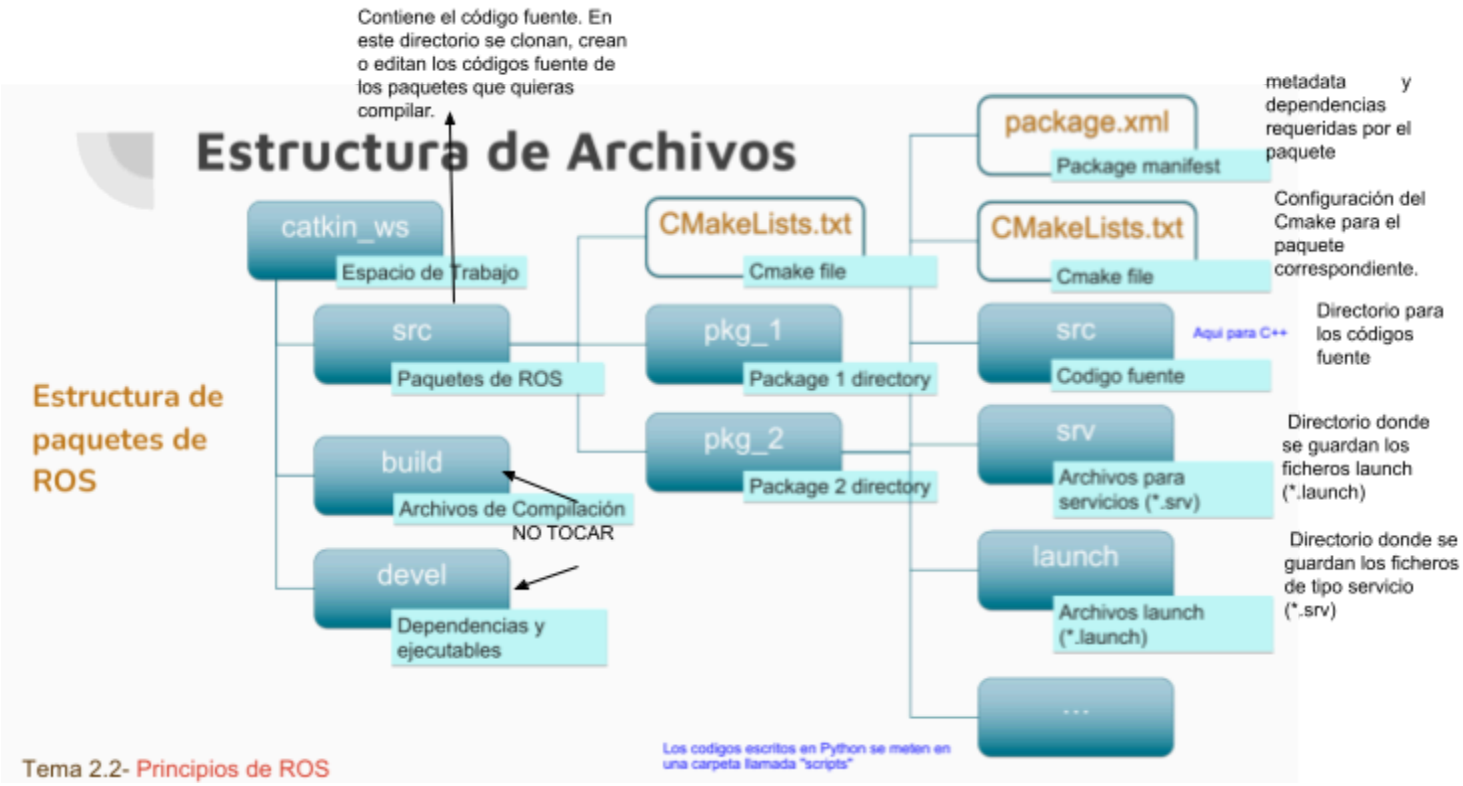
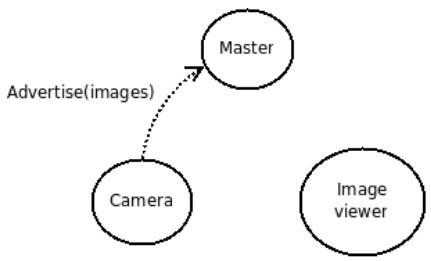
ROS: Robot Operating System

1. Principios de ROS:

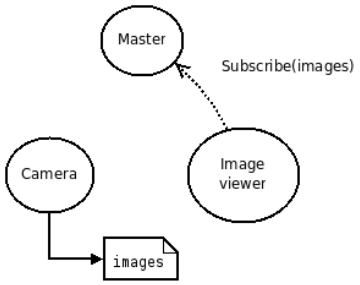


ROS Master:

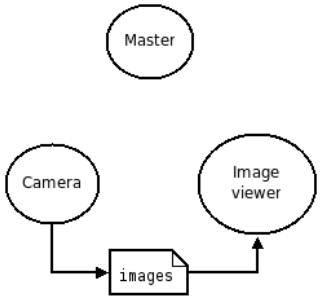
Proporciona servicios de nomenclatura y registro al resto de los nodos del sistema ROS. El master sirve para registrar los nodos, hace que cuando un publicador busque a un receptor, se cree un topic a través del cual puedan hablar. (La comunicación en ROS es asíncrona). Ejemplo:
Digamos que tenemos dos nodos: un nodo de cámara y un nodo de visor de imágenes. Una secuencia típica de eventos comenzaría con la cámara notificando al maestro que desea publicar imágenes sobre el tema "imágenes":



Ahora, Camera publica imágenes en el tema "imágenes", pero nadie se ha suscrito a ese tema todavía, por lo que no se envían datos. Ahora, Image_viewer quiere suscribirse al tema "imágenes" para ver si hay algunas imágenes allí:



Ahora que el tema "imágenes" tiene un publicador y un suscriptor, el nodo maestro notifica a Camera y a Image_viewer sobre la existencia de cada uno para que puedan comenzar a transferir imágenes entre sí:



Comandos:

- **roscore:** este comando lanza:
    - Un ROS Master
    - Un ROS Parameter Server
    - Un nodo de registro rosout

(un fichero launch, por defecto, lanza solo el roscore en caso de que no esté ya iniciado)

Nodos:

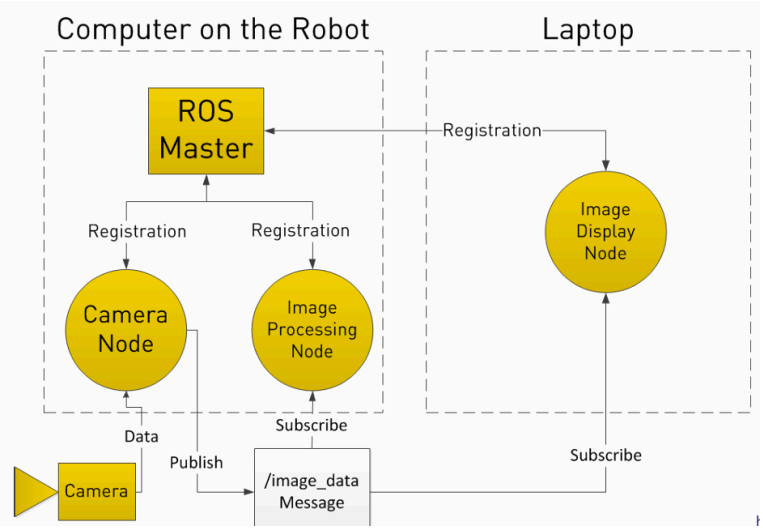
Es un proceso que realiza cálculos. Estos nodos se combinan en un gráfico y se comunican entre sí mediante topics. Los nodos están destinados a funcionar a una escala detallada, por lo que un sistema de control de un robot generalmente comprenderá muchos nodos, para que sean sencillos e ir interconectando entre ellos (similar a funciones en programación).

- **Publicador (Publisher)**: Un nodo que genera y envía mensajes a un *topic* se llama publicador. Un nodo puede publicar en uno o varios *topics*.
- **Suscriptor (Subscriber)**: Un nodo que recibe mensajes de un *topic* se llama suscriptor. Un nodo puede suscribirse a uno o varios *topics*.

(Un nodo puede ser a la vez publisher y subscriber sin ningún problema)

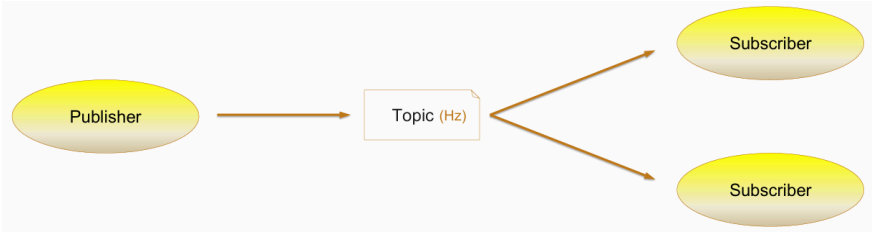
Comandos:

- **rosnode info <node-name> :** muestra información sobre el nodo.
- **rosnode kill <node-name> :** finaliza el nodo activo. (sirve para parar los nodos en el .launch por ejemplo, para ir parándolos uno a uno)
- **rosnode list <namespace> :** muestra todos los nodos activos.
- **rosrun <package-name> <node-name> :** lanza un nodo ROS.



Topics:

Son buses a través de los cuales se intercambian los mensajes entre nodos. En general, los nodos no saben con quién se están comunicando. Sin embargo, los nodos que están interesados en datos se suscriben al topic deseado, y los nodos que generan datos publican en el topic correspondiente. Además, puede haber varios publishers y subscribers a un mismo topic.



Comandos:

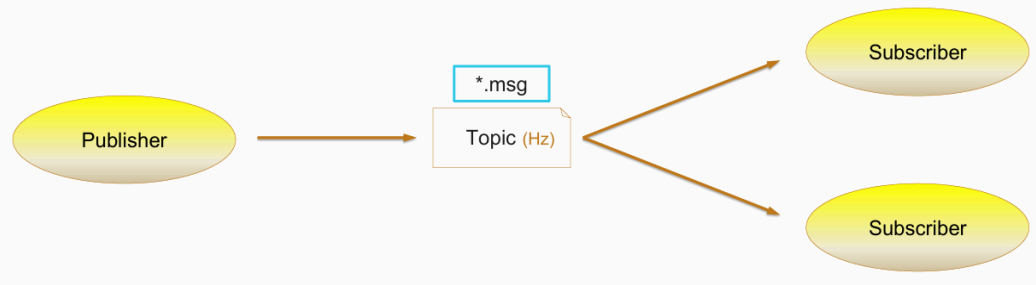
- **rostopic info <topic-name> :** muestra información sobre el topic
- **rostopic list <namespace> :** muestra todos los nodos activos.
- **rostopic echo <topic-name>** : muestra los mensajes publicados en ese topic.

(el resultado del comando rostopic echo -n1 /mitopic muestra únicamente un mensaje de ese topic.)

- **rostopic pub <topic-name> <topic-type> [data] :** publica datos sobre ese topic

Mensajes:

Los nodos se comunican entre ellos publicando mensajes en topics. Un mensaje es simplemente una estructura de datos creada para enviar la información deseada.



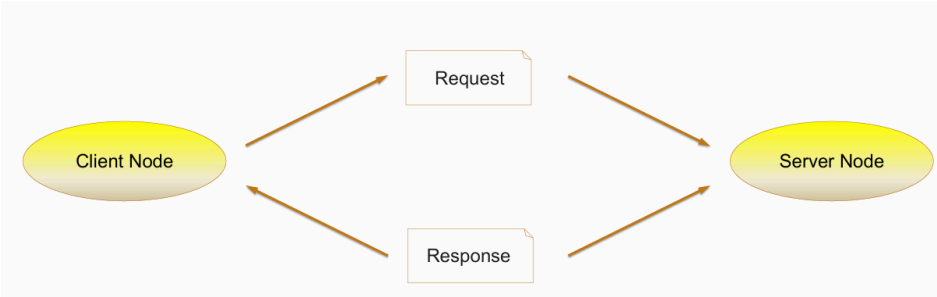
Comandos:

- **rosmsg show <msg-type> :** muestra información sobre ese mensaje.
- **rosmsg list :** muestra todos los mensaje.
- **rosmsg package <package-name> :** muestra todos los mensajes en un paquete.

(los comandos rosrun, rostopic y rosmsg se pueden ejecutar desde cualquier ruta)

Servicios:

Aunque la estructura publisher/subscriber es muy flexible, hay casos en los que se requiere de una comunicación de tipo petición/respuesta. Este tipo de comunicación se hace a través de los servicios. Son un mecanismo de comunicación síncrona que permite a un nodo solicitar una tarea específica a otro nodo y esperar una respuesta. A diferencia de los *topics*, que se utilizan para la comunicación asíncrona y continua, los servicios se emplean cuando se necesita una interacción puntual en la que un nodo envía una solicitud y otro nodo responde con un resultado.



Comandos:

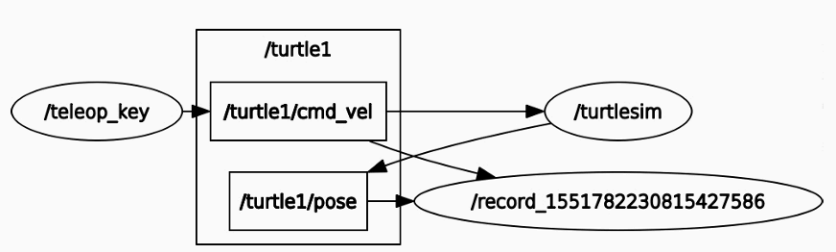
- **rosservice args <service-name> :** muestra los argumentos de un servicio.
- **rosservice call <service-name> [service-args] :** Llama a un servicio desde la línea de

comandos.

- **rosservice list :** muestra todos los servicios disponibles.
- **rosservice info <service-name> :** muestra la información de un servicio.
- **rosservice type <service-name> :** muestra el tipo de un servicio

ROSBag:

Un bag es un formato de archivo para almacenar la información de los mensajes que se mandan. Estos ficheros se crean principalmente a través de la herramienta rosbag , que se suscribe a uno o mas topics y almacena los mensajes de datos de forma consecutiva. Este fichero se usa para reproducir lo que ha ocurrido durante una experimentación y también para poder procesar los datos adquiridos, analizar o visualizar estos datos.
Según ChatGpt: Un **bag** en ROS (Robot Operating System) se refiere a un archivo de tipo `.bag`, que es un contenedor de datos utilizados para almacenar mensajes de ROS que se han registrado en tiempo real durante la operación de un sistema robótico. Estos archivos permiten guardar el estado completo de un sistema en funcionamiento, incluyendo los datos de sensores, comandos, y otros tipos de mensajes que circulan por los *topics*.



Comandos:

- **rosbag record <topic-names> :** guarda un fichero bag con el contenido de los topic seleccionados.
- **rosbag record-a :** guarda todos los topics.
- **rosbag info <bag-file> :** muestra el resumen del contenido del fichero bag.
- **rosbag play <bag-file> :** Reproduce (publica) el contenido del fichero bag dado.

ParamServer:

El Parameter Server es un diccionario multivariable compartido al que pueden acceder todos los nodos de la red. Este servidor se utiliza para datos estáticos, como parámetros de configuración. Es una estructura centralizada que almacena pares clave-valor, donde los nodos pueden leer y escribir parámetros. Estos parámetros son configuraciones globales o específicas que pueden influir en el comportamiento de los nodos. Por ejemplo:

/camera/left/name: leftcamera
/camera/left/exposure: 1

/camera/right/name: rightcamera
/camera/right/exposure: 1.1

Comandos:

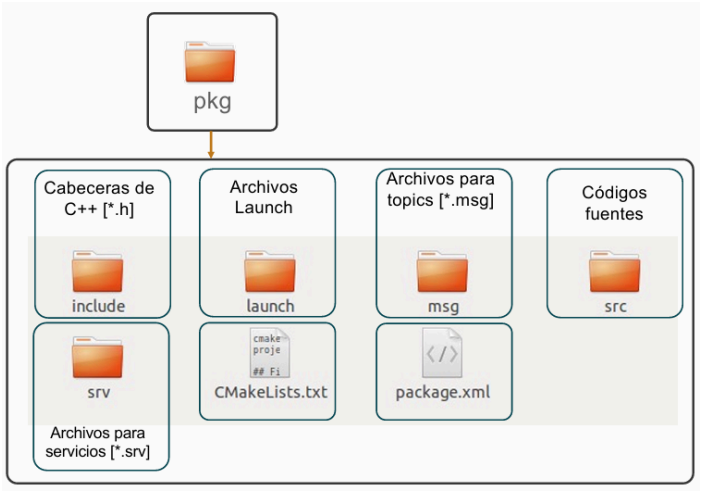
● **rosparam list <namespace> :** muestra todos los parámetros información sobre ese mensaje.

● **rosparam get <parameter-name> :** muestra el valor del parámetro.
● **rosparam set <parameter-name> [parameter-value] :** establece el valor de un parámetro.
● **rosparam delete <parameter-name> :** elimina un parámetro.

2. Primeros pasos:

Paquetes ROS:

El software de ROS se organiza en paquetes. Un paquete puede contener nodos, librerías, dataset, archivos de configuración, o cualquier cosa que sea útil para el paquete. El objetivo de estos paquetes es proporcionar esta funcionalidad útil de una manera fácil de consumir para que el software se pueda reutilizar fácilmente.



Comandos:

● **catkin_create_pkg <package_name> [depend1] [depend2] [depend3] :** crea un paquete ROS con los archivos necesarios y sus respectivas dependencias. El paquete se crea en la ruta de archivos desde la que se lanza.
● **roscd <package_name> :** ‘cd’ al paquete ROS especificado.

Mensajes (*.msg):

ROS utiliza un lenguaje de descripción de mensajes simplificado para describir los valores de datos que publican los nodos ROS. Las descripciones de los mensajes se almacenan en archivos .msg en el subdirectorio msg/ de un paquete ROS. Tiene dos partes importantes:

- Campos(field): son los datos que se envían dentro del mensaje.
  - Tipo: tipo de dato que se va a enviar a través de ese mensaje.
  - Nombre: nombre correspondiente.
- Constantes (constants): definen valores útiles que se puedan utilizar p sos campos (por ejemplo, constantes tipo enumeración para un valor entero).
  - Tipo: tipo de dato constante que se va a enviar a través de ese mensaje.
  - Nombre: nombre correspondiente.
  - Valor: Valor específico de dicha constante.

mi_mensaje.msg

```
int32 x
int32 y
```

```
int32 X=123
int32 Y=-123
```

Servicios (*.srv):

ROS utiliza un lenguaje de descripción de mensajes simplificado para describir los tipos de servicios ROS. Estas descripciones se almacenan en archivos .srv en el subdirectorio srv/ de un paquete ROS. Su estructura es muy similar a la de los ficheros de tipo mensaje. La descripción de un fichero de tipo servicio consiste en un mensaje request y uno response separados por '---':

mi_servicio.srv

```
string str
---
string str
```

mi_servicio2.srv

```
float32 x
float32 y
float32 theta
string name
---
string name
```

Launch (*.launch):

roslaunch es una herramienta para lanzar fácilmente múltiples nodos ROS, así como para configurar parámetros en el paramServer. Estos ficheros se almacenan en archivos .launch en el subdirectorio launch/ de un paquete ROS. Además, están escritos siguiendo un formato XML. Los ficheros *.launch lanzan el ROSMaster si este no está inicializado. Para lanzar este tipo de ficheros se utiliza el siguiente comando:

- roslaunch <pkg-name> <launch-filename> arg1:=value1 arg2:=value2 …

$ roslaunch mipaquete example.launch
$ roslaunch mipaquete example.launch arg:=value

Formato XML:

```
<launch>
 <!-- Load the urdf file in the param server variable robot_description if it wasnt loaded before-->
 <param name="robot_description" command="cat $(find pi_robot_pkg)/urdf/pi_robot_v2.urdf“ />
 <!-- Publish TF with robot_state_publisher-->
 <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher"
 respawn="false" output="screen“ >
 <remap from="/joint_states" to="/pi_robot/joint_states" />
 </node>
 </launch>
```

Launch (*.launch) - Etiquetas:

● **<launch> :** Etiqueta principal de cualquier roslaunch. Su único uso es contener el resto de elementos.

● **<node> :** Etiqueta para especificar el nodo que quieres lanzar. No se garantiza el orden de lanzamiento de los nodos.

○ Atributos:

■ **pkg=** “mypackage”: nombre del paquete donde se encuentra el nodo ROS.

■ **type** = “nodetype”: tipo del nodo que se debe corresponder al ejecutable correspondiente con el mismo nombre.

■ **name=** “nodename”: nombre con el que se desea lanzar el nodo.

■ **args** = “arg1 arg2 arg3” (opcional): argumentos necesarios para el nodo.

■ **respawn** = “true” (opcional): el nodo se reinicia automáticamente si se cierra.

■ **ns=** “cosa” (opcional): lanza el nodo en el espacio de trabajo (namespace) ‘cosa’.

```
<node name="listener1" pkg="rospy_tutorials" type="listener.py" args="--test" respawn="true" />
```

```
<node name="bar1" pkg="foo_pkg" type="bar" args="$(find baz_pkg)/resources/map.pgm" />
```

● **<remap> :** Permite ‘engañar’ a un nodo para que crea que se ha suscrito/está publicando a un topic específico, pero realmente está suscrito/publicando en otro. Este cambio afecta a los nodos que se lanzan después de la reasignación, no a los de antes. Esta etiqueta va dentro de la etiqueta <node>.

○ Atributos:

■ **from** = “original-name”: nombre del topic ROS que se va a reasignar.

■ **to** = “new-name”: nombre del topic ROS al que va a apuntar el nodo.

Esta reasignación se suele hacer en los nodos de tipo subscriber

● **<include> :** Permite importar otro roslaunch al fichero actual.

○ Atributos:

■ **file** = “$(find pkg-name)/path/filename.launch”: nombre del fichero a incluir.

■ **ns=** “cosa” (opcional): importa el fichero relativo al espacio de trabajo (namespace) ‘cosa’.

```
<include file="$(find simulation)/launch/spawn_car.launch">
```

● <group> : Etiqueta que permite aplicar configuraciones a un grupo determinado de nodos.
○ Atributos:
■ **ns=** “cosa”: asigna al grupo de nodos un espacio de trabajo (namespace) ‘cosa’.

```
<group ns="carl">
</group>
```

● **<arg> :** Permite declarar argumentos de entrada a tus nodos o ficheros.
○ Atributos:
■ **name=** “arg-name”: nombre del argumento.
■ **default=** “default-value” (opcional): valor por defecto del argumento.
■ **value=** “value” (opcional): valor del argumento. No puede ser combinado con el atributo default.

```
<include file="included.launch">
  <!-- all vars that included.launch requires must be set -->
  <arg name="hoge" value="fuga" />
</include>
```

```
<launch>
  <!-- declare arg to be passed in -->
  <arg name="hoge" />

  <!-- read value of arg -->
  <param name="param" value="$(arg hoge)"/>
</launch>
```

Compilación con Catkin:

Catkin es la herramienta oficial de ROS para compilar. Esta herramienta permite una mejor distribución de los paquetes que su sucesora, rosbuild. Es muy parecido a CMake pero añade una infraestructura automática para ‘encontrar los paquetes’ y compilar proyectos dependientes al mismo tiempo. Los comandos que más vamos a utilizar son:

● **catkin_make :** compila el espacio de trabajo. Se debe ejecutar desde la carpeta raíz de nuestro entorno de trabajo (catkin_ws). No debo ejecutarlo cada vez que he realizado alguna modificación sobre mi fichero *.py, siempre y cuando el paquete al que pertenece el nodo ya esté compilado con anterioridad.

●**catkin_make–DCATKIN_WHITELIST_PACKAGES="<pkg-name1>;<pkg-name2>":** compila únicamente los paquetes especificados.

● **catkin_make–DCATKIN_WHITELIST_PACKAGES="":** vuelve a compilar todos los paquetes.

● **catkin_make clean :** es la manera más segura para limpiar el espacio de trabajo. Elimina las carpetas /build y /devel.
● **source devel/setup.bash :** actualiza las dependencias del espacio de trabajo.

CMakeLists.txt:

Este fichero es la entrada para el sistema de compilación CMake. Describe cómo compilar el código y dónde instalarlo. El orden de configuración de este fichero **SI** que importa.

1. **cmake_mínimum_required() :** versión del CMake.
2. **project() :** nombre del paquete
3. **find_package() :** busca otros paquetes necesarios para la compilación.
4. **catkin_python_setup() :** habilita el soporte del módulo Python.
5. **add_message_files(), add_service_files(), add_action_files() :** añaden de los mensajes, servicios y acciones respectivamente.
6. **generate_messages() :** genera los mensajes, servicios y acciones.
7. **catkin_package() :** especifica la exportación de información de compilación del paquete.
8. **add_library()/add_executable()/target_link_libraries() :** librerías y ejecutables a compilar.
9. **install() :** reglas para la instalación.
10. **catkin_add_gtest() :** test de compilación

Package.xml:

Este fichero es el 'manifiesto del paquete', en formato XML, que debe incluirse en la carpeta raíz de cualquier paquete. Este fichero define propiedades sobre el paquete, como su nombre, el número de versión, los autores, o las dependencias.

```
<package format="2">
  <name>foo_core</name>
  <version>1.2.4</version>
  <description>
    This package provides foo capability.
  </description>
  <maintainer email="ivana@osrf.org">Ivana Bildbotz</maintainer>
  <license>BSD</license>

  <buildtool_depend>catkin</buildtool_depend>
</package>
```

Package.xml - Etiquetas requeridas:

● **<name>:** nombre del paquete.
● **<version> :** versión del paquete.
● **<description> :** descripción de los contenidos del paquete.
● **<maintainer> :** nombre de la/las personas que mantienen el paquete.
● **<license> :** licencia del software bajo las cuales se publica el código.

Package.xml - Etiquetas de dependencias:

● **<depend> :** especifica que la dependencia es una dependencia de compilación, ejecución y para exportar. Suele ser la mas usada.
● **<build_depend> :** especifica que la dependencia es una dependencia de compilación.
● **<build_export_depend> :** especifica que la dependencia es de compilación y para exportar.
● **<exec_depend> :** especifica que la dependencia es una dependencia de ejecución.
● **<buildtool_depend> :** especifica la dependencia con respecto a la herramienta de compilación. En nuestro caso siempre será catkin

Programación de un Nodo en Python:

Para la programación de un nodo se han de tener en cuenta las siguientes consideraciones:
● Deben estar contenidos en un paquete (catkin_create_pkg primer_paquete rospy std_msgs).
● Por convenio, como son ficheros de código Python, los almacenaremos en una carpeta que se llame scripts.
● El fichero *.py debe tener permisos de ejecución (chmod +x *.py).
● Se deben lanzar los nodos mediante el comando rosrun (nunca mediante python).
● El paquete debe estar compilado mediante catkin.

Subscriber:

nodo_recibo.py

catkin_ws > src > robot_v1 > scripts > nodo_recibo.py > ...

```
Set as interpreter
#!/usr/bin/env python

import rospy
from primer_paquete.msg import miMensaje

def callback(data):
    rospy.loginfo("Recibo: %s - x: %s; y: %s" % (data.nombre, data.x, data.y))

def nodo_recibe():
    rospy.init_node('nodo_recibe', anonymous=True)
    rospy.Subscriber("mi_topic", miMensaje, callback)
    rospy.spin()
if __name__ == '__main__':
    nodo_recibe()
```

Publisher:

nodo_envia.py

catkin_ws > src > robot_v1 > scripts > nodo_envia.py > ...

Set as interpreter

```python
#!/usr/bin/env python
import rospy
from primer_paquete.msg import miMensaje

def nodo_envia():
    mensaje = miMensaje()
    mensaje.x = 0
    mensaje.y = 0
    pub = rospy.Publisher('/mi_topic', miMensaje, queue_size=10)
    rospy.init_node('nodo_envia', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        mensaje.x += 1
        mensaje.y += 2
        envio_str = "Envio: %s - x: %s; y: %s" % (mensaje.nombre, mensaje.x, mensaje.y)
        rospy.loginfo(envio_str)
        pub.publish(mensaje)
        rate.sleep()

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

No lo pondré como nuevo apartado, pero mirar el pdf 2.4, que va de herramientas gráficas:

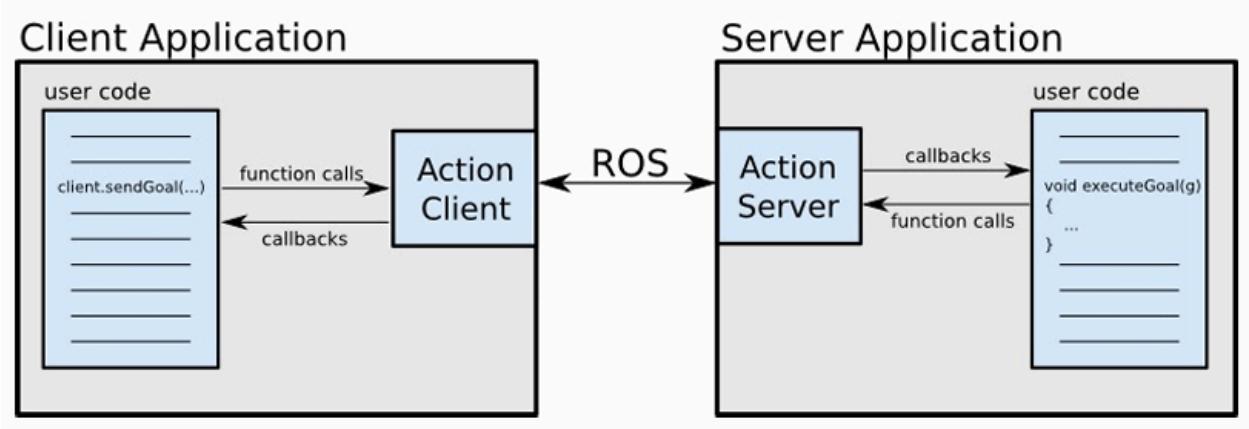
- **TF**: para diferentes sistemas de referencia
- **rqt**: framework de software ROS que implementa diversas herramientas GUI en forma de complementos.
- **rviz**: herramienta muy útil de visualización 3D de ROS
- **Gazebo**: interfaz necesaria para simular un robot utilizando mensajes ROS, servicios y reconfiguración dinámico.
- **Simulación Turtlebot**: Turtlebot 3: para Melodic y Noetic.

3. Acciones:

Especificaciones:

Las acciones (actionlib) proporcionan una herramienta para crear servidores que ejecutan objetivos de larga duración que se pueden interrumpir. También proporciona una interfaz de cliente para enviar solicitudes al servidor. El cliente y el servidor se comunican mediante un protocolo desarrollado para este fin, basado en mensajes ROS. (no es un mensaje estándar, es un mensaje especial con un formato especial)

Son un mecanismo de comunicación asíncrona que se utiliza para tareas que pueden tardar un tiempo indeterminado en completarse y que podrían necesitar feedback durante su ejecución. Las acciones son particularmente útiles cuando un nodo necesita realizar una tarea compleja o de larga duración, como mover un brazo robótico a una posición específica, realizar un proceso de navegación, o ejecutar un reconocimiento de objetos.



Para la comunicación entre cliente y servidor, se define un fichero (*.action), colocado en la carpeta /action, donde se especifican tres tipos de mensajes:

- **Goal:** Tipo de mensaje para enviar, desde el cliente, el objetivo deseado.
- **Result:** Tipo de mensaje para enviar, desde el servidor, el resultado final de la acción realizada.
- **Feedback:** Tipo de mensaje para enviar, desde el servidor, información sobre el progreso realizado.

```
# Define the goal
uint32 dishwasher_id  # Specify which dishwasher we want to use
--- ← separacion para cada tipo de mensaje
# Define the result
uint32 total_dishes_cleaned
---
# Define a feedback message
float32 percent_complete
```

En el paquete en el que vayamos a crear una acción, debemos añadir estas dependencias en el fichero CMakeLists.txt y en el fichero package.xml.

CMAKE

```
find_package(catkin REQUIRED genmsg actionlib_msgs)
add_action_files(DIRECTORY action FILES DoDishes.action)
generate_messages(DEPENDENCIES actionlib_msgs)
```

XML

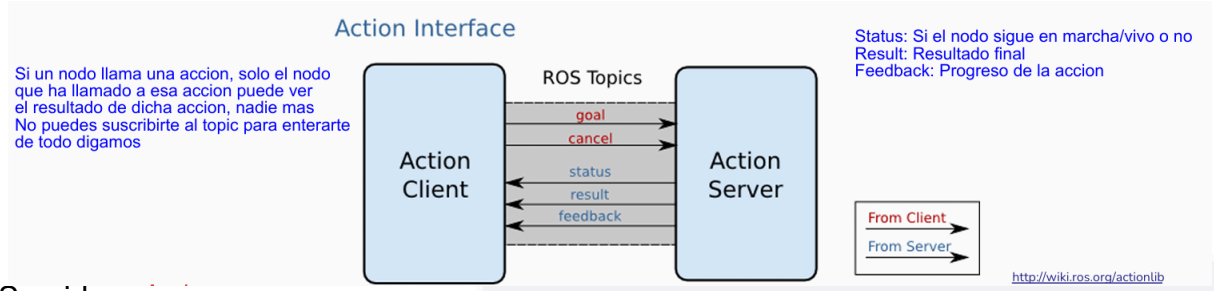
```
<depend>actionlib</depend>
<depend>actionlib_msgs</depend>
```

Tras la compilación de la acción, se generan una serie de mensajes, basados en el fichero *.action, para la comunicación cliente-servidor. Estos mensajes se generan internamente por el fichero genaction.py, ya definido en ROS. Estos mensajes son:

- MiAccionAction.msg
- MiAccionActionGoal.msg
- MiAccionActionResult.msg
- MiAccionActionFeedback.msg
- MiAccionGoal.msg
- MiAccionResult.msg
- MiAccionFeedback.msg

Todas las comunicaciones entre cliente y servidor se realizan a través de topics.

- El **action server** ofrece una acción que puede ser llamada por otros nodos.
- El **action client** permite a un nodo enviar una acción a otro nodo de tipo action server.



Servidor:

- Se encarga de realizar la acción que se le pide.
- Envía mensajes sobre el estado en el que se encuentra la acción (**status**).
- Envía mensajes sobre el resultado alcanzado (**result**).
- Envía mensajes con información sobre el estado del robot (**feedback**)

```
import roslib
roslib.load_manifest('my_pkg_name')
import rospy
import actionlib

from chores.msg import DoDishesAction

class DoDishesServer:
  def __init__(self):
    self.server = actionlib.SimpleActionServer('do_dishes', DoDishesAction, self.execute, Fals
e)
    self.server.start()

  def execute(self, goal):
    # Do lots of awesome groundbreaking robot stuff here
    self.server.set_succeeded()


if __name__ == '__main__':
  rospy.init_node('do_dishes_server')
  server = DoDishesServer()
  rospy.spin()
```

Cliente:

- Puede cancelar la acción.
- Recibe mensajes sobre el estado en el que se encuentra la acción (**status**).
- Recibe mensajes sobre el

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

import roslib
roslib.load_manifest('my_pkg_name')
import rospy
import actionlib

from chores.msg import DoDishesAction, DoDishesGoal

if __name__ == '__main__':
    rospy.init_node('do_dishes_client')
    client = actionlib.SimpleActionClient('do_dishes', DoDishesAction)
    client.wait_for_server()

    goal = DoDishesGoal()
    # Fill in the goal here
    client.send_goal(goal)
    client.wait_for_result(rospy.Duration.from_sec(5.0))
```

resultado alcanzado (**result**).
● Recibe mensajes
con información sobre el estado del
robot (**feedback**).

Un objeto cliente tiene dos funciones que se pueden usar para conocer si la acción que se está realizando ha sido finalizada o no:

● **wait_for_result():** Esta función, cuando se le llama, espera a que la acción termine y devuelva un valor.

● **get_state():** Esta función, cuando se le llama, devuelve un entero que indica en qué estado (status) de la acción que se está realizando.

○ 0->pendiente.
○ 1->activo.
○ 2->realizado.
○ 3->advertencia.
○ 4->error.

Extras:

#!usr/bin/env python sirve para especificar el programa que va a interpretar el código

El fichero **.bashrc** Se encuentra en nuestro directorio raíz y sirve para definir las configuraciones de la sesión de la terminal

Un mensaje se importa para poder ser usado en nuestro nodo con from .msg import