Teleoperación reactiva con bumper (Kobuki)

Teoría y herramientas para la práctica (ROS 2 + C++)

Francisco Martín Rico y Rodrigo Pérez Rodríguez

Qué hay que hacer en la práctica (y qué verás aquí)

- Objetivo: teleoperación **reactiva** del Kobuki usando el **bumper**.
- Implementa un nodo en C++ que:
  - se suscriba al topic del bumper y actualice un **estado** interno,
  - ejecute un **ciclo de control** periódico (p.ej., 10 Hz) con un timer,
  - publique comandos de velocidad en `/cmd_vel`.
- Valida el comportamiento con `ros2 topic echo`, logs y pruebas simples.
- Estas slides resumen **lo necesario** de ROS 2 en C++: callbacks, timers, publishers/subscribers, y cómo se ejecuta el nodo con `spin`.

Objetivo de esta mini-teoría

- Entender el **modelo de ejecución** de ROS 2 en C++: *eventos* y *callbacks*.
- Ver las piezas mínimas en `rclcpp`: nodos, timers, *publishers* y *subscribers*.
- Conectar el ejemplo `ch2_examples` con el patrón de la práctica: *bumper* $\rightarrow$ *estado* $\rightarrow$ *control* (10 Hz) $\rightarrow$ `/cmd_vel`.

Compilar y ejecutar los ejemplos (referencia rápida)

- Compila el paquete `ch2_examples`.
- Activa el overlay para que `ros2 run` encuentre binarios.
- Ejecuta el nodo `logger`.

```
colcon build --packages-select
    ch2_examples
source install/setup.bash
ros2 run ch2_examples logger
```

Introspección con `ros2cli` (validación rápida)

- `ros2cli` permite comprobar el sistema **sin tocar el código**.
- Para pub/sub, lo esencial es ver nodos, topics, tipos y datos.
- En la práctica, harás lo mismo con el topic del bumper y `/cmd_vel`.

```
ros2 node list
ros2 topic list
ros2 topic info /counter
ros2 topic echo /counter
```

Estructura del paquete ch2_examples

- `package.xml`: dependencias.
- `CMakeLists.txt`: cómo compila e instala.
- `include/`: cabeceras públicas (API).
- `src/ch2_examples/`: implementación de nodos.
- `src/*_main.cpp`: ejecutables que instancian nodos.

Estructura (resumen)

```
ch2_examples/
  package.xml
  CMakeLists.txt
  include/ch2_examples/*.hpp
  src/ch2_examples/*.cpp
  src/*_main.cpp
```

Patrón recomendado

Separa la clase del nodo del `main` para poder reutilizarla.

package.xml: dependencias principales

- ROS 2 declara dependencias en package.xml.
- Para estos ejemplos: rclcpp y std_msgs.
- En tu práctica, añadirás geometry_msgs y el mensaje del bumper.

Fragmento real

```
<buildtool_depend>ament_cmake</
    buildtool_depend>

<depend>rclcpp</depend>
<depend>std_msgs</depend>
```

CMakeLists.txt: librería + ejecutables

- Compila nodos como **librería** reutilizable.
- Cada `main` es un ejecutable que decide qué nodos instanciar.
- `install(TARGETS ...)` permite `ros2 run <pkg> <exe>`.

Fragmento real (resumido)

```
1  find_package(rclcpp REQUIRED)
2  find_package(std_msgs REQUIRED)
3
4  add_library(${PROJECT_NAME}
5    src/ch2_examples/LoggerNode.cpp
6    src/ch2_examples/PublisherNode.cpp
7    src/ch2_examples/SubscriberNode.cpp)
8
9  add_executable(logger src/logger_main.cpp)
10 target_link_libraries(logger ${
       PROJECT_NAME})
```

LoggerNode: un nodo con timer y estado

- `rclcpp::Node` encapsula APIs de ROS 2 (logging, pub/sub, timers...).
- El timer registra un callback periódico.
- `counter_` es **estado interno**: se actualiza dentro del nodo.

LoggerNode.hpp (real)

```
1  #include "rclcpp/rclcpp.hpp"
2
3  class LoggerNode : public rclcpp::Node
4  {
5  public:
6    LoggerNode();
7    void timer_callback();
8
9  private:
10   rclcpp::TimerBase::SharedPtr timer_;
11   int counter_;
12 };
```

Timers y *tiempo real de pared* (wall time)

- **Wall time**: reloj del sistema ("tiempo de pared").
- **Steady time**: monótono (no salta si cambia la hora).
- **ROS/sim time**: controlado por simulación (`/clock`); puede pausarse o acelerarse.
- En los ejemplos usamos `create_wall_timer` para que funcionen igual sin simulador.

create_wall_timer (real)

```
using namespace std::chrono_literals;

timer_ = create_wall_timer(
  500ms, std::bind(&LoggerNode::
       timer_callback, this));
```

Callback (real)

```
RCLCPP_INFO(get_logger(), "Hello %d",
     counter_++);
```

Despliegue: `main` + `spin`

- El `main` inicializa ROS 2 y crea el nodo.
- `spin` entra en el bucle del executor y atiende callbacks.
- Importante: `rclcpp::spin(node)` es un **atajo** que por debajo crea un executor, añade el nodo y hace `spin()`.
- `shutdown` libera recursos y finaliza ROS correctamente.

logger_main.cpp (real)

```
1  int main(int argc, char * argv[])
2  {
3    rclcpp::init(argc, argv);
4    auto node = std::make_shared<LoggerNode
         >();
5    rclcpp::spin(node);
6    rclcpp::shutdown();
7    return 0;
8  }
```

El modelo de ejecución reactivo en `rclcpp`

- En ROS 2 programamos **reacciones** a eventos:
  - llega un mensaje → callback de suscripción
  - vence un temporizador → callback del timer
- El executor es el bucle que:
  - monitoriza fuentes de eventos
  - decide cuándo ejecutar cada callback
- En esta práctica: monohilo (simplicidad, determinismo).
- **Idea clave:** el **executor** espera eventos y despacha callbacks (los nodos no *corren* solos).

Lo que hace `rclcpp::spin(node)` (idea)

```
rclcpp::executors::SingleThreadedExecutor ex
    ;
ex.add_node(node);
ex.spin();
```

Atajo habitual: `rclcpp::spin(node);`

PublisherNode: publicación periódica

- Un *publisher* publica mensajes en un topic.
- El timer dispara la publicación cada periodo.
- El 10 es un QoS simple (profundidad del historial).
- En la práctica, el publisher será `/cmd_vel` con `geometry_msgs/msg/Twist`.

PublisherNode.cpp (real)

```
1  publisher_ = create_publisher<std_msgs::
       msg::Int32>(
2    "counter", 10);
3
4  timer_ = create_wall_timer(
5    100ms, std::bind(&PublisherNode::
         timer_callback, this));
6
7  void PublisherNode::timer_callback()
8  {
9    RCLCPP_INFO(get_logger(), "Publishing %
         d", counter_message_.data++);
10   publisher_->publish(counter_message_);
11 }
```

SubscriberNode: callback por evento

- Un *subscriber* registra un callback.
- El callback se ejecuta **cuando llega un mensaje**.
- Este patrón es el que usarás con el bumper: evento → actualiza estado.
- En monohilo, callbacks se ejecutan secuencialmente.

SubscriberNode.cpp (real)

```
1  counter_subscription_ =
       create_subscription<std_msgs::msg::
       Int32>(
2    "counter", 10,
3    std::bind(&SubscriberNode::
         subscription_callback,
4             this, std::placeholders::_1));
5
6  void SubscriberNode::
       subscription_callback(
7    const std_msgs::msg::Int32::SharedPtr
         msg)
8  {
9    RCLCPP_INFO(get_logger(), "Received %d"
         , msg->data);
10 }
```

Varios nodos en el mismo proceso: `SingleThreadedExecutor`

- Puedes instanciar varios nodos en un mismo ejecutable.
- El executor monitoriza eventos de **todos** los nodos añadidos.
- Elegimos monohilo por simplicidad y facilidad de depuración.

pub_sub_main.cpp (real)

```
1  auto pub_node = std::make_shared<
       PublisherNode>();
2  auto sub_node = std::make_shared<
       SubscriberNode>();
3
4  rclcpp::executors::SingleThreadedExecutor
       executor;
5  executor.add_node(pub_node);
6  executor.add_node(sub_node);
7  executor.spin();
```

Aplicación directa a la práctica del bumper

- **Evento**: llega un mensaje del bumper.
- **Estado**: guardas el *último estado conocido* (izq/centro/der).
- **Control 10 Hz**: timer consulta ese estado y calcula velocidades.
- **Actuación**: publicas `Twist` en `/cmd_vel`.

Regla de oro

El robot nunca se queda *sin comando*: el timer publica siempre.

Esqueleto mental (pseudocódigo)

```
1  void on_bumper_msg(...) {
2    bumper_state_ = ...; // evento ->
         estado
3  }
4
5  void on_timer_10hz() {
6    auto cmd = compute_cmd(bumper_state_);
7    cmd_vel_pub_->publish(cmd); // estado
         -> control
8  }
```

Preguntas