# Proyecto 1: Generación de trayectorias en ROS

Eduardo David Martínez Neri

*Instituto Tecnológico Autónomo de México*

***Resumen*—Se describe la serie de pasos que llevo a producir el primer proyecto de la materia de Robótica, para la Maestría de Ciencias en Computación. El cual consta de crear un programa con ROS (Robot Operating System), que sea capaz de moverse a ciertas coordenadas indicadas por el usuario, de acuerdo a la posición y orientación en que se encuentre.**

***Keywords*—ROS, publish, subscribe, Turtlesim**

## I. Introducción

Este documento esta basado en: *A Gentle Introduction To ROS* por JasonM. O'Kane [1], el cual es una especie de curso introductorio a ROS y una serie de herramientas básicas, pero muy útiles, que un usuario de ROS debe conocer.

Pero ¿Qué es ROS?, ROS es un framework para el desarrollo de software para robots, sus siglas vienen de Robot Operating System (Sistema Operativo Robótico), en términos de software se puede describir como un middleware, open-source, que provee la funcionalidad de un sistema operativo en un clúster heterogéneo. Provee los servicios estándar de un sistema operativo tales como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes. Está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos que pueden recibir (suscribirse), mandar (publicar) y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros [2].

La librería está orientada para un sistema UNIX (Ubuntu (Linux)) aunque también se está adaptando a otros sistemas operativos como Fedora, Mac OS X, Arch, Gentoo, OpenSUSE, Slackware, Debian o Microsoft Windows, considerados como 'experimentales'. En este proyecto, se utilizó la versión Indigo (versión liberada en Julio de 2014), para poder instalar esta versión se requiere Ubuntu 14.04, se intentó instalar sobre MAC OSX El Capitan y Ubuntu 16.04 pero en ambos casos se encontraron errores.

ROS tiene dos partes básicas: la parte del sistema operativo, ROS, como se ha descrito anteriormente y ROS-PKG, una suite de paquetes aportados por la contribución de usuarios (organizados en conjuntos llamados pilas o en inglés stacks) que implementan la funcionalidades tales como localización y mapeo simultáneo, planificación, percepción, simulación, etc.

ROS provee también de una serie de herramientas y librerías para obtener, construir, escribir y ejecutar código en un ambiente de computo distribuido.

### I-A. *Turtlesim*

Es un simulador, el cual muestra un *turtlebot* en una ventana al centro fig. 1 y mediante interacción con otros nodos puede moverse fig. 2. *Tutlesim* utiliza el objeto $geometry\_msgs/Twist$[1] para moverse, objeto que recibe a través del tópico $cmd\_vel$ .

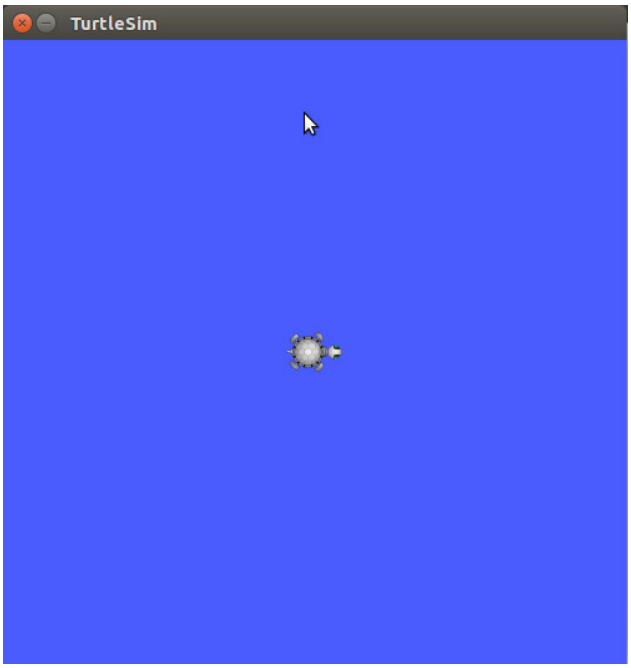


Figura 1. Ventana Turtlesim

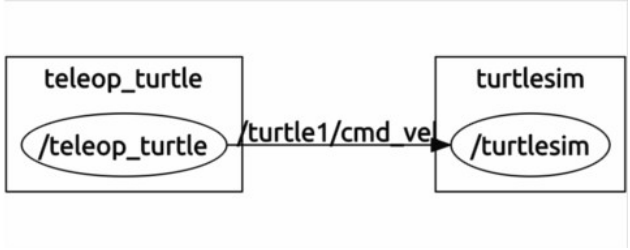


Figura 2. Diagrama de interacción de Turtlesim con el nodo $turtle\_teleop\_key$, el cual se encarga de mover la tortuga mediante el teclado.

## II. Planteamiento del problema

El proyecto consiste en entender el funcionamiento de *Turtlesim* (paquete de ROS), y la estructura de los tópicos

[1]Twist es un objeto compuesto por 2 vectores, los cuales almacenan cada uno 3 valores flotantes, y son utilizados para posición y orientación.

a los que se suscribe y publica, mediante el desarrollo de un programa para generar trayectorias para las tortugas de *Turtlesim*, la descripción del programa es la siguiente:

- Debe ejecutarse desde la terminal.
- Pedir al usuario la pose final $p_f$ (una pose está compuesta por las coordenadas $(x, y)$ y un ángulo $\theta$) y el tiempo de ejecución deseado $t_{ejecución}$.
- Leer la pose actual $p_a$ de la tortuga que publica *Turtlesim*.
- Enviar los mensajes de velocidad necesarios a una frecuencia de 10Hz para que la tortuga siga una trayectoria recta de pose actual $p_a$ a pose final $p_f$, en un tiempo de ejecución $t_{ejecución}$. Durante la ejecución, se debe mantener actualizada la pose actual $p_a$ con los datos publicados por *Turtlesim*.
- Regresar al paso 3 de forma iterativa, hasta llegar al destino.

### *II-A. Publish-Subscribe*

La codificación del problema planteado se basa en entender el funcionamiento del modelo Publish-Subscribe utilizado por ROS, este modelo y su implementación se describen en el capítulo 3 de *A Gentle Introduction To ROS*. Este modelo funciona con envío de mensajes mediante **publicadores o publishers**, los cuales distribuyen mensajes sin conocer de quién los está recibiendo, si es que hay alguien, en vez de esto categorizan los mensajes publicados en tópicos, y a su vez **suscriptores o suscribers** reciben mensajes en base a los tópicos de su interés a los que están suscritos, sin conocer qué **publicadores** estén enviando los mensajes, si es que hay alguno.

ROS utiliza el comando $rqt\_graph$ que sirve como ayuda para observar que nodos están suscritos a que tópicos, o que nodos están publicando hacia que tópicos, mediante este comando se pueden observar gráficas como la que se muestra en la fig. 2.

## III. Implementación

En el programa desarrollado implementó un publisher hacia $turtle1/cmd\_vel$, el cual es el nombre del tópico, hacia el cual deseamos publicar. *Turtlesim* está suscrito a este tópico y mediante este recibe la velocidad y dirección a la que se debe mover la tortuga.

```
// Un objeto publisher hacia el tópico
// cmd_vel, con cola de mensajes de
// tamaño 1000
ros::Publisher pub = nh.advertise<
geometry_msgs::Twist>("turtle1/cmd_vel"
, 1000);

// Un ejemplo del mensaje enviado por
// este publisher es el siguiente:
geometry_msgs::Twist msg;
//velocidad
msg.linear.x = velocidad;
// orientación en radianes
msg.angular.z = angulo;
// finalmente la instrucción que envía
// el mensaje
pub.publish(msg);
```

El código anterior debe publicar la orientación y velocidad a la que se debe mover el robot de acuerdo a su posición actual. Para lograr llegar al punto final en un tiempo $t_{ejecución}$, se debe calcular la velocidad mediante la formula:

$$velocidad = distancia/t_{ejecución} \tag{1}$$

Este cálculo se realiza solo una vez, con el objetivo de mantener una velocidad constante y se logre llegar en el tiempo $t_{ejecución}$, se calcula la $distancia$ euclideana entre las coordenadas actuales $(x_{actual}, y_{actual})$ y las coordenadas finales $(x_{final}, y_{final})$.

Por otro lado, la orientación a la que debe dirigirse la tortuga depende de su pose actual, este ángulo se calcula mediante la formula:

$$\theta = \arctan(y_{final} - y_{actual}/x_{final} - x_{actual}) \tag{2}$$

Y para obtener la pose actual se utiliza un suscriptor a *turtle1/pose*, este suscriptor mediante un callback proporciona la pose actual de la tortuga, y utilizando estos parámetros se codifica la solución dentro de este callback mediante las formulas anteriores, y posteriormente se envía la velocidad y angulo calculado a *Turtlesim*, mediante el publicador del código anterior.

Codificación del suscriptor

```
// declaración del suscriptor
ros::Subscriber sub =
    nh.subscribe("turtle1/pose", 1000,
    &poseMessageReceived);

// ciclar a 10Hz hasta que el nodo se apague
ros::Rate rate(10);
while(ros::ok()) {
  // let ROS take over
  ros::spinOnce();
  rate.sleep();
}
```

Callback utilizado por el suscriptor, el cual es invocado de manera asíncrona, en este código se utiliza la instruccion $ROS\_INFO\_STREAM$, la cual es una instrucción para impresión en pantalla de un mensaje.

```
void poseMessageReceived(const
                turtlesim::Pose& msg) {
  double xprima = x_fin - msg.x;
  double yprima = y_fin - msg.y;
  double distance = sqrt(pow(xprima,2)
                    +pow(yprima,2));

  // obtener angulo al cual se debe mover
  double angle = atan2 (yprima, xprima);
  angle = angle - msg.theta;

  double velocidad = velocidad_general;
```

```
  // la velocidad se mantiene, solo
  // se calcula una vez
  if(!calcular_velocidad) {
    velocidad_general=distance/tiempo;
    calcular_velocidad = true;
  }

  double holgura = 0.5;
  if(distance < holgura) {
    velocidad = 0;
    double dif = theta_fin - msg.theta;
    if(dif < 0.01 && dif > -0.01) {
      ROS_INFO_STREAM("POSE FINAL
                       ALCANZADA");
      ros::shutdown();
    }
    else {
      // enviar un angulo para buscar
      // la pose final
      angle = theta_fin - msg.theta;
    }
  }

  // crear mensaje.
  geometry_msgs::Twist msg2;
  msg2.linear.x = velocidad;
  msg2.angular.z = angle;

  // publish the message
  pub.publish(msg2);
}
```

### *III-A. Ejecución y compilación*

Para lograr ejecutar el programa se realiza mediante la instrucción *rosrun*, antes es necesario crear un workspace, el cual contiene el paquete que se compila mediante la instruccion *catkin_make*, las instrucciones completas para lograr compilar el primer proyecto en ROS se encuentran en [1], así como la configuración que se debe realizar para ligar las dependencias del proyecto y establecer los ejecutables. En nuestro caso el paquete se llama *agitr* y el nombre del ejecutable **agitr_moveto**, *agitr* es el nombre del paquete utilizado a lo largo del documento *A Gentle Introduction To ROS*, por tanto se mantuvo este nombre y se agregaron los parametros requeridos en la invocación, de acuerdo al requerimiento, finalmente el comando terminó de la siguiente manera:

```
rosrun agitr agitr_moveto _x_fin:=10
_y_fin:=10 _theta_fin:=0 _tiempo:=4
```

En esta invocación se le está indicando a la tortuga que se mueva a las coordenadas $(10, 10)$, con una pose final de 0 radianes, y en un tiempo de 4 segundos, en la fig. 3, se puede observar el movimiento realizado por la tortuga después de 2 movimientos.

Una vez alcanzada la pose final, se detiene la ejecución del nodo en ejecución **agitr_moveto** mediante la instrucción:

```
ros::shutdown();
```

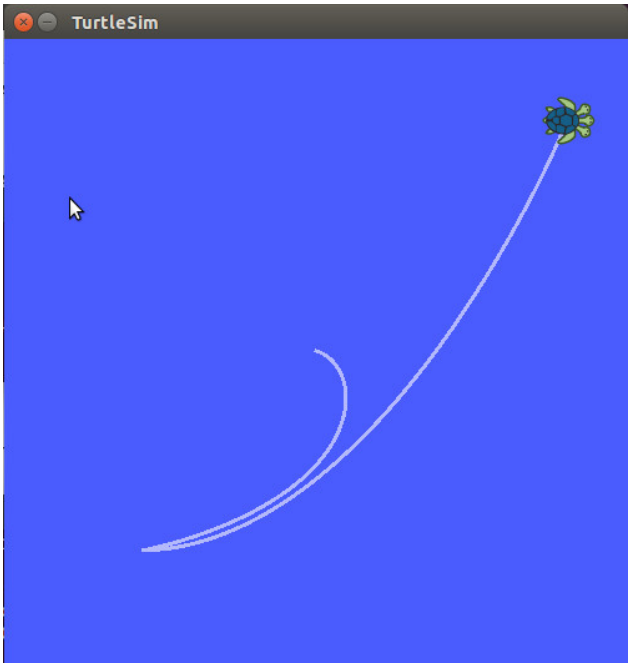


Figura 3. Ejecución del codigo final, muestra el movimiento de la tortuga a 2 puntos. 1) Coordenadas $(2, 2), \theta = 0$ y 2) $(10, 10), \theta = 0$.

## IV. Conclusion

Es muy ilustrativo ver como mediante el simulador *Turtlesim* y la guía introductoria *A Gentle Introduction To ROS*, podemos lograr desarrollar un primer programa que interactue con diferentes nodos, siga el modelo Publish-Subscribe y como podemos observar su comportamiento de manera gráfica. En el desarrollo no se encontraron grandes problemas para lograr poner ROS en marcha, una vez que se realizó con la versión correcta de Ubuntu, en cuanto a la compilación y ejecución del programa las herramientas de ROS no presentaron problemas y el compilador mostraba correctamente los errores de programación.

## Referencias

[1] Jason M O'Kane. A gentle introduction to ros, 2014.

[2] Wikipedia. Sistema operativo robótico — wikipedia, la enciclopedia libre, 2016. [Internet; descargado 29-agosto-2016].