Driver de ROS para el robot móvil RoMAA

Gonzalo Perez-Paina, David Gaydou y Gastón Araguás
Centro de Investigación en Informática para la Ingeniería (CIII)
Universidad Tecnológica Nacional, Facultad Regional Córdoba (UTN-FRC)
Córdoba, Argentina
Email: gperez@frc.utn.edu.ar

***Resumen*—El Robot Móvil de Arquitectura Abierta (RoMAA) fue desarrollado íntegramente en el Centro de Investigación en Informática para la Ingeniería (CIII) como una plataforma para llevar a cabo los experimentos en las áreas de investigación de la robótica móvil y visión por computadoras. Por su parte, ROS es un sistema operativo de código abierto que permite desarrollar software de robots como una colección de programas independientes que se ejecutan al mismo tiempo, ampliamente utilizado en investigación en robótica. El presente trabajo describe el desarrollo de un nodo driver de ROS para el robot móvil RoMAA. Dicho driver permite utilizar el robot RoMAA dentro ROS para aprovechar la gran cantidad de paquetes de software disponibles en el ecosistema ROS y poder utilizar dicho entorno como herramienta de desarrollo y evaluación para la generación de nuevos algoritmos de robótica. El nodo driver es el encargado de traducir la información de alto nivel manejada por ROS a comandos de bajo nivel del sistema embebido del robot RoMAA. Se muestra también algunos ejemplos que resultan de la integración del robot RoMAA con ROS.**

I. Introducción

El Robot Móvil de Arquitectura Abierta –RoMAA– (ver Fig. 1) fue desarrollado íntegramente en el Centro de Investigación en Informática para la Ingeniería (CIII) como una plataforma para llevar a cabo los experimentos en las áreas de investigación de la robótica móvil y visión por computadoras [1–4]. El desarrollo del robot RoMAA comienza como proyecto interno del CIII, concluyendo con la fabricación del primer prototipo presentado en [5]. A partir de la evaluación de las características constructivas, funcionales y de costos de fabricación del prototipo se decide introducir algunas modificaciones en el diseño, lo que deriva en la nueva plataforma móvil RoMAA-II [6–8]. Las principales modificaciones están relacionadas a: la mecánica del sistema de tracción, el hardware del controlador embebido junto con el firmware asociado, para el cual se crearon bibliotecas específicas de la arquitectura del microcontrolador utilizado [9].

ROS (Robot Operating System) [10] es un sistema operativo de código abierto para robots, el cual brinda una colección de herramientas y bibliotecas de programación para la creación de aplicaciones de robótica de manera flexible. El proyecto ROS se inició en 2007 en la Universidad de Stanford bajo el nombre de Switchyard. En el año 2008 el desarrollo estuvo a cargo de una empresa de investigación en robótica llamada Willow Garage, en la cual se produjo la mayor parte del desarrollo. En 2013, los investigadores de Willow Garage formaron la OSRF (Open Source Robotics Foundation), encargada en la actualidad del mantenimiento de ROS [11].

ROS proporciona los servicios más comunes de un sistema operativo, tales como la abstracción del hardware, el control de dispositivos de bajo nivel, comunicación entre procesos y mantenimiento de paquetes de software. También incluye herramientas y bibliotecas para obtener, construir y ejecutar código en múltiples ordenadores. ROS está basado en una estructura de grafos donde el procesamiento se lleva a cabo en los nodos, los cuales pueden intercambiar diferentes mensajes, ya sean de sensores, control, estados, planificaciones, etc. [12].

Figura 1. Robot móvil RoMAA-II.

El presente trabajo describe el desarrollo de un nodo driver de ROS para el robot móvil RoMAA[1]. Dicho driver permite utilizar el robot RoMAA dentro ROS para aprovechar la gran cantidad de paquetes de software disponibles en el ecosistema ROS y poder utilizar dicho entorno como herramienta de desarrollo y evaluación para la generación de nuevos algoritmos de robótica. El nodo driver es el encargado de traducir la información de alto nivel manejada por ROS (mensajes) a comandos de bajo nivel del sistema embebido del robot RoMAA. Se muestra el desarrollo de dicho nodo, los diferentes tipos de mensajes implementados por el nodo y algunos ejemplos de aplicación.

La sección II describe el robot móvil RoMAA y la biblioteca de comunicación. En la sección III se realiza una breve descripción de los conceptos básicos de ROS, la creación del espacio de trabajo y del nodo driver, para luego en la sección IV presentar en detalle la implementación de dicho nodo y los mecanismos de comunicación que utiliza. La sección V muestra algunos ejemplos de la integración del robot RoMAA con ROS. Finalmente, la sección VI presenta las conclusiones y los trabajos a futuro.

II. El robot RoMAA

El RoMAA [6] es un robot de tracción diferencial tipo uniciclo de tres ruedas, dos de las cuales son de tracción

[1] https://github.com/ciiiutnfrc/romaa_ros

ISBN: en trámite

controladas individualmente y una rueda castor en la parte trasera del robot que sirve de apoyo.

La estructura mecánica consiste en dos placas de aluminio de idénticas dimensiones unidas entre sí mediante separadores metálicos [5] (ver Fig. 1). La placa inferior da soporte a los componentes fundamentales del robot: baterías, motorreductores, ruedas y drivers de potencia. Los componentes de mayor peso como las baterías están ubicadas en la parte posterior lo cual, en conjunto con el reducido despeje respecto al piso, le confieren al robot una gran estabilidad. El área delantera alberga la electrónica para manejo de los motores de tracción.

La electrónica de a bordo para el control del robot incluye dos drivers de potencia en configuración llave H para la excitación de cada uno de los motores de tracción y una unidad de procesamiento o controlador diferencial [1]. El controlador diferencial es el encargado de establecer la comunicación con la PC de a bordo y de cerrar los lazos de control de velocidad para cada uno de los motores de tracción, también realiza el cálculo de odometría del robot a partir de las lecturas de los codificadores ópticos incrementales. La trama de comunicación a bajo nivel sigue el mismo esquela que la utilizada por el módulos de sensores presentado en [7].

II-A. Biblioteca de comunicación de bajo nivel

El controlador diferencial del robot RoMAA se comunica con la PC de a bordo mediante un puerto USB, la PC le envía comandos de velocidad al robot y lee la información de la odometría, entre otras funciones. Del lado de la PC, la comunicación se realiza utilizando una biblioteca[2] implementada como clase (Programación Orientada a Objetos) de C++, en la cual cada funsión miembro se corresponden uno a uno con los comandos de comunicación de bajo nivel. Algunas funciones miembros más relevantes para la implementación del nodo driver presentado aquí son:

- `set_speed`, `get_speed`: para ajustar y leer la velocidad lineal y angular,
- `reset_odometry`, `set_odometry`: para fijar a cero u otro valor la odometría,
- `enable_motor`, `disable_motor`: para habilitar y deshabilitar los motores de tracción.

III. El Sistema Operativo de Robótica – ROS

Uno de los objetivos principales de ROS es permitir a los desarrolladores diseñar el software de los robots como una colección de programas, en su mayoría independientes, llamados nodos que se ejecutan al mismo tiempo. Para que esto funcione, estos nodos deben ser capaz de comunicarse entre sí. La parte de ROS que facilita esta comunicación se llama *ROS Master*, el cual se ejecuta mediante el comando `roscore`. El ROS Master también contiene al servidor de parámetros. En este contexto, un sistema ROS queda representado como un grafo donde cada entidad se identifica por su nombre, en lo que se conoce como *nombre de recurso en el grafo*.

Algunos de la términos utilizados en ROS son [13]:

- **Nombre de recurso (en el grafo)**: es una cadena de texto que identifica a un recurso dentro del grafo computacional de ROS. El esquema de nombres es jerárquico y tiene muchos aspectos en común con las rutas del sistema de directorios de UNIX; como por ejemplo que pueden ser absolutos o relativos.
- **Nodo**: es cualquier proceso que utiliza una API cliente de ROS.
- **Tópico**: es el nombre del canal de comunicación, el cual es unidireccional, asíncrono, fuertemente tipado y utiliza el mecanismo de comunicación publicador/subscriptor.
- **Mensaje**: es una estructura de datos específica basada en un conjunto de datos básicos que describe el tipo de dato de un tópico particular. Los mensajes pueden ser anidados de forma arbitraria aunque no ofrece ningún tipo de mecanismo de herencia.
- **Servicio**: es una llamada a un procedimiento remoto síncrono.
- **Servidor de parámetro**: es un diccionario compartido accesible mediante la API de red. Está previsto para tener un intercambio lento de datos, tal como argumentos de inicialización.

Todos los recursos dentro de ROS, sean tópicos, parámetros o servicios, quedan identificados mediante nombres los cuales pueden ser globales, relativos o privados.

1. Los nombres globales son únicos y se identifican por la barra inicial `'/'`.
2. Los nombres relativos son aquellos que ROS resuelve internamente armando el nombre global anteponiendo espacios de nombres o el espacio de nombre global `'/'`.
3. Los nombres privados son similares a los nombres relativos, anteponiendo como espacio de nombre el nombre del propio nodo.

Si se declara el manejador de un nodo con el argumento `'~'` (como en el Listado 3), todos los nombres de recursos quedan definidos como privados.

III-A. Espacio de trabajo y creación de un paquete

Los paquetes de ROS se desarrollan dentro de un directorio conocido como *workspace* o espacio de trabajo, el cual debe contener un subdirectorio de nombre `src` para almacenar el código fuente de los nodos del paquete.

La creación del espacio de trabajo (directorio `catkin_ws`), de un paquete y su construcción (build/make) se muestra en el Listado 1. El comando crea el paquete `romaa_ros` el cual dependen de los paquetes `roscpp` (biblioteca cliente C++ de ROS), `std_msgs` (mensajes estándares) y de `nav_msgs` (mensajes comunes de navegación). Al crear el paquete se generan dos archivos:

- `package.xml`: es el archivo manifiesto del paquete donde se definen detalles sobre el paquete tales como: su nombre, versión, mantenedor y dependencias.
- `CMakeLists.txt`: es un script para el sistema de construcción `CMake`, que contiene una lista de instrucciones para crear los binarios a partir de los archivos fuentes y las bibliotecas a enlazar.

IV. Nodo driver del robot RoMAA

El nodo driver implementado para el robot RoMAA utiliza los mecanismos de comunicación de *publicador/subscriptor* y de llamadas a *servicios*. El subscriptor recibe comandos de velocidad, el publicador envía información de odometría y los servicios agregan funciones como el reinicio de odometría y la habilitación/deshabilitación de los motores de tracción.

[2] https://github.com/ciiiutnfrc/romaa_comm

ISBN: en trámite

Listado 1. Creación de espacio de trabajo y paquete.

```
> mkdir -p catkin_ws/src
> cd catkin_ws/src
> catkin create pkg romaa_ros --catkin-deps\
roscpp std_msgs nav_msgs
> cd ..
> catkin build
```

Los mensajes con información de la odometría se publican en el tópico `romaa/odom`, y la subscripción a comandos de velocidad en el tópico `romaa/cmd_vel`. Los servicios implementados son `reset_odometry`, `set_odometry` y `enable_motor`.

El código fuente del nodo debe encontrarse bajo el directorio `src` del paquete, y para su compilación es necesario agregar las líneas del Listado 2 al archivo `CMakeLists.txt` del paquete.

Listado 2. `CMakeLists.txt` para la construcción del nodo.

```
1 add_executable(romaa_driver romaa_driver.cpp)
2 target_link_libraries(romaa_driver
3                       ${catkin_LIBRARIES})
4 target_link_libraries(romaa_driver romaa_comm)
```

Listado 3. Nodo driver del robot RoMAA.

```
1  #include <ros/ros.h>
2  // Include header files of used messages.
3
4  int main(int argc, char **argv)
5  {
6    ros::init(argc, argv, "romaa");
7    ros::NodeHandle nh("~");
8
9    // Define publisher and subscribers and
10   // subscriber's callback functions.
11
12   ros::Time current_time;
13   ros::Rate loop_rate(10);// 10 Hz
14   while(ros::ok())
15   {
16     // Fill messages with data to publish and
17     // publish them. Use 'current_time' in msg
18     // timestamp.
19
20     // Process incoming messages via callbacks.
21     ros::spinOnce();
22
23     // Sleep for the time remaining to achieve the
24     // publish rate.
25     loop_rate.sleep();
26   }
27   return 0;
28 }
```

El código fuente base del nodo se muestra en el Listado 3. En la línea 6 se inicializa la biblioteca cliente de ROS y se debe llamar una única vez al comienzo del programa, su último parámetro define el nombre por defecto del nodo. La línea 7 define el manejador del nodo, cuyo parámetro indica que el nodo utiliza nombres privados para la identificación de sus recursos. En la línea 12 se define un objeto reloj para cargar los timestamps en los mensajes a publicar. En la línea 13 se especifica la frecuencia de ejecución del hilo del nodo, manteniendo un registro del tiempo transcurrido desde la última llamada (`Rate::sleep()`). La función `spinOnce` de la línea 21 permite a ROS ejecutar las funciones callback de los objetos `Subscriber` del nodo, y el método `sleep` de la línea 25 espera el tiempo necesario para cumplir la tasa de 10Hz de publicación de mensajes mediante objetos `Publisher`.

Para la ejecución del nodo se utilizan los comandos del Listado 4. La primer línea corre el Master de ROS y la segunda ejecuta el nodo indicando el nombre del paquete y del nodo.

Listado 4. Ejecución del nodo.

```
> roscore
> rosrun romaa_ros romaa_driver
```

IV-A. Publicador de mensajes de odometría

El Listado 5 muestra las líneas de código fuente a agregar al Listado 3 para la publicación de mensajes de odometría.

Listado 5. Publicación de odometría.

```
1 #include <nav_msgs/Odometry.h>
2
3 nav_msgs::Odometry odom_msg; // Odometry message.
4 ros::Publisher odom_pub =
5   nh.advertise<nav_msgs::Odometry>("odom", 100);
6
7 /* Fill 'odom' message with data */
8 odom_pub.publish(odom_msg); // Publish de message.
```

En la línea 3 se declara el objeto de mensaje de odometría. La línea 4 declara el objeto publicador de mensajes de tipo `nav_msgs/Odometry` bajo el tópico `odom`. Su último parámetro fija el tamaño de la cola de mensajes. La línea 7 publica el mensaje (broadcast), esta línea debe ir dentro del bucle principal del nodo (línea 14 del Listado 3) luego de cargar el mensaje de odometría (objeto `odom_msg`).

Las líneas de código del Listado 6 muestra como cargar el mensaje de odometría.

Listado 6. Mensaje de odometría.

```
1  odom_msg.header.stamp = current_time;
2
3  // 'odom' frame: world-fixed frame.
4  odom_msg.header.frame_id = "odom";
5  // 'base_link' frame: attached to the mobile robot.
6  odom_msg.child_frame_id = "base_link";
7
8  // Quaternion created from yaw.
9  geometry_msgs::Quaternion odom_quat_msg =
10   tf::createQuaternionMsgFromYaw(a);
11
12 // Set the position.
13 odom_msg.pose.pose.position.x = x;
14 odom_msg.pose.pose.position.y = y;
15 odom_msg.pose.pose.position.z = 0;
16 odom_msg.pose.pose.orientation = odom_quat_msg;
17
18 // Set the velocity (in the child frame).
19 odom_msg.twist.twist.linear.x = v;
20 odom_msg.twist.twist.linear.y = 0;
21 odom_msg.twist.twist.angular.z = w;
```

La línea 1 carga el timestamp al mensaje de odometría, donde `current_time` es un objeto `ros::Time` (línea 12 del Listado 3). En las líneas 4 y 6 se definen los nombres de los

ISBN: en trámite

marcos de referencia global y local al robot, respectivamente. La línea 9 calcula la orientación del robot como cuaternión unitario a partir del ángulo de guiñada. En las líneas 13 a 16 se carga el mensaje de la posición y orientación del robot y en las líneas 19 a 21 las velocidades.

Las variables de odometría (`x`, `y` y `a`) y de velocidad (`v` y `w`) son leídas desde el robot utilizando la biblioteca de comunicación.

IV-B. Subscriptor a comandos de velocidad

El Listado 7 muestra las líneas de código a agregar al Listado 3 para subscribirse a comandos de velocidad.

Listado 7. Subscripción de comandos de velocidad.

```
1  #include <geometry_msgs/Twist.h>
2
3  ros::Subscriber cmd_vel_sub =
4    nh.subscribe<geometry_msgs::Twist>("cmd_vel",
5    1, cmdVelCallback);
6
7  void cmdVelCallback(
8  const geometry_msgs::Twist::ConstPtr& cmd_vel)
9  {
10   // Setting speed to the robot.
11   ROS_DEBUG_STREAM("Setting speed. v: "
12     << cmd_vel->linear.x
13     << ", w: " << cmd_vel->angular.z);
14 }
```

La línea 3 declara el objeto subscriptor para el tópico de nombre `cmd_vel` y mensajes de tipo `geometry_msgs::Twist`. El segundo parámetro es el tamaño de la cola de mensajes y el último la función callback que procesa el mensaje. Esta se ejecuta si hay mensajes en la cola al ejecutarse la función `ros::spinOnce()` del Listado 3. La línea 7 define la función callback del mensaje de velocidad. Esta función debe enviar las velocidades al robot.

IV-C. Publicación de transformaciones (`tf`)

Para realizar la conversión entre los diferentes sistemas de coordenadas pertenecientes a un robot, ROS provee de la biblioteca de transformaciones (paquete `tf2`) [14]. El Listado 8 muestra la sección de código necesaria para publicar mensajes `tf`, el cual relaciona el sistema de coordenadas del robot (`base_link`) y el de odometría (`odom`).

En la línea 5 se define el objeto para la publicación de las transformaciones utilizando el objeto definido en la línea 6. De la línea 10 a la 20 se carga el mensaje de transformación y se publica en la línea 23. Esta transformación relaciona el sistema de coordenadas `odom` (global) con el sistema `base_link` del robot.

IV-D. Parámetros de comunicación por puerto serie

El Listado 9 muestra las líneas de código necesarias para utilizar parámetros dentro de un nodo. Al colocar esta sección de código dentro del Listado 3, los nombres de parámetros quedan bajo el espacio de nombre del nodo.

La línea 3 declara un parámetro de tipo `std::string` de nombre `port` almacenado en la variable `port` con valor por defecto `/dev/ttyUSB0`, y la línea 4 uno de tipo `int`

Listado 8. Mensaje de transformaciones (`tf`).

```
1  #include <tf2_ros/transform_broadcaster.h>
2  #include <tf2/LinearMath/Quaternion.h>
3  #include <tf2_geometry_msgs/tf2_geometry_msgs.h>
4
5  tf2_ros::TransformBroadcaster tf_broardcaster;
6  geometry_msgs::TransformStamped odom_tf;
7  tf2::Quaternion odom_quat_tf;
8  geometry_msgs::Quaternion odom_quat_msg;
9
10 odom_tf.header.stamp = current_time;
11 odom_tf.header.frame_id = "odom";
12 odom_tf.child_frame_id = "base_link";
13
14 odom_quat_tf.setRPY(0, 0, a);
15 odom_quat_msg = tf2::toMsg(odom_quat_tf);
16
17 odom_tf.transform.translation.x = x;
18 odom_tf.transform.translation.y = y;
19 odom_tf.transform.translation.z = 0.0;
20 odom_tf.transform.rotation = odom_quat_msg;
21
22 // Send the transform.
23 tf_broadcaster.sendTransform(odom_tf);
```

Listado 9. Definición de parámetros.

```
1  std::string port;
2  int baudrate;
3  nh.param<std::string>("port", port, "/dev/ttyUSB0");
4  nh.param<int>("baudrate", baudrate, 115200);
5
6  ROS_INFO_STREAM("Opening port " << port
7    << " at " << baudrate);
```

de nombre `baudrate` almacenado en la variable `baudrate` con valor por defecto `115200`.

El Listado 10 muestra el resultado al ejecutar el nodo, donde se observa los parámetros por defecto y luego cómo modificarlos desde la línea de comandos. Los parámetros `~port` y `~baudrate` desde la línea de comandos se acceden con el nombre `_port` y `_baudrate`, respectivamente; asignándoles valores utilizando `:=` (similar al remapeo de nombres).

Listado 10. Modificación de parámetros de comunicación.

```
> rosrun romaa_ros romaa_driver
[ INFO]: Opening port /dev/ttyUSB0 at 115200
> rosrun romaa_ros romaa_driver _port:=/ttyUSB1\
  _port:=38400
[ INFO]: Opening port /dev/ttyUSB1 at 38400
```

IV-E. Servidor para llamada a servicios

Los servicios de ROS permiten a un nodo *cliente* enviar datos llamados *petición* (request) a un nodo *servidor* y espera una respuesta. El servidor, al recibir la petición, realiza ciertas acciones y envía datos de regreso al cliente llamado *respuesta* (response). El nodo driver del robot RoMAA actúa como servidor para llamadas a servicios para funciones adicionales de operación del robot. El Listado 11 muestra la definición de los servicios utilizados.

Las líneas 2, 6 y 10 definen los servidores cuyos nombres son `reset_odometry`, `set_odometry` y `enable_motor`, respectivamente; indicando también las funciones callbacks correspondientes. Para el mensaje del servicio `reset_odometry` se utiliza un mensa-

ISBN: en trámite

Listado 11. Definición de servicios.

```
// Register servers with the Master
ros::ServiceServer reset_odom_srv =
  nh.advertiseService("reset_odometry",
  &resetOdometrySrvCb);

ros::ServiceServer set_odom_srv =
  nh.advertiseService("set_odometry",
  &setOdometrySrvCb);

ros::ServiceServer motor_srv =
  nh.advertiseService("enable_motor",
  &enableMotorSrvCb);

bool resetOdometrySrvCb(
    std_srvs::Empty::Request &req,
    std_srvs::Empty::Response &resp)
{
  // Send reset command to the robot
  return true;
}

bool setOdometrySrvCb(
  romaa_ros::SetOdometry::Request &req,
  romaa_ros::SetOdometry::Response &resp)
{
  // Set robot odometry using:
  // req.x, req.y and req.theta
  return true;
}

bool enableMotorSrvCb(
  std_srvs::SetBool::Request &req,
  std_srvs::SetBool::Response &resp)
{
  if(req.data == true) {
    romaa->enable_motor();
    resp.success = true;
    resp.message = "Motor enabled.";
  }
  else {
    romaa->disable_motor();
    resp.success = true;
    resp.message = "Motor disabled.";
  }
  return true;
}
```

je estándar vacío de tipo `std_msgs::Empty`, mientras que para el servicio `enable_motor` se utiliza el mensaje `std_msgs::SetBool`. Para el servicio `set_odometry` se utiliza el mensaje `romaa_ros::SetOdometry` definido en el propio paquete `romaa_ros`, mostrado en el listado 12. Este mensaje está definido en el archivo `SetOdometry.srv` bajo el directorio `srv` del paquete.

Listado 12. Mensaje de servicio `SetOdometry`.

```
> rossrv show romaa_ros/SetOdometry
float32 x
float32 y
float32 theta
---
```

V. Integración del robot RoMAA con ROS

V-A. *Teleoperación del robot*

Es posible realizar la teleoperación del robot utilizando un joystick y nodos de ROS para tal efecto. El Listado 13 muestra como ejecutar los nodos necesarios de ROS para realizar la teleoperación de un robot con un joystick. Estos nodos actúan de driver de un joystick USB y publica mensajes de comandos de velocidad para el robot.

Listado 13. Ejecución de nodos para teleoperación.

```
> roscore
> rosrun romaa_ros romaa_driver
> rosrun joy joy_node
> rosrun rosrun teleop_twist_joy teleop_node\
  cmd_vel:=romaa/cmd_vel
```

La Figura 2 muestra la interacción entre nodos. Las elipses representan los nodos y los rectángulos los tópicos. El nodo driver del robot se debe subscribir al tópico `cmd_vel` para recibir los comandos de velocidad desde el joystick, para lo cual hay que modificar el nombre de los tópicos. Al ejecutar el nodo de teleoperación (`teleop_node`) se modifica el nombre del tópico sobre el cual publica los mensajes de velocidad, haciendo `cmd_vel:=romaa/cmd_vel`.

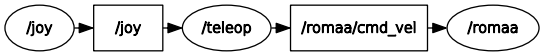

Figura 2. Nodos de ROS para teleoperación.

V-B. *Archivos de lanzamiento*

La herramienta `roslaunch` permite lanzar varios nodos de forma simultánea y definir el valor de sus parámetros, además de realizar el remapeo de nombres. Esta herramienta recibe como argumento un archivo de configuración con extensión `.launch` en formato XML donde se indica los nodos a lanzar.

El Listado 14 muestra el archivo de lanzamiento para lanzar el nodo driver del robot RoMAA con la opción de modificar los parámetros de comunicación del puerto serie con argumentos pasados desde la línea de comandos.

Listado 14. Archivo launch del nodo de RoMAA (`romaa.launch`).

```
<?xml version="1.0"?>
<launch>
  <arg name="port" default="/dev/ttyUSB0" />
  <arg name="baudrate" default="115200" />

  <node pkg="romaa_ros" type="romaa_driver"
   name="romaa" >
    <param name="port" value="$(arg port)" />
    <param name="baud" value="$(arg baudrate)" />
  </node>
</launch>
```

El Listado 15 muestra el archivo de lanzamiento para lanzar los nodos necesarios para la teleoperación del robot; el nodo driver del joystick (`joy`) y el nodo `teleop_node` que se subscribe a los mensajes del anterior. Al lanzar `teleop_node` se definen los parámetros que determinan los ejes del joystick a utilizar y los valores máximos de velocidad lineal y angular. Se realiza también el remapeo de nombres para lograr la conexión con el nodo driver del robot RoMAA, como se observa en la Figura 2.

El Listado 16 muestra el archivo de lanzamiento para ejecutar un nodo que permite tomar imágenes desde una cámara.

ISBN: en trámite

Listado 15. Archivo launch para teleoperación (`joy_teleop.launch`).

```
<?xml version="1.0"?>
<launch>
  <node pkg="joy" type="joy_node" name="joy" >
    <param name="autorepeat_rate" value="5.0" />
  </node>

  <node pkg="teleop_twist_joy" type="teleop_node"
   name="teleop" >
    <param name="enable_button" value="5" />
    <param name="axis_linear" value="1" />
    <param name="axis_angular" value="2" />
    <param name="scale_linear" value="0.35" />
    <param name="scale_angular" value="0.60" />

    <remap from="/cmd_vel" to="/romaa/cmd_vel" />
  </node>
</launch>
```

Listado 16. Archivo launch para cámara (`cam.launch`).

```
<?xml version="1.0"?>
<launch>
  <node pkg="uvc_camera" type="uvc_camera_node"
   name="cam" >
    <param name="device" value="/dev/video0" />
  </node>
</launch>
```

Los archivos de lanzamiento mostrados se puede lanzar de forma independiente o bien generar un archivo donde se incluyan todos los nodos.

V-C. *Herramienta* `rqt`

`rqt` es una herramienta con interfaz gráfica de usuario que permite visualizar información de mensajes de ROS, realizar llamadas a servicios, inspeccionar los nodos en ejecución y su interconexión, entre otras funciones. En la Fig. 3 se observa la interfaz gráfico de `rqt`. Arriba a la izquierda se puede observan los nodos en ejecución: nodo driver del robot RoMAA, nodo joystick y de teleoperación y nodo de cámara. En la parte de abajo a la izquierda se visualizan las mediciones de odometría (posición x e y) como gráfico y las imágenes obtenidas de la cámara en la sección de abajo a la derecha. También, se observa arriba a la derecha la interfaz para la llamada a servicios.

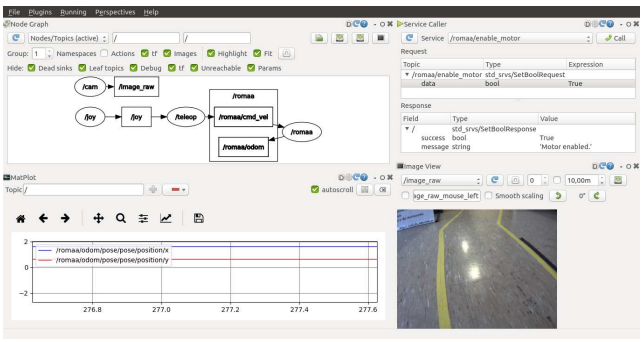

Figura 3. Interfaz gráfica de usuario `rqt`.

VI. Conclusiones y trabajos futuros

Se presentó el desarrollo de un nodo driver para el robot móvil RoMAA a fin de poder integrar dicho robot de desarrollo propio al ecosistema ROS. Se describió en detalle la programación de dicho nodo y los tipos de comunicación utilizados. Los ejemplos de integración del robot RoMAA con ROS muestra la flexibilidad que brinda ROS de generar diferentes configuraciones para llevar a cabo los experimentos en las áreas de investigación de la robótica móvil y visión por computadoras.

Como trabajo a futuro se plantea ampliar las posibilidades del nodo driver para incluir el ajuste de los parámetros del controlador PID de los motores del robot utilizando el paquete de ROS de reconfiguración dinámica (dynamic reconfiguration). Como también desarrollar los modelos de visualización del robot para utilizar la herramienta RViz y los modelos de simulación para el simulador dinámico Gazebo.

Agradecimientos

Este trabajo está financiado por la Universidad Tecnológica Nacional bajo el proyecto UTN-PID 4884 y UTN-PID-UTI 3832.

Referencias

[1] G. F. Perez-Paina, “Navegación autónoma de robot móvil en ambientes parcialmente estructurados usando visión artificial,” Ph.D. dissertation, National University of Technology (UTN), April 2015, (in spanish).

[2] G. M. Steiner, “Sistema de visión computarizada para reconocimiento de entorno,” Ph.D. dissertation, National University of Technology (UTN), 2012, (in spanish).

[3] D. Gonzalez-Dondo, “Algoritmos dinámicamente distribuidos de fusión de datos en redes de sensores,” Ph.D. dissertation, National University of Technology (UTN), 2018, (in spanish).

[4] R. G. Araguás, “Algoritmos de localización visual monocular para robots móviles,” Ph.D. dissertation, National University of Technology (UTN), 2012, (in spanish).

[5] D. A. Gaydou, G. F. Pérez Paina, G. M. Steiner, and J. Salomone, “Plataforma móvil de arquitectura abierta,” in *Proceedings of the V Jornadas Argentinas de Robótica (JAR)*. Ediuns, November 2008.

[6] G. Perez Paina, G. Araguas, D. Gaydou, G. Steiner, and L. Rafael Canali, “RoMAA-II, an open architecture mobile robot,” *Latin America Transactions, IEEE (Revista IEEE America Latina)*, vol. 12, no. 5, pp. 915–921, Aug 2014.

[7] G. F. Perez Paina, F. E. Elizondo, D. A. Suares, and L. R. Canali, “Design and implementation of a multi-sensor module for mobile robotics applications,” in *Proceedings of the Argentine Conference on Embedded Systems (CASE)*, 2012, pp. 269–274.

[8] G. F. Perez Paina and D. A. Gaydou, “Programación y simulación en robótica móvil utilizando Player/Stage,” in *Proceedings of the VI Jornadas Argentinas de Robótica (JAR)*, November 2010, pp. 150–155.

[9] G. F. Perez Paina, D. A. Gaydou, N. L. Palomeque, and L. A. Martini, “Librerías embebidas para microcontroladores LPC2000 de aplicación en robótica,” in *Proceedings of the Argentine Conference on Embedded Systems (CASE)*, 2011.

[10] M. Quigley, B. P. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Y. Ng, “Ros: an open-source robot operating system,” 2009.

[11] L. Joseph, *ROS Robotics Projects*. Packt Publishing, March 2017.

[12] J. M. O’Kane, *A Gentle Introduction to ROS*. Independently published, October 2013, available at http://www.cse.sc.edu/~jokane/agitr/.

[13] A. Bihlmaier and H. Wörn, *Hands-on Learning of ROS using Common Hardware*, ser. Studies in Computational Intelligence. Springer, Cham, 2015, vol. 625, pp. 29–50.

[14] T. Foote, “tf: The transform library,” in *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, ser. Open-Source Software workshop, April 2013, pp. 1–6.