

ROBÓTICA
CUARTO CURSO DEL GRADO EN
INGENIERÍA INFORMÁTICA



Práctica 5

ROS: TOPICS

F. Gómez Bravo
R. López de Ahumada
José Manuel Lozano

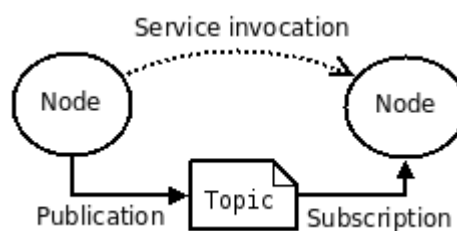
En esta práctica se describe como crear un topic, un nodo emisor o publicador, un nodo suscriptor y como crear un servicio y su correspondiente servidor y cliente

1. Introducción

En ROS el Maestro (*Máster*) actúa como un servicio de nombres. El Maestro se encarga de almacenar/registrar los Temas o *Topics* y los servicios para que los Nodos hagan uso de ellos. Los Nodos se comunican con el Maestro para indicar la información que desean registrar. Así mismo, el Maestro ofrece información a los Nodos sobre otros nodos registrados y como se deben de hacer las conexiones adecuadamente. El Maestro también hará llamada a estos nodos cuando se ejecutan estos cambios en la información de registro, lo que permite que los nodos para crear dinámicamente las conexiones como nuevos nodos.

Los Nodos se conectan a otros nodos directamente, mientras que el Maestro sólo proporciona información de búsqueda, al igual que un servidor DNS. Los Nodos que se subscriben a un tema solicitarán las conexiones a los nodos que publican sobre ese Tema o *Topic*, y se establecerá que la conexión a través de un acuerdo sobre el protocolo de conexión. El protocolo más común que se utiliza en un ROS se llama TCPROS, que utiliza sockets TCP / IP estándar.

En la siguiente imagen se puede observar gráficamente como sería la comunicación entre dos Nodos diferentes que publican información o se subscriben a un *Topic*, todo en la parte inferior de la imagen. En la parte superior vemos como se produce la comunicación de dos nodos mediante una petición/respuesta (Request/Response) de un servicio.

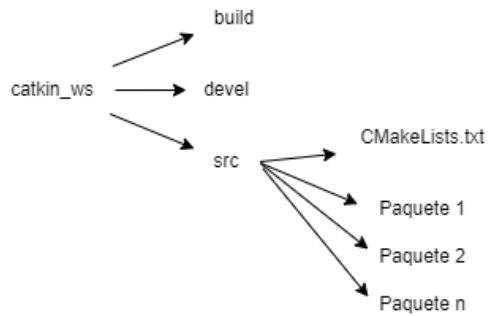


2. Creación de un topic, nodo publicador o emisor y subscriber

Para comenzar la creación de un topic y sus nodos, se describirá la estructura de ficheros necesarias para crear un paquete, sus topics y sus nodos correspondientes. Luego se expondrá como crear un topic, un nodo emisor para ese topic y un nodo subcriptor utilizando Python.

2.1. Creación de paquetes

En primer lugar, hay que recordar que nodos son software contenidos en paquetes. Para comenzar a crear paquetes primero se debe crear un espacio de trabajo (workspace). Este workspace deberá contener una carpeta llamada *src*. En *src* es donde se almacenarán todos los paquetes a utilizar en el entorno de trabajo. Un ejemplo gráfico se recoge en la siguiente figura.



El primer paso es crear la carpeta contenedora del paquete para ello se hace uso de la siguiente instrucción:

```
turtlebot@rosindigo:~$ mkdir -p catkin_ws/src
```

Posteriormente a esto, se procederá a crear un paquete dentro de la carpeta *src*. Este paquete será creado mediante la herramienta *catkin_create_pkg*. Catkin es una herramienta de líneas de comando que facilita la creación de entornos de trabajo, paquetes y etc., así como facilita el flujo de trabajo a la hora del desarrollo de paquetes. A continuación, se procederá a crear el primer paquete del entorno de trabajo a modo de prueba. Para ello se debe de hacer uso de la instrucción *catkin_create_pkg nombre_paquete [dependencias necesarias]*

```
turtlebot@rosindigo:~$ cd catkin_ws/src
```

```
turtlebot@rosindigo:~/catkin_ws/src$ catkin_create_pkg paq_inicial std_msgs rospy
```

```
turtlebot@rosindigo:~/catkin_ws/src$ catkin_create_pkg paq_inicial std_msgs rospy
Created file paq_inicial/CMakeLists.txt
Created file paq_inicial/package.xml
Created folder paq_inicial/src
Successfully created files in /home/turtlebot/catkin_ws/src/paq_inicial. Please adjust the values in package.xml.
turtlebot@rosindigo:~/catkin_ws/src$
```

Resultado 1: lanzamiento de *catkin_create*

En el paso anterior, el nombre del paquete es *paq_inicial* y las dependencias son *std_msgs*, la cual establece la base para poder crear posteriormente los mensajes que transportarán la información compartida por los nodos. Además, incluye como dependencia también la librería *rospy* que permite la creación de funciones del nodo emisor (*Publisher*), nodos suscriptor (*Subscriber*), cliente o servidor de servicios en el lenguaje de programación Python.

Al finalizar la ejecución de este paso, en la carpeta *src* se habrán creado los siguientes ficheros *CMakeLists.txt* y *package.xml*, así como las carpetas *include* y *src*. Los ficheros *CMakeLists.txt* y *package.xml* incluyen información importante para el correcto desarrollo del paquete, como son las dependencias del paquete. En *package.xml* además de las dependencias también incluye información sobre el autor/autores del paquete, una breve descripción del paquete y la licencia bajo la cual ha sido desarrollado. Si se desea modificar algunas de las partes mencionadas del fichero, esto se puede realizar mediante cualquier editor de texto.

En este punto, ya se puede compilar por primera vez el entorno de trabajo y por tanto también el paquete que se está creando. Para ello se hace uso de la instrucción *catkin_make*. Esta

instrucción debe de ejecutarse en la carpeta raíz del entorno, en el ejemplo la carpeta titulada *catkin_ws*.

```
turtlebot@rosindigo:~/catkin_ws/src$ cd ..  
turtlebot@rosindigo:~/catkin_ws$ catkin_make
```

```
turtlebot@rosindigo:~/catkin_ws/src$ cd ..  
turtlebot@rosindigo:~/catkin_ws$ catkin_make  
Base path: /home/turtlebot/catkin_ws  
Source space: /home/turtlebot/catkin_ws/src  
Build space: /home/turtlebot/catkin_ws/build  
Devel space: /home/turtlebot/catkin_ws/devel  
Install space: /home/turtlebot/catkin_ws/install  
Creating symlink "/home/turtlebot/catkin_ws/src/CMakeLists.txt" pointing to "/opt/ros/indigo/share/catkin/cmake/toplevel.cmake"  
####  
### Running command: "cmake /home/turtlebot/catkin_ws/src -DCATKIN_DEVEL_PREFIX=/home/turtlebot/catkin_ws/devel -DCA  
MAKE_INSTALL_PREFIX=/home/turtlebot/catkin_ws/install -G Unix Makefiles" in "/home/turtlebot/catkin_ws/build"  
####  
-- The C compiler identification is GNU 4.8.2  
-- The CXX compiler identification is GNU 4.8.2  
-- Check for working C compiler: /usr/bin/cc  
-- Check for working C compiler: /usr/bin/cc -- works  
  
-- Found Threads: TRUE  
-- Found gtest sources under '/usr/src/gtest': gtests will be built  
-- Using Python nosetests: /usr/bin/nosetests-2.7  
-- catkin 0.6.11  
-- BUILD_SHARED_LIBS is on  
--  
-- traversing 1 packages in topological order:  
-- - paq_inicial  
--  
-- +++ processing catkin package: 'paq_inicial'  
-- ==> add_subdirectory(paq_inicial)  
-- Configuring done  
-- Generating done  
-- Build files have been written to: /home/turtlebot/catkin_ws/build  
####  
### Running command: "make -j1 -l1" in "/home/turtlebot/catkin_ws/build"  
####  
turtlebot@rosindigo:~/catkin_ws$
```

Resultado 2: de la compilación del entorno de trabajo (resultados parciales)

Una vez compilado el espacio de trabajo del nuevo paquete se debe incluir la ruta del espacio de trabajo a las rutas de ejecución de la terminal. Esto se realiza mediante la instrucción *source ruta/setup.bash*. Esta instrucción debe ser ejecutada cada vez que se abre una nueva terminal para que se introduzca la ruta deseada del workspace a la ruta de ejecución.

```
turtlebot@rosindigo:~/catkin_ws$ source devel/setup.bash
```

Una vez alcanzado este punto, ya debería de estar disponible el nuevo paquete creado llamado *paq_inicial* en ROS. Para comprobarlo se puede probar a ejecutar la instrucción *roscd paq_inicial*. La instrucción *roscd* es la unión de *ros* y *cd* de Linux, lo cual permite mover entre los directorios de los paquetes. Así mismo existen otras operaciones con *ros* y comandos de Linux como son *rosls=ros + ls*, *rospack=ros + pack* y *roscp=ros + cp*.

```
turtlebot@rosindigo:~/catkin_ws$ roscd paq_inicial
```

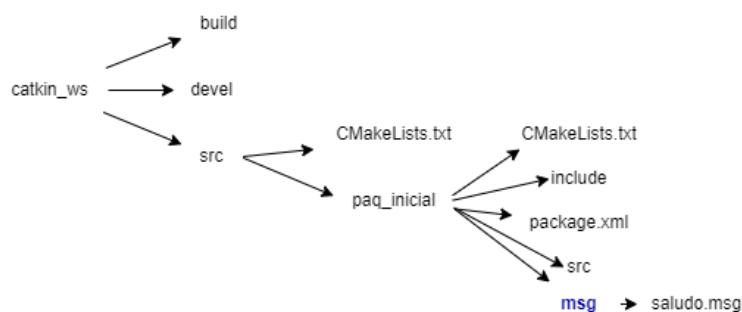
```
turtlebot@rosindigo:~/catkin_ws$ roscd paq_inicial  
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$
```

Resultado 3: moverse al nuevo directorio

2.2. Creación de un mensaje

El nuevo paquete que se ha creado en el punto anterior actualmente no tiene ningún tipo de mensaje asociado. Por ello, se debe crear un nuevo mensaje.

En primer lugar, se creará un mensaje orientado para ser enviado por un nodo *Publisher* o emisor y que sea recibido por un nodo *Subscriber* o subscriptor. El mensaje estará compuesto por un número entero, el cual será usado como contador para conocer cuantas veces ha saludado el nodo *Publisher* al resto de nodos subscriptores. Para poder lograr esto, se necesita crear un directorio llamado *msg* donde se almacenarán todos los mensajes que podrá utilizar el paquete creado. En esta carpeta se creará un fichero que contendrá los campos del mensaje, el fichero se nombrará como *saludo.msg*. En la siguiente figura se puede observar la distribución de los directorios y donde se ubica el mensaje de saludo.msg.



```
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ mkdir msg
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ vi msg/saludo.msg
```

Con las instrucciones anteriores se realiza la creación de la carpeta y se crea el fichero *saludo.msg* en la carpeta *msg*. El contenido del fichero *saludo.msg* debe ser *int64 num* como se observa más abajo.



Resultado 4: información a incluir en el fichero *saludo.msg*

Nota: para salir de vi una vez se ha finalizado la escritura en el fichero se realiza pulsando escape y luego escribiendo :wq para guardar y :x para salir sin guardar.

Para comprobar si se ha creado correctamente el mensaje que se enviará se puede comprobar mediante la instrucción *rosmg show saludo*. La instrucción *rosmg* permite mostrar información relacionada con los diferentes mensajes implementados en *ros*. En concreto la instrucción propuesta, muestra la información relativa al mensaje creado llamado *saludo*, básicamente debería mostrar el paquete que lo contiene, seguido del nombre del paquete y el contenido del paquete.

```
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ rosmmsg show saludo
```

```
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ rosmmsg show saludo
[paq_inicial/saludo]:
int64 num

turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$
```

Resultado 5: resultado `rosmmsg show saludo`

Como se puede observar en el anterior resultado, primero muestra `[paq_inicial/saludo]`, lo cual corresponde al paquete y al nombre del mensaje. Luego aparece `int64 num`, lo cual representa al contenido del mensaje. Este mensaje es muy simple, sin embargo, los mensajes pueden contener tantos campos como sean necesarios.

2.3. Diseño de un Publisher o emisor y un Subcriptor de ROS

Una vez acabada la creación del mensaje, se procede a diseñar el código para poder lanzar un nodo Publisher o emisor y un nodo Subscriber o subcriptor. El nodo emisor podrá enviar los mensajes de forma periódica o de forma puntual. Primero se diseñará el software para crear un nodo emisor y posteriormente se diseñará el nodo receptor. Este software puede ser diseñado en Python, ya que este lenguaje ha sido indicado en las restricciones del fichero `package.xml`.

Para desarrollar el software de Python hay que crear una nueva carpeta en el directorio `paq_inicial` llamada `scripts`. En esta carpeta se almacenarán todos los ficheros ejecutables por Python. Una vez creada la carpeta, se procede a diseñar el software del emisor, para ello se hace uso de la instrucción `gedit scripts/emisor.py &` y se inserta el siguiente código.

```
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ mkdir scripts
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ gedit scripts/emisor.py &
```

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

import rospy
from std_msgs.msg import String

def emisor():
    pub = rospy.Publisher('comunicacionTopic', String, queue_size=10)
    rospy.init_node('emisor', anonymous=True)
    rate = rospy.Rate(1) # 1hz o 1 vez por segundo
    entero=1
    while not rospy.is_shutdown():
        hola_str = "Hola %s" % entero
        rospy.loginfo(hola_str)
        pub.publish(hola_str)
        rate.sleep()
        entero=entero+1
    if __name__ == '__main__':
        try:
            emisor()
        except rospy.ROSInterruptException:
            pass
```

La descripción del código es la siguiente:

En primer lugar, se indica que el código debe ser ejecutado como un script de Python.

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

Luego se realiza una llamada a las librerías de las que depende el código, en este caso *rospy* y *std_msgs*.

```
import rospy
from std_msgs.msg import String
```

Posteriormente, se declara la función emisor. En ella se hace uso de la función *rospy.Publisher('comunicacionTopic', String, queue_size=10)*, esta función indica que el nodo actuará como emisor o *Publisher* del topic o tema “comunicacionTopic”, mediante un mensaje de tipo *String* que ha sido importado anteriormente. *Queue_size* indica el tamaño de la cola de mensajes que se podrá almacenar para un nodo receptor que este registrado, pero no esté recibiendo los mensajes. Además, se indica que el nombre del nodo es “emisor”, el uso de la variable “*anonymous=True*” evita que existan nodos con nombre repetidos. Si se repite el nombre del nodo con esta variable se permite añadir número detrás del nombre para evitar su repetición. Es importante tener esta configuración en la variable *anonymous*, ya que, si no es así y se repite el nombre de un nodo, el nodo anterior es eliminado y solo queda vivo el nuevo nodo creado.

```
pub = rospy.Publisher('comunicacionTopic', String, queue_size=10)
rospy.init_node('emisor', anonymous=True)
```

Las siguientes instrucciones se utilizan para establecer el número de veces por segundo que van a emitir los datos, así como se inicializa el contador de saludos.

```
rate = rospy.Rate(1) # 1hz o 1 vez por segundo
entero=1
```

Luego, las siguientes instrucciones se ejecutan hasta que se finalice el proceso donde se esté ejecutando el nodo emisor. En este bucle se creará un string con la palabra “Hola y el número de saludos”, esto se mostrará por la consola donde esté en ejecución el nodo. También, se enviará o publicará esta información para que sea recibida por todos los nodos que estén suscritos al topic “comunicacionTopic”. Finalmente, el proceso entrará en reposo por el tiempo establecido en *rate.sleep()* y sumará uno al contador de saludos.

```
while not rospy.is_shutdown():
    hola_str = "Hola %s" % entero
    rospy.loginfo(hola_str) # Muestra por consola
    pub.publish(hola_str) # Publica en la red de nodos
    rate.sleep()
    entero=entero+1
```

Por último, la ejecución del código de la función emisor se incluye en un bloque try-except con el fin de capturar las posibles excepciones generadas en el proceso de comunicación.

Una vez finalizada la implementación del emisor, se procede a implementar el software del receptor. Este software es más simple que el anterior, debido a que la importación de librerías

es igual y tan solo hay que iniciar el nodo y suscribirlo al topic y mostrar por pantalla la información recibida. El software del nodo subscriber es el siguiente:

```
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ gedit scripts/subscriber.py &
```

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

import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "Yo escucho: %s", data.data)

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

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

La función *rospy.Subscriber* necesita como parámetros el tema o Topic al cual se quiere suscribir, el tipo de mensaje y una función de callback, a la cual se enviarán los mensajes que sean recibidos por el subscriber. En este caso la función de callback muestra por pantalla el mensaje recibido. Finalmente, la función *rospy.spin()*, indica que el programa se mantiene sin finalizar hasta que la ejecución del nodo sea detenida.

Además, es necesario dar permisos de ejecución a los ficheros creados esto se puede realizar con *chmod +x scripts/emisor.py* y con *chmod +x scripts/subscriber.py* o bien *chmod +x scripts/**.

```
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ chmod +x scripts/emisor.py
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ chmod +x scripts/subscriber.py
```

Una vez llevada a cabo la modificación, es necesario ir a la carpeta raíz del espacio de trabajo *catkin_ws* y ejecutar la instrucción de compilación del entorno de trabajo de nuevo.

```
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ cd ../..
turtlebot@rosindigo:~/catkin_ws$ catkin_make
turtlebot@rosindigo:~/catkin_ws$ source devel/setup.bash
```

2.4. Ejecución del Publisher y Subscritor desarrollados

Para probar el software de los nodos que se ha desarrollado, es necesario lanzar en primer lugar al Master o Maestro de la red de nodos y el servidor de parámetros y posteriormente lanzar a los nodos. Para luego lanzar posteriormente a los nodos. Para lanzar al maestro, es

necesario utilizar la función *roscore*. Con esta función se lanza al maestro, al servidor de parámetros y un nodo denominado */rosout* que es la salida por consola del maestro.

Nota: recordad que hay que abrir varias ventanas en la consola o varias consolas para ejecutar de manera adecuada todas las instrucciones.

```
turtlebot@rosindigo:~/catkin_ws$ roscore
```

```
roscore http://localhost:11311/135x33
turtlebot@rosindigo:~/catkin_ws$ roscore
... logging to /home/turtlebot/.ros/log/a0a91d72-6847-11ed-b613-08002702c1df/roslaunch-rosindigo-17982.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://localhost:49934/
ros_comm version 1.11.10

SUMMARY
=====
PARAMETERS
* /rostdistro: indigo
* /rosversion: 1.11.10

NODES
auto-starting new master
process[master]: started with pid [17994]
ROS_MASTER_URI=http://localhost:11311/

setting /run_id to a0a91d72-6847-11ed-b613-08002702c1df
process[rosout-1]: started with pid [18007]
started core service [/rosout]
```

Resultado 6: roscore

La correcta ejecución de la instrucción devolverá al finalizar las palabras clave “*started core service [/rosout]*”. Una vez lanzado el maestro, se deben abrir dos nuevas terminales para lanzar al nodo emisor y al nodo suscriptor. Para lanzar a estos nodos es necesario utilizar la función *roslaunch [package name] [node_name]*, donde *package_name* corresponde al nombre del paquete y *node_name* al nombre del nodo. De modo que la llamada al emisor y al receptor quedarían del siguiente modo:

```
turtlebot@rosindigo:~/catkin_ws$ roslaunch paq_inicial emisor.py
```

```
turtlebot@rosindigo:~/catkin_ws$ roslaunch paq_inicial emisor.py
[INFO] [WallTime: 1668889383.647193] Hola 1
[INFO] [WallTime: 1668889384.649432] Hola 2
[INFO] [WallTime: 1668889385.648652] Hola 3
[INFO] [WallTime: 1668889386.649166] Hola 4
[INFO] [WallTime: 1668889387.648989] Hola 5
[INFO] [WallTime: 1668889388.647418] Hola 6
[INFO] [WallTime: 1668889389.651854] Hola 7
[INFO] [WallTime: 1668889390.655369] Hola 8
[INFO] [WallTime: 1668889391.654357] Hola 9
[INFO] [WallTime: 1668889392.648828] Hola 10
[INFO] [WallTime: 1668889393.647760] Hola 11
[INFO] [WallTime: 1668889394.662600] Hola 12
[INFO] [WallTime: 1668889395.649577] Hola 13
```

Resultado 7: roslaunch paq_inicial emisor.py

Nota: recordar que cada vez que se inicia una terminal o ventana de terminal es necesario incluir la ruta de ejecución del workspace. La instrucción es la siguiente:

```
turtlebot@rosindigo:~/catkin_ws$ source devel/setup.bash
```

Si no se realiza este paso se encuentra la terminal puede arrojar el siguiente error:

```
turtlebot@rosindigo:~/catkin_ws 66x18
turtlebot@rosindigo:~/catkin_ws$ rosrn paq_inicial subscriber.py
[rospack] Error: package 'paq_inicial' not found
turtlebot@rosindigo:~/catkin_ws$
```

```
turtlebot@rosindigo:~/catkin_ws$ rosrn paq_inicial subscriber.py
```

```
turtlebot@rosindigo:~/catkin_ws$ rosrn paq_inicial subscriber.py
[INFO] [WallTime: 1668889474.662025] /subscriber_18105_1668889474
008Yo escucho: Hola 92
[INFO] [WallTime: 1668889475.664683] /subscriber_18105_1668889474
008Yo escucho: Hola 93
[INFO] [WallTime: 1668889476.666653] /subscriber_18105_1668889474
008Yo escucho: Hola 94
[INFO] [WallTime: 1668889477.671848] /subscriber_18105_1668889474
008Yo escucho: Hola 95
[INFO] [WallTime: 1668889478.675138] /subscriber_18105_1668889474
008Yo escucho: Hola 96
[INFO] [WallTime: 1668889479.675492] /subscriber_18105_1668889474
008Yo escucho: Hola 97
[INFO] [WallTime: 1668889480.677126] /subscriber_18105_1668889474
008Yo escucho: Hola 98
[INFO] [WallTime: 1668889481.680419] /subscriber_18105_1668889474
008Yo escucho: Hola 99
[INFO] [WallTime: 1668889482.671581] /subscriber_18105_1668889474
008Yo escucho: Hola 100
```

Resultado 8: rosrn paq_inicial subscriber.py

Además de mostrar la información por consola que se muestra las figuras anteriores, los nodos pueden ser consultados de otra forma. La forma de consultarlo es mediante la instrucción *rostopic*. Mediante *rostopic list* se muestran todos los nodos activos actualmente en el sistema ROS y mediante *rostopic info /nombre_nodo* se puede obtener información sobre un nodo concreto. A continuación, se muestra un ejemplo para el nodo emisor y el nodo subscriber.

```
turtlebot@rosindigo:~/catkin_ws$ rostopic list
```

```
turtlebot@rosindigo:~/catkin_ws 66x18
turtlebot@rosindigo:~/catkin_ws$ rostopic list
/emisor_18048_1668889383484
/rosout
/subscriber_18105_1668889474008
turtlebot@rosindigo:~/catkin_ws$
```

Resultado 9: consulta de nodos activos

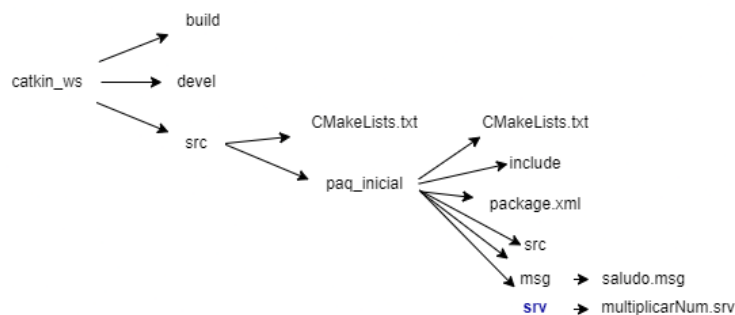
En la respuesta de la consulta se observan 3 nodos. Por un lado, la salida por consola del maestro llamado *"/rosout"*. Por otro lado, los nodos emisor y receptor con sus números para evitar la duplicidad.

3. Creación de un servicio, un servidor y un cliente

3.1. Creación de un servicio

El paquete `paq_inicial`, ya este paquete tiene un mensaje, así como un nodo emisor y un subscriptor. En este punto se procederá crear un servicio donde puedan existir un nodo cliente y un nodo servidor.

Para poder lograr esto, primero se necesita crear un directorio llamado `srv` donde se almacenarán todos los servicios que podrá utilizar el paquete creado. En esta carpeta se creará un fichero que contendrá los campos del servicio, el fichero se nombrará como `multiplicarNum.srv`. En la siguiente figura se puede observar la distribución de los directorios y donde se ubica el servicio `multiplicarNum.srv`.



El contenido del fichero `multiplicarNum.srv` debe tener como valores de entrada *int64 num1* y *int64 num2*, y como valor de salida tendrá *int64 multi* como se observa más abajo. El objetivo de este servicio es multiplicar dos números y obtener su resultado. Para crearlo es necesario realizar los siguientes pasos:

```
turtlebot@rosindigo:~/catkin_ws$ roscd paq_inicial
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ mkdir srv
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ vi srv/multiplicarNum.srv
```

```
turtlebot@rosindigo: ~/catkin_ws/src/paq_inicial 67x25
int64 num1
int64 num2
---
int64 multi
~
```

Resultado 10: información a incluir en el fichero `multiplicarNum.srv`

Una vez finalizado la creación del servicio para el correcto funcionamiento es necesario declarar en el fichero `CMakeLists.txt` ubicado en el directorio `catwin_ws/src/paq_inicial` las dependencias de los servicios utilizados mediante `add_service_files`. Se debe modificar el fichero `CMakeLists.txt` para que quede del siguiente modo:

```
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ gedit CMakeLists.txt &
```

Para modificar el fichero se recomienda buscar la frase clave `"add_service_"` y no borrar el contenido de lo comentado, sino añadir la configuración del recuadro debajo del comentario.

```
add_service_files(
  FILES
  multiplicarNum.srv
)
```

El resultado debería ser el siguiente:

```
continua el fichero ...
## Generate services in the 'srv' folder
# add_service_files(
#   FILES
#   Service1.srv
#   Service2.srv
# )
add_service_files(
  FILES
  multiplicarNum.srv
)
... continua el fichero
```

Así mismo, también se debe modificar las siguientes líneas del fichero, buscar “find_package” del siguiente modo:

```
find_package(catkin REQUIRED COMPONENTS
  rospy
  std_msgs
  message_generation #Nuevo a incluir
)
```

Por último, también hay que incluir lo siguiente en el fichero, buscar “generate_messages”:

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

El resultado debería ser el siguiente:

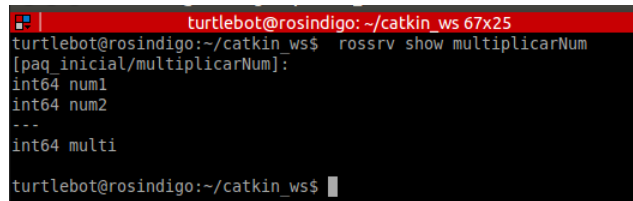
```
continua el fichero ...
## Generate added messages and services with any dependencies listed
here
# generate_messages(
#   DEPENDENCIES
#   std_msgs
# )
generate_messages(
  DEPENDENCIES
  std_msgs
)
... continua el fichero
```

Una vez llevada a cabo la modificación, es necesario ir a la carpeta raíz del espacio de trabajo *catkin_ws* y ejecutar la instrucción de compilación del entorno de trabajo de nuevo.

```
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ cd ../..  
turtlebot@rosindigo:~/catkin_ws$ catkin_make  
turtlebot@rosindigo:~/catkin_ws$ source devel/setup.bash
```

Para comprobar si todo se ha creado de manera correcta, es necesario hacer uso de la instrucción *rossrv*.

```
turtlebot@rosindigo:~/catkin_ws$ rossrv show multiplicarNum
```



```
turtlebot@rosindigo:~/catkin_ws 67x25  
turtlebot@rosindigo:~/catkin_ws$ rossrv show multiplicarNum  
[paq_inicial/multiplicarNum]:  
int64 num1  
int64 num2  
---  
int64 multi  
turtlebot@rosindigo:~/catkin_ws$
```

Resultado 11: resultado de ejecutar rossrv show multiplicarNum

Como se puede observar en el anterior resultado, primero muestra *[paq_inicial/multiplicarNum]*, lo cual corresponde al paquete y al nombre del servicio. Luego aparece *int64 num*, lo cual representa al contenido del servicio.

3.2. Creación de un servidor y un cliente

Una vez finalizada la creación del servicio, se procede a crear el software para que un nodo pueda actuar como servidor y otro como cliente. El nodo cliente podrá lanzar peticiones puntuales al nodo servidor que recibirá el mensaje lo procesará y devolverá una respuesta al cliente.

De nuevo se hará uso del directorio *scripts* para alojar el código relacionado con el lenguaje Python. El servidor debe ser alojado en *scripts/servidor.py*. El código a utilizar para crear el servidor es el siguiente:

```
turtlebot@rosindigo:~/catkin_ws$ roscd paq_inicial
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ gedit scripts/servidor.py &
```

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

from __future__ import print_function

from paq_inicial.srv import *
import rospy

def handle_multiplicarNum(req):
    print("Devolver [%s * %s = %s]"%(req.num1, req.num2, (req.num1 * req.num2)))
    return multiplicarNumResponse(req.num1 * req.num2)

def multiplicarNum_server():
    rospy.init_node('multiplicarNum_server')
    serv = rospy.Service('multiplicarNum', multiplicarNum, handle_multiplicarNum)
    print("Preparado para multiplicar:")
    rospy.spin()

if __name__ == "__main__":
    multiplicarNum_server()
```

Nota: num1, num2 (entradas) y multi (salida) son las variables previamente definidas en el servicio.

El código es muy similar a los anteriores, la única diferencia es la llamada al servicio que se basa en `rospy.Service` y tiene como parámetros el nombre del servicio, el tipo de servicio y la función de callback que es llamada cuando se recibe una petición al servicio.

```
serv = rospy.Service('multiplicarNum', multiplicarNum, handle_multiplicarNum)
```

El cliente al igual que el servidor debe ser alojado en *scripts*, pero modificando su nombre a *cliente.py*. El código a utilizar para crear el servidor es el siguiente:

```
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ gedit scripts/cliente.py &
```

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

from __future__ import print_function

import sys
import rospy
from paq_inicial.srv import *

def multiplicarNum_client(x, y):
    rospy.wait_for_service('multiplicarNum')
    try:
        multiplicarnum= rospy.ServiceProxy('multiplicarNum', multiplicarNum)
        resp1 = multiplicarnum(x, y)
        return resp1.multi
    except rospy.ServiceException as e:
        print("Servicio multiplicacion fallo: %s"%e)

def usage():
    return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print(usage())
        sys.exit(1)
    print("Peticion %s*s"% (x, y))
    print("%s * %s = %s"%(x, y, multiplicarNum_client(x, y)))
```

El código es muy similar a los anteriores, lo más destacable es que el cliente espera hasta que se encuentre disponible el servicio *multiplicarNum*, una vez el servicio está disponible, se realiza la petición al servidor y se espera su respuesta.

```
rospy.wait_for_service('multiplicarNum')
try:
    multiplicarnum= rospy.ServiceProxy('multiplicarNum', multiplicarNum)
    resp1 = multiplicarnum(x, y)
    return resp1.multi
except rospy.ServiceException as e:
    print("Servicio multiplicacion fallo: %s"%e)
```

Para finalizar el desarrollo de un servicio completo, es necesario dar permisos de ejecución a los ficheros de Python creados para que actúen como servidor y como cliente. Esto se realiza mediante la instrucción *chmod +x ruta/fichero*.

```
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ chmod +x scripts/servidor.py
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ chmod +x scripts/cliente.py
```

Para finalizar es necesario declarar en el fichero *CMakeLists.txt* ubicado en el directorio *catwin_ws/src/paq_inicial* la ejecución de los ficheros de Python. Se debe modificar el fichero *CMakeLists.txt* para que quede del siguiente modo:

```
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ gedit CMakeLists.txt &
```

```
install(PROGRAMS
  scripts/servidor.py
  scripts/cliente.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

El resultado debería ser el siguiente:

```
continua el fichero...
#install(PROGRAMS
#  scripts/my_python_script
#  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
# )
install(PROGRAMS
  scripts/servidor.py
  scripts/cliente.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

Una vez llevada a cabo la modificación, es necesario ir a la carpeta raíz del espacio de trabajo *catkin_ws* y ejecutar la instrucción de compilación del entorno de trabajo de nuevo.

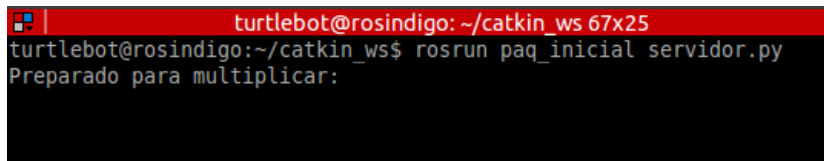
```
turtlebot@rosindigo:~/catkin_ws/src/paq_inicial$ cd ../..
turtlebot@rosindigo:~/catkin_ws$ catkin_make
turtlebot@rosindigo:~/catkin_ws$ source devel/setup.bash
```


3.3. Ejecución del servidor y del cliente desarrollados

Para probar el software de los nodos que se ha desarrollado, es necesario lanzar en primer lugar al Master como se hizo anteriormente. Para luego lanzar posteriormente a los nodos. Para lanzar al maestro, es necesario utilizar la función *roscore*.

Para lanzar a estos nodos es necesario utilizar la función *roslaunch* *[package name]* *[node_name]*, donde *package_name* corresponde al nombre del paquete y *node_name* al nombre del nodo. De modo que la llamada al servidor y al cliente quedarían del siguiente modo:

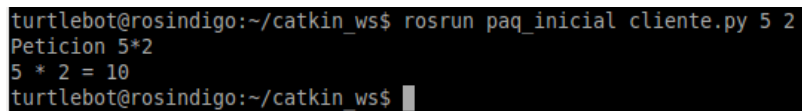
```
turtlebot@rosindigo:~/catkin_ws$ roslaunch paq_inicial servidor.py
```



```
turtlebot@rosindigo:~/catkin_ws$ roslaunch paq_inicial servidor.py
Preparado para multiplicar:
```

Resultado 12: lanzar nodo servidor

```
turtlebot@rosindigo:~/catkin_ws$ roslaunch paq_inicial cliente.py 5 2
```



```
turtlebot@rosindigo:~/catkin_ws$ roslaunch paq_inicial cliente.py 5 2
Petición 5*2
5 * 2 = 10
turtlebot@rosindigo:~/catkin_ws$
```

Resultado 13: lanzar nodo cliente