Introducción a ROS

Conceptos básicos

Impartida por: Mcs Edgar Macías García

Tabla de contenido

Tabla de contenido

Introducción a ROS

Conceptos básicos

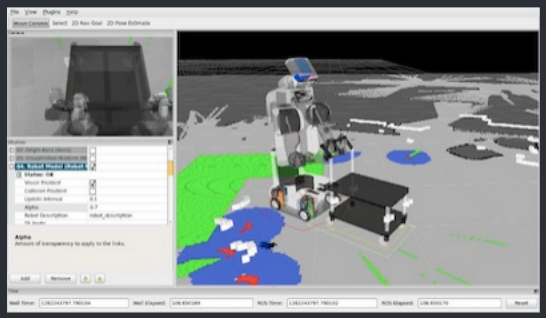
ROS

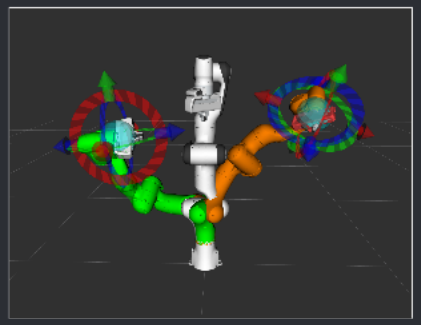
ROS (Robot Operating System) es un **framework** flexible que provee varios tipos de herramientas y librerias para el desarrollo de software aplicado en robótica. Entre sus principales ventajas se incluye:

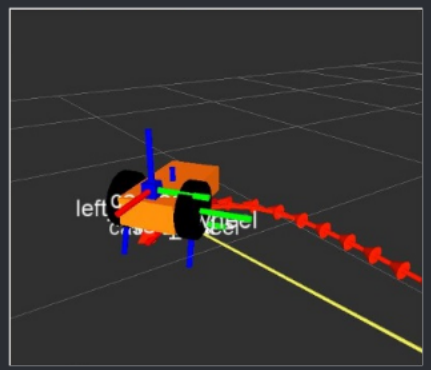
- **Multilenguaje**; compatibilidad con C, C++, Java y Python.
- **Libre y gratuito**; disponible en todas las versiones de Ubuntu.
- **Soporte**; miles de personas en todo el mundo utilizan y soportan el framework.

Introducción a ROS

Conceptos básicos







ROS framework

Introducción a ROS

Historia de desarrollo

Creación de ROS

- El proyecto ROS, comenzó en el 2007 bajo el nombre **Switchyard** por **Morgan Quigley** como parte del proyecto **Stanford Stair**.
- Desde 2013, el proyecto ha sido soportado por la **Open Source Robotics Foundation (OSR)**.
- En la actualidad, el framework es utilizado por diversas compañías y universidades en todo el mundo.

Introducción a ROS

Filosofía

Filosofía de ROS

- **De igual a igual:** Programas individuales pueden comunicarse entre sí a travéz del framework.
- **Distribuido:** Programas en diferentes equipos pueden conectarse a travéz de redes locales o externas.
- **Multilenguaje:** Compatibilidad con una gran variedad de lenguajes (C, C++, Python, MATLAB, Java).
- **Ligero:** Distribuido en miles de herramientas con funciones individuales.
- **Libre:** ROS es de código abierto, libre y de uso gratuito.

Introducción a ROS

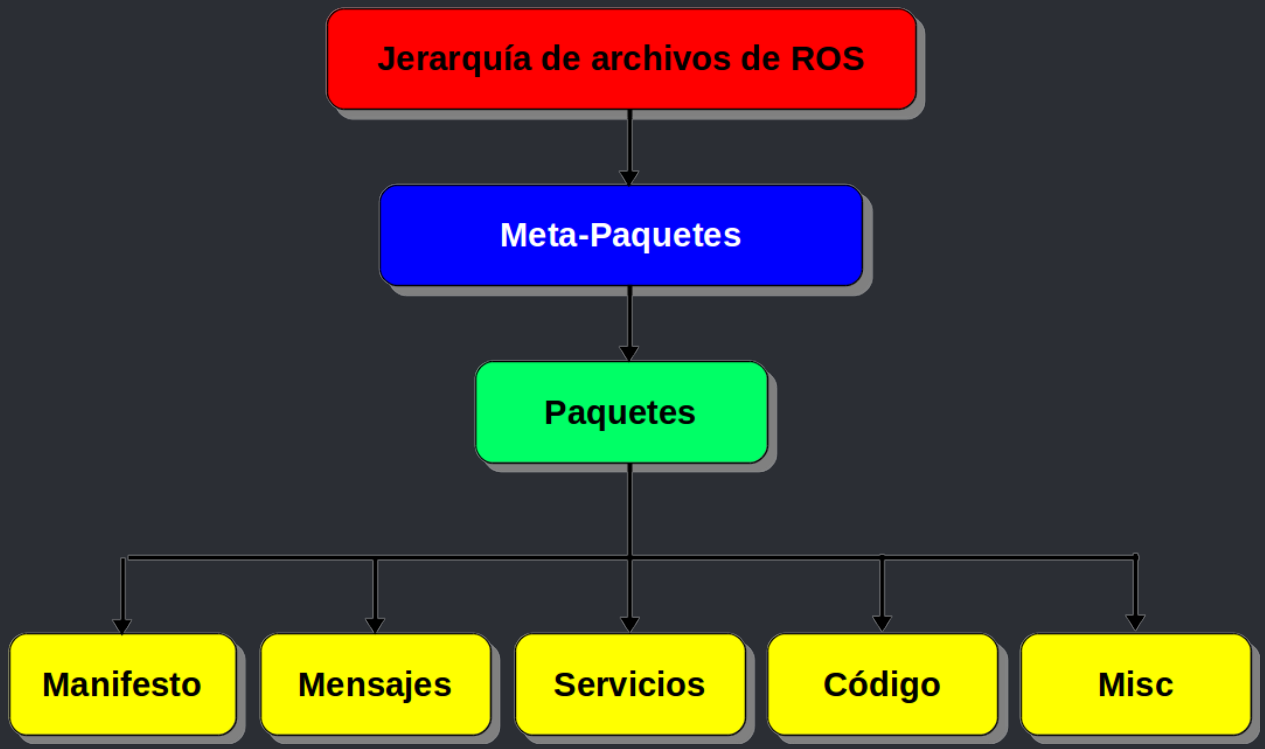
Filosofía

Ventajas de ROS

- **Implementaciones:** Gracias a la comunidad, ROS posee miles de algoritmos implementados para diferentes propósitos.
- **Herramientas:** ROS dispone de miles de herramientas para debug, visualización de mensajes y simulación.
- **Soporte de drivers:** ROS dispone de diferentes drivers genéricos y partículares para el habilitamiento de diferentes tipos de sensores y actuadores.
- **Comunidad:** Gracias a la comunidad, nuevos algoritmos y dispositivos son añadidos continuamente a diferentes repositorios oficiales.

Introducción a ROS

Filosofía



Estructura Jerárquica

Introducción a ROS

Filosofía

Jerarquía de paquetes

- **Paquetes:** Constituyen el bloque de construcción básico de ROS, permitiéndo englobar varios nodos en diferentes unidades.
- **Manifesto:** Es un archivo de configuración que se encuentra dentro de los paquetes, contiene información reelevante del mismo.
- **Metapaquetes:** Son similares a los paquetes, pero no contienen código fuente que se pueda editar.

Introducción a ROS

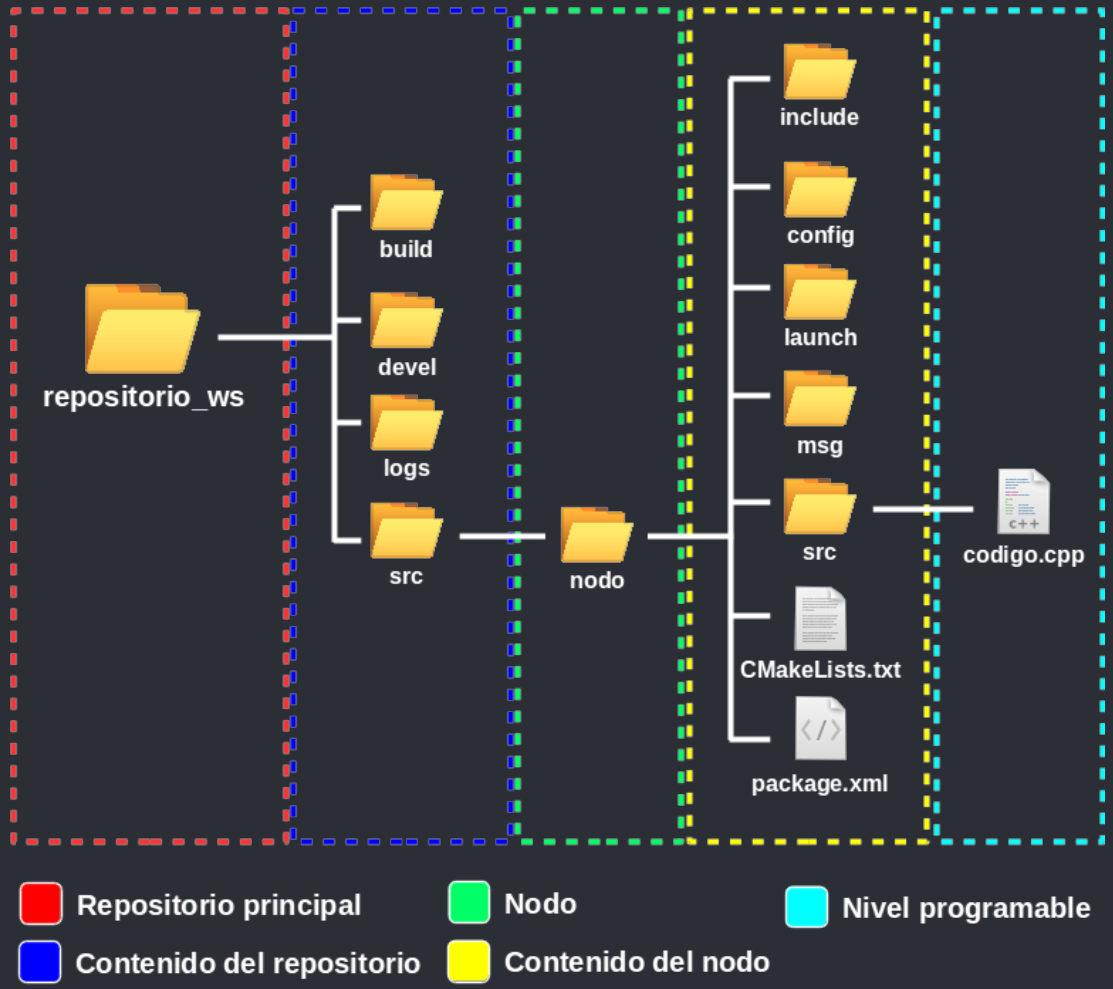
Filosofía

Jerarquía de elementos

- **Mensajes:** Se caracterizan por la extensión (.msg), constituyen el bloque básico de información que es compartida entre varios nodos.
- **Servicios:** Se caracterizan por la extensión (.srv), permiten establecer comunicación entre varios procesos.
- **Repositorios:** La mayoría de los paquetes de ROS estań soportados por un **gestor de versiones**, como GitHub.

Introducción a ROS

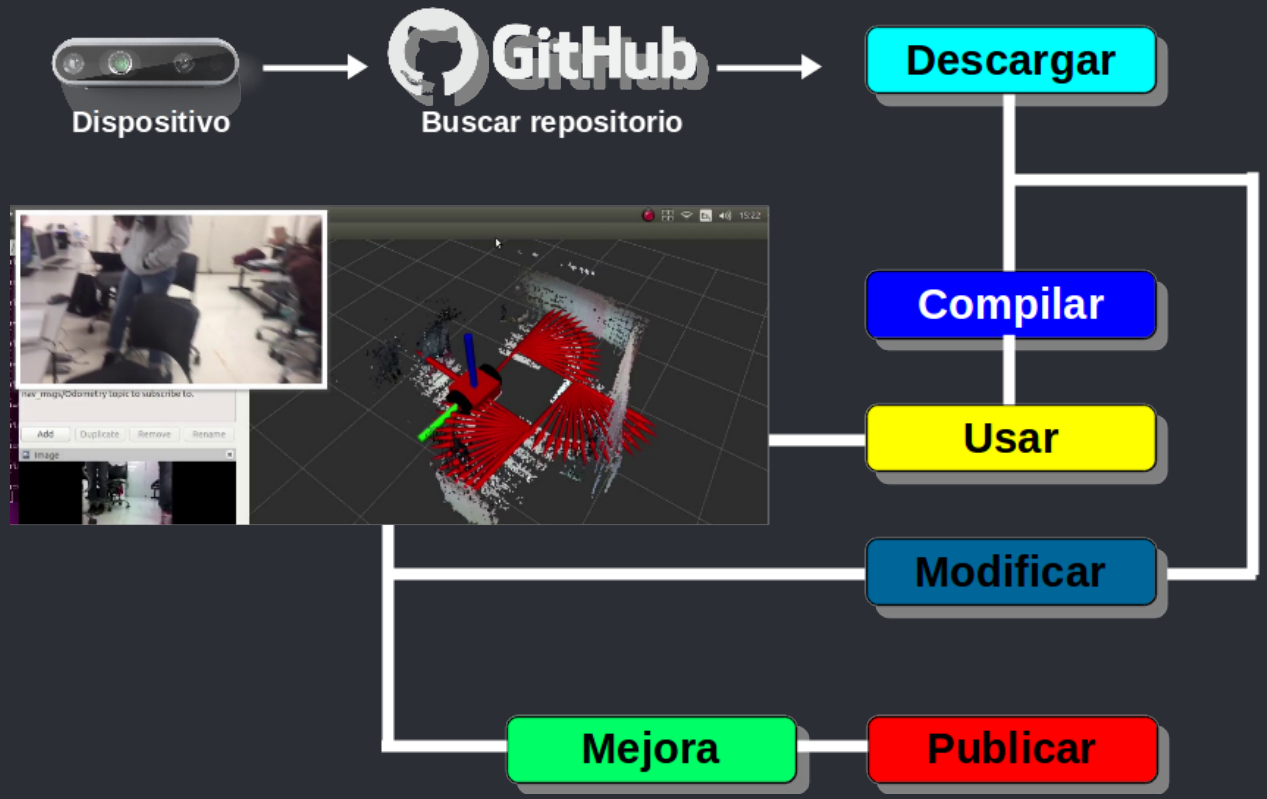
Filosofía



Estructura general de un repositorio.

Introducción a ROS

Filosofía



Proceso general de coolaboración.

Introducción a ROS

Instalación

Versión

ROS está disponible para diferentes versiones de la distribución Ubuntu y otros sistemas operativos:

- **Indigo:** Ubuntu 14.04 (Trustly)
- **Kinetic:** Ubuntu 16.04 (Xenial)
- **Lunar:** Ubuntu 17.04 (Zesty)
- **Melodic:** Ubuntu 18.04 (Bionic)
- **Noetic:** Ubuntu 20.04 (Focal)

Introducción a ROS

Instalación

Proceso de instalación (ROS Kinetic para Ubuntu 16.04)

1. Añadir permisos de descarga a repositorio de ROS.

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(
    lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.
    list'
```

2. Modificar llaves de acceso

```
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --
    recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

3. Actualizar lista de paquetes e instalar

```
sudo apt-get update
sudo apt-get install ros-kinetic-desktop-full
```

Introducción a ROS

Instalación

Proceso de instalación (ROS Kinetic para Ubuntu 16.04)

4. Iniciar servicio rosdep

```
sudo rosdep init
rosdep update
```

5. Configurar variables de arranque

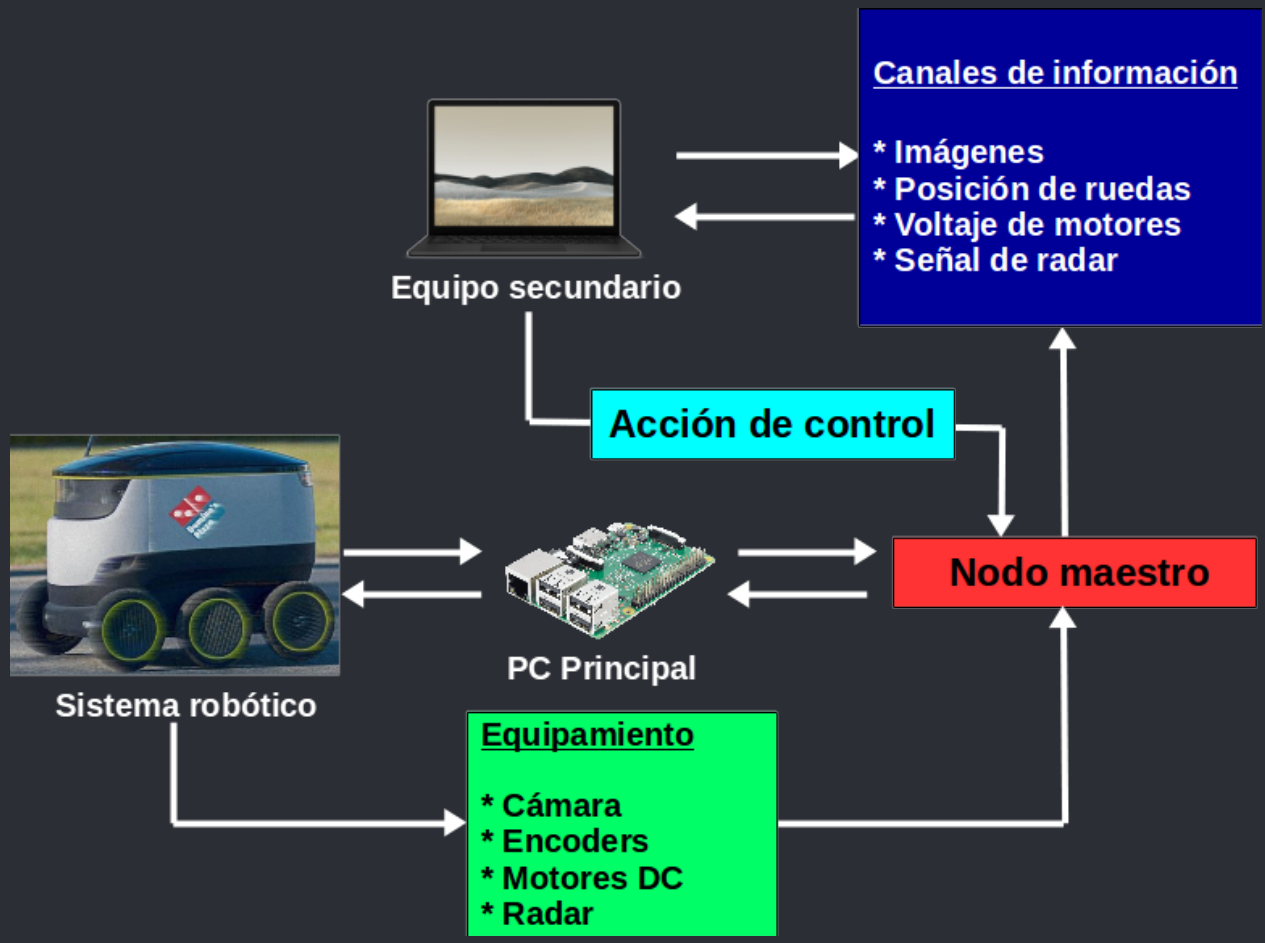
```
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

6. Probar servicio

```
roscore
```

Introducción a ROS

Instalación



Funcionamiento de ROS

Introducción a ROS

Instalación

Práctica 1: Instalación de ROS

Instalar framework ROS en equipo o máquina virtual con Ubuntu.

- El proceso varía en función del tipo de distribución de Ubuntu empleado.
- Para verificar que el proceso se realizó de forma correcta, el comando $roscore debe ejecutarse sin problemas.

Tabla de contenido

Funcionamiento de ROS

Conceptos básicos

Nodo maestro

El **nodo maestro** es el responsable de inicializar y administrar todos los servicios de ROS. A partir de su inicio, cualquier nodo (o herramienta) debe registrarse al maestro para comenzar sus operaciones.

Formas de iniciar un maestro

- **Inicio básico:** $roscore.
- **Inicio avanzado:** $roslaunch <nodo> <ejecutable>.

El inicio avanzado consiste en correr un nodo y el servicio roscore consecutivamente, adicionalmente permite el uso de un archivo de configuración.

Funcionamiento de ROS

Conceptos básicos

Nodos de procesamiento

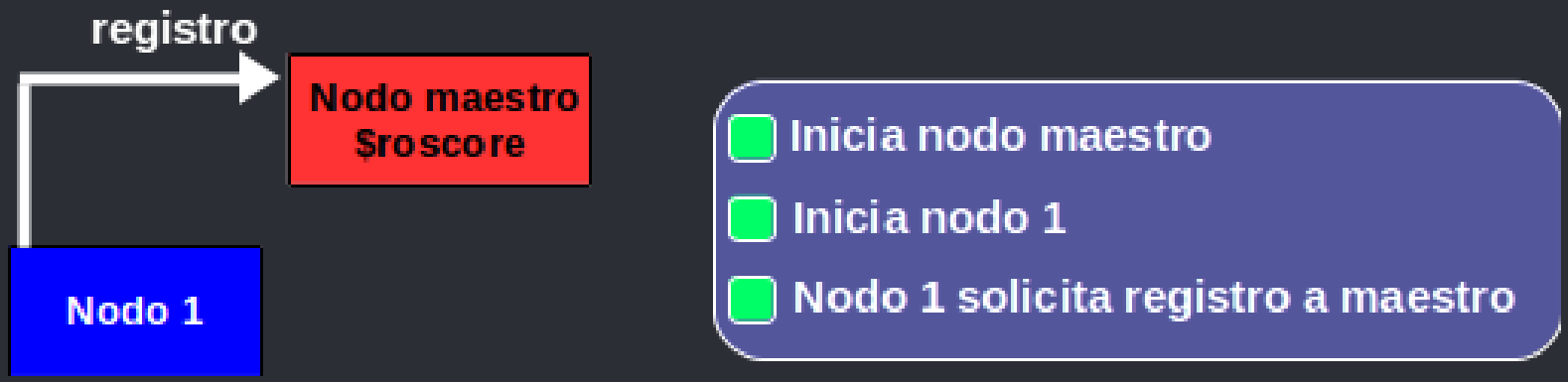
Un **nodo**, es un programa conectado a ROS que tiene la función de **recibir**, **procesar** o **publicar** información a travéz el nodo maestro. Estos se encuentran distribuidos en **paquetes**.

Formas de iniciar un nodo

- **Inicio básico:** $rosrun <paquete> <nodo>.
- **Inicio avanzado:** $roslaunch <paquete> <nodo>.

Funcionamiento de ROS

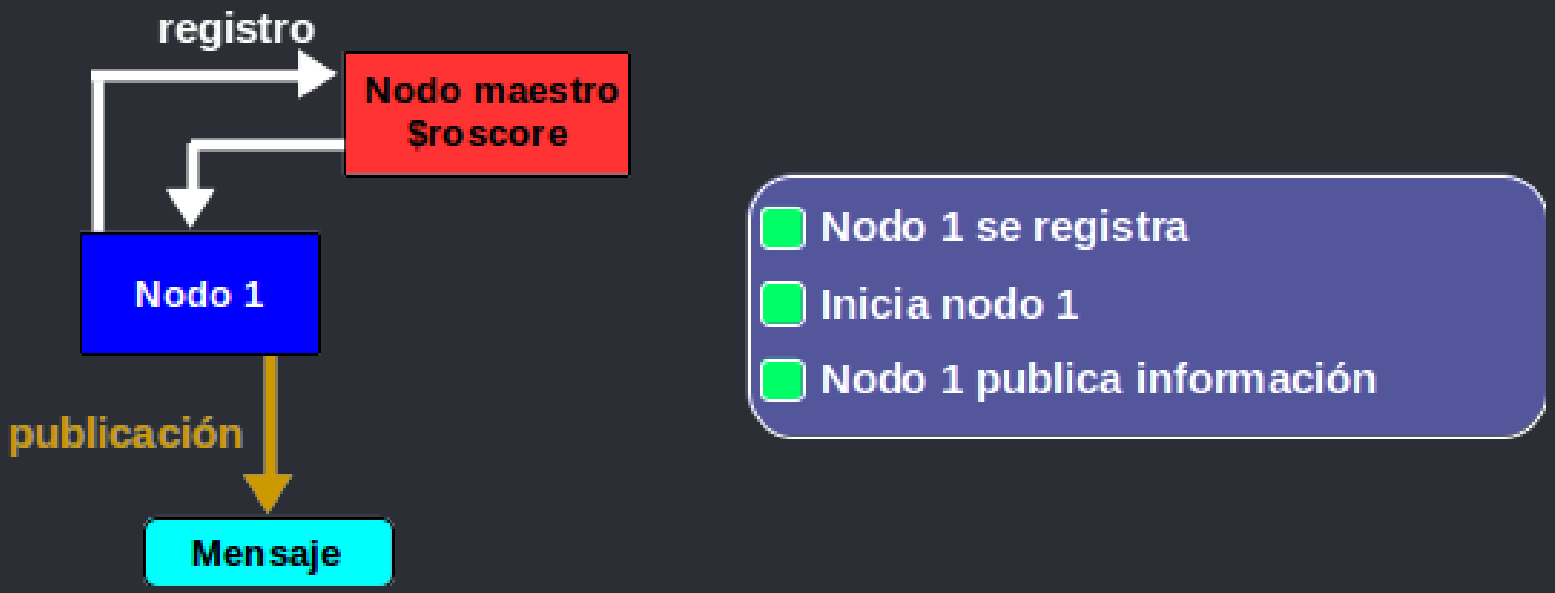
Conceptos básicos



Esquema de procesamiento de ROS

Funcionamiento de ROS

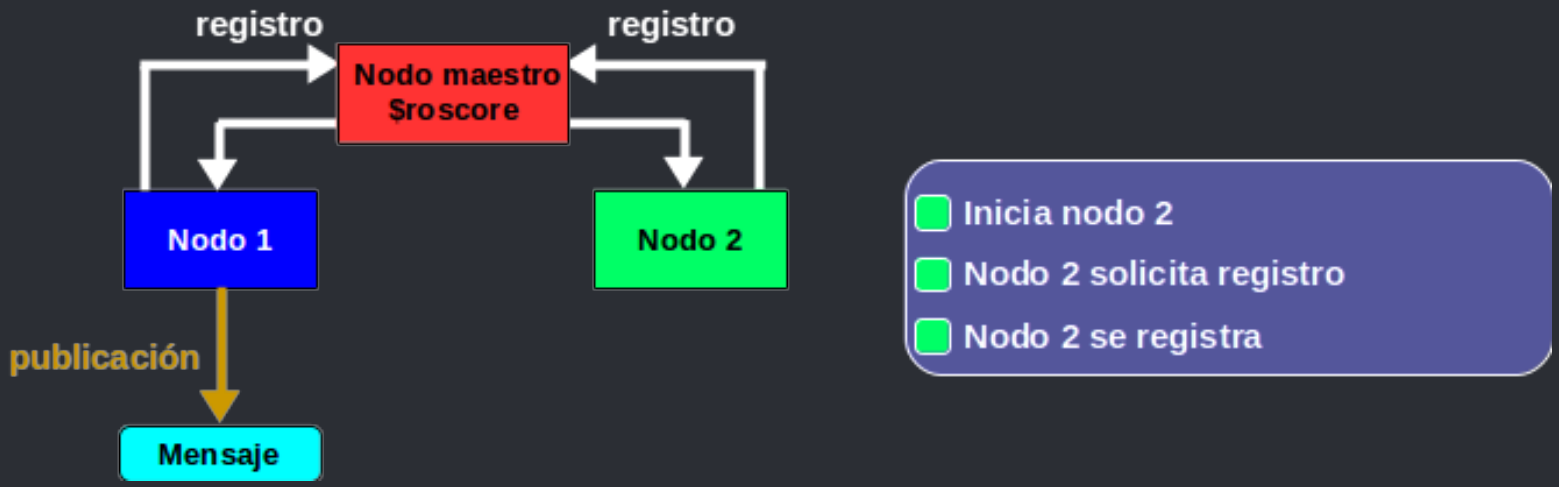
Conceptos básicos



Esquema de procesamiento de ROS

Funcionamiento de ROS

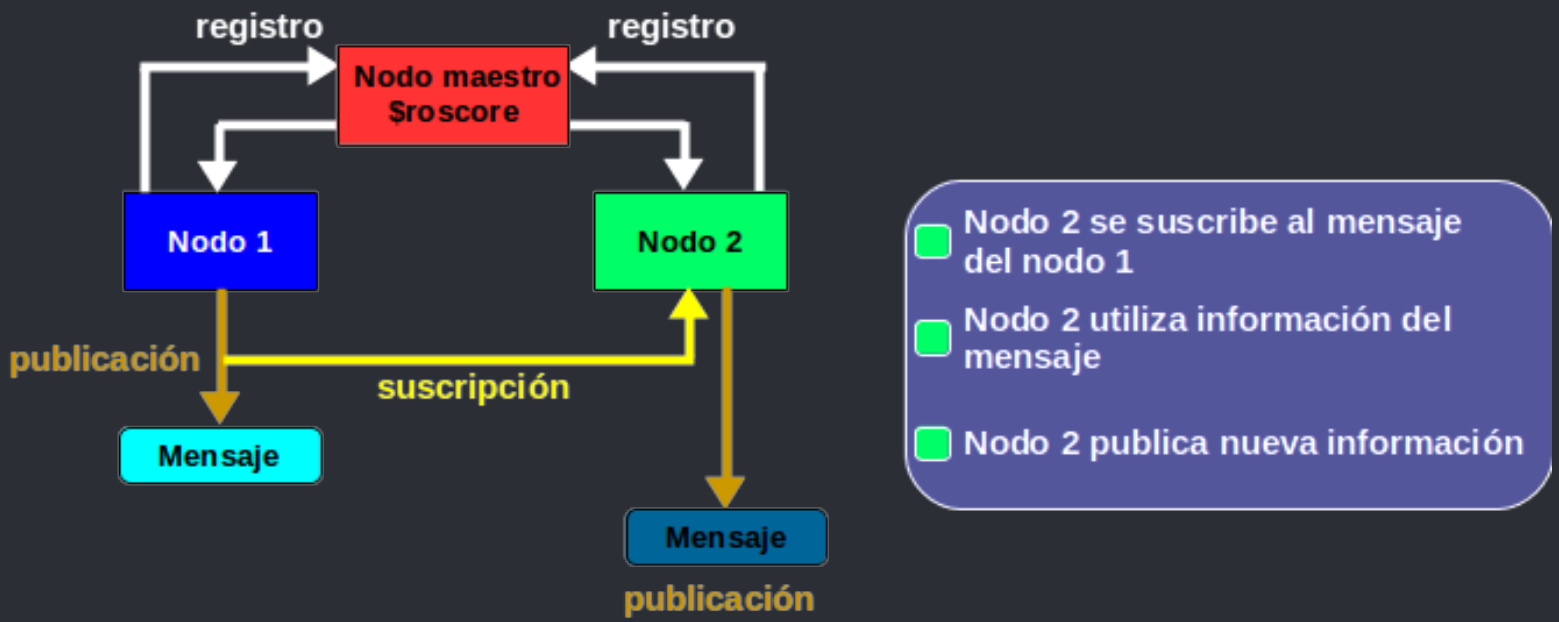
Conceptos básicos



Esquema de procesamiento de ROS

Funcionamiento de ROS

Conceptos básicos



Esquema de procesamiento de ROS

Funcionamiento de ROS

Nodos y paquetes

Estructura general de un paquete

Un **paquete** está compuesto de los directorios siguientes:

- **<paquete>/:** Directorio raíz del paquete, debe colocarse en la carpeta src/ del repositorio principal
- **nodos:** Archivos compilados formados a travéz de los paquetes.
- **src/:** Este directorio contiene el código fuente a compilar de cada nodo.
- **include/:** Incluye cabeceras o librerías dinámicas propias del nodo.

Funcionamiento de ROS

Nodos y paquetes

Estructura general de un paquete

- **launch/:** Incluye archivos de configuración (.launch) para inicio avanzado.
- **config/:** Archivos de configuración de arranque (.config, .yaml).
- **msg/:** Directorio de mensajes (.msg) propios del nodo.

Funcionamiento de ROS

Nodos y paquetes

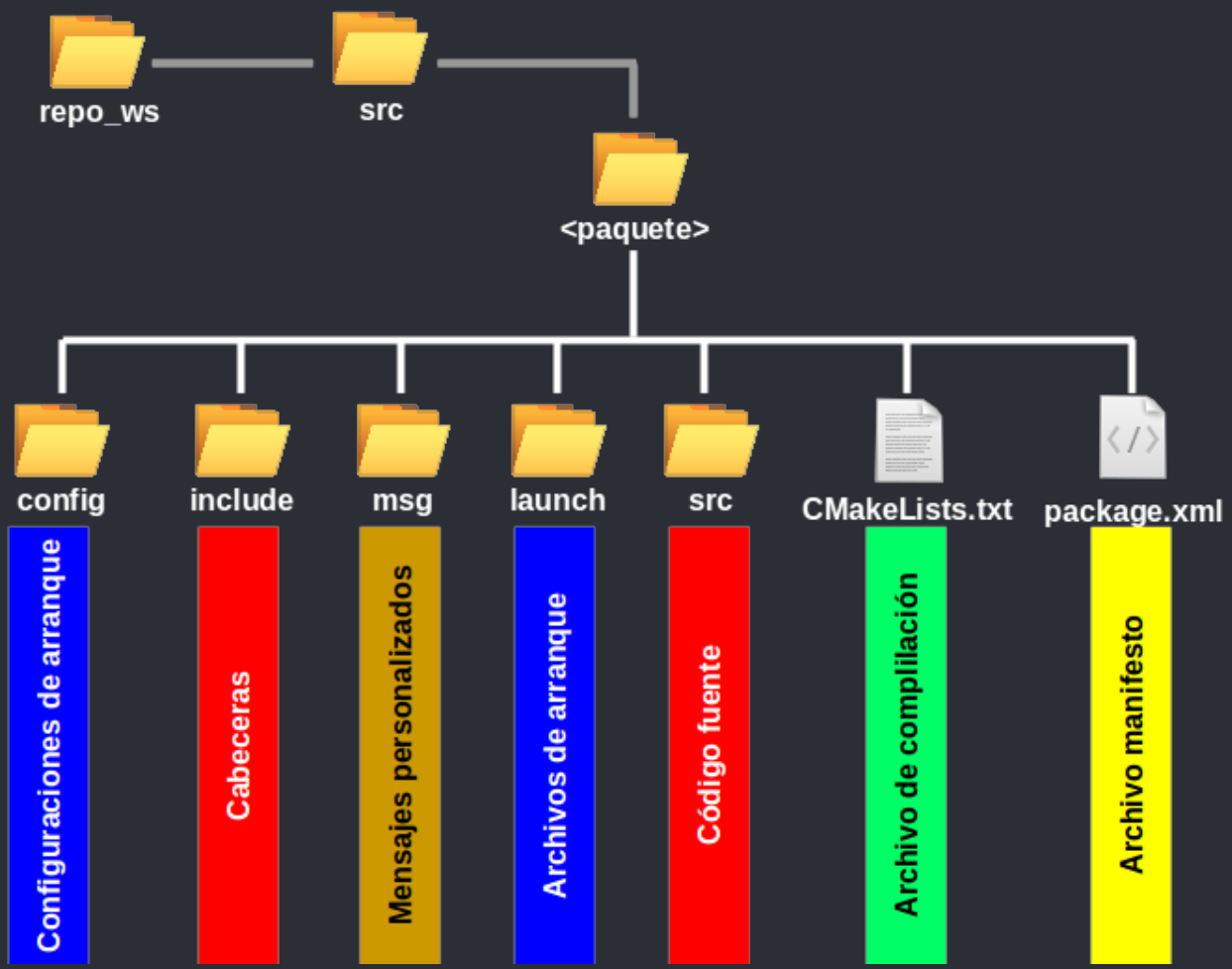
Estructura general de un nodo

Y los archivos siguientes:

- **CMakelists.txt:** Archivo de configuración para compilación por cmake.
- **package.xml:** Archivo manifesto, contiene información y requerimientos del nodo.

Funcionamiento de ROS

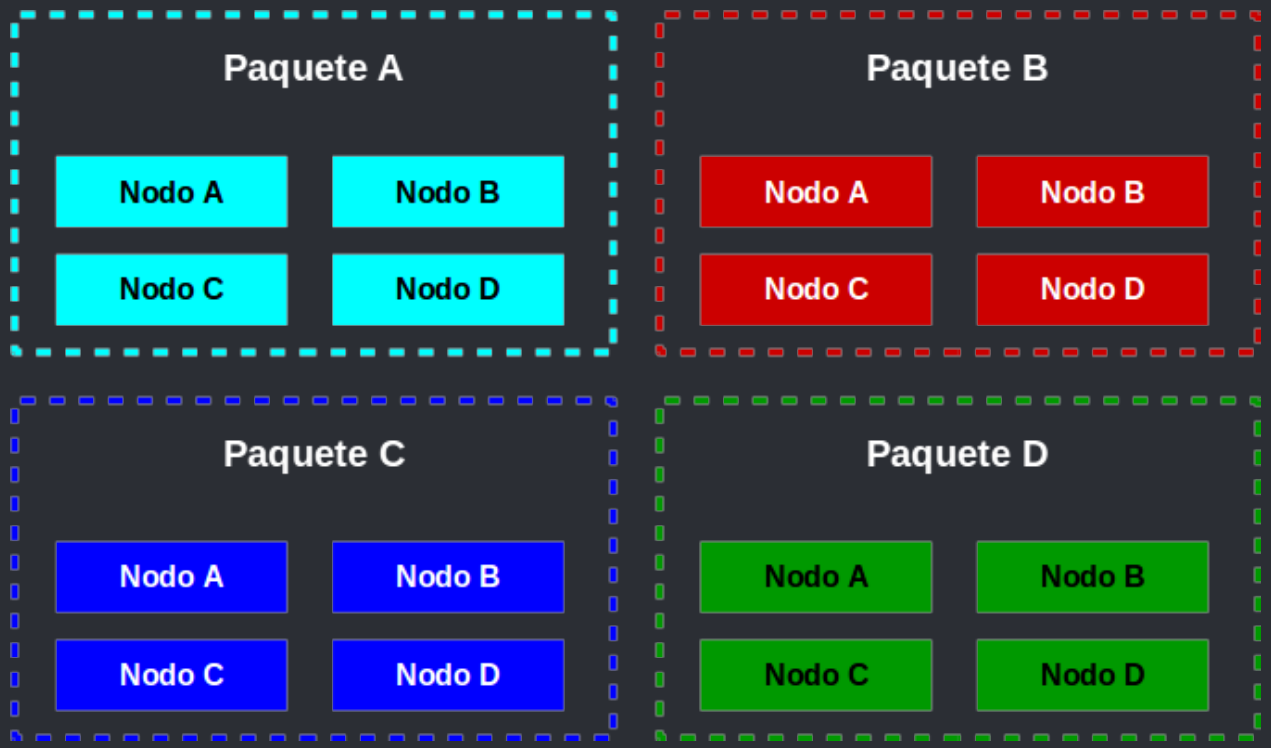
Nodos y paquetes



Estructura general de un nodo

Funcionamiento de ROS

Nodos y paquetes



Nodos y paquetes

Funcionamiento de ROS

Nodos y paquetes

Métodos de compilación

La herramienta catkin permite automatizar el proceso de compilación con ROS, actualmente se disponen de los siguientes métodos:

- **catkin_make:** Método de compilación default de ROS.
- **catkin build:** Método de compilación avanzado, añade mayores opciones de debug y permite una compilación más rápida.
- **catkin clean:** Limpiar compilados de repositorio.
- **catkin build <paquete>:** Compilar paquete en partícular.

Para instalar herramientas avanzadas de compilación:

```
sudo apt-get install python-catkin-tools
```

Funcionamiento de ROS

Nodos y paquetes

Archivo CMakeLists.txt

La utilidad catkin de ROS emplea la herramienta de CMake para compilar todos los nodos de un repositorio. Este archivo debe contener lo siguiente:

- Nombre del paquete.
- Dependencias
- De forma general, ubicación de cabeceras.
- De forma general, ubicación de librerías estáticas.
- Para cada nodo, ubicación de código fuente y dependencias.

Funcionamiento de ROS

Nodos y paquetes

Ejemplo de archivo CMakeLists.txt

```
#Version minima requerida de Cmake
cmake_minimum_required(VERSION 2.8.9)

#Nmbre del proyecto (o paquete)
project(proyecto)

#Librerias requeridas de ROS
find_package(catkin REQUIRED COMPONENTS
             roscpp
             rospy
             std_msgs
             )
```

Funcionamiento de ROS

Nodos y paquetes

Ejemplo de archivo CMakeLists.txt

```
#Librerias externas
find_package(OpenCV REQUIRED)

#Variables asignadas (librerias locales)
set(LIB_DIR "/path/libreria")

#Dependencias
catkin_package()

#Cabeceras de librerias
include_directories( include )
```

Funcionamiento de ROS

Nodos y paquetes

Ejemplo de archivo CMakeLists.txt

```
#Compilacion de nodos (c++)
add_executable(nodoA
    src/nodoA.cpp)

target_link_libraries(nodoA
    ${catkin_LIBRARIES} )

#Compilacion de nodos (python)
catkin_install_python
(
    PROGRAMS python/nodo.py
    DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

Funcionamiento de ROS

Nodos y paquetes

Archivo package.xml

El archivo **package.xml** es un archivo del tipo **manifesto** (.xml), cuya utilidad radica en brindar información general del paquete, así como de dependencias con otros paquetes. De formal general contiene:

- Nombre y versión del paquete.
- Mantenedor (nombre y correo).
- Tipo de licencia.
- Dependencias del paquete (principales).

Funcionamiento de ROS

Nodos y paquetes

Ejemplo de archivo package.xml

```
<?xml version="1.0"?>
<package>
  #Datos generales
  <name>Nombre del paquete</name>
  <version>0.0.0</version>
  <description>Descripcion del nodo</description>

  #Mantenedor
  <maintainer email="correo@gmail.com"> Autor </maintainer>

  #Tipo de Licencia
  <license>TODO</license>
```

Funcionamiento de ROS

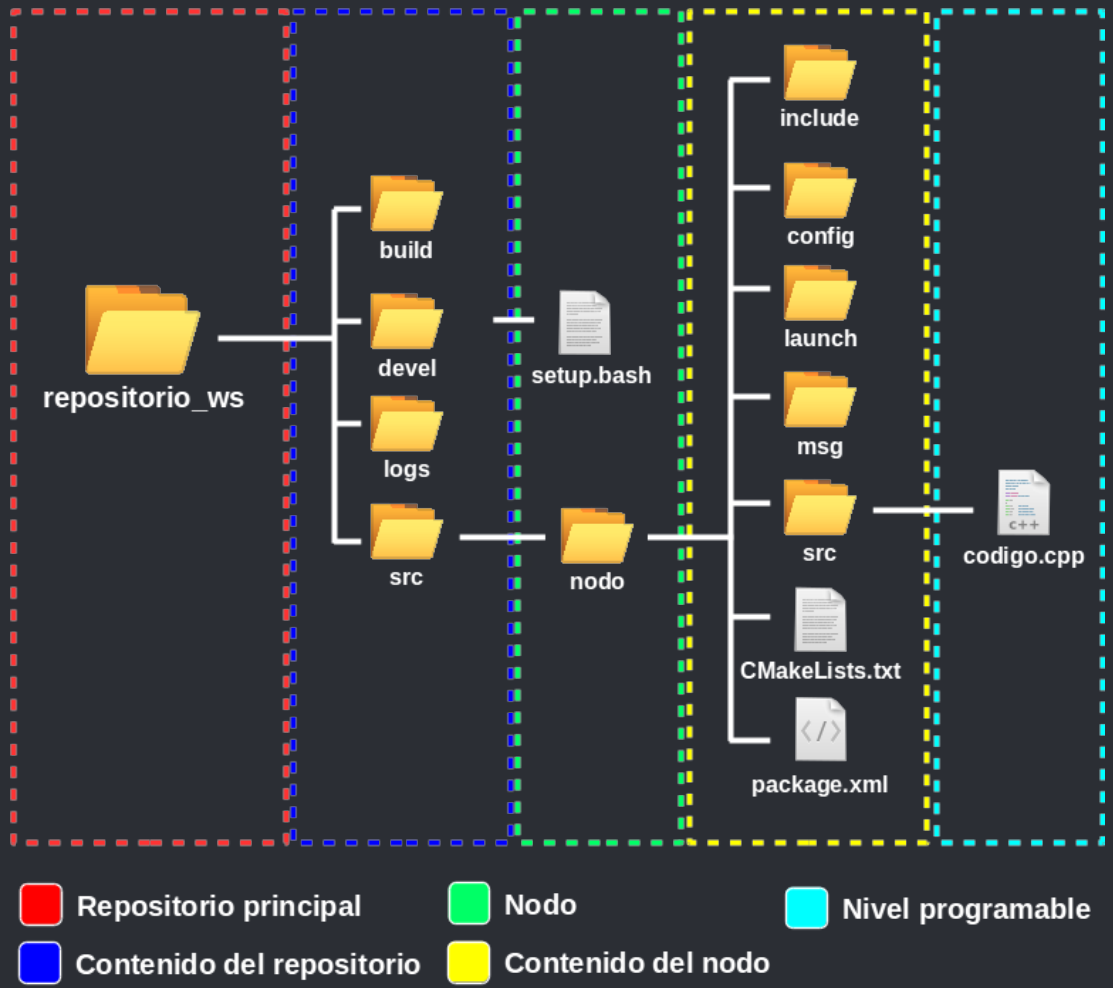
Nodos y paquetes

Ejemplo de archivo package.xml

```
  #Lista de dependencias
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>rospy</run_depend>
  <run_depend>std_msgs</run_depend>
</package>
```

Funcionamiento de ROS

Nodos y paquetes



Estructura general de un repositorio.

Tabla de contenido

Creación de nodos

Inicialización

Creando un nodo

ROS es compatible con una amplia variedad de lenguajes (C, C++, Java, Python, Matlab). De entre los objetos principales que maneja como librería se pueden distinguir los sigientes:

- **Nodehandle:** Es la representación de un nodo en ROS, y por ende es el primer objeto a crear en el programa.
- **Callbacks:** Funciones que permiten **recibir** información de un mensaje en partícular.
- **Publishers:** Objetos de ROS que permiten **publicar** información en forma de mensajes.

Creación de nodos

Inicialización

```
//Libreria base de ROS
#include <ros/ros.h>

int main( int argc, char** argv )
{
    //Inicializar nodo
    ros::init(argc, argv, "Nodo");

    //Iniciar nodehandle
    ros::NodeHandle n("~");

  return 0;
}
```

Creación de nodos

Inicialización

Inicialización de ros (C++)

1. **ros::init(argc, argv, <nombre>) :** Método de inicio de los servicios de ROS. Recibe 3 argumentos de entrada, dos del método **main()** y el nombre del nodo.
2. **ros::NodeHandle <var>(<path>) :** Creación del nodo de ROS, recibe como argumento la dirección base para carga de archivos de configuración.

Creación de nodos

Inicialización

ros::NodeHandle

Métodos principales

1. **param(<string>, <var>, <valor>) :** Leer variable de archivo de configuración.
2. **param<tipo>(<string>, <var>, <valor>) :** Leer arreglo de archivo de configuración.
3. **advertise<mensaje>(<nombre>, <frecuencia>) :** Avisar al nodo que se publicará un mensaje.
4. **subscribe(<nombre>, <frecuencia>, <callback>) :** Avisar al nodo que se suscribará a un mensaje por medio de la función **callback**.

Creación de nodos

Inicialización

Inicialización de un nodo en ROS

Creación de nodos

Inicialización

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

import rospy

#Metodo principal
def main():
  #Registrar nodo
  rospy.init_node('nodo')

  #Leer parametro
  param = rospy.get_param("interface/status", True)

#Llamada a metodo
if __name__ == '__main__':
  main()
```

Creación de nodos

Inicialización

Inicialización de ros (Python)

1. **rospy.init_node(<nombre>) :** Método de inicio de los servicios de ROS. Recibe como argumento el nombre del nodo.

En el caso de python, no es necesario crear un **NodeHandle**, ya que se encuentra implementado en forma de métodos estáticos sobre la clase **rospy**.

Creación de nodos

Inicialización

Objeto rospy

Métodos principales

1. **<var> = rospy.get_param(<string>, <valor>):** Leer variable de archivo de configuración.
2. **rospy.Publisher(<topico>, <msg>, queue_size=<freq>) :** Avisar al nodo que se publicará un mensaje.
3. **rospy.Subscriber(<topico>, <mensaje>, <callback>) :** Avisar al nodo que se suscribará a un mensaje por medio de la función **callback**.

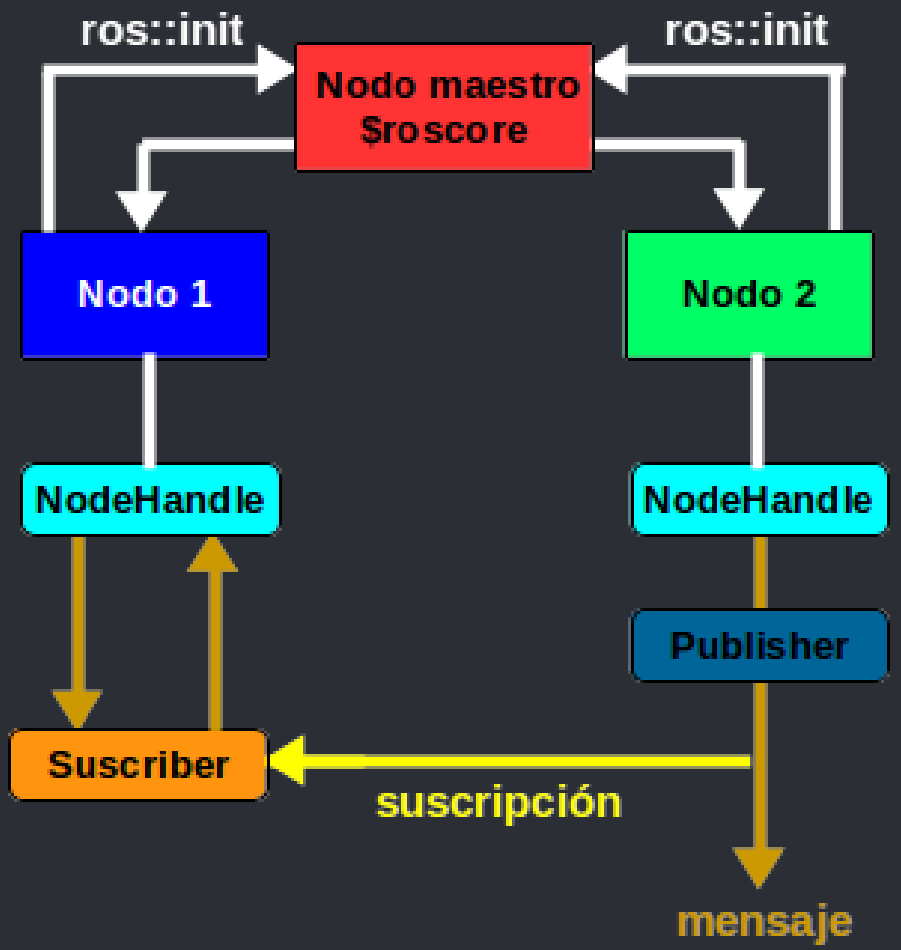
Creación de nodos

Inicialización

Inicialización de un nodo en ROS (python)

Creación de nodos

Inicialización



Función de los objetos de ROS

Creación de nodos

Inicialización

Creación de nodos en ROS

En resúmen, para crear un paquete en ROS es necesario efectuar los pasos siguientes:

1. Crear un repositorio.
2. Crear nodos y colocarlos en la carpeta **src/** del repositorio.
3. Compilar el repositorio usando **catkin_make** o **catkin build**.
4. Cargar configuración del repositorio:

```
source ./devel/setup.bash
```

5. Correr nodo deseado usando **rosrun** (arranque básico) o **roslaunch** (arranque avanzado).

Creación de nodos

Inicialización

Práctica 2: Compilando mi primer nodo

Crear y compilar un nodo básico

- Crear directorio de trabajo.
- Crear un nodo de funcionamiento básico (C++ o python)
- Configurar archivo CMakeLists.txt, indicando la configuración del nodo.
- Compilar y correr el programa con roscore.

Creación de nodos

Tipos de Mensajes

Mensajes en ROS

Los **mensajes** o **tópicos** son flujos de información que se comparten a travéz de **emisores** (publishers) y **receptores** (callbacks).

- Los mensajes pueden ser de diferentes tipos.
- Publishers y Callbacks deben especificar el **mismo tipo de mensaje** para poder compartir información.
- ROS dispone de cientos de diferentes tipos de mensajes, sin embargo existe la posibilidad de crear mensajes a la medida.
- Cada mensaje tiene diferentes tipos de **campos** en función de su uso.

Creación de nodos

Tipos de Mensajes

Tipos de mensajes básicos

- **std_msgs :** Mensajes primitivos básicos, incluyen flotantes, boleanos, cadenas de texto, vectores etc.
- **geometry_msgs :** Mensajes relacionados con representaciones espaciales de objetos, como pose, posicion, área en forma de poligonos etc.
- **sensor_msgs :** Mensajes relacionados con diferentes tipos de sensores, como cámaras, lásers, medidores de temperatura etc.
- **visualization_msgs :** Incluye diferentes tipos de mensajes con capacidad de visualización en 3D (con visualizadores como RViz).

Creación de nodos

Tipos de Mensajes

geometry_msgs/Pose Message

File: geometry_msgs/Pose.msg

Raw Message Definition

```
# A representation of pose in free space, composed of position and orientation.
Point position
Quaternion orientation
```

Compact Message Definition

```
geometry_msgs/Point position
geometry_msgs/Quaternion orientation
```

autogenerated on Sun, 09 Feb 2020 03:18:27

Documentación de mensaje "Pose".

Creación de nodos

Tipos de Mensajes

geometry_msgs/Point Message

File: geometry_msgs/Point.msg

Raw Message Definition

```
# This contains the position of a point in free space
float64 x
float64 y
float64 z
```

Compact Message Definition

```
float64 x
float64 y
float64 z
```

autogenerated on Sun, 09 Feb 2020 03:18:27

Documentación de mensaje "Position".

Creación de nodos

Tipos de Mensajes

Herramientas básicas

La utilidad **rostopic** provee de diferentes para la consulta y administración de los mensajes. Entre las principales podemos encontrar las siguientes:

- **rostopic list :** Imprimir lista de tópicos activos (mensajes publicándose).
- **rostopic echo <topico> :** Imprimir contenido de tópico.
- **rostopic hz <topico> :** Medir tasa de publicación de tópico (hz).
- **rostopic pub <topico> <tipo> <contenido> :** Publicar mensaje, se puede añadir bandera **-r** para publicar a cierta tasa.

Creación de nodos

Tipos de Mensajes

Práctica 3: Usando rostopic

Publicar un mensaje usando rostopic pub

1.- Iniciar nodo maestro

roscore

2.- Publicar un entero a 10 Hz

rostopic pub -r 10 /nodo/elemento std_msgs/Int64 2

3.- Mirar lista de tópicos

rostopic list

4.- Mirar contenido de topico

rostopic echo /nodo/elemento

rostopic hz /nodo/elemento

Creación de nodos

Publicando tópicos

Publicando información a nivel de código (C++)

1. Inicializar un objeto del tipo **Publisher**.
   ros::Publisher emisor;
2. Inicializar tópico de salida por medio de la función **advertise** del **NodeHandle** del nodo.
   emisor = n.advertisetipo (”*/nodo/valor*”, 1**);**
3. Construir objeto del mensaje y cubrir información.
   std_msgs::Int64 valor;
   valor.data = 2;
4. Usar función **publish** de **Publisher** para publicar el mensaje.
   emisor.publish(valor);

Creación de nodos

Publicando tópicos

```
int main( int argc, char** argv )
{
    //Inicializar nodo
    ros::init(argc, argv, "Nodo");

    //Iniciar nodehandle
    ros::NodeHandle n("~");

    //Inicializar publisher
    ros::Publisher emisor;

    //Construir publisher
    emisor = n.advertise<std_msgs::Int64>("nodo/valor", 1);
```

Creación de nodos

Publicando tópicos

```
    //Construir mensaje
    std_msgs::Int64 valor;
    valor.data = 2;

    //Fijar velocidad del nodo
    ros::Rate loop_rate(10);

    while (ros::ok())
    {
        //Publicar mensaje
        emisor.publish(valor);
        loop_rate.sleep();
    }
}
```

Creación de nodos

Publicando tópicos

Publicando información a nivel de código (Python)

1. Crear un objeto del tipo **Publisher**.
   emisor = rospy.Publisher("topico", Int64, queue_size=1)
2. Construir objeto del mensaje y cubrir información.
   msg = Int64()
   msg.data = 2;
3. Usar función **publish** del **Publisher** para publicar el mensaje.
   emisor.publish(msg);

Creación de nodos

Publicando tópicos

```
import rospy

#Importar librerias de mensajes
from std_msgs.msg import Int64

def main():
  #Registrar nodo
  rospy.init_node('nodo')

  #Crear publisher
  pub = rospy.Publisher('entero', Int64, queue_size=1)

  #Fijar tasa de publicacion
  rate = rospy.Rate(10) # 10hz
```

Creación de nodos

Publicando tópicos

```
    #Lazo principal
    while not rospy.is_shutdown():
      #Crear mensaje y llenar campos
      msg = Int64()
      msg.data = 2

      #Publicar mensaje
      pub.publish(msg)

      #Colocar retardo
      rate.sleep()

#Llamado a metodo
if __name__ == '__main__':
  main()
```

Creación de nodos

Publicando tópicos

Consideraciones

- Por lo general los **publishers** son variables globales.
- Se pueden crear tantos publishers como se desee.
- Al ser dependencias de ROS, es necesario incluir las librerias correspondientes en la cabecera del código, así como las dependencias en el archivo **CMakeLists.txt** y el **manifesto**.
- Por lo general, se fija una tasa máxima de procesamiento por medio del objeto **ros::Rate (c++) / rospy.Rate (python)**. A fin de controlar la tasa de publicación.

Creación de nodos

Publicando tópicos

Incluir cabeceras

```
//ROS
#include <ros/ros.h>
#include <std_msgs/Int64.h>
```

Añadir dependencias (Compilación)

```
## Librerias requeridas de catkin
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
)
```

Añadir dependencias (Manifesto)

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<run_depend>roscpp</run_depend>
<run_depend>rospy</run_depend>
<build_depend>std_msgs</build_depend>
<run_depend>std_msgs</run_depend>
```

Añadiendo dependencias de los mensajes.

Creación de nodos

Publicando tópicos

Práctica 4: Publicando un tópico

Crear un nodo que publique la información siguiente:

- Un mensaje del tipo **std_msgs::String** bajo el topico /nodo/-mensaje.
- Un mensaje del tipo **std_msgs::Bool** bajo el topico /nodo/estado.
- Un mensaje del tipo **geometry_msgs::Point** bajo el topico /nodo/punto.
- Un arreglo del tipo **std_msgs::UInt16MultiArray** bajo el topico /nodo/vector.

Nota: Recordar incluir librerías y dependencias correspondientes (de cada mensaje) en el código y archivo CMake.

Creación de nodos

Suscripción a tópicos

Proceso de suscripción a un tópico (C++)

1. Crear una función **callback**, cuyo argumento de entrada sea el tipo de mensaje a recibir.
   void IntCallback(const std_msgs::Int64 msg);
2. Crear objeto **Subscriber**
   ros::Subscriber int_subs;
3. Utilizar método **subscribe** del **NodeHandle** para relacionar **Subscriber** con la función **Callback**.
   int_subs = n.subscribe("topico", 1, IntCallback)
4. Usar función **ros::spin** sobre ciclo principal para llamar contínuamente al **callback**.
   ros::spinOnce();

Creación de nodos

Suscripción a tópicos

```
//ROS
#include <ros/ros.h>
#include <std_msgs/String.h>

//Callbacks
void StringCallback(const std_msgs::String& msg);

//Subscribers
ros::Subscriber r_str;

//Metodo principal
int main( int argc, char** argv )
{
    //Inicializar nodo
    ros::init(argc, argv, "subscriber");
```

Creación de nodos

Suscripción a tópicos

```
//Iniciar nodehandle
ros::NodeHandle n("~");

//Inicializar subscribers
r_str = n.subscribe("/nodo/mensaje", 1, StringCallback);

//Fijar tasa del nodo
ros::Rate loop_rate(10);

//Ciclo principal
while (ros::ok())
{
    ros::spinOnce();
    loop_rate.sleep();
}
```

Creación de nodos

Suscripción a tópicos

```
//Implementacion de callbacks
void StringCallback(const std_msgs::String& msg)
{
  std::cout << "Recibi String: " << msg.data << std::endl;
}
```

Creación de nodos

Suscripción a tópicos

Proceso de suscripción a un tópico (Python)

1. Crear una función **callback**.
   def callback(msg):
2. Crear objeto **Subscriber**
   rospy.Subscriber('mensaje', String, callback)
3. Usar función **spin** sobre ciclo principal para llamar contínuamente al **callback**.
   rospy.spin();

Creación de nodos

Suscripción a tópicos

```
import rospy
from std_msgs.msg import String

def callback(data):
  print("Recibo: " + data.data)

def main():

  # Registrar nodo
  rospy.init_node('listener')

  #Leer parametro
  param = rospy.get_param("/p_listener/ubicacion", 2)
```

Creación de nodos

Suscripción a tópicos

```
  #Crear subscriber
  rospy.Subscriber('chatter', String, callback)

  #Ciclo principal
  while not rospy.is_shutdown():
    #Llamar a los callbacks
    rospy.spin()

if __name__ == '__main__':
  #Llamar a metodo principal
  main()
```

Creación de nodos

Suscripción a tópicos

Consideraciones

- Al usarse únicamente en el método **main()**, los **subscribers** pueden ser variables locales.
- Las funciones de tipo **callback** pueden ser nombradas de cualquier forma, siempre que se usen correctamente.
- Al igual que los **publishers**, resulta necesario añadir dependencias al archivo **CMakeLists.txt** y el **manifesto**.
- A fin de verificar los **callbacks** durante cada ciclo del programa se emplea la función **spinOnce()**.
- Es una practica común colocar el procesamiento normal del programa dentro de un **callback**.

Creación de nodos

Suscripción a tópicos

Práctica 5: Subscribiendose a un tópico

Crear un nodo que se subscriba a los topicos generados en la práctica anterior:

- Un mensaje del tipo **std_msgs::String** bajo el topico /nodo/-mensaje.
- Un mensaje del tipo **std_msgs::Bool** bajo el topico /nodo/estado.
- Un mensaje del tipo **geometry_msgs::Point** bajo el topico /nodo/punto.
- Un arreglo del tipo **std_msgs::UInt16MultiArray** bajo el topico /nodo/vector.

Tabla de contenido

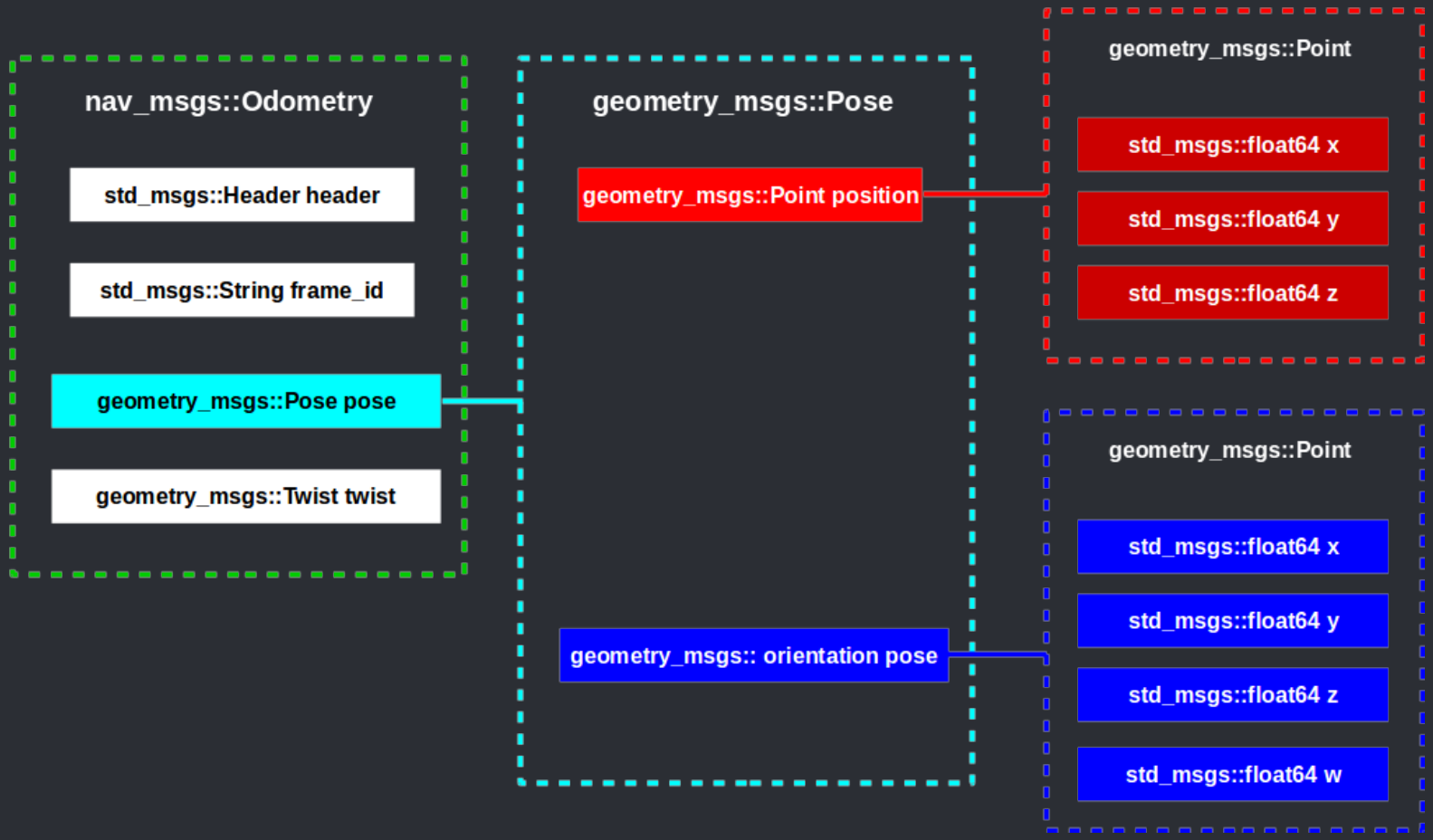
Mensajes Compuestos

Uso de mensajes avanzados

ROS dispone de cientos de diferentes tipos de mensajes especializados en diferentes tareas. Los **mensajes avanzados** son estructuras que se componen de otros mensajes más sencillos, a fin de brindar mayor información en un único tópico.

- En la práctica, los mensajes suelen ser **temporizados**, lo que significa que tienen información referente al tiempo en que fueron publicados.
- Son creados de la misma manera que los mensajes primitivos, únicamente requiriéndo más información en cada campo.
- Dependiendo del tipo de visualización que se requiera en el mensaje, la estructura del mensaje puede variar.

Mensajes Compuestos



Estructura de mensajes avanzados.

Mensajes Compuestos

Temporizados

Mensajes temporizados (c++)

Los mensajes temporizados, son estructuras que incluyen un **mensaje** del tipo **std_msgs::header**. El cual tiene un campo dedicado para almacenar la hora exacta de publicación del mensaje.

Elementos de header

- **seq :** ID asignado, cada vez que se publica el mensaje aumenta su valor.
- **stamp :** Guarda la información temporal (en segundos y milisegundos) en que publico el mensaje (**ros::Time::now()**).
- **frame_id :** Esta diseñado para propósitos de visualización, referentes a marcos de referencia.

Mensajes Compuestos

Temporizados

std_msgs/Header Message

File: std_msgs/Header.msg

Raw Message Definition

```
# Standard metadata for higher-level stamped data types.
# This is generally used to communicate timestamped data
# in a particular coordinate frame.
#
# sequence ID: consecutively increasing ID
uint32 seq
#Two-integer timestamp that is expressed as:
# * stamp.sec: seconds (stamp_secs) since epoch (in Python the variable is called 'secs')
# * stamp.nsec: nanoseconds since stamp_secs (in Python the variable is called 'nsecs')
# time-handling sugar is provided by the client library
time stamp
#Frame this data is associated with
string frame_id
```

Compact Message Definition

```
uint32 seq
time stamp
string frame_id
```

autogenerated on Thu, 13 Feb 2020 04:02:12

Header etiquetado con información temporal.

Mensajes Compuestos

Temporizados

```
//ROS
#include <ros/ros.h>
#include<geometry_msgs/PointStamped.h>

int main( int argc, char** argv )
{
    //Inicializar nodo
    ros::init(argc, argv, "Nodo");

    //Iniciar nodehandle
    ros::NodeHandle n("~");
```

Mensajes Compuestos

Temporizados

```
//Inicializar publisher y mensaje
ros::Publisher emisor_point_st;
geometry_msgs::PointStamped point_st;

//Inicializar posicion
point_st.point.x = 10.0;
point_st.point.y = 0.0;
point_st.point.z = -10.0;
```

Mensajes Compuestos

Temporizados

```
//Fijar tasa del nodo
ros::Rate loop_rate(10);

while (ros::ok())
{
  //Construir mensaje
  point_st.header.stamp = ros::Time::now();
  point_st.header.frame_id = "world";
  point_st.point.x = point_st.point.x + 1.0;
  point_st.point.y = point_st.point.y + 1.0;
  point_st.point.z = point_st.point.z + 1.0;
  emisor_point_st.publish(point_st);

  loop_rate.sleep();
}
```

Mensajes Compuestos

Temporizados

Mensajes temporizados (python)

Los mensajes temporizados, son estructuras que incluyen un **mensaje** del tipo **std_msgs.msg.header**. El cual tiene un campo dedicado para almacenar la hora exacta de publicación del mensaje.

Elementos de header

- **seq :** ID asignado, cada vez que se publica el mensaje aumenta su valor.
- **stamp :** Guarda la información temporal (en segundos y milisegundos) en que publico el mensaje (**rospy.get_rostime()**).
- **frame_id :** Esta diseñado para propósitos de visualización, referentes a marcos de referencia.

Mensajes Compuestos

Temporizados

std_msgs/Header Message

File: std_msgs/Header.msg

Raw Message Definition

```
# Standard metadata for higher-level stamped data types.
# This is generally used to communicate timestamped data
# in a particular coordinate frame.
#
# sequence ID: consecutively increasing ID
uint32 seq
#Two-integer timestamp that is expressed as:
# * stamp.sec: seconds (stamp_secs) since epoch (in Python the variable is called 'secs')
# * stamp.nsec: nanoseconds since stamp_secs (in Python the variable is called 'nsecs')
# time-handling sugar is provided by the client library
time stamp
#Frame this data is associated with
string frame_id
```

Compact Message Definition

```
uint32 seq
time stamp
string frame_id
```

autogenerated on Thu, 13 Feb 2020 04:02:12

Header etiquetado con información temporal.

Mensajes Compuestos

Temporizados

```
import rospy
from geometry_msgs.msg import PointStamped

def main():
  #Registrar nodo
  rospy.init_node('talker')

  #Crear publisher
  pub = rospy.Publisher('punto_estampado', PointStamped,
      queue_size=1)

  #Fijar tasa de publicacion
  rate = rospy.Rate(10) # 10hz
```

Mensajes Compuestos

Temporizados

```
#Lazo principal
while not rospy.is_shutdown():
  #Crear mensaje
  msg = PointStamped()
  msg.header.stamp = rospy.get_rostime()

  #Lllenar datos
  msg.point.x = 1.0
  msg.point.y = 0.0
  msg.point.z = 3.0
```

Mensajes Compuestos

Temporizados

```
    #Publicar mensaje
    pub.publish(msg)
    rate.sleep()

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

Mensajes Compuestos

Temporizados

Herramienta rqt_plot

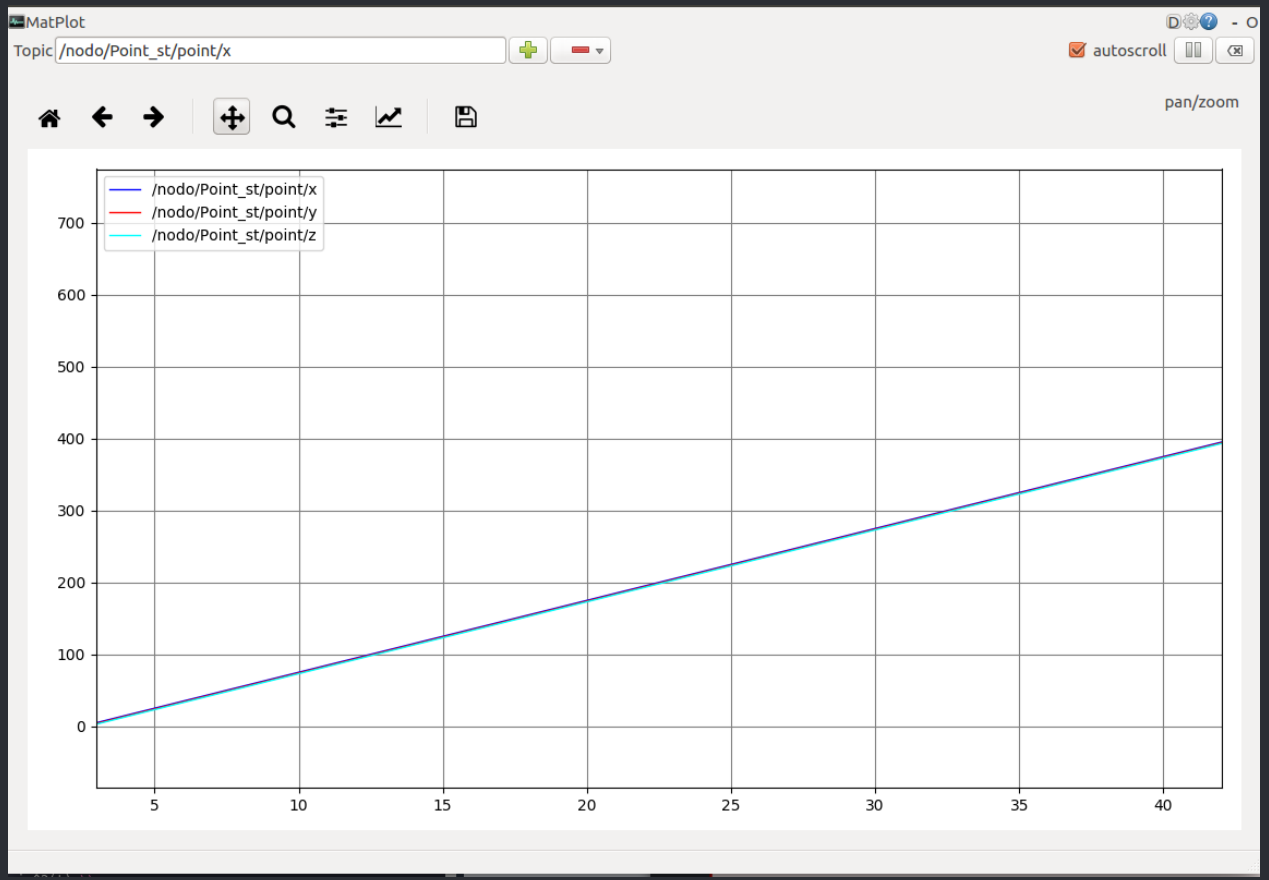
rqt_plot es una herramienta que permite visualizar mensajes por medio de una gráfica en dos dimensiones; señales dependientes del tiempo.

- **rqt_plot :** Iniciar interface.
- **rqt_plot <topico> :** Iniciar interface sobre topico.

Una vez iniciada, la interface provee un método sencillo de visualizar información en tiempo real.

Mensajes Compuestos

Temporizados



Herramienta rqt_plot

Mensajes Compuestos

Temporizados

Práctica 6: Generar un mensaje estampado

Generar un nodo que publique un mensaje del tipo **geometry_msgs::PointStamped** con la información siguiente:

- x = sin(t)
- y = cos(t)
- z = $\sin^2(t)$

Posteriormente, mirar evolución de la gráfica usando **rqt_plot**.

Mensajes Compuestos

Imágenes

Manejo de imágenes en tópicos de ROS

La clase **sensor_msgs** provee de varios tipos de mensajes que habilitan la publicación de arreglos de imágenes por medio de la libreria **OpenCV** y otras utilidades.

- Se debe crear un objeto de la clase **ImageTransport** sobre el **NodeHandle**, responsable de **publicar** y **recibir** información.
- La clase **cv_bridge** provee varias utilidades para hacer conversiones entre arreglos de la clase **sensor_msgs** y **OpenCV**.
- Resulta necesario conocer el tipo de **codificación** a utilizar en las imágenes, a fin de evitar la pérdida de información.
- Añadir libreria y dependencias correspondientes a código y archivo **CMakeLists.txt**.

Mensajes Compuestos

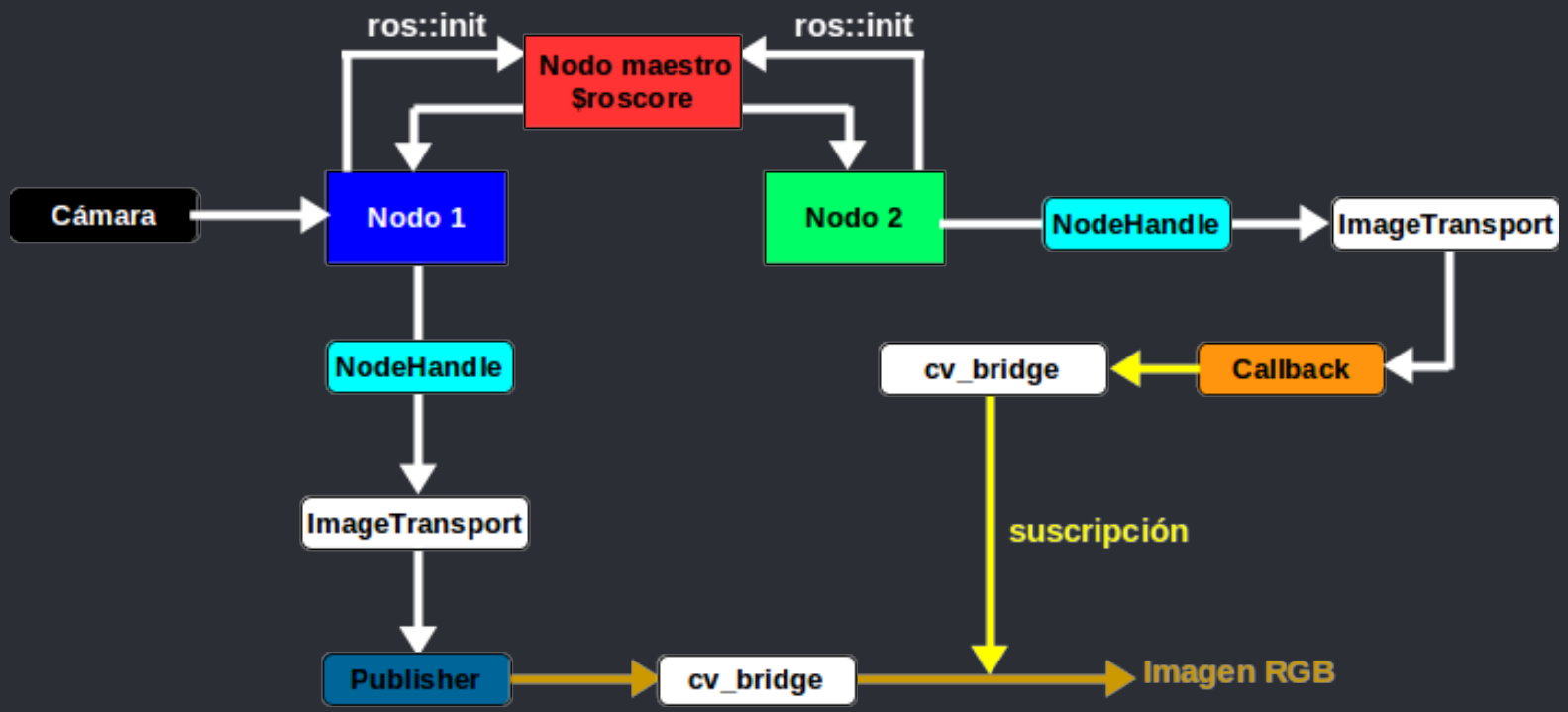
Imágenes

Publicar un tópico de imágenes (C++)

1. Crear un objeto de la clase ImageTransport sobre el **NodeHandle** del nodo.
2. Crear objeto **cvImage** de la clase **cv_bridge**.
3. Crear **Pulisher** de la clase **image_transport**.
4. Acceder a cámara o fuente de imagenes con librerías (como OpenCV).
5. Adquirir imagen en el ciclo principal del nodo y almacenarla el en campo **image** del objeto **cv_bridge**.
6. Usar función **toImageMsg()** del **cv_bridge** para generar mensaje y publicarlo.

Mensajes Compuestos

Imágenes



Publicación de imágenes con ROS

Mensajes Compuestos

Imágenes

```
int main( int argc, char** argv )
{
  //Initiazar nodo
  ros::init(argc, argv, "image_cpp");

  //Inicializar NodeHandle
  ros::NodeHandle n("~");

  //Iniciar ImageTransport sobre NodeHandle
  image_transport::ImageTransport it(n);

  //Inicializar publisher
  image_transport::Publisher  im_pub;
  im_pub =  it.advertise("camara/rgb", 1);
```

Mensajes Compuestos

Imágenes

```
//Crear objeto cv_bridge
cv_bridge::CvImage cvi;

//Crear objeto de OpenCV
cv::VideoCapture input_video;

//Acceder a camara con ID 0
input_video.open(0);

//Fijar tasa de procesamiento del nodo
ros::Rate loop_rate(30);
```

Mensajes Compuestos

Imágenes

```
//Ciclo principal
while (ros::ok())
{
    //Obtener imagen de la camara
    input_video.read(cvi.image);

    //Leer imagen del disco
    cvi.image = cv::imread("imagen.png");

    //Redimensionar imagen
    cv::resize(cvi.image, cvi.image, cv::Size(640,480));

    //Estampar mensaje
    cvi.header.stamp = ros::Time::now();
```

Mensajes Compuestos

Imágenes

```
        //Fijar tipo de codificaci n
        cvi.encoding = "bgr8";

        //Convertir imagen de OpenCV a mensaje y publicarlo
        im_pub.publish(cvi.toImageMsg());

        //Retardo del nodo
        loop_rate.sleep();
    }
```

Mensajes Compuestos

Imágenes

sensor_msgs/Image Message

File: sensor_msgs/Image.msg

Raw Message Definition

```
# This message contains an uncompressed image
# (0, 0) is at top-left corner of image
#

Header header        # Header timestamp should be acquisition time of image
                     # Header frame_id should be optical frame of camera
                     # origin of frame should be optical center of camera
                     # +x should point to the right in the image
                     # +y should point down in the image
                     # +z should point into to plane of the image
                     # If the frame_id here and the frame_id of the CameraInfo
                     # message associated with the image conflict
                     # the behavior is undefined

uint32 height         # image height, that is, number of rows
uint32 width          # image width, that is, number of columns

# The legal values for encoding are in file src/image_encodings.cpp
# If you want to standardize a new string format, join
# ros-users@lists.sourceforge.net and send an email proposing a new encoding.

string encoding       # Encoding of pixels -- channel meaning, ordering, size
                      # taken from the list of strings in include/sensor_msgs/image_encodings.h

uint8 is_bigendian    # is this data bigendian?
uint32 step           # Full row length in bytes
uint8[] data          # actual matrix data, size is (step * rows)
```

Compact Message Definition

```
std_msgs/Header header
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

autogenerated on Sun, 09 Feb 2020 03:18:29

Mensaje sensor_msgs::Image.

Mensajes Compuestos

Imágenes

Publicar un tópico de imágenes (Python)

1. Crear objeto de la clase **cv_bridge**.
   cvi = CvBridge()
2. Crear **Pulisher**.
3. Acceder a cámara o fuente de imagenes con librerías (como OpenCV).
4. Adquirir imagen en el ciclo principal del nodo y almacenarla el en campo **image** del objeto **cv_bridge**.
5. Usar función **cv2_to_imgmsg()** del **cv_bridge** para generar mensaje y publicarlo.

Mensajes Compuestos

Imágenes

```
#Metodo principal
def main():

  #Registrar nodo
  rospy.init_node('image_py')

  #Anunciar publicacion de mensaje y crear publisher
  im_pub = rospy.Publisher("/camara/rgb", Image, queue_size = 1)

  #Crear objeto cvbridge
  cvi = CvBridge()

  #Acceder a camara con ID 0
  input_video = cv2.VideoCapture(0)
```

Mensajes Compuestos

Imágenes

```
#Fijar tasa de publicacion
rate = rospy.Rate(30)

#Ciclo principal
while not rospy.is_shutdown():

  #Obtener imagen de la camara
  #ret_val, cvi.image = input_video.read()
  cvi.image = cv2.imread("imagen.png")
  cvi.image = cv2.resize(cvi.image, (640, 480))

  #Mostrar imagen
  cv2.imshow("imagen", cvi.image)
  cv2.waitKey(10)
```

Mensajes Compuestos

Imágenes

```
    #Crear mensaje
    im_msg = cvi.cv2_to_imgmsg(cvi.image, encoding="bgr8")

    #Construir mensaje
    im_msg.header.stamp = rospy.get_rostime()
    im_msg.encoding = "bgr8";

    #Publicar mensaje
    im_pub.publish(im_msg)

    #Esperar lazo
    rate.sleep()

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

Mensajes Compuestos

Imágenes

Herramienta rqt_image_view

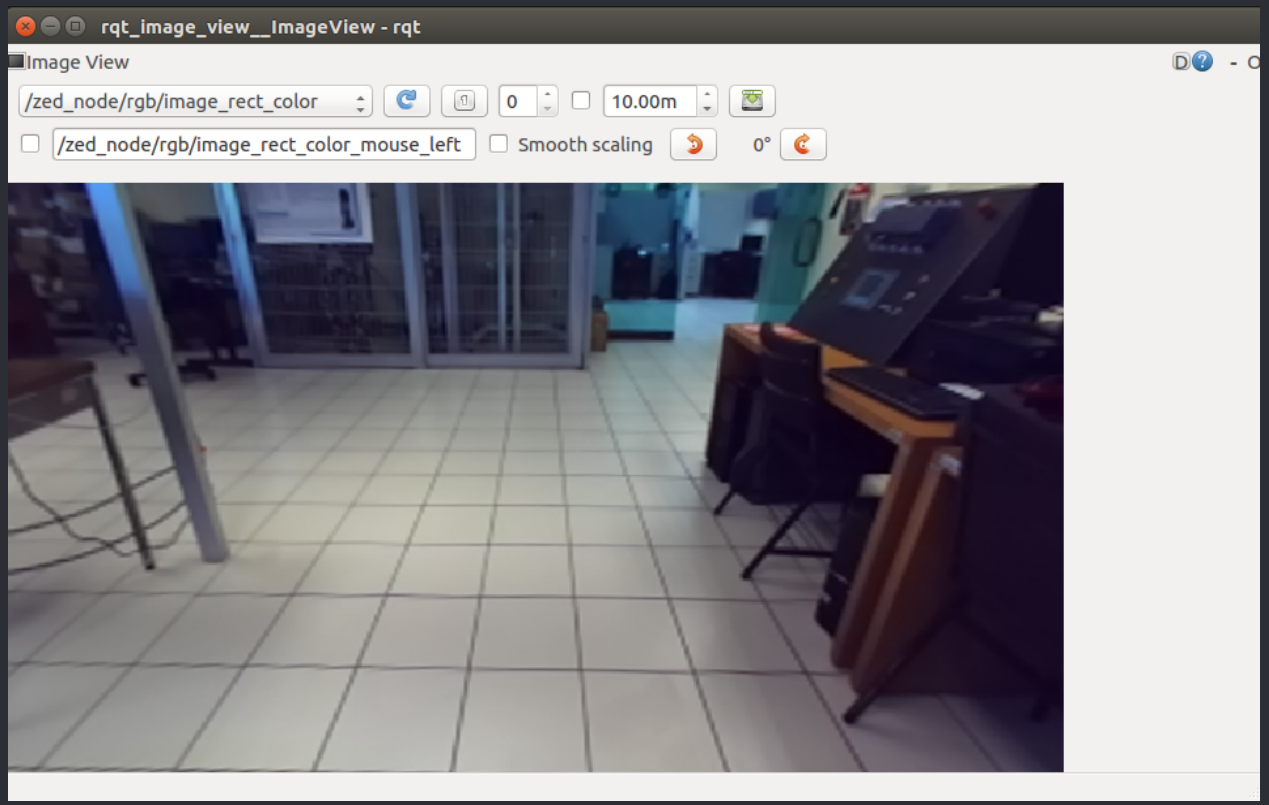
rqt_image_view es una herramienta que permite visualizar diferentes tipos de arreglos relacionados con imágenes.

- **rqt_image_view :** Iniciar interface.
- **rqt_image_view <topico> :** Iniciar interface sobre topico.

Una vez iniciada, la interface provee un método sencillo de visualizar información en tiempo real.

Mensajes Compuestos

Imágenes



Herramienta rqt_image_view

Mensajes Compuestos

Imágenes

Práctica 7: Publicando imagenes con OpenCV y ROS

Crear un nodo llamado **ros_camera** que realize las funciones siguientes:

- Adquirir imagen de una cámara conectada al sistema usando OpenCV.
- Publicar imagen en el tópico **camara/rgb**
- Visualizar imagen usando **rqt_image_view**.

Mensajes Compuestos

Imágenes

Proceso de suscripción a un tópico de imágenes

1. Crear una función **callback**
   void ImageCallback(const sensor_msgs::ImageConstPtr msg);
2. Crear objeto **Subscriber** de **image_transport**
   image_transport::Publisher im_pub;
3. Crear objeto **ImageTransport** sobre **NodeHandle**
   image_transport::ImageTransport it(n);
4. Registrar callback.
   image_transport::Subscriber im_s;
   im_s = it.subscribe("topico", 1, ImageCallback);

Mensajes Compuestos

Imágenes

Proceso de suscripción a un tópico de imágenes (C++)

5. Usar función **ros::spin** sobre ciclo principal para llamar contínuamente al **callback**.
6. En la implementación del **callback**, usar función **cv_bridge::toCvShare()** con el tipo de codificación correcto para generar un arreglo de OpenCV de la imagen del tópico.
7. Procesar arreglo resultante dentro del mismo callback.

Mensajes Compuestos

Imágenes

```
//Inicializar callback
void ImageCallback(const sensor_msgs::ImageConstPtr& msg);

int main( int argc, char** argv )
{
  //Inicializar nodo
  ros::init(argc, argv, "image_talker");

  //Iniciar NodeHandle
  ros::NodeHandle n("~");
  image_transport::ImageTransport it(n);
```

Mensajes Compuestos

Imágenes

```
    //Suscribirse a topico
    image_transport::Subscriber im_sub;
    im_sub = it.subscribe("camara/rgb", 1, ImageCallback);

    //Fijar tasa del nodo
    ros::Rate loop_rate(30);

    while (ros::ok())
    {
        ros::spinOnce();
        loop_rate.sleep();
    }
}
```

Mensajes Compuestos

Imágenes

```
//Implementar callback
void ImageCallback(const sensor_msgs::ImageConstPtr& msg)
{
    //Obtener imagen del mensaje
    cv::Mat img = cv_bridge::toCvShare(msg, "bgr8")->image;

    //Procesar imagen
    process(img);
}
```

Mensajes Compuestos

Imágenes

Proceso de suscripción a un tópico de imágenes (Python)

1. Crear una función **callback**
   def ImageCallback(img_msg);
2. Crear objeto **Subscriber**.
3. Registrar callback.
4. Usar función **rospy.spinOnce()** sobre ciclo principal para llamar contínuamente al **callback**.

Mensajes Compuestos

Imágenes

Proceso de suscripción a un tópico de imágenes (Python)

5. En la implementación del **callback**, usar función **bridge.imgmsg_to_cv** con el tipo de codificación correcto para generar un arreglo de OpenCV de la imagen del tópico.
6. Procesar arreglo resultante dentro del mismo callback.

Mensajes Compuestos

Imágenes

```
#Callbacks
def Imagecallback(img_msg):
  #Inicializar cvbridge
  cvi = CvBridge()

  #Convertir mensaje de ROS a OpenCV
  img = cvi.imgmsg_to_cv2(img_msg, "bgr8")

  #Visualizar topico
  cv2.imshow("imagen", img)
  cv2.waitKey(10)
```

Mensajes Compuestos

Imágenes

```
#Metodo principal
def main():

  #Registrar nodo
  rospy.init_node('image_subs_py')

  #Crear subscriber
  rospy.Subscriber('/camara/rgb', Image, Imagecallback)

  #Ciclo principal
  while not rospy.is_shutdown():

  #Llamar a los callbacks
  rospy.spin()
```

Mensajes Compuestos

Imágenes

```
#Llamada a metodo
if __name__ == '__main__':
  main()
```

Mensajes Compuestos

Imágenes

Tipos de codificación mas cómunes

- **mono8: CV_8UC1** Imagen en escala de grises.
- **mono16: CV_16UC1** Imagen de 16 bits.
- **bgr8: CV_8UC3** Imagen BGR de 8 bits.
- **rgb8: CV_8UC3** Imagen RGB de 8 bits.
- **32FC1: CV_32FC1** Imagen monocular de flotantes, generalmente empleada en mapas de profundidad.

Mensajes Compuestos

Imágenes

Leer valores de arreglos de OpenCV

C++

- **Array.at<tipo>(i, j):** Acceder a valor de arreglo de un solo canal.
- **Array.at<tipo>(i, j)[c]:** Acceder a valor de arreglo sobre un canal.

Python

- **Array[i, j, k]:** Acceder a valor de arreglo.

Mensajes Compuestos

Imágenes

Práctica 8: Suscribiendose a un tópico de imágenes

Crear un nodo llamado **image_suscriber** que realize las funciones siguientes:

- Se suscriba al tópico **camera/rgb** del nodo de la práctica anterior.
- El nodo debe convertir la imagen recibida a escala de gises, y posteriormente publicarla en el tópico **camera/mono**.

Mensajes Compuestos

Mensajes 3D

Mensajes en 3D

ROS dispone de diferentes tipos de mensajes a travez de la clase **visualization_msgs** que permiten su visualización en 3D, a fin de facilitar el proceso de **debug** y medición de rendimiento de los algoritmos. Entre estos mensajes se incluyen:

- Imágenes RGB y mapas de profundiad.
- Nubes de puntos
- Pose y posición de varios tipos de marcadores (esferas, lineas, marcos de referencia etc.)
- Pose y posición de objetos abstractos (meshes o dibujos de solidWorks).

Mensajes Compuestos

Mensajes 3D

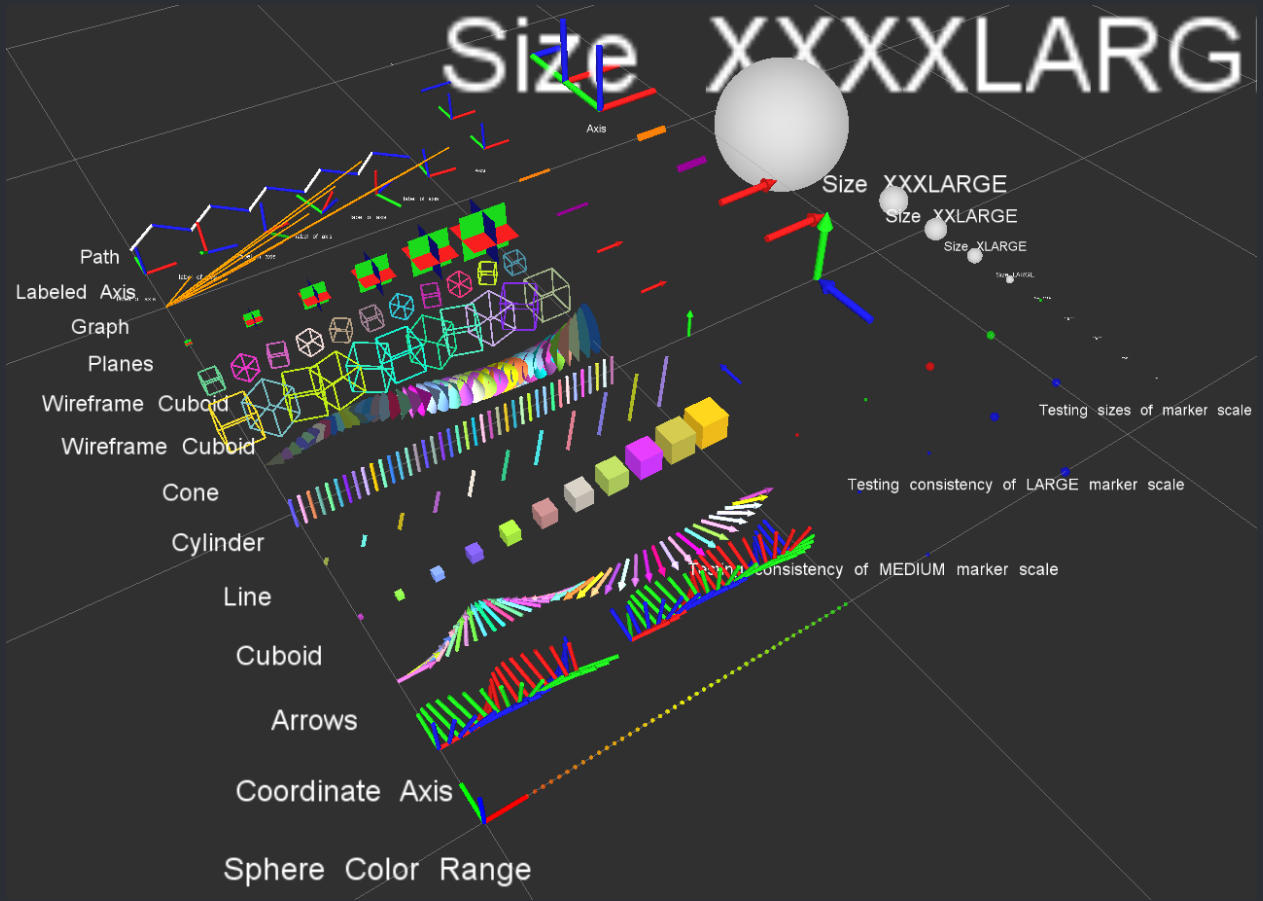
Herrramienta RViz

RViz es una herramienta de ROS que permite la visualización de diferentes tipos de datos como imágenes y mensajes en 3D a travéz de un entorno simulado, donde se puede controlar la pose y la posición de diferentes tipos de objetos.

- **Inicio:** rosrun rviz rviz
- Los objetos a visualizar deben especificar un **marco de referencia**, que es usado para representar diferentes planos en el visualizador.
- RViz utiza el tiempo de referencia del nodo maestro, por lo que es posible reproducir mensajes grabados.

Mensajes Compuestos

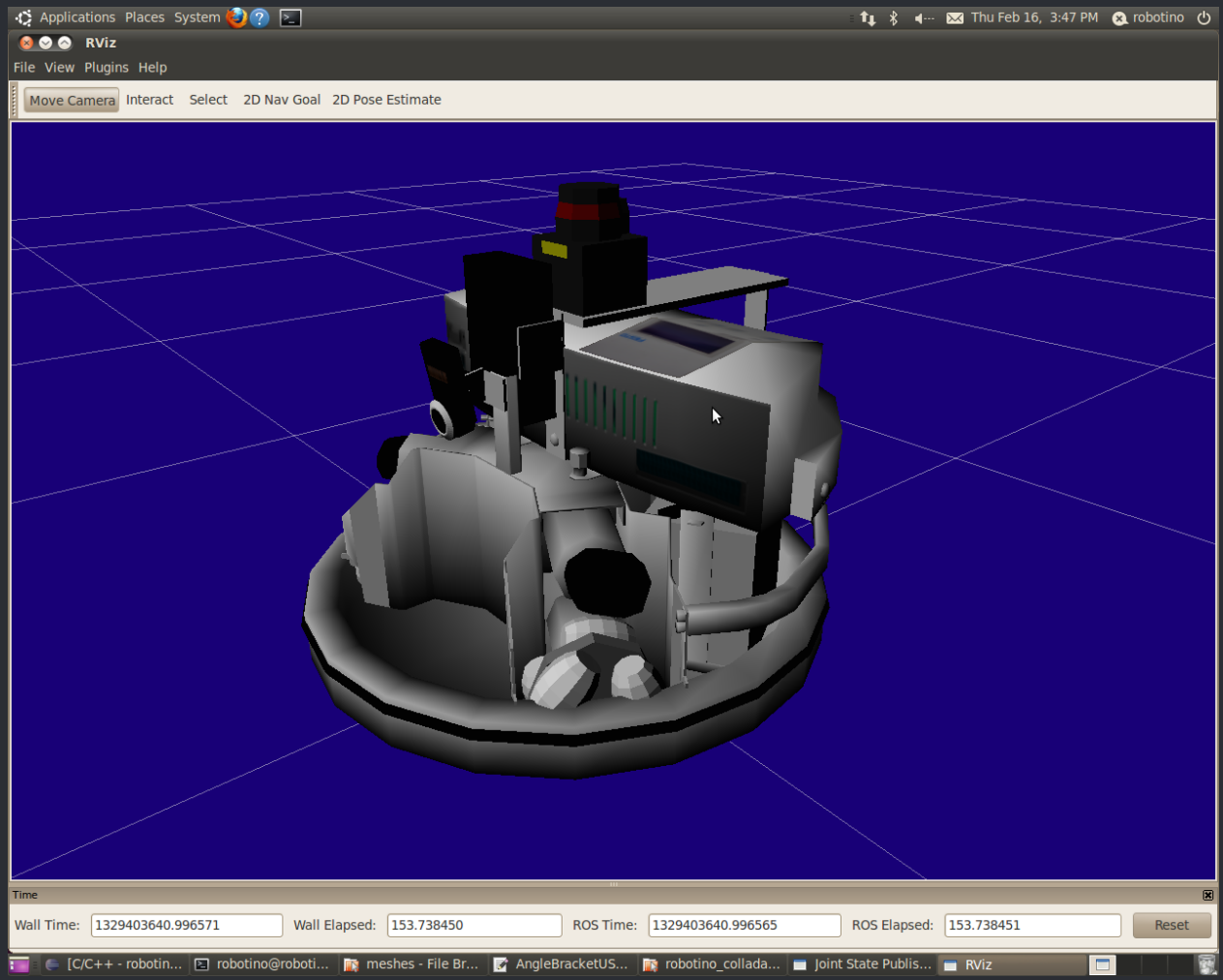
Mensajes 3D



Tipos de visualizaciones en RViz

Mensajes Compuestos

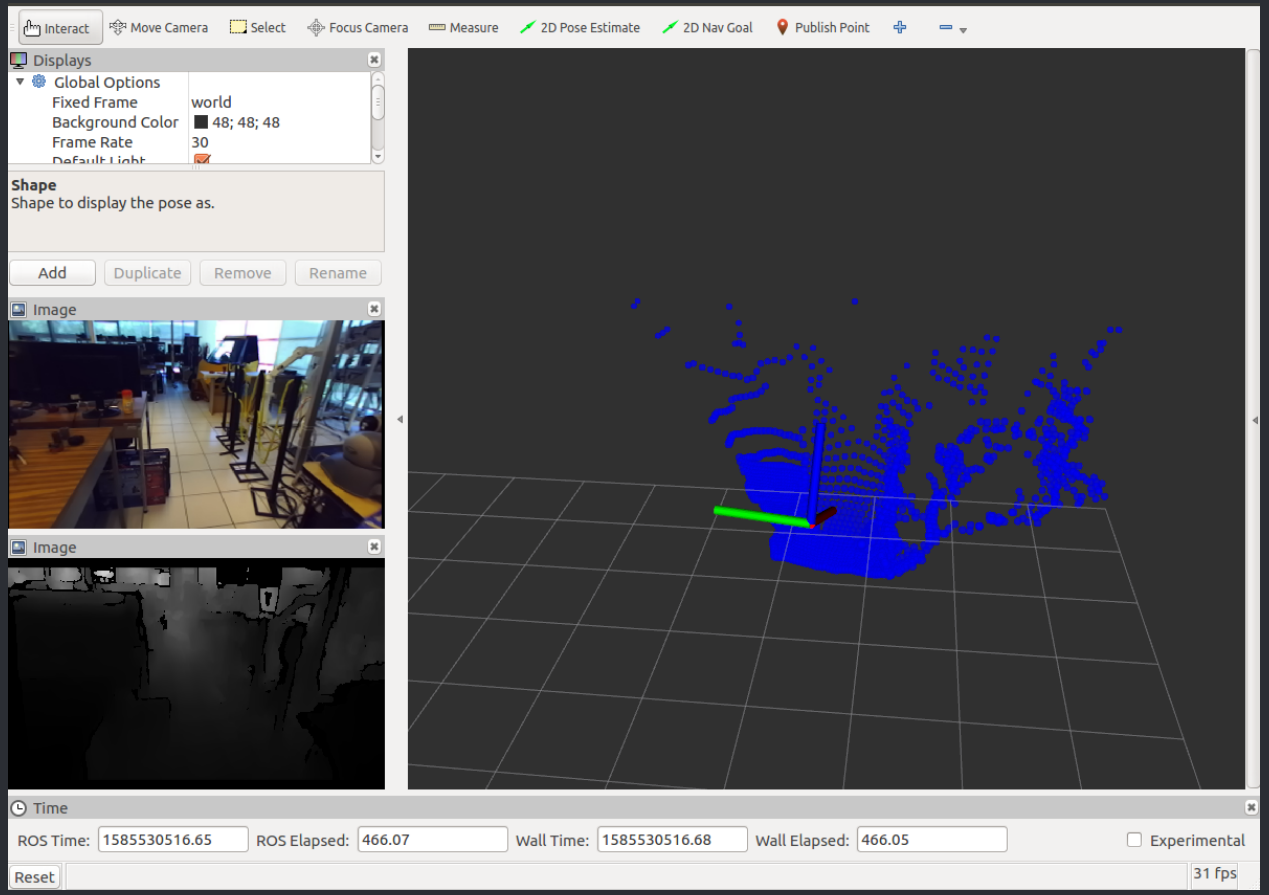
Mensajes 3D



Visualización de dibujos en RViz

Mensajes Compuestos

Mensajes 3D



Visualización de posiciones en RViz

Mensajes Compuestos

Mensajes 3D

visualization_msgs

visualization_msgs es una clase que provee los tipos de mensajes básicos para la construcción de mensajes en 3D visualizables por RViz.

Mensajes principales:

- **Marker:** Constituye el bloque básico de construcción de mensajes, el cual incluye flechas, esferas, cilindors, texto, puntos, lineas y dibujos. Así como **listas** de elementos básicos.
- **MarkerArray:** Es un arreglo de uno o más marcadores.

La publicación y lectura de este tipo de mensajes se maneja como el de un mensaje regular, requiriéndo únicamente llenar los campos correspondientes.

Mensajes Compuestos

Mensajes 3D

visualization_msgs/Marker Message

File: visualization_msgs/Marker.msg

Raw Message Definition

```
# See http://www.ros.org/wiki/rviz/DisplayTypes/Marker and http://www.ros.org/wiki/rviz/Tutorials/Markers%3A%20Basic%20Shapes for more informati

uint8 ARROW=0
uint8 CUBE=1
uint8 SPHERE=2
uint8 CYLINDER=3
uint8 LINE_STRIP=4
uint8 LINE_LIST=5
uint8 CUBE_LIST=6
uint8 SPHERE_LIST=7
uint8 POINTS=8
uint8 TEXT_VIEW_FACING=9
uint8 MESH_RESOURCE=10
uint8 TRIANGLE_LIST=11

uint8 ADD=0
uint8 MODIFY=0
uint8 DELETE=2
uint8 DELETEALL=3

Header header                       # header for time/frame information
string ns                           # Namespace to place this object in... used in conjunction with id to create a unique name for the object
int32 id                          # object ID useful in conjunction with the namespace for manipulating and deleting the object later
int32 type                        # Type of object
int32 action                        # 0 add/modify an object, 1 (deprecated), 2 deletes an object, 3 deletes all objects
geometry_msgs/Pose pose                 # Pose of the object
geometry_msgs/Vector3 scale             # Scale of the object 1,1,1 means default (usually 1 meter square)
std_msgs/ColorRGBA color            # Color [0.0-1.0]
duration lifetime                   # How long the object should last before being automatically deleted.  0 means forever
bool frame_locked                   # If this marker should be frame-locked, i.e. retransformed into its frame every timestep

#Only used if the type specified has some use for them (eg. POINTS, LINE_STRIP, ...)
geometry_msgs/Point[] points
#Only used if the type specified has some use for them (eg. POINTS, LINE_STRIP, ...)
#number of colors must either be 0 or equal to the number of points
#NOTE: alpha is not yet used
std_msgs/ColorRGBA[] colors

# NOTE: only used for text markers
string text

# NOTE: only used for MESH_RESOURCE markers
string mesh_resource
bool mesh_use_embedded_materials
```

Marcador estándar de visualization_msgs

Mensajes Compuestos

Mensajes 3D

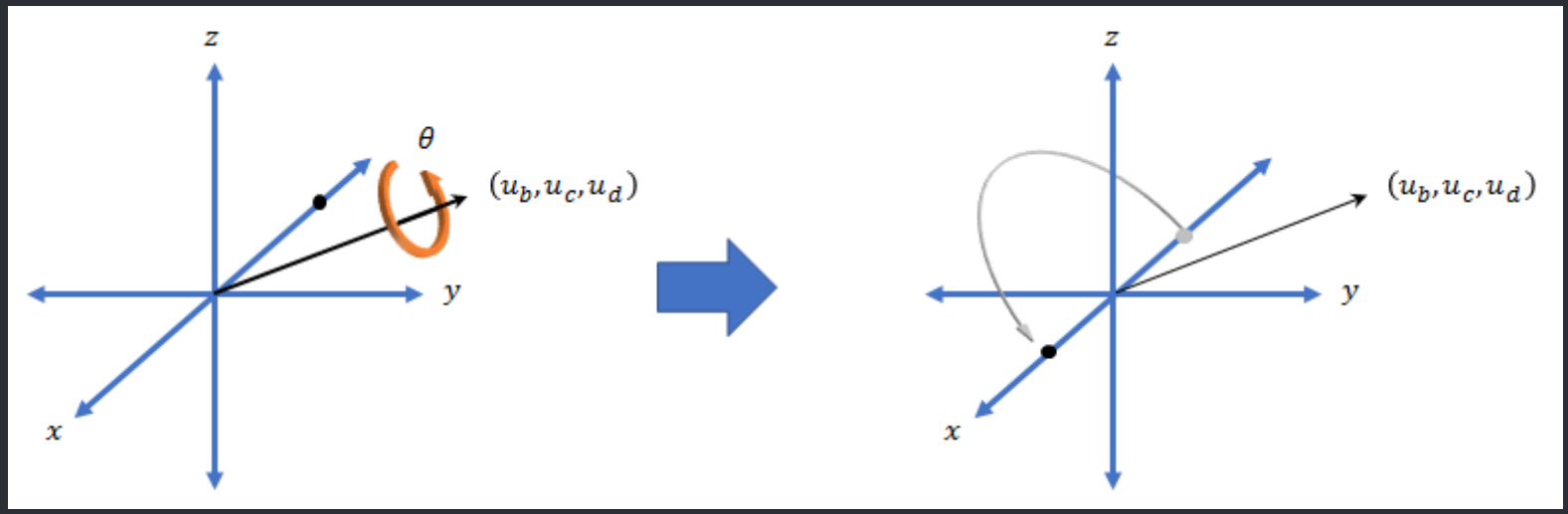
Representación de posiciones RViz

RViz emplea un **vector** para representar la posición de un objeto, y un **quaternion** para representar la pose.

- Un **quaternión** es un número complejo compuesto por un número real y 3 imaginarios:
  $q = q_0 + q_x\hat{i} + q_y\hat{j} + q_z\hat{k}$
- Los número imaginarios representan la posición de un vector unitario, que es el eje sobre el cual se rota el cuerpo.
- El número real representa el grado de rotación sobre dicho eje (en radianes).

Mensajes Compuestos

Mensajes 3D



Representación de una pose por medio de un quaternión

Mensajes Compuestos

Mensajes 3D

Representación de la pose por quaterniones

Conversión de ángulos de Euler a quaternión

$$q_0 = \frac{1}{8}(cos\alpha * cos\beta * cos\gamma + sin\alpha * sin\beta * sin\gamma) \quad (1)$$

$$q_x = \frac{1}{8}(sin\alpha * cos\beta * cos\gamma - cos\alpha * sin\beta * sin\gamma) \quad (2)$$

$$q_y = \frac{1}{8}(cos\alpha * sin\beta * cos\gamma + sin\alpha * cos\beta * sin\gamma) \quad (3)$$

$$q_z = \frac{1}{8}(cos\alpha * cos\beta * sin\gamma + sin\alpha * sin\beta * cos\gamma) \quad (4)$$

Mensajes Compuestos

Mensajes 3D

Representación de la pose por quaterniones

Conversión de quaterniones a ángulos de Euler

$$R = \begin{bmatrix} 1 - 2(q_y q_y + q_z q_z) & 2(q_x q_y - q_z q_w) & 2(q_x q_z + q_y q_w) \\ 2(q_x q_y + q_z q_w) & 1 - 2(q_x q_x + q_z q_z) & 2(q_y q_z - q_x q_w) \\ 2(q_x q_z - q_y q_w) & 2(q_y q_z + q_x q_w) & 2(q_y q_z + q_x q_w) \end{bmatrix} \tag{5}$$

Mensajes Compuestos

Mensajes 3D

Pose y posición de un objeto

De esta forma, cualquier objeto tridimensional representado en **RViz** requiere de los parámetros siguientes:

- **ID:** Identicador único asignado al objeto que lo distinga de otros del mismo tópico (en caso de ser arreglos).
- **Posición:** Dada por un **punto** P = [p.x, p.y. p.z].
- **Pose:** Dada por un **quaternión** q = [q.w, q.x, q.y, q.z].
- **Color**: Dado por una configuración **RGB** y un valor de **transparencia** c = [c.r, c.h, c.b, c.a].
- **Escala**: Dado por un vector tridimensional e = [e.x, e.y, e.z].

Mensajes Compuestos

Mensajes 3D

```
//Publishers
ros::Publisher mark_pub;

//Metodo principal
int main(int argc, char** argv)
{
    //Inicializar nodo (registro)
    ros::init(argc, argv, "markers_cpp");

    //Crear objeto nodehandle
    ros::NodeHandle n("~");

    //Informar a nodehandle de publicacion de mensaje
    mark_pub = n.advertise<visualization_msgs::Marker>("/
        crazy_sphere/sphere", 1);
```

Mensajes Compuestos

Mensajes 3D

```
//Definir la tasa de procesamiento
ros::Rate loop_rate(15);

//Crear mensaje
visualization_msgs::Marker body_msg;

//Configurar mensaje
body_msg.id = 0;
body_msg.type = 2;
body_msg.header.frame_id = "world";
```

Mensajes Compuestos

Mensajes 3D

```
//Fijar escala y color del marcador
body_msg.scale.x = 0.5;
body_msg.scale.y = 0.5;
body_msg.scale.z = 0.5;
body_msg.color.r = 0.0f;
body_msg.color.g = 0.0f;
body_msg.color.b = 1.0f;
body_msg.color.a = 1.0;

//Fijar pose
body_msg.pose.orientation.w = 1.0;
body_msg.pose.orientation.y = 0.0;
body_msg.pose.orientation.x = 0.0;
body_msg.pose.orientation.z = 0.0;
```

Mensajes Compuestos

Mensajes 3D

```
//Iniciar tiempo (para efectos de movimiento)
float time = 0.0;

//Ciclo principal
while(ros::ok())
{
    //Llenar cabecera
    body_msg.header.stamp = ros::Time::now();

    //Modificar posicion
    body_msg.pose.position.x = 2.0*sin(time);
    body_msg.pose.position.y = 2.0*cos(time);
    body_msg.pose.position.z = 2.0*sin(time);
```

Mensajes Compuestos

Mensajes 3D

```
        //Aumentar tiempo
        time = time + 0.1;

        //Publicar mensajes
        mark_pub.publish(body_msg);

        //Retardo
        loop_rate.sleep();
    }
```

Mensajes Compuestos

Mensajes 3D

```
#Metodo principal
def main():

  #Registrar nodo
  rospy.init_node('sphere_py')

  #Informar a nodehandle de publicacion de mensaje
  mark_pub = rospy.Publisher('/markers_py/sphere', Marker,
      queue_size=1)

  #Fijar tasa de publicacion
  rate = rospy.Rate(15)

  #Crear mensaje
  body_msg = Marker()
```

Mensajes Compuestos

Mensajes 3D

```
#Configurar mensaje
body_msg.id = 0
body_msg.type = 2
body_msg.header.frame_id = "world"

#Fijar escala y color del marcador
body_msg.scale.x = 0.5
body_msg.scale.y = 0.5
body_msg.scale.z = 0.5
body_msg.color.r = 0.0
body_msg.color.g = 0.0
body_msg.color.b = 1.0
body_msg.color.a = 1.0
```

Mensajes Compuestos

Mensajes 3D

```
#Inicializar tiempo
time = 0.0

#Ciclo principal
while not rospy.is_shutdown():

  #Llenar cabecera
  body_msg.header.stamp = rospy.get_rostime()

  #Modificar posicion
  body_msg.pose.position.x = 2.0*np.sin(time)
  body_msg.pose.position.y = 2.0*np.cos(time)
  body_msg.pose.position.z = 2.0*np.sin(time)
```

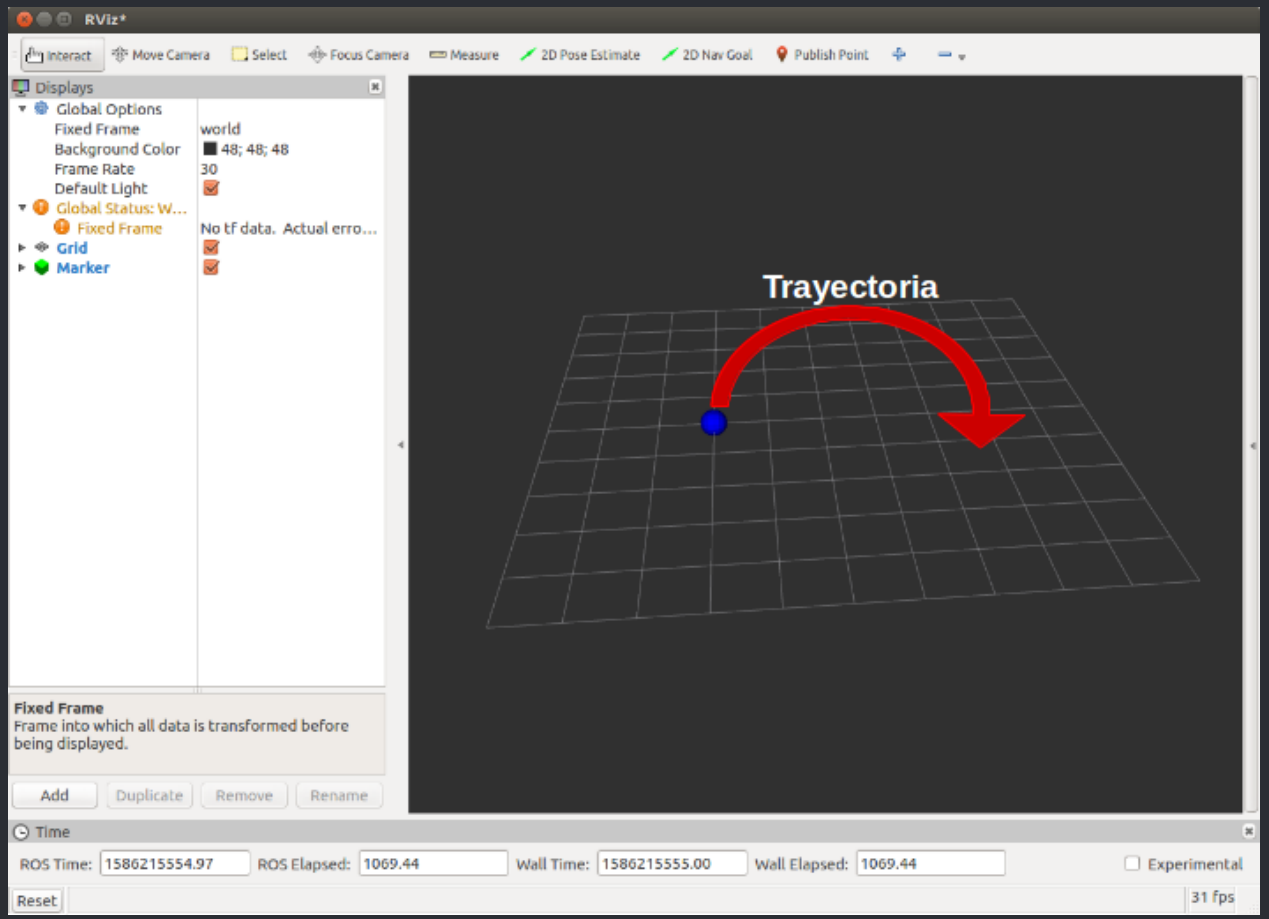
Mensajes Compuestos

Mensajes 3D

```
    #Aumentar tiempo
    time += 0.1

    #Publicar mensaje
    mark_pub.publish(body_msg)

    #Esperar lazo
    rate.sleep()

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

Mensajes Compuestos

Mensajes 3D



Movimiendo de una esfera en RViz

Mensajes Compuestos

Mensajes 3D

Práctica 9: Dibujar una esfera rotando en RViz

Crear un par de nodos llamados **center_sphere** y **mobile_sphere** que realizen las funciones siguientes:

- El primero, publique un tópico **center_sphere/sphere** del tipo **visualization_msgs::Sphere**.
- El segundo, publique un tópico **mobile_sphere/sphere** del tipo **visualization_msgs::Sphere**.
- La segunda esfera debe girar a 2 metros de la primera, y seguir los cambios en su posición.

Mensajes Compuestos

Personalizados

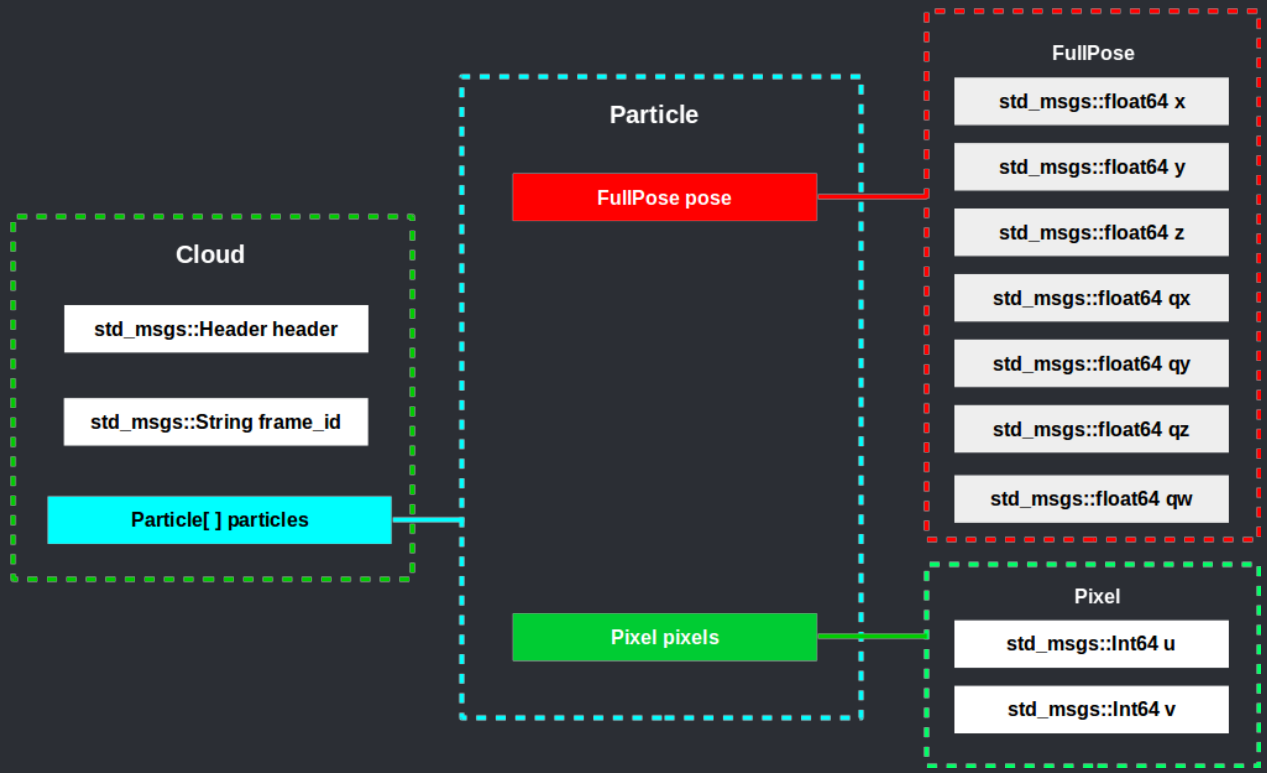
Creación de mensajes personalizados

ROS permite la creación de **mensajes personalizados**, a fin de ajustar el contenido de un tópico a una estructura específica.

- Los mensajes personalizados se construyen de mensajes de ROS **primitivos**.
- Se pueden crear estructuras a partir de mensajes personalizados, como cualquier otra estructura de ROS.
- Los mensajes pueden ser creados en **paquetes independientes**, o **dentro** de otro paquete.

Mensajes Compuestos

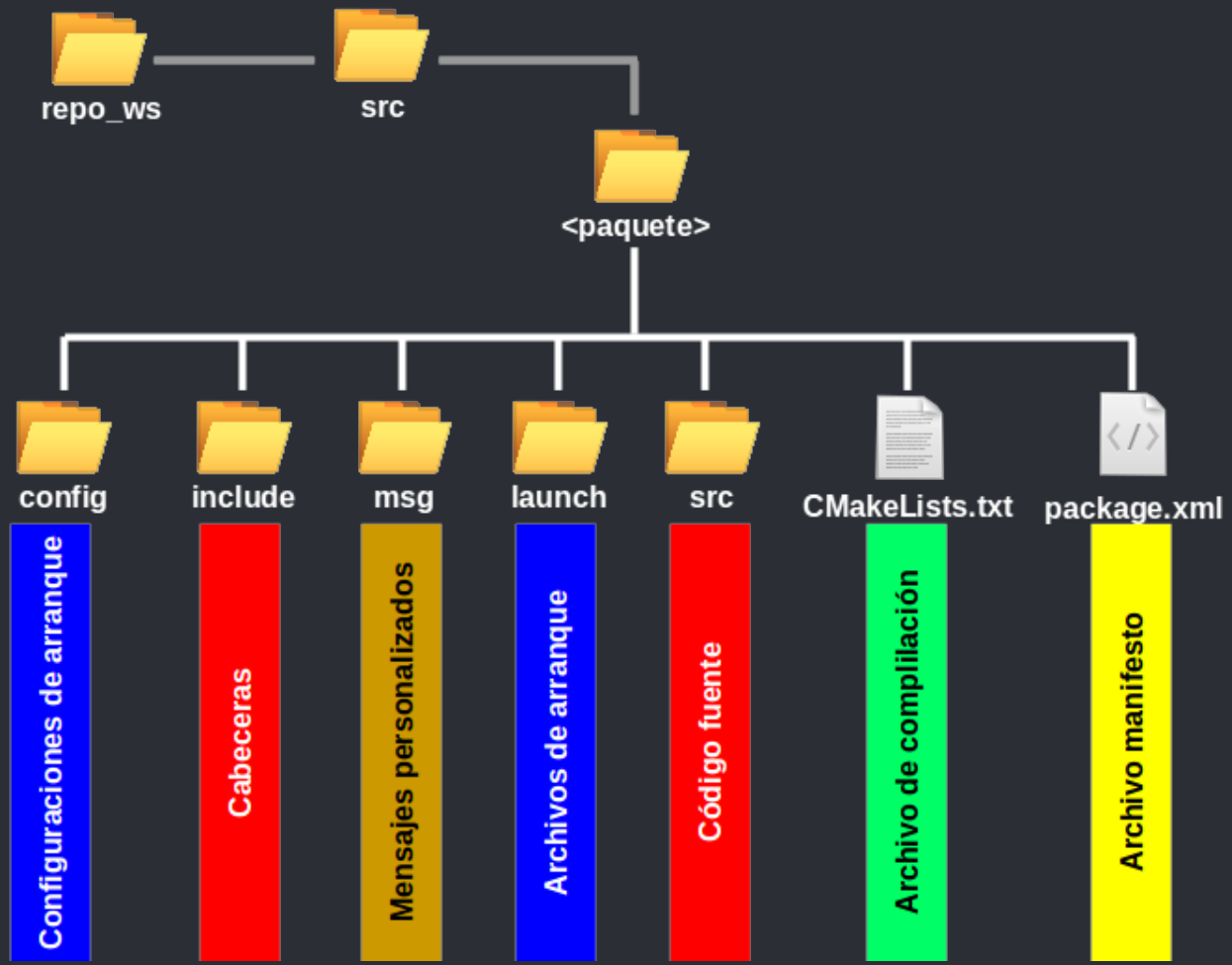
Personalizados



Construcción de mensajes personalizados

Mensajes Compuestos

Personalizados



Construcción de mensajes personalizados

Mensajes Compuestos

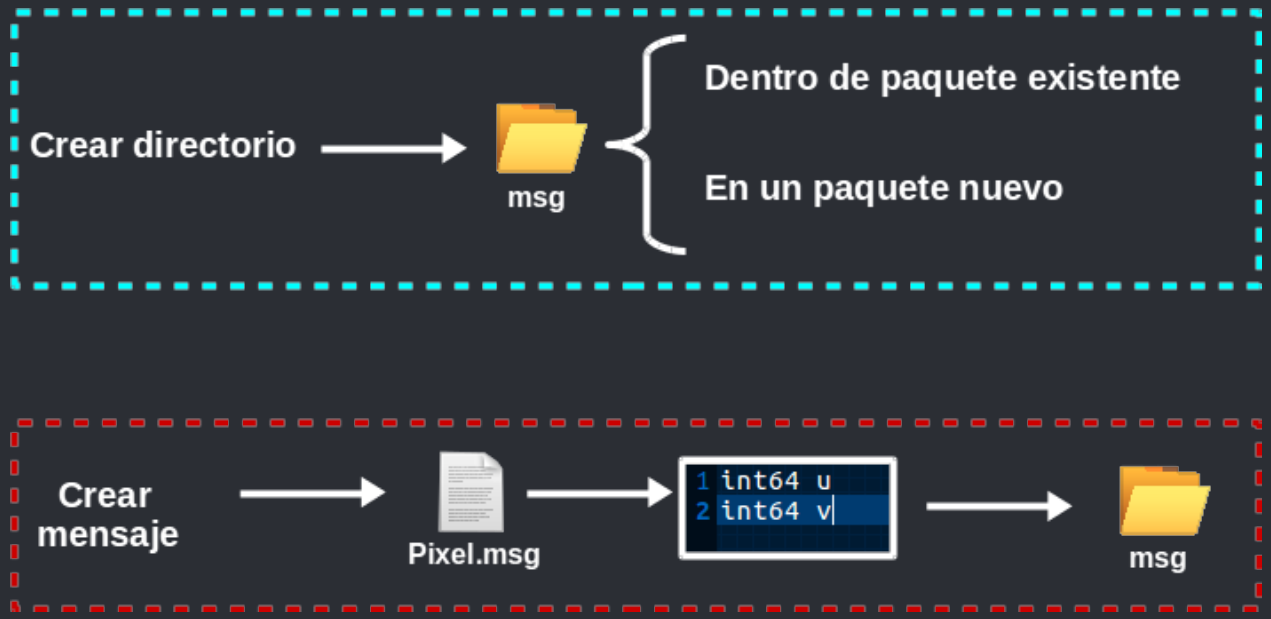
Personalizados

Proceso de creación de un mensaje personalizado

1. Crear un directorio **msg** dentro de un paquete, o crear un nuevo.
2. Crear archivos **.msg** con las estructuras deseadas.
3. Añadir dependencias a **CMakelist.txt** y **package.xml**. Adicionalmente, **message_runtime** y **message_generation**.
4. Incluir cabeceras de los mensajes en torno al nodo al paquete que las contiene:
   #include <nodo/mensaje.h>
5. Si se desea incluir mensajes de otro paquete, es necesario añadir las dependencias correspondientes en el manifesto (**package.xml**).

Mensajes Compuestos

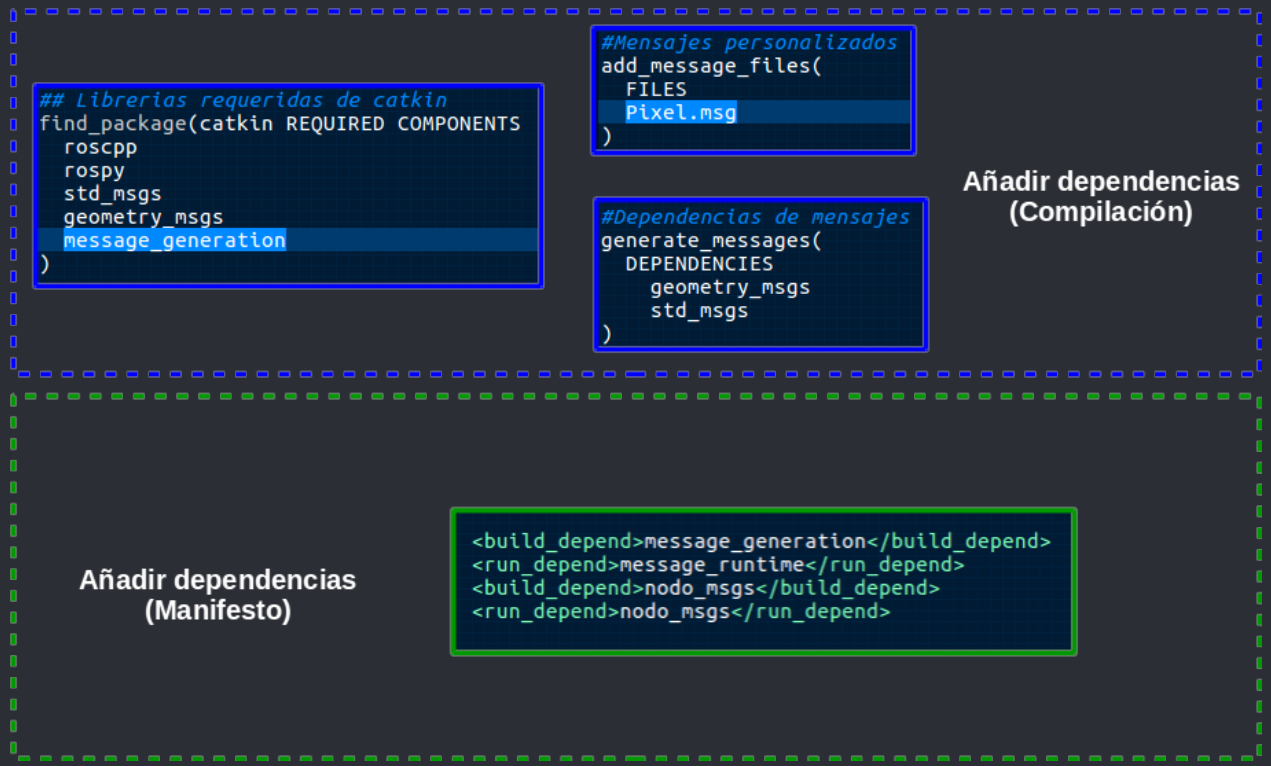
Personalizados



Creación de mensajes personalizados.

Mensajes Compuestos

Personalizados



Creación de mensajes personalizados.

Mensajes Compuestos

Personalizados

Consideraciones generales

- Es necesario correr el archivo de configuración del repositorio para poder leer el mensaje desde terminal.
- En caso de que se desee incluir un mensaje desde otro paquete, deberá añadirse la dependencia correspondiente al archivo **package.xml** con el nombre del paquete.
- Únicamente los campos de la clase **visualization_msgs** son visualizables en el plano de **RViz**.
- En un repositorio, es recomendable crear un paquete exclusivo de mensajes. A fin de facilitar su importación a otros paquetes.

Mensajes Compuestos

Personalizados

Práctica 10: Crear un mensaje personalizado

Crear un mensaje personalizado llamado **particula** con el contenido siguiente

- Campo **geometry_msgs::Point** llamado **posicion**.
- Campo **geometry_msgs::Quaternion** llamado **orientacion**.
- Campo **std_msgs::Header** llamado **header**.
- Crear mensaje **nodo::pixel** y añadirlo como campo **pixels**.
- Publicarlo en el topico **/nodo/particula**.

Tabla de contenido

Tópicos avanzados

Arranque avanzado

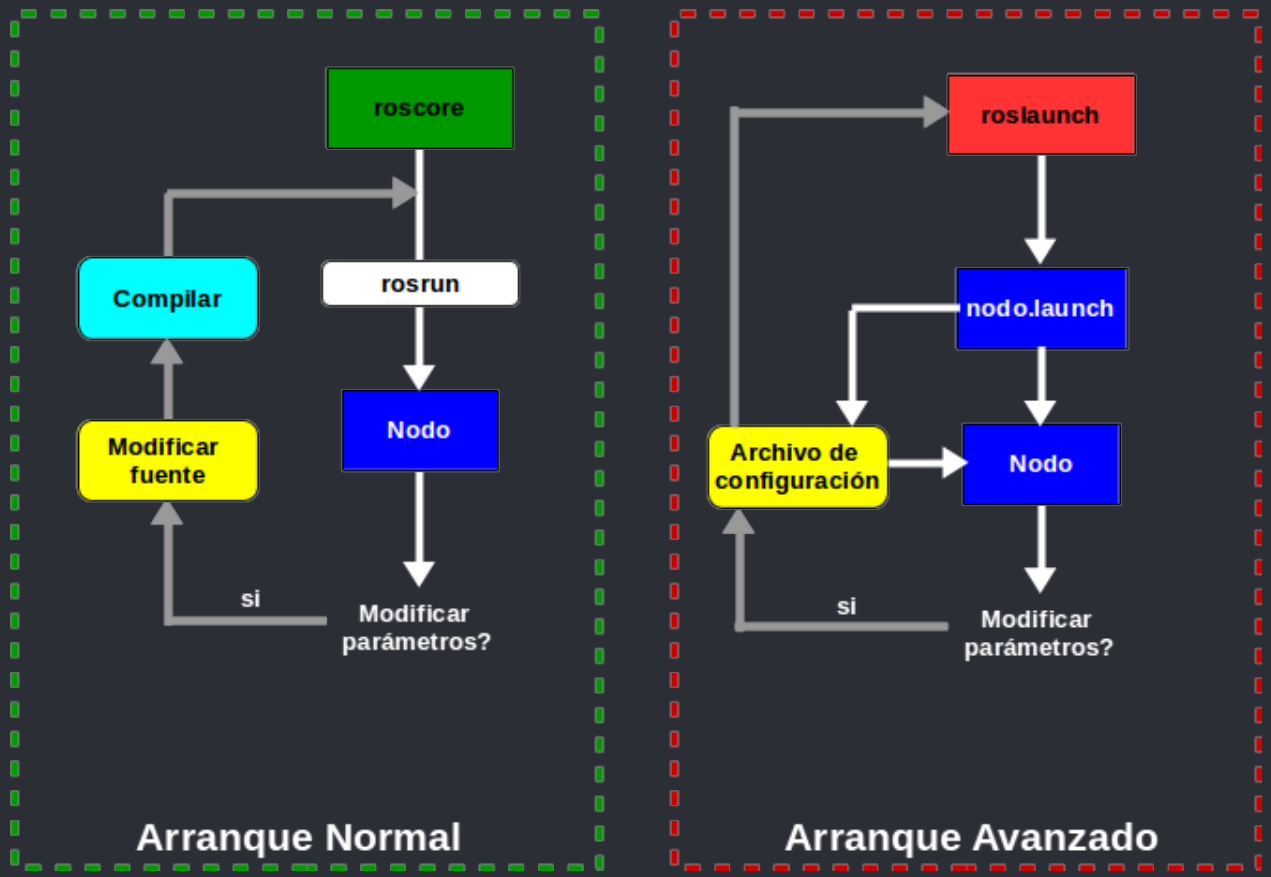
Arranque Avanzado

El **arranque avanzado** es un método para correr un nodo de ROS utilizando **parámetros diñamicos**, los cuales pueden ser modificados después de la compilación.

- El arranque avanzado se caracteriza por el uso del comando: **roslaunch <paquete> <archivo.launch>**.
- Requiere la creación de un archivo **.launch**, que puede o no tener un archivo de configuración para leer parámetros.
- El arranque avanzado inicia un **nodo maestro** si no existe ninguno en el momento.
- Las modificaciones del **archivo de configuración** o el **.launch** no requieren **recompilación**.

Tópicos avanzados

Arranque avanzado



Tipos de arranques de nodos.

Archivo launch

Los **launch** son archivos tipo **html** que permiten arrancar un nodo usando **parámetros dinámicos**.

Funciones básicas

- **<launch>:** Apertura y cierre de nodo.
- **<node name="", pkg="", type="", output="">:** Nombre de registro del nodo, paquete, binario (nodo compilado) y tipo de salida.
- **<rosparam command="load" file="">:** Cargar parámetros de archivos de configuración.
- **<param name="" type="" value="">:** Definir parámetro.

Tópicos avanzados

Arranque avanzado

```
<launch>
    <node name="moving_sphere" pkg="nodo"
        type="moving_sphere" output="screen">
    <rosparam command="load"
        file="$(find nodo)/config/params.yaml"/>
    <param name="node_path" type="string"
        value="$(find nodo)" />
    </node>
</launch>
```

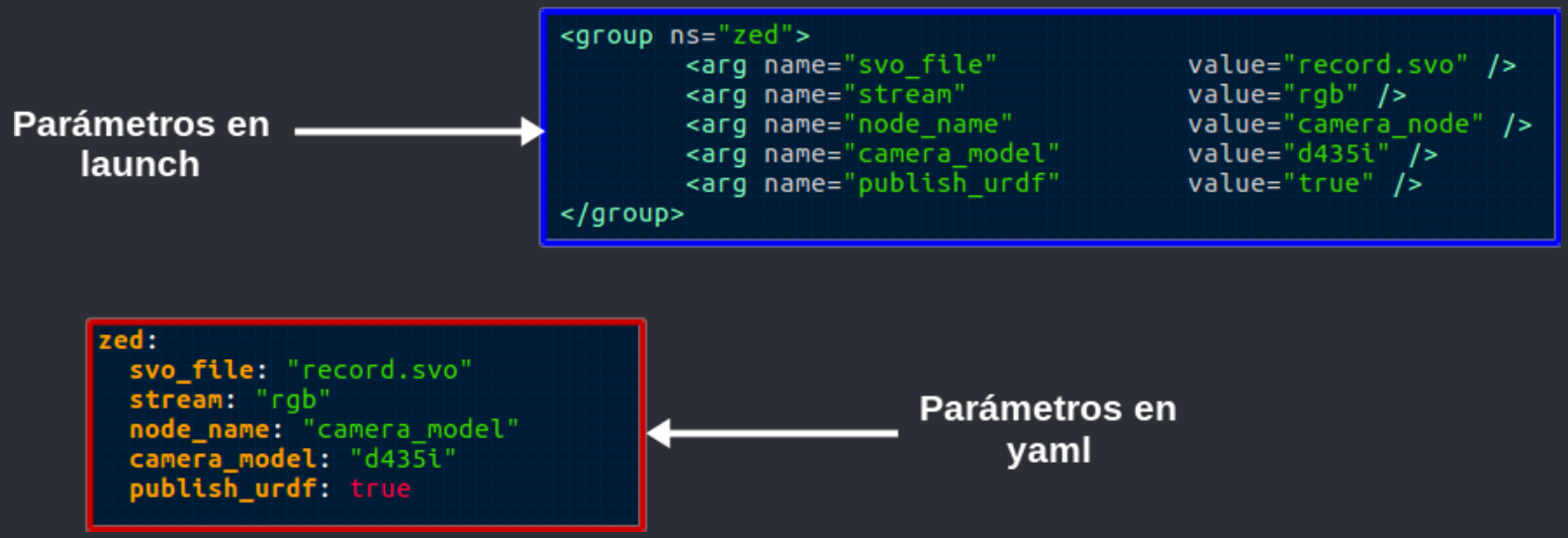
Archivos de configuración

Son archivos cargados con el comando **load** dentro del **launch**. A diferencia de colocar parámetros directamente, permiten tener un mayor orden y estructura de los parámetros.

- Generalmente, son archivos de extensión **.yaml**.
- Poseen una sintaxis mucho más simple.
- Pueden compartirse entre varios nodos especificando los campos apropiadamente.

Tópicos avanzados

Arranque avanzado



Métodos de carga de parámetros.

Tópicos avanzados

Arranque avanzado

Carga de parámetros a nivel de código (C++)

Una vez definidos los archivos de configuración es requerido realizar la lectura a nivel de código.

1. Usar método **param** de **NodeHandle** para leer parámetros de los archivos de configuración:
   n.param<cast>("parámetro", <variable>, <default>);
2. El método requiere asignar un valor default para los parámetros, en caso de que existan problemas con la lectura.
3. Los parámetros son cargados de los archivos en una única ocasión al inicio del nodo.

Tópicos avanzados

Arranque avanzado

```yaml
zed:
  svo_file: "record.svo"
  stream: "rgb"
  node_name: "camera_model"
  camera_model: "d435i"
  publish_urdf: true
```

Parámetros en yaml

Código fuente

```cpp
//Cargar parametros
n.param<std::string>("zed/svo_file", svo_file, "archivo.svo");
n.param<std::string>("zed/stream", stream, "rgb");
n.param<std::string>("zed/node_name", node_name, "camera_node");
n.param<std::string>("zed/camera_model", camera_model, "d435i");
n.param("zed/publish_urdf", publish_urdf, false);
```

Carga de parámetros en código fuente (C++).

Tópicos avanzados

Arranque avanzado

Carga de parámetros a nivel de código (Python)

Una vez definidos los archivos de configuración es requerido realizar la lectura a nivel de código.

1. Usar método **get_param** de **rospy** para leer parámetros de los archivos de configuración:
   param = rospy.get_param("parámetro", <default>)
2. El método requiere asignar un valor default para los parámetros, en caso de que existan problemas con la lectura.
3. Los parámetros son cargados de los archivos en una única ocasión al inicio del nodo.

Tópicos avanzados

Arranque avanzado

```yaml
zed:
  svo_file: "record.svo"
  stream: "rgb"
  node_name: "camera_model"
  camera_model: "d435i"
  publish_urdf: true
```

Parámetros en yaml

Código fuente

```python
#Leer parametro de archivo de configuracion
svo_file = rospy.get_param("zed/svo_file", "archivo.svo")
stream = rospy.get_param("zed/stream", "rgb")
node_name = rospy.get_param("zed/node_name", "camera_node")
camera_model = rospy.get_param("zed/camera_model", "d435i")
publish_urdf = rospy.get_param("zed/publish_urdf", False)
```

Carga de parámetros en código fuente (python).

Tópicos avanzados

Arranque avanzado

Práctica 11: Arranque avanzado

Crear un archivo tipo **launch** y un **archivo de configuración** para el nodo **crazy_sphere**. El archivo de configuración debe permitir configurar los parámetros siguientes:

- Periodo y magnitud de todos los desplazamientos.
- Radio y velocidad de segunda esfera.
- Tamaño y color de ambas esferas.

Tópicos avanzados

Herramientas básicas

Herramienta rosbag

Rosbag es una herramienta de ROS que permite guardar y cargar **bolsas de datos**, las cuales permiten almacenar y reproducir la información de los **tópicos** generados por uno o varios nodos en archivos de extensión **.bag**.

Comandos básicos

- **rosbag record <topic1> ... <topicn>:** Grabar información de uno o varios tópicos.
- **rosbag record -a:** Grabar todos los tópicos.
- **rosbag play <archivo.bag>:** Reproducir bolsa de datos.
- **rosbag info <archivo.bag>:** Ver información de archivo.

Tópicos avanzados

Herramientas básicas

Herramienta rosbag

- Es requerido que todos los topicos a guardar sean del tipo **temporizado**.
- Al reproducir un topico temporizado, los mensajes se ajustan con el reloj del **nodo maestro**, lo cual permite que se publiquen conforme transcurre el tiempo (simulando al nodo que los originó).
- Las características anteriores permiten capturar datos de diferentes sensores y después probar algoritmos fuera de línea.

Tópicos avanzados

Herramientas básicas

Práctica 12: Uso de rosbag

Modificar nodo **moving_sphere** para recibir su trayectoria de desplazamiento del tópico **/trajectory**.

- Crear un nodo que genere el tópico **/trajectory**.
- Guardar tópico generado en una bolsa de datos.
- Reproducir bolsa de datos junto al nodo **crazy_sphere**.

Tópicos avanzados

Conexión entre equipos

Conexión por red local

Ros permite la interconexión entre varias computadoras a travéz de una red local, para lo cual es necesario definir las variables de entorno siguientes:

- **ROS_MASTER_URI:** Define la dirección IP del maestro.
- **ROS_HOSTNAME, ROS_IP:** Dirección IP de la computadora local.

las variables son añadidas al archivo **.bashrc** de ubuntu, a fin de que la configuración se cargue con cada nueva terminal.

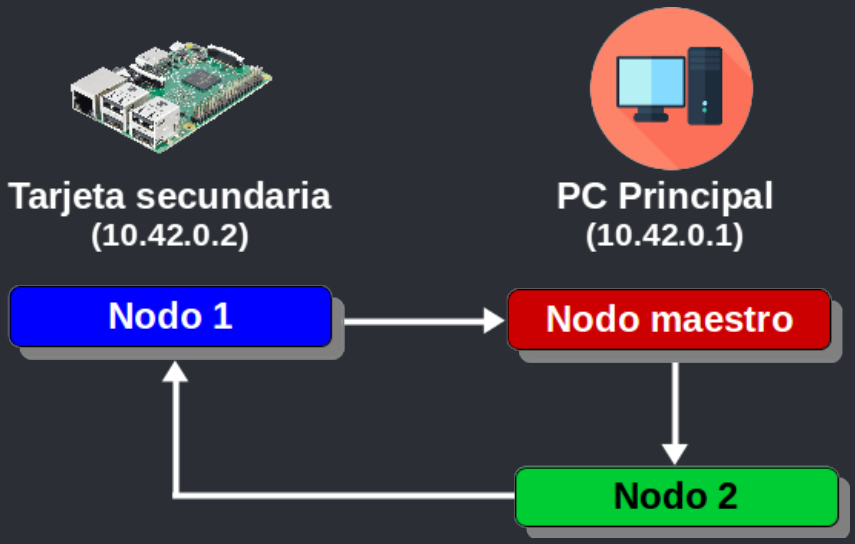
Tópicos avanzados

Conexión entre equipos

```
#Configurar IP 10.42.0.1 sobre maestro 10.42.0.2
export ROS_IP=10.42.0.1
export ROS_HOSTNAME=10.42.0.1
export ROS_MASTER_URI=http://10.42.0.2:11311/
```

Tópicos avanzados

Conexión entre equipos



PC Principal (Maestro)

```
#Configuracion de la pc principal
export ROS_IP=10.42.0.1
export ROS_HOSTNAME=10.42.0.1
export ROS_MASTER_URI=http://10.42.0.1:11311/
```

PC Secundaria

```
#Configuracion de la pc secundaria
export ROS_IP=10.42.0.2
export ROS_HOSTNAME=10.42.0.2
export ROS_MASTER_URI=http://10.42.0.1:11311/
```

Conexión local entre computadoras usando ROS.

Tópicos avanzados

Conexión entre equipos

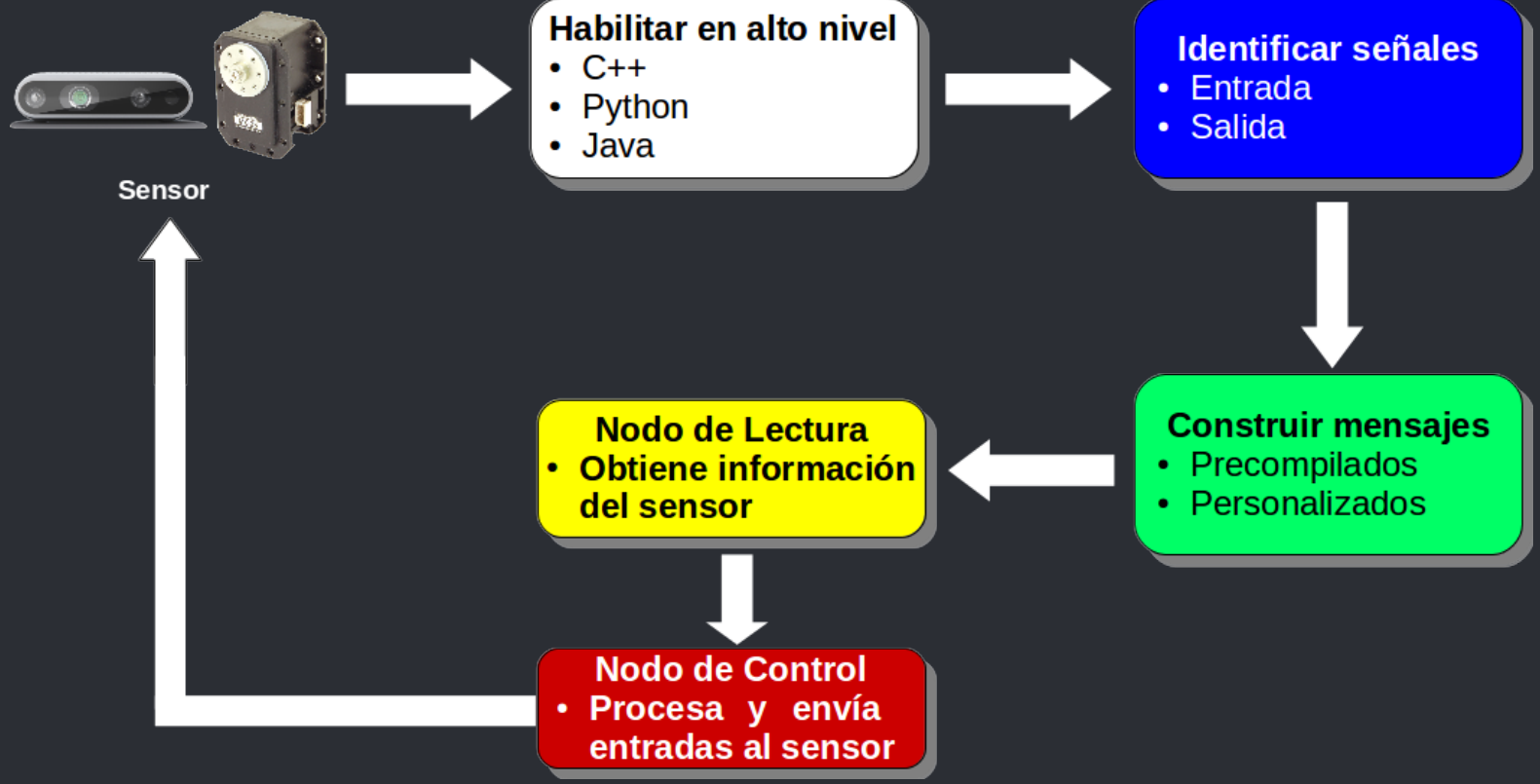
Práctica 13: Conexión entre equipos usando red local

Establecer comunicación entre dos equipos usando el framework ROS.

- Definir uno de los equipos con el nodo maestro, y otro como un nodo esclavo.
- Crear un publisher en el equipo maestro, y leerlo desde el esclavo.
- Aplicar el proceso inverso.

Tópicos avanzados

Habilitamiento de sensores



Habilitamiento de sensores.

Habilitamiento de sensores

Consideraciones

- Se requiere habilitar el nodo en un lenguaje de alto nivel (C++, Python, Java).
- Nodos escritos en diferentes lenguajes pueden comunicarse entre sí a travéz del nodo maestro.
- Se puede crear una librería, o utilizar alguna existente para habilitar el nodo.
- Se deben colocar las dependencias correspondientes en el archivo **CMakelists.txt**.