### **Nodo de Comunicación Serial (communication\_node)**

#### **Descripción**

Este nodo actúa como el intermediario exclusivo entre el sistema y el hardware del robot a través del puerto serial. Centraliza el acceso al puerto serie, asegurando que solo un nodo interactúe directamente con el robot. Publica datos provenientes de los sensores y recibe comandos de otros nodos para controlar el robot.

#### **Funciones**

1. Gestionar el acceso exclusivo al puerto serial para evitar conflictos entre nodos.
2. Publicar datos de los sensores y el estado del robot, como movimiento, energía y seguridad.
3. Recibir y procesar comandos de otros nodos, como movimiento, display, sonidos y LEDs, para enviarlos al robot.
4. Gestionar la prioridad de los mensajes para garantizar la ejecución eficiente de las acciones críticas.

#### **Interfaces**

**Parámetros:**

* connection\_serial: Nombre del puerto serial utilizado para conectarse al robot.
* connection\_mode: Modo de conexión con el robot:
  + **FULL:** Acceso completo al robot.
  + **SAFE:** Modo de seguridad, limita las acciones cuando los sensores de seguridad se activan.
* movement\_frequency: Frecuencia de publicación de los datos de movimiento (en Hz).
* security\_frequency: Frecuencia de publicación de los datos de seguridad (en Hz).
* power\_frequency: Frecuencia de publicación de los datos de energía (en Hz).
* misc\_frequency: Frecuencia de publicación de datos misceláneos (en Hz).
* state\_communication\_frequency: Frecuencia de publicación de datos estado (en Hz).
* movement\_filter: Lista de sensores de movimiento a monitorear.
* security\_filter: Lista de sensores de seguridad a monitorear.
* power\_filter: Lista de sensores de energía a monitorear.
* misc\_filter: Lista de otros sensores, como LEDs y botones.

**Tópicos:**

* **Publica:**
  + /roomba/communication**/**security: Datos de sensores relacionados con caídas, colisiones y proximidad.
    - Ejemplo: Señales de sensores de acantilado y colisión.
  + /roomba/communication**/**movement: Datos de los encoders, distancia y ángulo recorrido por el robot.
  + /roomba/communication**/**power: Información relacionada con el estado de la batería (carga, voltaje, corriente, temperatura).
  + /roomba/communication**/**misc: Datos misceláneos como estado de los botones, LEDs y sonidos.
  + /roomba/communication**/**state: Datos relacionados con el estado del nodo..
* **Suscribe:**
  + /roomba/motion/motors\_command: Recibe comandos de movimiento, como velocidades de las ruedas.
  + /roomba/misc/leds\_command: Recibe comandos relacionados con el display y los LEDs.
  + /roomba/misc/song\_command: Recibe comandos para reproducir sonidos y canciones.

**Servicios:**

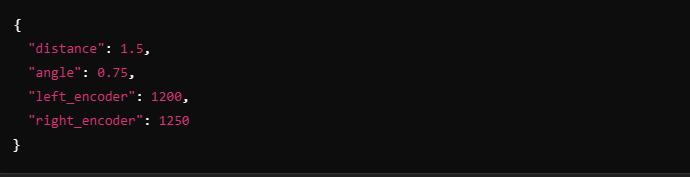
* /roomba/communication/connect: Establece la conexión inicial con el robot.
* /roomba/communication/priority: Permite ajustar las prioridades de los mensajes.
* /roomba/system/reconfigure Permite actualizar dinámicamente los

#### **Interacción**

* **Con otros nodos:**
  + **Nodo Maestro:**
    - Establece la conexión inicial con el robot.
    - Recibe comandos para configurar la conexión serial y ajustar parámetros.
  + **Nodo de Movimiento:**
    - Recibe comandos relacionados con velocidades y motores.
    - Publica datos de movimiento como encoders, distancia y ángulo.
  + **Nodo de Seguridad:**
    - Publica datos de sensores de colisiones y caídas.
    - Recibe comandos para detener el robot en caso de emergencia.
  + **Nodo de Energía:**
    - Publica datos de energía como carga de batería, voltaje y temperatura.
  + **Nodo Misceláneo:**
    - Publica datos relacionados con botones, LEDs y sonidos.
* **Con el hardware del robot:**
  + Envia comandos seriales utilizando los protocolos del Open Interface.
  + Procesa y publica las respuestas recibidas desde el robot.

#### **Ejemplo de Datos Publicados**

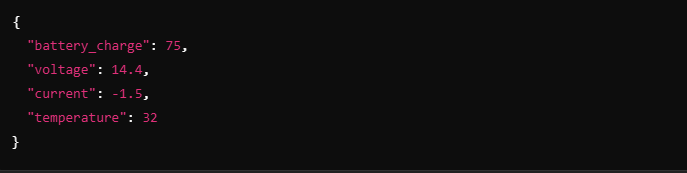
**Tópico /roomba/movement:**



Tópico /roomba/security:



Tópico /roomba/power:



#### **Consideraciones Técnicas:**

1. **Prioridades:** Utiliza el servicio /roomba/system/priority para priorizar mensajes críticos (por ejemplo, datos de sensores de seguridad sobre comandos de movimiento).
2. **Frecuencia de Publicación:** Configura las frecuencias de publicación (movement\_frequency, security\_frequency, etc.) según las necesidades del sistema.
3. **Modo de Conexión:** Define el modo de conexión más adecuado (FULL o SAFE) según el entorno y las posibles restricciones de seguridad.
4. **Errores de Conexión:** Implementa manejo de errores robusto en el servicio /roomba/system/connect para gestionar fallos en la conexión serial.

### **1. Gestión de la Conexión Serial**

* Implementaremos compatibilidad para dos modos de conexión:
  + **USB Serial directo**.
  + **Redireccionamiento TCP mediante socat** (esto será el predeterminado por ser el más usado).
* Un parámetro dinámico (connection\_mode) permitirá cambiar entre FULL y SAFE durante la ejecución del nodo.

### **2. Frecuencias de Publicación**

* Las frecuencias iniciales se establecerán como:
  + **Movimiento (movement\_frequency)**: 10 Hz.
  + **Seguridad (security\_frequency)**: 5 Hz.
  + **Energía (power\_frequency)**: 1 Hz.
  + **Misceláneos (misc\_frequency)**: 1 Hz.
  + Estos valores se ajustarán dinámicamente según la necesidad mediante un servicio.
* Estas frecuencias son configurables como parámetros para mayor flexibilidad.

### **3. Filtros de Sensores**

* Cada tópico publicará únicamente los valores de los sensores incluidos en su filtro correspondiente (movement\_filter, security\_filter, etc.).
* El filtro es un parámetro dinámico y podrá ajustarse en tiempo de ejecución.

### **4. Sistema de Colas y Prioridades**

* Usaremos un sistema de **colas de mensajes** para manejar las prioridades de las publicaciones:
  + **Mensajes críticos (seguridad)** tendrán prioridad más alta.
  + **Datos de movimiento, energía y misceláneos** tendrán prioridad media o baja.
* La cola de mensajes priorizará los datos basándose en un parámetro configurable, lo que permitirá ajustar el comportamiento del sistema.

### **5. Reconfiguración Dinámica Persistente**

* Parámetros como la frecuencia de publicación, el modo de conexión y los filtros de sensores se guardarán y se restaurarán al reiniciar el nodo.
* Esto será manejado mediante los servicios roomba/communication/reconfigure y roomba/communication/priority.

### **6. Mensajes ROS2**

* Crearemos mensajes personalizados para cada tópico (security, movement, power, misc) para reflejar los datos de los sensores tal como se reciben del robot.
* Estructuraremos los mensajes según las propiedades del robot, como se describe en las librerías compartidas.

### **7. Gestión de Errores**

* Implementaremos las siguientes estrategias:
  + **Desconexión Serial**: Intentos de reconexión automáticos con un contador máximo de reintentos.
  + **Fallos de Sensores**: Publicación de un mensaje de error con el identificador del sensor afectado en un tópico de diagnóstico.

### **8. Pruebas**

* El nodo incluirá un modo de prueba para observar los mensajes publicados en los tópicos y validar la recepción de comandos de movimiento.
* Los comandos enviados se reflejarán en los logs para verificar el flujo completo.

import rclpy

from rclpy.node import Node

from rclpy.parameter import Parameter

from std\_msgs.msg import String

from std\_msgs.msg import Float32

from roomba\_msgs.msg import Movement, Security, Power, Misc # Mensajes personalizados que se definirán

from roomba\_msgs.srv import Connect, Priority, Reconfigure

import threading

from lib\_irobot import connect\_robot, leer\_sensores\_basicos, leer\_estado\_bateria, leer\_estado\_movimiento, leer\_proximidad

class CommunicationNode(Node):

def \_\_init\_\_(self):

super().\_\_init\_\_('communication\_node')

# Parámetros configurables

self.declare\_parameter('connection\_serial', '/dev/roomba')

self.declare\_parameter('connection\_mode', 'FULL')

self.declare\_parameter('movement\_frequency', 10.0)

self.declare\_parameter('security\_frequency', 5.0)

self.declare\_parameter('power\_frequency', 1.0)

self.declare\_parameter('misc\_frequency', 1.0)

self.declare\_parameter('movement\_filter', [])

self.declare\_parameter('security\_filter', [])

self.declare\_parameter('power\_filter', [])

self.declare\_parameter('misc\_filter', [])

# Publicadores

self.movement\_pub = self.create\_publisher(Movement, '/roomba/communication/movement', 10)

self.security\_pub = self.create\_publisher(Security, '/roomba/communication/security', 10)

self.power\_pub = self.create\_publisher(Power, '/roomba/communication/power', 10)

self.misc\_pub = self.create\_publisher(Misc, '/roomba/communication/misc', 10)

# Servicios

self.connect\_service = self.create\_service(Connect, '/roomba/communication/connect', self.connect\_callback)

self.priority\_service = self.create\_service(Priority, '/roomba/communication/priority', self.priority\_callback)

self.reconfigure\_service = self.create\_service(Reconfigure, '/roomba/system/reconfigure', self.reconfigure\_callback)

# Conexión inicial

self.robot = None

self.connection\_thread = None

self.stop\_threads = threading.Event()

# Timers de publicación

self.movement\_timer = None

self.security\_timer = None

self.power\_timer = None

self.misc\_timer = None

# Inicializar conexión

self.initialize\_connection()

def initialize\_connection(self):

serial\_port = self.get\_parameter('connection\_serial').get\_parameter\_value().string\_value

mode = self.get\_parameter('connection\_mode').get\_parameter\_value().string\_value

try:

self.robot = connect\_robot(serial\_port)

if mode == 'FULL':

self.robot.start()

self.robot.oi\_mode = MODES.FULL

elif mode == 'SAFE':

self.robot.start()

self.robot.oi\_mode = MODES.SAFE

self.get\_logger().info(f'Conectado al robot en {serial\_port} en modo {mode}')

# Iniciar publicación de datos

self.start\_timers()

except Exception as e:

self.get\_logger().error(f'Error al conectar con el robot: {e}')

def start\_timers(self):

self.movement\_timer = self.create\_timer(1.0 / self.get\_parameter('movement\_frequency').value, self.publish\_movement\_data)

self.security\_timer = self.create\_timer(1.0 / self.get\_parameter('security\_frequency').value, self.publish\_security\_data)

self.power\_timer = self.create\_timer(1.0 / self.get\_parameter('power\_frequency').value, self.publish\_power\_data)

self.misc\_timer = self.create\_timer(1.0 / self.get\_parameter('misc\_frequency').value, self.publish\_misc\_data)

def publish\_movement\_data(self):

if self.robot:

data = leer\_estado\_movimiento(self.robot)

msg = Movement()

msg.distance = data['distance']

msg.angle = data['angle']

msg.left\_encoder\_counts = data['left\_encoder\_counts']

msg.right\_encoder\_counts = data['right\_encoder\_counts']

self.movement\_pub.publish(msg)

def publish\_security\_data(self):

if self.robot:

data = leer\_sensores\_basicos(self.robot)

msg = Security()

msg.bump\_left = data['bump\_left']

msg.bump\_right = data['bump\_right']

msg.wheel\_drop\_left = data['wheel\_drop\_left']

msg.wheel\_drop\_right = data['wheel\_drop\_right']

msg.wall = data['wall']

msg.virtual\_wall = data['virtual\_wall']

self.security\_pub.publish(msg)

def publish\_power\_data(self):

if self.robot:

data = leer\_estado\_bateria(self.robot)

msg = Power()

msg.voltage = data['voltage']

msg.current = data['current']

msg.temperature = data['temperature']

msg.charge = data['charge']

msg.capacity = data['capacity']

self.power\_pub.publish(msg)

def publish\_misc\_data(self):

if self.robot:

# Agregar lógica de datos misceláneos si se define

pass

def connect\_callback(self, request, response):

try:

if self.robot:

self.robot.stop()

self.robot = connect\_robot(request.port)

if request.mode == 'FULL':

self.robot.start()

self.robot.oi\_mode = MODES.FULL

elif request.mode == 'SAFE':

self.robot.start()

self.robot.oi\_mode = MODES.SAFE

response.success = True

response.message = 'Conexión exitosa'

except Exception as e:

response.success = False

response.message = str(e)

return response

def priority\_callback(self, request, response):

# Implementar manejo de prioridades según request.message\_type y request.priority

response.success = True

response.message = 'Prioridad actualizada'

return response

def reconfigure\_callback(self, request, response):

try:

if hasattr(self, request.parameter):

self.set\_parameter(Parameter(request.parameter, Parameter.Type.DOUBLE, request.value))

response.success = True

response.message = 'Parámetro reconfigurado'

else:

response.success = False

response.message = 'Parámetro no encontrado'

except Exception as e:

response.success = False

response.message = str(e)

return response

def destroy\_node(self):

self.stop\_threads.set()

if self.robot:

self.robot.stop()

super().destroy\_node()

def main(args=None):

rclpy.init(args=args)

node = CommunicationNode()

rclpy.spin(node)

node.destroy\_node()

rclpy.shutdown()

if \_\_name\_\_ == '\_\_main\_\_':

main()

He creado un nodo básico llamado CommunicationNode con las siguientes características:

1. **Parámetros configurables** para la conexión, frecuencias de publicación y filtros de sensores.
2. **Publicadores** para los tópicos de movimiento, seguridad, energía y misceláneos.
3. **Servicios** para la conexión, reconfiguración y manejo de prioridades.
4. **Timers** para publicar datos a las frecuencias definidas.
5. **Gestión de la conexión** inicial y reconexión mediante un servicio.

Próximos pasos:

* **Definir mensajes personalizados** (Movement, Security, etc.) según los datos descritos.
* **Implementar más lógica en misceláneos y prioridades** si es necesario.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### **1. Movimiento (Movement)**

Incluye información sobre:

* Distancia recorrida.
* Ángulo girado.
* Contadores de los encoders.

# Archivo: msg/Movement.msg

float32 distance

float32 angle

int32 left\_encoder\_counts

int32 right\_encoder\_counts

### **2. Seguridad (Security)**

Incluye información sobre:

* Sensores de colisión.
* Caídas y paredes.
* Sensores de virtual wall.

# Archivo: msg/Security.msg

bool bump\_left

bool bump\_right

bool wheel\_drop\_left

bool wheel\_drop\_right

bool wall

bool virtual\_wall

### **3. Energía (Power)**

Información del estado de la batería, voltaje y temperatura.

# Archivo: msg/Power.msg

float32 voltage

float32 current

float32 temperature

float32 charge

float32 capacity

### **4. Misceláneos (Misc)**

Datos de proximidad y otros sensores no categorizados.

# Archivo: msg/Misc.msg

int32 cliff\_left\_signal

int32 cliff\_front\_left\_signal

int32 cliff\_front\_right\_signal

int32 cliff\_right\_signal

int32 wall\_signal

### **5. Estado (State)**

Información sobre el estado general del robot.

# Archivo: msg/State.msg

string oi\_mode

bool is\_charging

float32 battery\_charge

### **Siguientes pasos:**

1. Crear estos archivos de mensaje en la carpeta msg de tu paquete ROS2.
2. Modificar el archivo CMakeLists.txt y package.xml para incluirlos.
3. Generar los mensajes al compilar el paquete con colcon build.

### **Modificaciones en CMakeLists.txt**

1. **Añadir dependencias de mensajes:**
   * Asegúrate de que el paquete tiene dependencias necesarias como std\_msgs.
2. **Incluir los archivos de mensaje:**
   * Define los archivos .msg creados.
3. **Configurar la generación de mensajes:**

cmake\_minimum\_required(VERSION 3.5)

project(roomba\_msgs)

# Encuentra los paquetes necesarios

find\_package(ament\_cmake REQUIRED)

find\_package(rosidl\_default\_generators REQUIRED)

find\_package(std\_msgs REQUIRED)

# Especificar los archivos de mensaje

set(msg\_files

"msg/Movement.msg"

"msg/Security.msg"

"msg/Power.msg"

"msg/Misc.msg"

"msg/State.msg"

)

# Generar mensajes

rosidl\_generate\_interfaces(${PROJECT\_NAME}

${msg\_files}

DEPENDENCIES std\_msgs

)

# Incluir directorios

ament\_package()

### **Modificaciones en package.xml**

1. **Agregar dependencias para la generación de mensajes:**

# Estructura de archivos del paquete `roomba\_msgs`

## package.xml

<package format="2">

<name>roomba\_msgs</name>

<version>0.1.0</version>

<description>Mensajes personalizados para la comunicación con Roomba.</description>

<maintainer email="tuemail@ejemplo.com">Tu Nombre</maintainer>

<license>Apache-2.0</license>

<!-- Dependencias -->

<buildtool\_depend>ament\_cmake</buildtool\_depend>

<build\_depend>rosidl\_default\_generators</build\_depend>

<exec\_depend>rosidl\_default\_runtime</exec\_depend>

<depend>std\_msgs</depend>

<export>

<build\_type>ament\_cmake</build\_type>

</export>

</package>

## CMakeLists.txt

cmake\_minimum\_required(VERSION 3.5)

project(roomba\_msgs)

# Encuentra los paquetes necesarios

find\_package(ament\_cmake REQUIRED)

find\_package(rosidl\_default\_generators REQUIRED)

find\_package(std\_msgs REQUIRED)

# Especificar los archivos de mensaje

set(msg\_files

"msg/Movement.msg"

"msg/Security.msg"

"msg/Power.msg"

"msg/Misc.msg"

"msg/State.msg"

)

# Generar mensajes

rosidl\_generate\_interfaces(${PROJECT\_NAME}

${msg\_files}

DEPENDENCIES std\_msgs

)

# Incluir directorios

ament\_package()

## Directorio msg/

# Crear un directorio llamado `msg/` y añadir los siguientes archivos:

# msg/Movement.msg

float32 distance

float32 angle

int32 left\_encoder\_counts

int32 right\_encoder\_counts

# msg/Security.msg

bool bump\_left

bool bump\_right

bool wheel\_drop\_left

bool wheel\_drop\_right

bool wall

bool virtual\_wall

# msg/Power.msg

float32 voltage

float32 current

float32 temperature

float32 charge

float32 capacity

# msg/Misc.msg

int32 cliff\_left\_signal

int32 cliff\_front\_left\_signal

int32 cliff\_front\_right\_signal

int32 cliff\_right\_signal

int32 wall\_signal

# msg/State.msg

string oi\_mode

bool is\_charging

float32 battery\_charge

### **1. Diseño del sistema de prioridades**

Las prioridades se asignarán por tipo de mensaje y se manejarán de esta manera:

* **Alta prioridad:** Mensajes críticos como seguridad (e.g., evitar caídas o colisiones).
* **Media prioridad:** Comandos de movimiento y estado (e.g., motores, modos).
* **Baja prioridad:** Misceláneos y comandos de LEDs o música.

### **2. Implementación de colas**

Cada tipo de mensaje tendrá una cola individual, y se procesarán en el siguiente orden:

1. Procesar mensajes de alta prioridad hasta que la cola esté vacía.
2. Procesar mensajes de media prioridad.
3. Procesar mensajes de baja prioridad.

### **3. Uso de Python queue.PriorityQueue**

La clase PriorityQueue será utilizada para gestionar las colas. Los elementos en la cola estarán en forma de (priority, data) para que se procesen en el orden correcto.

### **4. Ejemplo de código**

Voy a incluir el manejo de prioridades y colas en el nodo CommunicationNode.

#### **Modificaciones al nodo:**

python

CopyEdit

import queue

class CommunicationNode(Node):

def \_\_init\_\_(self):

super().\_\_init\_\_('communication\_node')

# Inicializar colas de prioridad

self.message\_queue = queue.PriorityQueue()

# Configurar niveles de prioridad

self.priority\_levels = {

"security": 1, # Alta prioridad

"movement": 2, # Media prioridad

"misc": 3, # Baja prioridad

}

# Timers para procesar la cola

self.create\_timer(0.1, self.process\_message\_queue)

def enqueue\_message(self, priority, data):

"""Agregar un mensaje a la cola con su prioridad."""

self.message\_queue.put((priority, data))

def process\_message\_queue(self):

"""Procesar los mensajes en la cola según su prioridad."""

while not self.message\_queue.empty():

priority, data = self.message\_queue.get()

self.get\_logger().info(f'Procesando mensaje con prioridad {priority}: {data}')

# Lógica específica según el tipo de mensaje

if data["type"] == "security":

self.handle\_security\_message(data)

elif data["type"] == "movement":

self.handle\_movement\_message(data)

elif data["type"] == "misc":

self.handle\_misc\_message(data)

def handle\_security\_message(self, data):

self.get\_logger().info(f'Manejando mensaje de seguridad: {data}')

def handle\_movement\_message(self, data):

self.get\_logger().info(f'Manejando mensaje de movimiento: {data}')

def handle\_misc\_message(self, data):

self.get\_logger().info(f'Manejando mensaje misceláneo: {data}')

# Ejemplo: Callback para recibir mensajes de seguridad y encolarlos

def security\_callback(self, msg):

data = {"type": "security", "content": msg.data}

self.enqueue\_message(self.priority\_levels["security"], data)

def movement\_callback(self, msg):

data = {"type": "movement", "content": msg.data}

self.enqueue\_message(self.priority\_levels["movement"], data)

def misc\_callback(self, msg):

data = {"type": "misc", "content": msg.data}

self.enqueue\_message(self.priority\_levels["misc"], data)

### **5. Flujo de ejecución**

1. Los mensajes de sensores y comandos entran a las colas respectivas mediante los callbacks (security\_callback, movement\_callback, etc.).
2. Un timer procesa los mensajes en orden de prioridad desde la cola.
3. Cada mensaje se maneja con una función específica (e.g., handle\_security\_message).

### **Ventajas de este enfoque**

* Evita la saturación de recursos al priorizar mensajes importantes.
* Garantiza que mensajes críticos como seguridad se manejen antes que los secundarios.
* Escalable para nuevos tipos de mensajes.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Nodo2.py

El nodo ha sido actualizado para incluir un sistema de **colas de prioridades** con los siguientes ajustes:

1. **Colas de prioridad:**
   * Los mensajes ahora se agregan a una cola prioritaria basada en su importancia (seguridad, movimiento y misceláneos).
   * El procesamiento se realiza de acuerdo con los niveles de prioridad definidos.
2. **Callbacks actualizados:**
   * Los callbacks de tópicos encolan los mensajes con su nivel de prioridad respectivo.
3. **Procesador de mensajes:**
   * Un timer procesa la cola cada 0.1 segundos, manejando los mensajes de acuerdo con su tipo.