Cada vez que **modifiquemos** un programa → hacer colcon build --packages-select paquete
orden antes del colcon **si va lento** → $ sudo /etc/init.d/clamav-daemon stop

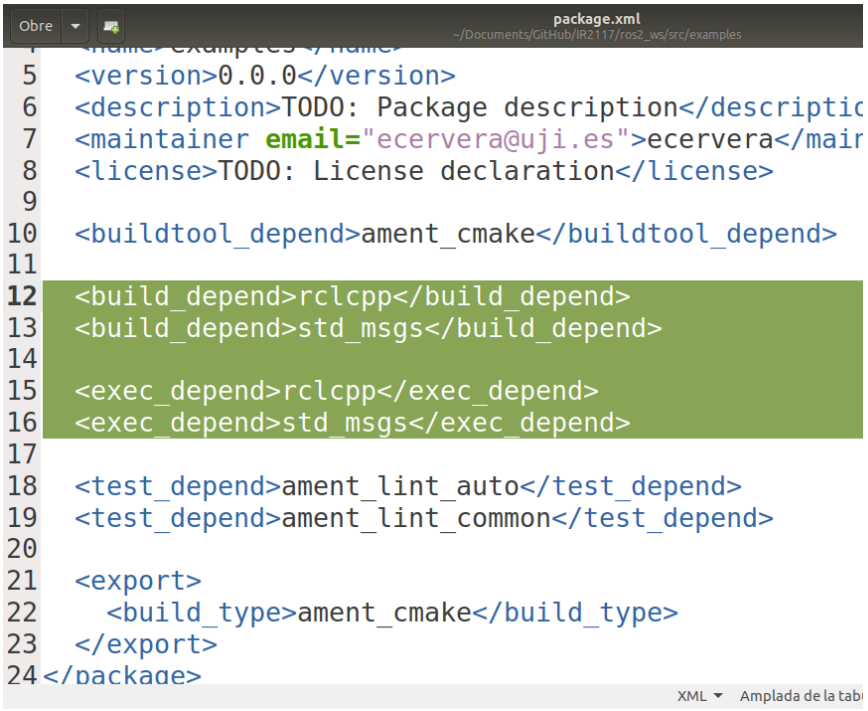
Para usar ros en una carpeta:
TERMINAL 1 - SUBSCRIBER
$ source /opt/ros/foxy/setup.bash → hace **ajustes necesarios**
$ colcon build → builder/compilador de ROS, si queremos **compilar el directorio entero**
$ colcon build --packages-select paquete → si queremos **compilar un paquete** en específico dentro del directorio
$ source install/setup.bash → para **utilizar programa**
$ export ROS_DOMAIN_ID=4 → para **no interferir** con los compañeros
$ export ROS_LOCALHOST=1 → **indicar IP** para **comunicación** entre nodos y host
$ros2 run examples_topics subscriber → **ejecutar el programa**
pckg programa

TERMINAL 2 - PUBLISHER
$ source /opt/ros/foxy/setup.bash
$ source install/setup.bash
$ export ROS_DOMAIN_ID=4
$ export ROS_LOCALHOST=1
$ ros2 run examples_topics publisher
pkg programa
—----------
pongo la orden para mostrar los mensajes x si acaso:

- ros2 topic pub /number std_msgs/Int32 “{data : ’123’}”
- ros2 topic echo /<nombre_topic>

TERMINAL 3- VER NODOS EN GRAFO
$ source /opt/ros/foxy/setup.bash
$ colcon build
$ source install/setup.bash
$ export ROS_DOMAIN_ID=4
$ export ROS_LOCALHOST=1
$ rqt_graph → **ver nodos**

Package.xml → fichero con **info sobre pckg**: autor, licencia… Sirve para conocer al desarrollador por si se publica. También tiene **dependencias que nuestro programa necesita de otros módulos**.
habrá que cambiar lo subrayado q básicamente es especificar q paquetes hacen falta para construir el nuestro (build_depend), y cuales hacen falta para ejecutar el código de nuestro pack (exec_depend)



lo pongo pa copiar y pegar:
<build_depend>rclcpp</build_depend>
<build_depend>std_msgs</build_depend>

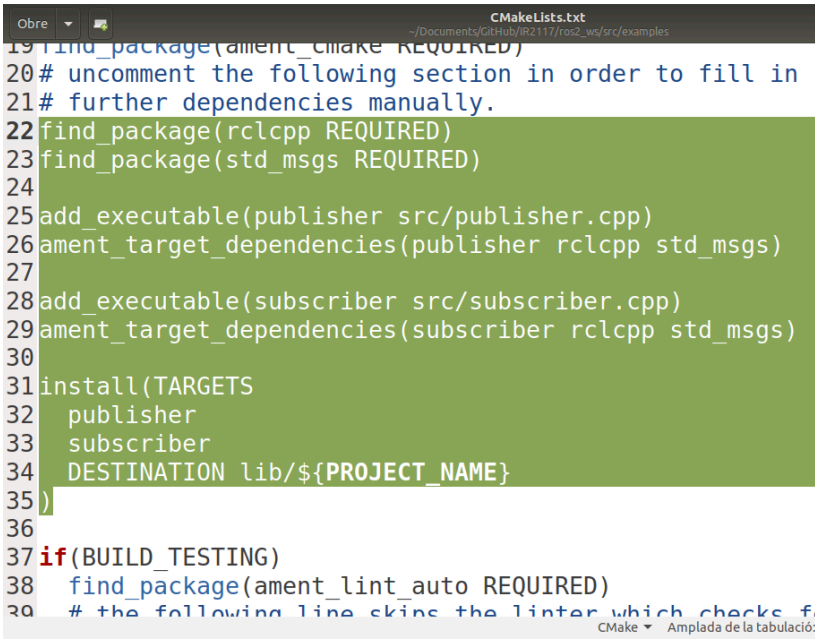
<exec_depend>rclcpp</exec_depend>
<exec_depend>std_msgs</exec_depend>

CMakeLists.txt → fichero de **instrucciones para compilar e instalar nuestro paquete**

- Find_package → **busca** las **dependencias**
- **Genera** un **ejecutable a partir** de los **nodos** (publisher.cpp y subscriber.cpp)
- Instala cosas de ros 2

Normalmente se generan de manera automática y si procede hay que cambiar alguna dependencia, ejecutable…
Hay que cambiar lo verde:

(acordarse de cambiar el topic (publisher, subscriber) en todas las líneas)



COMANDOS ROS2

- ros2 node list → te dice **nodos activos**
- ros2 node info <nodo_name> → **info** más **concreta** sobre un **nodo** → topic
- ros2 topic list → **topics** del **sistema,** unos de **uso interno y** otros **creados por nosotros**
- ros2 topic echo <topic_name> → **muestra msgs**, como ejecutar el subscriber
- ros2 topic info <topic_name> → **info** del **topic**
- ros2 interface show <msg type> → msg type lo sacamos del info esto devuelve int, string…??
- ros2 topic pub <topic_name> <msg_type> '<args>' → **publica el mensaje**, hacerlo en la terminal de publisher
- ros2 topic pub /topic std_msgs/String "data: Hello ROS Developers"
- ros2 topic pub /number std_msgs/Int32 "{data : '123'}"
- ros2 topic hz <topic_name> → **frecuencia con la que publica**
- ros2 service call <service_name> <service_type> <arguments> → arguments es opcional, sirve para saber como encontrar el tipo de servicio, y como encontrar la estructura del tipo de argumentos

PARÁMETROS

- ros2 param list → **parámetros de cada nodo**
- ros2 param get <node_name> <parameter_name> → para saber el **valor de un parámetro**, hay que indicar tanto el nodo como el parámetro
- ros2 param set <node_name> <parameter_name> <value>→ para **cambiar el parámetro**, también hay que indicar el nodo, parámetro e indicar el valor que quieres cambiar
- ros2 param dump <node_name> → te **guarda** todos los **parámetros del nodo en** un **fichero**
- ros2 param load <node_name> <parameter_file> → te **carga** los **valores** del **fichero en** el **nodo** indicado
- ros2 run <package_name> <executable_name> –param-file <file_name>

TOPICS

- Es un nombre con un tipo de mensaje asociado
- Publishers publican mensajes en el Topic y Subscribers (suscritos a ese topic y ese tipo de mensaje) lo reciben

ACCIONES

forma de comunicarse dos nodos de ros parecida al servicio, pero se pueden cancelar antes de que acaben, dan un feedback mientras que se están ejecutando, y al terminar de ejecutar te devuelve el resultado. La acción son dos servicios y un topic. Forma de comunicar de los nodos.

Tenemos dos nodos:

- Uno es la acción del cliente
- Otro es la acción del servicio

Comandos de las acciones:

- ros2 node info /turtlesim → **muestra** todos los **topics, servicios y acciones del nodo**
- ros2 action list → **muestra** las **acción** que se está **ejecutando**
- ros2 action list -t → lo mismo de antes para que te **muestre** el **tipo de acción** que se está **ejecutando**
- ros2 action info /turtle1/…. → las **acciones** de un **nodo**
- ros2 interface show /turtlesim/action/RotateAbsolute → te **devuelve** el **tipo de la entrada, del feedback y del resultado**
- ros2 action send_goal <action_name> <action_type> <value> → para **enviar** una **acción a** un **nodo**, crea un nodo que se lo envía al terminal.

APUNTES ROS2, C++, BASH

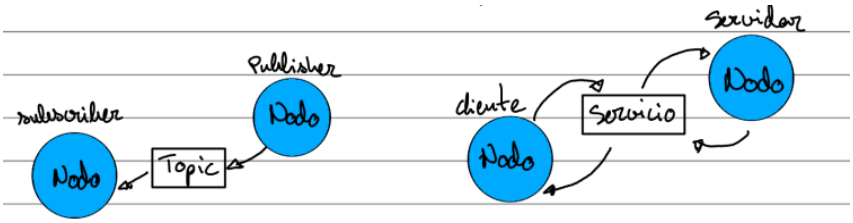
- workspace → es el directorio superior, con paquetes (subdirectorios) con diferentes ficheros. En nuestro caso es ros2_ws/src/
- int argc → nº de parámetros
- char** argv/ char* argv[ ] → doble puntero o array de punteros, se puede iterar
- std::cin.eof() → puede servir para indicar cuando queremos para de dar argumentos den terminar con ctrl+d
- ls > fichero.txt → la salida de ls te la escribe en un fichero (> redirección)
- ls | cat → la salida de ls te la redirecciona a la siguiente instrucción (cat) (| = tubería)

ros2 pkg create --build-type ament_cmake paquete –dependencies rclcpp → crea un nuevo paquete en ROS2 de nombre utilizando el sistema de construcción Ament con CMake. También añade las dependencias que se han indicado (rclcpp) a package.xml y CMakeLists.txt

PRÁCTICA 1

- Lo primero es crear una carpeta donde trabajaremos con el ros.
    + Dentro de esta carpeta debe de haber dos cosas:
        - Carpeta src → donde irán los paquetes de ros que creemos.

        - .gitignore (opcional) → escribimos los nombres de las carpetas que no queremos que se suban al repositorio
- Para poder utilizar comandos de ros deberemos configurar el entorno de ros (source /opt/ros/foxy/setup.bash).
- Al ejecutar “export ROS_LOCALHOST_ONLY=1” → indicamos que las conexiones de red en ROS deben limitarse solo al local_host, solo se aceptarán conexiones desde la misma máquina donde se ejecuta el ROS
- Para crear un paquete de ROS
    + Desde src: ros2 pkg create –build-type ament_cmake <nombre>
    + Para compilar un paquete y que podamos usar lo que hay dentro usaremos la orden: (desde la carpeta principal ros2_ws)
        - colcon build → esta compila todo
        - colcon build –packages-select <nombre> → para seleccionar lo que quieres compilar
- En la carpeta src que se crea cuando hacemos un paquete, se encuentra dentro del paquete, es donde escribiremos nuestros códigos en c++. En estos códigos, es donde describimos nuestro nodo. Las asociaciones de nodos pueden ser:



De momento solo hemos hecho publisher/subscriber

- Cosas que tiene ambas:
    + <include> “rclcpp/rclcpp.hpp”
    + <include> “std_msgs/msg/(tipo(int32,string…)).hpp”
    + int main(int argc, char*agv[])

PUBLISHER Y SUBSCRIBER

El publisher es un nodo en el cual necesitamos lo siguiente:

- Se define su comportamiento dentro de un archivo de c++, python,... Lo importante de elegir un lenguaje es que otro sitio se tiene que poner. Nosotros utilizamos c++.

Este archivo se debe localizar en la carpeta src (si hay otra dentro del paquete que hayamos creado). Para este, lo importante es:

- Hacer un include de las librerías necesarias.
    + De c++, como #include <vector>, <string>...
    + De ros
        - #include “rclcpp/rclcpp.hpp”
        - #include “std_msgs/msg/Tipo.hpp” : para poner que tipo de mensajes lanzaremos al topic
- Todos los nodos tienen definida la función **int main:**
    + argc es un entero que le indica el número de parámetros y argv es un array de punteros a caracteres.
    + Dentro de esta función se llama al constructor del rclcpp con los argumentos que le hemos pasado en el main. Par inicializar un elemento del tipo nodo:

        - Si lo hacemos dentro del main() → auto node = rclcpp::Node::make_shared("publisher")
        - Si lo hacemos fuera (solo para declararlo) hay que decir el tipo…
    + Crear un topic al que publicar:
        - auto publisher = node->create_publisher <tipo de mensaje> ("nombre del topic" (que concuerde con el parámetro de entrada: *node->create_subscription<std_msgs::msg::**Int32**>(**"number"**, 10, topic_callback)*, nº de elementos almacenables por defecto lo dejo 10, osea el máximo de mnsjs que pueden esperar en el buffer)
    + Para crear algo que podamos publicar en un topic → creamos el objeto → esto es lo que se manda (paquete) → std::msgs::msg::<Tipo> nombre_objeto (Ej: std::msgs::msg::String message) bon dia
        - La info que queramos transmitir debe ir en → message.data = "Lo que sea", [lo que sea], …
        - Para mandar la info al topic → publisher-> publish(message)
    + Cuando tenemos un bucle en el que queramos que un nodo haga algo de forma controlada → spin_some(nodo)
        - Si lo que queremos es que el bucle se repita hasta que termine su trabajo → spin(nodo)
    + Para finalizar → shutdown()

El subscriber funciona parecido:

- Tiene una función llamada **topic_callback**, la cual define el que hacer con la info que le llega a través del topic.
    + Importante el tipo de lo que le pasemos, debe ser el mismo que el que recibe, luego ya dentro lo podremos alterar.
- Para suscribirte a un topic en vez de node->create_publisher, es node->create_subscription, además en los argumentos que se le pasan añadimos la función topic_callback

Cosas a recordar:

- Tipos, ser coherentes con ellos a la hora de mandarlas, el topic no tiene tipo, lo que tiene que concordar son el tipo que se publica y el que recibe con el suscriptor.
- Tener en cuenta las operaciones entre tipos
    + Un int +, -, *, / para un float igual que int → hay que comprobarlo

RQT_CONSOLE → para mostrar mensajes en la terminal
ros2 run rqt_console rqt_console → para mostrar los mensajes de warning (como por ejemplo la hora)
fatal → errores que hacen que el sistema deje de funcionar
error → no tiene porque dañar el sistema
warn → algo que no ha ido bien
info → indica el evento o estado
debug → mensajes detallados de todo el paso a paso del proceso de la ejecución del sistema
El std::cout no se es un mensaje de este tipo

LAUNCHING NODES

otra alternativa cuando queremos lanzar varios nodos con un solo mensaje

ros2 launch turtlesim multisim.launch.py → hace que runee el siguiente launch (al hacer estar orden se abriran 2 nodos con una tortuga cada uno)

Para controlar (las tortugas empezaran a girar cada una en un sentido)

segundo terminal: ros2 topic pub /turtlesim1/turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"

tercer terminal: ros2 topic pub /turtlesim2/turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: -1.8}}"

RECORDING AND PLAYING BACK DATA

ros2 bag record topic → para grabar y este crea un fichero para grabarlo

ros2 bag play → para reproducir

ros2 bag record -o subset /turtle1/cmd_vel /turtle1/pose → con el -o podemos cambiar el nombre del bag file (subset en este caso). La orden sirve para grabar varios topics

ros2 bag info <file_name> → aparece información de la grabación como la duración

ros2 bag play subset → comienza a publicar las cosas que ha grabado