot obtener información del entorno que lo rodea, procesarla y ejecutar alguna acción.

Existe una gran variedad de fabricantes que diseñan y fabrican componentes (sensores y actuadores) “exclusivos” para sus sistemas robóticos. Tales sistemas son pocos flexibles y obligan al usuario a comprar el repuesto de dicho fabricante en caso de avería. Estos sistemas suelen ser “incompatibles” con otros sistemas robóticos debido a que usan protocolos o interfaces propios de su fabricante que no pueden ser integrados fácilmente con los sistemas de terceros. Este tipo de sistemas se los conoce como sistemas cerrados.

Los sistemas cerrados son costosos. Los investigadores en robótica no siempre pueden adquirir todos los componentes de los sistemas cerrados debido a la falta de presupuesto para cubrir los gastos de inversión y mantenimiento. Este problema afecta particularmente a entornos académicos donde el alcance de las investigaciones está limitado por las características del sistema robótico se pueda adquirir para usar como plataforma experimental.

Hay varios problemas que son consecuencias de trabajar con sistemas cerrados, altos costos de adquisición del hardware y software, escasa compatibilidad con otras marcas, falta de documentación del funcionamiento interno de los controladores, necesidad de contratar al fabricante para implementar nuevas funcionalidades, entre otros. Ante esta situación en los últimos años han surgido propuestas de herramientas de software (Player, OROCOS, CARMEN, YARP, ORCA, ROS.) orientados a integrar componentes de distintos fabricantes mediante la utilización de interfaces estándares de programación. Este enfoque facilita la implementación de los sistemas robóticos híbridos. Es decir sistemas robóticos que incorporan y utilizan componentes de diferentes fabricantes de forma coordinada, a un costo potencialmente menor que asociado a un sistema cerrado.

En este proyecto de tesis, estudiaremos diversas plataformas que proveen mecanismos para comunicación entre procesos, interfaces de hardware y rutinas de control de robots. Elaboraremos una metodología que permita implementar un sistema robótico híbrido utilizando componentes de diferentes fabricantes. Pondremos en práctica la metodología elaborada implementando un sistema robótico

híbrido que sea capaz de navegar en exteriores, y daremos pautas y recomendaciones a investigadores interesados en la implementación de este tipo de sistemas robóticos.

1.2. OBJETIVO GENERAL

- Obtener una metodología que permita implementar sistemas robóticos híbridos.

1.3. OBJETIVOS ESPECÍFICOS

- Estudiar y evaluar de forma cuantitativa los frameworks de programación que sirven para implementar sistemas robóticos híbridos.
- Desarrollar una metodología para utilizar el framework que sea el más adecuado para la implementación de sistemas robóticos híbridos.
- Implementar un sistema robótico híbrido utilizando la metodología desarrollada.

1.4. JUSTIFICACIÓN

Actualmente en la robótica, los fabricantes de sistemas robóticos ofrecen librerías de programación que implementan interfaces y protocolos de comunicación propietarios. En consecuencia, el uso de software para controlar los componentes físicos (actuadores,

sensores) está sujeto a los límites de un sistema robótico impuestos por el fabricante. Esto agrega complejidad innecesaria al proceso de construcción de un robot encareciendo la investigación y el desarrollo tecnológico. La metodología desarrollada en esta tesis permitirá sistematizar el diseño e implementación de sistemas robóticos complejos sin las restricciones antes mencionadas.

En el caso particular de la robótica dentro del Ecuador no existen fabricantes locales de componentes electrónicos en la industria. La importación implica gastos adicionales relacionados a impuestos y transporte. Además de retrasos en los proyectos debido al tiempo que toman los componentes en llegar al país y retirarlos de la aduana. Nuestra metodología hará posible reutilizar diferentes componentes de sistemas robóticos disponibles e integrar partes construidas manualmente en los centros de investigación. Esto es de particular importancia para los laboratorios de instituciones de educación superior como la ESPOL donde se han adquirido distintos sistemas robóticos, cuyo mantenimiento y operación es complicado debido a la falta de licencias de software y el costo de los repuestos.

En la práctica hay 3 opciones para hacer investigación robótica que son:

1. Comprar el robot y el software provisto por el fabricante.
2. Comprar partes de robot, de diferentes fabricantes y hacer ingeniería inversa de sus controladores para modificarlos y crear un sistema robótico nuevo.
3. Utilizar un software libre con un extenso repositorio de algoritmos de control para poder usarlos para crear nuevos sistemas robóticos híbridos.

La aparición de software de licencia libre ha reducido en gran medida las complicaciones mencionadas anteriormente ya que brindan la posibilidad de reutilizar y ejecutar un mismo algoritmo de control en sistemas diferentes. Estos programas proveen una interfaz estándar de programación para combinar componentes individuales de fabricantes distintos en un solo sistema. Dependiendo del software, es posible crear interfaces de programación compatibles para nuevos componentes de hardware.

En resumen, la primera opción demanda una gran inversión de dinero. La segunda opción demanda una gran inversión de tiempo. La tercera, aprovecha los recursos existentes del software libre por lo que es la opción más viable para lograr nuestro objetivo.

En Ecuador una aplicación de la metodología que proponemos en esta tesis es compartir entre universidades sus sistemas robóticos existentes para trabajar colaborativamente en actividades académicas y de investigación. Esto puede realizarse en línea aprovechando las capacidades de las redes de alta velocidad a la que tienen acceso las universidades que forman parte del Consorcio Ecuatoriano para el Desarrollo de Internet Avanzado (CEDIA).

1.5. METODOLOGÍA

Para realizar este proyecto es necesaria una investigación bibliográfica acerca de centros y laboratorios dedicados a la implementación de sistemas robóticos híbridos para conocer el estado de los diferentes softwares que sirven para la construcción de estos sistemas. Además estudiamos y comparamos las diferentes características de dichos softwares. Analizamos algunas de las librerías y herramientas de procesamiento de datos que más se utilizan en conjunto con los sistemas robóticos.

En base a los resultados de nuestra investigación y la experiencia adquirida en el Centro de Visión y Robótica de la ESPOL (CVR), proponemos una metodología que pueda servir como guía para la implementación de sistemas robóticos híbridos. Para demostrar la funcionalidad de nuestra metodología implementamos un sistema

robótico híbrido compuesto por una plataforma móvil y un sensor láser, ambos de diferentes fabricantes.

Finalmente exponemos nuestras conclusiones y recomendaciones, que esperamos sean de ayuda a las personas que quieran emplear o extender la metodología presentada.

1.6. RESULTADOS ESPERADOS

Los resultados que se pretenden conseguir con el desarrollo de esta tesis son los siguientes:

- Identificar el framework para control de robots que más se ajuste a las condiciones de un laboratorio de investigación académica.
- Desarrollar una metodología que sirva como guía para implementar sistemas robóticos híbridos.
- Realizar un experimento que evidencie la implementación de sistemas robóticos híbridos utilizando la metodología desarrollada y el framework para control de robots seleccionado.

CAPÍTULO 2

2. FRAMEWORKS

2.1. INTRODUCCIÓN

En ingeniería del software, un framework, es una estructura conceptual y tecnológica que sirve como base para la organización y desarrollo de software. Típicamente el framework puede incluir soporte de programas, librerías, un lenguaje interpretado, entre otras herramientas. Estos componentes ayudan a desarrollar y unir las diferentes partes de un sistema computacional. Además los frameworks proporcionan servicios a las aplicaciones que van más allá de los disponibles en el sistema operativo.

2.2. DESCRIPCIÓN

A medida que los robots han adquirido más capacidades, se ha hecho evidente la necesidad del desarrollo de software e implementación de metodologías de apoyo. En el pasado era posible, tal vez incluso de moda, desarrollar todo el software para sistemas robóticos por aficionados a la robótica en sus hogares. Pero como, los sistemas robóticos se han vuelto más complejos, este enfoque se ha vuelto impráctico. Los investigadores en robótica han respondido a estos retos mediante la adopción y adaptación de metodologías de desarrollo de software usadas en otros campos que ya han enfrentado el problema de la complejidad.

La metodología que hemos tomado como referencia es la Ingeniería de Software a base de componentes (CBSE). Este es un paradigma emergente en la cual las aplicaciones son desarrolladas integrando componentes de software ya existentes y ha sido la inspiración para frameworks de control de robots construidos sobre los resultados de varios grupos de investigación. [3]. Un componente es un fragmento de software que puede ser ensamblado con otros fragmentos para formar piezas más grandes o ejecutar operaciones complejas [4]. Típicamente, un robot requiere capacidades para ejecutar tareas de detección, modelización y de planificación complejas, el paradigma

CBSE divide estas tareas en fragmentos y los combina para resolverlos.

Algunos frameworks populares para robótica se basaron en el paradigma CBSE y eventualmente emplearon licencias propietarias para proteger varios fragmentos de software. Otros frameworks como LEGO MINDSTORMS, y SKILLIGENT emplean CBSE pero desde sus inicios han sido cerrados. En la práctica las restricciones de acceso al código fuente de un framework propietario representan una barrera para aplicar el paradigma CBSE. Una alternativa es usar frameworks con licencias libres. Una lista parcial de Frameworks de licencia libre para control de robots incluye: Player, ROS, MIRO, YARP, OROCOS, CARMEN, ORCA y MOOS.

A primera vista, la diversidad de frameworks parece un obstáculo para la reutilización de los fragmentos de software. Sin embargo, existen mecanismos como APIs (Application programming Interface) o RPC (Remote Procedure Calls) que permiten agregar componentes de software a estos frameworks. Para que esto funcione se requieren mecanismos de intercambio de datos entre componentes que se adhieran a enfoques estandarizados. Por ejemplo el paradigma de sistemas de mensajería, más conocido como MOM (Message-

Oriented Middleware) que implementa dos modelos (1) “peer-to-peer” y (2) “publish-subscriber” [5]. MIRO, OROCOS, ORCA, MOOS y ROS implementan el paradigma de”publish-subscriber”.

En los casos donde el framework y el intercambio de datos sean desacoplados de la rutina del control del robot se puede maximizar la posibilidad de reutilización del software. La siguiente sección se enfoca en describir brevemente la arquitectura de los frameworks libres.

- MIRO (Middleware for robots).
- YARP (Yet another Robot Platform).
- ROS (Robotic Operating System).
- OROCOS (C++ Framework para componentes robóticos).

2.3. ARQUITECTURA DE FRAMEWORKS LIBRES

2.3.1. MIRO

MIRO (Middleware for robots) es un framework distribuido orientado a objetos para control de robots, se basa en tecnología de CORBA (Common object Request Broker Architecture) que es un estándar definido por Object Management Group (OMG) que permite que diversos componentes de software escritos en múltiples lenguajes de

programación y que corren en diferentes computadoras, puedan trabajar juntos, es decir facilita el desarrollo de aplicaciones distribuidas [6]. Los componentes básicos de Miro se han desarrollado bajo la ayuda de ACE (Adaptive Communication Environment), una librería orientada a objetos para comunicación interprocesos del sistema operativo, la red y la comunicación en tiempo real. Utilizan también la aplicación TAO (The Ace Orb), como su ORB (Object Request Broker), una aplicación CORBA diseñada para aplicaciones de alto rendimiento y ejecución en tiempo real.

2.3.2. OROCOS

OROCOS (Open Robot Control Software) es un conjunto de herramientas de código abierto para el desarrollo de sistemas robóticos basados en componentes, proporciona los medios para definir y desarrollar componentes que se combinan entre ellos para formar sistemas robóticos arbitrariamente complejos. OROCOS ofrece herramientas de ejecución en tiempo real como RTT (Real Toolkit), que están dirigidas a la implementación de sistemas de control [7].

2.3.3. YARP

YARP (Yet another Robot Platform) es un apoyo a la construcción de un sistema de control de robot como una colección de programas que se comunican punto a punto, con una extensa familia de tipos de conexiones entre componentes que se pueden usar en base a las necesidades del sistema robótico. El objetivo es aumentar la cantidad de los proyectos de software para robots.

YARP está escrito en c++ y usa la librería ACE, que es una librería orientada a objetos para comunicación interprocesos del sistema operativo, la red y la comunicación en tiempo real, entre las tareas que realiza incluye organizador de eventos y demultiplexación de eventos, manejo de señales, iniciación del servicio, la comunicación entre procesos, gestión de memoria compartida, el enrutamiento de mensajes, configuración dinámica de servicios distribuidos, ejecución concurrente y sincronización [8].

2.3.4. ROS

ROS (Robotic Operating System) es un paquete de código abierto que proporciona servicios del sistema operativo como abstracción de hardware, control de dispositivos de bajo nivel y comunicación entre procesos, así como herramientas de desarrollo. El objetivo de ROS es crear una plataforma común sobre la que los investigadores pueden construir y compartir algoritmos de control de robot de alto nivel en áreas tales como navegación, localización, planificación y manipulación [9].

2.4. CARACTERÍSTICAS DE LOS FRAMEWORKS

Para la realización de nuestro proyecto hemos decidido escoger un framework de los mencionados en la sección 2.3, para evaluar los frameworks hemos seleccionado cinco características que se relacionan al objetivo de la tesis que es proponer una metodología para integrar dispositivos de diferentes fabricantes en un sistema robótico. Adicionalmente es necesario tener en cuenta que se necesita que el framework funcione para el sistema operativo Linux por varias razones técnicas que se explican en el capítulo 3.

2.4.1. LICENCIA DE SOFTWARE

La licencia de software una especie de contrato, en donde se especifican todas las normas y cláusulas que rigen el uso de un determinado software, principalmente se estipulan los alcances de uso, instalación, y copia de estos productos. El software suele distribuirse con licencias propietarias o cerradas, es decir que el usuario no es el propietario del software, solo tiene derecho a usarlo en un ordenador o tantos como permita la licencia y no puede modificar el programa ni distribuirlo.

También hay licencias libres, en las cuales el usuario es el dueño del software, él puede modificarlo y distribuirlo de acuerdo a la licencia que use, hay varias licencias libres por ejemplo las mencionadas en tabla 2.1: GPL (General Public License), LGPL (Lesser GPL) y BSD (Berkeley Software Distribution). La licencia GPL tiene la característica de que una vez que se publica un programa con los términos de la licencia GPL, ya no habrá modo de cambiarla, ni mucho menos de querer volverlo propietario. Otras características incluyen:

- Puede copiar, modificar y distribuir a terceros el software, sin tener obligación de pagar por ello.
- Cualquier código fuente licenciado bajo GPL, debe estar disponible y accesible, para copias ilimitadas y a cualquier persona que lo solicite.
- Es obligatorio mencionar a los autores originales del código fuente.

La licencia LGPL es similar a la GPL, pero la LGPL permite que el software licenciado bajo sus términos se pueda integrar o cambiar a software propietario, permitiendo crear aplicaciones privativas basadas en software libre.

La licencia BSD no tiene ninguna restricción a los desarrolladores de software en lo que se refiere a utilizar posteriormente el código modificado y en que licencia publican su trabajo, pueden optar por licencia libre o propietaria. BSD es similar a LGPL pero con la diferencia de que BSD no tiene la obligación de mencionar a los autores ni proporcionar el código fuente [10].

2.4.2. SOPORTE DE LENGUAJES

Hoy en día, los robots se desarrollan por equipos de personas muy heterogéneos no necesariamente usan el mismo lenguaje de programación. De hecho, la elección de un lenguaje de programación puede variar en función de dos criterios: (1) que tan rápido se quiere que el investigador programe la tarea a realizar por el robot y (2) la rapidez que se desea que el robot realice una tarea específica. El soporte de diferentes lenguajes es, por lo tanto, una característica muy deseable debido a que la creación de software para robótica es muy compleja. Una característica fundamental de nuestra metodología es que el desarrollo de software no se vea dificultado por la programación de mecanismos de comunicación entre componentes.

De los frameworks que analizamos, ROS es el único que proporciona una sencilla interfaz de programación orientada a objetos. Con ella, la invocación de métodos de objetos remotos, se mantiene la misma sintaxis que la invocación de métodos de objetos locales. Esto no solo simplifica la estructura de un programa, sino que también permite al desarrollador concentrarse en el problema concreto a resolver [11].

La Tabla 1 muestra la licencia, plataformas y lenguajes soportados por los diferentes frameworks mencionados en la sección anterior.

Tabla 1: Licencia y lenguajes de Frameworks

FRAMEWORK	LICENCIA	SOPORTE DE LENGUAJES
MIRO	GPL	C,C++
OROCOS	LGPL	C++
ROS	BSD	C++,JAVA,PYTHON,OCTAVE,LISP
YARP	GPL	C++,JAVA,RUBY,PYTHON,LISP

2.4.3. IDL

Una IDL (Interface Description Language) es usada para describir una interfaz de comunicación en un lenguaje común. Se usa como puente de comunicación entre componentes de software escritos en diferentes lenguajes de programación. Por ejemplo entre componentes desarrollados en C++ y en Python.

Los frameworks OROCOS Y MIRO son los únicos de los evaluados que proporcionan una interfaz basada en IDL. Por ejemplo CORBA IDL es fácil de leer y es orientado a objetos, su sintaxis es similar a la de C++ pero existen diferencia como por ejemplo no utiliza varias palabras claves de C++ (p.e. public, private) [12].

El framework ROS usa su propio IDL basado en archivos de texto corto para describir los mensajes enviados entre los componentes, hay generadores de código para cada lenguaje soportado que se serializan y de serializan por ROS automáticamente los mensajes se envíen y reciben. Esto le ahorra al programador tiempo y errores, el mensaje anterior de ejemplo automáticamente se expande en 137 líneas de C++ o 96 líneas de python depende el lenguaje del componente [13]. Ver Figura 2.1.

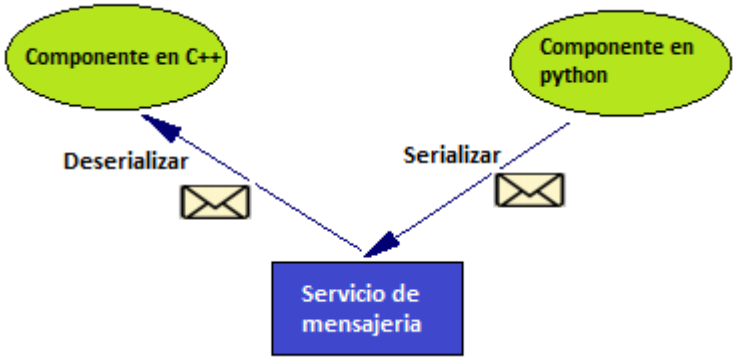


Figura 2. 1: Paso de mensaje serializar y de serializar

2.4.4. MÉTODOS DE COMUNICACIÓN

Son los métodos por los cuales los componentes envían información a otros componentes del sistema y por los cuales reciben información enviada por ellos. Se puede n distinguir diversos métodos de comunicación, los más comunes son: paso de mensajes, cliente-servidor y Publish-Subscriber. Ver Figura 2.2.

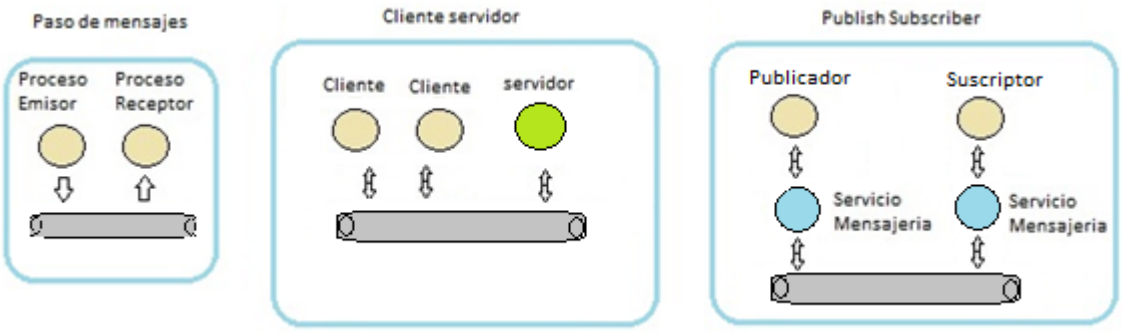


Figura 2. 2: Modelos De Comunicación

El paso de mensajes consiste en enviar de un componente a otro un mensaje a través de un canal de comunicación, el mensaje es un conjunto de información estructurada, y este método consiste en tres elementos principales que son: el proceso que envía, el que recibe y el mensaje, el paso de mensaje es la base de los modelos publish-subscribe y cliente-servidor.

El modelo cliente-servidor se trata de la asignación de dos funcionalidades diferentes a cada uno de los componentes que se comunican. El proceso servidor proporciona una serie de servicios y espera la llegada de peticiones por parte de los clientes. En este modelo, el servidor tiene un papel pasivo, dedicándose a esperar las solicitudes de los clientes, que son los que toman el papel activo. En el modelo basado en publish-subscribe, la comunicación de un proceso publicador (publisher) puede ser a varios procesos suscriptores (subscribers) con soporte a redundancia tanto en las publicaciones, como en los procesos que se suscriben al servicio de mensajería. El servicio de mensajería es el encargado de recibir los mensajes de uno o varios publisher y envía copias a todos los subscribers que

estén suscriptos a él. Los suscriptores registran su interés en la cola de mensajes por medio del nombre de la cola, es un modelo basado en multicast. Ver Figura 2.3.

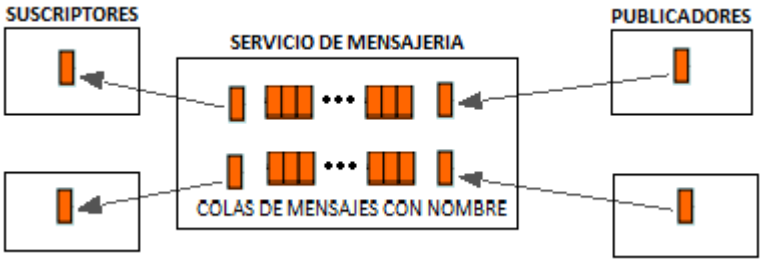


Figura 2. 3: Modelo Publish-Subscribe

Dependiendo del sistema robótico a desarrollar se escoge uno o más métodos de comunicación a utilizar, por ejemplo en un sistema robótico móvil sencillo que tenga un sensor y dos actuadores (motores de ruedas).La tarea del robot es moverse en línea recta si el sensor detecta una señal, entonces si escogemos el modelo publish-subscribe, el sensor (publisher) envía la señal a los dos actuadores (subscribers) al mismo tiempo sin la necesidad de esperar una solicitud por parte de los actuadores, en cambio con el modelo cliente-servidor, el sensor (servidor) está esperando la solicitud de cada actuador (cliente) para poder recién enviar la señal para que se mueven los actuadores (motores). Con el método publish-subscribe se

puede crear rutinas de control de robot que se ejecuten en un menor tiempo que lo hiciera el modelo cliente-servidor.

La Tabla 2 muestra los tipos de IDL y métodos de comunicación utilizados por los diferentes frameworks.

Tabla 2: Tipo de IDL y métodos de comunicación

FRAMEWORK	IDL	Métodos de comunicación
MIRO	CORBA IDL	CLIENTE/SERVIDOR
OROCOS	CORBA IDL	PUBLISH-SUBSCRIBER, CLIENTE/SERVIDOR
ROS	ROS IDL	PUBLISH-SUBSCRIBER, CLIENTE/SERVIDOR
YARP	no	PASO DE MENSAJES ASINCRONO

2.4.5. HERRAMIENTAS

La Tabla 2.3 muestra las herramientas proporcionadas por los distintos frameworks. Significa que la herramienta se ejecuta en modo consola y GUI (Graphic User Interface) si tiene una interfaz gráfica. La columna 'plantilla de código' representa la disponibilidad de una herramienta de generación de código. ROS proporciona una herramienta que crea el árbol de directorios de un paquete ROS así como los ficheros necesarios para su compilación. La columna 'Administrador' representa la existencia de una herramienta específica para ejecutar y

supervisar la ejecución de los componentes. Las columnas 'replay', y 'logs' representan la disponibilidad de herramientas para recrear el comportamiento de componentes y registrar mensajes.

La columna 'simulador' lista los simuladores con los que cada framework trabaja. Permitir a los componentes trabajar con un simulador sin modificar su código, es una herramienta extremadamente útil, sobre todo en las fases más tempranas del desarrollo de un componente. La columna 'monitor' representa la disponibilidad de una herramienta para visualizar la información que procesa un componente, no sólo su estado (p.e. imágenes de una cámara, objetos detectados, o la posición de un robot).

La Tabla 3 muestra las herramientas utilizadas por los diferentes frameworks.

Tabla 3: Herramientas de los Frameworks

Framework	Plantilla de código	Administrador	Replay	Simulador	Log	Monitor
Miro	no	no	no	no	gui	no
OROCOS	no	texto	no	no	no	no
ROS	si	texto	gui	gazebo	gui	gui
YARP	no	gui	no	icubSim	texto	no

2.5. CONCLUSIÓN

Se han descrito algunos de los frameworks libres más documentados para el desarrollo de la robótica, y después de evaluar algunas de sus características y las herramientas que ellos ofrecen hemos concluido que ROS es el mejor framework para evaluar nuestra metodología por las siguientes razones:

1) Ros usa la licencia BSD que es la mejor para fomentar la investigación, esto se refleja en que su comunidad de usuarios es mayor que la de los otros frameworks.

2) ROS y YARP permite escribir programas en 5 lenguajes de alto nivel y bajo nivel más que Miro y OROCOS. Se pueden usar intérpretes o programas compilados dependiendo de a que se le dé prioridad si a facilidad de programación o tiempo de ejecución.

3) Con respecto a los métodos de comunicación, ROS y OROCOS proveen los dos métodos estudiados: publish-subscribe y cliente-servidor.

4) ROS y YARP son los que tienen un mayor número de herramientas, en particular simuladores que permiten probar el sistema robótico

en un ambiente virtual, también tienen herramientas de registro y administración. Solo ROS tiene un monitor y replay que permiten realizar una mejor depuración del sistema y además ROS permite generar una plantilla de código para el software robótico.

Por las razones mencionadas anteriormente, ROS es el que presenta las mayores ventajas de los frameworks evaluados en este capítulo, entonces la metodología que proponemos será realizada en el entorno de ROS.

CAPÍTULO 3

3. HERRAMIENTAS DE SOFTWARE

3.1. LINUX COMO ENTORNO PARA FRAMEWORKS DE CONTROL DE ROBOTS

Hay muchas características que tiene el sistema operativo GNU- Linux que lo hacen mejor para implementar un sistema robótico híbrido, estas características se mencionan a continuación:

- Linux se ejecuta en múltiples plataformas de hardware y no necesita tener un teclado ni pantalla conectada.

- Los controladores de los dispositivos tienen una interfaz de programación (API) estándar y pueden ser desarrollados usando un compilador de lenguaje C.

- No se requiere reiniciar la computadora cuando se realizan la mayoría de las tareas de reconfiguración del sistema. Estas incluso pueden ser realizadas remotamente.

- Los controladores de los dispositivos pueden ser cargados y descargados sin necesidad de reiniciar la computadora. De hecho es posible usar múltiples controladores para un mismo dispositivo. Esto es útil cuando un sensor o actuador robótico necesita ejecutar varias acciones al mismo tiempo. Por ejemplo un sensor de ultrasonido puede medir la distancia de objetos cercanos y evadirlos, como también se puede usar para parquear un robot móvil a cierta distancia de la pared, estas dos acciones se las puede realizar al ejecutar diferentes controladores para el mismo sensor.

- El uso de Linux simplifica en gran medida la implementación de mecanismos de comunicación con un robot: Primero, Linux fue concebido para ser utilizado con un terminal, por lo tanto la

conexión serial con otra computadora es sencilla. Segundo, su sub-sistema de red se usa eficientemente, es flexible y provee protocolos estándares de comunicación.

En resumen Linux es una plataforma conveniente para el desarrollo embebido, especialmente para los robots. Dado que el código fuente está disponible, el sistema puede ser fácilmente modificado para adaptarse a los requisitos especiales de hardware. Además, el desarrollo de controladores de dispositivos requiere las mismas herramientas utilizadas para el desarrollo de aplicaciones y están disponibles en línea.

3.2. HERRAMIENTAS DE PROCESAMIENTO DE DATOS.

Existen varias herramientas externas para procesamiento de los datos, algunas de ellos son usadas con los frameworks descritos en la sección 2, en particular con ROS, por ejemplo:

3.2.1. OPENCV

Open Source Computer Vision Library (OpenCV) es una librería de visión y aprendizaje automático por computadora de código abierto. Su objetivo es proporcionar una infraestructura común

para aplicaciones de visión por computadora. Tiene interfaces en C++, C, Python, y Java, con soporte para los sistemas operativos Windows, Linux, Mac OS, iOS y Android. OpenCV fue diseñado con un fuerte énfasis en aplicaciones que se ejecutan en tiempo real.

Esta librería cuenta con más de 2.500 algoritmos optimizados de visión por computadora y aprendizaje automático. Estos algoritmos permiten detectar y reconocer rostros, identificar objetos, clasificar las acciones humanas en videos, seguir movimientos de la cámara, seguir objetos en movimiento, extraer modelos 3D de objetos, producir nubes de puntos 3D a partir de cámaras estéreo, unir imágenes para producir una imagen de alta resolución de una escena completa, encontrar imágenes similares de una base de datos de imágenes, eliminar los ojos rojos de las imágenes tomadas utilizando el flash, seguir los movimientos de los ojos, reconocer el paisaje y establecer marcadores para revestirlo de la realidad aumentada, entre otros.

OpenCV tiene más de 47 mil personas en su comunidad de usuarios. Parte de la comunidad es Willow Garage un instituto

de robótica formado por un equipo de expertos en diseño de robots, control, percepción, y aprendizaje automático por computadora que utiliza OpenCV y ROS para detectar y categorizar objetos en 3D [14].

3.2.2. PCL

Cloud Library Point (PCL) es de código libre, para el procesamiento de imágenes 2D/3D y nubes de puntos. Una nube de puntos es una estructura de datos utilizada para representar una colección de puntos multidimensionales y se utiliza comúnmente para representar los datos en tres dimensiones. En una nube de puntos 3D, los puntos por lo general representan puntos en coordenadas geométricas X, Y y Z de una muestra subyacente. Las nubes de puntos se crean habitualmente con un láser escáner tridimensional. Este instrumento mide de forma automática la distancia de un gran número de puntos en la superficie de un objeto.

PCL contiene numerosos algoritmos que incluyen el filtrado, estimación de funciones, reconstrucción de la superficie, el modelo adecuado y segmentación. Estos algoritmos se pueden utilizar, por ejemplo, para filtrar los valores extremos de los

datos ruidosos, segmentar partes correspondientes de una escena, extraer puntos clave para reconocer objetos en el mundo en base a su apariencia geométrica, y recrear superficies desde nubes de puntos y visualizarlas.

3.2.3. GAZEBO

Es un simulador de robot que permite probar rápidamente algoritmos, diseño mecánico de robots y realizar pruebas de regresión, con la posibilidad de simular con precisión y eficacia cualquier tipo de robots en entornos interiores y exteriores complejos. Se pueden generar datos de sensores, opcionalmente con ruido, de láser, cámara 2D/3D, de contacto y muchos más.

Utiliza las librerías ODE, Bullet, Simbody y Dart para simular la dinámica del cuerpo y soporta múltiples robots, Gazebo es útil para experimentos que incluyen el movimiento de planificación, control y percepción de vehículos de tierra y los objetos en el mundo virtual son descritos en un archivo .world utilizando formato XML. Para la representación de ambientes, gazebo usa la librería Ogre, que permite la iluminación de alta calidad, sombras y texturas.

Además, se puede realizar la simulación en servidores remotos, a través de mensajes basados en sockets que se envían por medio de google Protobufs. Los protocolos buffers (Protobuf) son una forma de codificar datos estructurados en un eficiente formato extensible. Google usa los Protocolos Buffers para casi todos sus protocolos RPC internos y formatos de archivo [15].

CAPÍTULO 4

4. ARQUITECTURA DEL FRAMEWORK SELECCIONADO

4.1. ANÁLISIS EN ALTO NIVEL DEL FUNCIONAMIENTO INTERNO DEL FRAMEWORK

Buena parte de este capítulo fue escrita basado en la información disponible en la wiki oficial de ROS que está protegida bajo los términos de la licencia Creative Commons Attribution 3.0 y que complementamos con la experiencia y conocimientos que hemos adquirido durante nuestra experiencia en el Centro de Visión y Robótica de la ESPOL (CVR) y la elaboración de nuestra tesis [16].

Un robot en ROS es representando por uno o más procesos llamados nodos que son agrupados en un grafo. Los nodos representan a cada uno de los componentes del robot, sean físicos o virtuales, y son independientes unos de otros. Es decir, un nodo de un robot puede ejecutarse incluso si los demás no están en ejecución, por esta razón se dice que están ligeramente acoplados.

Para poder desempeñar las complejas tareas que realiza un robot es necesario que todos los nodos involucrados en realizar una tarea se comuniquen entre sí. ROS representa la comunicación de sus nodos por medio de una red punto-a-punto (P2P, por sus siglas en inglés peer-to-peer). Los nodos usan el paradigma de cliente/servidor. Un nodo puede actuar como cliente, servidor, o cliente y servidor respecto a los demás nodos de la red. Los mecanismos de comunicación que implementa ROS, incluyen:

- Comunicación sincrónica similar a RPC (Remote Procedure Call) que permite a un nodo comunicarse con otro a través de la invocación de una función remota.
- Comunicación asincrónica de datos a través de mensajes.
- Almacenamiento de datos compartidos.

Estos mecanismos se describen en detalle en la sección 4.1.1.1.

4.1.1. NIVELES DE ROS

La arquitectura de ROS se divide en tres niveles: el nivel de Grafo de Cómputo, el nivel de Sistema de Archivos, y el nivel Comunitario. Todos los elementos que conforman cada uno de los niveles son considerados como recursos por ROS.

4.1.2. GRAFO DE CÓMPUTO DE ROS

El conjunto de procesos de ROS que procesan datos de manera coordinada en un robot se llama Grafo de Cómputo. Todos los robots en ejecución se encuentran representados a través del conjunto de los recursos que usa. El grafo contiene varios tipos de recursos: Nodos, el nodo Maestro, Servidor de Parámetros, Mensajes, Servicios, Tópicos, y Bags. A continuación se describe cada uno de ellos:

- **Nodos:** Son procesos que realizan alguna tarea definida por el programador del sistema robótico. Cada nodo puede estar escrito en un lenguaje de programación diferente y utilizar el API de ROS.
- **Maestro:** Es el nodo (proceso) que permite registrar los recursos del Grafo de Cómputo de ROS y buscarlos por sus

nombres. Sin el Maestro, los nodos no son capaces de encontrarse entre ellos e intercambiar información.

- **Servidor de Parámetros:** Almacena datos en una ubicación central utilizando variables de entorno (conocidas como parámetros) y referenciarlos mediante el identificador de la variable. Actualmente se ejecuta como parte del Maestro pero se estudia de forma separada debido a que tiene un API independiente.
- **Mensajes:** Los mensajes son estructuras de datos que comprenden campos de algún tipo de dato. Soportan tipos de datos primitivos y arreglos de los mismos. También pueden incluir tipos de datos abstractos (similar a las estructuras de C). No se pueden pasar referencias de memoria como un mensaje.
- **Tópicos:** Un tópico es una interfaz de comunicación asincrónica que es identificada usando un nombre. Los mensajes son dirigidos a través de un esquema de publicadores/subscriptores a los tópicos. Un nodo envía datos publicando un mensaje en un tópico dado y otro nodo interesado en esos datos se subscribirá a dicho tópico.
- **Servicios:** Es un mecanismo de comunicación sincrónico que está definido por un par de estructuras de mensajes: una

para la solicitud y otra para la respuesta. Un nodo proveedor ofrece un servicio bajo un cierto nombre y un cliente usa ese servicio enviando un mensaje de solicitud en espera de un mensaje de respuesta. Las librerías cliente de ROS generalmente presentan esta interacción al programador como si se tratase de una llamada a un procedimiento remoto (RPC).

- **Bags:** Mecanismo para guardar y reproducir datos de mensajes de ROS enviados usando tópicos o servicios, tales como datos de sensores, que en ocasiones pueden ser difíciles de guardar pero que son necesarios para desarrollar y probar algoritmos. Pueden ser configurados de tal forma que el investigador defina qué guardar.

Algunos de estos recursos del Grafo de Computo de ROS son analizados más a detalle en la sección 4.1.4. Todos estos recursos proveen información al grafo de formas diferentes y se identifican mediante nombres. Más información acerca del sistema de nombre de ROS se da en la sección 4.1.2.

4.1.2.1. SISTEMA DE ARCHIVOS DE ROS

El nivel de Sistema de Archivos es una estructura jerárquica similar al sistema de archivos de un sistema

operativo. Contiene aquellos recursos que se encuentran en el disco duro, organizados en:

- **Paquetes:** Un paquete es un directorio. Los paquetes son el elemento de construcción más atómica. Un paquete puede contener nodos, una librería de la que dependa ROS, un conjunto de datos, archivos de configuración, o cualquier otra cosa que sea útil agrupar. Cada paquete cuenta con un archivo de manifiesto (manifest.xml) que provee información acerca del mismo, su licencia, dependencias, y variables de compilación (banderas).
- **Pilas:** Son conjuntos de paquetes que se encuentran dentro de un mismo directorio. Se usan para proporcionar una funcionalidad agregada por ejemplo la “pila de navegación 2D”. También son la forma en la cual se distribuyen y se asocian los números de las versiones de ROS. Un conjunto de pilas versionadas pueden formar una distribución de ROS. Cada pila también contiene un archivo de manifiesto (stack.xml) que provee información sobre una pila, incluyendo su licencia y la dependencia de otras pilas.

- **Definición de datos contenidos en el mensaje:** Los datos de la estructura de los mensajes se definen en archivos de texto de la forma: tipodato1 nombrecampo1 (ej: float32 cell_width). Existen archivos de Tipos de datos de Mensajes y también de Servicios:
  - **Tipos de Mensaje:** Son archivos de extensión .msg que definen las estructuras de los mensajes que se envían a través de los tópicos de ROS. Se almacenan en el directorio mi_paquete/msg/.
  - **Tipos de Servicio:** Son archivos de extensión .srv que definen las estructuras de datos de la solicitud y la respuesta para los servicios en ROS. Se almacenan en el directorio mi_paquete/srv/.

4.1.2.2. NIVEL COMUNITARIO

En el nivel comunitario de ROS se refiere a aquellos medios a través de los cuales quienes usan la plataforma intercambian información acerca de recursos ya sea conocimiento, código fuente o incluso ideas. Estos medios son:

- **Distribuciones:** Las distribuciones de ROS son conjuntos de pilas y paquetes que tienen una versión consistente entre ellos y que se pueden instalar fácilmente.
- **Repositorios:** La red de repositorios en los cuales diferentes instituciones y comunidades publican el código de sus componentes.
- **Wiki de ROS:** Es el foro principal para documentar la información acerca de ROS cualquiera se registra y contribuye con su propio conocimiento publicando revisiones, correcciones o nuevas ideas.
- **Bug Ticket System:** Utilizado para solicitar correcciones o nuevas características a los desarrolladores principales de ROS.
- **Respuestas de ROS (ROS Answers):** Es un sitio web donde los desarrolladores responden preguntas relacionadas al funcionamiento de ROS.
- **Blog:** El blog de Willow Garage provee actualizaciones regulares de información respecto a ROS.

4.1.3. BUSQUEDA E IDENTIFICACIÓN DE RECURSOS EN ROS

Los nombres de los recursos además de cumplir el papel de identificadores sirve para relacionar los diferentes niveles de la

arquitectura de ROS. Los nombre se registran para poder encontrar cada recurso después. Dependiendo si se trata de un recurso almacenado en memoria o en el sistema de archivos, ROS aplica las siguientes reglas:

4.1.3.1. BUSQUEDA DE NOMBRES DE LOS RECURSOS EN EL GRAFO DE COMPUTO

Los nombres de los recursos que se encuentran cargados en memoria dentro del grafo tienen una estructura jerárquica. Estos nombres son un eje central con respecto a cómo se componen los sistemas complejos en ROS. Es por esto que describiremos su funcionamiento antes de ahondar más en el detalle de los recursos del Grafo de Cómputo de ROS.

Los nombres de los recursos en el grafo sirven para proveer encapsulamiento. Cada recurso es definido dentro de un espacio de nombres que no es más que un área virtual que puede ser compartida con muchos otros recursos, bajo las cual ciertos nombres de recursos tienen validez. El encapsulamiento aísla diferentes porciones del sistema para evitar utilizar accidentalmente un recurso equivocado. Esto evita la ocurrencia de

conflictos entre nombres iguales de recursos de diferentes robots.

En general, un recurso puede crear otro recurso dentro de su espacio de nombres, y cada recurso puede acceder a otros recursos que se encuentren dentro o por encima del espacio de nombres en el que se encuentra. Se pueden realizar conexiones entre recursos incluso cuando se encuentran en espacios de nombres distintos, pero esto generalmente es hecho implementando funcionalidades en un nivel que se encuentre por encima de ambos espacios de nombres.

La resolución de nombres es relativa, por lo que los recursos no necesitan saber dentro de cual espacio de nombres se encuentran. Esto simplifica la programación de nodos que trabajen en conjunto. Los nodos de un robot pueden ser escritos como si todos ellos estuvieran en un nivel superior del espacio de nombres. Cuando estos nodos se integran a un sistema más grande, ellos pueden ser colocados dentro de un nuevo espacio de nombres que defina su colección de código en un nivel inferior. De esta forma se evita sobrecargar recursos

cuyos nombres coincidan con los de otros recursos ya existentes en el sistema más grande y se asegura que los nodos añadido sigan teniendo los recursos que necesitan a la mano.

Existen cuatro tipos de Nombres de los Recursos en el Grafo en ROS: base, relativo, global y privado, y tienen la siguiente sintaxis:

- base
- relativo/nombre
- /global/nombre
- ~privado/nombre

Por defecto, la resolución es hecha con respecto al espacio de nombres del nodo (dentro del mismo nivel). Los nombres sin calificadores de espacio de nombres se consideran como nombres de base, que son de hecho una subclase de nombres relativos a los que se aplica la misma regla de resolución y son frecuentemente utilizados para inicializar el un nodo.

Los nombres que empiezan con una barra diagonal ' / ' son globales y se consideran que están completamente resueltos. Es decir, no hace falta buscar el espacio de

nombres en el que se encuentran. Estos deben ser evitados tanto como sea posible ya que limitan la portabilidad del código.

Los nombres que empiezan con una tilde ' ~ ' son considerados como privados. Esto convierte los nombres de nodos en espacios de nombres. Son muy útiles para el envío de parámetros a un nodo específico a través del servidor de parámetros.

La resolución de los nombres de los Recursos en el Grafo se ilustra en la siguiente tabla:

Tabla 4: Resolución de Nombres de Recursos del Grafo

	Relativa		**Global**		**Privada (nodo se convierte en espacio de nombres)**	
Nodo	**Referencia**	**Resolución**	**Referencia**	**Resolución**	**Referencia**	**Resolución**
/nodo1	YYY	/YYY	/YYY	/YYY	~YYY	/nodo1/YYY
/XX/nodo2	YYY	/XX/YYY	/YYY	/YYY	~YYY	/XX/nodo2/YYY
/XX/nodo3	ZZZ/YYY	/XX/ZZZ/YYY	/ZZZ/YYY	/ZZZ/YYY	~ZZZ/YYY	/XX/nodo3/ZZZ/YYY

En la *Tabla 4*, podemos ver como se lleva a cabo una resolución de nombres para tres escenarios diferentes. Asumiendo que nos encontramos en los nodos que se indican dentro de la primera columna del lado izquierdo y dentro de ellos se hace referencia a los recursos en las columnas siguientes.

4.1.3.2. BUSQUEDA DE NOMBRES DE RECURSOS EN EL SISTEMA DE ARCHIVOS

Dentro del sistema de archivos la forma de hacer esta referencia es muy sencilla, basta con colocar el nombre del paquete en el que se encuentra el recurso para poder acceder a el. Algunos de los archivos de ROS a los que se puede hacer referencia usando el nombre del paquete del recurso incluyen:

- Tipos de mensajes (msg)
- Tipos de servicios (srv)
- Tipos de nodos.

Los nombres de los recursos del sistema de archivos son muy similares a las rutas de los archivos y directorios del sistema operativo, con la diferencia de que son mucho más cortos y simples. Están formado solo por el nombre del paquete que contiene al recurso, y el nombre del recurso. Esto se debe a la habilidad que tiene ROS de localizar paquetes en el disco y hacer suposiciones adicionales acerca de sus contenido.

Por ejemplo, las descripciones de los mensajes siempre se alojan en la sub-ruta msg y tienen la extensión .msg

de modo que std_msgs/String es el modo abreviado de referirse al archivo /ruta/a/std_msgs/msg/String.msg. Si hiciéramos referencia a un nodo de la forma X/Y estaríamos diciendo que buscamos un archivo llamado Y que tiene permisos de ejecución y se encuentra dentro del paquete X.

4.1.4. LIBRERÍAS CLIENTE DE ROS

Una librería cliente de ROS es una colección de código que facilita el trabajo de los programadores de ROS. Estas toman muchos de los conceptos de ROS, los implementan y los hace disponible a través de API's para diferentes lenguajes de programación. Las API's de ROS sirven para escribir nodos, publicar y suscribirse a los tópicos, escribir e invocar servicios, y utilizar el Servidor de Parámetros.

Las librerías cliente pueden ser implementadas en cualquier lenguaje de programación, sin embargo actualmente el objetivo de ROS es proveer un soporte principalmente para C++ y Python. Las librerías clientes de ROS implementan una misma funcionalidad base con la diferencia de que buscan tomar ventaja de las características del lenguaje de programación en el que están escritas para proveer diferentes ventajas respecto

a las demás librerías clientes.

En la práctica un robot en ROS puede estar formado por muchos nodos y cada uno de ellos puede estar escrito utilizando un lenguaje de programación diferente siempre que exista una librería cliente para ese lenguaje. Las principales librerías clientes en ROS son:

- **roscpp:** da soporte al lenguaje C++. Esta es la librería cliente mayormente utilizada y está diseñada para ser la librería de alto desempeño de ROS (menor tiempo de respuesta, menor uso de CPU, menor uso de memoria). En principio es posible crear nodos que se ejecuten en tiempo real y usen las llamadas al sistema operativo.
- **rospy:** da soporte al lenguaje Python, y fue creada para proveer a ROS las ventajas de los lenguajes orientados a objetos. El diseño de esta librería promueve la velocidad de codificación por sobre el desempeño en tiempo de ejecución. Es útil para elaborar y probar rápidamente algoritmos. También es ideal para programar porciones de código que no contengan sección crítica tal como la configuración e inicialización de variables.

- **roslisp:** es una librería para dar soporte al lenguaje LISP. Permite implementar nodos rápidamente y depurar de forma interactiva una instancia de ROS en ejecución. A pesar de que esta librería forma parte del núcleo de ROS aún se considera que se le pueden implementar mejoras por lo que está sujeta a cambios.

Además de las librerías clientes, ROS provee diferentes herramientas que permiten al investigador interactuar con un robot en ejecución sin utilizar nodos. Muchas de estas herramientas de ROS están escritas utilizando rospy para tomar ventaja de su capacidad para determinar el tipo de un objeto en tiempo de ejecución. El Maestro de ROS, el roslaunch, roscore, rostopic, rosservice, rosparam y otras herramientas han sido desarrolladas usando rospy.

4.1.5. FLUJO DE EJECUCIÓN DE ROS

Una vez que todos los recursos de un robot han sido definidos/implementados en ROS el software de control de robot en ROS se ejecuta invocando al “núcleo de ROS” en un terminal con el comando roscore y luego se carga el paquete con los nodos del robot usando el comando roslaunch. El

comando roslaunch utiliza un archivo de configuración (con extensión .launch) que se encuentra dentro del paquete para saber cuáles son los nodos que debe de ejecutar y sus parámetros.

Como parte del "núcleo de ROS" se ejecuta el Maestro, que actúa como un proveedor de servicio de nombres en el Grafo de Cómputo de ROS. Así como los nodos se registran con el Maestro, ellos pueden recibir información acerca de los otros nodos. El Maestro provee la función de búsqueda de los nombres, similar a un servidor DNS (Domanin Name System).

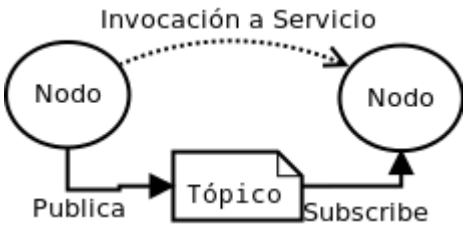


Figura 4. 1: Formas de comunicación entre nodos

Como se muestra en la *Figura 4.1*, los nodos intercambian datos con otros nodos a través de tópicos o servicios directamente una vez que se realiza la conexión entre ellos. La forma en que se realiza la comunicación a través de tópicos y servicios es detallada en las secciones 4.1.4.3 y 4.1.4.4 respectivamente.

El Maestro también hace llamadas de regreso a los nodos (callbacks) cuando la información de registro de los recursos que utiliza o provee un nodo cambia, lo que permite a los nodos crear conexiones dinámicamente a medida que nuevos nodos se ejecuten.

Todas las librerías clientes de ROS soportan reasignación de nombres por línea de comandos, lo que significa que un programa ya compilado puede ser reconfigurado en tiempo de ejecución para que se ejecute con otro nombre o incluso opere en un espacio de nombres diferente dentro del Grafo de Cómputo de ROS. Esta arquitectura hace posible que se realicen operaciones desacopladas entre nodos, es decir que podemos inicializar, terminar, y reiniciar los nodos en cualquier orden sin incurrir en una condición de error.

4.1.6. EL NUCLEO DE ROS

Es una colección de nodos y programas necesarios para poder ejecutar un sistema basado en ROS. Para poder ejecutar un nodo y que este pueda comunicarse con otros nodos, siempre será necesario que exista una instancia del proceso roscore en ejecución.

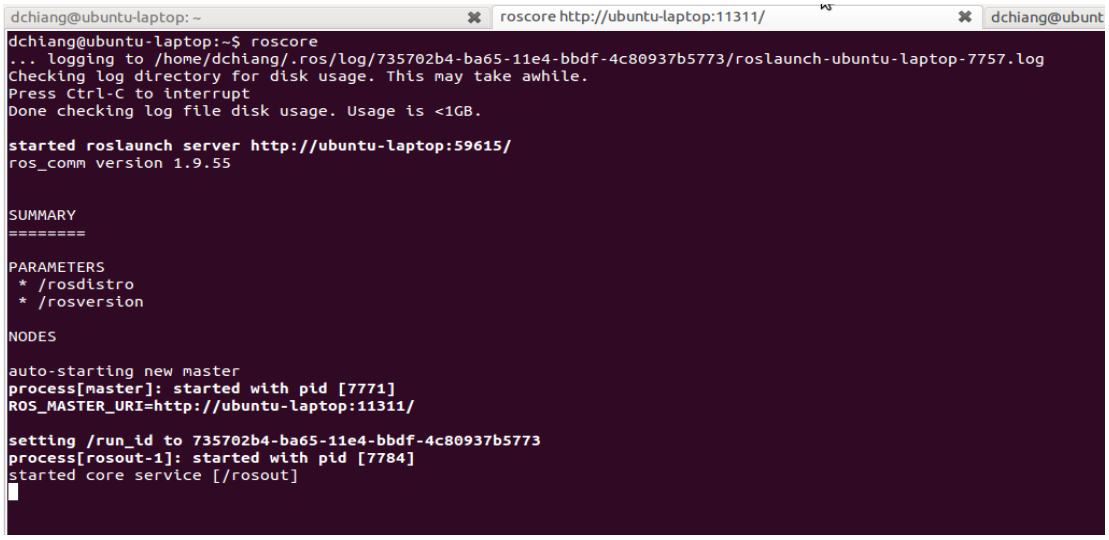


Figura 4. 2: Ejecución del núcleo de ROS

En la *Figura 4.2*, se muestra la salida que se genera cuando se ejecuta el núcleo de ROS. Los elementos que se cargan cuando se ejecuta el programa roscore son:

1. Maestro de ROS
2. Servidor de Parámetros de ROS
3. rosout

Como se muestra en la *Figura 4.3*, cuando preguntamos al sistema operativo acerca de los procesos que se ejecutan inmediatamente después de ejecutar el programa roscore que levanta el núcleo de ROS,vemos que no se muestra el Servidor de Parámetros. Esto ocurre ya que como mencionamos anteriormente el Servidor de Parámetros se ejecuta como parte del Maestro.

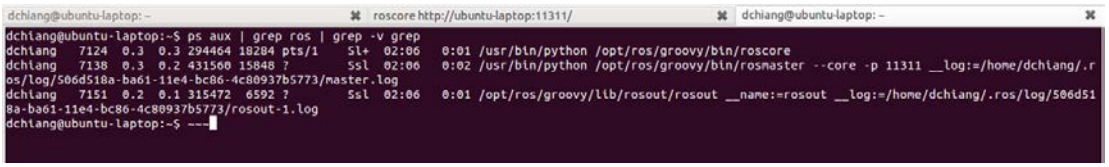


Figura 4. 3: Procesos del núcleo de ROS.

4.1.6.1. MAESTRO

El Maestro está escrito usando Python. Implementa el protocolo de llamada a procedimiento remoto XML-RPC debido a que se trata de un protocolo que es soportado por una gran variedad de lenguajes de programación. El XML-RPC almacena los datos como XML y los transmite utilizando el protocolo HTTP.

La forma de interactuar con el Maestro es a través de las API's (librerías clientes de ROS) que contienen funciones de registro. El cual permite que los nodos se registren como publicadores, suscriptores, o proveedores de servicios. Adicionalmente el maestro dispone un Identificador de Recursos Uniformes (URI por sus siglas en ingles), que es una cadena de caracteres que identifica los recursos de una red de forma unívoca. Esta URI corresponde al host:puerto en que se está ejecutando el servidor XML-RPC además del nombre del

recurso al que hace referencia. El puerto de red por defecto al cual está relacionado el Maestro es el 11311.

4.1.6.2. SERVIDOR DE PARÁMETROS

Es un conjunto de parámetros (variables) compartidas dentro de ROS. Los nodos lo utilizan para extraer parámetros en tiempo de ejecución. Al igual que el Maestro, utiliza XML-RPC lo cual facilita su integración con las librerías clientes de ROS. Puede almacenar datos escalares como los enteros de 32 bits, booleanos, cadenas de texto (strings), números de punto flotante de doble precisión (doubles), fechas y horas en el estándar ISO-8601, listas, datos binarios codificados en base 64, y diccionarios. Pese a que el Servidor de Parámetros es realmente parte del Maestro tiene un API independiente.

El Servidor de Parámetros usa un diccionario de diccionarios para representar los espacios de nombres, donde cada diccionario representa un nivel en la jerarquía de nombres. Cada llave '{ }' en el diccionario representa un espacio de nombres, dando como resultado un conjunto del tipo { clave : valor }. Si el valor

es otro diccionario, el Servidor de Parámetros asume que está almacenando los valores de otro espacio de nombres. Por ejemplo, si quisiéramos definir el parámetro /en1/en3/foo (en = espacio de nombres) y asignarle el valor 1, el valor de /en1/en3/ sería el diccionario { foo : 1 } y el valor de /en1/ sería el diccionario { en3 : { foo : 1 } }. Lo mismo ocurriría si quisiéramos definir los parámetros detallados en la *Tabla 5* y se ilustra en la *Figura 4.4*.

Tabla 5: Diccionario del servidor de parámetros

PARÁMETRO	VALOR (DICCIONARIO)
/en1/en3/foo	1
/en1/en3/	{ foo : 1 }
/en1/foobar	1
/en1/	{ en3 : { foo : 1 } , foobar : 1 }
/en2/bar	1
/en2/	{ bar : 3 }

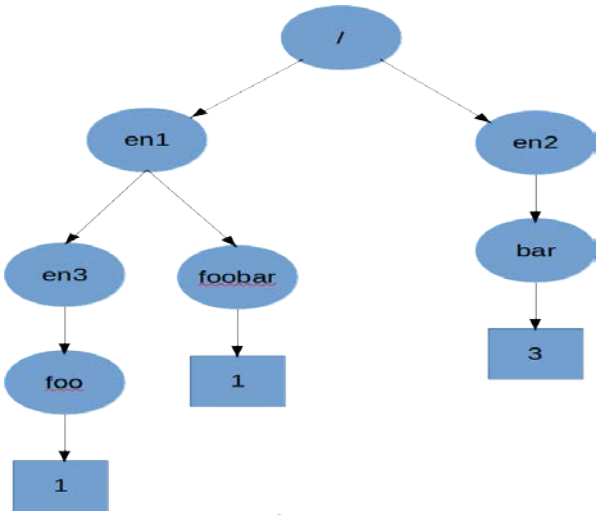


Figura 4. 4: Servidor de Parámetros Representado en Árbol.

Sin importar en que lenguaje desarrollemos es posible integrar las llamadas al Servidor de Parámetros de ROS, siempre que se tenga acceso a una librería XML-RPC cliente incluso sin la necesidad de utilizar alguna de las librerías cliente de ROS.

4.1.6.3. ROSOUT

Es el nombre del mecanismo de registro de información en ROS que está basado en tópicos. Es un equivalente de los dispositivos de salida estándar stdout y stderr. Cada librería cliente registra información diferente en rosout. Los niveles de detalle con que se registran los mensajes en ROS son los siguientes:

DEPURADO (DEBUG): Información que no es requerida siempre que el sistema funcione de forma adecuada. Ejemplo:

1. Mensaje recibido en el tópico X desde publicador Y.
2. 100 bytes enviados a través del socket 3.

INFORMATIVO (INFO): Cantidades pequeñas de información que puede ser útil para los usuarios. Ejemplo:

1. Nodo inicializado.
2. Mensaje tipo Y se ha publicado en el tópico X.
3. Nuevo subscriptor al tópico X: Y

ADVERTENCIA (WARN): Información que puede resultar alarmante para el usuario, que pueda afectar el procesamiento de la data y su resultado, pero que es parte de lo que se espera del sistema robótico. Ejemplo:

1. No se pudo cargar el archivo de configuración desde la <ruta_a_archivo>. Se utiliza la configuración por defecto.

ERROR: Alguna tarea importante ha tenido problemas. Ejemplos:

1. No se ha recibido una actualización a través del tópico X durante 10 segundos. Se detendrá la ejecución del robot hasta que continúe la transmisión.
2. Se recibe valor inesperado NaN (no numérico) en la transformada X. Se salta...

FATAL: Algo irrecuperable ha ocurrido. Ejemplo:

1. Los motores se han incendiado!.

En la *Tabla 6*. se muestra con 'x' los niveles de detalle que se pueden registrar en rosout cuando se utilizan las librerías roscpp o rospy.

Tabla 6: Niveles de Detalle que se Puede Registrar en rosout.

	Depurado	Informativo	Advertencia	Error	Fatal
roscpp	x	x	x	x	x
rospy	o	x	x	x	x

4.1.6.4.NODOS

Un nodo puede contener los algoritmos de control, adquisición y manipulación de datos del robot. Pueden ser escritos utilizando lenguajes de programación diferentes así que los investigadores pueden llamar en un robot a funciones de librerías de programación tales como OpenCV, PCL, Gazebo, entre otras. Incluso es posible integrar ROS con otros frameworks robóticos.

Los nodos de ROS utilizan algunas APIs nativas de ROS:

- API Esclavo (Slave API): Es un API que le permite al nodo comunicarse con el Maestro para solicitar información de otros recursos y también permite

establecer la conexión a través de un tópico con otro nodo. Usualmente este API es encapsulado por las librerías clientes de ROS de modo que el investigador no tiene necesidad de usar esta Api directamente.

- API de protocolos de transporte de datos. Estas permiten establecer conexiones a los tópicos y servicios usando un protocolo acordado. Las librerías de protocolos mayormente utilizados son TCPROS y UDPROS.
- API de línea de comandos (command-line API): Cada nodo debe soportar la reasignación de parámetros a través de línea de comandos, esto permite configurar los nombres dentro del nodo en tiempo de ejecución.

4.1.6.5. TÓPICOS

La transportación de los mensajes de datos a través de los tópicos es negociada cuando un subscriptor solicite al servidor XML-RPC una conexión a un tópico. El subscriptor envía al publicador una lista de protocolos soportados. El publicador selecciona un protocolo de esa lista, y retorna las configuraciones necesarias para ese protocolo. Entonces el subscriptor establecerá una

conexión separada utilizando las configuraciones recibidas.

Un ejemplo de intercambio de datos entre nodos usando un tópico sería:

- Nodo A se registra como subscriptor a un tópico.
- Nodo B se registra como publicador a ese mismo tópico.
- El nodo subscrito A queda en espera hasta que recibe un mensaje enviado por el nodo publicador B a través del tópico.
- El mensaje es tratado por una llamada de retorno (callback) en el nodo A donde continua con su tarea asociada al mensaje.
- El nodo A vuelve a su estado de espera. Cabe recalcar que esta espera debe ser implementada por el investigador invocando al método spin() que implementa la técnica sleeping que genera un pequeño bloqueo y permite que los demás hilos se ejecuten.

ROS no está comprometido a un solo protocolo de

transporte. Utilizando la URI de un publicador, un nodo suscriptor negocia una conexión, usando el transporte adecuado con ese publicador a través de XML-RPC. El resultado de la negociación es que los dos nodos están conectados, y se genera un flujo de mensajes del publicador al subscriptor sin pasar por el maestro.

Cada método de transporte tiene su propio protocolo para el intercambio de mensajes de datos. Por ejemplo, usando TCP, la negociación implicaría que el publicador proporcione al suscriptor la dirección IP y puerto al cual debe conectarse. El subscriptor entonces creará un socket TCP/IP a la dirección y puerto especificados. Los nodos intercambian una cabecera de conexión (Connection Header), y luego el publicador empieza a enviar mensajes de datos directamente a través del socket.

4.1.6.5.1.ESTABLECIENDO CONEXIÓN A UN TÓPICO

Los nodos que se subscriben a un tópico solicitan conexiones desde los nodos que publican a ese tópico, y esa conexión se

establecerá a través de un protocolo de conexión acordado. El protocolo mayormente utilizado en ROS es llamado TCPROS.

En la Figura 4.5 se ilustran las acciones que permiten que se realice un intercambio de mensajes entre nodos a través de un tópico. Estas son:

- Se ejecuta el nodo subscriptor A y lee sus parámetros para saber el nombre que lo identifica.
- El subscriptor A registra su nombre con el Maestro utilizando XML-RPC.
- El publicador B de igual forma se registra con el Maestro.
- El maestro informa al subscriptor A acerca del nuevo publicador B.
- El subscriptor A contacta al publicador B para solicitar una conexión a un tópico y acordar el protocolo de comunicación.
- El publicador B envía al suscriptor A las

configuraciones necesarias para el protocolo de comunicación seleccionado.

- El subscriptor A se conecta al publicador B usando el protocolo seleccionado

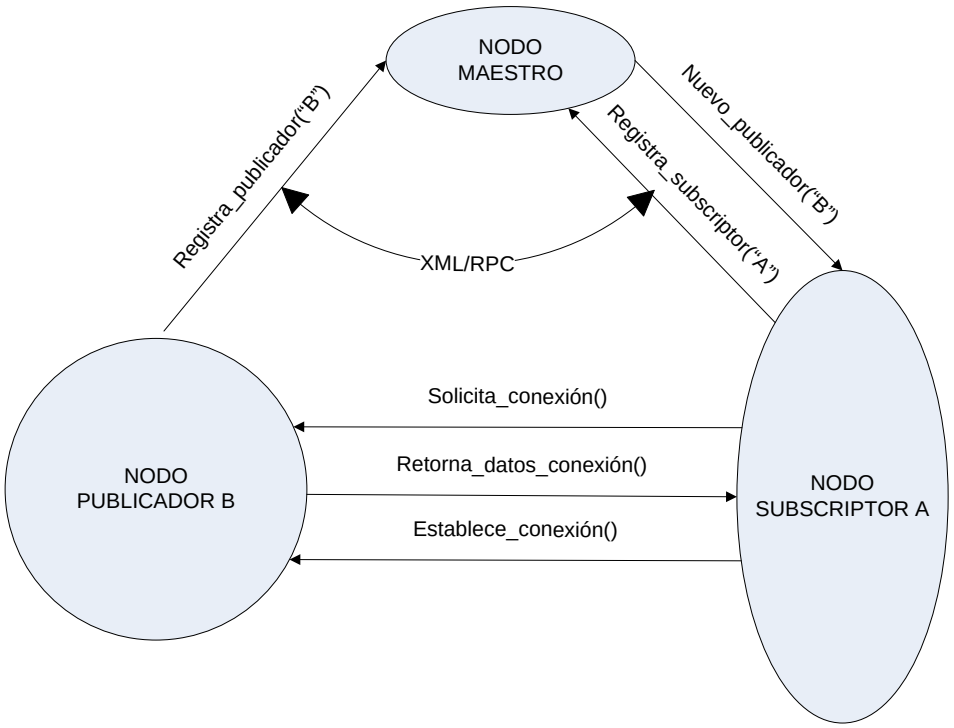


Figura 4. 5: Estableciendo Conexión A través de Tópicos.

4.1.6.6.INVOCACIÓN A SERVICIOS

El modelo de publicadores/subscriptores de los tópicos es un paradigma de comunicación bastante flexible, pero tiene una relación de muchos a muchos, es de una sola vía lo cual no lo hace apropiado para interacciones de solicitud/respuesta y que es a menudo requerido en sistemas distribuidos. Una forma sencilla de analizar a

los servicios es verlos como una versión simplificada de tópicos. A diferencia de los tópicos donde pueden haber muchos nodos que publican en el, para cada servicio solo puede haber un nodo que lo provee. Siempre se considera al último nodo que se registra con el Maestro como el nodo que provee el servicio. La forma en que se lleva a cabo esta conexión es la siguiente:

1. El nodo que provee el servicio se registra con el Maestro.
2. Los nodos clientes del servicio crean conexiones TCP/IP al servicio.
3. Los nodos clientes y el nodo proveedor del servicio intercambian una cabecera de conexión.
4. Los nodos clientes envían mensajes de datos.
5. El nodo proveedor del servicio responde con un mensaje de respuesta.

Debido a que no se utiliza devolución de llamadas (callbacks) desde el Maestro cuando se registra un nuevo servicio, muchas librerías cliente proveen un API de “espera por servicio” que pregunta constantemente al

Maestro si se ha registrado un nuevo servicio (polling). Los servicios incluyen un byte 'ok' en el mensaje de respuesta a cada solicitud de servicio. Si el byte es verdadero (1), será acompañado por la respuesta solicitada. En caso que el byte sea falso (0) estará acompañado por una cadena vacía.

Por defecto las conexiones a un servicio son libres de estados (stateless), es decir que cada solicitud es tratada como una transacción independiente que no está relacionada a ninguna solicitud anterior. No se mantiene información de una sesión o su estado. Por cada invocación a servicio el nodo cliente, solicita información al Maestro e intercambia datos de solicitud/respuesta a través de una nueva conexión con el servicio.

Las operaciones libres de estados son generalmente más robustas ya que permiten a los nodos de servicios ser reiniciados, pero esta sobrecarga puede ser alta si se invoca de forma repetida al mismo servicio.

4.1.6.6.1. CONEXIÓN PERSISTENTE DE SERVICIOS

ROS permite realizar conexiones persistentes a servicios, lo cual provee una conexión de mejor rendimiento al momento de realizar llamadas repetidas a un servicio. Debido a que la conexión entre el cliente y el servicio se mantiene abierta para que el cliente del servicio pueda continuar enviando solicitudes usando la misma conexión en lugar de negociar nuevas conexiones para cada petición. Se debe tener cuidado al momento de utilizar conexiones persistentes. Si un nuevo proveedor de servicio aparece, este no interrumpe una conexión en marcha. Si una conexión persistente falla, no se realiza ningún intento de reconexión.

4.1.7. PROTOCOLOS DE COMUNICACIÓN

Existen muchas formas de enviar datos a través de una red, y cada una de ellas tiene sus ventajas y desventajas, dependiendo en gran medida de la aplicación. El protocolo TCP es ampliamente utilizado porque provee un flujo de comunicación simple y confiable. Los paquetes TCP siempre

llegan en orden, y los paquetes perdidos son reenviados hasta que lleguen a su destino. Si bien es muy útil en redes cableadas, estas características llegan a ser una fuente de errores cuando se trabaja utilizando una red WiFi o un módem de conexión celular. En esta situación, el protocolo UDP es más apropiado. Cuando múltiples suscriptores están agrupados en una única subred, podría ser más eficiente para el publicador comunicarse con todos ellos simultáneamente a través de una difusión UDP.

4.1.7.1. TCPROS

Es una capa de transporte para mensajes y servicios en ROS. Utiliza sockets TCP/IP estándar para transportar datos de mensajes. Las conexiones entrantes son recibidas a través de un servidor de sockets TCP con una cabecera de conexión que contiene los tipos de datos del mensaje y la información de enrutamiento.

Las conexiones entrantes a un servidor de sockets TCPROS son enrutadas usando la información contenida en los campos de la cabecera de conexión. Si la cabecera contiene el campo 'topic', esta será enrutada como una conexión a un nodo publicador en ROS. Si la

cabecera contiene el campo 'service', esta será enrutada como una conexión a un nodo de servicio en ROS.

4.1.7.2.UDPROS

Es una capa de transporte de mensajes y servicios que aún está en desarrollo. Utiliza paquetes de datagramas UDP estándar para el transporte de datos de mensaje. Es muy útil cuando la latencia es más importante que un transporte de mensajes y servicios confiable. Entre sus usos está el control remoto de robots y la transmisión de audio. UDPROS intercambia la cabecera de conexión durante la negociación XML-RPC donde se negocia el tamaño máximo de los datagramas que serán enviados..

Cuando se utiliza UDPROS los mensajes de ROS son enviados como una serie de datagramas. El primer datagrama contiene el total de número de datagramas esperados. Cada datagrama contiene una cabecera seguida de mensajes de datos. La máxima cantidad de mensajes de datos por datagrama es determinada durante la negociación de conexión. Solo el datagrama final de mensaje va a contener menos del número

máximo de bytes permitidos. La cabecera del datagrama contiene la siguiente información:

ID de conexión. Es un valor de 32 bits determinado durante la negociación de conexión y es usado para denotar a que conexión está destinado el datagrama. Este parámetro hace posible que en un mismo socket provea múltiples conexiones UDPROS.

Opcode. UDPROS soporta múltiples tipos de datagramas. El tipo de datagrama es especificado por este campo que está formado por 8 bits y puede tener uno de los siguientes valores:

- DATA0 (0) – Este opcode es enviado en el primer datagrama de un mensaje de ROS.
- DATAN (1) – Todos los datagramas subsecuentes de un mensaje de ROS usan este opcode.
- PING (2) – Un paquete de latidos es enviado periódicamente para permitir al otro lado saber que la conexión sigue con vida.
- ERR (3) – Un paquete de error es usado para indicar que una conexión fue cerrada inesperadamente.

ID del Mensaje. Es un valor de 8 bits que se incrementa

por cada nuevo mensaje y es usado para determinar si los datagramas han sido descartados.

Block #. Este es un valor de 16 bits cuyo valor es 0 cuando el valor del opcode es DATA0, luego este campo contiene el total datagramas UDPROS esperados para completar el mensaje de ROS. Cuando el opcode es DATAN, el campo contiene el número actual del datagrama.

4.2. DEFINICIÓN DE METODOLOGÍA

Se define metodología como un conjunto de métodos. A su vez, método se define como un grupo de reglas a seguir para solucionar un problema.[17] Nuestra metodología que proponemos para este proyecto, se divide en:

1) ENCAPSULAR SISTEMA ROBOTICO EN UN GRAFO DE NODOS

El diseño funcional del robot debe reflejar de manera general el funcionamiento del sistema robótico, debido a que es el punto del cual se empieza a identificar y desarrollar las partes del robot según su aporte para el funcionamiento del sistema final. Acorde a ROS una vez identificados el hardware a emplear, se hace un grafo de

cómputo para visualizar la topología de nuestro sistema, que es lo que representa cada nodo, si un nodo es de hardware o es de procesamiento de datos y que nodos se va a comunicar entre sí, quien va a mandar los mensajes, quienes los van a recibir y después observar los resultados. En el grafo de computo siempre va a estar el nodo de ROS que es el /rosout.

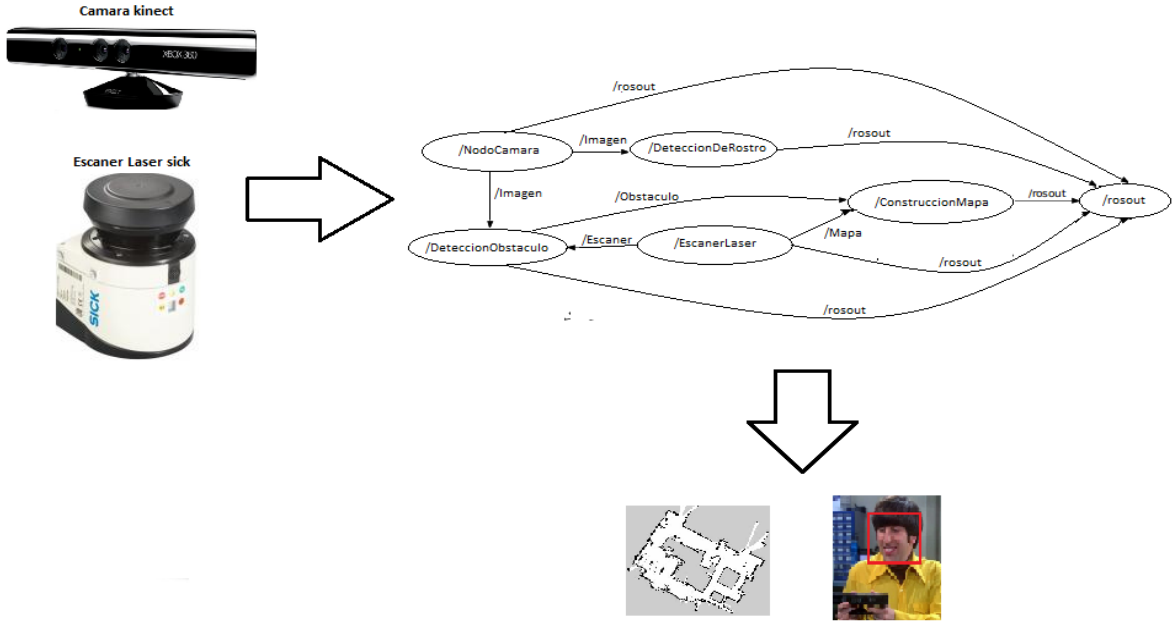


Figura 4. 6: Hardware + Grafo de nodos + Resultados

En la figura 4.6 se describe un sistema robótico, el que tiene dos nodos de hardware, uno para el sensor de la cámara Kinect y el otro para el sensor del láser Sick y además tiene tres nodos de procesamiento de datos, el nodo /DeteccionDeRostro recibe

información de la cámara y procesa la información para mostrar un cuadro rojo alrededor del rostro de una persona mientras la está grabando en vivo, el nodo /DeteccionObstaculo recibe los mensajes del nodo del láser el cual procesa la información para detectar obstáculos y mostrarlos en el mapa, el nodo /ConstruccionMapa recibe información del láser y del nodo de detección de obstáculos, el cual procesa la información y da como resultado un mapa.

En el caso de los nodos de hardware se pueden usar nodos existentes o hay que desarrollar nuevos nodos si no hubiera existentes.

2) REUTILIZACION DE NODOS EXISTENTES

Los repositorios de ROS tienen una colección de nodos y es probable que el hardware que se tiene ya tenga desarrollado un nodo. Si ese fuera el caso, ya no habría que modificar un nodo anterior ni desarrollar uno nuevo, solo utilizar el existente.

Para el hardware que no tenga nodo ya existente en ros, se tendrá que buscar un nodo que corresponda a un hardware similar por ejemplo de otro fabricante para que nos sirva de plantilla. Una vez

que se tenga este nodo, los siguientes criterios pueden seguir para adaptarlo al modelo del hardware que se tiene.

- Cambiar el nombre del nodo y los nombres de parámetros del nodo de plantilla, para que al momento de ejecutar los dos nodos de los diferentes modelos no haya conflicto de nombres en el grafo de ros.

- Encontrar la hoja de datos (datasheet) del hardware para acoplar las nuevas características al nodo anterior, siguiendo el ejemplo se puede cambiar la abertura del ángulo del láser sick.

- Editar el tipo de mensaje que se tiene con el nodo de plantilla para poder adaptarlo a los nuevos tipos de datos del dispositivo que se tiene. Por lo general la estructura de los mensajes no se cambian porque siempre tienden a seguir con los mismos tipos de datos a transmitir.

3) CREAR NUEVOS NODOS DE SER NECESARIO

Eventualmente se llegara a la etapa donde no se puede reutilizar código y se tendrá que crear nuevos nodos para nuevo hardware, se considera los siguientes criterios:

- Obtener la hoja de datos (datasheet) del dispositivo.

- Es necesario tener el código del controlador del hardware para crear nuestro nodo, se puede encontrar en internet pero si no es el caso, entonces hay que crearlo por los medios necesarios por ejemplo realizar ingeniería inversa a las señales eléctricas.
- Escoger el lenguaje para desarrollar el nuevo nodo, por ejemplo:
  - ❖ roscpp (C++): Librería de alto rendimiento, actualmente el mejor en soporte de diagnóstico.
  - ❖ rospy (Python): Una librería de bajo rendimiento, lenguaje fácil de entender.
- Revisar documentación guía para creación de nodos de hardware y de procesamientos de datos.

REP (ROS enhancement proposals) significan ROS propuestas de mejora. Un REP es un documento de diseño de suministro de información a la comunidad ROS, o la descripción de una nueva característica para ROS o sus procesos. El REP debe proporcionar una especificación técnica concisa de la función y la justificación de dicha función. Por ejemplo:

- ❖ 104 – Camera Info updates.

- ❖ 105 - Coordinate Frames for Mobile Platforms.
- ❖ 107 - Diagnostic System for Robots Running ROS.
- ❖ 117 - Information Distance Measurements.
- ❖ 118 - Depth Images.
- ❖ 120 - Coordinate Frames for Humanoids Robots.

- Se desarrolla el formato de mensajes que va a publicar o recibir nuestro nodo estos es en combinación de los tipos de datos que el hardware use con documentación previa de ros, esto sirve como guía para saber si los datos son admitidos por el servicio de mensajería, por ejemplo Rep 117 información sobre la unidades de medida y sus tipos de datos que podemos enviar en los mensajes.

- La estructura de un nodo en ROS es importante, primero antes que nada tienes que incluir a las librerías que vas a usar por ejemplo la de tus mensajes, después se tiene que llamar a la función de inicio de nodo, aquí por ejemplo nombras a tu nodo y le dices al grafo de computo que hay un nuevo nodo registrado, este debe estar identificado con un nombre único, si otro nodo se inicia con el mismo nombre, el que primero que se ejecutó se desactivara.

Después es necesario llamar a la librería cliente de ROS por ejemplo con roscpp es así: ros::NodeHandle nh;

Con la variable nh puedes publicar, suscribirte a tópicos, crear tópicos y usar los mensajes para comunicarte con el resto de nodos. Cuando terminas con la tarea de tu nodo tienes que apagarlo o probar que el nodo este bien.

4.3. CONCLUSIONES

Se propuso una metodología desarrollada en el framework seleccionado en el capítulo 3, primero se explicó la arquitectura del framework donde se explica las características que lo hacen un framework adecuado para desarrollar software robótico sin la necesidad de comprar una licencia, se explicó la comunicación entre componentes que lo hace un framework apropiado para una gran variedad de programadores ya que trabaja con varias librerías clientes como C++, Pyhton y entre otros, después haber revisado lo que es la arquitectura de ROS, se explicó la metodología que se usó en nuestro experimento del capítulo 5, donde dice ciertos criterios que tomar en cuenta antes de realizar un sistema robótico hibrido.

CAPÍTULO 5

5. IMPLEMENTACIÓN DE UN SISTEMA ROBÓTICO HÍBRIDO

5.1. INTRODUCCIÓN

En este capítulo se aplica la metodología propuesta en el capítulo anterior en un problema práctico, que consiste en implementar un sistema robótico que realice un mapa en 2D de una habitación mientras se moviliza dentro de ella, este sistema está compuesto principalmente por 2 componentes de distintos fabricantes:

1. Plataforma móvil Pioneer-3AT (Adept Technology, Inc.).
2. Escáner laser LMS-151 (Sick AG).

En este capítulo se explica cada uno de los nodos del grafo de cómputo del sistema que representa el sistema robótico, sus mensajes, parámetros y algoritmos. Además se describe como se realizó la adaptación de los nodos que se reutilizaron en esta implementación y la creación de un nuevo nodo. Finalmente se muestran los resultados obtenidos de la reconstrucción 2D generada por nuestro sistema robótico aplicando la metodología propuesta.

5.2. DESCRIPCIÓN

Para generar la reconstrucción 2D se necesita emplear el método de localización y mapeo simultáneos (SLAM), donde robot con un sistema de visión construye de forma incremental un mapa de su entorno, y mientras utiliza este mapa estima la trayectoria realizada por la plataforma móvil al recorrer el ambiente desconocido sin ninguna información previa.

Para el presente proyecto se usó una plataforma móvil (Pioneer-3AT) provisto de un sistema de visión laser (LMS-151) que permite obtener una nube de puntos (el conjunto de las lecturas del láser) del entorno de navegación del robot.

Se empezó desde un punto indeterminado con la plataforma móvil que realizo una trayectoria determinada por el usuario. La meta de SLAM es usar información del entorno para actualizar la posición de la plataforma móvil. Los datos de odometría de la plataforma móvil tiende a ser errónea, entonces no se puede confiar directamente en la Odometría (posición del robot en un plano x,y,θ) para actualizar la posición.

Se puede usar la nube de puntos del ambiente que proporciona el láser para corregir la posición que proporciona la odometría de la plataforma. Esto se realiza con la extracción de características (puntos de referencia) del entorno y volviéndolas a observar cuando la plataforma se mueva a otra posición.

La clave para el proceso del SLAM es el filtro de Kalman Extendido (EKF), es el responsable de actualizar la posición del robot basándose en los puntos de referencia. Entonces cuando cambia la odometría al moverse la plataforma móvil, en el EKF actualiza esa información y borra la odometría anterior, y los puntos de referencia son extraídos del ambiente de la nueva odometría de la plataforma móvil, el proceso de SLAM asocia estos puntos de referencia actuales a los extraídos de los lugares que previamente ha visto el sistema robótico. Observando

de nuevo los puntos de referencia actuales con los anteriores entonces se actualiza la posición del sistema robótico, con el EKF lo que se obtiene un estimado de la posición del sistema robótico en base a la odometría y los puntos de referencia marcados por el sistema laser.

En la secuencia de imágenes a continuación se explica el proceso del SLAM más a detalle:

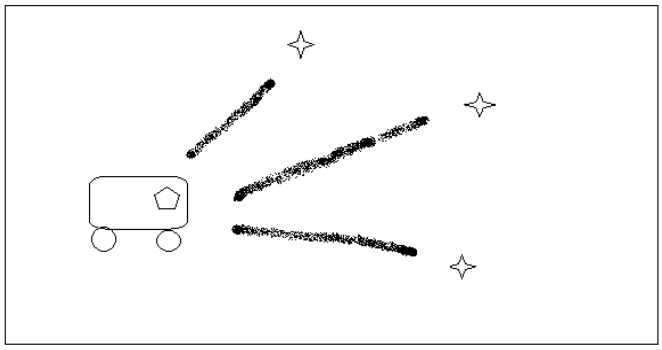


Figura 5. 1: Ilustración SLAM

En la figura 5.1 se muestra la plataforma móvil representada por un rectángulo con esquinas redondeadas, el sistema laser por un pentágono, el sistema robótico extrae los puntos de referencia representados por estrellas y los guarda.

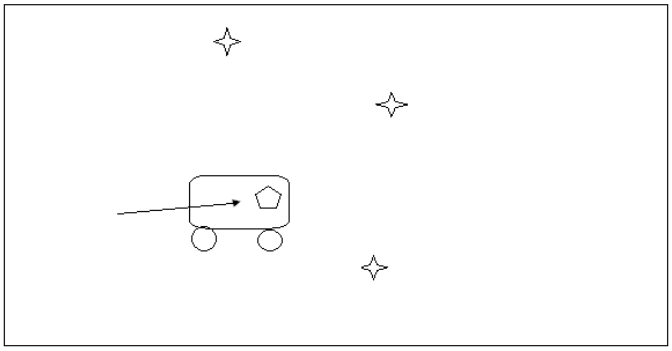


Figura 5. 2: Ilustración SLAM 2

En la figura 5.2 la plataforma móvil se mueve cierta distancia de su posición inicial, la distancia que se movió la plataforma la proporciona la odometría de la plataforma móvil para ser procesada por el método SLAM.

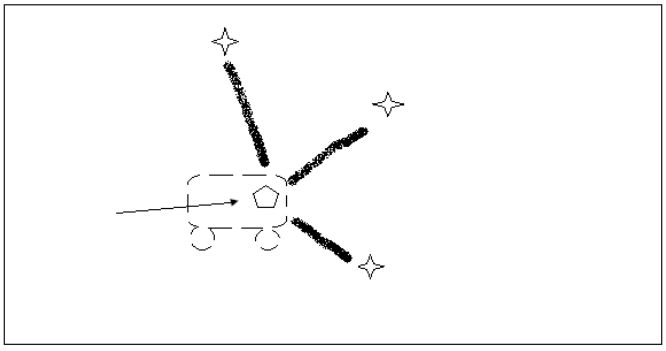


Figura 5. 3: Ilustración SLAM 3

En la figura 5.3 el sistema robótico mide una vez más la ubicación de sus puntos de referencia usando su sistema laser pero el sistema se da cuenta que los puntos de referencia no coinciden con los

anteriores, entonces el sistema nota que no se encuentra en la supuesta posición que tiene actualmente.

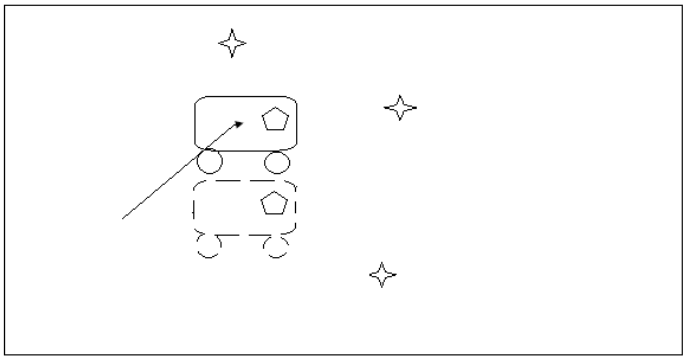


Figura 5. 4: Ilustración SLAM 4

En la figura 5.4 Después de comparar los puntos de referencia actuales con los anteriores, se muestran dos gráficos del sistema robótico, el de línea segmentada es donde el sistema cree que esta por la odometría de la plataforma móvil, y el otro es donde cree que esta por los puntos de referencia marcados por el sistema laser.

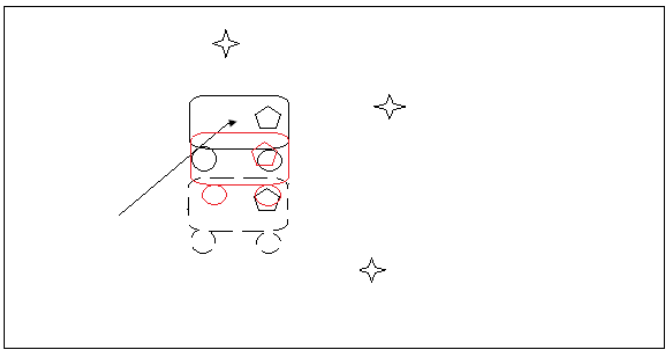


Figura 5.5: Ilustración SLAM 5

Ahora que se recopilo la información de odometría y los puntos de referencia en dos posiciones diferentes, donde el filtro EKF procesa esa información dando un estimado a su posición como se ve en la Figura 5.5 donde el grafico del sistema robótico con líneas segmentadas representa la ubicación del sistema en base a la odometría, el grafico con líneas rojas en base a la información de los puntos de referencia y las líneas negras será el resultado estimado después de pasar por el filtro EKF. [18]

Para la reconstrucción 2D se combina la información de la odometría de robot, con el ángulo de incidencia de un punto de referencia extraído del sistema laser, con el proceso de SLAM una vez actualizada la posición, se actualiza el mapa marcando las casillas de ocupación del mapa como en la figura 5.6.

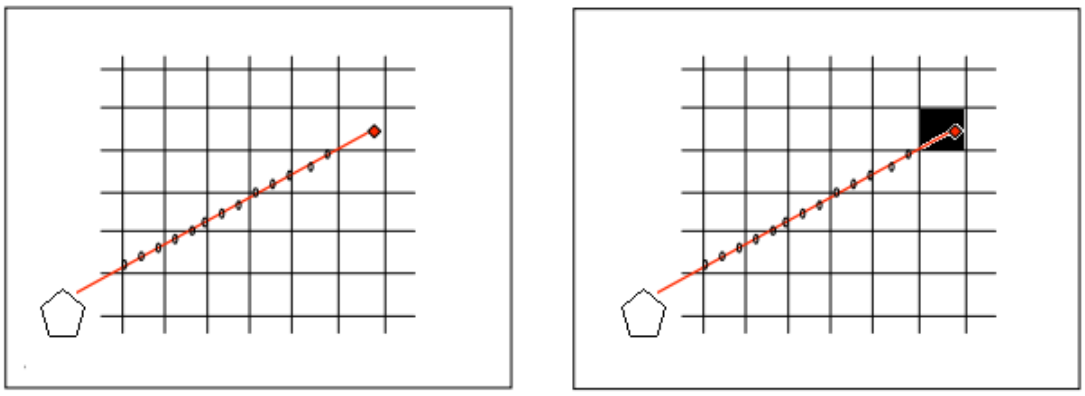


Figura 5. 6: Ilustración Mapeo

Una vez que se explicó superficialmente el proceso de reconstrucción 2D, procedemos a describir los pasos de la metodología propuesta para crear el experimento de la construcción un mapa.

I. DISEÑO DEL GRAFO DE COMPUTO

Dado que se usa el método del SLAM para resolver el problema de reconstrucción 2D el diseño del grafo de cómputo del robot, requiere al menos cinco nodos:

a) Un nodo para extraer y publicar la información de odometría de la plataforma móvil.

b) Un nodo para leer los datos de distancia del láser y definir y publicar la ubicación de los puntos de referencia.

c) Un nodo que proceses los mensajes del láser y la reconstruya en 2D.

d) Un nodo para visualizar el mapa, casillas de ocupación (depende de la resolución que se quiera) para mostrar la reconstrucción.

e) Un nodo de control de motores para mover la plataforma móvil dentro de la habitación.

Una vez definidos los criterios para el diseño de sistema hibrido, se obtiene el grafo de la figura 5.7.

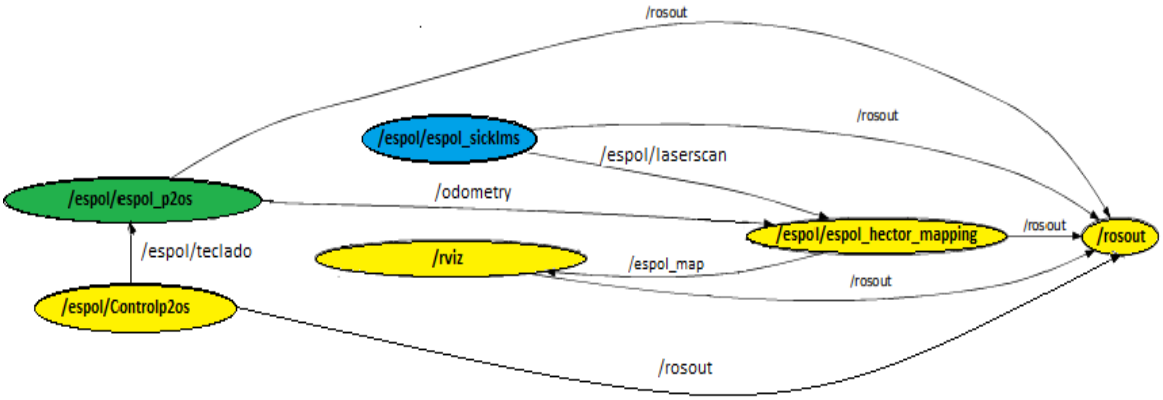


Figura 5.7: Grafo de computo del sistema robótico

El grafo consta de seis nodos, de los cuales un nodo es el del maestro, y los otros cinco son: dos nodos controladores de hardware y los otros tres son de procesamiento de datos, los nodos que están en amarillo se encuentran en los repositorios, el verde es el nodo que vamos a modificar y el nodo azul es el que vamos a crear, a continuación se describe de la función de cada uno de ellos:

- /rosout: Es el mecanismo de registro de información de ROS, todos los nodos publican y se suscriben a él, para registrarse y para saber quiénes están registrados.

- /espol/espol_p2os: Este es un nodo controlador de hardware que puede controlar a algunos modelos Pioneer.

- /espol/espol_sicklms: Nodo controlador del láser, la misión de este nodo es publicar información a tópico /espol/laserscan, el tipo de información es la distancia en metros de todas las lecturas de la nube de puntos generada por el láser. En el nodo original la librería de acceso al hardware y mensajes se encuentran en paquetes diferentes, para darle portabilidad al robot nosotros generamos nuevamente el nodo /SickLMS incluyendo todas las dependencias y lo renombramos /espol/sicklms.

- /espol_hectormapping: Nodo de procesamiento de datos, este nodo está suscrito a dos nodos de hardware para la construcción 2D, el /espol/ publica la distancia y la abertura de cada lectura del láser y el /espol/espol_p2os publica la odometría de la plataforma móvil. Este nodo usa el algoritmo de

SLAM y el filtro de Kalman extendido donde el resultado del proceso es la ubicación de los puntos en el mapa (x, y, θ), estas coordenadas se publican al nodo /espol/rviz.

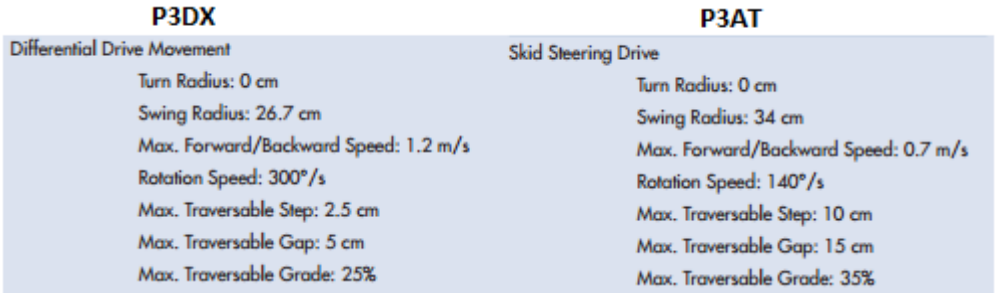
- /espol/rviz: Nodo de procesamiento de datos, se trata de un visualizador, este nodo se suscribe al nodo de /espol/hector_mapping donde recibe las coordenadas (x, y, θ), y según eso se marca las casillas de ocupación del mapa, reconstruyendo así un mapa en 2D mostrado por este visualizador.

- /espol/controlp2os: Nodo de procesamiento de datos, este nodo publica dirección y velocidad al nodo /espol/espol_p2os que es el nodo controlador de la plataforma móvil.

II. REUTILIZACIÓN DE NODOS EXISTENTES

Los nodos existentes que se usaron fueron los de procesamiento de datos: /espol/rviz y /espol/hectormapping. El nodo que correspondían a un hardware similar al nuestro fue: /espol/espol_p2os (ver anexo A), el cual fue creado para el robot Pioneer P3DX, una versión anterior a nuestro robot móvil Pioneer P3AT, para adaptar el nodo que se usará como plantilla se

requiere cambiar los parámetros que se encuentran dentro del nodo según se indica en la hoja de especificaciones, ver Fig. 5.8, para que nuestro robot pueda funcionar con el nodo plantilla.



P3DX	P3AT
Differential Drive Movement	Skid Steering Drive
Turn Radius: 0 cm	Turn Radius: 0 cm
Swing Radius: 26.7 cm	Swing Radius: 34 cm
Max. Forward/Backward Speed: 1.2 m/s	Max. Forward/Backward Speed: 0.7 m/s
Rotation Speed: 300°/s	Rotation Speed: 140°/s
Max. Traversable Step: 2.5 cm	Max. Traversable Step: 10 cm
Max. Traversable Gap: 5 cm	Max. Traversable Gap: 15 cm
Max. Traversable Grade: 25%	Max. Traversable Grade: 35%

Figura 5.8: Parámetros de P3DX y P3AT

III. CREAR NUEVOS NODOS DE SER NECESARIO

En esta parte crearemos un nodo para el láser sick LMS-151 /espol/espol_sicklms (ver anexo B) aunque. Este nodo ya existe en los repositorios de ROS. Sin embargo generamos de nuevo el nodo comprimiéndolo dentro de un paquete para darle portabilidad.

Se busca la datasheet del láser para adquirir conocimiento de sus parámetros físicos por ejemplo que interfaces de comunicación tiene, después se busca el código de su controlador, este nos indicara los datos que podemos obtener del láser. Ahora para

nuestro ejemplo se ha escogido la librería cliente de ROS para trabajar a roscpp (C++) por ser de alto rendimiento.

Antes de empezar a codificar nuestro nodo tenemos que definir el mensaje que se va a crear, nos podemos guiar por dos cosas, la primera los tipos de datos que se va a obtener del código del controlador, y la segunda en un documento guía para unidades de medida que ROS proporciona que es el REP-117 para saber qué tipo de datos poder declarar en los mensajes.

Una vez definido el mensaje, empezamos codificamos el nodo del láser:

1. Declarar las librerías por ejemplo: La librería del controlador.
2. Declarar las variables.
3. Iniciar el nodo.
4. Declarar una variable tipo ros::NodeHandle para crear el tópico /espol/laserscan, al cual se le enviaran los mensajes del láser.
5. Terminar el nodo.

5.3. RECURSOS FÍSICOS DEL EXPERIMENTO

En la figura 5.9 se muestran los elementos de nuestro proyecto. Para la realización de este experimento se usaron los siguientes elementos:

- 1 Laser Sick LMS-151.
- 1 Robot móvil Pioneer P3AT.
- 1 Cable M12 – Ethernet.
- 1 Cable Serial – USB.
- 1 cable M12 – DC.

Estos elementos fueron provistos por el Centro de visión y Robótica (CVR) de la ESPOL.

Figura 5.9: Elementos Del experimento

5.4. RESULTADOS

A continuación se muestra las imágenes que resultaron de las pruebas con el sistema robótico hibrido construido.

Para realizar las pruebas los nodos fueron ejecutados de manera incrementada primero el nodo /espol/laserscan, después el nodo /espol/espol_p2os más /espol/hectormapping y por ultimo al resto. Se comprobó el correcto funcionamiento del láser. Esto se realizó mostrando los datos en el visualizador RVIZ como se muestra en la Figura 5.10.

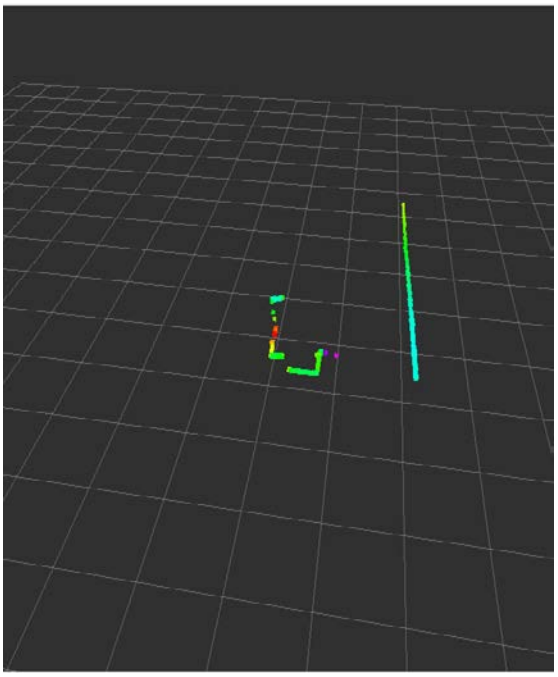


Figura 5. 10: Visualizar Datos del Láser

Después se ejecutó los nodos /espol/espol_p2os y /espol/hector_mapping como se ve en la Figura 5.11 que se muestra el mapa inicial generado por el robot desde una posición fija.

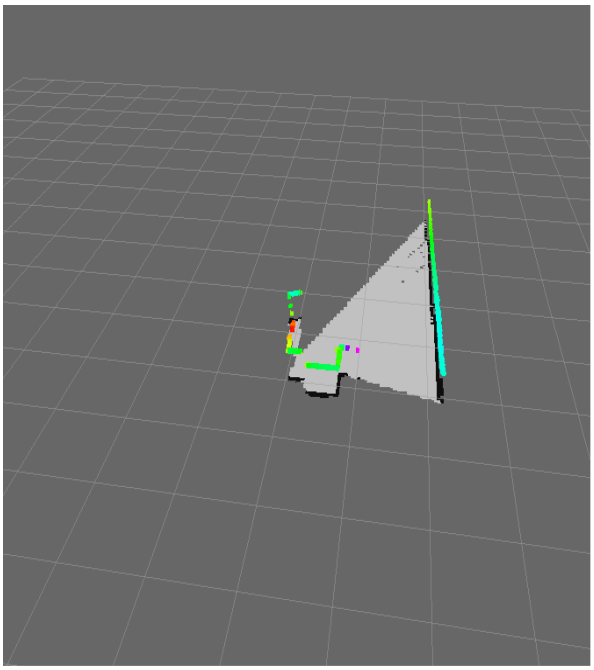


Figura 5. 11: Visualizar Datos del Mapa Parcial

Una vez que el robot ha realizado un recorrido operado manualmente por medio del nodo /espol/Controlp2os por la habitación recolectando datos y procesándolos, podrá mostrar un mapa completo de una habitación en 2D como se muestra en la Figura 5.10.

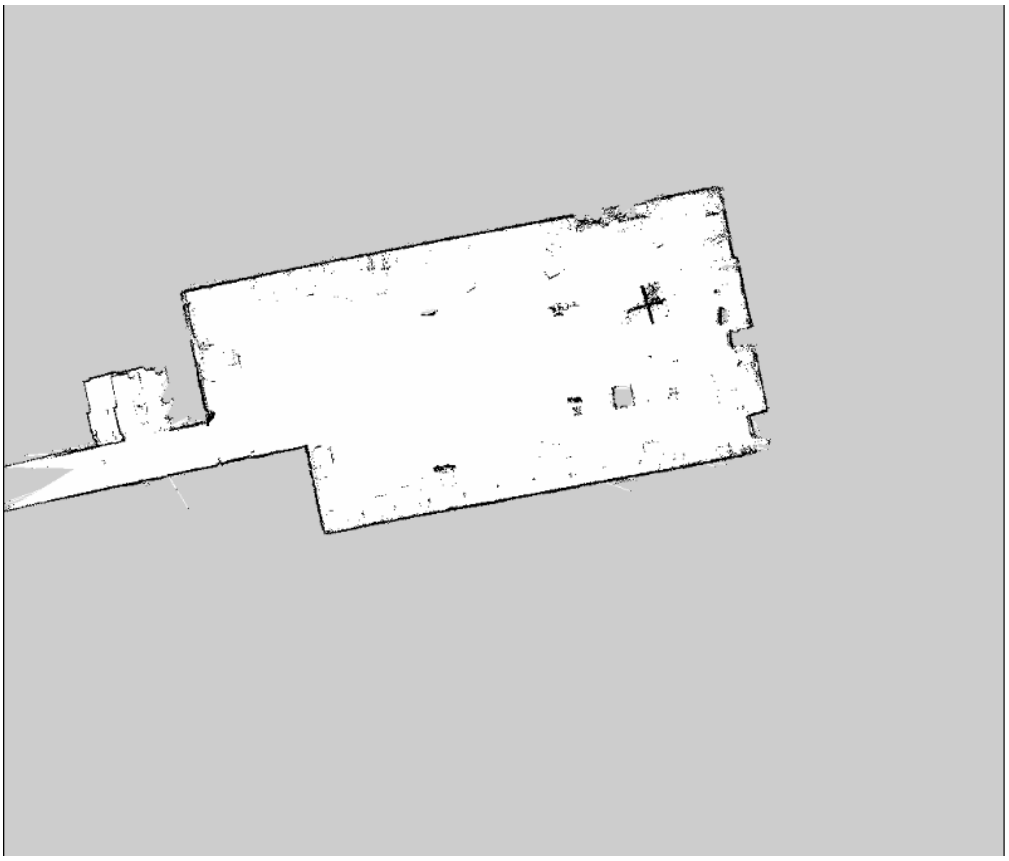


Figura 5.12: Mapa Completo

CONCLUSIONES Y RECOMENDACIONES

Al finalizar el presente trabajo se han obtenido las siguientes conclusiones:

1. Hoy en día existen diversas herramientas (frameworks y librerías) de software que hacen posible implementar sistemas robóticos híbridos. Tomando en cuenta la falta de presupuesto y acceso al código de los sistemas propietarios, consideramos que son las herramientas libres las más idóneas para la implementación de estos sistemas.

2. En nuestro caso en particular, ha sido el uso de ROS lo que nos ha permitido implementar estos sistemas y elaborar nuestra metodología. Framework que elegimos por la variedad de lenguajes de programación que soporta, las diferentes técnicas de comunicación que implementa, su

adaptabilidad y adaptabilidad con diferentes herramientas y el hecho de estar liberado bajo la licencia BSD.

3. Se implementó un sistema robótico hibrido con la metodología propuesta, generando una reconstrucción en 2D empleando el método SLAM.

4. ROS brinda una gran cantidad de recursos de licencia libre, lo que permite reutilizarlos y tener resultados más eficientes sin tener que utilizar complejos códigos.

5. Utilizar el sistema de visión que se implementó en el sistema robótico hibrido en ambientes hostiles y peligrosos.

6. Durante la etapa de adquisición de imágenes 2D, se extraen una gran cantidad de puntos que se encuentran alejados de su ubicación real, se sugiere mejorar con procesos de procesamiento de imágenes.

BIBLIOGRAFÍA

[1] Matariç, M., “The Robotics Primer”, http://hci.ucsd.edu/hutchins/cogs8/mataric-primer.pdf, fecha de consulta enero 2013

[2] Koreis, V., “Čapek's R.U.R.”, http://www.booksplendour.com.au/capek/rur.htm, fecha de consulta enero 2013

[3] Sommerville, I., Alfonso, M., “Ingeniería del software”, Pearson Educación, 2005

[4] Monarca, M., “Desarrollo de Software Basado en Componentes”, http://www.iupuebla.com/Ingenieria/Sis_compu/MA_sistemas/MA_PROGRAMACION_WINDOWS-sistema-basado-en-componentes-LAURA_MONARCA.pdf, fecha de consulta junio 2014

[5] Poza, J., “Revisión de los Sistemas de Comunicaciones más empleados en Control Distribuido”, https://riunet.upv.es/bitstream/handle/10251/6408/Comunicaciones en los sistemas distribuidos.pdf, fecha de consulta junio 2014

[6] Coulouris, G., Dollimore, J., Kindberg, Kindberg, T., Blair, G., “Distributed Systems Concepts and Design”, Addison-Wesley 5th Ed, 2012

[7] Orocos.org, “The Orocos Real-Time Toolkit”, http://www.orocos.org/rtt, fecha de consulta diciembre 2013

[8] Fitzpatrick , P., " Toward Long-Lived Robot Software", http://wiki.icub.org/yarp/media/humanoids06long.pdf, fecha de consulta diciembre 2013

[9] Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A., "ROS: an open-source robot operating system", http://www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf, fecha de consulta enero 2014

[10] Montague, B., "Why you should use a BSD style license for your Open Source Project", https://www.freebsd.org/doc/en/articles/bsdl-gpl/article.html, fecha de consulta febrero 2014

[11] Robotnik.es, "Interfaces de usuario en ROS", http://www.robotnik.es/interfaces-de-usuario-en-ros/, fecha de consulta junio 2014

[12] McKinnon, A. "Interface Definition Language", http://csis.pace.edu/~marchese/CS865/Papers/interface-definition-language.pdf, fecha de consulta junio 2014

[13] Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A., "ROS: an open-source Robot Operating System", http://ai.stanford.edu/~mquigley/papers/icra2009-ros.pdf, fecha de consulta junio 2014

[14] Opencv.org, "Open Source Computer Vision", http://opencv.org, fecha de consulta agosto 2014

[15] Gazebosim.org, “Features”, http://www.gazebosim.org/, fecha de consulta agosto 2014

[16] Open Source Robotics Foundation, “ROS”, http//www.ros.org/, fecha de consulta agosto 2014

[17] Muñoz, L., José, L. “Control en robótica Móvil, Arquitectura y Metodología”. Tesis Doctoral. Universidad de Murcia. Departamento de Automática, Electricidad y Electrónica Industrial.1998, fecha de consulta septiembre 2014.

[18] Riisgaard, S., Rufus, M., “SLAM”, http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-412j-cognitive-robotics-spring-2005/projects/1aslam_blas_repo.pdf,fecha de consulta octubre 2014.

ANEXOS

ANEXO A

CODIGO FUENTE DEL NODO /espol/espol_p2os:

```
#include <p2os_driver/p2os.h>
#include <termios.h>
#include <fcntl.h>
#include <string.h>


P2OSNode::P2OSNode( ros::NodeHandle nh ) :
    n(nh), gripper_dirty_(false),
    batt_pub_( n.advertise<p2os_msgs::BatteryState>("battery_state",1000),
            diagnostic_,
            diagnostic_updater::FrequencyStatusParam( &desired_freq, &desired_freq, 0.1),
            diagnostic_updater::TimeStampStatusParam() ),
    ptz_(this)
{
  ros::NodeHandle n_private("~");
  n_private.param(std::string("odom_frame_id"), odom_frame_id,std::string("odom"));
  n_private.param(std::string("base_link_frame_id"),
base_link_frame_id,std::string("base_link"));
  n_private.param("use_sonar", use_sonar_, false);


  n_private.param( "bumpstall", bumpstall, -1 );

  n_private.param( "pulse", pulse, 1 );

  n_private.param( "rot_kp", rot_kp, 140 );

  n_private.param( "rot_kv", rot_kv, 140 );

  n_private.param( "rot_ki", rot_ki, 140 );

  n_private.param( "trans_kp", trans_kp, 140 );

  n_private.param( "trans_kv", trans_kv, 140 );

  n_private.param( "trans_ki", trans_ki, 140 );
```

```
  std::string def = DEFAULT_P2OS_PORT;
  n_private.param( "port", psos_serial_port, def );
  ROS_INFO( "using serial port: [%s]", psos_serial_port.c_str() );
  n_private.param( "use_tcp", psos_use_tcp, false );
  std::string host = DEFAULT_P2OS_TCP_REMOTE_HOST;
  n_private.param( "tcp_remote_host", psos_tcp_host, host );
  n_private.param( "tcp_remote_port", psos_tcp_port,
DEFAULT_P2OS_TCP_REMOTE_PORT );

  n_private.param( "radio", radio_modemp, 0 );

  n_private.param( "joystick", joystick, 0 );

  n_private.param( "direct_wheel_vel_control", direct_wheel_vel_control, 0 );

  double spd;
  n_private.param( "max_xspeed", spd, MOTOR_DEF_MAX_SPEED);
  motor_max_speed = (int)rint(1e3*spd);

  n_private.param( "max_yawspeed", spd, MOTOR_DEF_MAX_TURNSPEED);
  motor_max_turnspeed = (short)rint(RTOD(spd));

  n_private.param( "max_xaccel", spd, 0.0);
  motor_max_trans_accel = (short)rint(1e3*spd);

  n_private.param( "max_xdecel", spd, 0.0);
  motor_max_trans_decel = (short)rint(1e3*spd);

  n_private.param( "max_yawaccel", spd, 0.0);
  motor_max_rot_accel = (short)rint(RTOD(spd));
  n_private.param( "max_yawdecel", spd, 0.0);
  motor_max_rot_decel = (short)rint(RTOD(spd));


  desired_freq = 10;


  pose_pub_ = n.advertise<nav_msgs::Odometry>("pose", 1000);

  mstate_pub_ = n.advertise<p2os_msgs::MotorState>("motor_state",1000);
  grip_state_pub_ = n.advertise<p2os_msgs::GripperState>("gripper_state",1000);
  ptz_state_pub_ = n.advertise<p2os_msgs::PTZState>("ptz_state",1000);
  sonar_pub_ = n.advertise<p2os_msgs::SonarArray>("sonar", 1000);
  aio_pub_ = n.advertise<p2os_msgs::AIO>("aio", 1000);
  dio_pub_ = n.advertise<p2os_msgs::DIO>("dio", 1000);
```

```
  cmdvel_sub_ = n.subscribe("cmd_vel", 1, &P2OSNode::cmdvel_cb, this);
  cmdmstate_sub_ = n.subscribe("cmd_motor_state", 1, &P2OSNode::cmdmotor_state,
                               this);
  gripper_sub_ = n.subscribe("gripper_control", 1, &P2OSNode::gripperCallback,
                             this);
  ptz_cmd_sub_ = n.subscribe("ptz_control", 1, &P2OSPtz::callback, &ptz_);


  veltime = ros::Time::now();



          diagnostic_.add("Motor Stall", this, &P2OSNode::check_stall );
          diagnostic_.add("Battery Voltage", this, &P2OSNode::check_voltage );



  initialize_robot_params();
}


P2OSNode::~P2OSNode()
{
}


void P2OSNode::cmdmotor_state( const p2os_msgs::MotorStateConstPtr &msg)
{
  motor_dirty = true;
  cmdmotor_state_ = *msg;
}


void P2OSNode::check_and_set_motor_state()
{
  if( !motor_dirty ) return;
  motor_dirty = false;


  unsigned char val = (unsigned char) cmdmotor_state_.state;
  unsigned char command[4];
  P2OSPacket packet;
  command[0] = ENABLE;
```

```
  command[1] = ARGINT;
  command[2] = val;
  command[3] = 0;
  packet.Build(command,4);

  p2os_data.motors.state = cmdmotor_state_.state;
  SendReceive(&packet,false);
}


void P2OSNode::check_and_set_gripper_state()
{
  if( !gripper_dirty_ ) return;
  gripper_dirty_ = false;
  unsigned char grip_val = (unsigned char) gripper_state_.grip.state;
  unsigned char grip_command[4];
  P2OSPacket grip_packet;
  grip_command[0] = GRIPPER;
  grip_command[1] = ARGINT;
  grip_command[2] = grip_val;
  grip_command[3] = 0;
  grip_packet.Build(grip_command,4);
  SendReceive(&grip_packet,false);
  unsigned char lift_val = (unsigned char) gripper_state_.lift.state;
  unsigned char lift_command[4];
  P2OSPacket lift_packet;
  lift_command[0] = GRIPPER;
  lift_command[1] = ARGINT;
  lift_command[2] = lift_val;
  lift_command[3] = 0;
  lift_packet.Build(lift_command,4);
  SendReceive(&lift_packet,false);
}


void P2OSNode::cmdvel_cb( const geometry_msgs::TwistConstPtr &msg)
{


  if( fabs( msg->linear.x - cmdvel_.linear.x ) > 0.01 || fabs( msg->angular.z-cmdvel_.angular.z)
> 0.01 )
  {
    veltime = ros::Time::now();
    ROS_DEBUG( "new speed: [%0.2f,%0.2f](%0.3f)", msg->linear.x*1e3, msg->angular.z,
veltime.toSec() );
```

```
    vel_dirty = true;
    cmdvel_ = *msg;
  }
  else
  {
    ros::Duration veldur = ros::Time::now() - veltime;
    if( veldur.toSec() > 2.0 && ((fabs(cmdvel_.linear.x) > 0.01) || (fabs(cmdvel_.angular.z) >
0.01)) )
    {
      ROS_DEBUG( "maintaining old speed: %0.3f|%0.3f", veltime.toSec(),
ros::Time::now().toSec() );
      vel_dirty = true;
      veltime = ros::Time::now();
    }
  }


}


void P2OSNode::check_and_set_vel()
{
  if( !vel_dirty ) return;


  ROS_DEBUG( "setting vel: [%0.2f,%0.2f]",cmdvel_.linear.x,cmdvel_.angular.z);
  vel_dirty = false;


  unsigned short absSpeedDemand, absturnRateDemand;
  unsigned char motorcommand[4];
  P2OSPacket motorpacket;


  int vx = (int) (cmdvel_.linear.x*1e3);
  int va = (int)rint(RTOD(cmdvel_.angular.z));


  {
    motorcommand[0] = VEL;
    if( vx >= 0 ) motorcommand[1] = ARGINT;
    else motorcommand[1] = ARGNINT;


    absSpeedDemand = (unsigned short)abs(vx);
```

```
    if( absSpeedDemand <= this->motor_max_speed )
    {
      motorcommand[2] = absSpeedDemand & 0x00FF;
      motorcommand[3] = (absSpeedDemand & 0xFF00) >> 8;
    }
    else
    {
      ROS_WARN( "speed demand thresholded! (true: %u, max: %u)", absSpeedDemand,
motor_max_speed );
      motorcommand[2] = motor_max_speed & 0x00FF;
      motorcommand[3] = (motor_max_speed & 0xFF00) >> 8;
    }
    motorpacket.Build(motorcommand, 4);
    SendReceive(&motorpacket);


    motorcommand[0] = RVEL;
    if( va >= 0 ) motorcommand[1] = ARGINT;
    else motorcommand[1] = ARGNINT;


    absturnRateDemand = (unsigned short)abs(va);
    if( absturnRateDemand <= motor_max_turnspeed )
    {
      motorcommand[2] = absturnRateDemand & 0x00FF;
      motorcommand[3] = (absturnRateDemand & 0xFF00) >> 8;
    }
    else
    {
      ROS_WARN("Turn rate demand threshholded!");
      motorcommand[2] = this->motor_max_turnspeed & 0x00FF;
      motorcommand[3] = (this->motor_max_turnspeed & 0xFF00) >> 8;
    }


    motorpacket.Build(motorcommand,4);
    SendReceive(&motorpacket);
  }
}


void P2OSNode::gripperCallback(const p2os_msgs::GripperStateConstPtr &msg)
{
  gripper_dirty_ = true;
  gripper_state_ = *msg;
```

```
}


int P2OSNode::Setup()
{
  int i;
  int bauds[] = {B9600, B38400, B19200, B115200, B57600};
  int numbauds = sizeof(bauds);
  int currbaud = 0;
  sippacket = NULL;
  lastPulseTime = 0.0;


  struct termios term;
  unsigned char command;
  P2OSPacket packet, receivedpacket;
  int flags=0;
  bool sent_close = false;
  enum
  {
    NO_SYNC,
    AFTER_FIRST_SYNC,
    AFTER_SECOND_SYNC,
    READY
  } psos_state;


  psos_state = NO_SYNC;


  char name[20], type[20], subtype[20];
  int cnt;


  ROS_INFO("P2OS connection opening serial port %s...",psos_serial_port.c_str());


  if((this->psos_fd = open(this->psos_serial_port.c_str(),
                O_RDWR | O_SYNC | O_NONBLOCK, S_IRUSR | S_IWUSR )) < 0 )
  {
    ROS_ERROR("P2OS::Setup():open():");
    return(1);
  }
```

```
if(tcgetattr( this->psos_fd, &term ) < 0 )
{
  ROS_ERROR("P2OS::Setup():tcgetattr():");
  close(this->psos_fd);
  this->psos_fd = -1;
  return(1);
}


cfmakeraw( &term );
cfsetispeed(&term, bauds[currbaud]);
cfsetospeed(&term, bauds[currbaud]);


if(tcsetattr(this->psos_fd, TCSAFLUSH, &term ) < 0)
{
  ROS_ERROR("P2OS::Setup():tcsetattr():");
  close(this->psos_fd);
  this->psos_fd = -1;
  return(1);
}


if(tcflush(this->psos_fd, TCIOFLUSH ) < 0)
{
  ROS_ERROR("P2OS::Setup():tcflush():");
  close(this->psos_fd);
  this->psos_fd = -1;
  return(1);
}


if((flags = fcntl(this->psos_fd, F_GETFL)) < 0)
{
  ROS_ERROR("P2OS::Setup():fcntl()");
  close(this->psos_fd);
  this->psos_fd = -1;
  return(1);
}


int num_sync_attempts = 3;
while(psos_state != READY)
{
  switch(psos_state)
```

```
{
  case NO_SYNC:
    command = SYNC0;
    packet.Build(&command, 1);
    packet.Send(this->psos_fd);
    usleep(P2OS_CYCLETIME_USEC);
    break;
  case AFTER_FIRST_SYNC:
    ROS_INFO("turning off NONBLOCK mode...");
    if(fcntl(this->psos_fd, F_SETFL, flags ^ O_NONBLOCK) < 0)
    {
      ROS_ERROR("P2OS::Setup():fcntl()");
      close(this->psos_fd);
      this->psos_fd = -1;
      return(1);
    }
    command = SYNC1;
    packet.Build(&command, 1);
    packet.Send(this->psos_fd);
    break;
  case AFTER_SECOND_SYNC:
    command = SYNC2;
    packet.Build(&command, 1);
    packet.Send(this->psos_fd);
    break;
  default:
    ROS_WARN("P2OS::Setup():shouldn't be here...");
    break;
}
usleep(P2OS_CYCLETIME_USEC);


if(receivedpacket.Receive(this->psos_fd))
{
  if((psos_state == NO_SYNC) && (num_sync_attempts >= 0))
  {
    num_sync_attempts--;
    usleep(P2OS_CYCLETIME_USEC);
    continue;
  }
  else
  {
    if(++currbaud < numbauds)
    {
      cfsetispeed(&term, bauds[currbaud]);
```

```
      cfsetospeed(&term, bauds[currbaud]);
      if( tcsetattr(this->psos_fd, TCSAFLUSH, &term ) < 0 )
      {
        ROS_ERROR("P2OS::Setup():tcsetattr():");
        close(this->psos_fd);
        this->psos_fd = -1;
        return(1);
      }


      if(tcflush(this->psos_fd, TCIOFLUSH ) < 0 )
      {
        ROS_ERROR("P2OS::Setup():tcflush():");
        close(this->psos_fd);
        this->psos_fd = -1;
        return(1);
      }
      num_sync_attempts = 3;
      continue;
    }
    else
    {
      // tried all speeds; bail
      break;
    }
  }
}
switch(receivedpacket.packet[3])
{
  case SYNC0:
    ROS_INFO( "SYNC0" );
    psos_state = AFTER_FIRST_SYNC;
    break;
  case SYNC1:
    ROS_INFO( "SYNC1" );
    psos_state = AFTER_SECOND_SYNC;
    break;
  case SYNC2:
    ROS_INFO( "SYNC2" );
    psos_state = READY;
    break;
  default:

    if(!sent_close)
    {
```

```
        ROS_DEBUG("sending CLOSE");
        command = CLOSE;
        packet.Build( &command, 1);
        packet.Send(this->psos_fd);
        sent_close = true;
        usleep(2*P2OS_CYCLETIME_USEC);
        tcflush(this->psos_fd,TCIFLUSH);
        psos_state = NO_SYNC;
       }
       break;
   }
   usleep(P2OS_CYCLETIME_USEC);
  }
  if(psos_state != READY)
  {
   if(this->psos_use_tcp)
   ROS_INFO("Couldn't synchronize with P2OS.\n"
         "  Most likely because the robot is not connected %s %s",
         this->psos_use_tcp ? "to the ethernet-serial bridge device " : "to the serial port",
         this->psos_use_tcp ? this->psos_tcp_host.c_str() : this->psos_serial_port.c_str());
   close(this->psos_fd);
   this->psos_fd = -1;
   return(1);
  }
  cnt = 4;
  cnt += snprintf(name, sizeof(name), "%s", &receivedpacket.packet[cnt]);
  cnt++;
  cnt += snprintf(type, sizeof(type), "%s", &receivedpacket.packet[cnt]);
  cnt++;
  cnt += snprintf(subtype, sizeof(subtype), "%s", &receivedpacket.packet[cnt]);
  cnt++;


  std::string hwID = std::string( name ) + std::string(": ") + std::string(type) + std::string("/") +
std::string( subtype );
  diagnostic_.setHardwareID(hwID);


  command = OPEN;
  packet.Build(&command, 1);
  packet.Send(this->psos_fd);
  usleep(P2OS_CYCLETIME_USEC);
  command = PULSE;
  packet.Build(&command, 1);
  packet.Send(this->psos_fd);
```

```
usleep(P2OS_CYCLETIME_USEC);


ROS_INFO("Done.\n   Connected to %s, a %s %s", name, type, subtype);


for(i=0;i<PLAYER_NUM_ROBOT_TYPES;i++)
{
  if(!strcasecmp(PlayerRobotParams[i].Class.c_str(),type) &&
     !strcasecmp(PlayerRobotParams[i].Subclass.c_str(),subtype))
  {
    param_idx = i;
    break;
  }
}
if(i == PLAYER_NUM_ROBOT_TYPES)
{
  ROS_WARN("P2OS: Warning: couldn't find parameters for this robot; "
         "using defaults");
  param_idx = 0;
}



if(!sippacket)
{
  sippacket = new SIP(param_idx);
  sippacket->odom_frame_id = odom_frame_id;
  sippacket->base_link_frame_id = base_link_frame_id;
}
this->ToggleSonarPower(0);

P2OSPacket accel_packet;
unsigned char accel_command[4];
if(this->motor_max_trans_accel > 0)
{
  accel_command[0] = SETA;
  accel_command[1] = ARGINT;
  accel_command[2] = this->motor_max_trans_accel & 0x00FF;
  accel_command[3] = (this->motor_max_trans_accel & 0xFF00) >> 8;
  accel_packet.Build(accel_command, 4);
  this->SendReceive(&accel_packet,false);
}
```

```
if(this->motor_max_trans_decel < 0)
{
  accel_command[0] = SETA;
  accel_command[1] = ARGNINT;
  accel_command[2] = abs(this->motor_max_trans_decel) & 0x00FF;
  accel_command[3] = (abs(this->motor_max_trans_decel) & 0xFF00) >> 8;
  accel_packet.Build(accel_command, 4);
  this->SendReceive(&accel_packet,false);
}
if(this->motor_max_rot_accel > 0)
{
  accel_command[0] = SETRA;
  accel_command[1] = ARGINT;
  accel_command[2] = this->motor_max_rot_accel & 0x00FF;
  accel_command[3] = (this->motor_max_rot_accel & 0xFF00) >> 8;
  accel_packet.Build(accel_command, 4);
  this->SendReceive(&accel_packet,false);
}
if(this->motor_max_rot_decel < 0)
{
  accel_command[0] = SETRA;
  accel_command[1] = ARGNINT;
  accel_command[2] = abs(this->motor_max_rot_decel) & 0x00FF;
  accel_command[3] = (abs(this->motor_max_rot_decel) & 0xFF00) >> 8;
  accel_packet.Build(accel_command, 4);
  this->SendReceive(&accel_packet,false);
}


P2OSPacket pid_packet;
unsigned char pid_command[4];
if(this->rot_kp >= 0)
{
  pid_command[0] = ROTKP;
  pid_command[1] = ARGINT;
  pid_command[2] = this->rot_kp & 0x00FF;
  pid_command[3] = (this->rot_kp & 0xFF00) >> 8;
  pid_packet.Build(pid_command, 4);
  this->SendReceive(&pid_packet);
}
if(this->rot_kv >= 0)
{
  pid_command[0] = ROTKV;
  pid_command[1] = ARGINT;
  pid_command[2] = this->rot_kv & 0x00FF;
```

```
  pid_command[3] = (this->rot_kv & 0xFF00) >> 8;
  pid_packet.Build(pid_command, 4);
  this->SendReceive(&pid_packet);
}
if(this->rot_ki >= 0)
{
  pid_command[0] = ROTKI;
  pid_command[1] = ARGINT;
  pid_command[2] = this->rot_ki & 0x00FF;
  pid_command[3] = (this->rot_ki & 0xFF00) >> 8;
  pid_packet.Build(pid_command, 4);
  this->SendReceive(&pid_packet);
}
if(this->trans_kp >= 0)
{
  pid_command[0] = TRANSKP;
  pid_command[1] = ARGINT;
  pid_command[2] = this->trans_kp & 0x00FF;
  pid_command[3] = (this->trans_kp & 0xFF00) >> 8;
  pid_packet.Build(pid_command, 4);
  this->SendReceive(&pid_packet);
}
if(this->trans_kv >= 0)
{
  pid_command[0] = TRANSKV;
  pid_command[1] = ARGINT;
  pid_command[2] = this->trans_kv & 0x00FF;
  pid_command[3] = (this->trans_kv & 0xFF00) >> 8;
  pid_packet.Build(pid_command, 4);
  this->SendReceive(&pid_packet);
}
if(this->trans_ki >= 0)
{
  pid_command[0] = TRANSKI;
  pid_command[1] = ARGINT;
  pid_command[2] = this->trans_ki & 0x00FF;
  pid_command[3] = (this->trans_ki & 0xFF00) >> 8;
  pid_packet.Build(pid_command, 4);
  this->SendReceive(&pid_packet);
}


if(this->bumpstall >= 0)
{
  if(this->bumpstall > 3)
```

```
      ROS_INFO("ignoring bumpstall value %d; should be 0, 1, 2, or 3",
               this->bumpstall);
    else
    {
      ROS_INFO("setting bumpstall to %d", this->bumpstall);
      P2OSPacket bumpstall_packet;;
      unsigned char bumpstall_command[4];
      bumpstall_command[0] = BUMP_STALL;
      bumpstall_command[1] = ARGINT;
      bumpstall_command[2] = (unsigned char)this->bumpstall;
      bumpstall_command[3] = 0;
      bumpstall_packet.Build(bumpstall_command, 4);
      this->SendReceive(&bumpstall_packet,false);
    }
  }


  if(use_sonar_) {
    this->ToggleSonarPower(1);
    ROS_DEBUG("Sonar array powered on.");
  }
  ptz_.setup();


  return(0);
}


int P2OSNode::Shutdown()
{
  unsigned char command[20],buffer[20];
  P2OSPacket packet;


  if (ptz_.isOn())
  {
    ptz_.shutdown();
  }


  memset(buffer,0,20);


  if(this->psos_fd == -1)
    return -1;
```

```
  command[0] = STOP;
  packet.Build(command, 1);
  packet.Send(this->psos_fd);
  usleep(P2OS_CYCLETIME_USEC);


  command[0] = CLOSE;
  packet.Build(command, 1);
  packet.Send(this->psos_fd);
  usleep(P2OS_CYCLETIME_USEC);


  close(this->psos_fd);
  this->psos_fd = -1;
  ROS_INFO("P2OS has been shutdown");
  delete this->sippacket;
  this->sippacket = NULL;


  return 0;
}




void
P2OSNode::StandardSIPPutData(ros::Time ts)
{


  p2os_data.position.header.stamp = ts;
  pose_pub_.publish( p2os_data.position );
  p2os_data.odom_trans.header.stamp = ts;
  odom_broadcaster.sendTransform( p2os_data.odom_trans );


  p2os_data.batt.header.stamp = ts;
  batt_pub_.publish( p2os_data.batt );
  mstate_pub_.publish( p2os_data.motors );


  p2os_data.sonar.header.stamp = ts;
  sonar_pub_.publish( p2os_data.sonar );
```

```
  aio_pub_.publish( p2os_data.aio);
  dio_pub_.publish( p2os_data.dio);


  grip_state_pub_.publish( p2os_data.gripper );
  ptz_state_pub_.publish( ptz_.getCurrentState() );
}


int P2OSNode::SendReceive(P2OSPacket* pkt, bool publish_data)
{
  P2OSPacket packet;


  if((this->psos_fd >= 0) && this->sippacket)
  {
    if(pkt)
      pkt->Send(this->psos_fd);


    pthread_testcancel();
    if(packet.Receive(this->psos_fd))
    {
      ROS_ERROR("RunPsosThread(): Receive errored");
      pthread_exit(NULL);
    }


    if(packet.packet[0] == 0xFA && packet.packet[1] == 0xFB &&
       (packet.packet[3] == 0x30 || packet.packet[3] == 0x31 ||
       packet.packet[3] == 0x32 || packet.packet[3] == 0x33 ||
       packet.packet[3] == 0x34))
    {


      this->sippacket->ParseStandard( &packet.packet[3] );
      this->sippacket->FillStandard(&(this->p2os_data));


      if(publish_data)
        this->StandardSIPPutData(packet.timestamp);
    }
    else if(packet.packet[0] == 0xFA && packet.packet[1] == 0xFB &&
```

```
          packet.packet[3] == SERAUX)
    {
      if(ptz_.isOn())
      {
        int len = packet.packet[2] - 3;
        if (ptz_.cb_.gotPacket())
        {
          ROS_ERROR("PTZ got a message, but alread has the complete packet.");
        }
        else
        {
          for (int i=4; i < 4+len; ++i)
          {
            ptz_.cb_.putOnBuf(packet.packet[i]);
          }
        }
      }
    }
    else
    {
      ROS_ERROR("Received other packet!");
      packet.PrintHex();
    }
  }


  return(0);
}


void P2OSNode::updateDiagnostics()
{
  diagnostic_.update();
}


void P2OSNode::check_voltage( diagnostic_updater::DiagnosticStatusWrapper &stat )
{
          double voltage = sippacket->battery / 10.0;
          if( voltage < 11.0 )
          {
                    stat.summary( diagnostic_msgs::DiagnosticStatus::ERROR, "battery voltage
critically low" );
          }
          else if( voltage < 11.75 )
```

```
        {
                stat.summary( diagnostic_msgs::DiagnosticStatus::WARN, "battery voltage
getting low" );


        }
        else stat.summary( diagnostic_msgs::DiagnosticStatus::OK, "battery voltage OK" );


        stat.add("voltage", voltage );
}


void P2OSNode::check_stall( diagnostic_updater::DiagnosticStatusWrapper &stat )
{
        if( sippacket->lwstall || sippacket->rwstall )
        {
                stat.summary( diagnostic_msgs::DiagnosticStatus::ERROR, "wheel stalled"
);
        }
        else stat.summary( diagnostic_msgs::DiagnosticStatus::OK, "no wheel stall" );


        stat.add("left wheel stall", sippacket->lwstall );
        stat.add("right wheel stall", sippacket->rwstall );
}


void P2OSNode::ResetRawPositions()
{
  P2OSPacket pkt;
  unsigned char p2oscommand[4];


  if(this->sippacket)
  {
    this->sippacket->rawxpos = 0;
    this->sippacket->rawypos = 0;
    this->sippacket->xpos = 0;
    this->sippacket->ypos = 0;
    p2oscommand[0] = SETO;
    p2oscommand[1] = ARGINT;
    pkt.Build(p2oscommand, 2);
    this->SendReceive(&pkt,false);
    ROS_INFO("resetting raw positions" );
```

```
  }
}


void P2OSNode::ToggleSonarPower(unsigned char val)
{
  unsigned char command[4];
  P2OSPacket packet;


  command[0] = SONAR;
  command[1] = ARGINT;
  command[2] = val;
  command[3] = 0;
  packet.Build(command, 4);
  SendReceive(&packet,false);
}


void P2OSNode::ToggleMotorPower(unsigned char val)
{
  unsigned char command[4];
  P2OSPacket packet;
  ROS_INFO( "motor state: %d\n", p2os_data.motors.state );
  p2os_data.motors.state = (int) val;
  command[0] = ENABLE;
  command[1] = ARGINT;
  command[2] = val;
  command[3] = 0;
  packet.Build(command, 4);
  SendReceive(&packet,false);
}


inline double P2OSNode::TicksToDegrees (int joint, unsigned char ticks)
{
  if ((joint < 0) || (joint >= sippacket->armNumJoints))
    return 0;


  double result;
  int pos = ticks - sippacket->armJoints[joint].centre;
  result = 90.0 / static_cast<double> (sippacket->armJoints[joint].ticksPer90);
  result = result * pos;
  if ((joint >= 0) && (joint <= 2))
```

```
    result = -result;


  return result;
}


inline unsigned char P2OSNode::DegreesToTicks (int joint, double degrees)
{
  double val;


  if ((joint < 0) || (joint >= sippacket->armNumJoints))
    return 0;


  val = static_cast<double> (sippacket->armJoints[joint].ticksPer90) * degrees / 90.0;
  val = round (val);
  if ((joint >= 0) && (joint <= 2))
    val = -val;
  val += sippacket->armJoints[joint].centre;


  if (val < sippacket->armJoints[joint].min)
    return sippacket->armJoints[joint].min;
  else if (val > sippacket->armJoints[joint].max)
    return sippacket->armJoints[joint].max;
  else
    return static_cast<int> (round (val));
}


inline double P2OSNode::TicksToRadians (int joint, unsigned char ticks)
{
  double result = DTOR (TicksToDegrees (joint, ticks));
  return result;
}


.
inline unsigned char P2OSNode::RadiansToTicks (int joint, double rads)
{
  unsigned char result = static_cast<unsigned char> (DegreesToTicks (joint, RTOD (rads)));
  return result;
}
```

```
inline double P2OSNode::RadsPerSectoSecsPerTick (int joint, double speed)
{
  double degs = RTOD (speed);
  double ticksPerDeg = static_cast<double> (sippacket->armJoints[joint].ticksPer90) / 90.0f;
  double ticksPerSec = degs * ticksPerDeg;
  double secsPerTick = 1000.0f / ticksPerSec;


  if (secsPerTick > 127)
    return 127;
  else if (secsPerTick < 1)
    return 1;
  return secsPerTick;
}


inline double P2OSNode::SecsPerTicktoRadsPerSec (int joint, double msecs)
{
  double ticksPerSec = 1.0 / (static_cast<double> (msecs) / 1000.0);
  double ticksPerDeg = static_cast<double> (sippacket->armJoints[joint].ticksPer90) / 90.0f;
  double degs = ticksPerSec / ticksPerDeg;
  double rads = DTOR (degs);


  return rads;
}
void P2OSNode::SendPulse (void)
{
  unsigned char command;
  P2OSPacket packet;


  command = PULSE;
  packet.Build(&command, 1);
  SendReceive(&packet);
}
```

ANEXO B

Código Fuente del nodo /espol/laserespol

```
#include <csignal>
#include <cstdio>
#include "LMS1xx.h"
#include "ros/ros.h"
#include <LaserEspol/LaserEspol.h>
#define DEG2RAD M_PI/180.0

int main(int argc, char **argv)
{

  LMS1xx laserEspol; //Declaramos la variable laserEspol que tiene la clase LMS1xx
  scanCfg cfg;
  scanDataCfg dataCfg;
  scanData data

  LaserEspol::LaserEspol scan_msg;

  std::string host;
  std::string frame_id;

  ros::init(argc, argv, "lms1xx");
  ros::NodeHandle nh;
  ros::NodeHandle n("~");
  ros::Publisher scan_pub = nh.advertise<LaserEspol::LaserEspol>("scan", 1);

  n.param<std::string>("host", host, "169.254.47.116");

  n.param<std::string>("frame_id", frame_id, "laser

  ROS_INFO("connecting to laser at : %s", host.c_str());

  laserEspol.connect(host);

  if (laserEspol.isConnected())
  {
    ROS_INFO("Connected to laser.");

    laserEspol.login
    cfg = laserEspol.getScanCfg();

    scan_msg.header.frame_id = frame_id;

    scan_msg.range_min = 0.01;
    scan_msg.range_max = 20.0;
```

```
scan_msg.scan_time = 1000.0/cfg.scaningFrequency;

scan_msg.angle_increment = cfg.angleResolution/10000.0 * DEG2RAD;

scan_msg.angle_min = cfg.startAngle/10000.0 * DEG2RAD;

scan_msg.angle_max = cfg.stopAngle/10000.0 * DEG2RAD;

int num_values;

if (cfg.angleResolution == 2500)
{
  num_values = 1081;
}
else if (cfg.angleResolution == 5000)
{
  num_values = 541;
}
else
{
  ROS_ERROR("Unsupported resolution");
  return 0;
}


scan_msg.time_increment = scan_msg.scan_time/num_values;


scan_msg.ranges.resize(num_values);

scan_msg.intensities.resize(num_values);

dataCfg.outputChannel = 1;
dataCfg.remission = true;
dataCfg.resolution = 1;
dataCfg.encoder = 0;
dataCfg.position = false;
dataCfg.deviceName = false;
dataCfg.outputInterval = 1;

laserEspol.setScanDataCfg(dataCfg);

laserEspol.startMeas();

status_t stat;
do
{
  stat = laserEspol.queryStatus();
  ros::Duration(1.0).sleep();
}
while (stat != ready_for_measurement);
```

```
    laserEspol.scanContinous(1);

    while (ros::ok())
    {
      ros::Time start = ros::Time::now();

      scan_msg.header.stamp = start;
      ++scan_msg.header.seq;

      laserEspol.getData(data);

      for (int i = 0; i < data.dist_len1; i++)
      {

            scan_msg.ranges[i] = data.dist1[i] * 0.001;

      }

      for (int i = 0; i < data.rssi_len1; i++)
      {


        scan_msg.intensities[i] = data.rssi1[i];

      }

      scan_pub.publish(scan_msg);


      ros::spinOnce();
    }

    laserEspol.scanContinous(0);

    laserEspol.stopMeas();

    laserEspol.disconnect();
  }
  else
  {
    ROS_ERROR("Connection to device failed");
  }
  return 0;
}
```