# Anexo IV: Documentación técnica de programación

# App para la navegación basada en comandos de voz del robot iRobot Create 3

Trabajo de Fin de Grado de Ingeniería Informática

VNiVERSiDAD D SALAMANCA

Julio de 2023

**Autor:**

Jorge Sánchez Rubio

**Tutores:**

Francisco Javier Blanco Rodríguez

Belén Curto Diego

# ÍNDICE DE CONTENIDOS

## Índice de figuras

# 1. INTRODUCCIÓN

Este anexo describe la estructura del proyecto a nivel de código con el objetivo de facilitar al programador la comprensión del software empleado para su implementación, modificación o ampliación.

El proyecto se divide en dos partes. La parte de la aplicación Android, que es con la que el usuario se comunica y la parte del servidor de ROS2, donde se encuentra la lógica de control del robot. En este anexo se explicará el funcionamiento de ambas.

# 2. APLICACIÓN ANDROID

En los siguientes apartados se explican cada uno de los aspectos relevantes de la aplicación Android diseñada; tanto el entorno de desarrollo en el que se ha trabajado como los principales archivos y funciones necesarias para el correcto funcionamiento de la aplicación.

## 2.1. Android Studio

El entorno de desarrollo integrado (IDE) que se ha decidido emplear para el desarrollo de la aplicación es Android Studio puesto que es el IDE oficial que se usa en el desarrollo de aplicaciones Android.


*Figura 1: Android Studio IDE*

Android Studio ofrece una renderización en tiempo real, así como una consola de desarrollador que ofrece consejos de optimización de código. También ofrece un sistema de compilación flexible basado en Gradle y un emulador rápido y lleno de funciones que permite al desarrollador probar la aplicación en un entorno seguro.

El lenguaje empleado para la aplicación en la parte lógica es Java, puesto que es el lenguaje nativo de Android, y XML para la interfaz gráfica de usuario ya que ambos son bien conocidos y objeto de estudio en el grado y son sencillos de manejar.

## 2.2. Compilación y ejecución de la aplicación

Para la compilación y ejecución de la aplicación en Android Studio basta con hacer clic en el botón '`Run app`' o pulsando '`Mayus + F10`'.

A la izquierda de este botón se puede seleccionar el dispositivo donde se instalará y ejecutará la aplicación pudiendo ser un dispositivo virtual de entre los que ofrece el propio IDE o a través del smartphone físico elegido. Para este último es necesario que estén activadas las opciones de depuración USB.

## 2.3. build.gradle

El sistema de compilación de Gradle de Android Studio [1] facilita al desarrollador incluir archivos binarios externos u otras librerías en la compilación que funcionan como dependencias. Estas dependencias pueden estar ubicadas en el equipo o en un repositorio remoto. Cualquier dependencia transitiva que se declare también se incluirá.

El archivo build.gradle es un archivo de texto sin formato que usa un lenguaje específico de dominio (DSL) [2] para describir y manipular la lógica de compilación mediante Groovy, que es un lenguaje dinámico para la máquina virtual Java (JVM). Existen dos formatos de este archivo que se encuentran en el directorio raíz del proyecto, en '`Gradle Scripts`'.

El build.gradle de nivel superior se reconoce como (`Project: iRobotCreate3`), y define las dependencias que se aplican a todos los módulos del proyecto. En este proyecto es el siguiente (Figura 2):

```
// Top-level build file where you can add configuration options
plugins {
    id 'com.android.application' version '7.4.2' apply false
    id 'com.android.library' version '7.4.2' apply false
}
```

*Figura 2: build.gradle de nivel superior*

El build.gradle de nivel de módulo, que se conoce como (`Module:app`), permite configurar ajustes de compilación para el módulo específico en el que se encuentra y añadir dependencias adicionales.

Hay que destacar varios aspectos de este archivo en el proyecto:

- En la sección ‘`defaultConfig`’ se configuran las opciones predeterminadas de compilación de la aplicación. Dentro de todos los parámetros a tener en cuenta, uno de los más importantes es el parámetro ‘`targetSdk`’ que especifica la versión del SDK de Android. En el proyecto se ha especificado la versión 33 (Figura 3).

```
android {
    namespace 'com.example.irc3'
    compileSdk 33

    defaultConfig {
        applicationId "com.example.irc3"
        minSdk 24
        targetSdk 33
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }
```

*Figura 3: Sección defaultConfig*

- En la sección de dependencias ‘`dependencies`’(Figura 4), además de las que vienen por defecto con la creación de un proyecto en Android Studio, se ha añadido la siguiente línea:

  ‘`implementation‘ com.airbnb.android:lottie:6.0.0’’`

  De esta forma se incluye la biblioteca Lottie [3] , que es una biblioteca de código libre desarrollada por la empresa Airbnb, para permitir la reproducción de animaciones utilizando archivos JSON en dispositivos. Esta biblioteca permite utilizar un archivo JSON como animación de un micrófono para reproducirla en la interfaz de usuario cuando se pulsa.

```
dependencies {

    implementation 'androidx.appcompat:appcompat:1.6.1'
    implementation 'com.google.android.material:material:1.9.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.4'
    implementation 'com.airbnb.android:lottie:6.0.0'
    testImplementation 'junit:junit:4.13.2'
    androidTestImplementation 'androidx.test.ext:junit:1.1.5'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.5.1'
}
```

*Figura 4: Sección de dependencias*

## 2.4. AndroidManifest.xml

Este archivo ubicado en la carpeta ‘`manifest`’ del proyecto describe la información esencial de la aplicación para las herramientas de creación de Android, el sistema operativo Android y Google Play. En este archivo de manifiesto [4] se declaran atributos como el nombre de la aplicación (la etiqueta ‘`android:label`’ o los componentes de la aplicación los cuales incluyen todas las actividades, servicios y permisos que necesita la aplicación para acceder a las partes protegidas del sistema o a otras aplicaciones.

Concretamente para este proyecto se han añadido los permisos de acceso a Internet y la grabación de audio para la captación de voz y transcripción de voz a texto por medio de los Servicios de Voz de Google que Android utiliza. En la siguiente figura se muestra la parte del código de este archivo en la que se declaran estos permisos. En función de si se otorgan o no por el usuario, los intentos de acceder a las funciones que necesiten estos permisos fallarán.

```
<uses-permission android:name="android.permission.RECORD_AUDIO" />
<uses-permission android:name="android.permission.INTERNET" />
```

*Figura 5: Permisos añadidos*

## 2.5. activity_main.xml

Se trata de un archivo de diseño que se encuentra en la carpeta ‘`layout`’ y es donde se diseña la interfaz gráfica de usuario de la aplicación. Está escrito en lenguaje XML y permite separar la presentación de la aplicación del código que controla el comportamiento. De esta manera, el framework de Android permite la flexibilidad de declarar los diseños predeterminados de la app y luego modificarlos durante la ejecución a través de la instanciación de estos componentes [5] .

En el caso de este proyecto, en el archivo `activity_main.xml` se definen todos los componentes principales de la interfaz gráfica de la aplicación, así como los elementos de realimentación para que el usuario conozca el estado de la orden que ha mandado al robot. Cada componente tiene un identificador y sus respectivos parámetros. El resto de la información respecto al funcionamiento de este archivo se encuentra en el Anexo III – Especificación de diseño.

## 2.6. config_realimentacion.xml

Este archivo que se encuentra en la carpeta ‘`layout`’ define un componente `SwitchCompat` [6] [7] que es utilizado en el cuadro de diálogo que surge cuando el usuario pulsa el botón de configuración. Sirve para establecer si el usuario quiere o no ver el estado de la orden dada en la aplicación. Por lo tanto, puede tomar dos valores, verdadero o falso. En la Figura 6 se muestra la definición del componente.

```
<androidx.appcompat.widget.SwitchCompat
    android:id="@+id/switchConfig"
    android:layout_width="229dp"
    android:layout_height="59dp"
    android:text="          No/Sí"
    android:textSize="17sp"

    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

*Figura 6: Componente SwitchCompat del archivo config_realimentación.xml*

## 2.7. texto_ip.xml

Este archivo que se encuentra en la carpeta ‘`layout`’ es utilizado para definir un cuadro de texto en el que el usuario introduce la IP del ordenador que ejecuta el servidor de ROS2. Se muestra cuando el usuario pulsa sobre el icono de configuración de la IP al abrirse una ventana.

## 2.8. Carpeta values

En esta carpeta se encuentran los archivos `colors.xml` y la carpeta ‘themes’ con el archivo `themes.xml`. Este tipo de archivos se utilizan para definir los colores que se quieran emplear para diseñar la interfaz gráfica de la aplicación y aplicarlos a los temas principales con mayor sencillez o añadir otros nuevos. En la Figura 7 y Figura 8 se muestra el ejemplo de una definición para un color y su posterior asignación a uno de los temas de la aplicación.

```
<color name="irobot">#6CB86A</color>
```

*Figura 7: Definición del color principal de la aplicación. Archivo colors.xml*

```
<item name="colorPrimary">@color/irobot</item>
```

*Figura 8: Asignación del color definido ‘irobot’ al tema ‘colorPrimary’. Archivo themes.xml*

## 2.9. MainActivity.java

La clase `Activity` es clave en una aplicación Android ya que con ella se inician y crean las actividades de la aplicación. Al contrario de los paradigmas de programación donde los programas se inician con el método `main()`, Android inicia el código en una instancia de la clase `Activity` [8].

Otra definición válida es que una `Activity` [9] en Android se corresponde con una pantalla de la aplicación que Android puede cargar en cualquier momento. Se compone de una clase que se extiende de la clase `AppCompatActivity` y de un `layout`, que define la vista y diseño de la aplicación.

En el caso de este proyecto el fichero `MainActivity.java` define la clase `MainActivity` que se extiende de la clase AppCompatActivity y tiene como layout el fichero `activity_main.xml`. En esta clase se define la lógica de la aplicación.

A continuación, se procede a explicar las partes más importantes.

### 2.9.1. Carga de la interfaz gráfica - setContentView()

Cuando se inicia la aplicación en el `MainActivity` a través del método `onCreate()`, se llama al método `setContentView()` [10] con el que se establece el contenido de la actividad principal en una vista explícita, es decir, establece el diseño de la interfaz de usuario que se muestra por pantalla al ejecutarse la actividad.

### 2.9.2. Comprobación de permisos – checkSelfPermission()

Cuando se inicia la aplicación, se realiza al comienzo de la actividad (en tiempo de ejecución) la comprobación de permisos [11] de grabación de audio para la aplicación a través del método `checkSelfPermission()`, que recibe como parámetros el contexto de la aplicación y el nombre del permiso y devuelve el estado del permiso `RECORD_AUDIO`. Este resultado se compara con la constante `PERMISSION_GRANTED`; de manera que, si no se tienen permisos, se solicita al usuario que otorgue a la aplicación el permiso mencionado a través de un cuadro de diálogo.

Esta solicitud al usuario se hace a través del método `requestPermissions()`, al que se le pasan los parámetros del contexto actual de la aplicación (información global del entorno de la aplicación) y el código de solicitud del permiso que sirve para identificar la solicitud del permiso en la posterior llamada al método `onRequestPermissionsResult()`. Se utiliza esta forma para administrar la solicitud de permiso por parte del usuario, aunque también lo podría hacer el sistema por sí mismo.

A continuación, se muestra en la Figura 9 la comprobación del permiso de grabación de audio de la aplicación y posible solicitud.

```
//Comprobación de permisos de audio para la aplicación, se devuelve el estado del permiso RECORD_AUDIO
if(ContextCompat.checkSelfPermission( context: this, Manifest.permission.RECORD_AUDIO) != PackageManager.PERMISSION_GRANTED){
    ActivityCompat.requestPermissions( activity: this, new String[]{Manifest.permission.RECORD_AUDIO}, requestCode: 1);
}
```

*Figura 9: Comprobación y solicitud del permiso de grabación de audio*

Una vez que el usuario responde al diálogo de permisos del sistema, se invoca la implementación del método mencionado anteriormente `onRequestPermissionsResult()`. El sistema pasa la respuesta del usuario al diálogo de permisos junto al código de solicitud definido.

En función de si se ha concedido o no el permiso, el método muestra una pequeña ventana emergente con un mensaje de aprobación o negación. Esto se realiza a través de la clase Toast.

En la siguiente figura podemos observar cómo está definido el método `onRequestPermissionsResult()`.

```
14 usages
@Override
public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);
    if (requestCode == 1) {
        if (grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            Toast.makeText( context: this, text: "Permiso concedido", Toast.LENGTH_SHORT).show();
        }
        else{
            Toast.makeText( context: this, text: "Permiso denegado", Toast.LENGTH_SHORT).show();
        }
    }
```

*Figura 10: onRequestPermissionsResult()*

### 2.9.3. Reconocimiento de voz – SpeechRecognizer

La clase `SpeechRecognizer` del paquete `android.speech` proporciona acceso al servicio de reconocimiento de voz de Android, que utiliza los Servicios de Voz de Google [12] . Esta clase debe ser invocada desde el hilo principal de la aplicación [13], y para poderse usar han de estar activos los permisos de grabación de audio mencionados anteriormente.

El primer paso es la creación de una instancia del objeto `SpeechRecognizer` definido en un primer momento en la clase

`MainActivity`, a través del método `createSpeechRecognizer()`. Para ello, primero se comprueba si el reconocimiento está habilitado mediante el método `isRecognitionAvailable()`. Se puede observar este paso en la Figura 11.

```
//Creación del objeto speech recognizer
if(SpeechRecognizer.isRecognitionAvailable( context: this)){
    speechRecognizer = SpeechRecognizer.createSpeechRecognizer( context: this);
}
else{
    Toast.makeText( context: this,  text: "Reconocimiento no habilitado", Toast.LENGTH_SHORT).show();
}
```

*Figura 11: Creación del SpeechRecognizer*

El siguiente paso es la creación de un objeto `Intent` [14]. Un `Intent` es un objeto de mensajería que se utiliza para solicitar una acción en otros componentes de la aplicación; de manera que se comunica con ellos.

Con su creación se establece el tipo de acción del `Intent`. En este caso se utiliza `ACTION_RECOGNIZE_SPEECH` para indicar al sistema que se desea iniciar una actividad de reconocimiento de voz.

Una vez el `Intent` es creado se pueden configurar los parámetros del reconocimiento de voz mediante el uso del método `putExtra()` [15] [16].

El uso de `EXTRA_LANGUAGE_MODEL` y `LANGUAGE_MODEL_FREE_FORM` en el método `putExtra()` sirve para establecer el modelo de reconocimiento de voz de manera que el sistema de reconocimiento de voz pueda interpretar el habla en un formato libre, sin restricciones específicas en términos de gramática o estructura de la oración.

Este modelo es ideal para aplicaciones que necesitan capturar la entrada de voz del usuario en texto sin restricciones.

El uso de `EXTRA_LANGUAGE` sirve para especificar el idioma de los modelos de lenguaje que el sistema de reconocimiento de voz va a utilizar. En este caso se usa el español de España “`es-ES`”.

En la Figura 12 se muestra el código de la explicación anterior.

```
//Creación del objeto Intent
final Intent speechRecognizerIntent = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);

//Configuración de los parámetros del reconocimiento de voz
speechRecognizerIntent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL, RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
speechRecognizerIntent.putExtra(RecognizerIntent.EXTRA_LANGUAGE, value: "es-ES");
```

*Figura 12: Creación del Intent y configuración del reconocimiento de voz*

Cabe destacar que la calidad del reconocimiento de voz depende de la calidad del micrófono y del entorno donde se realiza el reconocimiento.

### 2.9.4. Reconocimiento de voz – RecognitionListener

Una vez creado el `SpeechRecognizer` y el `Intent` y este último es configurado, el siguiente paso es manejar los eventos relacionados con el reconocimiento de voz. `RecognitionListener` es una interfaz que define una serie de métodos que son llamados durante el reconocimiento de voz y que se establece en un objeto `SpeechRecognizer` para manejar los eventos relacionados con el reconocimiento de voz. De manera que el `RecognitionListener` recibe notificaciones del `SpeechRecognizer` cuando los eventos relacionados con el reconocimiento ocurren [17] . En la Figura 13 se muestra cómo se establece un `RecognitionListener` al `SpeechRecognizer` utilizando el método `setRecognitionListener()`.

```
speechRecognizer.setRecognitionListener(new RecognitionListener() {
    @Override
    public void onReadyForSpeech(Bundle bundle) { tv.setText("Escuchando"); }

    @Override
    public void onBeginningOfSpeech() {
    }

    @Override
    public void onRmsChanged(float v) {
    }

    @Override
    public void onBufferReceived(byte[] bytes) {
    }

    @Override
    public void onEndOfSpeech() { speechRecognizer.stopListening(); }

    @Override
    public void onError(int i) {...}

    @Override
    public void onResults(Bundle resultado) {...}

    @Override
    public void onPartialResults(Bundle partialResults) {
    }

    @Override
    public void onEvent(int i, Bundle bundle) {
    }
});
```

*Figura 13: Creación y adjudicación del RecognitionListener al SpeechRecognizer*

Cuando se crea el `RecognitionListener` vienen por defecto una serie de métodos que son los que se utilizan para manejar los eventos del reconocimiento de voz. Los métodos que se han utilizado en este proyecto son:

- **onReadyForSpeech()** : Este método se ejecuta al iniciarse el reconocimiento de voz y usa el objeto `TextView` para mostrar al usuario por pantalla que el reconocimiento está siendo realizado.

```
@Override
public void onReadyForSpeech(Bundle bundle) { tv.setText("Escuchando"); }
```

*Figura 14: onReadyForSpeech()*

- **onError()** : Este método se ejecuta cuando el reconocimiento de voz no ha detectado ningún sonido. Utiliza los objetos `TextView` e `ImageView` para hacer saber al usuario de que ha habido un error en la escucha.

```
@Override
public void onError(int i) {
    if(i == SpeechRecognizer.ERROR_NO_MATCH){
        checkKO.setVisibility(View.VISIBLE);
        tv.setText("No se ha podido obtener ninguna orden");
    }
}
```

*Figura 15: onError()*

- **onEndOfSpeech()** : Este método se ejecuta cuando se detecta que se ha dejado de hablar. Cuando esto ocurre se llama al método `stopListening()` del `SpeechRecognizer`.

```
@Override
public void onEndOfSpeech() {
    speechRecognizer.stopListening();
}
```

*Figura 16: onEndOfSpeech()*

- **onResults()** : Este método se ejecuta cuando se obtienen los resultados del reconocimiento de voz. Tiene como parámetro un objeto de tipo `Bundle` [18], [19] cuya función es almacenar datos y pasarlos entre componentes de la aplicación. A este objeto `Bundle` se le aplica el método `getStringArrayList()` junto con la constante definida `RESULTS_RECOGNITION`, para acceder a los resultados del reconocimiento de voz.

  Los resultados se recogen en un `ArrayList` de `Strings` de manera que al aplicarle el método `get()` se almacena en una variable de tipo `String` la orden del usuario.

Una vez recogida la orden, se muestra en la interfaz para que el usuario la visualice utilizando el `TextView` y posteriormente se pasa como parámetro al método `initCliente()` de la clase `EnvioSocket`, donde se realiza el envío de la orden al servidor de ROS2.

```
@Override
public void onResults(Bundle resultado) {

    ArrayList<String> data = resultado.getStringArrayList(SpeechRecognizer.RESULTS_RECOGNITION);
    String envio = data.get(0);
    tv.setText(data.get(0));

    cliente.initCliente(envio, realimentacion);

}
```

*Figura 17: onResults()*

### 2.9.5. Objetos de escucha de eventos de entrada – View.OnClickListener

En Android existen diferentes formas para interceptar eventos desde una interacción con el usuario en una aplicación [20]. El enfoque consiste en capturar los eventos desde el objeto de vista específico con el que interactúa el usuario; es decir, si el usuario pulsa sobre algún componente de la interfaz de usuario de la aplicación, este evento se recoge en una acción determinada.

Un objeto de escucha de eventos es una interfaz de la clase `View` que contiene un solo método de devolución de llamada. El framework de Android llamará a estos métodos cuando la vista con la que se haya registrado el objeto de escucha se active por la interacción del usuario con el elemento de la interfaz de usuario.

El método `setOnClickListener` [21] [22] aplicado a un componente de la vista registra una llamada de retorno (listener) que se invocará cuando se haga clic en el contexto del componente de la vista de la aplicación como puede ser un `Button, ImageView, TextView` ...

En la Figura 18 se muestra un ejemplo de cómo se ha implementado la definición de un objeto de escucha de evento de entrada con la interfaz `View.OnClickListener`, que define el método `onClick()` donde se ejecuta la lógica de la aplicación para cuando se hace clic en el elemento.

```
//Botón configuración
btnConfig.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
```

*Figura 18: Definición de un objeto de escucha de evento de entrada*

También se puede hacer uso de una expresión lambda como en el siguiente ejemplo que implementa el método `onClick()` de una forma más concisa.

```
microfono.setOnClickListener(view -> {
        microfono.playAnimation();
        checkOK.setVisibility(View.INVISIBLE);
        checkKO.setVisibility(View.INVISIBLE);
        //Se empieza a escuchar, proporcionamos un Intent al recognizer
        speechRecognizer.startListening(speechRecognizerIntent);
});
```

*Figura 19: Objeto de escucha con expresión lambda*

En este caso concreto, cuando el usuario pulsa el componente de la vista `LottieAnimationView` 'micrófono', se procede a iniciar el reconocimiento de voz, así como la animación de la imagen con el método `playAnimation()`. También se esconden de la vista cualquier imagen de realimentación que se esté mostrando.

En la aplicación también se emplean los objetos de escucha en los dos componentes de tipo `Button` de la vista, el de la configuración se explica con más detalle en el siguiente apartado.

Respecto al botón de diccionario de órdenes, su método `onClick()` configura un cuadro de diálogo utilizando el objeto `AlertDialog.Builder` [23].

El fin de este cuadro de diálogo es mostrar al usuario las posibles órdenes que puede decir para que el robot se mueva.

En esta configuración del cuadro de diálogo se establece el diseño de su interfaz, así como el diccionario y se aplica un botón usando el método `setPositiveButton()` que añade un submétodo `onClick()` para cerrar el cuadro de diálogo cuando el usuario lo pulse. En la Figura 20 y Figura 21 se muestran algunas partes del código.

```
btnDic.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

        AlertDialog.Builder builder = new AlertDialog.Builder( context: MainActivity.this);
```

*Figura 20: Declaración del cuadro de diálogo*

```
builder.setMessage("\n- AVANZA X METROS (De 1 a 10 metros)" +
        "\n\n- SAL DE CASA" +
        "\n\n- VE A CASA\n\n- GIRA A LA IZQUIERDA\n\n" +
        "- GIRA A LA DERECHA\n\n- DA LA VUELTA\n\n- PARA")
        .setPositiveButton( text: "Salir", new DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialogInterface, int i) {
        dialogInterface.dismiss();
    }
}).show();
```

*Figura 21: Descripción del diccionario y uso del método setPositiveButton()*

### 2.9.6. Botón de configuración de la realimentación – SharedPreferences

Sobre este componente de la vista se emplea también un objeto de escucha para mostrar un cuadro de diálogo cuando el usuario pulse el botón. En este caso, el cuadro de diálogo tiene como objetivo que el usuario decida a través de un botón `Switch` si quiere ver en la aplicación la realimentación del robot o no.

Para mostrar el botón `Switch` se establece una referencia a la vista del archivo '`config_realimentación.xml`' y se almacena en un objeto de tipo `SwitchCompat` la referencia que se hace al componente `Switch`. `SwitchCompat` es una variación de la clase `Switch` [24] [25] de Android y es la que el propio Android aconseja usar.

De esta forma, al almacenar la referencia al componente `Switch` en este objeto, se pueden hacer operaciones sobre él como guardar y comprobar su estado.

Para que la aplicación sepa en todo momento qué valor tiene el `Switch` independientemente de si el usuario ha cerrado la aplicación anteriormente se utiliza `SharedPreferences`.

Un objeto `SharedPreferences` [26] se emplea en colecciones relativamente pequeñas de pares clave-valor que se quieren guardar o recuperar de manera persistente. Apunta a un archivo que contiene los pares clave-valor y proporciona métodos para leerlos y escribirlos.

En la siguiente figura se puede observar la instanciación de la vista del archivo '`config_realimentación.xml`' y la instanciación del botón `Switch`. Posteriormente se almacena en una variable el valor del par clave-valor del archivo '`ArchivoConfig`', que será el último que se estableció por el usuario (si no se encuentra ningún valor, se establece falso como valor predeterminado). Para ello se utiliza la instancia de `SharedPreferences` definida en un inicio. Cuando se obtiene el valor, se establece en el `SwitchCompat` mediante el método `setChecked()`. De esta forma, cada vez

que el usuario acceda al cuadro de diálogo, el botón `Switch` se mostrará con el último valor válido.

```
View vista = getLayoutInflater().inflate(R.layout.config_realimentacion, root: null);
SwitchCompat opcionSw = vista.findViewById(R.id.switchConfig);

//Para almacenar la información de la configuración de realimentación
boolean estadoSW = sp.getBoolean( s: "ArchivoConfig",  b: false); //Valor predeterminado: false
opcionSw.setChecked(estadoSW);

builder.setView(vista);
```

*Figura 22: Configuración del Switch*

A continuación, se muestra en la Figura 23 la definición del objeto `SharedPreferences`, donde se establece el archivo de configuración donde se va a guardar el par clave-valor del `Switch`. Este valor se recoge en una variable global '`realimentación`' que es la que se utiliza para determinar si se muestra o no la realimentación del robot en la aplicación.

```
SharedPreferences sp = getApplicationContext().getSharedPreferences( s: "ArchivoConfig", Context.MODE_PRIVATE);
realimentacion = sp.getBoolean( s: "ArchivoConfig",  b: false);
```

*Figura 23: Definición del objeto SharedPreferences*

Cabe destacar que, en el cuadro de diálogo del botón de configuración, se establece la lógica necesaria para que cuando el usuario pulse el botón 'Aceptar' del cuadro, el valor del `Switch` se compruebe y posteriormente se almacene por medio del uso del objeto `Editor` de `SharedPreferences`. Esto se hace a través del uso de los métodos `isChecked()` de `SharedPreferences` y de `edit()`, `putBoolean()` y `apply()` del `Editor` de `SharedPreferences`. Además, se sobrescribe la variable global '`realimentación`' para determinar si se muestra o no la realimentación del robot en la aplicación.

En la Figura 24 se muestra una parte de esta lógica implementada.

```
builder.setCustomTitle(titulo).
        setPositiveButton( text: "Aceptar", new DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialogInterface, int i) {
                boolean opcion = opcionSw.isChecked();
                if (opcion) {
                    SharedPreferences.Editor ed = sp.edit();
                    ed.putBoolean( s: "ArchivoConfig", opcionSw.isChecked()); //opcion
                    ed.apply();
                    realimentacion = true;
                    Toast.makeText( context: MainActivity.this,  text: "Realimentación activada"
                } else {
```

*Figura 24: Lógica de almacenamiento del valor del botón Switch*

### 2.9.7. Botón de configuración de la IP – SharedPreferences

El uso de `SharedPreferences` se emplea también con el objetivo del almacenamiento de la IP del ordenador que ejecuta el servidor de ROS2.

Cuando el usuario pulsa el botón de configuración de la IP se muestra una ventana con un cuadro de texto para escribir la dirección IP. Al pulsar aceptar, se almacena el valor de la misma manera que se ha explicado anteriormente, pero utilizando un dato de tipo `String`.

Cuando se inicia la aplicación se toma el último valor almacenado y en el caso de no existir aun un valor, se obtiene un valor por defecto.

```
String ipSP = spIP.getString( s: "ArchivoIP",  s1: "0.0.0.0");

builder.setView(vista);
edT.setText(ipSP);

builder.setCustomTitle(titulo)
.setPositiveButton( text: "Aceptar", new DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialogInterface, int i) {
        IPHOST = edT.getText().toString();
        SharedPreferences.Editor editorSP = spIP.edit();
        editorSP.putString( s: "ArchivoIP", IPHOST);
        editorSP.apply();
        iptv.setText("IP del servidor de ROS2 " + IPHOST);
        Toast.makeText( context: MainActivity.this,  text: "IP configurada", Toast.LENGTH_SHORT).show();
    }
}).show();
```

*Figura 25: Lógica de SharedPreferences para la IP del servidor*

## 2.10. EnvioSocket.java

La clase `EnvioSocket` es la encargada de transmitir los datos (las órdenes) que el sistema de reconocimiento de voz de la aplicación obtiene del usuario. También recibe una respuesta del servidor cuando se ha ejecutado una orden o ha ocurrido alguna incidencia.

Se instancia en la clase `MainActivity` y se le pasa como parámetros al constructor un manejador de tipo `Handler`, y dos componentes de la vista `ImageView`.

```
Handler handler = new Handler(Looper.getMainLooper());
EnvioSocket cliente = new EnvioSocket(handler, checkOK, checkKO);
```

*Figura 26: Instanciación de la clase EnvioSocket en MainActivity*

Como se ha explicado en apartados anteriores; cuando se obtienen los resultados del reconocimiento de voz en la actividad principal, dentro del método `onResults()` del `RecognitionListener` se llama al método `initCliente()` de la clase `EnvioSocket` para iniciar el envío de la orden. A este método `initCliente()` se le pasa como parámetros la orden en formato `String` como cadena de caracteres y la variable booleana '`realimentacion`'.

### 2.10.1. initCliente() - Executor

En Android está prohibida la ejecución de operaciones de red en el hilo principal de las aplicaciones por razones de seguridad. Cuando se inicia una aplicación, el sistema crea un hilo de ejecución denominado ‘principal’. Este hilo es el más importante porque está a cargo de distribuir eventos a los widgets correspondientes de la interfaz de usuario, incluidos los eventos de dibujo.

Cuando la aplicación realiza un trabajo intensivo como en este caso son las operaciones de envío y recibimiento de datos a través de la red, el hilo principal de la aplicación puede generar un rendimiento deficiente llegando a bloquear toda la interfaz de usuario. Cuando se bloquea este hilo principal no se pueden distribuir los eventos dando al usuario la sensación de que la aplicación no responde [27].

Por esta razón, el envío de las órdenes al servidor de ROS2 desde la aplicación se realiza en un hilo secundario que se ejecuta en la clase `EnvioSocket`, concretamente en el método `initCliente()` mediante el uso de un objeto `Executor` [28] .

En la Figura 27 se muestra la creación de este objeto. Mediante el uso de su método `execute()` se ejecuta la tarea definida del hilo secundario.

```
1 usage
public void initCliente(String datos, Boolean realimentacion) {

    Executor executor = Executors.newSingleThreadExecutor();
    executor.execute(() -> {
```

*Figura 27: Objeto Executor*

La tarea del hilo secundario se basa en la creación de un objeto `Socket` [29] al que se le pasa como argumentos el puerto y la dirección IP donde se mandan las órdenes que el usuario dicta por voz.

Además, en este hilo secundario se recibe también del servidor de ROS2 un mensaje en función de si todo ha ido bien o ha ocurrido algo mal.

Una vez creado el socket, se crean dos objetos de tipo `DataInputStream` [30] y `DataOutputStream` [31] definidos anteriormente. Este tipo de objetos permiten leer y escribir respectivamente en la aplicación tipos de datos primitivos de Java.

En la Figura 28 se muestra el código del hilo secundario con la creación del socket, así como el envío y recibimiento de los datos por la red a través del uso de los métodos `writeUTF()` y `readUTF8()` respectivamente.

```
sc = new Socket(HOST, PUERTO);

entrada = new DataInputStream(sc.getInputStream());
salida = new DataOutputStream(sc.getOutputStream());

salida.writeUTF(datos);      //Enviamos mensaje
aux = readUTF8(entrada);     //Recibimos mensaje
```

*Figura 28: Gestión de los datos en hilo secundario*

Los datos se envían y reciben como cadena de caracteres [32] y para ello es necesario tratarlos. Como se ha explicado en apartados anteriores, la orden de movimiento que el sistema de reconocimiento de voz recoge se trata y se recoge en forma de `String` y esta se pasa ya por lo tanto como cadena de caracteres a la clase `EnvioSocket` para su posterior envío mediante el método `writeUTF()` que proporciona el objeto `DataOutputStream` `'salida'`.

Ahora bien; existe un problema. Estos métodos de envío y recibimiento de datos en Java no tienen una equivalencia en Python, que es el lenguaje de programación en el que está escrito el servidor de ROS2, y por lo tanto esto da lugar a una entrega y recibimiento de los datos con un formato diferente.

Para solventarlo se ha seguido una solución propuesta en un foro de Internet y en donde se desarrolla con mayor profundidad el problema [33].

La solución consiste en abandonar los métodos propios que ofrece Java que son `readUTF()` y `writeUTF()` y reemplazarlos por una versión propia que es la que se ha seguido, con el fin de que los datos en ambos lados tengan el formato correcto para poder manejarlos como se desea.

A continuación, se muestra en la Figura 29 el método `readUTF8()` que sustituye al original, y que se encarga de recibir y tratar el dato del servidor de manera que tenga el formato adecuado para poder utilizarse.

```
1 usage
public String readUTF8(DataInput in) throws IOException {
    int length = in.readInt();
    byte [] encoded = new byte[length];
    in.readFully(encoded);
    return new String(encoded, StandardCharsets.UTF_8);
}
```

*Figura 29: Método readUTF8()*

En el caso de este proyecto no ha sido necesario utilizar una versión propia para el método `writeUTF()` ya que se podía acceder correctamente a los datos en el lado del servidor. Sin embargo, sí ha sido necesario también establecer nuevos métodos de envío por la parte del servidor que se explicarán en el apartado que corresponde.

En este hilo secundario, además de enviar y recibir los datos, se muestra la realimentación en la interfaz gráfica de la aplicación. El problema reside en que sólo el hilo principal puede modificar componentes de la vista de la aplicación y, por lo tanto, al estar en el hilo secundario, es necesario emplear algún mecanismo para lograr este objetivo. Para ello se ha utilizado un manejador de tipo `Handler` [34] que se pasa al constructor de la clase `EnvioSocket` desde la clase principal.

A través del manejador `Handler` [35] se da soporte a la gestión del trabajo entre dos hilos. De esta manera, se transfiere trabajo al hilo principal desde el hilo secundario.

Se utiliza el método `post()` del `Handler` para enviar una tarea concreta al hilo principal definida en un objeto de tipo `Runnable`. El hilo principal de la actividad de la aplicación ejecuta esta tarea de forma que se consigue mostrar al usuario la realimentación en la interfaz gráfica.

El objeto `Runnable` ejecuta el método `run()` donde se define la tarea a realizar por el hilo principal. En la siguiente figura se muestra un ejemplo del código en el caso de que la realimentación esté activada.

Se pone como visible uno de los dos `ImageView` de la vista en función de si la orden se ejecutó correctamente o si hubo algún problema (Figura 30).

```
if(realimentacion){
    if (aux.contains("OK")) {
        manejadorPrincipal.post(new Runnable() {
            @Override
            public void run() {
                checkKOPrincipal.setVisibility(View.INVISIBLE);
                checkOKPrincipal.setVisibility(View.VISIBLE);
            }
        });
    } else {
```

*Figura 30: Uso del Handler para el hilo principal*

# 3. SERVIDOR DE ROS2

En esta sección se explica detalladamente ROS2 (*Robot Operating System*), su instalación y su implementación en el servidor que gestiona las órdenes que se reciben del usuario para producir el movimiento del robot.

## 3.1. ROS2

ROS2 [36] es un conjunto de bibliotecas que forman un marco de trabajo de software libre para construir, ejecutar y administrar aplicaciones para robots y sistemas robóticos, proporcionando un conjunto estándar de configuración y una variedad de herramientas para trabajar ya integradas que simplifican la tarea de crear un comportamiento robusto y complejo del sistema.

Concretamente, la distribución de ROS2 que se ha utilizado es Humble; disponible para Ubuntu 22.04, que es la versión de Linux con la que se ha trabajado. Todas estas herramientas y bibliotecas tienen como objetivo simplificar la tarea de crear un comportamiento complejo y potente a través de una variedad de plataformas robóticas.

La comunicación con el sistema robótico en este proyecto es uno de los aspectos más importantes y es por eso por lo que se ha elegido establecerla mediante el uso de ROS2.

## 3.2. Instalación de ROS2 Humble

La instalación de ROS2 puede hacerse fácilmente desde un terminal de Ubuntu siguiendo los pasos de la documentación oficial sobre el iRobot Create 3 [37]. Aunque también se pueden seguir los pasos de la documentación oficial de ROS2 [38], es importante destacar que la instalación desde la página oficial del robot es mejor ya que proporciona las instrucciones de instalación del paquete '`ros-humble-irobot-create-msgs`' para poder hacer uso de los mensajes específicos del robot. Además, también recomienda la instalación de otros paquetes que pueden ser de utilidad posteriormente.

Tras la instalación de ROS2, es recomendable configurar el middleware por defecto de ROS2 (RMW) que queramos. El RMW que establezcamos aquí deberá ser compatible con el RMW del robot. Esto se puede comprobar en la página '*Application Configuration*' del servidor web del robot cuando este se configura; esto se explica más en detalle en el apartado dedicado a la configuración del robot. La documentación oficial de la página recomienda emplear el middleware 'Fastrtps_cpp' ya que es el que viene por defecto en la versión Humble de ROS2.

Finalmente, si tanto ordenador como robot se encuentran conectados a la misma red se puede hacer una prueba de comunicación entre ambos para comprobar que se reconocen entre los dos.

A través de la ejecución del comando '`ros2 topic list`' en la terminal de Ubuntu se mostrarán los diferentes *topics* que el robot tiene disponibles y a los que el servidor va a poder acceder para su uso.

```
$ ros2 topic list -t
/battery_state [sensor_msgs/msg/BatteryState]
/cmd_audio [irobot_create_msgs/msg/AudioNoteVector]
/cmd_lightring [irobot_create_msgs/msg/LightringLeds]
/cmd_vel [geometry_msgs/msg/Twist]
/dock [irobot_create_msgs/msg/Dock]
/hazard_detection [irobot_create_msgs/msg/HazardDetectionVector]
/imu [sensor_msgs/msg/Imu]
/interface_buttons [irobot_create_msgs/msg/InterfaceButtons]
/ir_intensity [irobot_create_msgs/msg/IrIntensityVector]
/ir_opcode [irobot_create_msgs/msg/IrOpcode]
/kidnap_status [irobot_create_msgs/msg/KidnapStatus]
/mouse [irobot_create_msgs/msg/Mouse]
/odom [nav_msgs/msg/Odometry]
/parameter_events [rcl_interfaces/msg/ParameterEvent]
/rosout [rcl_interfaces/msg/Log]
/slip_status [irobot_create_msgs/msg/SlipStatus]
/stop_status [irobot_create_msgs/msg/StopStatus]
/tf [tf2_msgs/msg/TFMessage]
/tf_static [tf2_msgs/msg/TFMessage]
/wheel_status [irobot_create_msgs/msg/WheelStatus]
/wheel_ticks [irobot_create_msgs/msg/WheelTicks]
/wheel_vels [irobot_create_msgs/msg/WheelVels]
```

*Figura 31: ros2 topic list*

## 3.3. Configuración del entorno de trabajo de ROS2

Una vez instalado ROS2, el siguiente paso es definir un entorno de trabajo donde se almacenarán todos los archivos y carpetas necesarias para el correcto funcionamiento del servidor.

En el caso de este proyecto, el directorio de trabajo se ha nombrado '`ros2_ws`' y dentro de este se tiene la carpeta '`src`' donde se almacena el paquete ROS2 que contiene todos los archivos del servidor.

El entorno de desarrollo de ROS 2 necesita ser configurado correctamente antes de su uso. Esto se puede hacer cargando los archivos de configuración en cada nueva ventana de la terminal que se abra con el comando '`source /opt/ros/humble/setup.bash`' [39] o '`source install/setup.bash`' si se tiene ya un paquete ROS2 que incluye la carpeta '`install`' al crearse. De esta manera se tiene acceso a los comandos de ROS2.

El archivo `setup.bash` establece variables de entorno y configuraciones que son necesarias para el uso de los paquetes de ROS2.

## 3.4. Creación del paquete ROS2 del servidor

Un paquete en ROS2 es la unidad de organización del código. La creación de paquetes en ROS2 utiliza 'ament' como sistema de compilación y 'colcon' como herramienta de compilación [40] y en este proyecto se lleva a cabo a través de Python, aunque también está soportado oficialmente CMake.

La creación del paquete del servidor de ROS2 se lleva a cabo en el espacio de trabajo definido anteriormente dentro de la carpeta src a través del comando '`ros2 pkg create --build-type ament_python nombre_paquete`'.

Cuando se crea el paquete, se crean por defecto una serie de archivos que son imprescindibles para su funcionamiento. Son los siguientes:

- **package.xml** : contiene metadatos sobre el paquete. También se definen las dependencias con otros paquetes como son el paquete

‘irobot_create_msgs’, ‘geometry_msgs’ o ’rclpy’ necesarios para la implementación del servidor.

```
<exec_depend>rclpy</exec_depend>
<exec_depend>std_msgs</exec_depend>

<depend>irobot_create_msgs</depend>
<depend>geometry_msgs</depend>
```

*Figura 32: Dependencias de package.xml*

- **resource/servidorROS2** : archivo de marcadores para el paquete.
- **setup.cfg** : necesario al tener ejecutables, para que el comando ‘ros2 run’ pueda encontrarlos y ejecutarlos.
- **setup.py** : contiene instrucciones sobre cómo instalar el paquete. Cuando se instala, se define el punto de entrada ‘irc3’ asociado a la función ‘main’ del archivo ‘irobot.py’ del paquete ‘servidorROS’

```
entry_points={
    'console_scripts': [
        'irc3 = servidorROS.irobot:main',
    ],
},
```

*Figura 33: setup.py*

- **servidorROS2** : es un directorio con el mismo nombre que el paquete usado por las herramientas de ROS2 para encontrar el paquete. Contiene el archivo `__init__.py` y el archivo irobot.py que contiene el código del servidor.

## 3.5. Compilación y ejecución

Para la compilación y construcción del paquete del servidor se ejecuta en la terminal el comando ‘`colcon build`’.

Seguidamente es necesario la ejecución en un primer momento del comando ‘`source install/setup.bash`’ para configurar correctamente el entorno de trabajo donde se utiliza el paquete y configurar además las variables de entorno que son necesarias para trabajar en un entorno de trabajo

de ROS2. Para las siguientes compilaciones del código del servidor no es necesario este comando si uno se encuentra en la misma terminal.

Por último, sólo queda ejecutar el servidor a través del comando ‘`ros2 run servidorROS irc3`’. El primer parámetro especifica el nombre del paquete y el segundo parámetro el punto de entrada definido en el archivo ‘`setup.py`’ para ejecutar la función ‘`main`’.

### 3.5.1. Problema de compilación

Es posible que al compilar el paquete de ROS2 salte un error del tipo ‘`SetuptoolsDeprecationWarning: setup.py install is deprecated. Use build and pip and other standards-based tools`’.

Esto es debido a que el paquete setuptools utilizado por `ros2 pyhton packages` está obsoleto. Se puede comprobar la versión de Python y la de setuptools.

Para solucionar este problema hay que degradar la versión de setuptools por debajo de la 58.2.0 a través del comando `pip install setuptools==versión` ya que si está por encima no funciona. En el caso de este proyecto hubo que instalar primero el administrador de paquetes de Python PIP [41] .

## 3.6. robot.py

Este archivo define el código del servidor del proyecto y tiene como objetivo recibir las órdenes que el usuario envía desde la aplicación móvil para posteriormente ejecutar los comandos de movimiento del robot en función de la orden recibida, así como enviar una realimentación del estado de la orden a la aplicación.

Se definen dos clases. La clase `ServidorROS` es la clase principal que establece la conexión con la aplicación y se encarga de recibir los datos, tratarlos y, dependiendo de la orden de movimiento que se haya recibido,

llamar a la función que corresponda de la clase secundaria `OrdenesClient` para ejecutar el movimiento del robot.

Entre las bibliotecas utilizadas, las más importantes son las siguientes:

- **socket**: proporciona funciones y clases imprescindibles para la comunicación mediante el uso de sockets entre el servidor y la aplicación.
- **rclpy**: es la biblioteca de ROS2 para la programación en Python. Imprescindible para la creación de nodos y la publicación y suscripción a *topics* y llamadas a servicios del robot. Establece el contexto de ROS2 para poder realizar operaciones.
- **irobot_create_msgs**: permite crear, enviar y recibir mensajes y acciones específicas del robot.

### 3.6.1. Función main()

Se encarga de inicializar el contexto de la biblioteca 'rclpy' y de llamar al constructor y ejecutar la función principal de la clase `ServidorROS` para iniciar la ejecución del servidor.

```
def main(args=None):

    rclpy.init(args = args)
    servidorROS = ServidorROS()

    try:
        servidorROS.run()
    except KeyboardInterrupt:
        print('\nSe ha pulsado salir')
    finally:
        print('Saliendo')

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

*Figura 34: Función main del servidor*

### 3.6.2. Clase ServidorROS

Cuando se crea la instancia de esta clase automáticamente se llama al constructor '`__init__`' para que el socket se cree, se asocie a las interfaces de red y al puerto 8080 y se habilite para aceptar conexiones entrantes y recibir así información.

```
def __init__(self):

    self.sc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.sc.bind(("", PUERTO))
    self.sc.listen()

    print("Servidor iniciado. Esperando órdenes ...")
```

*Figura 35: Constructor de la clase ServidorROS*

La función `write_utf8()` es la encargada de tratar y enviar el dato de realimentación a la aplicación una vez se ha ejecutado una orden. Esta función se define para solventar el problema mencionado en apartados anteriores de que no existe una equivalencia entre Python y Java para el envío y recibimiento de datos a través de la red.

```
def write_utf8(self, s: str, sock: socket.socket):
    encoded = s.encode(encoding='utf-8')
    sock.sendall(struct.pack('>i', len(encoded)))
    sock.sendall(encoded)
```

*Figura 36: Función write_utf8 para el envío de la realimentación*

No ha sido necesario definir otra función distinta a la que se utiliza para el recibimiento de los datos en el servidor, simplemente se comprueba que la orden esté dentro del mensaje recibido.

La función `run()` se encarga de esperar por una conexión entrante y una vez conseguida recibir y tratar los datos procedentes de la aplicación, que se almacenan en una variable como cadena de caracteres.

```
while True:
    cliente, addr = self.sc.accept()
    try:
        data = cliente.recv(1024)
        if not data:
            break;

        mensajeRecibido = data.decode("utf-8").upper()
```

*Figura 37: Recibimiento y almacenamiento de los datos*

Una vez el dato recibido se almacena en la variable '`mensajeRecibido`', se compara con cada una de las constantes globales definidas para determinar si el dato recibido es alguna de las órdenes que se pueden ejecutar para proporcionar el movimiento del robot.

Si el dato recibido no se reconoce como orden, entonces el servidor se encarga de que el robot muestre en rojo su anillo LED y de enviar a la aplicación una realimentación (un mensaje) para indicar al usuario que ha habido un error.

```
else:
    print("No entiendo esa orden")
    mensajeEnviado = "KO"
    self.write_utf8(mensajeEnviado, cliente)
    orden_client.led_callback(r=255, g=0, b=0)
    orden_client.led_callback(r=255, g=255, b=255)
```

*Figura 38: Orden no definida*

En los demás casos en los que el dato recibido se corresponde con alguna orden definida se llama a la función de la clase `OrdenesClient` respectivamente para la ejecución de la orden. Seguidamente se envía una realimentación a la aplicación para mostrar al usuario que la orden se ha ejecutado satisfactoriamente.

En la Figura 39 se muestra un ejemplo de lo que se hace si el dato recibido se corresponde con alguna de las órdenes definidas.

```
elif GIRAIZQ in mensajeRecibido:
    if FLAG_CASA == False:
        print("Girando a la izquierda")

        future = orden_client.girarIZQ()

        mensajeEnviado = "OK"
        self.write_utf8(mensajeEnviado, cliente)

    else:
        print("No puedo, estoy en casa")
        mensajeEnviado = "KO"
        self.write_utf8(mensajeEnviado, cliente)
```

*Figura 39: Orden 'Gira a la izquierda'*

### 3.6.3. Clase OrdenesClient

Esta clase define las funciones que ejecutan el movimiento del robot. Hereda de la clase `Node`, que es una clase proporcionada por la biblioteca rclpy y que permite que la clase `OrdenesClient` funcione como un nodo de ROS2 para poder comunicarse con el robot a través de la publicación y suscripción a los *topics* que ofrece, así como realizar peticiones de acciones al robot.

A través del constructor, se crean las instancias de la clase `ActionClient` [42], una para cada acción que queremos que realice el robot; salir de la estación de carga, volver a la estación de carga, girar, y avanzar una determinada distancia. En la creación se especifica la acción que se quiere ejecutar, así como el identificador de la acción en ROS2 para el robot.

```
def __init__(self):
    super().__init__('ordenes_client')
    self._dock_client = ActionClient(self, Dock, 'dock')
    self._undock_client = ActionClient(self, Undock, 'undock')
    self._girar_client = ActionClient(self, RotateAngle, '/rotate_angle')
    self._mover_client = ActionClient(self, DriveDistance, '/drive_distance')
```

*Figura 40: Creación de las instancias de acción con ActionClient*

En el constructor también se crea la instancia de un publicador [43] para enviar publicaciones al robot con el objetivo de que este cambie las luces del anillo LED a la hora de que se ejecute o no una orden.

```
self.lights_publisher = self.create_publisher(LightringLeds, '/cmd_lightring', 10)
self.lightring = LightringLeds()    #Definición de los mensajes del anillo LED
self.lightring.override_system = True #Esto nos permite cambiar el color de las luces
```

*Figura 41: Publicador para el anillo LED del robot*

Al crear el publicador se especifica el tipo de mensajes (`LightringLeds`) que se quieren publicar en el *topic*, así como el nombre del *topic* donde se van a publicar los mensajes `/cmd_lightring`.

En el nodo de ROS2, es decir, la clase `OrdenesClient` se definen las funciones que van a ejecutar cada uno de los diferentes movimientos del robot.

En cada función se establece primero una instancia de la meta (la acción que se quiere realizar) como mensaje.

Posteriormente se comprueba si el servidor de la acción está disponible, es decir, si el robot no está en la misma red que el servidor o ambos no se reconocen, entonces no se encontrará el servidor de acción.

Por último, si se ha encontrado el servidor de acción se realiza una publicación al anillo LED del robot para que cambie de color (llamando a la función que ejecuta la publicación) y se envía el mensaje de meta al robot para que se ejecute el movimiento.

En la Figura 42 se puede observar un ejemplo de una de las funciones de movimiento.

```python
#Ve a casa
def dock(self):
    goal_msg = Dock.Goal()

    if not self._dock_client.wait_for_server(timeout_sec=5.0):
        raise Exception("No se ha encontrado el servidor de acción")

    self.led_callback(r=0, g=255, b=0)
    instFuture = self._dock_client.send_goal_async(goal_msg)
    self.led_callback(r=255, g=255, b=255)

    global FLAG_CASA
    FLAG_CASA = True

    return instFuture
```

*Figura 42: Función para hacer que el robot vuelva a la estación de carga*

### 3.6.4. Paquete Infrarrojos

El paquete `Infrarrojos` se encarga de ejecutar un nodo de ROS2 para la suscripción a un *topic* del robot.

La suscripción se realiza al *topic* `/ir_intensity` para obtener información sobre los siete sensores infrarrojos que proporciona el robot y tiene el objetivo de mostrar la detección de obstáculos.

Este paquete se ejecuta en otra terminal en conjunción con el servidor para mostrar por pantalla las lectura de los sensores infrarrojos que el robot tiene en la parte delantera. De manera que, si alguno de los sensores alcanza cierto valor, se notifique por la terminal que el robot ha detectado un obstáculo.

El procedimiento es similar al nodo del servidor. Se crea un suscriptor al *topic* del robot que proporciona la información de los sensores infrarrojos. En la creación se define una función de callback que se ejecuta cada vez que el nodo se suscribe al *topic*. En esta función de callback se ejecutan las instrucciones necesarias para mostrar por pantalla los valores de los sensores.

Para que se esté continuamente recibiendo información se hace uso de la función '`spin()`' [44] de la biblioteca rclpy, que mantiene al nodo en ejecución de manera bloqueante hasta que se interrumpa el contexto asociado.

En la Figura 43 se muestra la instanciación del nodo y su puesta en ejecución en la función `main`.

```
def main(args = None):
    rclpy.init(args = args)
    IR_subscriber = IRSubscriber()
    print('Se llama a las rellamadas.')

    try:
        rclpy.spin(IR_subscriber)
    except KeyboardInterrupt:
        print('\n Se ha pulsado salir')
    finally:
        print('HECHO')
        IR_subscriber.destroy_node()
```

*Figura 43: Función main donde se ejecuta el nodo de ROS2*

En la Figura 44 se muestra el nodo que constituye la función de callback y su llamada la función `printIR()` para tratar la lectura de los sensores.

```
class IRSubscriber(Node):
    def __init__(self):
        super().__init__('IR_subscriber')
        print('Creando suscripción al tipo IrIntensityVector sobre el tc
        #Creamos el suscriptor
        self.subscription = self.create_subscription(IrIntensityVector,
            '/ir_intensity', self.listener_callback,
            qos_profile_sensor_data)

    def printIR(self, msg):

        print('Mostrando lecturas del sensor IR: ')
        for reading in msg.readings:
            val = reading.value
            if(val >= 250):
                print('Obstáculo encontrado')
                print('IR sensor; ' + str(val))

            print('IR sensor; ' + str(val))
        time.sleep(0.5)

    def listener_callback(self, msg:IrIntensityVector):
        print('Escuchando las lecturas del sensor IR ...')
        self.printIR(msg)
```

*Figura 44: Nodo de ROS2*

# 4. REFERENCIAS


[1] «Add build dependencies | Android Studio | Android Developers». https://developer.android.com/build/dependencies (accedido 8 de junio de 2023).

[2] «Cómo configurar tu compilación | Desarrolladores de Android | Android Developers». https://developer.android.com/studio/build?hl=es-419 (accedido 8 de junio de 2023).

[3] «Featured animations from our community». https://lottiefiles.com/featured?utm_medium=web&utm_source=navigation-featured (accedido 9 de junio de 2023).

[4] «Descripción general del manifiesto de una app | Desarrolladores de Android | Android Developers». https://developer.android.com/guide/topics/manifest/manifest-intro?hl=es-419 (accedido 9 de junio de 2023).

[5] «Diseños | Desarrolladores de Android | Android Developers». https://developer.android.com/guide/topics/ui/declaring-layout?hl=es-419 (accedido 10 de junio de 2023).

[6] «Switch En Android - Develou». https://www.develou.com/switch-en-android/ (accedido 10 de junio de 2023).

[7] «SwitchCompat | Android Developers». https://developer.android.com/reference/androidx/appcompat/widget/SwitchCompat (accedido 10 de junio de 2023).

[8] «Introducción a las actividades | Desarrolladores de Android | Android Developers». https://developer.android.com/guide/components/activities/intro-activities?hl=es-419 (accedido 10 de junio de 2023).

[9] «Qué es una Activity, el componente más importante de una App Android». https://devexperto.com/activity-android/ (accedido 10 de junio de 2023).

[10] «Activity | Android Developers». https://developer.android.com/reference/android/app/Activity#setContentView(android.view.View) (accedido 11 de junio de 2023).

[11] «Solicitud permisos de tiempo de ejecución | Android Developers». https://developer.android.com/training/permissions/requesting?hl=es-419 (accedido 12 de junio de 2023).

[12] «Servicios de voz de Google - Aplicaciones en Google Play». https://play.google.com/store/apps/details?id=com.google.android.tts&hl=es&gl=US (accedido 12 de junio de 2023).

[13] «SpeechRecognizer | Android Developers». https://developer.android.com/reference/android/speech/SpeechRecognizer (accedido 4 de junio de 2023).

[14] «Intents y filtros de intents | Desarrolladores de Android | Android Developers». https://developer.android.com/guide/components/intents-filters?hl=es-419 (accedido 12 de junio de 2023).

[15] «How to set the language in speech recognition on android? - Stack Overflow». https://stackoverflow.com/questions/10538791/how-to-set-the-language-in-speech-recognition-on-android (accedido 13 de junio de 2023).

[16] «Example usage for android.speech RecognizerIntent EXTRA_CALLING_PACKAGE». http://www.java2s.com/example/java-api/android/speech/recognizerintent/extra_calling_package-0.html (accedido 13 de junio de 2023).

[17] «RecognitionListener | Android Developers». https://developer.android.com/reference/android/speech/RecognitionListener (accedido 13 de junio de 2023).

[18] «¿Cuál es la diferencia entre intent y bundle en Android? - CompuHoy.com». https://www.compuhoy.com/cual-es-la-diferencia-entre-intent-y-bundle-en-android/ (accedido 13 de junio de 2023).

[19] «Bundle | Android Developers». https://developer.android.com/reference/android/os/Bundle (accedido 13 de junio de 2023).

[20] «Descripción general de eventos de entrada | Desarrolladores de Android | Android Developers». https://developer.android.com/guide/topics/ui/ui-events?hl=es-419 (accedido 13 de junio de 2023).

[21] «View | Android Developers». https://developer.android.com/reference/android/view/View#setOnClickListener(android.view.View.OnClickListener) (accedido 13 de junio de 2023).

[22] «Método setOnClickListener en Android». https://keepcoding.io/blog/metodo-setonclicklistener-en-android/ (accedido 13 de junio de 2023).

[23] «Cuadros de diálogo | Desarrolladores de Android | Android Developers». https://developer.android.com/guide/topics/ui/dialogs?hl=es-419 (accedido 14 de junio de 2023).

[24] «Switch En Android - Develou». https://www.develou.com/switch-en-android/ (accedido 14 de junio de 2023).

[25] «SwitchCompat | Android Developers». https://developer.android.com/reference/androidx/appcompat/widget/SwitchCompat (accedido 14 de junio de 2023).

[26] «Cómo guardar datos simples con SharedPreferences | Android Developers». https://developer.android.com/training/data-storage/shared-preferences?hl=es-419#java (accedido 14 de junio de 2023).

[27] «Descripción general de los procesos y subprocesos | Desarrolladores de Android | Android Developers». https://developer.android.com/guide/components/processes-and-threads?hl=es-419 (accedido 15 de junio de 2023).

[28] «Executor | Android Developers». https://developer.android.com/reference/java/util/concurrent/Executor (accedido 16 de junio de 2023).

[29] «Socket | Android Developers». https://developer.android.com/reference/java/net/Socket (accedido 16 de junio de 2023).

[30] «DataInputStream | Android Developers». https://developer.android.com/reference/java/io/DataInputStream (accedido 16 de junio de 2023).

[31] «DataOutputStream | Android Developers». https://developer.android.com/reference/java/io/DataOutputStream (accedido 16 de junio de 2023).

[32] «android - How to get String from DataInputStream in Java? - Stack Overflow». https://stackoverflow.com/questions/27827803/how-to-get-string-from-datainputstream-in-java (accedido 16 de junio de 2023).

[33] «Python – Create a python server to send data to Android app using socket – iTecNote». https://itecnote.com/tecnote/python-create-a-python-server-to-send-data-to-android-app-using-socket/ (accedido 16 de junio de 2023).

[34] «Handler | Android Developers». https://developer.android.com/reference/android/os/Handler (accedido 16 de junio de 2023).

[35] «Handlers – Aplicaciones Móviles Multimedia». https://umhandroid.momrach.es/handlers/ (accedido 16 de junio de 2023).

[36] «ROS 2 Documentation — ROS 2 Documentation: Humble documentation». https://docs.ros.org/en/humble/index.html (accedido 17 de junio de 2023).

[37] «Humble on Ubuntu 22 - Create® 3 Docs». https://iroboteducation.github.io/create3_docs/setup/ubuntu2204/#before-you-start (accedido 17 de junio de 2023).

[38] «Ubuntu (Debian) — ROS 2 Documentation: Humble documentation». https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debians.html (accedido 17 de junio de 2023).

[39] «Configuring environment — ROS 2 Documentation: Humble documentation». https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Configuring-ROS2-Environment.html (accedido 17 de junio de 2023).

[40] «Creating a package — ROS 2 Documentation: Humble documentation». https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Creating-Your-First-ROS2-Package.html (accedido 17 de junio de 2023).

[41] «SetuptoolsDeprecationWarning: setup.py install is deprecated. Use build and pip and other standards-based tools. - ROS Answers: Open Source Q&A Forum». https://answers.ros.org/question/396439/setuptoolsdeprecationwarning-setuppy-install-is-deprecated-use-build-and-pip-and-other-standards-based-tools/ (accedido 22 de junio de 2023).

[42] «Actions». https://design.ros2.org/articles/actions.html (accedido 17 de junio de 2023).

[43] «Writing a simple publisher and subscriber (Python) — ROS 2 Documentation: Foxy documentation». https://docs.ros.org/en/foxy/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Py-Publisher-And-Subscriber.html (accedido 18 de junio de 2023).

[44] «Initialization, Shutdown, and Spinning — rclpy 0.6.1 documentation». https://docs.ros2.org/foxy/api/rclpy/api/init_shutdown.html#rclpy.spin (accedido 18 de junio de 2023).

[45] «Overview - Create® 3 Docs». https://iroboteducation.github.io/create3_docs/releases/overview/ (accedido 18 de junio de 2023).