

IIC2685 - Robótica Móvil I – 2022

Clase Práctica 1

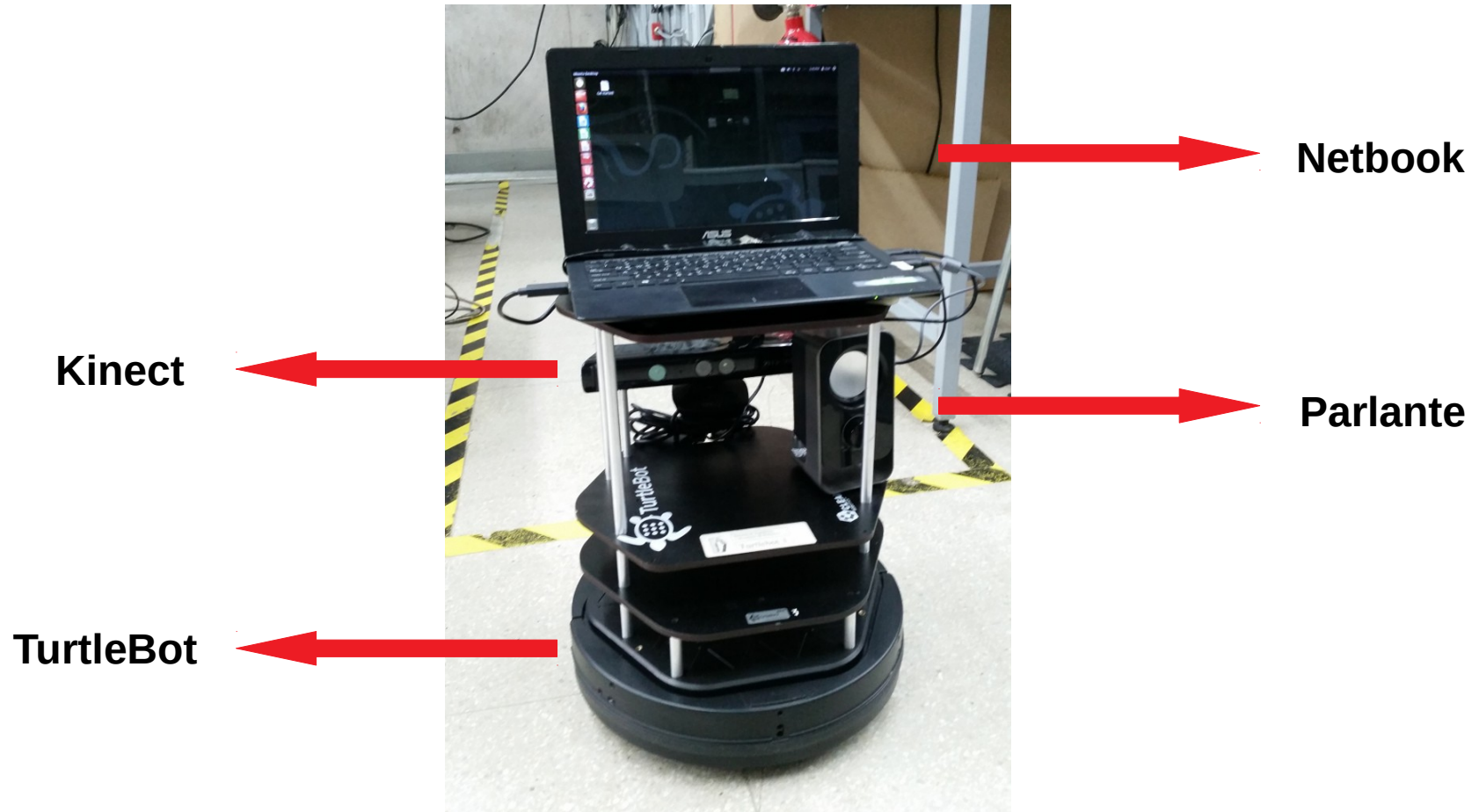
TurtleBot y ROS

Profesor: Gabriel Sepúlveda V.
grsepulveda@ing.puc.cl

TurtleBot



TurtleBot



TurtleBot

**Velcro de
Sujeción**



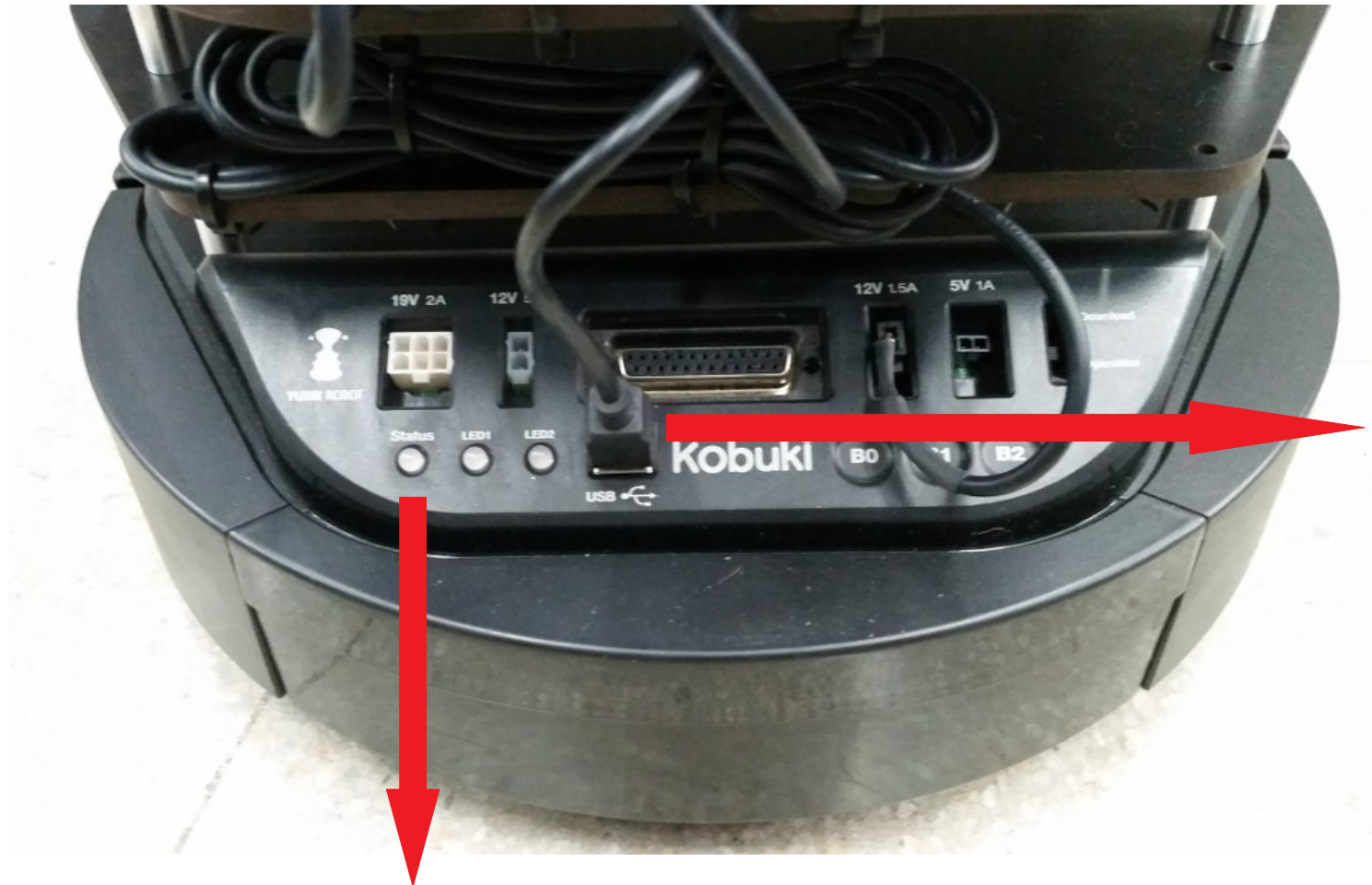
TurtleBot



**Switch
on/off**

Plug de alimentación

TurtleBot



**Conexión
USB a
netbook**

Status led

ROS

Robotic Operating System

ROS

- Sistema operativo y framework para el desarrollo de software de robótica.
- Creado por Erick Berger y Keenan WYROBEK
- Motivación: un desarrollador de software no tiene por qué ser especialista en hardware, así como un experto en planeamiento de rutas no tiene por qué ser experto en *computer vision*
- Idea base: “Something that didn’t suck, in all of those different dimensions”, Erick Berger




ROS

- Primer prototipo: *Switchyard*, mayo 2007.
- Proyecto se establece en *Willow Garage*: noviembre 2007.
- Abstrae al desarrollador de las complejidades de paso de mensajes, estructuras de datos, almacenamiento de datos, etc.
- Lenguajes soportados: C++ y Python.
- En este curso trabajaremos con la versión 1 de ROS
- Enlaces relevantes: <http://www.ros.org>, <http://wiki.ros.org>.



ROS

- ROS es versionado en distribuciones, las que contienen un conjunto de paquetes compatibles.

Distro	Release date	Poster	Tuturtle, turtle in tutorial	EOL date
ROS Noetic Ninjemys (Recommended)	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018			May, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)

ROS

ROS Jade Turtle

May 23rd, 2015



May, 2017

ROS Indigo Igloo

July 22nd, 2014



April, 2019
(Trusty EOL)

ROS Hydro Medusa

September 4th, 2013



May, 2015




ROS Groovy Galapagos

December 31, 2012



July, 2014

ROS

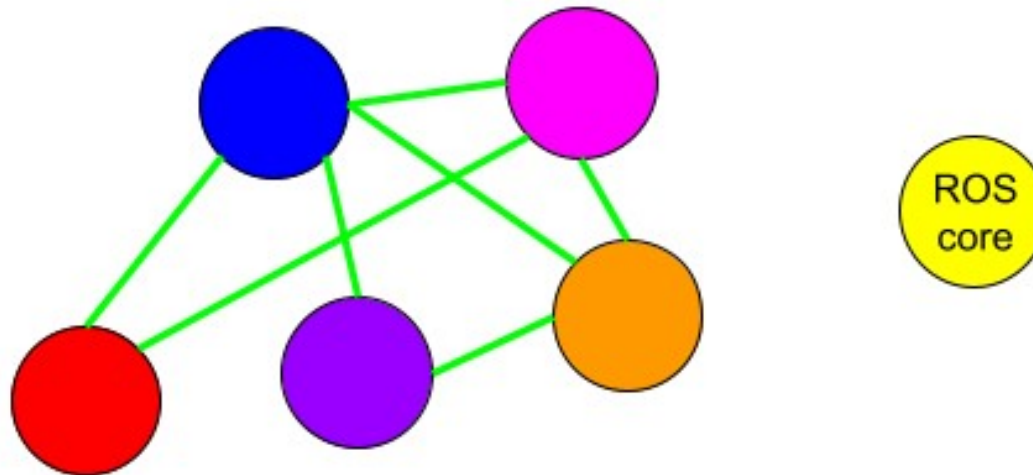
ROS Fuerte Turtle	April 23, 2012			--
ROS Electric Emys	August 30, 2011			--
ROS Diamondback	March 2, 2011			--
ROS C Turtle	August 2, 2010			--
ROS Box Turtle	March 2, 2010			--
		⌘ Box Turtle		

ROS

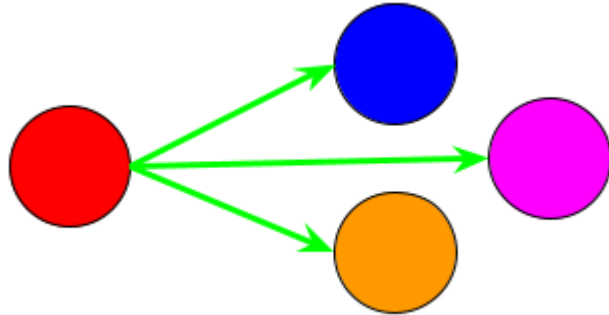
Conceptos Generales y API

Características Generales

- Sistema operativo basado en modelo de **grafos**
- Nodos: procesos / programa
- Aristas: envío y/o recepción de mensajes (*Topics, Client/Server*)
- Un nodo principal de coordinación: *ROS Core*



Características Generales

- Creación y destrucción de nodos
 - Múltiples **tipos de mensajes**: imágenes, estéreo, láser, control, actuador, contacto, etc.
 - Multiplexación de la información
- 
- Nodos pueden ser **distribuidos en múltiples CPUs** o núcleos para multiprocesamiento, o bien, en **múltiples máquinas** dentro de un cluster
 - En Python, las funciones base están contenidas en el módulo: *rospy*

ROS: Conceptos y API

Node

- Programa ejecutable que realiza una determinada tarea dentro del ambiente ROS.
- Declaración de un proceso como nodo ROS:

```
rospy.init_node( 'node_name', anonymous = True )
```

- 'node_name' debe ser **único** dentro del grafo, "anonymous = True" garantiza esta situación asignando un **sufijo irrepetible** al nombre
- Cada nodo puede comunicarse con otro utilizando alguno de los siguientes mecanismos:
 - Topics
 - Modelo Cliente/Servidor

Callbacks

- Función (o método) pasada como argumento a otra función, en donde la segunda se encargará de ejecutar a la primera

Callbacks

- Función (o método) pasada como argumento a otra función, en donde la segunda se encargará de ejecutar a la primera
- Permite abstraer la implementación de tareas específicas (ej: detección de rostro a partir de imagen) de tareas genéricas (ej: recepción de mensajes)

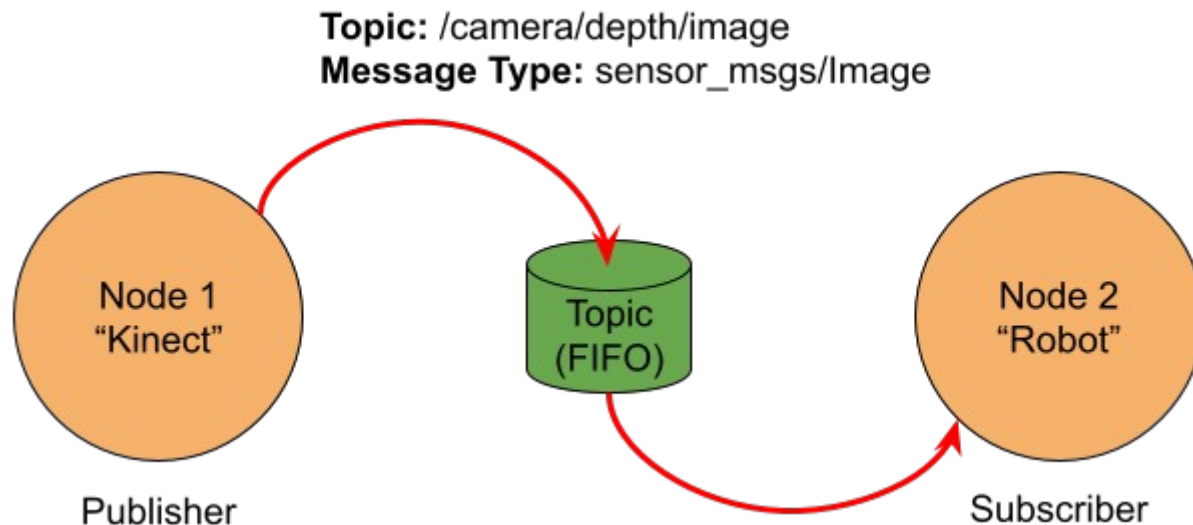
Callbacks

- Función (o método) pasada como argumento a otra función, en donde la segunda se encargará de ejecutar a la primera
- Permite abstraer la implementación de tareas específicas (ej: detección de rostro a partir de imagen) de tareas genéricas (ej: recepción de mensajes)
- Casos de uso:
 - Paso de mensajes
 - Expiración de timers
 - Máquinas de estado
 - Etc.

ROS: Conceptos y API

Topics

- **Canal de comunicación** entre nodos para el envío y recepción de mensajes
- Cada *topic* será identificado por un nombre (ej: “my_topic”)
- Cada nodo puede asumir el rol de *Publisher*, *Subscriber*, o ambos a la vez.



Topics

- Inicialización de publisher dentro de un nodo:

```
pub = rospy.Publisher( 'topic_name', <data_type>, queue_size = 10 )
```

- Inicialización de subscriber dentro de un nodo:

```
sub = rospy.Subscriber( 'topic_name', <data_type>, <callback> )
```

ROS: Conceptos y API

Tipos de mensajes base

- *std_msgs/Int32*
- *std_msgs/Float32*
- *nav_msgs/Odometry*
- *geometry_msgs/Twist*
- *sensor_msgs/Image*

...

- *Una lista completa puede ser obtenida a través del comando:*

\$ rosmmsg list

```
[geometry_msgs/Twist]:  
geometry_msgs/Vector3 linear  
  float64 x  
  float64 y  
  float64 z  
geometry_msgs/Vector3 angular  
  float64 x  
  float64 y  
  float64 z
```


Tipos de mensajes base

- ¿ Cómo importar estos mensajes en Python ?
 - `from std_msgs.msg import Int32`
 - `from std_msgs.msg import Float32`
 - `from nav_msgs.msg import Odometry`
 - `from geometry_msgs.msg import Twist`
 - `from sensor_msgs.msg import Image`

ROS

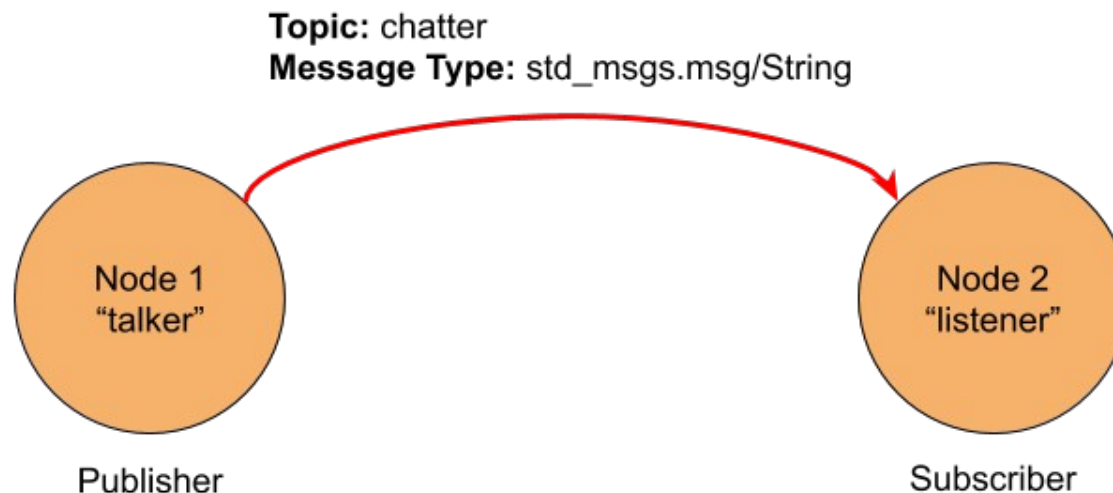
Ejemplo: 'Chatter'

Ref: <http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29>

ROS: Chatter

Chatter

- **Talker:** envía mensajes a una tasa de 0.1 [s], cuyo contenido es el string *"Hello World"* más un timestamp con la hora actual
- **Listener:** recibe mensajes provenientes del *talker* y los escribe en la salida standard



ROS: Chatter

Listener node

```
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("chatter", String, callback)
    rospy.spin()

if __name__ == '__main__':
    listener()
```

ROS: Chatter

Talker node

```
import rospy
from std_msgs.msg import String

def talker():
    rospy.init_node('talker', anonymous=True)
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()
```

Talker node

```
if __name__ == '__main__':  
    try:  
        talker()  
    except rospy.ROSInterruptException:  
        pass
```

Talker node, version 2: “wait for active subscriber”

- En ocasiones es deseable que la publicación de mensajes comience una vez que exista al menos un subscriptor activo

```
def talker():
```

```
    rospy.init_node('talker', anonymous=True)
```

```
    pub = rospy.Publisher('chatter', String, queue_size=10)
```

```
    while not rospy.is_shutdown() and pub.get_num_connections() == 0:
```

```
        pass
```

```
    rate = rospy.Rate(10) # 10hz
```

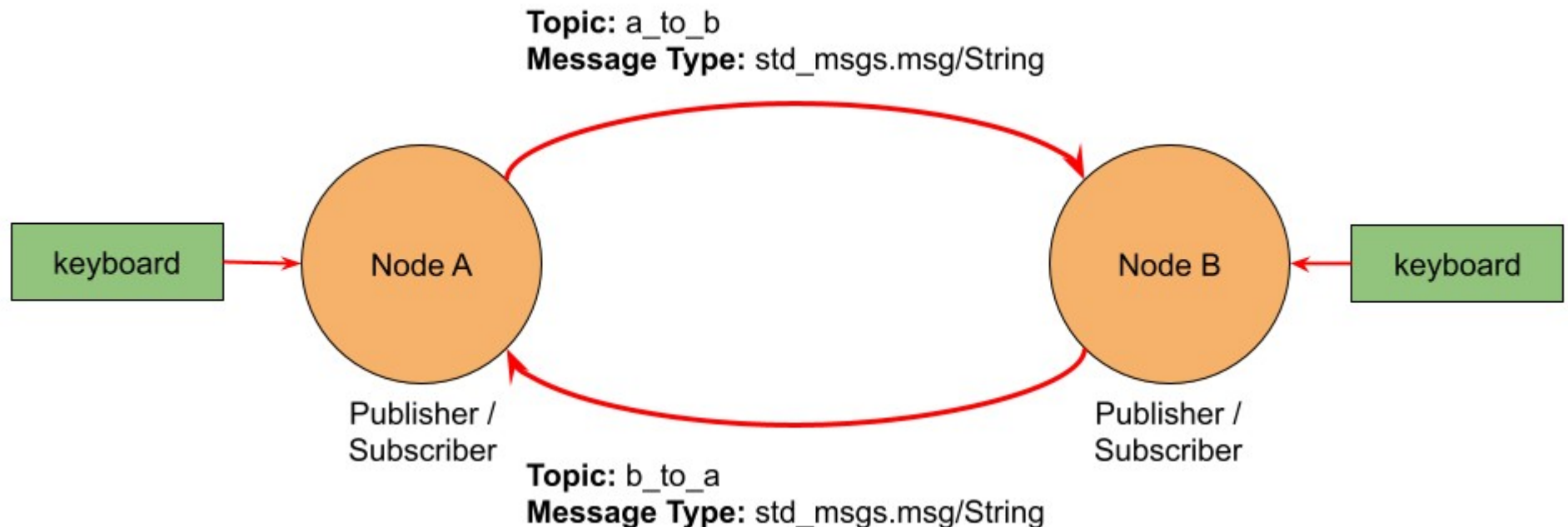
```
    while not rospy.is_shutdown():
```

```
        ...
```


ROS: Chatter

Mini-tarea !:

Construya una aplicación de chat en el que dos personas se puedan comunicar ingresando mensajes por la terminal (standard input), utilizando topics como canal de comunicación



ROS

Workspace

ROS: Workspace

- **Workspace:** directorio donde es posible construir, modificar e instalar paquetes.
- Organización de paquetes dentro del workspace

```
workspace_folder/      -- WORKSPACE
  src/                 -- SOURCE SPACE
    CMakeLists.txt     -- 'Toplevel' CMake file, provided by catkin
  package_1/
    CMakeLists.txt     -- CMakeLists.txt file for package_1
    package.xml        -- Package manifest for package_1
  ...
  package_n/
    CMakeLists.txt     -- CMakeLists.txt file for package_n
    package.xml        -- Package manifest for package_n
```

ROS: Workspace

¿ Cómo crear un Workspace ?

- Secuencia de comandos para creación de *Workspace*:
\$ mkdir -p ~/catkin_ws/src
\$ cd ~/catkin_ws/src
\$ catkin_init_workspace
- El nombre de directorio (“catkin_ws”) puede ser escogido a su elección
- Solo es necesario ejecutar estos pasos **una vez** por cada nuevo **Workspace**

ROS: Workspace

¿ Cómo crear un paquete dentro de un Workspace ?

- Secuencia de comandos para creación de paquete:

```
$ cd ~/catkin_ws/src
```

```
$ catkin_create_pkg <pkg_name> std_msgs rospy roscpp
```
- Esto creará el directorio “<pkg_name>” dentro de “~/catkin_ws/src”
- Se recomienda almacenar los códigos en Python dentro del directorio:
 - ~/catkin_ws/src/<pkg_name>/scripts/
- Solo es necesario ejecutar estos pasos **una vez** por cada nuevo **paquete**

ROS: Workspace

¿ Cómo construir/compilar paquetes una vez creados ?

- Commandos para compilación de paquetes dentro del workspace
\$ cd ~/catkin_ws
\$ catkin_make
- Si programamos en Python, solo es necesario ejecutar estos pasos **una vez** por cada nuevo **workspace**
- Si programamos en C o C++, debemos ejecutar estos comandos **por cada modificación de nuestro código** para poder compilar

ROS: Workspace

¿ Cómo hacer “visible” el contenido del Workspace para el ambiente ROS ?

- Agregar workspace al ambiente ROS

```
$ echo "source $HOME/catkin_ws/devel/setup.bash" >> ~/.bashrc
```

```
$ source ~/.bashrc
```

- Solo es necesario ejecutar estos pasos **una vez** por cada nuevo **Workspace**

ROS

Ejecución de Nodos

ROS: Ejecución de Nodos

Ejecución simple

- Inicialización de nodo maestro: **roscore**
- Ejecución de nodo:
`$ rosrun <pkg_name> <node_name>`

ROS: Ejecución de Nodos

Ejecución de múltiples nodos

- ¿ Cómo ejecutar más de un nodo a la vez usando *roslaunch* ?

```
$ roslaunch pkg1 node_1
```

```
$ roslaunch pkg1 node_2
```

```
...
```

```
$ roslaunch pkg1 node_N
```

- Existe la posibilidad de generar un archivo xml, que contenga un conjunto de nodos a ejecutar y sus parámetros: *launch file*
- Cada archivo *launch* puede llamar a otro archivo *launch* estableciendo una jerarquía
- Ejecución:

```
$ roslaunch <pkg_name> <launch_name>
```

ROS: Ejecución de Nodos

Ejecución múltiple a través de archivo *launch*

```
<launch>
```

```
<include file="$(find turtlebot_bringup)/launch/minimal.launch" />
```

```
<include file="$(find openni_launch)/launch/openni.launch" />
```

```
<include file="$(find sound_play)/soundplay_node.launch" />
```

```
<node pkg="pkg_name1" name="node_name1" type="node_name1.py" />
```

```
<node pkg="pkg_name1" name="node_name12" type="node_name1.py" />
```

```
<node pkg="pkg_name2" name="node_name2" type="node_name2.py" />
```

```
</launch>
```

ROS

Comandos de Administración

ROS: Comandos de Administración

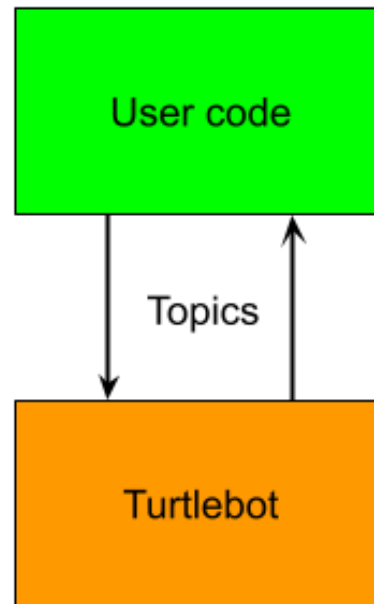
- rospack list
- roscd <pkg_name>
- rosls <pkg_name>
- rosmmsg <list|info|...>
- rosrn <pkg_name> <node_name>
- rosnnode list
- rosnnode info <node_name>
- rostopic list
- rosservices list
- roslaunch <pkg_name> <node_name>

ROS

Interacción con Turtlebot

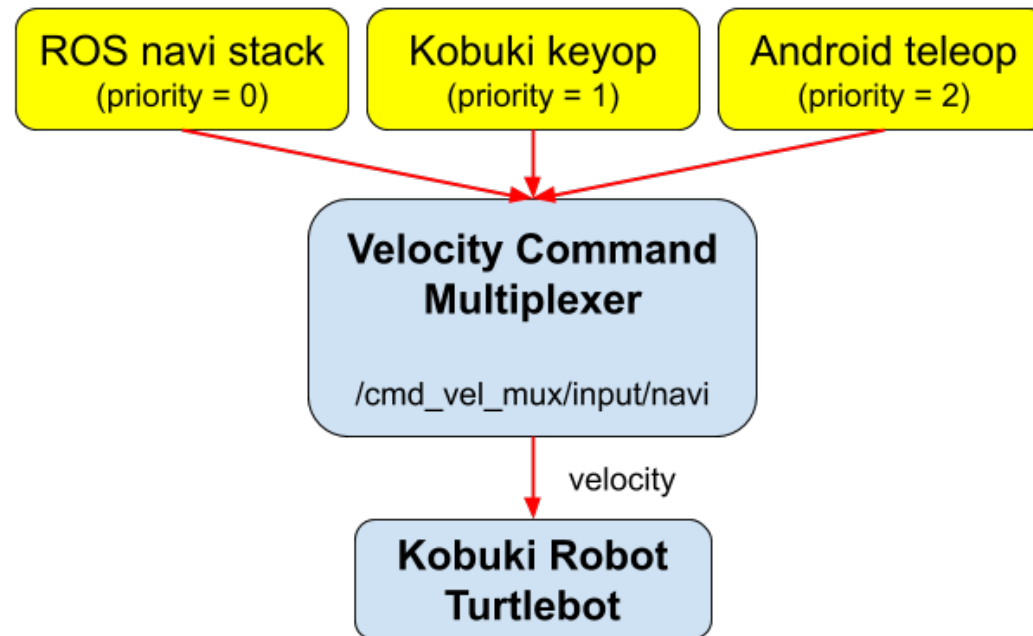
ROS: Interacción con Turtlebot

- La interacción con el TurtleBot será a través de tópicos, que permitirán
 - Enviar comandos/instrucciones
 - Recibir información desde los sensores



ROS: Interacción con Turtlebot

- Y entonces ... ¿ Cómo movemos el robot ?
- Utilizaremos el Sistema de Control *Kobuki*
- Este sistema permite mover el robot enviando comandos de **velocidad**
- Permite multiplexar comandos de velocidad asignando prioridad

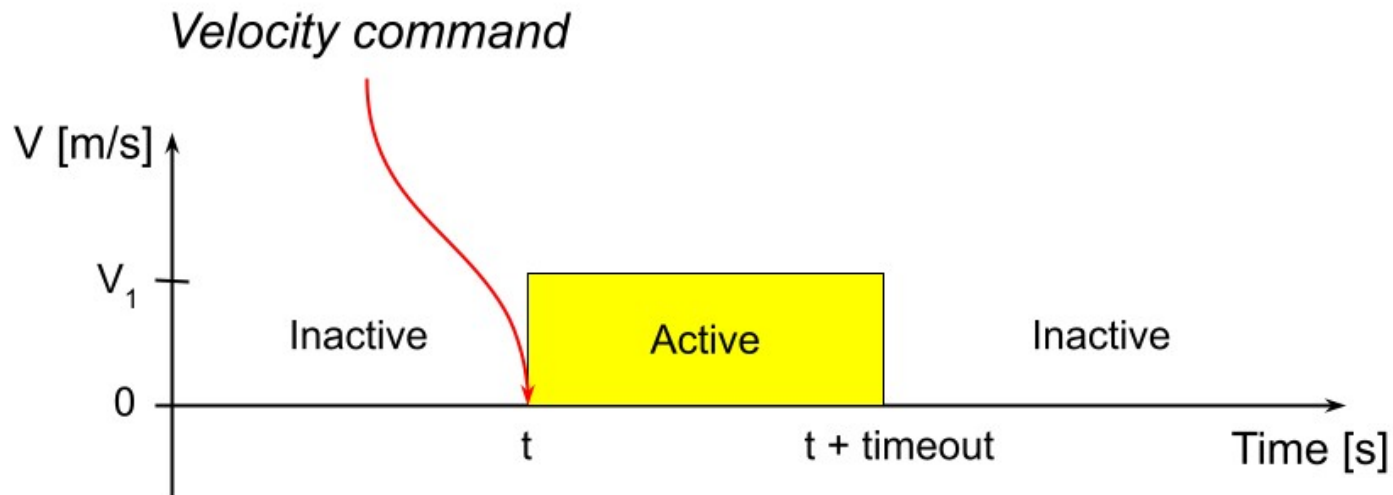


ROS: Interacción con Turtlebot

- Ejemplo de sistema de prioridad
 - 3 (highest priority): safety controller
 - 2: keyboard teleop
 - 1: android teleop
 - 0 (lowest priority): navi stack teleop

ROS: Interacción con Turtlebot

- Cada tópico de recepción definirá un tiempo máximo sin recepción mensajes (*timeout*), tras el cual se considerará dicho tópico como **inactivo**
- Dado lo anterior, nuestro nodo de navegación deberá enviar comandos de navegación con un período menor al *timeout* definido (0.1 [s])



ROS: Interacción con Turtlebot

- Para enviar instrucciones de velocidad utilizaremos el mensaje ROS: *Twist*
- Twist está compuesto de dos vectores tridimensionales que definen:
 - 3 coordenadas lineales: (V_x, V_y, V_z)
 - 3 coordenadas angulares: $(\omega_x, \omega_y, \omega_z)$
- \$ rosmmsg show Twist

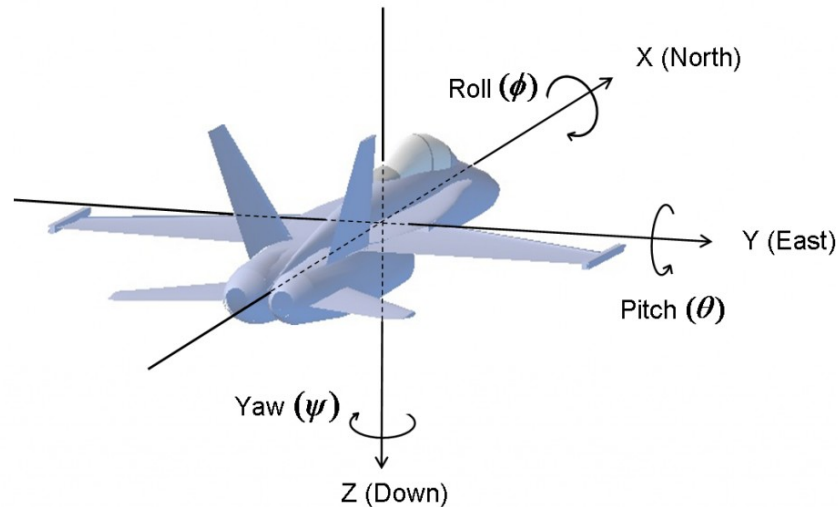
```
[geometry_msgs/Twist]:  
geometry_msgs/Vector3 linear  
  float64 x  
  float64 y  
  float64 z  
geometry_msgs/Vector3 angular  
  float64 x  
  float64 y  
  float64 z
```

ROS: Interacción con Turtlebot

- Para estimar la *pose* del robot utilizaremos la **odometría**
- El origen del sistema de coordenadas será la posición inicial del robot
- Podemos medir la odometría a través de la subscripción al tópico **/odom** y utilizando el mensaje: ***Odometry***
- *Odometry* está compuesto por dos campos:
 - Velocidad: *Twist*
 - Posicionamiento: *Pose*
- A su vez, el campo *Pose* contiene dos sub-campos:
 - Posición: *Point*
 - Orientación: *Quaternion*

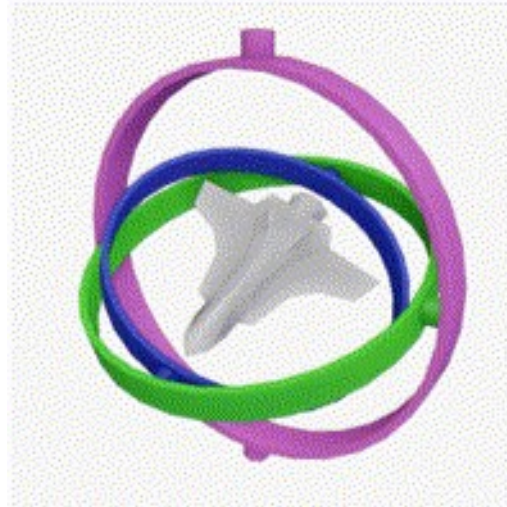
ROS: Interacción con Turtlebot

- Comúnmente se utilizan ángulos de Euler para medir la orientación:
 - Roll
 - Pitch
 - Yaw



ROS: Interacción con Turtlebot

- Sin embargo este sistema se ve afectado por un fenómeno conocido como *Gimbal Lock*, el cual hace imposible medir la orientación real cuando el *pitch* se acerca a los $\pm 90^\circ$



- **Solución:** Transformar sistema de Euler a un nuevo sistema de 4 dimensiones: *Quaternions*

ROS: Interacción con Turtlebot

- En ROS, esta transformación y su inversa pueden realizarse mediante las funciones del módulo *tf.transformations*:
 - `quaternion_from_euler`
 - `euler_from_quaternion`
- Finalmente, el campo *pose* toma la siguiente forma:

```
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
    float64[36] covariance
```


ROS: Interacción con Turtlebot

- Todo listo para comenzar a programar, pero ...
 - ¿ Qué es x ?
 - ¿ Qué es y ?
 - ...
 - ¿ Qué es ω_x ?
 - ¿ Qué es ω_y ?
 - ...
- REP 103: Unidades de medida estándar y convenciones de coordenadas
 - <https://www.ros.org/reps/rep-0103.html>

ROS: Haciendo Hablar al TurtleBot

- Nodo Soundplay
- Utiliza tecnología *Text To Speech* (TTS)
- Interacción con nodos clientes: biblioteca actionlib
- Instalación en Ubuntu

```
sudo apt install ros-$ROS_DISTRO-sound-play
```

- Launch file

```
<launch>
```

```
...
```

```
<include file="$(find sound_play)/soundplay_node.launch" />
```

```
...
```

```
</launch>
```

ROS: Haciendo Hablar al TurtleBot

```
import rospy
```

```
from sound_play.libsoundplay import SoundClient
```

- ```
if __name__ == '__main__':
 sound_handler = SoundClient(blocking = True)
 sound_handler.say(s, voice = 'voice_kal_diphone', volume = 1.0)
```

# Bibliografía

- <https://www.ros.org/>
- <http://wiki.ros.org/>
- <http://wiki.ros.org/ROS/Tutorials>
- ***ROS Robotics By Example***, Harman, Thomas L. and Fairchild, Carol