Wiki para las memorias de prácticas

La *wiki* colaborativa tiene que servir para aprender a trabajar en equipos heterogéneos y con gente que tenga diversas actitudes y aptitudes, lo cual es muy importante en ingeniería hoy en día. Por tanto:

- Es muy importante ser consciente en todo momento de que la nota completa de la *wiki* **influye en la nota de todos y cada uno de los alumnos**. Si algún alumno decide no colaborar, debería ser honesto y pasarse cuanto antes al modo de evaluación no-continua, permitiendo así que el resto se reparta de nuevo el trabajo para obtener la mejor calificación posible.
- El trabajo de hacer las memorias para la *wiki* no consiste en un simple reparto de prácticas, aislándose los grupos entre sí: los alumnos **pueden ayudar a otros a mejorar el trabajo**. Esto es importante porque la *wiki* es un recurso que todos tendréis disponible en el examen, así que debería estar lo mejor hecho posible desde el punto de vista de cada uno de vosotros.
- Aunque un alumno participe en subir a la *wiki* una práctica, no es esa práctica la única que debe realizar: todas las prácticas deben ser resueltas por todos los alumnos, aunque el trabajo de elaborar memorias y subirlas a la *wiki* sea colaborativo. **No confundáis el trabajo de elaborar memorias con el de resolver las prácticas y ejercicios.**

En cuanto a los aspectos técnicos:

- En la página inicial debe haber un índice con una entrada por cada práctica o relación de problemas, y cada entrada debe ir a una página propia para ella. **No uséis una estructura distinta de ésa.**
- En la página de cada práctica o relación de problemas debéis indicar quiénes se han encargado de subirla a la *wiki*.
- Las memorias no pueden contener únicamente el código de los programas realizados, o sólo los resultados finales: deben incluir los pasos, explicaciones y comentarios oportunos para resolver cada práctica o relación de problemas.
- **No se pueden subir a la wiki partes relevantes de apuntes de clase o teoría de libros, etc.; únicamente las explicaciones necesarias de cómo se ha resuelto cada práctica o relación de problemas, trozos de código si hace falta, figuras, ecuaciones usadas, etc.**

Práctica sobre diseño e implementación de un sistema de tiempo real para dispositivos móviles

Tabla de contenidos

---

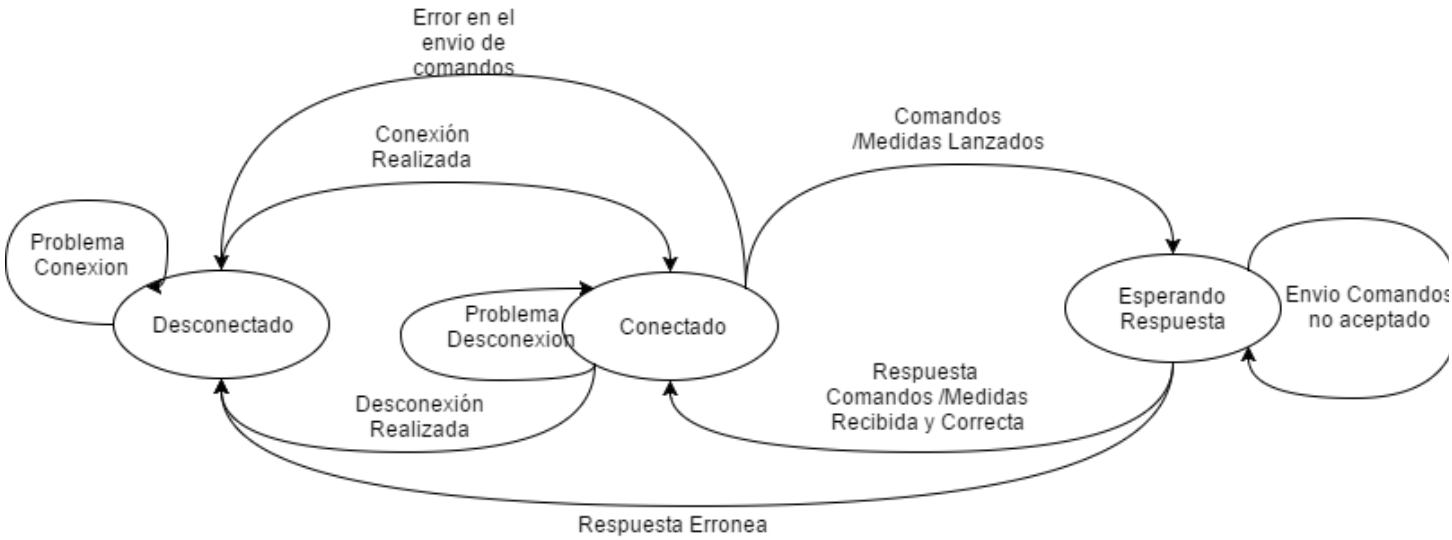
1. Introducción

En esta práctica se va a manejar un Robot Lego vía Bluetooth con una aplicación que tiene que cumplir requisitos de tiempo real suave. En un principio se va a estimar un retardo para la conexión inicial por bluetooth de 500 ms y un retardo en la transmisión y recepción de los comandos de 200 ms (cada vez que se quiere enviar una instrucción al robot hay que enviar y recibir 2 comandos, se ha estimado un retardo de 50 ms para cada una de estas operaciones). El análisis del tiempo se realizará al final de esta wiki y de la práctica, ya que es necesario recoger suficientes datos con el sistema en funcionamiento para poder realizar una estimación de dichos deadlines.

Enlace al proyecto:
https://www.dropbox.com/sh/ccq971487vrw4ye/AADVewoY3aBNZdz504S9NmVUa?dl=0

name="toc-2">2. Maquina de Estados Mealy

Para una correcta comunicación con el robot es necesario tener una maquina de estados que nos permita identificar el estado en que nos encontramos con respecto a la comunicación con el robot.
La máquina de estados que se va a emplear es la siguiente:



Como se pude apreciar en el diagrama, en esta implementación únicamente se han empleado los tres estados siguientes:

1. **Desconectado**. No hay una conexión establecida con el Robot, los únicos botones que realizan su función son el de conectar y el de guardar (Por si se quieren guardar datos de una ejecución anterior).
2. **Conectado**. Hay una conexión establecida y es posible enviar un comando al robot. Todos los botones están habilitados. El botón de conexión marcará "Discconect".
3. **Esperando Respuesta.** Ningún botón estará activo. No es posible realizar ninguna acción puesto que el sistema esta a la espera (con el socket del bluetooth ocupado) de la respuesta del robot.

Durante el desarrollo de la práctica se decidió **no incluir un estado error,** ya que este estado solo sería una transición hasta otro estado. Cuando se produce un error en el envío de comandos o durante la recepción de la respuesta, el sistema deberá pasar a estado de desconexión automáticamente. Si por el contrario se produce un error en la desconexión el sistema interpretará que no se ha podido desconectar por algún motivo y seguirá conectado mientras no haya un error en el socket (si posteriormente hay un error en la comunicación si que pasa al estado desconectado). El conjunto de señales de transición entre estados especificadas en el diagrama anterior tienen nombres similares a las usadas en la implementación.

3. Implementación.

a) Division de Módulos

A la hora de abordar esta práctica se ha decidido dividir la misma en diferentes clases o modulos. En esta implementación se han utilizado las siguientes clases:

1. **Clase MainActivity.** Modulo inicial encargado de gestionar la interfaz, de recibir las peticiones del usuario mediante los botones, de actualizar la interfaz etc...
2. **Clase BluetothConnectionController.** Clase encargada de gestionar las interacciones con el robot. Almacenará el estado de la conexión con el robot. Incluirá el conjunto de hebras encargadas de realizar las comunicaciones. Esta clase extiende a la clase SharedSocker proporcionada en clase, donde se definen los métodos para llevar a cabo la conexión, desconexión y el envío de comandos con el robot.
3. **Clase PowerController.** Clase que agrupa las funciones necesarias para gestionar la potencia de las ruedas del robot. Esta clase será un atributo de la clase del punto 2.
4. **Clase GrabarDatos.** Clase que agrupa las funciones con las que se realiza el volcado de las mediciones temporales en distintos ficheros. Por simplicidad se ha decidido que los valores que se quieren guardar sean accesibles para su modificación desde otras clases.

principal, que será la encargada de habilitar o no los botones. En este caso se ha hecho uso del módulo **`MyBroadcastReceiver.`** Este módulo genera un receptor de señales en la interfaz principal. Además nos permite habilitar un conjunto de señales que pueden ser lanzadas desde cualquier hebra (indicando la interfaz receptora *Context*) que se este ejecutando permitiéndonos comunicarnos de esta manera con la interfaz principal y llevar a cabo las actualizaciones necesarias desde el código de la propia interfaz, en nuestra caso **MainActivity.**

Dentro del código del **MainActivity()** debemos crear una clase **MybroadastReceiver()** que herede de **BroadcastReceiver(),** donde definiremos las acciones que se han de realizar cuando lleguen cada una de las nuevas señales que se han implementado. La función **onReceive()** de esta clase se activa siempre que se lanza alguna de las señales previamente habilitadas. A continuación se muestra el código de la clase **MybroadastReceiver()** .

```
// Clase que extiende a BroadcastReceiver, sirve para manejar el bluetooth y las señales
//que se utilizarán en el programa
class MyBroadcastReceiver extends BroadcastReceiver {
public void onReceive(android.content.Context context, android.content.Intent intent) {
        String action = intent.getAction();

switch (action) {


//Cuando se realiza una conexión se actualiza el texto del botón, el texto que está
            //debajo de este y se pasa la estado conectado
case BluetothConnectionController.AccionesPosibles.ConexionRealizada:
                Log.d("BroadcastReceiver", "ConexionRealizada");
BTController.ModifyStatus(BluetothConnectionController.EstadosPosibles.Conectado);
button_connect.setText(ButtonLegend.Leyenda_Desconectar);
StatusText.setText(StateLegend.Conectado + BTController.mensaje);
break;

//Cuando se realiza una desconexión se actualiza el texto del botón, el texto
            // que está debajo de este y se pasa la estado desconectado
case BluetothConnectionController.AccionesPosibles.DesconexionRealizada:
                Log.d("BroadcastReceiver", "DesconexionRealizada");
BTController.ModifyStatus(BluetothConnectionController.EstadosPosibles.Desconectado);
button_connect.setText(ButtonLegend.Leyenda_Conectar);
StatusText.setText(StateLegend.Desconectado + BTController.mensaje);
break;

//Si hay un problema en la conexión el programa permanece en el estado desconectado
            // e imprime el error en el cuadro de texto
case BluetothConnectionController.AccionesPosibles.ProblemaConexion:
                Log.d("BroadcastReceiver", "ProblemaConexion");
BTController.ModifyStatus(BluetothConnectionController.EstadosPosibles.Desconectado);
button_connect.setText(ButtonLegend.Leyenda_Conectar);
StatusText.setText(StateLegend.Desconectado + " " + BTController.mensaje); //StateLegend.D
esconectado
break;

//Si hay un problema en la desconexión muestra el error por pantalla y
            //permanece conectado. Hay que tener en cuenta que si el error se produce en l
a
            //transmisión de los mensajes por bluetooth salta la señal Error, que si desco
necta
            //los dispositivos
case BluetothConnectionController.AccionesPosibles.ProblemaDesconexion:
                Log.d("BroadcastReceiver", "ProblemaDesconexion");
BTController.ModifyStatus(BluetothConnectionController.EstadosPosibles.Conectado);
button_connect.setText(ButtonLegend.Leyenda_Desconectar);
StatusText.setText(StateLegend.Conectado + BTController.mensaje);
break;


//Si hay un error en el envio o recepción de los mensajes muestra el error por
            // pantalla y pasa al estado desconectado
```

```
case BluetothConnectionController.AccionesPosibles.Error:
                Log.e("BroadcastReceiver", "Error");
BTController.ModifyStatus(BluetothConnectionController.EstadosPosibles.Desconectado);
button_connect.setText(ButtonLegend.Leyenda_Conectar);
                Log.e("Accion error", BTController.mensaje);
StatusText.setText(StateLegend.Desconectado + " Error: " + BTController.mensaje);
                LimpiezaPorError();
break;

//Cuando se lanza un comando se pasa al estado esperando respuesta, desde el cual
            // no se pueden enviar nuevos comandos
case BluetothConnectionController.AccionesPosibles.ComandosLanzados:
                Log.d("BroadcastReceiver", "ComandosLanzados");
BTController.ModifyStatus(BluetothConnectionController.EstadosPosibles.EsperandoRespuesta);
StatusText.setText(StateLegend.EsperandoRespuestaComandos);
break;

//Cuando se reciben los comandos se pasa al estado conectado de nuevo para
            // poder recibir mas comandos
case BluetothConnectionController.AccionesPosibles.ComandosRecibidos:
                Log.d("BroadcastReceiver", "Comandos Recibidos");
BTController.ModifyStatus(BluetothConnectionController.EstadosPosibles.Conectado);
BTController.hebra_comandos = null;
StatusText.setText(StateLegend.Conectado);
break;

//Si se piden datos de manera periodica también se pasa al estado esperando respuesta
case BluetothConnectionController.AccionesPosibles.MedidasPedidas:
                Log.d("BroadcastReceiver", "MedidasPedidas");
BTController.ModifyStatus(BluetothConnectionController.EstadosPosibles.EsperandoRespuesta);
StatusText.setText(StateLegend.Conectado);
break;

//Cuando se reciben las medidas periodicas se vuelve al estado conectado para poder
            //enviar comandos
case BluetothConnectionController.AccionesPosibles.PeriodicasRecibidas:
                Log.d("BroadcastReceiver", "PeriodiasRecibidas");
BTController.ModifyStatus(BluetothConnectionController.EstadosPosibles.Conectado);
PeriodicText.setText("Periodic: " + BTController.Periodic_mensaje_1 + ", " +
BTController.Periodic_mensaje_2);
BTController.hebra_medidas = null;
break;

//Cuando se modifica la velocidad se actualizan en el texto correspondiente los
            // valores de v y w
case BluetothConnectionController.AccionesPosibles.VelocidadModificada:
VelocitiesText.setText("V: " + String.format("%.3f", BTController.ControladorPotencia.v) +
 ", "
+ "W: " + String.format("%.3f", BTController.ControladorPotencia.w)) ;
break;

//Si un comando no se puede enviar porque ya se han enviado otros se muestra
            // un mensaje por pantalla
case BluetothConnectionController.AccionesPosibles.ComandoNoAceptado:
```

```
StatusText.setText(StatusText.getText() + " Comando No aceptado, socket Bluetooth ocupado
 (Medidas en paralelo)");
break;
        }
    }
}
```

Se ve como en función de la señal recibida en cada ocasión se modificara el estado del sistema siguiendo la maquina de estados anterior y se actualizarán los mensajes de texto y el primer botón si corresponde.

El código necesario para la inicialización de este receptor de señales desde la interfaz este el siguiente:

```
//Inicializa una extensión del broadcast receiver para recibir las señales
//que cambian los estados
mybroadcastreceiver = new MyBroadcastReceiver();

//Inicializa las nuevas señales que se van a utilizar desde Bluetooth conection controller
//para cambiar entre estados
InicializarIntents();
```

Siendo la función InicializarIntents() la encargada de habilitar las señales:

```
//Inicializa las señales que se usarán para cambiar de estados
//En este caso las posibles señales y el motivo por el que se lanzan son:
//Conexión realizada - se realiza la conexión
//Dexconexión realizada - se realiza la desconexión
//Problema conexión - hay un problema en la conexión
//Problema dexconexión - hay un problema en la desconexión
//Comandos lanzados - se envian comandos al robot
//Comandos recibidos - se reciben comandos del robot
//Error - hay un error en el envio o recepción de comandos
//Pediodicas recibidas - se reciben los mensajes de la hebra que pide el estado de las
//ruedas cada 50ms
//Medidas pedidas - se han pedido las medidas por parte de la hebra periódica
//Velocidad modificada - se ha modificado la velocidad
//Comando no aceptado - no se pudo enviar un comando porque ya se había enviado otro y
// se estaba esperando la recepción de una respuesta
public void InicializarIntents() {

filter = new android.content.IntentFilter(BluetoothAdapter.ACTION_STATE_CHANGED);
    registerReceiver(mybroadcastreceiver, filter);

    LocalBroadcastManager.getInstance(this).registerReceiver(mybroadcastreceiver,
new IntentFilter(BluetothConnectionController.AccionesPosibles.ConexionRealizada));

    LocalBroadcastManager.getInstance(this).registerReceiver(mybroadcastreceiver,
new IntentFilter(BluetothConnectionController.AccionesPosibles.DesconexionRealizada));

    LocalBroadcastManager.getInstance(this).registerReceiver(mybroadcastreceiver,
new IntentFilter(BluetothConnectionController.AccionesPosibles.ProblemaConexion));

    LocalBroadcastManager.getInstance(this).registerReceiver(mybroadcastreceiver,
new IntentFilter(BluetothConnectionController.AccionesPosibles.ProblemaDesconexion));

    LocalBroadcastManager.getInstance(this).registerReceiver(mybroadcastreceiver,
new IntentFilter(BluetothConnectionController.AccionesPosibles.ComandosLanzados));

    LocalBroadcastManager.getInstance(this).registerReceiver(mybroadcastreceiver,
new IntentFilter(BluetothConnectionController.AccionesPosibles.ComandosRecibidos));

    LocalBroadcastManager.getInstance(this).registerReceiver(mybroadcastreceiver,
new IntentFilter(BluetothConnectionController.AccionesPosibles.Error));

    LocalBroadcastManager.getInstance(this).registerReceiver(mybroadcastreceiver,
new IntentFilter(BluetothConnectionController.AccionesPosibles.PeriodicasRecibidas));

    LocalBroadcastManager.getInstance(this).registerReceiver(mybroadcastreceiver,
new IntentFilter(BluetothConnectionController.AccionesPosibles.MedidasPedidas));

    LocalBroadcastManager.getInstance(this).registerReceiver(mybroadcastreceiver,
new IntentFilter(BluetothConnectionController.AccionesPosibles.VelocidadModificada));

    LocalBroadcastManager.getInstance(this).registerReceiver(mybroadcastreceiver,
new IntentFilter(BluetothConnectionController.AccionesPosibles.ComandoNoAceptado));
}
```

Estas posibles señales o acciones vienen definidas dentro de una clase estática interna de BluetothConnectionController.

```
//Señales que se pueden enviar, se corresponden con las del programa principal
public static class AccionesPosibles {
public static final String ConexionRealizada = "ConexionRealizada";
public static final String ProblemaConexion = "ProblemaConexion";
public static final String DesconexionRealizada = "DesconexionRealizada";
public static final String ProblemaDesconexion = "ProblemaDesconexion";
public static final String ComandosLanzados = "ComandosLanzados";
public static final String ComandosRecibidos = "ComandosRecibidos";
public static final String Error = "Error";
public static final String MedidasPedidas = "MedidasPedidas";
public static final String PeriodicasRecibidas = "PeriodicasRecibidas";
public static final String VelocidadModificada = "VelocidadModificada";
public static final String ComandoNoAceptado = "ComandoNoAceptado";

}
```

Es importante notar como son estas mismas señales las que se han incluido en la maquina de estados como las encargadas de llevar a cabo la transición entre estados.

c) Implementación de la Maquina de estados

El almacenamiento de los estados esta dentro de la clase **BluetoothConnectionController().** Se han definido los posibles estados y una serie de métodos para su gestión:

```
//Funciones para modificar y leer el estado en el que se encuentra el programa, se usan pa
ra
//actualizar el estado e impedir el envio de comandos en momentos no indicados
public void ModifyStatus(int NewStatus) {
Status = NewStatus;
}

public int getStatus() {
return Status;
}


//Descripción de estados, se corresponden con los estados del programa principal
public static class EstadosPosibles {
public static final int Desconectado = 1;
public static final int Conectado = 2;
public static final int EsperandoRespuesta = 3;
}
```

Estos métodos de modificación del estado son llamados desde el receptor de señales de la interfaz principal cuando se reciben las posibles señales habilitadas para la transición de estado.

4. Interfaz

La inicialización de la interfaz se realiza dentro del **onResume()** del **MainActivity():**

```
public void onResume() {

super.onResume();
    setContentView(R.layout.activity_main);
```

```
//Inicializa y nombra los TextView y el botón que tiene que cambiar su texto
//para poder manejarlos después
//Los botones tienen definido en el layout la función a la que llaman
InicializarReferencias();
```

Siendo dicha función de inicialización:

```
//Inicializa y nombra los TextView y el botón que tiene que cambiar su texto
private void InicializarReferencias() {
button_connect = (Button) findViewById(R.id.connect);
StatusText = (TextView) findViewById(R.id.textView1);
VelocitiesText = (TextView) findViewById(R.id.textView2);
PeriodicText = (TextView) findViewById(R.id.textView3);
}
```

Cada uno de los botones de la interfaz tiene una función asignada. Esto se consigue dentro del activity main con este comando: **android:onClick="<Función_Asignada".**
A modo de ejemplo se incluye una imagen con el código completo para un botón, en este caso el de conexión, donde se señala la línea que designa la función:

```
<Button
    android:text="CONNECT TO LEGO"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/connect"
    android:onClick="Connect_Called"
    android:layout_weight="0.37" />
```

Cada uno de los botones esta configurado para ser responsivo en cualquier momento de operación del sistema, es decir, no debe dar la sensación de quedarse bloqueados en ningún momento. Para ello, la función de cada botón será realizada por una hebra de forma que su actividad no interfiera con el resto de botones. El sistema de hebras se explicará más tarde.

Por otro lado, en función del estado en que nos encontremos se deberá decidir si el botón estará habilitado o no. En el caso del botón de conexión/desconexión además se tiene que actualizar su leyenda. Esto se ha implementado en las funciones que se llaman desde cada botón evaluando el estado en el que estamos cuando se pulsan. Por ejemplo, para el botón de conexion/desconexión tendremos el siguiente código:

```
//Cuando se pulsa el botón para conectar/desconectar se habilita el bluetooth si no está h
abilitado
// y se llama a la función connect o desconect del bluetooth conection controller para con
ectar
//o desconectar
public void Connect_Called(View view) {

    EnableBluetoth();

if (mBluetoothAdapter.isEnabled()) {
switch (BTController.getStatus()) {
case BluetothConnectionController.EstadosPosibles.Desconectado:
BTController.Connect();
break;
case BluetothConnectionController.EstadosPosibles.Conectado:
BTController.Disconnect();
break;
        }
    }
}
```

Todos botones que cambian la velocidad para W y V tienen una implementación similar, evalúan si estamos en estado conectado y si es así actualizan la velocidad y llaman a la función encargada del envío de los comandos. Si la maquina no se encuentra en el estado conectado mandan una señal de comando no aceptado para que se imprima en el primer cuadro de texto. A modo de ejemplo se incluye el código de uno de estos botones:

```
//Si se pulsa el botón para aumentar v y el programa se encuentra en el estado conectado
//se modifica v y se llama a la funcion de bluetooth conection controller que envia el com
ando
//Si el socket está ocupado no se envia el comando y se muestra por pantalla el mensaje in
dicandolo
public void V_plus(View view) {
switch (BTController.getStatus()) {
case BluetothConnectionController.EstadosPosibles.Conectado:
            Log.d("V_plus", "Ha entrado en la función V_plus");
            Log.d("V_plus", "Aumentando la potencia");
ControladorPotencia.incrementar_V();
            Log.d("V_plus", "Enviando los comandos");
BTController.EnviarComando();
break;
default:
            LocalBroadcastManager.getInstance(this).sendBroadcast(new Intent(BluetothConne
ctionController.AccionesPosibles.ComandoNoAceptado));
break;
    }
}
```

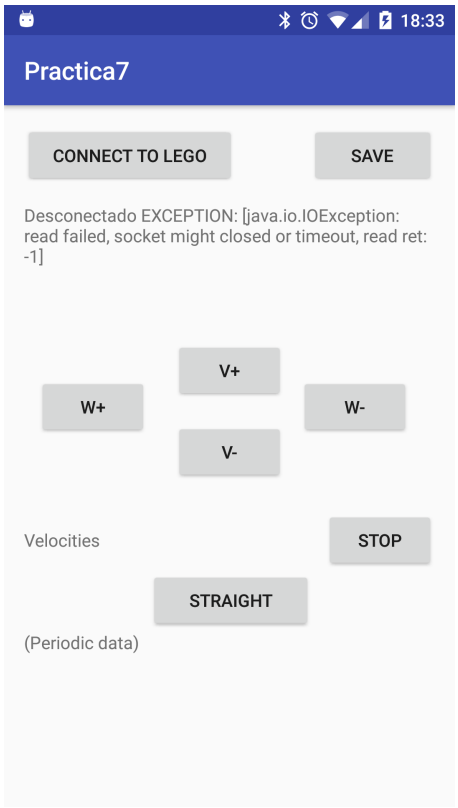
Un botón con una implementación diferente es el "SAVE", puesto que da igual en el momento en el que nos encontremos ya que siempre se realizará el volcado de datos a un fichero. Se ha realizado de esta manera porque el usuario puede querer guardar datos de una ejecución anterior estando desconectado y dicha acción no influye en el envío de mensajes bluetooth. Aunque puede ralentizar el tiempo de ejecución en el envío y recepción se ha preferido guardar los datos en todo momento para evitar que cada medio segundo esta acción esté bloqueada por la hebra que solicita datos periodicamente a las ruedas. A continuación se muestra el código correspondiente a la función save:

```
public void Save_Called(View view) {
BTController.save_Datos.Llamar_salvar();
}


//Si hay un error que no sea de desconexión y el socket está conectado se desconecta el so
cket
private void LimpiezaPorError() {
    Log.d("LimpiezaError", "Desconectando y Matando hebra periodica");

if (BTController.socket.isConnected()) {
BTController.Disconnect();
    }
}
```

En función del estado en que nos encontremos los elementos de la interfaz variarán sus contenidos de texto en caso de ser *TextView o* sus etiquetas sin son botones. Se van a añadir algunas capturas de la interfaz para diferentes estados del sistema.

Para el estado desconectado:

Para el estado desconectado posterior a un intento fallido de conexión:



Para el estado conectado sin haber enviado ningún comando de movimiento:

Para el estado conectado habiendo enviado comandos de movimiento y, por lo tanto, con el robot moviéndose:

5. Clase BluetoothConectionController

Debido a su importancia en este proyecto, se va a explicar esta clase en una sección a parte.

La gestión de la conexión bluetooth y el envío de mensajes se realiza con esta clase, que **es una extensión de la clase SharedSocket** disponible en el campus virtual de la asignatura. En el onResume de la clase MainActivity se inicializa un controlador llamado BTController al que se llamará para realizar las funciones relacionadas con el envío de paquetes. Cuando BTController recibe una instrucción desde MainActivity se encarga de lanzar las hebras correspondientes para la petición y recepción de datos y de responder con las señales adecuadas. Es importante mencionar que esta clase dispone de un campo para guardar la dirección del robot con la que se conecta, una variable pública para guardar texto que quiera ser leído desde otras clases y una referencia a PowerController para poder acceder a los valores de potencia de las diferentes ruedas.

Un herramienta importante empleada en esta clase son los semáforos o elementos *Lock.* Uno de los atributos de esta clase es un elemento de este tipo que nos permitirá evitar que dos hebras distintas (una de medidas y otra de comandos) intenten acceder al socket simultáneamente lo cual provocaría un error en el sistema, ya que, por ejemplo, tras enviar un comando de movimiento estaremos a la espera de la respuesta, si en ese momento entra una hebra de medidas enviará su solicitud y el próximo mensaje de vuelta será tomado por ambas hebras y provocará un error en la hebra de medidas al no ser la respuesta esperada.

A continuación se va a realizar una explicación de como se maneja la conexión bluetooth.

- **Connect() / Disconnect():** Cuando el MainActivity quiere conectarse o desconectarse mediante bluetooth llama a las funciones **Connect() o Disconnect()**. Estas funciones lanzan dos hebras, **connect y desconectarse** respectivamente. La hebra connect hace uso de la función de SharedSocket connect (en este caso la elección de nombres no es la más adecuada) para realizar la conexión y mide el tiempo que tarda en realizarse la conexión. Si la conexión es correcta se pasa al estado conectado, si no se envía una señal de error. En el caso de la hebra desconectarse se hace uso de la función unconnect se SharedSocket para desconectarse, pasando al estado desconectado si se realiza con éxito o mandando una señal de problema en la desconexión si no.

Mencionar como será el siguiente comando:

```
LocalBroadcastManager.getInstance(ctx).sendBroadcast(new Intent(AccionesPosibles.ConexionRealizada));
```

el encargado de crear la señal que avise a la interfaz principal de que se ha realizado la conexión correctamente o este otro comando:

```
LocalBroadcastManager.getInstance(ctx).sendBroadcast(new Intent(AccionesPosibles.ProblemaConexion));
```

En caso de que que haya habido algún tipo de error durante la conexión.

Para la hebra de la desconexión tendremos algo similar pero en este caso utilizando las señales pertinentes.

```
//Cuando conecta se llama a la hebra que se encarga de la conexión
public void Connect() {

    (new Thread(new Connect())).start();
}

//Cuando desconecta se llama a la hebra que se encarga de la desconexión
public void Disconnect() {

    Thread desconectar_hebra = new Thread(new Desconectar());
    desconectar_hebra.start();
}
```

```
//Hebra de conexión
//Prmero crea una variable para medir el tiempo y utiliza la función connect de sharedsock
et
// para realizar la conexión. Si la conexión se realiza con exito connect devuelve una cad
ena
// vacía, por lo que se compara esta y si ese es el caso se manda la señal de conexión rea
lizada
//, se inicia la hebra que pide datos periodicamente a las ruedas y se guarda el tiempo qu
e ha
// tardado la conexión. Si por el contrario hay un error en la conexión se manda una señal
// indicando que hay un problema en la conexión. El programa principal leerá la variable m
ensaje
// para mostrar el error que se ha producido.

public class Connect implements Runnable {
public void run() {
long tiempo_inicial, tiempo_final;
        tiempo_inicial = System.nanoTime();
        create(BluetoothAdapter.getDefaultAdapter().getRemoteDevice(address));
        String out = connect();

//Log.d("Estado Socker", toString(socket.isConnected()));
mensaje = out;
        Log.d("Conectando", mensaje);
if ("".equals(out)) {
            LocalBroadcastManager.getInstance(ctx).sendBroadcast(new Intent(AccionesPosibl
es.ConexionRealizada));
hebra_periodica = new Thread(new Periodic_heb());
hebra_periodica.start();
            out = "";

            tiempo_final = System.nanoTime();
save_Datos.tiempo_conexion[save_Datos.indice_tiempo_conexion] =
                    tiempo_final - tiempo_inicial;
save_Datos.indice_tiempo_conexion++;

        } else {
            LocalBroadcastManager.getInstance(ctx).sendBroadcast(new Intent(AccionesPosibl
es.ProblemaConexion));
        }
    }
}

//Hebra de desconexión
//Utiliza la función unconnect de sharedsockets para relizar la desconexión, si esta se pr
oduce
// con exito se manda la señal correspondiente. Si hay un error se manda una señal de prob
lema
// en la desconexión
public class Desconectar implements Runnable {

public void run() {
```

```
        String out = unconnect();

if ("".equals(out)) {
            LocalBroadcastManager.getInstance(ctx).sendBroadcast(new Intent(AccionesPosibl
es.DesconexionRealizada));
        } else {
            LocalBroadcastManager.getInstance(ctx).sendBroadcast(new Intent(AccionesPosibl
es.ProblemaDesconexion));
        }
    }
}
```

- **EnviarComando():** Si se quiere enviar un comando de cambio de velocidad se llama a esta función. Esta arranca la hebra **SendCommad** que es la encargada de enviar y recibir dichos comandos y sus respuestas. Esta hebra bloquea el envio de mensajes con un lock, envia dos comando seguidos y espera ambas respuestas. Si se produce un error en algún paso envía una señal de error, si no envía una señal de comandos recibidos. Cuando se manda el primer comando además se envía una señal de comandos enviados para impedir que otros botones lancen nuevos comandos. Por último libera el lock y mide el tiempo que ha tardado en realizar esta operaciones.

```
//Cuando se envia un comendo llama a la hebra que hace esta acción
public void EnviarComando() {

hebra_comandos = new Thread(new SendCommad());
hebra_comandos.start();
}
```

```
//Hebra que envía los comandos cuando se pulsa un botón para cambiar la velocidad. Primero
// declara una variable para medir el tiempo y bloquea el lock para impedir que otras hebr
as
// envien comandos a la vez. Envia un comando SetOutputState, envia una señal para cambiar
 de estado
// y si la retransmision se produce correctamente envia el segundo comando para la otra ru
eda.
// A continuación espera las dos respuestas, si hay un error en alguna manda una señal de
 error,
// si no, una de comandos recibidos. Por último libera el lock y mide el tiempo que ha tar
dado en
// enviar y recibir ambas respuestas. El envio y recepción de comandos se hacen con las fu
nciones
// de SharedSocket correspondientes y la comprobación de errores con los string que dichas
// funciones devuelven.
public class SendCommad implements Runnable {

public void run() {
long tiempo_inicial, tiempo_final;
        tiempo_inicial = System.nanoTime();
        Command.SetOutputState c;
        String out;

        c = new Command.SetOutputState((byte) 0x00, ControladorPotencia.powerForWheel(fals
e));

lock.lock();

try {
            out = sendCommand(c);

if (out.equals("")) {
                c = new Command.SetOutputState((byte) 0x02, ControladorPotencia.powerForWh
eel(true));
                LocalBroadcastManager.getInstance(ctx).sendBroadcast(new Intent(AccionesPo
sibles.VelocidadModificada));

                out = sendCommand(c);
if (out.equals("")) {
mensaje = "";
                    LocalBroadcastManager.getInstance(ctx).sendBroadcast(new Intent(Accion
esPosibles.ComandosLanzados));

                    Response resp1 = receiveResponse(timeout_respuesta_nano);

if (resp1 instanceof Response.Error) {
                        Log.d("Respuesta", "Error en la respuesta");
                        LocalBroadcastManager.getInstance(ctx).sendBroadcast(new Intent(Ac
cionesPosibles.Error));
mensaje = ((Response.Error) resp1).errortxt;

                    } else {
                        Log.d("Respuestas", "Repuesta 1 Recibida");
```

```
                        Response resp2 = receiveResponse(timeout_respuesta_nano);

if (resp2 instanceof Response.Error) {
                            Log.d("Respuesta", "Error en la respuesta");
                            LocalBroadcastManager.getInstance(ctx).sendBroadcast(new Inten
t(AccionesPosibles.Error));
mensaje = ((Response.Error) resp2).errortxt;
                        } else {
                            Log.d("Respuestas", "Repuesta 2 Recibida");
                            LocalBroadcastManager.getInstance(ctx).sendBroadcast(new Inten
t(AccionesPosibles.ComandosRecibidos));
                        }
                    }
                } else {
                    LocalBroadcastManager.getInstance(ctx).sendBroadcast(new Intent(Accion
esPosibles.Error));
mensaje = out;
                }
            } else {
                LocalBroadcastManager.getInstance(ctx).sendBroadcast(new Intent(AccionesPo
sibles.Error));
mensaje = out;
            }
        } finally {
lock.unlock();
        }
        tiempo_final = System.nanoTime();
save_Datos.tiempo_comandos[save_Datos.indice_tiempo_comandos] =
                tiempo_final - tiempo_inicial;
save_Datos.indice_tiempo_comandos++;
    }
}
```

- **Periodic_heb():** Esta hebra se inicia automáticamente cuando se realiza la conexión con el robot y termina cuando se corta la conexión del socket. Se encarga de lanzar cada 500 ms una nueva hebra **SendCommand_ReadData.**

```
//Hebra que realiza las peticiones periodicas, cuando se arranca y hasta que el socket se
// desconecte lanza una nueva hebra cada 500 ms que se encargará de pedir los datos de las
 ruedas.
public class Periodic_heb implements Runnable {
public void run() {
while (socket.isConnected()) {
try {
                Thread.sleep(500);
hebra_medidas = new Thread(new SendCommad_ReadData());
hebra_medidas.start();

            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

- **SendCommand_ReadData:** Esta hebra comprueba si hay conexión con el socket y si es así es la encargada de pedir el estado de las ruedas. Para ello bloquea el lock, manda un comando y espera una respuesta. A continuación manda otro mensaje y espera otra respuesta. Si en algún momento hay un error manda una señl de error, si no manda una señal de comandos recibidos. Al igual que en el caso anterior cuando envía el primer comando manda una señal de comando enviado para cambiar a dicho estado.

```
//Hebra que se lanza desde la hebra Periodic_heb. Al igual que SendCommand se encarga de e
nviar y
// recibir comandos, en esta ocasión GetOutputState. Si el socket se encuentra conectado b
loquea
// el lock, envia un comando y una señal de comando enviado y espera la respuesta. A conti
nuación envia el siguiente comando y
// espera la otra respuesta. Si en algún momento se produce un error se envia una señal de
// error, esto se comprueba con las cadenas que devuelven las funciones de SharedSocket. S
i no
// se produce un error envia una señal periodicas recibidas para pasar a conectado. Por úl
timo
// libera el lock y mide el tiempo que ha tardado en la retransmisión.
public class SendCommad_ReadData implements Runnable {

public void run() {

        Command.GetOutputState c;
        String out;
long tiempo_inicial, tiempo_final;
        Log.d("Log", "Entrando en hebra de solicitar medidas");

if (socket.isConnected()) {
            tiempo_inicial = System.nanoTime();

            LocalBroadcastManager.getInstance(ctx).sendBroadcast(new Intent(AccionesPosibl
es.MedidasPedidas));
            c = new Command.GetOutputState((byte) 0x00);
lock.lock();

try {
                out = sendCommand(c);
                Log.d("Periodic", "Primer Comando Lectura Enviado");

if (out.equals("")) {
                    Response resp1 = receiveResponse(timeout_respuesta_nano);

if (resp1 instanceof Response.Error) {
                        Log.d("Periodic", "Error en la respuesta");
mensaje = ((Response.Error) resp1).errortxt;
                        LocalBroadcastManager.getInstance(ctx).sendBroadcast(new Intent(Ac
cionesPosibles.Error));

                    } else {
                        c = new Command.GetOutputState((byte) 0x02);
                        out = sendCommand(c);

if (out.equals("")) {

                            Response resp2 = receiveResponse(timeout_respuesta_nano);
if (resp2 instanceof Response.Error) {
                                Log.d("Periodic", "Error en la respuesta");
mensaje = ((Response.Error) resp2).errortxt;
                                LocalBroadcastManager.getInstance(ctx).sendBroadcast(new I
```

```
ntent(AccionesPosibles.Error));
                            } else {
                                Log.d("Periodic", "Repuesta 2 Recibida");
Periodic_mensaje_1 = Double.toString((double) ((Response.GetOutputState) resp1).power);
Periodic_mensaje_2 = Double.toString((double) ((Response.GetOutputState) resp2).power);
                            LocalBroadcastManager.getInstance(ctx).sendBroadcast(new I
ntent(AccionesPosibles.PeriodicasRecibidas));
                            }
                        } else {
mensaje = out;
                            LocalBroadcastManager.getInstance(ctx).sendBroadcast(new Inten
t(AccionesPosibles.Error));
                        }
                    }

                } else {
mensaje = out;
                    LocalBroadcastManager.getInstance(ctx).sendBroadcast(new Intent(Accion
esPosibles.Error));

                }
            } finally {
lock.unlock();
                tiempo_final = System.nanoTime();
save_Datos.tiempo_datos_periodico[save_Datos.indice_tiempo_datos_periodico] =
                        tiempo_final - tiempo_inicial;
save_Datos.indice_tiempo_datos_periodico++;
            }
        }
    }
}
```

Cuando se envían o reciben comandos se hacen uso de las funciones de SharedSocket. Dichas funciones devuelven cadenas vacías si el paso de mensajes se realiza con éxito. Si no, devuelven un texto con el error que se ha producido. Para comprobar el correcto funcionamiento del bluetooth se comparan los mensajes devueltos, que se guardan en la variable **mensaje**, con cadenas vacías. Desde el MainActivity se tiene acceso a esta variable para poder actualizar el texto correspondiente. Junto al código se han adjuntado mensajes explicativos para facilitar su comprensión.

6. Almacenamiento de las mediciones de tiempo

7. Análisis de Tiempos

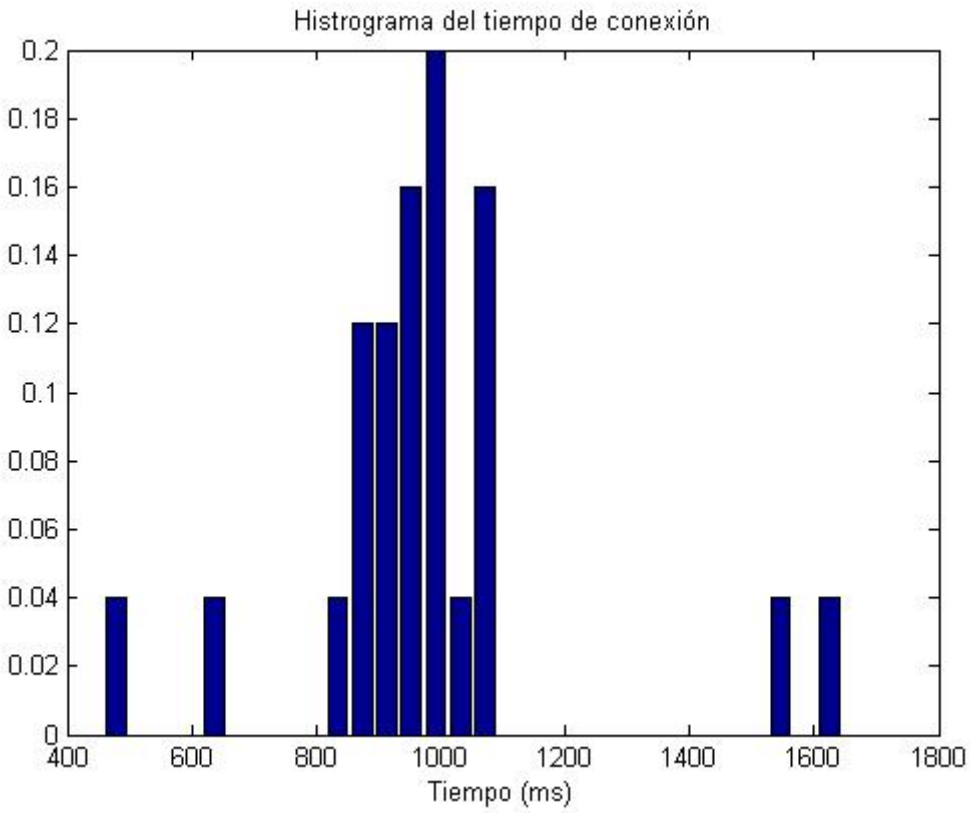
Uno de los principales objetivos de la práctica ha sido analizar y determinar los tiempos empleados en las comunicaciones con el Robot, ya que esto nos permitirá posteriormente establecer los deadlines esperados del sistema. Los tiempos medidos han sido los siguientes:
a. **Tiempo de conexión,** esto es, desde que se envia el comando con la petición de la conexión hasta que el Robot nos contesta aceptando dicha conexión.
b. **Tiempo entre el envío del comandos y las respuestas**, esto es, desde que se envía el primer comando de movimiento (recordemos que debemos enviar dos comandos seguidos uno para cada rueda), hasta que se recibe la respuesta del segundo comando de movimiento
c. **Tiempo entre el envío de la petición de la medida y las respuestas,** esto es, desde que se envía el primer comando de medida de la potencia de las ruedas hasta que se recibe la respuesta de la medida para la otra rueda. Recordar que a diferencia del caso anterior, ahora mandaremos una petición de medidas y deberemos esperar la respuesta antes de poder solicitar la medida para la otra a ruedad, por lo que apriori este intercambio de mensajes debería durar más.

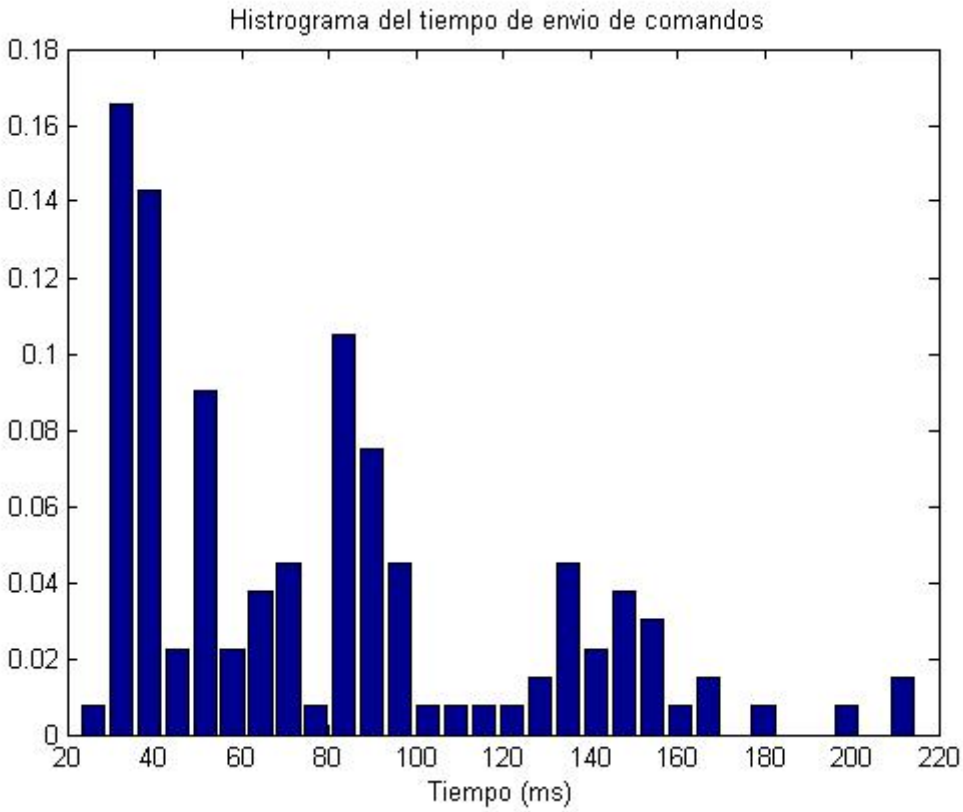
Para comenzar con el análisis de estos tiempos se ha hecho uso de Matlab, usando este Script:

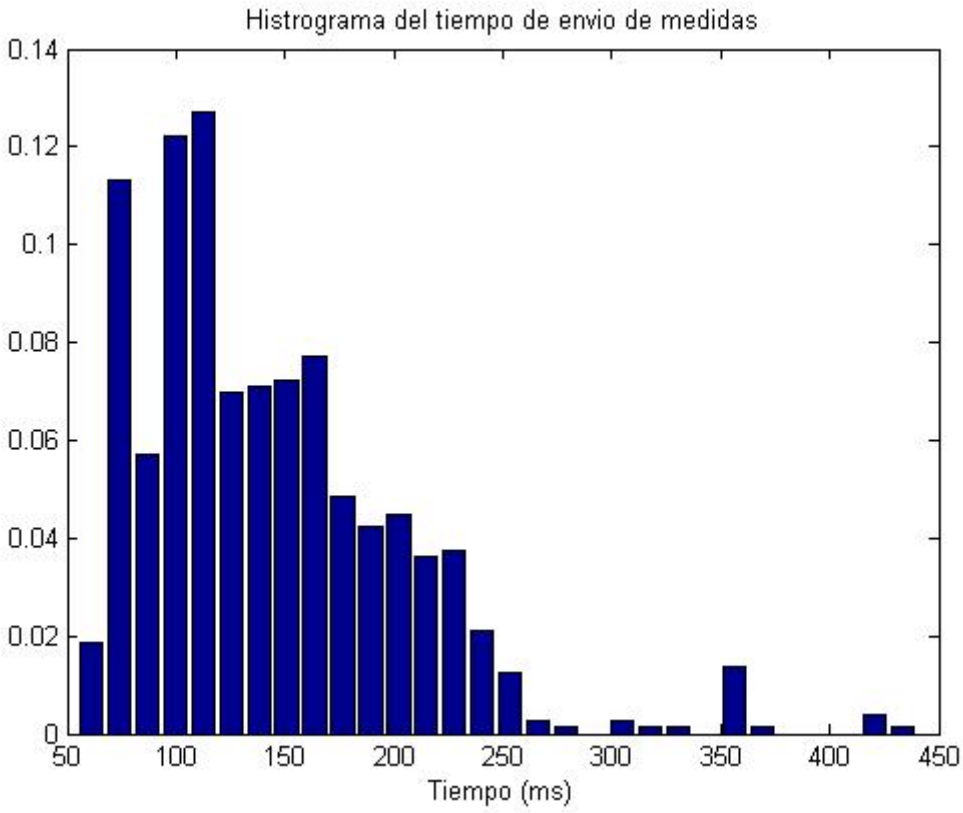
```
figure()
Datos_tiempo_comandos_todos  = 'Datos_tiempo_comandos_todos.txt';
tiempos = load(Datos_tiempo_comandos_todos);
tiempos = tiempos/1e6;
[f,x] = hist(tiempos, 30);
bar(x,f/sum(f))
title('Histrograma del tiempo de envio de comandos ')
xlabel('Tiempo (ms)')

media_1 = mean(tiempos);
Y_1 = prctile(tiempos,[50 80 90 95 99]);
```

Donde deberemos cambiar *'Datos_tiempo_comandos_todos.txt'* , por el fichero que contenga las medidas de tiempo que se quieran evaluar.

Los histogramas obtenidos para las 3 diferentes medidas son los siguientes:







Observando dichos histogramas se puede comprobar que:
1. Es la realización de la conexión lo que más tiempo consume.
2. Es el envío de comandos de movimiento lo que menos tiempo consume. Esto es lo esperado ya que en este caso se envían dos comandos a la vez mientras que en las medidas debemos esperarnos a que llegue la respuesta del primer comando para poder enviar el siguiente.
3. El envió de medidas tiene unos tiempos que facilitan su predictibilidad, lo cual es ideal en los Sistema de Tiempo Real, ya que estan "mas agrupados" en la parte izquierda de la gráfica (esto es para valores pequeños por debajo de 250 ms) en comparación con las distribuciónes de tiempos de los demás elementos evaluados.

Para analizar mejor estos tiempos se incluye la siguiente tabla:

Tiempos Evaluados	Media	Mediana	Perc. 80	Perc. 90	Perc. 95	Perc.99
Tiempos Evaluados	**Media**	**Mediana**	**Perc. 80**	**Perc. 90**	**Perc. 95**	**Perc.99**
Conexión	981.8 ms	971.5 ms	1053.7 ms	1075.9 ms	1577.8 ms	1644.2 ms
Comandos	78.7 ms	69.9 ms	125.3 ms	149.2 ms	159.8 ms	209.9 ms
Medidas	142.6 ms	129.3 ms	190.4 ms	221.5 ms	232.9ms	356.5 ms

Tomando dicha tabla y en función de esos resultados se puede garantizar que en un 90 % de las veces tendremos un sistema que tardará en torno 1.1 segundos en la conexión, 150 milisegundos en el envío de comandos y 222 milisegundos en la solicitud de medidas.

8. Vídeos ejemplo del comportamiento de la interfaz y del robot