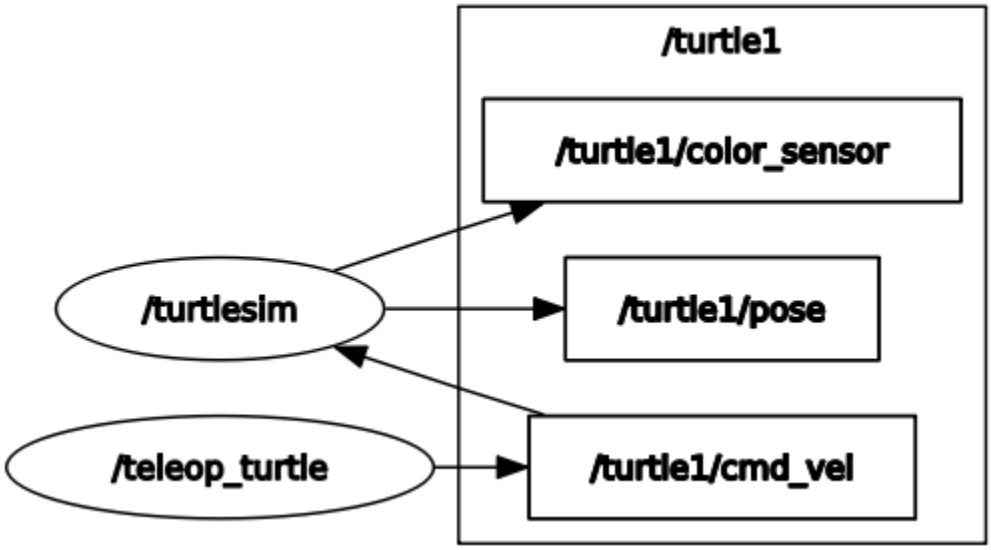
ROS Tutorial (ROS1) - ROS Noetic 2H30 [Crash Course]
https://www.youtube.com/watch?v=wfDJAYTMTdk

(Para la instalación, seguir este vídeo paso a paso: https://youtu.be/ZEfh7NxLMxA?feature=shared)

1. Start Your First Node:

Necesario inicializar el Ros Master antes de poder crear nodo: *roscore*
rosrun <package-name> <node-name> : lanza un nodo ROS. (en este caso rosrun rospy_tutorials talker y después listener)
rqt_graph para ver gráficamente los nodos, topics…
Para mover el turltesim: *rosrun turtlesim turtlesim_node* y en otra terminal *rosrun turtlesim turtle_teleop_key*:



Aquí podemos observar nodos publishers: /teleop_turtle, nodos subscribers: /turle1/pose y nodos que son ambas cosas: /turtlesim

2. Create and Setup a Catkin Workspace:

Es el espacio de trabajo de ROS para que se pueda trabajar de manera organizada con los paquetes, ya que con los paquetes es como se organiza ROS.

- Crear carpeta catkin: *mkdir catkin_ws*
- Una vez dentro, crear la carpeta src, pero no meterse en esta, quedarse en el catkin_ws y ejecutar catkin_make
- *source ~/catkin_ws/devel/setup.bash*, ejecutarlo, ejecutar *gedit~/.bashrc* y poner al final del archivo lo de antes

3. Create a ROS Package:

Dentro de src añadiremos paquetes, y dentro de ellos escribiremos nuestros nodos, de tal manera que cada paquete va a ser una subparte de la aplicación destinada.
Dentro de src poner *catkin_create_pkg <nombre paquete> <dependencias>*:
catkin_create_pkg my_robot_controller rospy (librería python que permite dentro de código python tener acceso a funcionalidades de ROS) (se podría poner también roscpp para código con C++) *turtlesim* (pq también voy a poner esta dependencia).
Dentro de src hacer *code .* (ejecuta VS en esa carpeta) y se modifica el package.xml, después volvemos a hacer catkin_make, ya no se volverá a compilar si se usa python, ya que es interpretado, así que podremos iniciar el nodo de python sin volver a ejecutar, a diferencia de C++.

4. Create your first ROS Node with Python:

Como en los apuntes de PDR, creamos una carpeta scripts dentro de mi paquete (my_robot_controller).
Creamos archivo: *touch my_first_node.py* y después *chmod +x my_first_node.py* para que sea un archivo ejecutable
Desde src abrimos VS con code . e instalamos las extensiones de Python, CMake y ROS
Una vez listo el programa, podemos ver su ejecución con *python3 my_first_node.py* y después *rosrun my_robot_controller (nombre paquete) my_first_node.py (nombre ejecutable)*
Con *rosnode list* y el nodo creado en marcha, se ve una lista de los nodos actuales, entre los que aparece rosout, un nodo que se crea desde el ros master y que va a manejar funcionalidades login.
Solo se puede ejecutar a la vez un nodo con el mismo nombre.

5. Understand what is a ROS Topic:

Parecido a antes, podemos usar rostopic list, para saber más de cada uno podemos hacer *rostopic info /<nombre topic>*. Para el caso de chatter, vemos:

Type: std_msgs/String

Publishers:
 * /talker_2242_1723909205520 (http://victor-virtual-machine:41397/)

Subscribers:
 * /listener_2283_1723909216465 (http://victor-virtual-machine:36393/)

Si ponemos *rosmsg std_msgs/String* y devuelve string data para ver los tipos que hay en ese mensaje.También está *rostopic echo*.

Topic es la manera de comunicar entre nodos
También hay *rosmsg show*
Los nodos solo saben de qué topic reciben, no de qué nodo proviene la info
Resumen pasos:

1. *rostopic list* y te salen todos los que hay. Pillas uno, por ej: /turtle1/cmd_vel
2. *rostopic info /turtle1/cmd_vel* y te sale el tipo de mensaje que envían (lo necesario para los publishers y subscribers). En este caso, geometry_msgs/Twist
3. Si quieres más información sobre el mensaje, para saber de qué está compuesto el Twist: *rosmsg info geometry_msgs/Twist*. *info* también puede ser cambiado por *show*.

6. Write a publisher node with python:

Para pub = rospy.Publisher() le tengo que dar un nombre, que lo veo con *rostopic list*, una vez he hecho *roscore* y *rosrun turtlesim turtlesim_node*. Hago *rostopic list* y encuentro: /turtle1/cmd_vel entre varios, que es el nombre que debo poner porque es lo que queremos controlar. También ponemos data class, que se ve con *rostopic info* y lo encontrado antes, consiguiendo geometry_msgs/Twist.

Penúltimo: Understand what is a ROS Service:

Ejecutar rosrun *rospy_tutorials add_two_ints_server* y después *rosservice list*.
Para llamar a un servicio desde la terminal: *rosservice call /add_two_ints* y pulsar tab dos veces para autocompletar. Así conseguimos: rosservice call /add_two_ints "a: 0
b: 0"
, que se pueden cambiar los valores predeterminados a lo que quieras.
Uso de *rossrv show <Type>*, siendo el type rospy_tutorials/AddTwoInts, que se ve con el comando info de antes. Al final, es lo mismo que hacemos con los mensajes y ves:
int64 a
int64 b
---
int64 sum
, que lo primero es la request y lo segundo la response.

Último: Escribir un Servicio Cliente ROS con Python:

Queremos que cuando esté a la izquierda, la línea que deje sea verde y a la derecha rojo.

Código comentado paso a paso:

```
#!/usr/bin/env python3
# Esta línea indica que el script debe ejecutarse con el intérprete de Python 3.
# El uso de `env` permite que el sistema localice automáticamente la ubicación del intérprete de Python 3.
```

```python
import rospy
# Importa la biblioteca `rospy`, que se utiliza para escribir nodos de ROS en Python.

from turtlesim.msg import Pose
# Importa el mensaje `Pose` del paquete `turtlesim`, que se utiliza para recibir la posición y orientación
# de la tortuga en el simulador `turtlesim`.

from geometry_msgs.msg import Twist
# Importa el mensaje `Twist` del paquete `geometry_msgs`, que se utiliza para enviar comandos de
# velocidad lineal y angular a la tortuga.

from turtlesim.srv import SetPen
# Importa el servicio `SetPen` del paquete `turtlesim`, que permite cambiar el color y el grosor de la línea
# que dibuja la tortuga en `turtlesim`.

previous_x = 0
# Se define una variable global `previous_x` que se utilizará para almacenar la posición `x` anterior de la
# tortuga. Esto es necesario para evitar llamadas repetidas al servicio `SetPen` cuando la tortuga se mueve
# entre diferentes áreas.

def call_set_pen_service(r, g, b, width, off):
    # Esta función es un wrapper para llamar al servicio `SetPen`, que permite cambiar el color del trazo
    # de la tortuga. Recibe los parámetros `r`, `g`, `b` para los valores de color (rojo, verde, azul),
    # `width` para el grosor del trazo, y `off` para activar o desactivar el trazo.

    try:
        set_pen = rospy.ServiceProxy("/turtle1/set_pen", SetPen)
        # Crea un objeto `ServiceProxy` que permite llamar al servicio `SetPen` del nodo `turtle1`.
        # `ServiceProxy` actúa como un cliente que se comunica con el servidor del servicio.

        response = set_pen(r, g, b, width, off)
        # Llama al servicio con los parámetros proporcionados y guarda la respuesta (si es que hay alguna)
        # en la variable `response`. Este servicio establece los parámetros de color y grosor del trazo de la
        # tortuga.

        # rospy.loginfo(response)
        # Esta línea está comentada, pero si se descomenta, imprimiría la respuesta del servicio
        # en la terminal, lo que es útil para depuración.
    except rospy.ServiceException as e:
        rospy.logwarn(e)
        # Si hay un problema al llamar al servicio, se captura la excepción `ServiceException`
        # y se emite una advertencia con el mensaje de error en la terminal.

def pose_callback(msg: Pose):
    # Esta función es el callback que se ejecuta cada vez que se recibe un mensaje de posición (`Pose`)
    # en el tópico suscrito. El argumento `msg` es el mensaje de tipo `Pose` que contiene la posición
    # y orientación actuales de la tortuga.

    cmd = Twist()
    # Se crea un objeto `Twist` que se utilizará para definir la velocidad lineal y angular que se enviará
    # a la tortuga.

    if msg.x > 9.0 or msg.x < 2.0 or msg.y > 9.0 or msg.y < 2.0:
        # Si la tortuga se encuentra cerca de los bordes del espacio de simulación (dentro de un margen de
        # 2 unidades del borde), entonces...

        cmd.linear.x = 1.0
        cmd.angular.z = 1.4
        # ...se establece una velocidad lineal baja (`1.0` en `x`) y una velocidad angular positiva (`1.4`
        # en `z`), lo que hará que la tortuga gire para evitar chocar con el borde.
```

```
    else:
        cmd.linear.x = 5.0
        cmd.angular.z = 0.0
        # Si la tortuga está lejos de los bordes, se establece una velocidad lineal alta (`5.0` en `x`)
        # y sin giro (`0.0` en `z`), lo que hará que la tortuga avance en línea recta a mayor velocidad.

    global previous_x
    # Se utiliza `global previous_x` para poder modificar la variable `previous_x` que se definió fuera de
    # la función. Esto es necesario porque `previous_x` se utiliza para rastrear la posición `x` anterior
    # de la tortuga.

    if msg.x < 5.5 and previous_x > 5.5:
        # Si la posición actual de la tortuga en `x` es menor que `5.5` y en la iteración anterior
        # estaba a la derecha de `5.5` (es decir, `previous_x` > `5.5`), entonces...

        call_set_pen_service(0, 255, 0, 3, 0)
        # ...se llama al servicio `SetPen` para cambiar el color del trazo a verde (`0` rojo, `255` verde,
        # `0` azul) y se establece el grosor del trazo en `3`.

        rospy.loginfo("Set color to green")
        # Se imprime un mensaje informativo en la terminal indicando que el color del trazo se ha cambiado a
verde.

    elif msg.x > 5.5 and previous_x < 5.5:
        # Si la posición actual de la tortuga en `x` es mayor que `5.5` y en la iteración anterior estaba
        # a la izquierda de `5.5` (es decir, `previous_x` < `5.5`), entonces...

        call_set_pen_service(255, 0, 0, 3, 0)
        # ...se llama al servicio `SetPen` para cambiar el color del trazo a rojo (`255` rojo, `0` verde,
        # `0` azul) y se establece el grosor del trazo en `3`.

        rospy.loginfo("Set color to red")
        # Se imprime un mensaje informativo en la terminal indicando que el color del trazo se ha cambiado a
rojo.

    previous_x = msg.x
    # Finalmente, se actualiza `previous_x` con la posición actual en `x` de la tortuga para su uso
    # en la siguiente iteración.

    pub.publish(cmd)
    # Publica el comando de velocidad (`cmd`) en el tópico `/turtle1/cmd_vel` para controlar el movimiento
    # de la tortuga en la simulación.

if __name__ == '__main__':
    rospy.init_node("turtle_controller")
    # Inicializa el nodo de ROS con el nombre `"turtle_controller"`. Este nodo será responsable de controlar
    # el movimiento de la tortuga y cambiar el color de su trazo según su posición.

    rospy.wait_for_service("/turtle1/set_pen")
    # Esta línea bloquea la ejecución del programa hasta que el servicio `/turtle1/set_pen` esté disponible.
    # Esto es importante porque no se puede llamar al servicio hasta que el servidor del servicio esté listo.

    # Creamos el publisher antes que el subscriber (tiene sentido)
    pub = rospy.Publisher("/turtle1/cmd_vel", Twist, queue_size=10)
    # Se crea un publisher que publicará mensajes de tipo `Twist` en el tópico `/turtle1/cmd_vel`.
    # Este tópico es utilizado para controlar la velocidad de la tortuga en `turtlesim`. `queue_size=10`
    # define el tamaño de la cola de mensajes, es decir, cuántos mensajes se pueden almacenar antes de que
    # se descarten si no se han procesado.

    sub = rospy.Subscriber("/turtle1/pose", Pose, callback=pose_callback)
    # Se crea un suscriptor que escucha mensajes de tipo `Pose` en el tópico `/turtle1/pose`.
    # Cada vez que se recibe un mensaje, se ejecuta la función `pose_callback`, que maneja la lógica de
```

```
    # movimiento de la tortuga y cambia el color del trazo según la posición.
    #Al hacer sub = rospy.Subscriber("/turtle1/pose", Pose, callback=pose_callback), estás diciendo
    # "llama a pose_callback cada vez que recibas un nuevo mensaje en /turtle1/pose", pero pose_callback no se ejecuta inmediatamente.
    # Si hubieras hecho sub = rospy.Subscriber("/turtle1/pose", Pose, callback=pose_callback()), pose_callback se ejecutaría
    # inmediatamente cuando se crea el suscriptor (lo cual no tiene sentido en este contexto)
    # y el suscriptor recibiría el valor de retorno de esa función (que en este caso, sería None si pose_callback no devuelve nada).

    rospy.loginfo("Node has started")
    # Se imprime un mensaje informativo en la terminal indicando que el nodo ha iniciado correctamente.

    rospy.spin()
    # Mantiene el nodo activo y escuchando eventos hasta que se detenga (por ejemplo, con Ctrl+C).
    # Esta función es esencial en nodos que suscriben tópicos o están esperando servicios, ya que sin ella,
    # el script terminaría inmediatamente.
```