UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y DISEÑO INDUSTRIAL

TRABAJO FIN DE MÁSTER

Máster en Ingeniería Electromecánica (Mecatrónica)

# CONTROL Y MONITORIZACIÓN DE UN INVERNADERO A TRAVÉS DE UNA APLICACIÓN MÓVIL

Departamento de Ingeniería Eléctrica, Electrónica, Automática y Física Aplicada

Autor: Barroso García, Andrés

Tutores: Cedazo León, Raquel
M. Al–Hadithi Abdul Qadir, Basil

Firma autor

Firma tutores

Madrid
Junio de 2015

*Control y Monitorización de un Invernadero a través de una Aplicación Móvil*

*2015*

*Barroso García, Andrés*

*Escuela Técnica Superior de Ingeniería y Diseño Industrial*

*Universidad Politécnica de Madrid*

*https://www.youtube.com/user/becaelai* *Proyecto ICRA*

*http://androidelectronic.esy.es/* *Proyectos- TFM: Control y Monitorización de un Invernadero a través de una Aplicación Móvil*

*http://icra.blogspot.es/*

# AGRADECIMIENTOS

Primero de todo agradecer la ayuda y dedicación de mis tutores, sobre todo a mi tutora Raquel, porque debido a varios motivos personales, he tenido que desplazar la fecha de entrega, y me he visto obligado a robarle más tiempo del que había planeado.

Aprovechar para agradecerle también todo el tiempo que me dedicó en esos dos años de beca que realicé en la universidad, en los cuales aprendí muchas cosas que no hubiera podido aprender de otra forma gracias a ella. Y ya sea dicho de paso, felicitarle por haber dado a luz a su primera hija mientras estaba desarrollando este trabajo.

Por supuesto agradecer a mis padres su confianza y apoyo desde siempre, el cual me ha permitido alcanzar lo que me he propuesto en esta vida y poder escribir las líneas de este proyecto.

Agradecer a mi novia Beatriz todo el apoyo y cariño que siempre tiene almacenado en su corazón para mí, dándomelo siempre que lo he necesitado. Sin ella jamás hubiera llegado tan lejos. Y darle gracias por su paciencia, debido a que el tiempo invertido en esta tesis es tiempo que no hemos podido disfrutar el uno del otro.

Mencionar la ayuda de ese pequeño, pero intenso, grupo de amigos que tengo, porque una sugerencia siempre se agradece para dar forma a una idea. Y porque de poco vale dedicar tanto tiempo a un proyecto si no tienes a nadie con quien compartirlo.

Dar las gracias a todos los amigos y familiares que se han interesado por esas fotos raras que ponía en el *whatsapp* y *facebook*, mostrando mis últimos avances en la maqueta.

Y por último, agradecer a la comunidad de internet su estupenda contribución a la divulgación de la ciencia y el conocimiento, porque gracias a esa labor, es posible desarrollar ideas como esta, permitiendo que este proyecto sirva de ayuda y guía a futuras personas que lo necesiten.

*Andrés Barroso García*

# MOTIVACIÓN

*Y llegará el día en el que, al darle otra vuelta de tuerca a las cosas, descubrirás que no hay nada tan gratificante como poder comprender lo que nos rodea y ser capaz de mejorarlo.*

# RESUMEN

Los invernaderos permiten tener un mayor control del entorno donde crecen las plantas. Son utilizados para aumentar la calidad y rendimiento de las plantaciones en ciertas ubicaciones que tienen estaciones cortas de crecimiento, o bien malas condiciones de iluminación debido a las localizaciones geográficas, por lo que permiten mejorar la producción de alimentos vegetales en entornos extremos.

En este proyecto se ha desarrollado una maqueta de un invernadero y se propone el uso del microcontrolador *Arduino* y del sistema operativo *Android*, con el objetivo de lograr una tarea de control y monitorización sobre dicha maqueta.

Por una parte, se utiliza la placa *Arduino* como tarjeta controladora del sistema y, a su vez, como tarjeta adquiridora de datos, y por otra parte se ha desarrollado una aplicación *Android* capaz de monitorizar y supervisar el estado del invernadero.

Para llevar a cabo el flujo de información entre el invernadero y los dispositivos de monitorización, se ha desarrollado una aplicación servidor bajo código *C++*, capaz de administrar la información del invernadero en una base de datos *MySQL* y, de forma concurrente, atender las peticiones de los clientes *Android* registrados, proporcionándoles la información que soliciten, y ejecutando las acciones que reciben.

**Palabras clave:** invernadero, *Android*, *Arduino*, hardware libre, código abierto

# ABSTRACT

Greenhouses allow farmers to control everything on the place where the plants are bound to grow. This contributes to raise both quality and gross production output of their crops in some areas where the viable growth span is short, or the environmental conditions such as day light time due to the geographical location. Therefore because of this farmer are able to increase the production of eatables vegetables in harsh environments.

In this project a greenhouse model has been made and the use of the *Arduino* micro controller and operative system *Android* is suggested in order to be able to monitor and perform some control over the model.

On the one hand the *Arduino* board is used both as the controller board of the whole system and as the data acquisition card. On the other hand an *Android* app has been developed in order to track and monitor the state of the greenhouse

In order to perform de data transmission between the greenhouse and the monitoring devices, a server app under *C++* code has been developed. This server app is able to manage the information captured at the greenhouse into a *MySQL* database and concurrently deal with the request made by previous registered *Android* app user providing them the information requested and executing all the orders given.

**Keywords:** greenhouse, *Android*, *Arduino*, open hardware, open source

# Contenido

# Lista de figuras

# LISTA DE TABLAS

# Lista de acrónimos

Tabla 1: Listado de acrónimos

| Acrónimo | Significado |
|---|---|
| **ADT** | *Android* Development Tools |
| **AJAX** | Asynchronous JavaScript And XML |
| **API** | Application Programming Interface |
| **APK** | Application Package File |
| **APP** | Application |
| **BD** | Base de Datos |
| **GCM** | Google Cloud Messaging |
| **GLP** | Gas Licuado del Petróleo |
| **HTTP** | HyperText Transfer Protocol |
| **HTTPS** | HyperText Transfer Protocol Secure |
| **ICRA** | Invernadero Controlado Remotamente por *Android* |
| **IDE** | Entorno de Desarrollo Integrado |
| **IGU** | Interfaz Gráfica de Usuario |
| **IU** | Interfaz de Usuario |
| **JSON** | JavaSript Object Notation |
| **OASIS** | Open Acess Same-Time Information System |
| **POO** | Programación Orientada a Objetos |
| **PWM** | Pulse Width Modulation |
| **REST** | REpresentational State Transfer |
| **SDK** | Software Development Kit |
| **SGL** | Skia Graphics Library |
| **SOAP** | Simple Object Acess Protocol |
| **SQL** | Structructered Query Language |
| **SSL** | Secure Sockets Layer |
| **TTL** | Transistor-Transistor Logic |
| **UART** | Universal Asynchronous Receiver-Transmitter |
| **UECS** | Ubiquitous Environmental Control System |
| **URL** | Uniform Resource Locator |
| **W3C** | World Wide Web Consortium |
| **XML** | Extensible Markup Language |

# Capítulo 1.
# Introducción

En este capítulo se expone la motivación del proyecto, así como los objetivos fijados para su realización. Además se describe la estructura de los capítulos y la simbología utilizada en el documento.

## 1.1. MOTIVACIÓN

La idea de este proyecto surge a partir de la Tesis Final de Máster "*Puesta en Marcha de un Sistema de medición de Humedad y Temperatura en un Invernadero*", desarrollada por el compañero de máster, Carlos Loor. Tras su labor en la construcción de un invernadero *(Ver "Figura 1")* y el control de la temperatura y la humedad, surge la necesidad de poder monitorizar el estado de las variables de una forma rápida y cómoda. Es así como nace el **proyecto ICRA**, *Invernadero Controlado Remotamente por Android,* con el propósito de desarrollar unos servicios que permitan al usuario controlar y monitorizar el estado del invernadero desde cualquier punto geográfico. Se ofrece la posibilidad de modificar rangos de trabajo de actuadores, gestionar alarmas, visualizar variables… incluso poder trazar gráficas de evolución de los parámetros que intervienen en el proceso de crecimiento de la plantación.

Es aquí donde los móviles de última generación hacen posible desarrollar esta herramienta gracias a plataformas como *Android,* la cual da la posibilidad de crear aplicaciones móviles al gusto del consumidor, que integren estas funcionalidades y permitan llevar a cabo tareas de mantenimiento y supervisión del sistema de forma remota.

La necesidad del ser humano de controlar su entorno y modificarlo en base a sus necesidades y la búsqueda del hombre por conseguir una vida más cómoda y segura, son los pilares que empujan este proyecto.

**Figura 1: Invernadero SIGrAL**

## 1.2. OBJETO Y ALCANCE DEL PROYECTO

El presente proyecto tiene varios objetivos:

- **Diseñar y construir una maqueta** representativa de un invernadero, que integre los sensores y actuadores necesarios para poder llevar a cabo las tareas de control y monitorización.

  Se llevará a cabo el control de tres variables: temperatura, humedad y calidad del aire; y se actuará sobre tres periféricos de salida: sistema de calefacción, humidificación y ventilación del recinto.

- **Desarrollo del software** para la **placa Arduino** que controla todos los componentes del invernadero y gestiona las alarmas.

  Se implementarán métodos para el control de variables y se desarrollarán funciones de comunicación cliente-servidor para el flujo de información.

- **Desarrollo de un servidor** para la recepción de datos y control del invernadero, almacenamiento de los valores en una base de datos y que a su vez ofrezca los servicios necesarios para su monitorización y control remoto.

- **Desarrollo de una aplicación** cliente en ***Android***.

  Se ofrecerán las herramientas de monitorización de alarmas, control de rangos de trabajo, visor de gráficas temporales y demás *gadgets* que permitan al usuario controlar y visualizar el estado del invernadero.

## 1.3. ESTRUCTURA DE LA MEMORIA

La memoria está estructura en varios capítulos, donde se aborda en el primer capítulo, como se ha visto, una pequeña introducción y declaración de objetivos; y posteriormente se continúa con el resto de secciones que encierran el diseño, construcción y desarrollo del proyecto.

- **Capítulo 1: Introducción**. Ideas y necesidades que han dado lugar al desarrollo de este proyecto, así como las metas y objetivos fijados para ser alcanzados con la elaboración de este trabajo.
- **Capítulo 2: Estado del arte.** Recogida de información sobre el estado de los invernaderos, analizando su origen y estudiando los últimos avances en tareas de control y monitorización.
- **Capítulo 3: Descripción del hardware desarrollado.** Descripción de los materiales empleados para la construcción de la maqueta. Instalación y puesta en marcha de los sensores y actuadores, y desarrollo del panel de control del sistema.
- **Capítulo 4: Descripción del software desarrollado**. Descripción del código implementado para el control del microcontrolador, el programa en el equipo servidor y la aplicación para dispositivos móviles.
- **Capítulo 5: Conclusiones y líneas futuras.** Reflexiones sobre los logros alcanzados y posibles caminos a seguir a partir de esta idea desarrollada.
- **Glosario:** Breve descripción de algunos términos relevantes.
- **ANEXO I: Presupuesto.** Coste del material empleado para la construcción de la maqueta.
- **ANEXO II: Hojas de Características.** Documentación técnica de los componentes electrónicos empleados en la maqueta.
- **ANEXO III: Manual de Instalación.** Guía de instalación del software requerido para el funcionamiento del sistema.
- **ANEXO IV: Manual de usuario serverICRA.** Guía de uso para la aplicación servidor.
- **ANEXO V: Manual de usuario usersICRA.** Guía de uso para la aplicación de gestión de usuarios en el sistema.
- **ANEXO VI: Manual de usuario ICRApp.** Guía de uso para la aplicación móvil.
- **ANEXO VII: Montaje de ICRA.** Fotografías del proceso de montaje de la maqueta del invernadero.
- **Bibliografía.** Fuentes bibliográficas empleadas para este proyecto.

## 1.4. SIMBOLOGÍA

Las inclusiones de código en esta memoria utilizan cajetines para ello. En función del color del cajetín, el código de su interior hace referencia a diferentes lenguajes y partes del proyecto:

- Comandos empleados en la aplicación terminal de *Linux* (cajetín **violeta**):

```
sudo apt-get update
```

- Código *Arduino* (Microcontrolador) (cajetín **azul**):

```
void setup(){

    icra.Inicializa();
}
```

- Código *Android* (*ICRApp*) (cajetín **verde**):

```
public void onStart(){
  super.onStart();
  Log.d(tag,"In the onStart() event");
}
```

- Código *C++* (*serverICRA*) (cajetín **amarillo**):

```
int main(){
   …
   }
```

La inclusión en los cajetines de "…" refiere a la omisión de parte del código, con el fin de mostrar las líneas de código relevantes en cada bloque.

Para hacer referencia a acrónimos, aplicaciones, directorios, instrucciones de código… se emplea el formato *cursiva*; y con objetivo de resaltar palabras relevantes, o elementos importantes en ciertas tareas, se emplea el formato **negrita**. Por ejemplo:

"La captación de los valores de **temperatura** y **humedad** en *ICRA* se ha realizado con un único accesorio, el sensor **DHT22**."

# CAPÍTULO 2.
# ESTADO DEL ARTE

Este capítulo aborda un pequeño estudio sobre la evolución de los invernaderos. Comienza con una breve reseña histórica, y continua con la descripción de alguno de los sistemas actuales de monitorización y control de cultivos. Por último se estudia el uso de dispositivos móviles en tareas de monitorización remotas, destacando el empleo de sistemas operativos *Android*.

## 2.1. INVERNADEROS CONVENCIONALES

Un invernadero es toda aquella estructura cerrada y cubierta, dentro de la cual es posible obtener unas **condiciones artificiales de microclima**, y con ello cultivar plantas en condiciones óptimas y fuera de temporada.

La idea y la necesidad de este sistema se remota a 1850, donde la horticultura neerlandesa comenzó a utilizarlos para el cultivo de uvas (*Ver "Figura 2").* Se descubrió que el cultivo en invernaderos con calefacción y con el más alto nivel de cristal incrementaba el rendimiento. Las plantas crecían más rápidamente cuando se les daba más luz y cuando el entorno cálido era constante. Esto significa que, si no hubiera invernaderos, en los Países Bajos no se podrían explotar plantaciones solamente cultivables en países cálidos.

**Figura 2 : Invernaderos Reales de Laeken en Bruselas. Construidos durante los años 1874 y 1895**

Las tormentas de 1972 y 1973 fueron la razón de llevar a cabo investigaciones científicas técnicas y sistemáticas en la construcción de invernaderos. Conjuntamente con pioneros de la industria y comercio, se redactó la primera normativa para la construcción de invernaderos neerlandesa, NEN 3859.

En España, debido a las condiciones climáticas de la costa mediterránea, se desarrolló a finales de la década de los 70 una proliferación del cultivo en invernaderos, siendo las provincias de Alicante, Murcia, Almería y Granada las principales áreas de proliferación. Se notó un impacto mayor en la costa almeriense, donde casi toda su superficie de costa está cubierta por el conocido como "mar de plástico".

El clima terrestre es caótico y complejo. Se debe a una multiplicidad de factores en los que el hombre no tiene influencia sustancial alguna. Esto afecta de manera directa a los diferentes tipos de cultivos.

En las décadas siguientes, la agricultura deberá afrontar, por una parte, una demanda creciente en alimentos y materias primas básicas, y a la necesidad de utilizar los recursos sin causar degradación o agotamiento del ambiente. Las civilizaciones generalmente han prosperado durante los periodos de clima benigno, incluso muchas fueron incapaces de optimizar sus prácticas agrícolas para ayudar al control del sistema natural; por ello la historia documenta la caída de los sistemas socioeconómicos que no tuvieron capacidad para responder a los cambios del clima o en los recursos de agua y suelo.

Se tiene muchas ventajas al tener cultivos bajo invernadero, esto evita los cambios bruscos del clima como la variación de temperatura, la escasez o exceso de humedad. También se puede producir cultivos en las épocas del año más difíciles teniendo

cosechas fuera de temporal, sustituyendo el clima de otras regiones y alargando el ciclo del cultivo. Otra de las ventajas es el de obtener productos de mejor calidad y una mayor producción en la cosecha, y así incrementar la economía. (1)

Este incremento del valor de los productos permite que el agricultor pueda **invertir tecnológicamente en su explotación** mejorando la estructura del invernadero, los sistemas de riego localizado, los sistemas de gestión del clima, etc, que se reflejan posteriormente en una mejora de los rendimientos y de la calidad del producto final.

En los últimos años son muchos los agricultores que han iniciado la instalación de artilugios que permiten la automatización de la apertura de las ventilaciones, radiómetros que indican el grado de luminosidad en el interior del invernadero, instalación de equipos de calefacción, etc.

## 2.2. CONTROL Y MONITORIZACIÓN DE INVERNADEROS

Durante la década de los 70's se comienza a desarrollar un elemento fundamental en la rama de la electrónica y el control, el **microcontrolador** (*Ver "Figura 3").* Un elemento formado por un circuito integrado programable, capaz de ejecutar las órdenes grabadas en su memoria, diseñado para reducir el costo económico y el consumo de energía de un sistema en particular.

Desde su origen, se han perfeccionado y mejorado los distintos tipos de microcontroladores en el mercado. Este potencial desarrollado durante estos últimos años ha permitido la facilidad de integrar microcontroladores en todo tipo de **sistemas de control**, de forma que se pueda realizar una regulación del sistema por un módico precio y una gran fiabilidad y precisión. Jugando un papel fundamental como cerebro y controlador de señales en cualquier sistema de lazo cerrado, el microcontrolador permite recibir señales de entrada (sensores), interpretarlas, y tomar decisiones en sus salidas (actuadores) (*Ver Figura 4").* (2)

Este elemento es el pilar básico en el control ambiental, basado en manejar de forma adecuada todos aquellos sistemas instalados en el invernadero: **sistema de calefacción**, **sistema humidificador** y de **ventilación**, para mantener los niveles adecuados de temperatura, humedad relativa y calidad del aire. Con ello conseguir la mejor respuesta del cultivo y por tanto, mejoras en el rendimiento, precocidad, calidad del producto y calidad del cultivo.

Figura 3: Microcontrolador Arduino.

El usuario introduce las condiciones ambientales deseadas y el microcontrolador actúa sobre sus salidas para conseguirlas, corrigiendo sus decisiones gracias a la labor de los sensores, los cuales permiten cerrar el

lazo de control. De esta forma, puede crearse un **microclima con unas condiciones determinadas** de temperatura, humedad y calidad de aire en el invernadero.

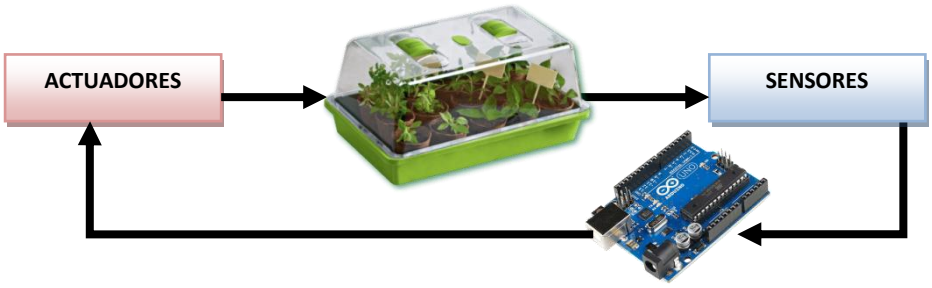



Figura 4: Controlador cerrando el lazo de control en un invernadero

Otra tarea importante que ocupa el controlador, además de establecer la relación entre entradas y salidas, es la de otorgar información al usuario sobre el estado de sus variables. La monitorización de variables permite una ventana de visualización al operario sobre la evolución y el estado de los parámetros del invernadero (*Ver "Figura 5").* Este proceso es de suma importancia, pues permite un vínculo entre el usuario y el sistema, de manera que el individuo pueda realizar una labor de supervisión y, en caso de requerirse, de actuación sobre el invernadero.

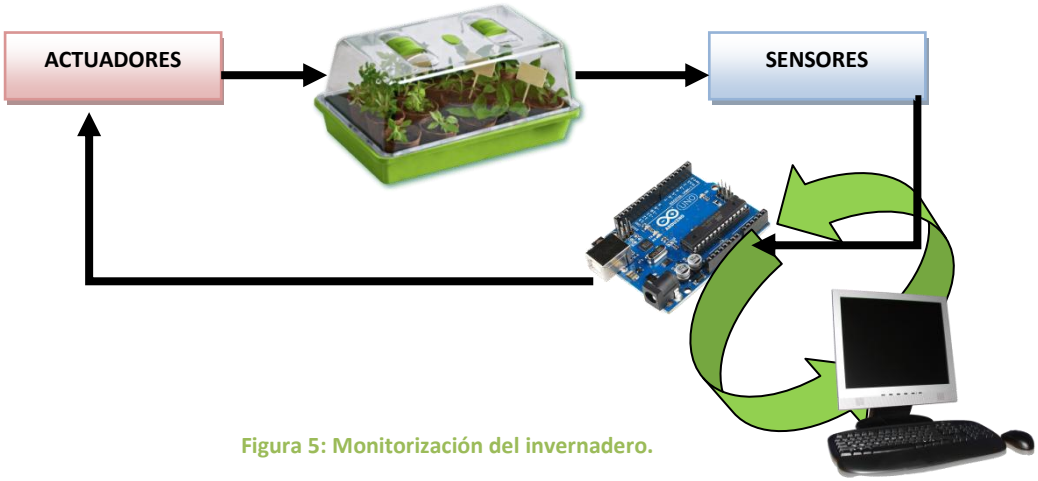



Figura 5: Monitorización del invernadero.

Mediante un sistema de adquisición de datos, un equipo informático obtiene información sobre el estado de las variables del sistema, comunicándose con el microcontrolador. Una correcta representación y una interfaz clara proporcionan una herramienta fundamental y determinante al usuario, el cual puede conocer el estado del sistema en todo momento. (3)

Se puede encontrar extensa documentación sobre invernaderos automatizados, desarrollados desde la primera década del siglo XXI hasta el momento actual. Ya son varias las empresas que brindan sus servicios al consumidor en materia de **automatizar invernaderos,** incorporando **centralitas en los cuadros de control** (con sus diferentes sondas de medición disponibles) necesarias para el control y gestión de los diferentes

elementos y equipos que regulan el ambiente del invernadero (*Ver "Figura 6").*

En lo referente a la actuación sobre los parámetros climáticos, cuentan con la instalación de **pantallas térmicas y de sombreo** aluminizadas, enrollamientos laterales, etc. Las cuales utilizan el sistema de tracción accionado por cremalleras, el cual puede motorizarse y automatizarse mediante un temporizador horario digital o mediante una sonda de radiación/iluminación (*watts/m*$^2$ o *Lux*). Los **sistemas de calefacción por aire y agua**, contribuyen a proporcionar una temperatura adecuada a los cultivos de alto requerimiento. Sumado a un **control de ventilación** y renovación de aire, que ofrece un equilibrio y calidad del ambiente en el interior, necesario para la correcta evolución de la plantación.

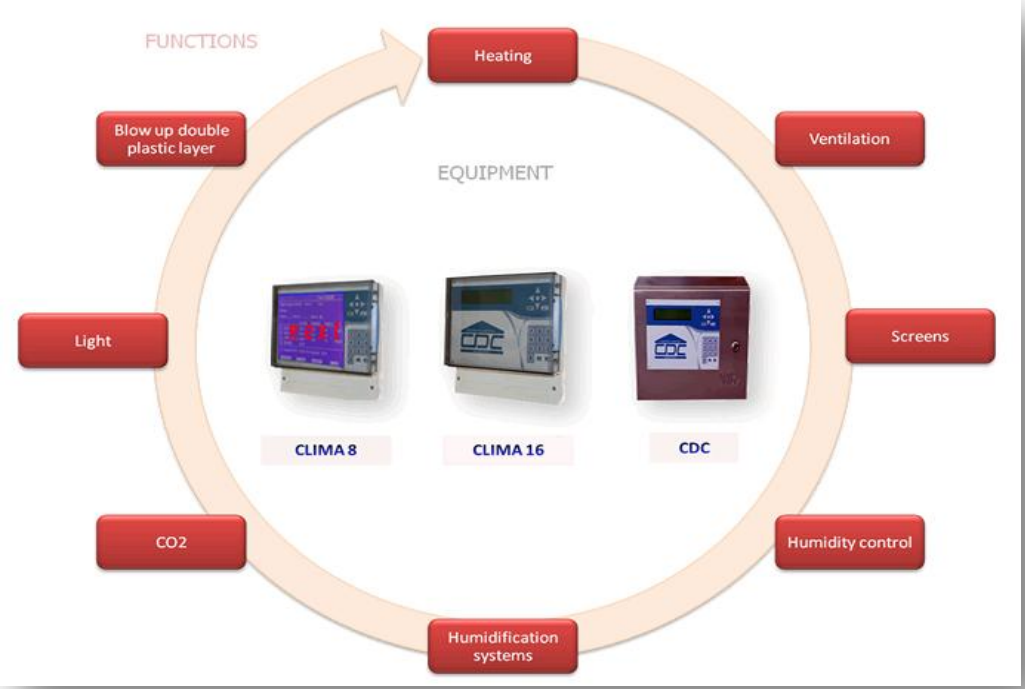



**Figura 6: Funciones de control climático en un invernadero.**
INTA. Empresa dedicada al control de fertirrigación de cultivos y control ambiental de invernaderos
www.inta.com.es

Se pueden encontrar diferentes bloques en el mercado atendiendo a las necesidades y la complejidad del sistema a estudiar, comenzando por **soluciones sencillas y robustas**, como sistema mostrado en la *"Figura 7".* Un equipo compacto, sencillo y fiable, destinado a aquellos agricultores que quieren disponer de un equipo avanzado pero muy fácil de manejar y a un coste muy competitivo. Pantalla grafica para facilitarle el manejo, con un menú muy simple y reducido para nuevos usuarios de equipos de control climático. Es ampliable tanto en entradas como en salidas y puede comunicar en red con otros equipos y dispositivos de la misma gama. Múltiples versiones configurables y central meteorológica compacta.

Y terminando por equipos de monitorización y supervisión **más complejos**, diseñados para cubrir los requerimientos de los invernaderos más sofisticados. Estos equipos permiten controlar, de manera inteligente y coordinada, parámetros vitales para el cultivo. Sumado a un potente **sistema de alarmas** que controlan en todo momento el estado del invernadero: temperatura, humedad del aire, nivel de $CO_2$, fallos de calefacción, sensores, suministro eléctrico, fallo de estación meteorológica... Y todas estas alarmas se pueden enviar **vía SMS**, para mantener al usuario en todo momento informado. Estos sistemas pueden configurarse en base a las necesidades del cliente, modificando parámetros de trabajo desde el terminal en la plantación. La evolución en el tiempo de estos parámetros queda registrada por programa y se puede visualizar de forma **gráfica** (*Ver "Figura 8"),* combinando a voluntad aquellos parámetros que resulten útiles en cada caso en los gráficos para poder planificar las estrategias de control climático. (4)

**Figura 7: Sistema de monitorización Clima 8**
INTA. www.inta.com.es


**Figura 8: Gráfico de evolución de parámetros. Software SYSCLIMA.**
INTA. www.inta.com.es

## 2.3. INTEGRACIÓN DE MÓVILES ÚLTIMA GENERACIÓN

La telefonía móvil está cambiando la sociedad actual de una forma tan significativa como lo ha hecho Internet. Esta revolución no ha hecho más que empezar, los nuevos terminales ofrecen unas capacidades similares a las de un ordenador personal, lo que permite que puedan ser utilizados para leer el correo o navegar por Internet. Pero a diferencia de un ordenador, un teléfono móvil siempre está en el bolsillo del usuario y esto permite un nuevo abanico de aplicaciones mucho más cercanas al usuario. De hecho, muchos autores coinciden en que el nuevo ordenador personal del siglo veintiuno será un terminal móvil. (5)

El lanzamiento de *Android* como nueva plataforma para el desarrollo de aplicaciones móviles ha causado una gran expectación y está teniendo una importante aceptación, tanto por los usuarios como por la industria. En la actualidad se está convirtiendo en la alternativa estándar frente a otras plataformas como *iPhone, Windows Phone o BlackBerry* (*Ver "Figura 9").* (6)

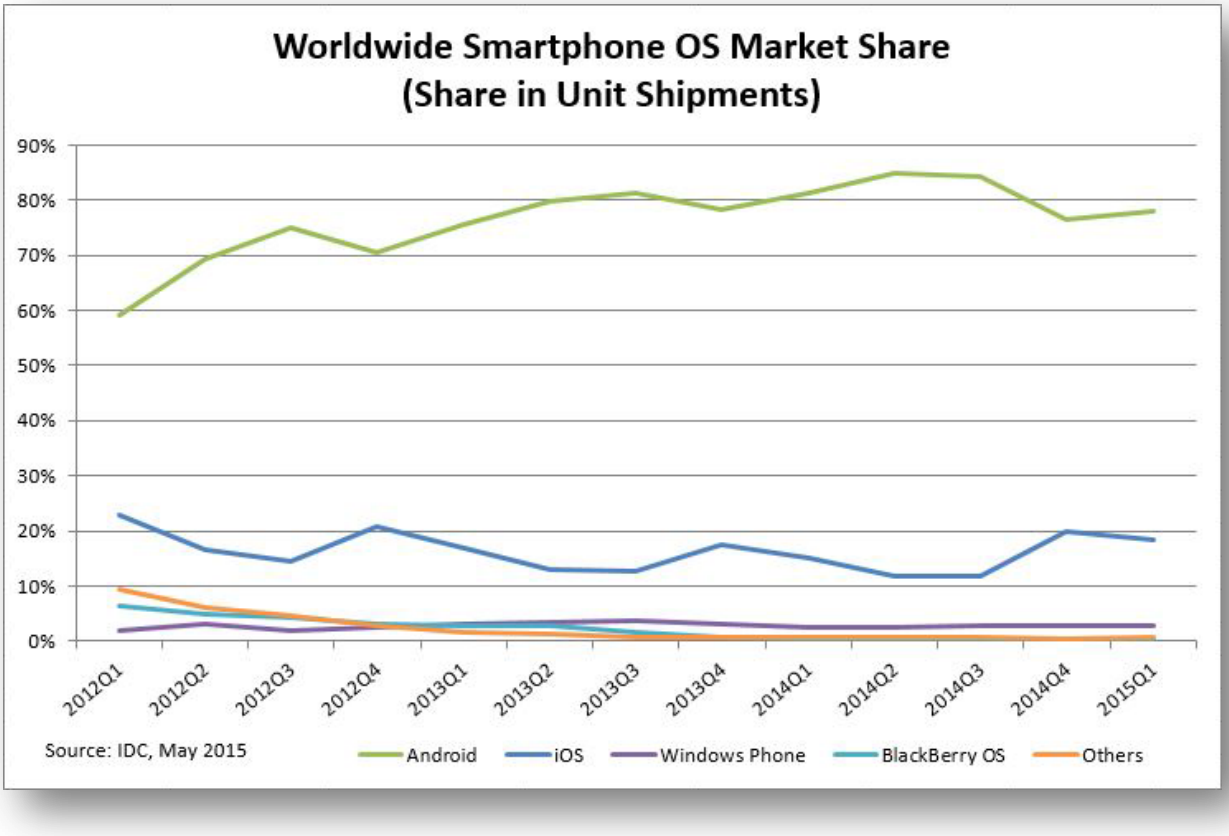



**Figura 9: Cuota de mercado en SO móviles**
*http://www.idc.com/prodserv/smartphone-os-market-share.jsp*

Varios proyectos han empleado estos recursos para desarrollar sistemas remotos de control y monitorización, capaces de montar bajo el sistema operativo *Android* una interfaz de mando sobre el sistema de cultivo (*Ver "Figura 10").*

Emplean para ello un equipo *PC* con conexión a Internet capaz de enviar por la red los datos recopilados por el microcontrolador, y a su vez permite escuchar instrucciones provenientes de los terminales *Android* de forma remota. Mediante el uso del protocolo *HTTP*, se establece un intercambio de mensajes cliente-servidor para recopilar toda la información necesaria en la aplicación móvil. Las órdenes dadas desde el terminal móvil son cargadas en el microcontrolador, elemento que regula los actuadores del cultivo. (7)

Cada vez es mayor el uso de sistemas descentralizados de control de invernaderos, como el sistema *UECS* (*Ubiquitous Environmental Control System*), que ofrece una solución barata, flexible y fiable capaz de monitorizar y controlar un recinto de cultivo desde un terminal móvil (*Ver "Figura 11").* (8)

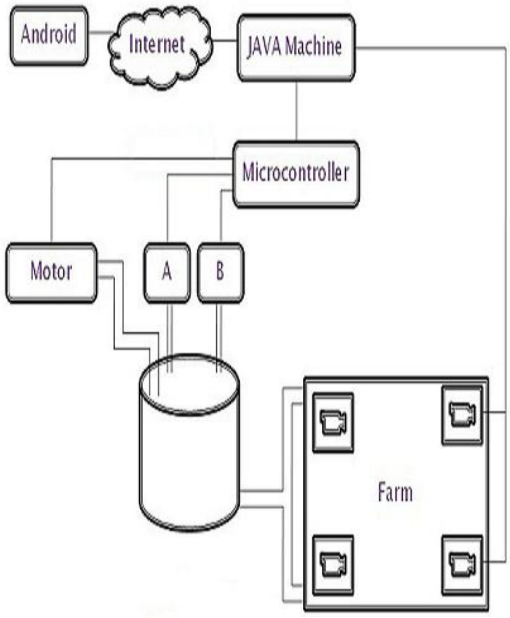



**Figura 10: Arquitectura del sistema**
*Remote Control and Automation of Agriculture Devices Using Android Technology*

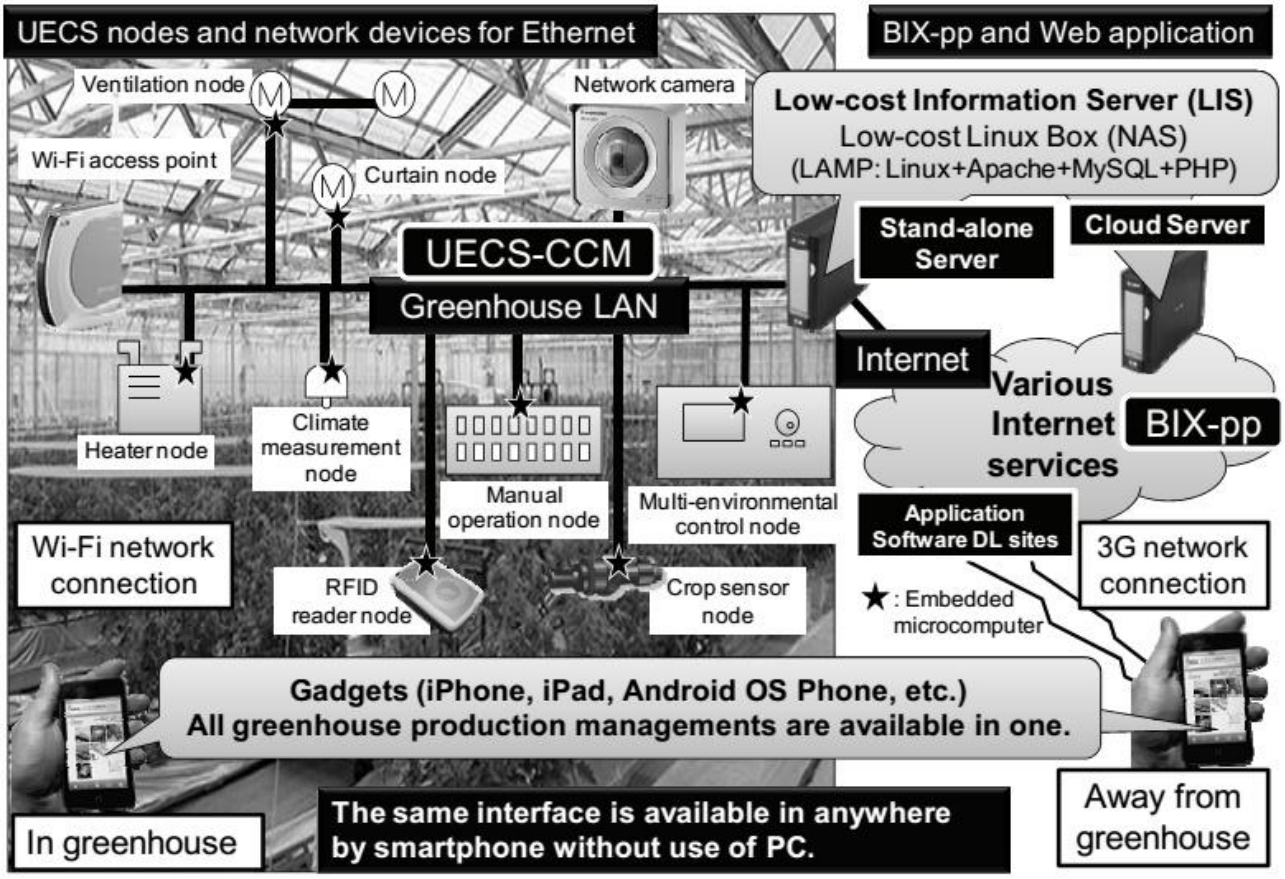



**Figura 11: Sistema de control de invernaderos empleando la plataforma UECS**
***A Gadget-Based Information Management System for Environmental Measurement and Control in Greenhouses***

Algunas de las principales características que han motivado el continuo desarrollado de este tipo de aplicaciones bajo soporte *Android* son (9):

- Plataforma realmente abierta. Es una plataforma de desarrollo libre basada en *Linux* y de código abierto. Una de sus grandes ventajas es que se puede usar y "customizar" el sistema sin pagar royalties.
- Portabilidad asegurada. Las aplicaciones finales son desarrolladas en *Java*, lo que nos asegura que podrán ser ejecutadas en una gran variedad de dispositivos, tanto presentes como futuros. Esto se consigue gracias al concepto de máquina virtual.
- Arquitectura basada en componentes inspirados en Internet. Por ejemplo, el diseño de la interfaz de usuario se hace en *XML*, lo que permite que una misma aplicación se ejecute en un móvil de pantalla reducida o en un netbook.

Estas propiedades permiten ampliar la arquitectura mostrada en la *Figura 5,* y obtener una plataforma descentralizada como la representada en la *Figura 12:*

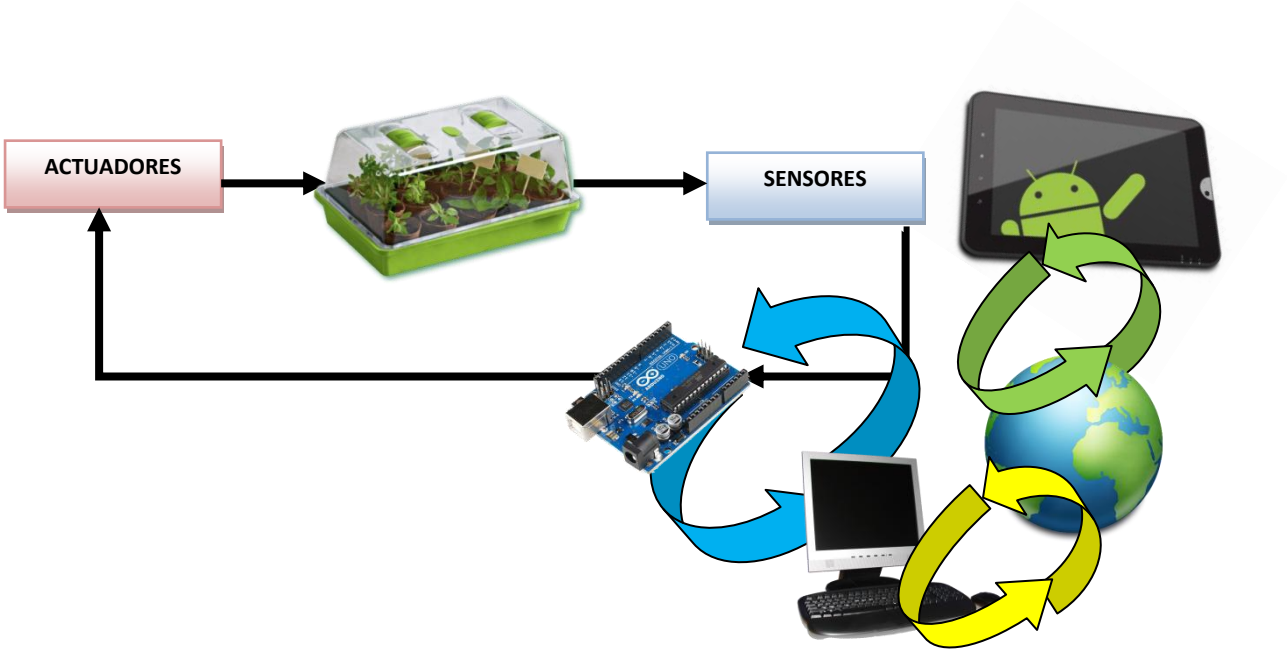



**Figura 12: Monitorización del invernadero. Comunicación entre dispositivos móviles, servidor y microcontrolador.**

# CAPÍTULO 3.
# DESCRIPCIÓN DEL HARDWARE DESARROLLADO

En este capítulo se describen los principales componentes empleados para la construcción de la maqueta, mostrando sus principales características así como su funcionalidad y su conexionado.

## 3.1. ESTRUCTURA DE LA MAQUETA

El invernadero *ICRA* está constituido por dos zonas bien diferenciadas (*Ver* "*Figura 13*"). El recinto interior, el cual alberga la plantación y crea un entorno adecuado para esta, y la zona de control y conexión, formada por el panel de control de usuario, la zona de conexionado y la tarjeta microcontroladora.[1]

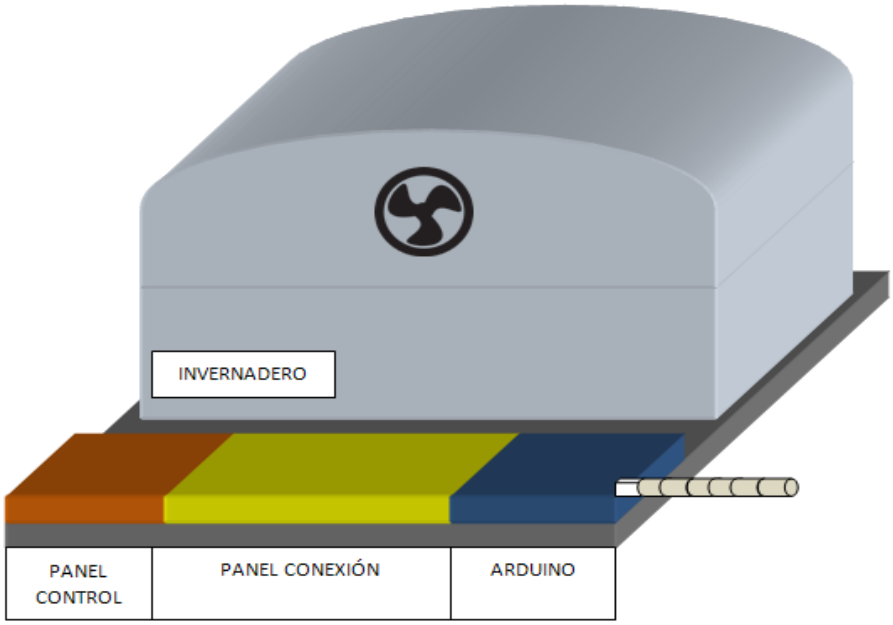



**Figura 13: Esquema representativo maqueta ICRA**

El recinto interior está formado por dos partes: una zona prismática inferior, y una cúpula cilíndrica superior. Las dimensiones de la zona inferior son de 240 mm de largo por 215 mm de ancho, con un alto de 100 mm. La cúpula posee una altura de 105 mm medida desde la parte alta de la zona inferior.

El material empleado para la construcción de la maqueta es el siguiente:

- Madera de samba para formar la estructura principal inferior.
- Alambre de 1mm de diámetro para formar la cúpula superior.
- Forro de plástico para confeccionar las paredes y techos.
- Fieltro, velcro, pegamento para madera, celofán, cinta adhesiva… y demás materiales de unión para realizar los empalmes necesarios entre las distintas partes.

[1] Ver "ANEXO I: Presupuesto" para obtener más información sobre los componentes de la maqueta.

Véase *Figura 14* para observar el proceso de montaje.[2]

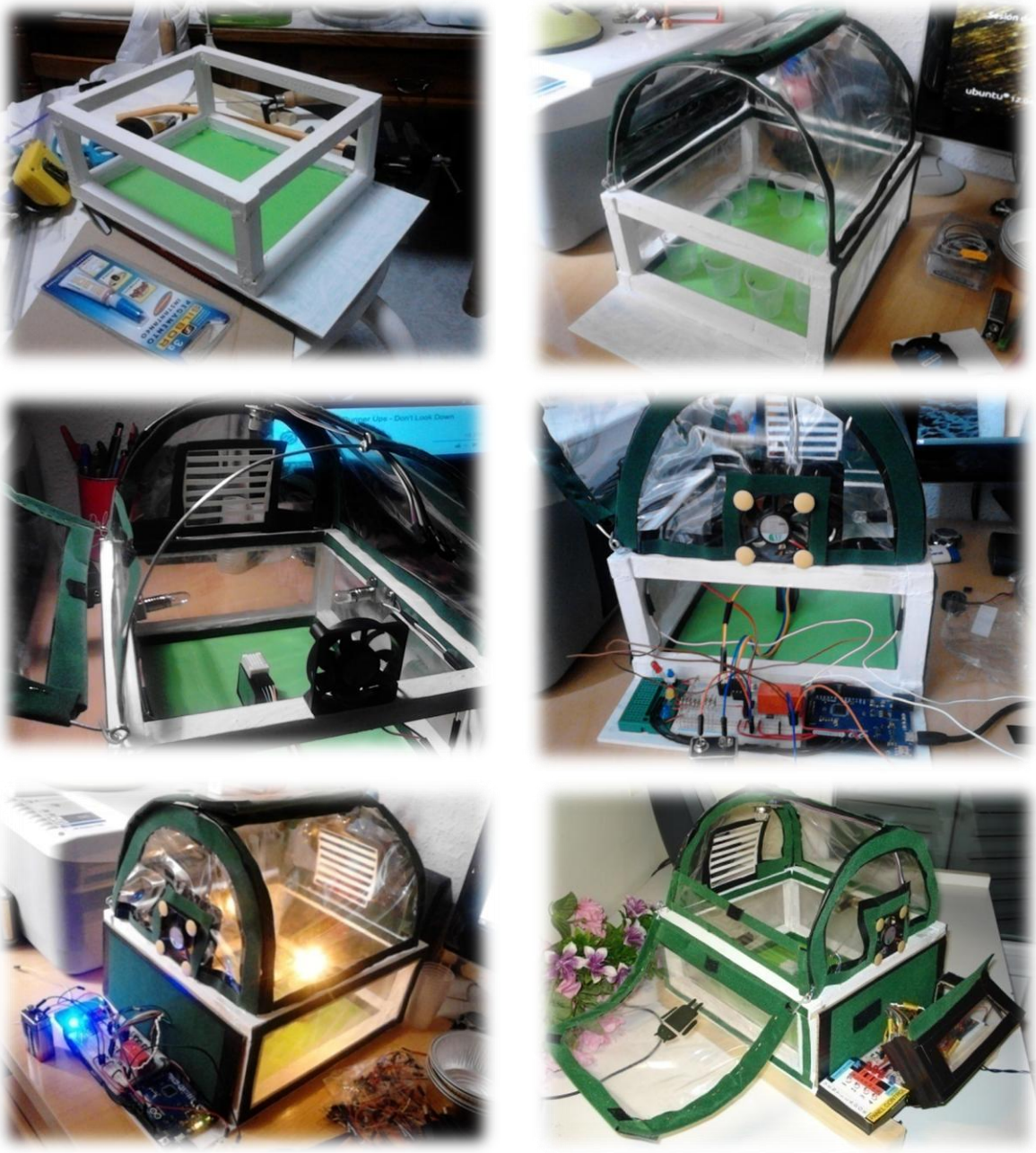


**Figura 14: Montaje de ICRA**

A continuación se va a pasar a describir cada uno de los componentes utilizados en la maqueta.

[2] Ver "*ANEXO VII: Montaje de ICRA"* para obtener más información acerca del montaje.

## 3.2. PLACA ARDUINO LEONARDO

El papel del microcontrolador en un proyecto de esta índole es fundamental. Su tarea recae en la responsabilidad de adquirir el estado de los sensores, para poder actuar sobre los periféricos de salida que controlan las variables del entorno, incluyendo una labor de comunicación con el exterior, emitiendo y recibiendo información con los usuarios.

***Arduino Leonardo*** recoge todas estas características, siendo una placa de bajo coste, con unas altas prestaciones técnicas, y de software/hardware libre, se adapta perfectamente a las necesidades de este proyecto (*Ver "Figura 15").*


**Figura 15: Placa Arduino Leonardo**

*Arduino* es una plataforma de hardware libre, basada en una placa con un microcontrolador y un entorno de desarrollo, diseñada para facilitar el uso de la electrónica en proyectos multidisciplinares.

El hardware consiste en una placa con un microcontrolador *Atmel AVR* y puertos de entrada/salida. Los microcontroladores más usados son el *Atmega168*, *Atmega328*, *Atmega1280*, *ATmega8* por su sencillez y bajo coste que permiten el desarrollo de múltiples diseños. Por otro lado el software consiste en un entorno de desarrollo que implementa el lenguaje de programación *Processing/Wiring* y el cargador de arranque que es ejecutado en la placa, empleando un puerto *micro USB* para comunicarse con el *PC*.

A continuación se muestran las principales características (*Ver "Tabla 2").*

**Tabla 2: Características Arduino Leonardo**

| | |
|---|---|
| Microcontroller | ATmega32u4 |
| Operating Voltage | 5V |
| Input Voltage (recommended) | 7-12V |
| Input Voltage (limits) | 6-20V |
| Digital I/O Pins | 20 |
| PWM Channels | 7 |
| Analog Input Channels | 12 |
| DC Current per I/O Pin | 40 mA |
| DC Current for 3.3V Pin | 50 mA |
| Flash Memory | 32 KB (ATmega32u4) of which 4 KB used by bootloader |
| SRAM | 2.5 KB (ATmega32u4) |
| EEPROM | 1 KB (ATmega32u4) |
| Clock Speed | 16 MHz |

Además, la placa dispone de interrupciones externas programables, mediante las cuales se pueden crear subrutinas de atención para determinadas situaciones (*Ver "Tabla 3").*

**Tabla 3: Pines Interrupciones Arduino**

| Board | int.0 | int.1 | int.2 | int.3 | int.4 | int.5 |
|---|---|---|---|---|---|---|
| Uno, Ethernet | 2 | 3 | | | | |
| Mega2560 | 2 | 3 | 21 | 20 | 19 | 18 |
| Leonardo | 3 | 2 | 0 | 1 | 7 | |

*Leonardo* ofrece hasta 5 interrupciones programables en los pines 0,1, 2, 3 y 7.

## 3.3. SENSORES Y ACTUADORES

Se presentan los diferentes periféricos instalados en la maqueta del invernadero.

### 3.3.1. SENSOR DE TEMPERATURA/HUMEDAD

La captación de los valores de **temperatura** y **humedad** en *ICRA* se ha realizado con un único accesorio, el sensor **DHT22**.

Los sensores *DHT11* o *DHT22* son unos pequeños dispositivos que permiten medir la temperatura y la humedad. A diferencia de otros sensores, éstos han de conectarse a pines digitales, ya que la señal de salida es digital al integrar un pequeño microcontrolador para hacer el tratamiento de la señal.

Los sensores de gama *DHT* se componen de un sensor capacitivo para medir la humedad y de un termistor, ambos sensores ya calibrados por lo que no es necesario añadir ningún circuito de tratamiento de señal. Esto es, sin duda, una ventaja ya que simplifican las conexiones a realizar en la placa. Además, como los *DHT* han sido calibrados en laboratorios, presentan una gran fiabilidad (*Ver "Figura 16")*

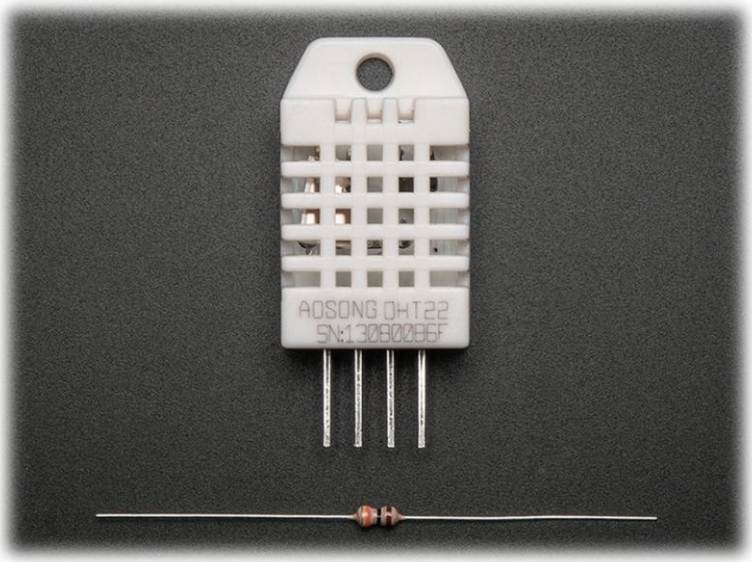



**Figura 16: Sensor DHT22**

Ambos sensores funcionan con ciclos de operación de duración determinada (1s en el caso del *DHT11* y 2s en el caso del *DHT22*). En este tiempo, el microcontrolador externo (*Arduino)*, y el microcontrolador que lleva integrado el sensor, se hablan entre sí de la siguiente manera:

- El microcontrolador (*Arduino*) inicia la comunicación.
- El sensor responde estableciendo un nivel bajo de 80us y un nivel alto de 80us.
- El sensor envía 5 bytes.
- Se produce el handshaking.

La siguiente tabla *(Ver "Tabla 4")* muestra las principales características de los dos modelos de sensores[3].

**Tabla 4: Características sensores DHT**

| Parámetro | DHT11 | DHT22 |
|---|---|---|
| Alimentación | 3Vdc ≤ Vcc ≤ 5Vdc | 3.3Vdc ≤ Vcc ≤ 6Vdc |
| Señal de Salida | Digital | Digital |
| Rango de medida de Temperatura | De 0 a 50 °C | De -40°C a 80 °C |
| Precisión Temperatura | ±2 °C | <±0.5 °C |
| Resolución Temperatura | 0.1°C | 0.1°C |
| Rango de medida Humedad | De 20% a 90% RH | De 0 a 100% RH |
| Precisión Humedad | 4% RH | 2% RH |
| Resolución Humedad | 1%RH | 0.1%RH |
| Tiempo de sensado | 1s | 2s |
| Tamaño | 12 x 15.5 x 5.5mm | 14 x 18 x 5.5mm |

En lo que se refiere al pinout, los pines del *DHT22* siguen el mismo orden *(Ver "Figura 17").* De izquierda a derecha se tiene:

1. VCC: Alimentación del sensor
2. Señal: Información de temperatura y humedad
3. NC: No Conexión
4. GND: Masa del circuito

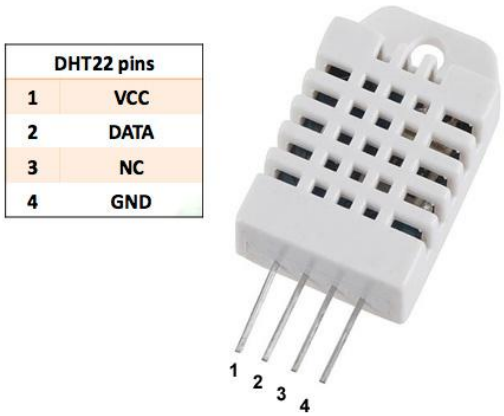



**Figura 17: Pinout DTH Sensor**

[3] Ver *"ANEXO II. Hojas de Características"* para obtener más información sobre los sensores.

### 3.3.2. SENSOR DE GAS INFLAMABLE

El módulo sensor de gas analógico **MQ2** se utiliza en la detección de fugas de gas de equipos en los mercados de consumo y la industria, este sensor es adecuado para la detección de GLP, i-butano, propano, metano, alcohol, hidrógeno, tiene una alta sensibilidad (ajustable mediante un potenciómetro.) y un tiempo de respuesta rápido.

La finalidad de este sensor en *ICRA* es la de **detectar sustancias nocivas y peligrosas** en el interior del invernadero, que puedan afectar a la seguridad y desarrollo del entorno, principalmente humo y sustancias inflamables. (*Ver "Figura 18").*

La siguiente tabla *(Ver "Tabla 5")* muestra las principales características del sensor:

**Tabla 5: Características sensor MQ2**

| Parámetro | MQ2 |
|---|---|
| Rango de detección | 300 a 1000 ppmm |
| Gas | Iso-butano |
| Sensibilidad R | 5 |
| Sensibilidad Resistencia | 1k a 20k |
| Tiempo de respuesta | 1s |
| Tiempo de recuperación | 30 años |
| Alimentación Resistencia | 31±3Ω |
| Corriente alimentación | 180mA |
| Voltaje alimentación | 5.0V±0.2V |
| Potencia alimentación | 900mW |
| Condiciones de funcionamiento | -20ºC a 55ºC, <95% HR, >21% $O_2$ |

El módulo dispone de cuatro pines de conexión. De izquierda a derecha el pineado es:

1. Voltaje de entrada: 5 ± 0.2V
2. DOUT: Señal digital de salida. LED se ilumina cuando se detecta gas.
3. AOUT: Señal analógica de salida, la salida es el valor analógico del voltaje detectado
4. GND: Polo negativo de alimentación.

**Figura 18: MQ2 Gas Sensor**

### 3.3.3. VENTILADOR

La ventilación del recinto es un aspecto fundamental. Realizar una renovación del aire en el interior, para conseguir unos valores adecuados de $O_2$ y $CO_2$ es crucial. El ventilador permite eliminar del invernadero sustancias nocivas en el ambiente, que puedan dañar el correcto crecimiento de la plantación.

Las características principales del aparato son las siguientes (*Ver "Figura 19"*):

- Tensión alimentación: 12VC
- Consumo de corriente: 0.06A
- Diámetro: 40mm


**Figura 19: Ventilador ICRA**

Mediante la implementación de un pequeño circuito de control, basado en un transistor bipolar *NPN 2N2222*, se han establecido cuatro modos de ventilación.

Haciendo uso de los canales *PWM* de *Arduino Leonardo*, puede regularse la inyección de corriente en la base del transistor, lo que permite regular el flujo de electrones entre colector y emisor, y conseguir de esta forma cuatro velocidades de ventilación en *ICRA*.[4]

Los dos principales objetivos de este periférico son:

- Ofrecer un **actuador** que pueda ser **controlado** a partir de la aplicación *Android*.
- Formar un sistema de **renovación de aire forzado**, realimentado con el sensor de gas inflamable **MQ2**, que permita la evacuación de sustancias nocivas y peligrosas para el crecimiento y evolución de las plantas.

[4] Ver punto *"3.5 Planos eléctricos"* para obtener más información sobre el conexionado.

### 3.3.4. SISTEMA DE CALEFACCIÓN

Una de las variables fundamentales en el crecimiento de la plantación es la **temperatura**. Los invernaderos permiten crear un rango de temperatura óptimo para la evolución de las plantas que albergan.

En *ICRA* se ha representado ese circuito de **calor** mediante dos bombillas incandescentes de casquillo pequeño, cada una con una potencia eléctrica de 2W.

Dado que la placa *Arduino* no está pensada para ser conectada a elementos que consuman una potencia media-alta, se ha utilizado un relé de conexión para llevar a cabo el control del encendido del sistema calorífico (*Ver "Figura 20").*

**Figura 20: Luminarias de calefacción y relé de encendido**

### 3.3.5. SISTEMA DE HUMIDIFICACIÓN

La segunda variable básica en el desarrollo de la plantación es la **humedad**. El recinto cubierto y cerrado del invernadero permite establecer valores de humedad ideales para el crecimiento.

La representación del estado de activación del sistema de humidificación se ha llevado a cabo mediante un **diodo led** (color azul), pensado para ser sustituido en un futuro por un actuador humidificador real.

### 3.3.6. ZUMBADOR

Un zumbador es un transductor electroacústico que produce un sonido o zumbido continuo o intermitente de un mismo tono.

Su construcción consta de dos elementos, un electroimán y una lámina metálica de acero. Cuando se acciona, la corriente pasa por la bobina del electroimán y produce un campo magnético variable que hace vibrar la lámina de acero sobre la armadura (*Ver "Figura 21").*

Mediante la conexión *PWM* de *Arduino*, pueden configurarse diferentes tonos de sonido, simplemente variando la frecuencia de la señal de entrada al periférico.

La misión de este componente es crear una **alarma acústica** que actúe en el caso de que se active alguna alarma del sistema.

Figura 21: Zumbador piezoeléctrico

## 3.4. PANEL DE CONTROL

*ICRA* dispone de un pequeño panel de control físico, situado en el lazo izquierdo de la maqueta (*Ver "Figura 22").* El cuadro de mandos consta de:

- Piloto de potencia del sistema.
- Pilotos de estado de alarmas del sistema.
- Pilotos de nivel de ventilación.
- Interruptor parada / rearme.

El **piloto de encendido** indica si el sistema está alimentando a los actuadores de la maqueta. En el caso de que el piloto este apagado, significará que los sistemas de ventilación, calefacción, humidificación y de alarmas están desconectados.

El panel contiene tres **pilotos de alarmas**, referidos a las variables de temperatura, humedad y gas. En caso de que alguna estas variables rebase algunos de los límites establecidos en el programa, se iluminará su piloto correspondiente. Además de emitir una señal sonora mediante el zumbador. Tras volver a situarse el valor de la variable dentro del rango establecido, el piloto se apagará y la alarma sonora cesará.

Cuatro pilotos indican el **nivel de ventilación** del recinto. En caso de la desconexión del ventilador, permanecerán todos apagados.

El interruptor de **parada / rearme** permite cortar el suministro eléctrico de los actuadores. Su estado se representa en el *piloto de potencia*. Si se pulsa de nuevo, se rearma el sistema, devolviendo los actuadores al estado que defina el programa en ese instante.

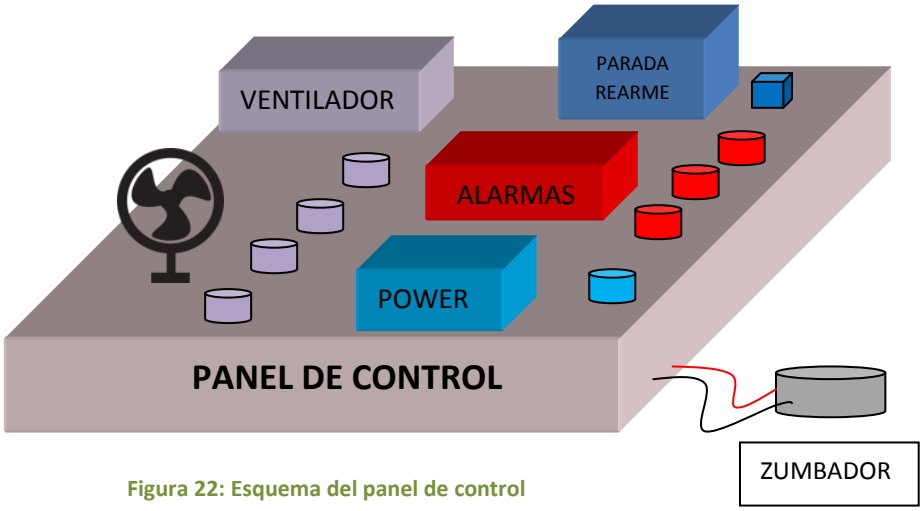
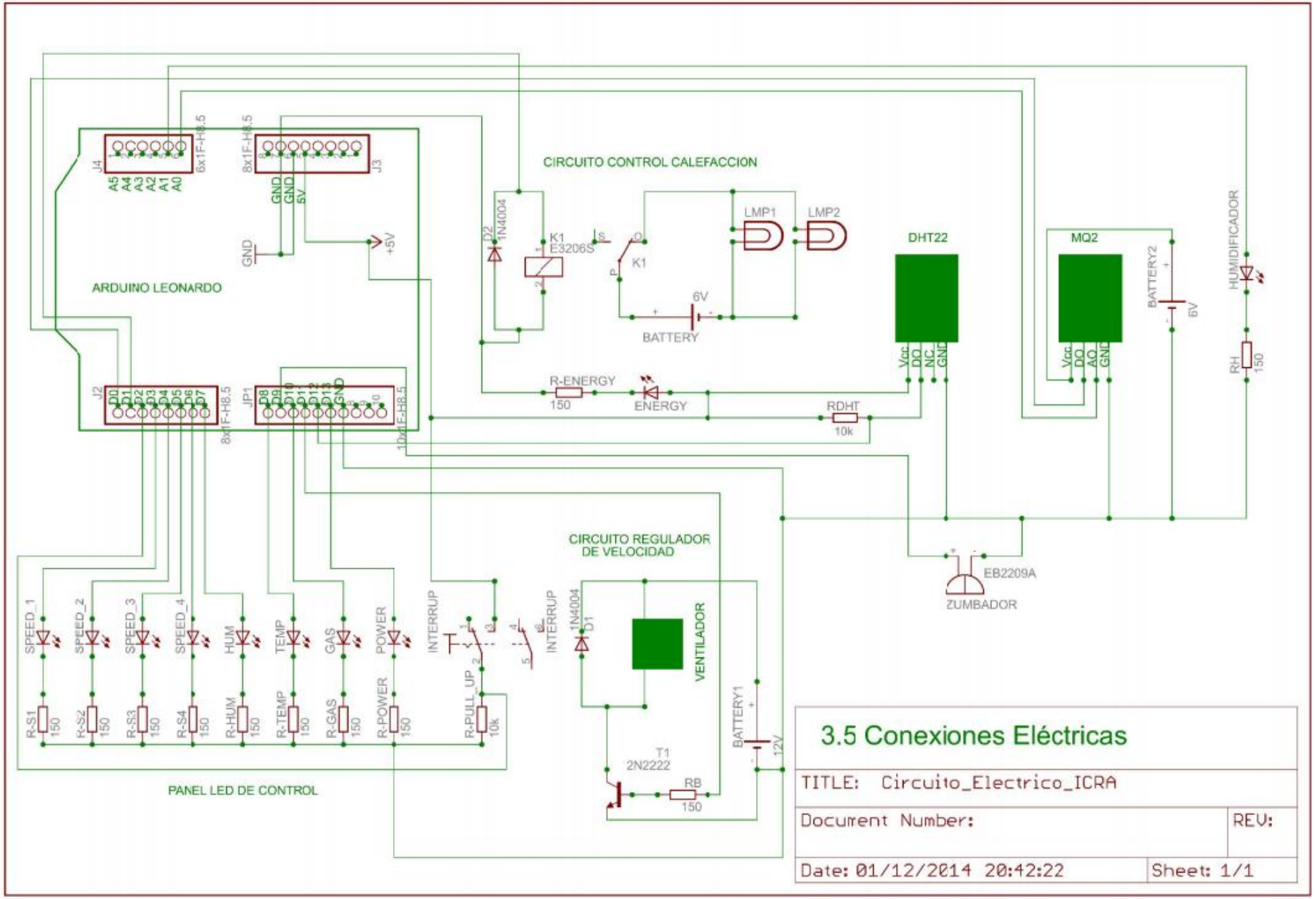



Figura 22: Esquema del panel de control


CIRCUITO CONTROL CALEFACCION
ARDUINO LEONARDO
J4
J3
J2
JP1
A5 A4 A3 A2 A1 A0
GND GND 5V
GND
+5V
D2
1N4004
K1
E3206S
K1
6V
BATTERY
LMP1
LMP2
DHT22
MQ2
BATTERY2
6V
HUMIDIFICADOR
RH
150
Vcc DO NC GND
Vcc DO AO GND
R-ENERGY
150
ENERGY
RDHT
10k
D0 D1 D2 D3 D4 D5 D6 D7
D8 D9 D10 D11 D12 D13 GND
8x1F-H8.5
10x1F-H8.5
CIRCUITO REGULADOR
DE VELOCIDAD
EB2209A
ZUMBADOR
SPEED_1
SPEED_2
SPEED_3
SPEED_4
HUM
TEMP
GAS
POWER
R-S1
R-S2
R-S3
R-S4
R-HUM
R-TEMP
R-GAS
R-POWER
150
INTERRUP
R-PULL_UP
10k
INTERRUP
1N4004
D1
VENTILADOR
BATTERY1
12V
T1
2N2222
RB
150
PANEL LED DE CONTROL
3.5 Conexiones Eléctricas
TITLE: Circuito_Electrico_ICRA
Document Number:
REV:
Date: 01/12/2014 20:42:22
Sheet: 1/1

# CAPÍTULO 4.
# DESCRIPCIÓN DEL SOFTWARE DESARROLLADO

En este capítulo se describe el diseño y desarrollo del software utilizado para el control del invernadero, abarcando desde el código de control de la maqueta hasta el código empleado en la aplicación móvil. El capítulo se estructura en tres puntos principales:

- Desarrollo del **código Arduino**. Se estudian los diferentes bloques de código que componen el sketch del microcontrolador, y se explican las clases implementadas para el control del recinto y la comunicación con el equipo servidor.
- Desarrollo del **servidor C++**. Se aborda la comunicación del equipo *PC* con el microprocesador *Arduino*, la implementación de los servicios *REST* empleados para la comunicación con los terminales móviles, y la conexión y acceso de la aplicación servidor a la base de datos *MySQL*.
- Desarrollo de la **aplicación Android**. Se analizan las diferentes *activities* que conforman la aplicación móvil, estudiando los *layouts* que las integran y su código *java* fundamental asociado.

## 4.1. DESARROLLO DEL CÓDIGO ARDUINO

Un programa diseñado para ejecutarse sobre una placa *Arduino* (un "*sketch*") siempre se compone, al menos, de tres secciones:

1) La sección de declaraciones de **variables globales**: ubicada directamente al principio del sketch.

2) La sección llamada **"void setup()"**: delimitada por llaves de apertura y cierre.

3) La sección llamada **"void loop()"**: delimitada por llaves de apertura y cierre.

La primera sección del *sketch* (que no tiene ningún tipo de símbolo delimitador de inicio o de final) está reservada para escribir, tal como su nombre indica, las diferentes declaraciones de variables que se necesiten.

En el interior de las otras dos secciones (es decir, dentro de sus llaves) deben de escribirse las instrucciones que se deseen ejecutar en la placa, teniendo en cuenta lo siguiente:

Las instrucciones escritas dentro de la sección "*void setup()*" se ejecutan una única vez, en el momento de encender (o resetear) la placa *Arduino*.

Las instrucciones escritas dentro de la sección "*void loop()*" se ejecutan justo después de las de la sección "*void setup()*" infinitas veces hasta que la placa se apague (o se resetee), es decir, el contenido de "*void loop()*" se ejecuta desde la 1ª instrucción hasta la última, para seguidamente volver a ejecutarse desde la 1ª instrucción hasta la última, para seguidamente ejecutarse desde la 1ª instrucción hasta la última, y así una y otra vez.

Por tanto, las instrucciones escritas en la sección "*void setup()*" normalmente sirven para realizar ciertas preconfiguraciones iniciales y las instrucciones del interior de "*void loop()*" son, de hecho, el programa en sí que está funcionando continuamente. (2)

Además de estas funciones básicas, pueden implementarse nuevas **funciones**, de forma que se puede encapsular tareas específicas que se necesiten utilizar de forma periódica en la función *loop().* Estas funciones se pueden crear directamente en el fichero principal, o pueden utilizarse **librerías** ya implementadas, las cuales ofrecen funciones ya programadas destinadas a tareas concretas (p.ej lectura de sensores, conversión de variables…).

Por ejemplo si se quiere realizar control de un servomotor no hay que programar una función que lleve a cabo las tareas de mando, ya que se dispone en *Internet* de librerías que incorporan varias funciones de control de servomotores.

Un nivel más allá de la encapsulación, es utilizar una **Programación Orientada a Objetos.**

El principal objetivo de la *POO*[5] es lograr un código más reutilizable, por lo que se mejora notablemente la eficiencia a la hora de programar.

Un **objeto** es una unidad que engloba en sí mismo datos y procedimientos necesarios para el tratamiento de esos datos. Cada objeto contiene datos y funciones, y un programa se construye como un conjunto de objetos, o incluso como un único objeto. Los objetos se agrupaban en clases, de la misma forma que en la realidad, existiendo muchas *mesas* distintas (la de mi cuarto, la del comedor, la del laboratorio, la del compañero o la mía) agrupamos todos esos objetos reales en un concepto más abstracto denominado *mesa* (10). De tal forma que, se puede decir, que un objeto específico es una realización o instancia de una determinada **clase**.

En base a este estilo de programación se ha implementado el código de control de la placa, de forma que se puedan crear objetos que hagan referencia a partes o componentes del invernadero, englobando sus atributos y métodos de control. La siguiente imagen identifica los diferentes bloques del programa (*Ver "Figura 23").*

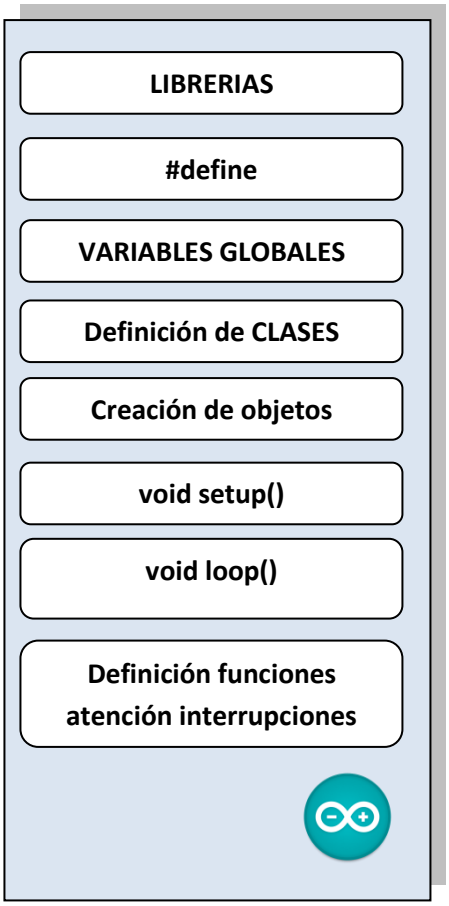



Figura 23: Estructura del código Arduino

El desarrollo y programación del código se ha llevado a cabo mediante el software *Arduino IDE 1.0.6* (*Ver "Figura 24").* Actualmente *Arduino* ha lanzado la versión *beta 1.6.4*. El código desarrollado para este proyecto no se ha testeado con tal versión, y por lo tanto no se garantiza su correcto funcionamiento.

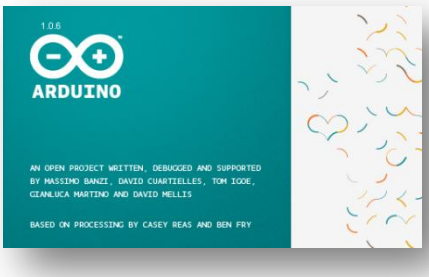



Figura 24: Arduino IDE 1.0.6

[5] La programación orientada a objetos (POO) se fue convirtiendo en el estilo de programación dominante a mediados de los años ochenta, en gran parte debido a la influencia de *C++*, una extensión del lenguaje de programación C.

### 4.1.1. ESTRUCTURA

A continuación se estudian los diferentes bloques que constituyen el programa de la placa *Arduino*.

#### 4.1.1.1. LIBRERÍAS

Inserción de ficheros externos que contengan funciones que vayan a utilizarse en el código principal.

Para este proyecto solo ha sido necesario incluir una librería externa: `#include <DHT.h>`. La librería ***DHT.h***[6] contiene las funciones necesarias para la adquisición de valores del *sensor DHT*.

#### 4.1.1.2. #DEFINE

Permite establecer variables constantes de forma que, al compilar el fichero el microprocesador, se sustituyan las referencias a la variable *#define* por el valor que tengan asociado.

A continuación se explican los *#define* creados en el programa.

```
#define T0 20 //Cota inferior rango temperatura plantacion
#define T1 22 //Cota superior rango temperatura plantacion
#define TEMP_MAX 24 //Valor maximo de temperatura
#define TEMP_MIN 14 //Valor minimo de temperatura
#define H0 55 //Cota inferior rango humedad plantacion
#define H1 60 //Cota superior rango humedad plantacion
#define HUM_MAX 70 //Valor maximo de humedad
#define HUM_MIN 35 //Valor minimo de humedad
#define CONTAMINACION 50 //Valor limite de pureza del aire
```

```
//Pines Arduino
#define LED_POWER 13//Panel de control
#define LED_TEMP 8
#define LED_HUM 7
#define LED_GAS 10
#define SWITCH 2
#define LED_SPEED_1 3//Panel velocidad ventilador
#define LED_SPEED_2 4
#define LED_SPEED_3 5
#define LED_SPEED_4 6
#define ZUMB 9//Zumbador Alarma
#define VENT 11//Ventilador
#define SONDA 12//Sensor Temperatura-Humedad
#define LAMP 1//Sistema Calefaccion
#define HUM 19//Sistema Humidificaccion
#define GAS A0//Sensor analogico gas
#define INTERRUP_GAS 0//Pin interrupcion sensor gas
```

[6] El fichero de librería DHT se incluye en el CD adjunto a este documento.

El primer bloque de código hace referencia a los valores iniciales de rangos de variables, tanto de trabajo de actuadores como de alarmas, esto permite a *ICRA* poder trabajar de forma independiente a la conexión servidor, realizando el control del recinto con los valores aquí fijados. Una vez conectado un usuario *Android*, puede modificar los rangos en base a sus necesidades.

El segundo bloque de código refiere a los pines de conexión de la placa *Arduino*, definiendo a que pin de la tarjeta está conectado cada periférico. En caso de realizar alguna modificación en el conexionado, en el código únicamente ha de cambiarse el valor del pin en la variable oportuna.

#### 4.1.1.3. VARIABLES GLOBALES

Encierra variables de ámbito global, es decir, variables que tienen visibilidad desde cualquier parte del programa. Son, por lo general, variables que hacen referencia a **estados del sistema o situaciones** que afectan al funcionamiento general del programa.

En el código se tiene:

```
boolean SwAndroid=false;//INTERRUPTOR PARADA ANDROID
boolean pauseAndroid=false;//PARADA ANDROID
boolean pauseSwitch=false;//PARADA INTERRUPTOR MANUAL
boolean PAUSE=false;//ESTADO PAUSA SISTEMA
boolean DANGER=false;//ESTADO PELIGRO AIRE
int aux_vent;//Variable aux velocidad ventilador
```

Donde, como se ha mencionado, son variables que intervienen en el ritmo principal del ciclo, como es el estado de *peligro* o el estado de *parada.*

#### 4.1.1.4. DEFINICIÓN DE CLASES

Este bloque encierra las implementaciones de las clases que van a usarse en el código. Tras su declaración, se pueden crear instancias de estas (*objetos*) en el programa principal para su utilización, incluso pueden crearse instancias dentro de otras clases, aumentando las posibilidades y opciones de estas.

Se procede a estudiar una clase creada en *ICRA* para entender mejor este concepto, por ejemplo, una clase sencilla como *Calefaccion*:

```
class Calefaccion{
  private:
    int pin;
    boolean estado;
  public:
    Calefaccion(int );
    void SetCalefaccion(boolean );//ON/OFF Lamparas
    void Monitor();//Salida por pantalla
    void ToServer();//Comunicacion con Server
      };
```

Se establecen los atributos de la clase como *privados* y los métodos de esta como *públicos*, de forma que puedan ser invocados libremente.

A continuación se analizan los dos atributos de la clase:

```
int pin;
boolean estado;
```

El atributo tipo entero *pin* almacena la patilla de conexión del sistema de calefacción a la placa *Arduino*. El atributo de tipo bit *estado*, representa la situación de encendido o apagado del sistema de calefacción.

Seguido a los atributos, se declaran las cabeceras de los métodos que contendrá la clase:

```
Calefaccion(int );
void SetCalefaccion(boolean );//ON/OFF Lamparas
void Monitor();//Salida por pantalla
void ToServer();//Comunicacion con Server
```

En este caso, la clase dispondrá de cuatro métodos o funciones propias.

El método con el mismo nombre que la clase, *Calefaccion(int )*, hace referencia al *constructor* de la clase; función que se invoca nada más crear una instancia de la clase, es decir, un *objeto* de ella. Más adelante se analizará el contenido de los métodos.

El siguiente método *SetCalefaccion* es de tipo *void*, por lo que no retornará ningún valor. Como se deduce de su nombre, hace referencia a la activación (y desactivación) del sistema de calefacción.

Los dos siguientes métodos, *Monitor()* y *ToServer()*, contienen el código necesario para imprimir el estado y situación del objeto a través del puerto serie.[7]

El método *Monitor()* está previsto para la representación de los valores a través de la herramienta *Monitor Serie*, disponible en la interfaz de programación de *Arduino* para *PC*.

[7] Las placas *Arduino* disponen de una unidad UART que operan a nivel TTL 0V / 5V, por lo que son directamente compatibles con la conexión USB.

*ToServer()* está destinado a la comunicación con la aplicación *serverICRA*, ya que incluye las cabeceras necesarias para el protocolo de comunicación.

Tras la definición de la clase, se implementa el contenido de sus métodos. Sería posible realizar la programación de cada método tras la declaración de su cabecera, pero para mejorar la eficiencia del código se hace a posteriori. Sólo en casos en los que el contenido de la función sea breve y conciso, se implementa tras su declaración.

Se analiza ahora el contenido de los métodos de la clase:

```
Calefaccion::Calefaccion(int p){
  pin=p;
  pinMode(pin,OUTPUT);
  estado=false;
}
```

El constructor de la clase *Calefaccion* recibe una variable tipo entera, que contiene el pin de conexión del periférico, y lo asocia al atributo *pin* de la clase. Acto seguido, define este pin como salida, *OUTPUT*, y por último inicializa el atributo *estado* a nivel bajo, *false*.

```
void Calefaccion::SetCalefaccion(boolean a){

    if(a)
      digitalWrite(pin,HIGH);
    else
      digitalWrite(pin,LOW);
    estado=a;
  }
```

El método *SetCalefaccion* recibe una variable tipo booleana, a través de la cual determina la activación del pin conectado al periférico. Tras ello almacena en el atributo *estado* el estado de activación del pin.

```
void Calefaccion::Monitor(){

  Serial.print("SISTEMA CALEFACCION: ");
  Serial.println(estado);
}

void Calefaccion::ToServer(){

  Serial.print("C");
  Serial.print(estado);
}
```

Los métodos *Monitor* y *ToServer* emplean las funciones *Serial.print*, a través de las cuales se realiza la comunicación con el *PC*. El primer método imprime la frase "SISTEMA CALEFACCIÓN: ", y seguido el valor del atributo *estado*, para cerrar con un final de carro (debido al uso de *.println*).

El segundo método envía la letra "C" seguido del valor del atributo *estado*, de forma que, si el sistema de calefacción está apagado, el mensaje que se envía es: "*C0"*, donde la letra "**C**" indica que es un mensaje proveniente de un objeto de la clase ***C**alefaccion*, y la cifra contigua representa el **estado** del periférico.

La "*Figura 25*" muestra las clases de las que se compone un invernadero *ICRA*, donde la clase *Invernadero* está formada por una **composición** del resto de clases[8]. Dicho de otra forma, se pude decir que para poder tener un invernadero *ICRA* es necesario que este disponga de un *sensor de temperatura/humedad*, un *detector de humo*, una *ventilación*, un *sistema de calefacción*, un *sistema de humidificación* y un *panel de control*. Los atributos y métodos de la clase *Invernadero* se detallan en el siguiente punto del capítulo: *"4.1.2 La clase Invernadero".*

[8] Ver *el proyecto de Arduino, disponible en el CD de la memoria, para obtener más información sobre el código.*

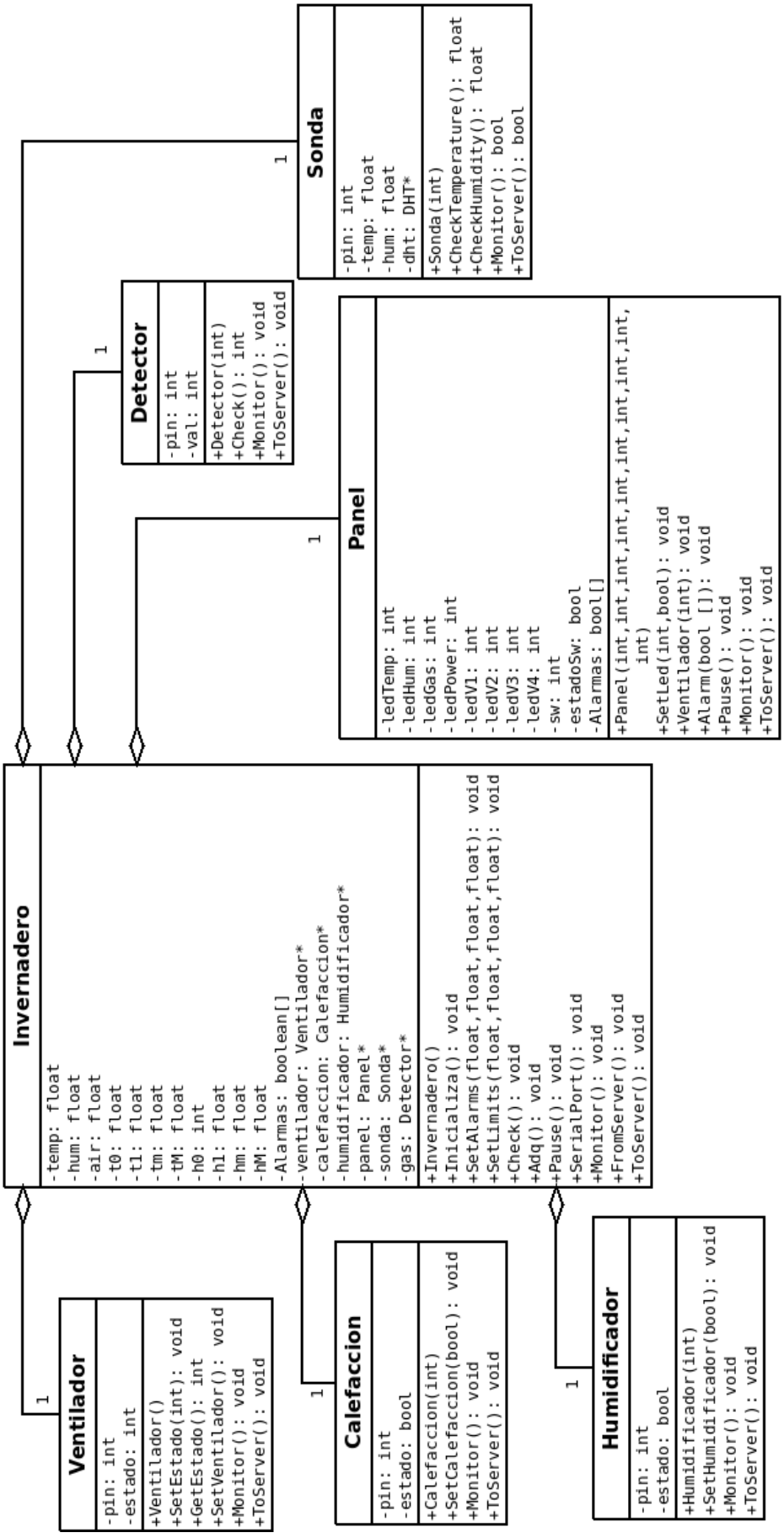



Figura 25: Diagrama de clases UML en Arduino

#### 4.1.1.5. CREACIÓN DE OBJETOS

En esta sección del código se crean las instancias de las clases, y por tanto se definen los *objetos globales* que se utilizarán en el programa.

Una vez definidas todas las clases, lo único que ha de hacerse es crear un invernadero que controlar.

```
Invernadero icra;
```

La línea de código anterior crea un objeto llamado *icra* de la clase *Invernadero*, *objeto* encargado de controlar el invernadero real.

#### 4.1.1.6. VOID SETUP()

Como se ha mencionado previamente, el contenido de esta función se ejecuta una única vez, y tiene lugar en el arranque de la placa.

Las instrucciones que se escriben aquí suelen hacer referencia a comandos de inicialización (definición de entradas/salidas, definición de valores iniciales, establecer conexiones…).

```
void setup(){
  icra.Inicializa();
      }
```

La línea anterior ejecuta el método *Inicializa( )* del objeto *icra*, el cual encapsula el código necesario para la puesta en marcha de todos los componentes del invernadero[9].

[9] Véase el punto *"4.2.2 Método Inicializa"* para más información

#### 4.1.1.7. VOID LOOP()

La función *void loop()* es el bucle principal del programa y ejecuta su contenido de forma cíclica. Se pasa a ver su contenido:

```
void loop(){

  icra.Adq();//Refresh sensores

  if(pauseSwitch||pauseAndroid)
    incra.Pause();//Paro
  else
    icra.Check();//Gestion actuadores

  icra.SerialPort();//Comunicacion serie

  delay(1000);
}
```

Gracias a la creación de clases y funciones, solamente es necesario escribir unas pocas líneas de código en la función principal de la placa.

La primera acción que toma el invernadero es adquirir el valor de sus sensores utilizando el método *Adq()*. Seguido, se evalúa si el sistema se encuentra en estado de pausa (ya sea por interruptor físico o por software) y en caso afirmativo se detiene la gestión de los actuadores y se llevan a un estado de desconexión.

El método *SerialPort()* gestiona la comunicación de la placa a través de puerto serie.

La última función ejecutada, *delay(1000)*, realiza una pausa de 1seg en el ciclo de trabajo del microprocesador, creando una situación de *sleep*[10], de esta forma el código se ejecuta cada segundo. Dada la índole de este proyecto, no es necesario utilizar la capacidad máxima de trabajo de *Arduino*, ya que se tratan variables lentas, que no requieren una frecuencia de muestreo elevada.

[10] "sleep" es un comando de la familia de los Sistemas Operativos Unix que permite suspender (bloquear) la ejecución actual por un intervalo de tiempo determinado.

#### 4.1.1.8. DEFINICIÓN DE INTERRUPCIONES

En la parte final del código se definen las funciones de tratamiento de interrupciones. Inmediatamente después de detectar una interrupción en el pin configurado, se invoca la función asociada al número de la interrupción[11] y una vez que se ejecuta su contenido, el programa retorna al punto en el que se lanzó la interrupción.

El siguiente código muestra las dos interrupciones programadas en *ICRA*.

```
void danger(){//Rutina atencion GAS

  DANGER=true;//Estado de PELIGO
  analogWrite(VENT,255);//ON inmedato de Ventilacion MAX
}

void pause(){//Rutina atencion SWITCH

    PAUSE=pauseSwitch=true;
    digitalWrite(LAMP,LOW);//OFF inmediato de actuadores de potencia
    digitalWrite(VENT,LOW);
    digitalWrite(HUM,LOW);
}
```

La función *void danger()* se encarga del tratamiento de peligro en el invernadero por sustancias peligrosas en el ambiente. Para ello acciona la máxima potencia de extracción y activa el estado de emergencia. Esta interrupción es lanzada por el sensor de gas **MQ2**.

*void pause()* atiende la solicitud de parada del sistema y se encarga de desconectar los equipos de potencia y activar el estado de parada. Dicha interrupción es activada por el **interruptor de parada** en el panel de control del invernadero.

[11] Véase el punto *4.1.2.1 Método Inicializa* para ver la asignación de interrupciones

### 4.1.2. LA CLASE INVERNADERO

Como se ha visto en el punto anterior, "*4.1.1.5 Creación de objetos*", todo el control y monitorización del invernadero real se lleva a cabo mediante el *objeto icra*, el cual es una instancia de la clase denominada *Invernadero()*, la cual contiene instancias de las otras clases definidas.

Viendo el diagrama de la *Figura 25* se aprecia rápidamente como la clase *Invernadero* reúne mayor complejidad que el resto de clases implementadas, debido a que es la clase con mayor carga, ya que ocupa la gestión de todos los periféricos y la monitorización. Se comentan brevemente sus atributos:

Los atributos *temp, hum, air* almacenan el valor de temperatura, humedad y estado del aire en el interior del recinto, actuando como contenedores del valor devuelto por los sensores del invernadero.

*t0, t1, h0, h1* definen los rangos de temperatura y humedad en el invernadero, determinando cotas máximas y mínimas de zonas de trabajo *(Ver "Figura 26").*

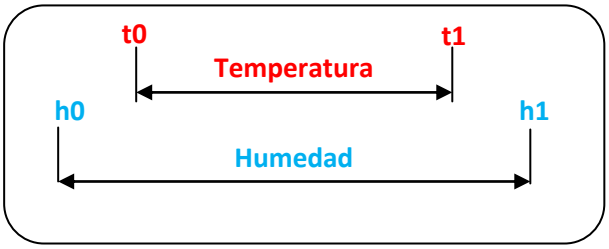



**Figura 26: Rangos de trabajo**

Por lo que, por ejemplo, si se establecen los siguientes valores:

- *t0: 20.50*
- *t1: 22.00*
- *h0: 55.80*
- *h1: 60.00*

El *sistema de calefacción* entrará en activación si la *temperatura* cae por debajo de *20.50ºC* en el interior, y se mantendrá activo hasta superar los *22ºC*.

El *humidificador* se activará si la *humedad* relativa desciende de *55.80%*, y no dejará de estar en funcionamiento hasta alcanzar el *60%* de *humedad*.

Los atributos *tm, tM, hm, hM* establecen los valores límite de *temperatura* y *humedad*, es decir, determinan aquellos valores a partir de los cuales las *alarmas* del sistema se activarán.

El vector booleano *alarmas[3]* contiene el estado de activación de las tres *alarmas* del sistema: *temperatura, humedad y aire*.

Los últimos atributos hacen referencia a los periféricos, y son *punteros* que apuntarán a las direcciones de memoria de los *objetos* que se creen en el constructor de la clase.

#### 4.1.2.1. MÉTODO INICIALIZA

La mayor parte del código del siguiente método se basa en realizar una prueba de encendido y apagado de los dispositivos conectados a la placa, pero hay algunas lineas que resaltar:

```
void Invernadero::Inicializa(){

  Serial.begin(9600);

  …

  attachInterrupt(2,danger, FALLING);//Rutina atencion GAS
  attachInterrupt(1,pause, HIGH);//Rutina atencion SWITCH
}
```

Este método se encarga de inicializar el puerto serie de comunicación en *9600 baudios*, además de crear instancia de dos interrupciones:

- `attachInterrupt(2,danger, FALLING);` Establece una interrupción en el pin número 0 de *Arduino*[12], activado por caida de valor, lanzando la función *danger()* para atender la interrupción. Dicha interrupción, atiende la situación de **atmósfera peligrosa** en el recinto interior.
- `attachInterrupt(1,pause, HIGH);` Establece una interrupción en el pin número 2 de *Arduino*, activado por valor alto, lanzando la función *pause()* para atender la interrupción. Dicha interrupción atiende la situación de **parada** solicitada por la activación del interruptor.

[12] Véase el punto *"3.2 Arduino Leonardo"* para observar la relación de pines e interrupciones.

#### 4.1.2.2. MÉTODOS SETALARMAS & SETLIMITS

Los métodos *SetAlarmas* y *SetLimits* simplemente insertan el valor recibido en sus parámetros de función en los atributos correspondientes del invernadero, reajustando el rango de trabajo a los nuevos valores.

#### 4.1.2.3. MÉTODO CHECK

El método *Check()* encierra el **control de actuadores y alarmas**, y emplea los valores recogidos por los sensores para regular los actuadores. A continuación se muestra el diagrama de flujo del método *(Ver "Figura 27").*

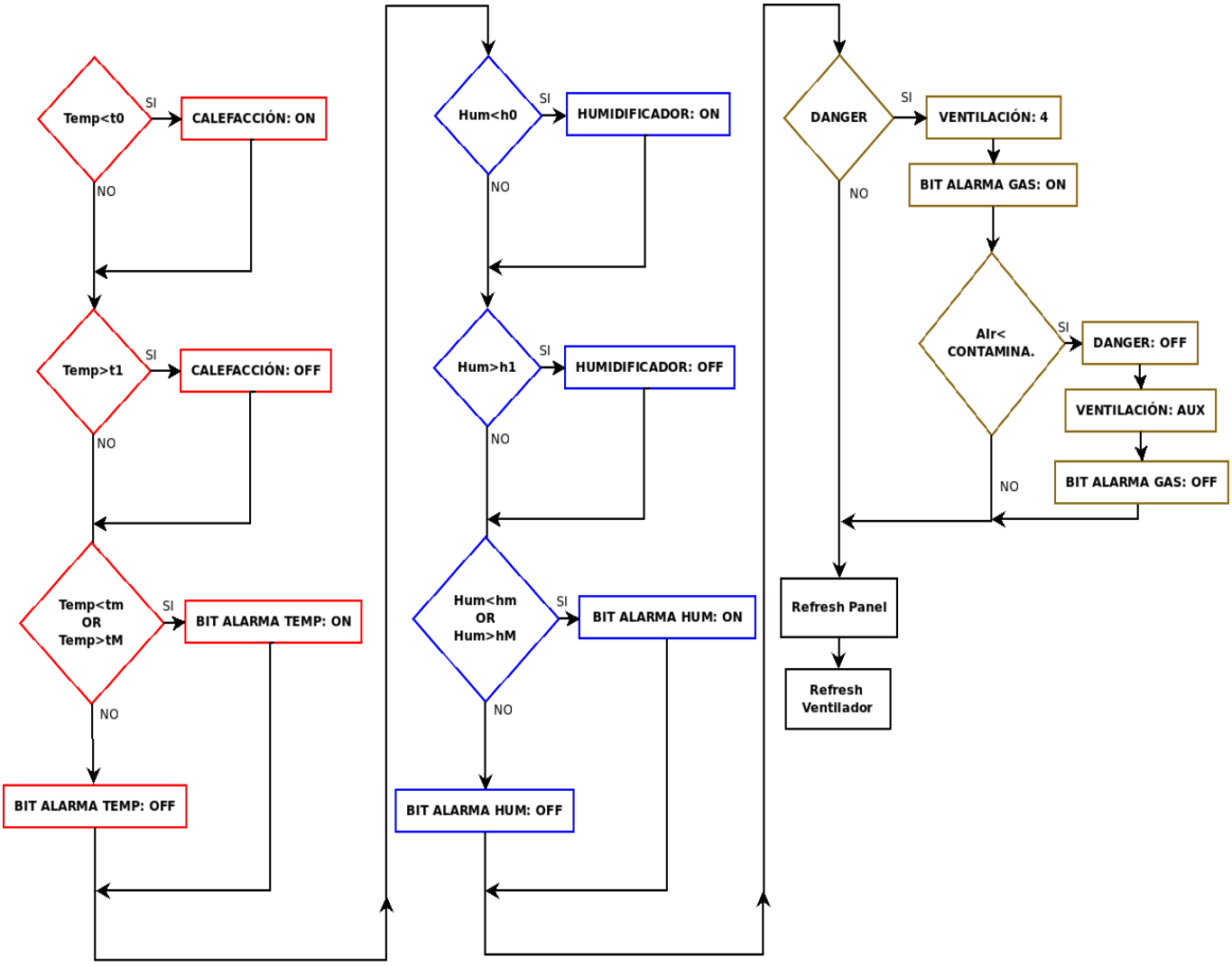



Figura 27: Diagrama de flujo método Check

El método se divide en cinco partes:

- Control temperatura
- Control humedad
- Control aire
- Control panel
- Control ventilador

El control de temperatura y humedad es similar, se evaluan los valores adquiridos del sensor *DHT22* y se activa o desactiva el actuador correspondiente. De igual forma, se comparan los valores obtenidos con los límites de alarmas, en caso de no encontrarse dentro del rango se activa el bit de alarma correspondiente en el vector *alarmas[ ].*

El control de aire evalua la variable de estado *DANGER*, en caso de haber sido activada por la interrupción del sensor *MQ2* se gestiona la renovación del aire en el interior, además del bit de *alarma* correspondiente.

A continuación se envía al objeto *panel* la información del *ventilador* y del vector de *alarmas[ ]* para que pueda ser representada.

Por último, se invoca el método *SetVentilador( )* encargado de establecer la potencia del *ventilador* a la velocidad fijada en su atributo *estado*.

#### 4.1.2.4. MÉTODO ADQ_

Este método, de contenido breve pero imprescindible, es el responsable de adquirir los valores de los sensores en el invernadero.

```
void Invernadero::Adq(){

  temp=sonda->CheckTemperature();
  hum=sonda->CheckHumidity();
  air=gas->Check();

}
```

Los atributos de *temp*, *hum* y *air* de *icra* actualizan su contenido a partir de los valores retornados por los metodos de lectura de los sensores.

#### 4.1.2.5. MÉTODO PAUSE

El método *Pause* es el complementario al método *Check*, ya que gestiona la **parada de todos los actuadores y alarmas** del sistema *(Ver "Figura 28")*.

La primera operación que se lleva a cabo es el apagado de los periféricos. En este modo de funcionamiento del sistema, no se registran alarmas ya que la placa corta todo suministro eléctrico en sus salidas.

La segunda parte del método gestiona el desenclavamiento de los interruptores de emergencia. La primera sentencia *if* evalua la desactivación del interruptor manual de paro, mientras que la segunda sentencia comprueba el estado del pulsador de parada remoto por *Android*.

En el caso de estar los dos desactivados, el sistema anula el estado PAUSA, reinicia el ventilador y alimenta de nuevo el LED de potencia del panel, indicando la salida de la parada.

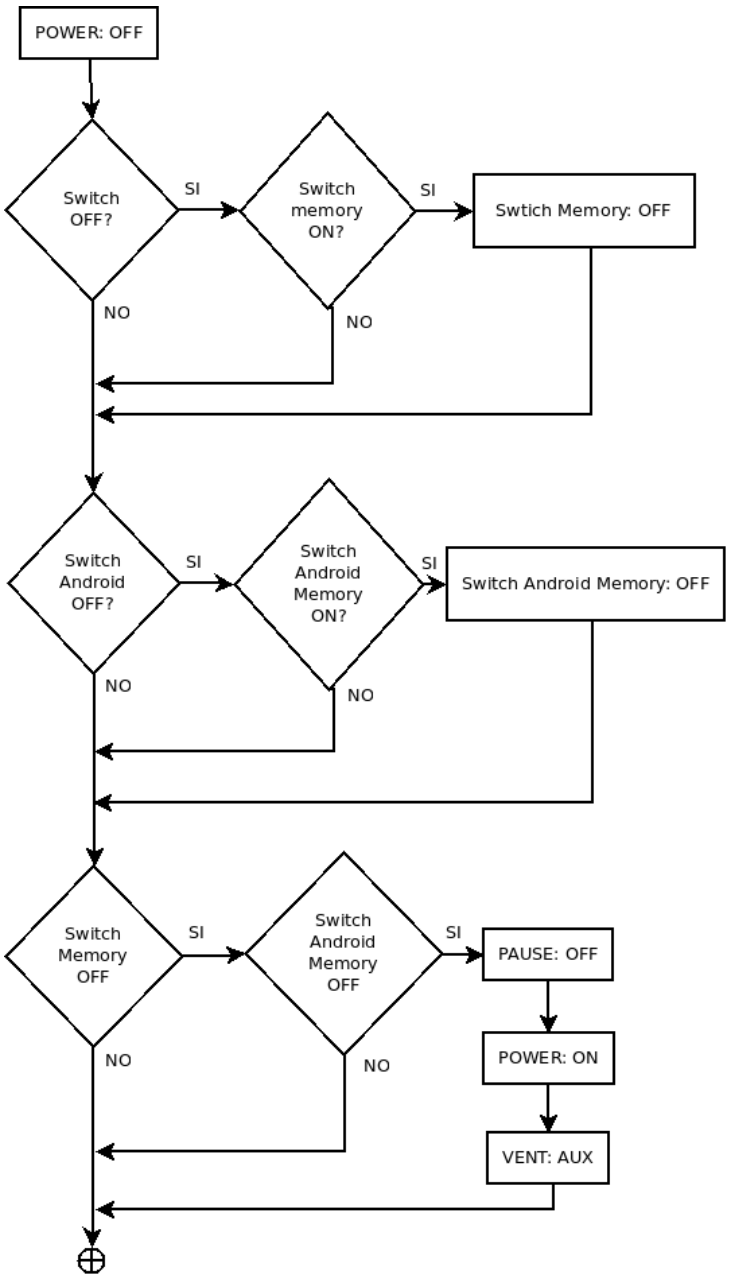



**Figura 28: Diagrama de flujo método Pause**

#### 4.1.2.6. MÉTODO SERIALPORT

El método *SerialPort* se encarga de gestionar las **funciones de comunicación** a través del puerto serie *(Ver "Figura 29").*

El método atiende el puerto serie, y en la situación de que exista un mensaje, extrae el primer carácter y lo evalua:

- y: El **servidor envía** un mensaje con nuevos parámetros desde *Android*.
- z: El **servidor solicita** el estado de las variables del invernadero.
- x: El **servidor solicita** el estado de los parámetros de trabajo.

Los métodos *FromServer* y *ToServer* se encargan de la comunicación entre la placa y el equipo servidor a través del puerto serie. Siendo el método *FromServer* el responsable de recibir los valores definidos por el usuario *Android* desde el servidor, mientras que la función *ToServer* lleva a cabo el envío de la información del invernadero al servidor.

En el caso de recibir el carácter "**y**", la placa debe recoger la información enviada por el servidor. Para ello es necesario definir el formato del mensaje esperado y establecer una trama de mensaje:

```
CMD="axx.xxbxx.xxcxx.xxdxx.xxexx.xxfxx.xxgxx.xxhxx.xxixjx";
```

Donde el significado de las diferentes partes del *string* son las siguiente:

- **axx.xx**: Valor mínimo de temperatura de trabajo
- **bxx.xx**: Valor máxima de temperatura de trabajo
- **cxx.xx**: Valor mínimo de temperatura para alarma
- **dxx.xx**: Valor máximo de temperatura para alarma
- **exx.xx**: Valor mínimo de humedad de trabajo
- **fxx.xx**: Valor máximo de humedad de trabajo
- **gxx.xx**: Valor mínimo de humedad para alarma
- **hxx.xx**: Valor máximo de humedad para alarma

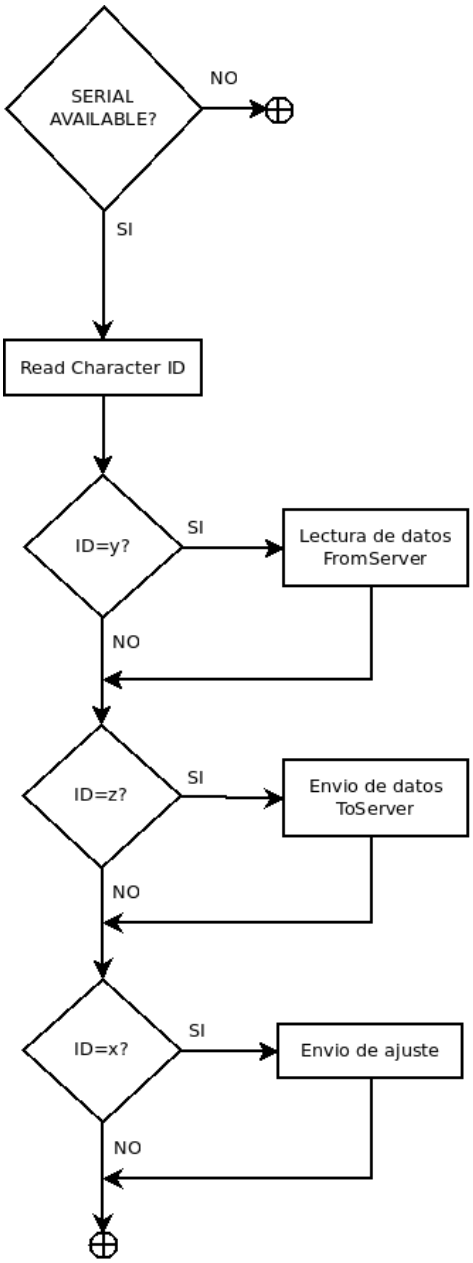



Figura 29: Diagrama de flujo método SerialPort

- **ix**: Valor de velocidad del ventilador
- **jx**: Estado del pulsador de parada

Cada dato está identificado con una *letra inicial,* seguido del tamaño esperado a recibir, representado por caracteres *x.*

Si el carácter recibido es "***z***", se responde con un mensaje conformado por los valores del invernadero. El método *ToServer( )* de la clase *Invernadero* invoca a los métodos *ToServer( )* de los objetos que forman sus atributos, de forma que, cada instancia de cada clase ya tiene conocimiento de cómo ha de enviar su información al equipo *servidor*. Además, se envía el estado de la variable PAUSE, cuya cabecera es la letra "*S*". El mensaje construido sigue la siguiente trama:

**Txx.xxHxx.xxAxxxVxCxWxPxxxSx**

- **Txx.xx**: Valor actual de temperatura [ºC]
- **Hxx.xx**: Valor actual de humedad [%]
- **Axxx**: Calidad del aire [%]
- **Vx**: Velocidad del ventilador [0-4]
- **Cx**: Estado del sistema de calefacción [0-1]
- **Wx**: Estado del sistema de humidificación [0-1]
- **Pxxx**: Panel de alarmas {temperatura, humedad, aire} [0-1, 0-1,0-1]
- **Sx**: Estado interruptor de parada / rearme [0-1]

Por último, en el caso de que el servidor envíe el carácter "***x***", *Arduino* responderá con el siguiente mensaje:

**axx.xxbxx.xxcxx.xxdxx.xxexx.xxfxx.xxgxx.xxhxx.xx**ixjx

Donde el significado de las diferentes partes es el siguiente:

- **axx.xx**: Valor mínimo de temperatura de trabajo
- **bxx.xx**: Valor máxima de temperatura de trabajo
- **cxx.xx**: Valor mínimo de temperatura para alarma
- **dxx.xx**: Valor máximo de temperatura para alarma
- **exx.xx**: Valor mínimo de humedad de trabajo
- **fxx.xx**: Valor máximo de humedad de trabajo
- **gxx.xx**: Valor mínimo de humedad para alarma
- **hxx.xx**: Valor máximo de humedad para alarma
- **ix**: Valor de velocidad del ventilador
- **jx**: Estado del pulsador de parada

Como se puede observar, este mensaje sigue la misma estructura que el mensaje recibido por el microcontrolador cuando el servidor desea establecer una nueva configuración. La finalidad de este mensaje es la de informar al servidor de los valores de trabajo configurados en el invernadero en ese instante.

En el siguiente punto del capítulo, se explica el software encargado de recibir esta información en la aplicación servidor.

## 4.2. DESARROLLO DEL SERVIDOR

Un servidor es un proceso que está pendiente de recibir órdenes de trabajo que provienen de otros procesos, que se denominan **clientes**. Una vez recibida la orden, la ejecuta y responde al peticionario con el resultado. La siguiente Figura *(Ver "Figura 30")* muestra cómo el proceso servidor atiende a los procesos clientes.

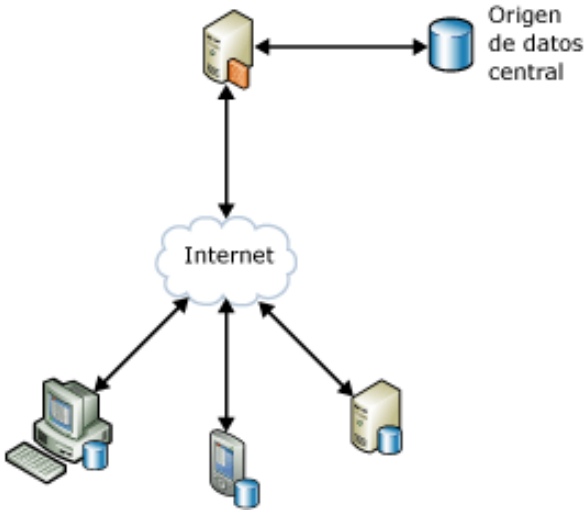



Figura 30: Esquema Servidor

El proceso servidor tiene la siguiente estructura de bucle infinito:

- Lectura de orden. El proceso está bloqueado esperando a que llegue una orden.
- Recibida la orden, el servidor la ejecuta.
- Finalizda la ejecución, el servidor responde con el resultado al proceso cliente y vuelve al punto de lectura de orden. (11)

Para implementar la comunicación entre el servidor y los clientes se ha optado por el uso de **servicios web**.

Un *servicio web* (en inglés, *Web Service* o *Web services*) es una tecnología que utiliza un conjunto de protocolos y estándares que sirven para intercambiar datos entre aplicaciones. Distintas aplicaciones de software desarrolladas en lenguajes de programación diferentes, y ejecutadas sobre cualquier plataforma, pueden utilizar los servicios web para intercambiar datos en redes de ordenadores como *Internet*. La interoperabilidad se consigue mediante la adopción de estándares abiertos (12). Véase la siguiente Figura para visualizar el concepto *(Ver "Figura 31")*:


**Figura 31: Esquema servicio web**

*Fuente: Aransay, Alberto Los Santos. Metodologías para el Desarrollo de Servicios en la Web. Universidad de Vigo. 2009. pág. 6.*

El concepto de *Servicio Web* puede resultar confuso, ya que no existe una definición única universalmente aceptada sobre lo que son y lo que el concepto engloba. Los *Servicios Web* surgieron como un conjunto de protocolos, estándares y recomendaciones, definidos por la *W3C* (*World Wide Web Consistorium*) y *OASIS* (*Organization for the Advancement of Structured Information Standards*), para lograr la interoperabilidad en la interacción entre máquinas, sistemas software y aplicaciones a través de la red

Estos servicios proporcionan mecanismos de comunicación estándares entre diferentes aplicaciones, que interactúan entre sí para presentar información dinámica al usuario. Para proporcionar interoperabilidad y extensibilidad entre estas aplicaciones, y que al mismo tiempo sea posible su combinación para realizar operaciones complejas, es necesaria una arquitectura de referencia estándar. (13)

Aunque se pueden encontrar diversos estilos de *Servicios Web*, en este proyecto se ha optado por los llamados servicios **REST**[13], descartando por ejemplo otras líneas como *XML-RPC* (considerado el precursor de *SOAP*).

A continuación se describe la estructura y funcionalidad del servidor.

### 4.2.1. CARACTERÍSTICAS DEL PC SERVIDOR

La aplicación servidor se ejecuta en un *PC* bajo soporte **LINUX**, en concreto la versión **Ubuntu 14.04 32bits.**

El equipo dispone de un servidor *HTTP* **Apache** y un sistema de gestión de bases de datos **MySQL**. El sistema de bases de datos *MySQL* ofrece los servicios de almacenamiento y control de tablas, requeridos para registrar los datos adquiridos desde la placa *Arduino*.

Por otro lado, el servidor tiene instaladas las dependencias **Boost**[14] necesarias para lanzar los servicios **REST**.

Para realizar la conexión física con la placa controladora, el equipo *PC* cuenta con un puerto **USB 2.0** y con conexión a Internet para comunicar con los terminales *Android*.

El *PC* servidor dispone de dos aplicaciones ejecutables:

- *serverICRA*: Software encargado de lanzar los servicios de comunicación con la tarjeta microcontroladora *Arduino*, escribir en la base de datos y atender peticiones *Android*.
- *usersICRA*: Aplicación de gestión de usuarios para el uso de *ICRApp*.

[13] La Transferencia de Estado Representacional (Representational State Transfer) o REST es una técnica de arquitectura software para sistemas hipermedia distribuidos como la World Wide Web. El término se originó en el año 2000, en una tesis doctoral sobre la web escrita por Roy Fielding, uno de los principales autores de la especificación del protocolo HTTP y ha pasado a ser ampliamente utilizado por la comunidad de desarrollo.

[14] Boost es un conjunto de bibliotecas de software libre y revisión por pares preparadas para extender las capacidades del lenguaje de programación C++. Su licencia, de tipo BSD, permite que sea utilizada en cualquier tipo de proyectos, ya sean comerciales o no.

Ambas aplicaciones están programadas en lenguaje *C++*, incluyendo las librerías y recursos necesarios para lanzar los servicios requeridos.

La siguiente Figura *(Ver Figura 32)* muestra los flujos de información del servidor con el resto de elementos:

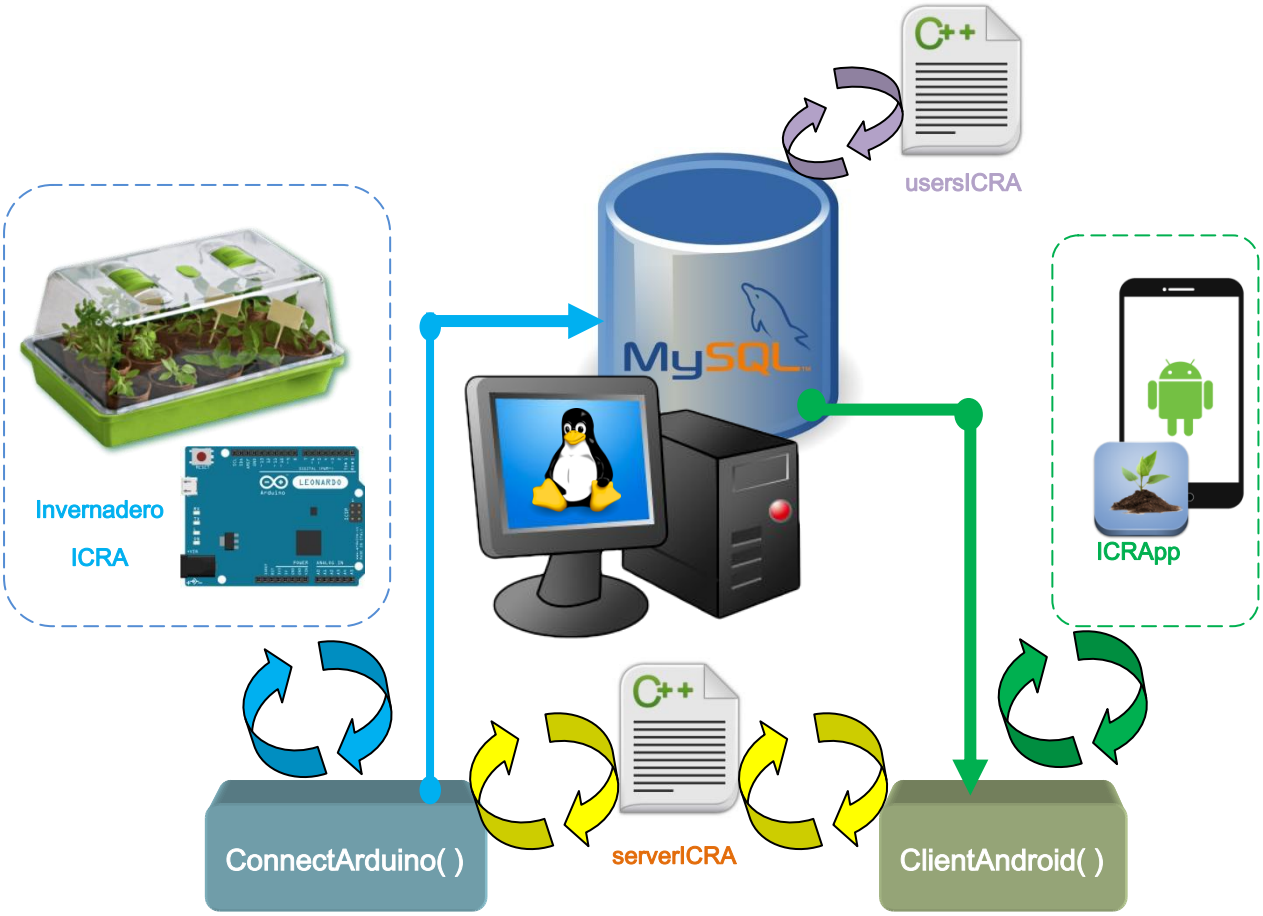



Figura 32: Esquema del servidor ICRA

Donde se observa como la tarea de la aplicación *serverICRA* recae en dos hilos concurrentes, *ConnectArduino* y *ClientAndroid*. El primer hilo se encarga de la comunicación entre la placa *Arduino* y el *PC*, y el segundo hilo mantiene los servicios *REST* para atender las peticiones de los clientes. La función *ConnectArduino* lleva a cabo la insercción de nuevos resgistros en la base de datos, y la función *ClientAndroid* realiza las consultas necesarias en base a las peticiones *Android*.

Por otro lado, la aplicación *usersICRA* realiza lecturas y escrituras en la base de datos para gestionar los grupos de usuarios del sistema.

A continuación se describen las diferentes partes que forman la aplicación servidor.

### 4.2.2. CLASE INVERNADEROSERVER

Con el fin de encapsular y recoger los valores y parámetros que hacen referencia a un invernadero, se ha desarrollado la clase *InvernaderoServer*. Dicha clase contiene los métodos necesarios para llevar a cabo la comunicación con la maqueta *ICRA* y gestionar la información para los usuarios *Android*, y además contiene atributos que permiten almacenar toda la información referente a parámetros, formato de mensajes…

Se muestra la estructura de la clase:

```
class InvernaderoServer{

      private:
            //Variables Arduino
            string port;
            char t[10],h[10],a[10],panel[5],vent[2],calef[2],
            humidi[2],state[2];
            //Variables Android
            string t0,t1,T0,T1,h0,h1,H0,H1,mode,estado;
            string CMD;//Data
            string CMD_0;//Adjuntment
      public:
            InvernaderoServer();
            void SetPort();
            string GetPort(){return port;}
            string GetCMD_0(){return CMD_0;}
            string GetCMD(){return CMD;}
            string SendMessage();
            void AdqAdjustment(char );
            void AdqData(char );
            void Monitor();
            string InsertMySQL(string ,char [],char []);
            string JsonMonitor();
            string JsonRangos();
            string JsonPanel();
            string JsonAlarmas();
            string JsonSetLim(string ,string);
            string JsonSetVent(string );
            string JsonSetSwitch(string );
  };
```

El atributo *port* almacena el puerto serie a emplear para la comunicación, utilizando los métodos *SetPort(int )* y *GetPort( )* para su tratamiento.

Las cadenas *char[ ]*:

```
t[10],h[10],a[10],panel[5],vent[2],calef[2],humidi[2],state[2];
```

se emplean para almacenar los valores de *temperatura, humedad, calidad de aire, nivel de ventilación, estado de calefacción, humidificador y parada del sistema* recogidos mediante el método *AdqData(char )* tras la recepción de valores desde la placa.

Los atributos:

```
string t0,t1,T0,T1,h0,h1,H0,H1,mode,estado;
```

contienen los parámetros de trabajo del *sistema de calefacción y humidificación, alarmas de temperatura y humedad*, nivel de *ventilación* y estado del *pulsador de parada* fijados por el usuario *Android.*

Los *strings CMD* y *CMD_0* contienen el formato de los mensajes de recepción desde la tarjeta *Arduino*; siendo el primero para el mensaje recibido mediante el método *AdqData(char )* y el segundo para la petición inicial de ajuste, *AdqAdjustment(char )*.

Mediante el método *AdqAdjustment(char )* se recopilan los parámetros enviados por la placa *Arduino* sobre sus rangos de trabajo, y a través de *AdqData(char )* se extrae el estado de las variables del invernadero. El método *SendMessage( )* envía los parámetros de trabajo definidos por el usuario *Android* al microcontrolador *Arduino*.

El método *InsertMySQL(string ,char [ ],char [ ])* retorna el *string* necesario para incluir en la base de datos toda la información del invernadero recogida en sus atributos.

Los métodos con la nomenclatura inicial *Json…* devuelven un *string* con estructura *json*, preparado para ser enviado en respuesta a una petición *GET* mediante los servicios *REST* lanzados.

*Monitor()* lleva a cabo la impresión por pantalla de la información recibida desde el microncontrolador, de forma que pueda visualizarse el estado de las variables del sistema desde el equipo servidor.

### 4.2.3. FUNCIÓN PRINCIPAL

El siguiente código muestra la **función main** de la aplicación *serverICRA*:

```
…
InvernaderoServer icraServer;//Objeto invernadero

int main()
{
     pthread_t httpget;

     Login();//Identificacion. Conexion MySQL. Insercion SerialPort

     pthread_create(&httpget, NULL, ClientAndroid, NULL);//Hilo REST
     cout<<"\033[1;34mLanzado hilo ClientAndroid\033[0m"<<endl;

     ConnectArduino();//Intercomunicacion con placa

     pthread_exit(NULL);
     conn.disconnect();//Desconexion MySQL
     cout<<"END PROGRAM"<<endl;
     return 0;
}
```

Antes de lanzar la función *main*, se crea una instancia de la clase *InvernaderoServer* con ámbito global, con el fin de emplear este objeto en la comunicación y almacenamiento de información.

Dentro de la función principal, se crea la instancia *httpget* de la clase *pthread_t*, objeto encargado de lanzar el *hilo*[15] de atención a los clientes *Android.*

La siguiente línea de código lanza la función ***Login( )***, responsable de crear la interfaz de identificación, y solicitar los valore referentes a la creación de la base de datos, tablas, *fichero de registro* y puerto de conexión con la placa *Arduino*.

Acto seguido, se lanza el *hilo* de atención a usuarios, el cual contiene los servicios *REST*. Para ello se utiliza la función ***ClientAndroid( )***, contenedora de estos servicios.

```
pthread_create(&httpget, NULL, ClientAndroid, NULL);//Hilo REST
```

Lanzado el hilo de servicios *REST*, la función de conexión con *Arduino* es ejecutada. Esta función, ***ConnectArduino( )***, mantiene el protocolo de comunicación con la tarjeta microcontroladora, realizando con ello, el envío y recepción de información, y la inserción de los valores del invernadero en la base de datos.

[15] En sistemas operativos, un hilo de ejecución, hebra o subproceso es la unidad de procesamiento más pequeña que puede ser planificada por un sistema operativo. La creación de un nuevo hilo es una característica que permite a una aplicación realizar varias tareas a la vez (concurrentemente). Los distintos hilos de ejecución comparten una serie de recursos tales como el espacio de memoria, los archivos abiertos, situación de autenticación, etc. Esta técnica permite simplificar el diseño de una aplicación que debe llevar a cabo distintas funciones simultáneamente.

Por último, se ejecutan las instrucciones de finalización de hilos y desconexión con la base de datos.

A continuación se pasa a explicar la creación de la base de datos y la conexión con esta.

### 4.2.4. Conexión MySQL

**MySQL** es la base de datos de código abierto más popular del mundo. Código abierto significa que todo el mundo puede acceder al código fuente, es decir, al código de programación de *MySQL*, y todo el mundo puede contribuir para incluir elementos, arreglar problemas, realizar mejoras o sugerir optimizaciones.

*MySQL* es un sistema de administración de bases de datos relacional (*RDBMS*). Se trata de un programa capaz de almacenar una enorme cantidad de datos de gran variedad y de distribuirlos para cubrir las necesidades de cualquier tipo de organización, desde pequeños establecimientos comerciales a grandes empresas y organismos administrativos. *MySQL* compite con sistemas *RDBMS* propietarios conocidos, como *Oracle, SQL Server* y *DB2*.

*MySQL* incluye todos los elementos necesarios para instalar el programa, preparar diferentes niveles de acceso de usuario, administrar el sistema y proteger y hacer volcados de datos. (14)

Para poder llevar a cabo la conexión de la aplicación servidor con la base de datos se ha utilizado una librería específica:

```
#include <MySQL++/MySQL++.h>
```

Dicha librería permite llevar a cabo las operaciones de lectura y escritura en las tablas de la base de datos.

A continuación se muestra el proceso de creación de la base de datos y tabla a utilizar *(Ver Figura 33”).*

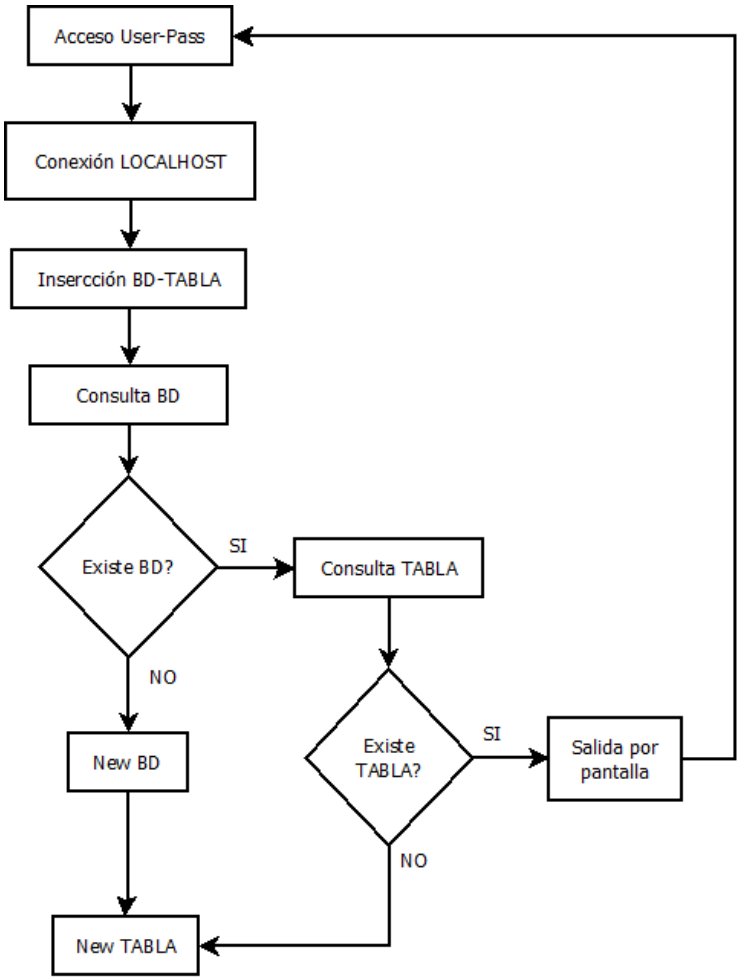



**Figura 33: Diagrama de flujo conexión MySQL**

La función *Login()* se ejecuta al iniciar la función principal y, como se ha dicho previamente, lleva a cabo la labor de crear la interfaz de usuario de la aplicación y la recogida de información introducida por el usuario.

Tras recibir los valores correctos de *user* y *password*, se procede a la creación de la base de datos, por lo que solicita el nombre de la *tabla* a emplear. En el caso de que la *tabla* exista, se mostrará por pantalla su contenido y se solicitará ingresar un nombre que no esté en uso.

En el "*ANEXO IV: Manual usuario IcraServer"* se explican los pasos para llevar a cabo el arranque de la aplicación.

### 4.2.5. FICHERO DE REGISTRO

Una vez definida la *base de datos* y *tabla* a emplear, la aplicación solicita insertar una descripción de la plantación a monitorizar.

Para el almacenamiento de esta información se ha dedicado una función denominada *File()*:

```
void Login(){

      …

      registro=File(db,tabla);//Creacion fichero registro plantación

      …
}
```

Dicha función[16] contiene las instrucciones necesarias para almacenar la información tecleado por el usuario en un fichero *.txt* de registro, donde el nombre del fichero sigue el siguiente formato:

BASEdeDATOS_TABLA.txt

La función retorna un *string* con el nombre establecido y el fichero se guarda en el directorio de la aplicación.

### 4.2.6. COMUNICACIÓN ARDUINO - SERVIDOR

La comunicación entre la placa *Arduino* y la aplicación *serverICRA* está encapsulada en una función denominada *ConnectArduino(),* la cual establece conexión con el microcontrolador y mantiene un protocolo de comunicación.

La función *ConnectArduino()* es invocada en el *main* del programa, y mantiene un bucle infinito de envío-recepción de mensajes entre el invernadero y el equipo servidor, siguiendo el siguiente flujo *(Ver "Figura 34")*:

- *ICRAserver* comprueba si un usuario *Android* ha establecido un nuevo parámetro de trabajo en el invernadero, en tal caso envía un mensaje con los valores actuales a la placa *Arduino*.
- *ICRAserver* solicita los valores del invernadero a la placa *Arduino.*

[16] Ver *el proyecto de Arduino, disponible en el CD de la memoria, para obtener más información sobre el código.*

- *Arduino* responde a la anterior solicitud con un mensaje confeccionado con los valores de sus variables.

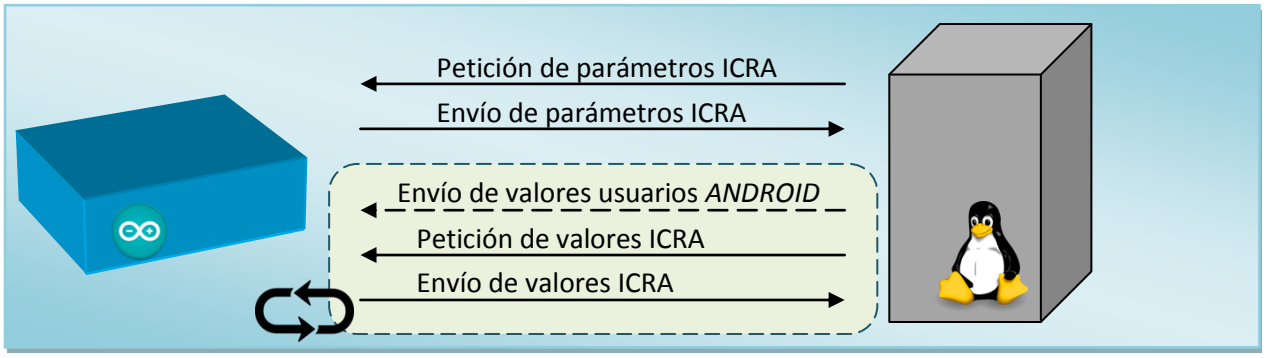



Figura 34: Comunicación Arduino-Servidor

Antes de adentrarse en este bucle, el servidor solicita al microcontrolador los parámetros de trabajo que almacena la placa, de forma que el usuario *Android* pueda conocer el estado de los parámetros con los que la placa estaba trabajando.

A continuación se muestra el diagrama de flujo que refiere a este intercambio de datos *Servidor-Microcontrolador (Ver "Figura 35").*

Se crea un objeto *SerialPort* para llevar a cabo la comunicación a través del puerto que ha insertado el usuario en la interfaz de la aplicación *serverICRA*.

Lo primero de todo el servidor realiza una petición a la placa controladora solicitando los parámetros de trabajo que tiene establecidos. Tras recibir respuesta, un bucle *for* se encarga de desmenuzar el buffer recibido para extraer todos los parámetros y recogerlos en los atributos correspondientes.

Realizado esto, la función entra en un bucle permanente. Se evalúa el bit **permiso,** encargado de indicar si un usuario *Android* ha enviado nuevos parámetros al servidor, y en caso afirmativo se lleva a cabo el proceso de envío de datos al invernadero mediante el método *SendMessage().*

```
string InvernaderoServer::SendMessage(){
     //ID cabecera
     string message="y";
     //Parametros usuario Android
     message+="a";message+=t0;
     message+="b";message+=t1;
     message+="c";message+=T0;
     message+="d";message+=T1;
     message+="e";message+=h0;
     message+="f";message+=h1;
     message+="g";message+=H0;
     message+="h";message+=H1;
     message+="i";message+=mode;
     message+="j";message+=estado;
     cout<<"Send: "<<message<<endl;
     return message;
}
```

Para ello, se forma la trama del mensaje encabezada por el *ID "**y**"*, que identifica la cadena como el mensaje de envío de usuario *Android*. Seguido, se concatenan los valores de usuario con sus *IDs* respectivos y se lanza el mensaje por el puerto serie (Ver el punto "*4.1.2.6 Método SerialPort()"* para observar la recepción por parte de la placa *Arduino*).

El mensaje quedaría de esta forma:

```
axx.xxbxx.xxcxx.xxdxx.xxexx.xxfxx.xxgxx.xxhxx.xxixjx
```

El bit de permiso se desactiva, a la espera de ser reactivado por alguna petición *Android* desde los servicios *REST*.

A continuación el servidor envía una petición a la placa *Arduino* solicitando la información del invernadero (en este caso, se ha establecido el carácter "***z***" como identificador). La tarjeta microcontroladora tras recibir la petición, envía por el puerto serie toda la información del sistema al servidor.

El servidor almacena la información en un buffer de datos, cuyo tamaño está definido por la longitud del *string* CMD. El formato de la cadena es el siguiente:

```
string CMD="Txx.xxHxx.xxAxxxVxCxWxPxxxSx";
```

donde se almacena la información de: *temperatura, humedad, estado del aire, velocidad ventilador, estado calefacción, estado humidificador, alarmas y estado del interruptor de parada*. A continuación se desglosa el contenido:

- **Txx.xx**: Valor actual de temperatura
- **Hxx.xx**: Valor actual de humedad
- **Axxx**: Calidad del aire
- **Vx**: Velocidad del ventilador
- **Cx**: Estado del sistema de calefacción
- **Wx**: Estado del sistema de humidificación
- **Pxxx**: Panel de alarmas {temperatura, humedad, aire}
- **Sx**: Estado interruptor de parada / rearme

Al igual que se vio en la trama del mensaje que recibe la placa *Arduino*, la primera letra de cada porción de mensaje representa el identificador de la variable.

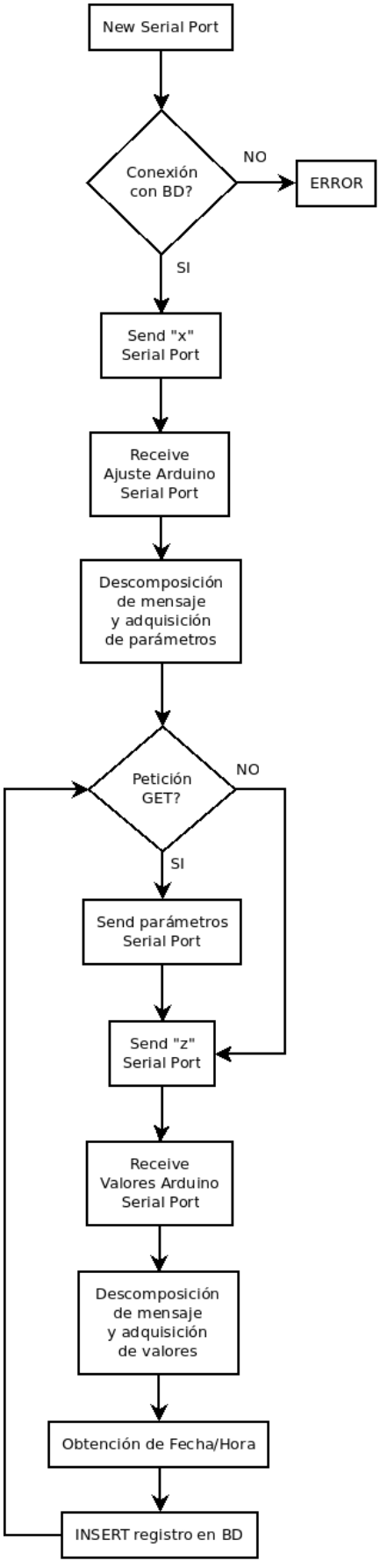



Figura 35: Diagrama de flujo Serial Server

Recogida toda esta información en el *buffer*, se recorre el contenido mediante un bucle *for{ },* con el fin de extraer el contenido de cada variable para su correcto procesamiento. Esta operación es llevada a cabo por el método *AdqData(char ):*

```
void InvernaderoServer::AdqData(char c){

        char *p;//puntero recepcion
        int i;//indice puntero p[ii]
        //Identificacion de valores
        if(c=='T'){//Temperatura
              p=t;//asignacion del puntero
              i=0;//reset indice
        }
        else if(c=='H'){//Humedad
              p=h;
              i=0;
        }
        …
        else {//Recogida del valor
              if(c==' ')c='0';//Sustitucion ' ' por '0'
              p[i++]=c;//Obtencion del valor
              p[i]='\0';//Fin de cadena
        }
}
```

Para ello, se recoge cada componente del buffer en una variable tipo *char* y se pasa por valor su contenido al método *AdqData(char )* del objeto *invernaderoServer*. Dicho método emplea un puntero de extracción, *char *p*, el cual tendrá la función de pasarela entre la componente del *buffer* y el atributo correspondiente. Tras detectar un *ID* de variable, el puntero señala al atributo adecuado. En el caso de detectar una cifra en el *buffer*, está se almacenará en la variable conectada mediante el puntero.

El siguiente punto del capítulo explica la inserción de los valores recogidos en la base de datos *MySQL*.

### 4.2.7. ALMACENAMIENTO EN LA BASE DE DATOS

Una vez que se han adquirido los valores enviados por la placa *Arduino*, la aplicación servidor debe crear un nuevo registro en la *tabla* asignada, dentro de la *base de datos* enlazada.

Los valores tomados serán guardados en una nueva fila de la *tabla*, además de la *hora* y *fecha* en la que se tomo dicha medida.

```
void ConnectArduino(){
    …
    while(1){//Bucle de comunicación
          …
          //Obtencion Fecha/Hora actual
          Date(time,date);
          //MySQL
          query.reset();
          query << icraServer.InsertMySQL(tabla,time,date);
          if(!query.execute())
                cout << "Fallo al introducir datos"<<endl;

          …
}
```

El método *InsertMySQL()* recibe por parámetro la *tabla* en la que se debe insertar la nueva fila y la *fecha* y *hora* de la adquisición:

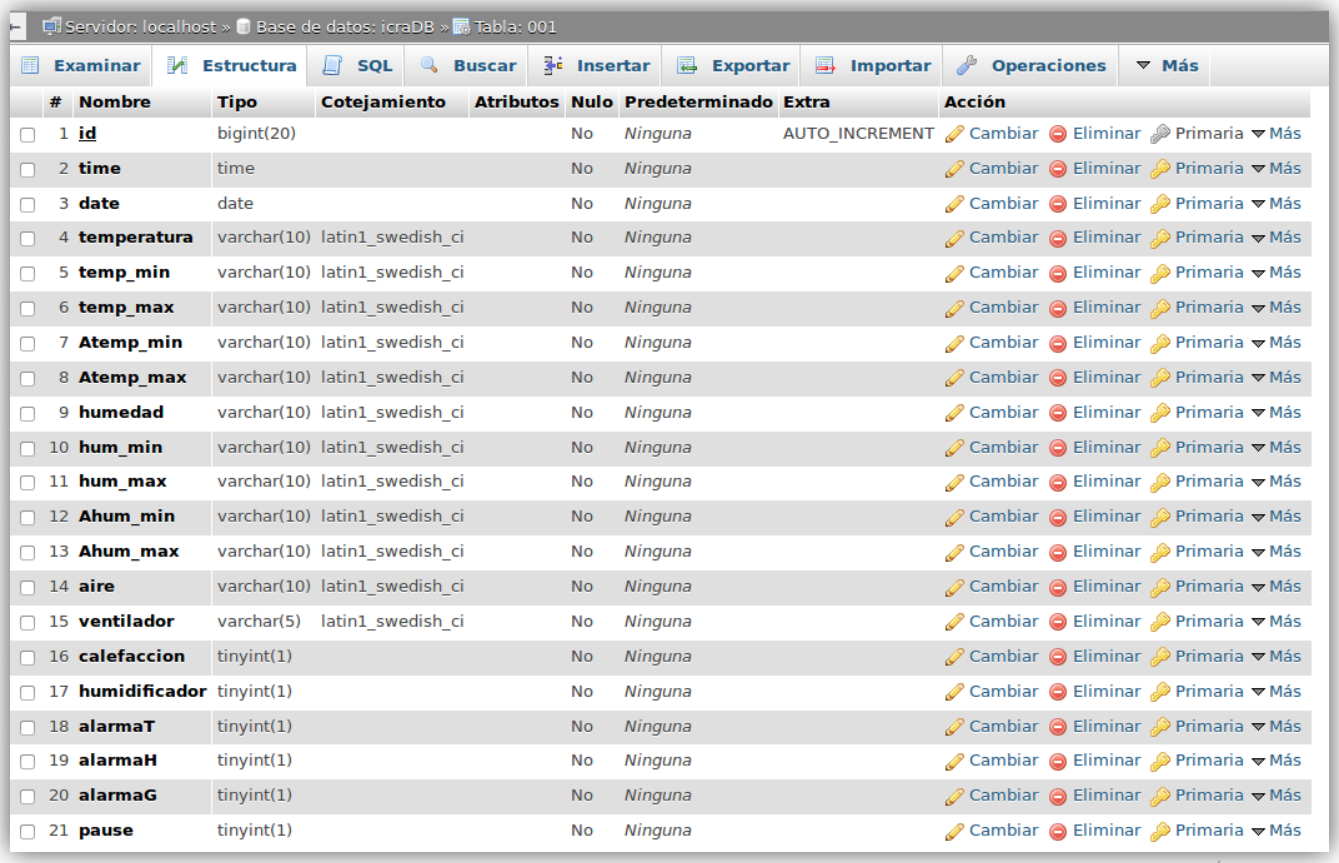
```
string InvernaderoServer::InsertMySQL(string tabla,char time[],char
date[]){

string aux="INSERT INTO `"+tabla+"` VALUES (NULL, '"+time+"',
'"+date+"', '"+t+"', '"+t0+"', '"+t1+"', '"+T0+"', '"+T1+"', '"+h+"',
'"+h0+"', '"+h1+"', '"+H0+"', '"+H1+"', '"+a+"', '"+vent+"','"+calef+"',
'"+humidi+"', '"+panel[0]+"', '"+panel[1]+"', '"+panel[2]+"',
'"+state+"');";
return(aux);
}
```

La función cose una sentencia *MySQL* a partir de todas las variables que intervienen en el invernadero y retorna dicho código para insertar un nuevo registro en la *tabla*. Cada registro tiene asociado un "*primary key*" que permite identificar cada fila de forma unívoca.

La siguiente imagen *(Ver "Figura 36")* muestra la estructura de la *tabla* vista a través de la aplicación *phpMyAdmin*[17]:

[17] phpMyAdmin es una herramienta escrita en PHP con la intención de manejar la administración de *MySQL* a través de páginas web, utilizando Internet.



Servidor: localhost » Base de datos: icraDB » Tabla: 001

Examinar | Estructura | SQL | Buscar | Insertar | Exportar | Importar | Operaciones | Más

| # | Nombre | Tipo | Cotejamiento | Atributos | Nulo | Predeterminado | Extra | Acción |
|---|---|---|---|---|---|---|---|---|
| 1 | id | bigint(20) | | | No | Ninguna | AUTO_INCREMENT | Cambiar Eliminar Primaria Más |
| 2 | time | time | | | No | Ninguna | | Cambiar Eliminar Primaria Más |
| 3 | date | date | | | No | Ninguna | | Cambiar Eliminar Primaria Más |
| 4 | temperatura | varchar(10) | latin1_swedish_ci | | No | Ninguna | | Cambiar Eliminar Primaria Más |
| 5 | temp_min | varchar(10) | latin1_swedish_ci | | No | Ninguna | | Cambiar Eliminar Primaria Más |
| 6 | temp_max | varchar(10) | latin1_swedish_ci | | No | Ninguna | | Cambiar Eliminar Primaria Más |
| 7 | Atemp_min | varchar(10) | latin1_swedish_ci | | No | Ninguna | | Cambiar Eliminar Primaria Más |
| 8 | Atemp_max | varchar(10) | latin1_swedish_ci | | No | Ninguna | | Cambiar Eliminar Primaria Más |
| 9 | humedad | varchar(10) | latin1_swedish_ci | | No | Ninguna | | Cambiar Eliminar Primaria Más |
| 10 | hum_min | varchar(10) | latin1_swedish_ci | | No | Ninguna | | Cambiar Eliminar Primaria Más |
| 11 | hum_max | varchar(10) | latin1_swedish_ci | | No | Ninguna | | Cambiar Eliminar Primaria Más |
| 12 | Ahum_min | varchar(10) | latin1_swedish_ci | | No | Ninguna | | Cambiar Eliminar Primaria Más |
| 13 | Ahum_max | varchar(10) | latin1_swedish_ci | | No | Ninguna | | Cambiar Eliminar Primaria Más |
| 14 | aire | varchar(10) | latin1_swedish_ci | | No | Ninguna | | Cambiar Eliminar Primaria Más |
| 15 | ventilador | varchar(5) | latin1_swedish_ci | | No | Ninguna | | Cambiar Eliminar Primaria Más |
| 16 | calefaccion | tinyint(1) | | | No | Ninguna | | Cambiar Eliminar Primaria Más |
| 17 | humidificador | tinyint(1) | | | No | Ninguna | | Cambiar Eliminar Primaria Más |
| 18 | alarmaT | tinyint(1) | | | No | Ninguna | | Cambiar Eliminar Primaria Más |
| 19 | alarmaH | tinyint(1) | | | No | Ninguna | | Cambiar Eliminar Primaria Más |
| 20 | alarmaG | tinyint(1) | | | No | Ninguna | | Cambiar Eliminar Primaria Más |
| 21 | pause | tinyint(1) | | | No | Ninguna | | Cambiar Eliminar Primaria Más |

Figura 36: Estructura de una tabla de plantación en phpMyAdmin

Donde cada campo tiene el siguiente significado:

1- **id**: Identificador clave de la tabla
2- **time**: Hora en la que se adquirió la medida
3- **date**: Fecha en la que se adquirió la medida
4- **temperatura**: Valor de temperatura del invernadero
5- **temp_min**: Umbral inferior de temperatura de trabajo
6- **temp_max**: Umbral superior de temperatura de trabajo
7- **Atemp_min**: Umbral inferior de alarma de temperatura
8- **Atemp_max**: Umbral superior de alarma de temperatura
9- **humedad**: Valor de humedad del invernadero
10- **hum_min**: Umbral inferior de humedad de trabajo
11- **hum_max**: Umbral superior de humedad de trabajo
12- **Ahum_min**: Umbral inferior de alarma de humedad
13- **Ahum_max**: Umbral superior de alarma de humedad
14- **aire**: Valor % de calidad de aire
15- **ventilador**: Velocidad de ventilación.
16- **calefaccion**: Estado de activación del sistema de calefacción
17- **humidificador**: Estado de activación del sistema humidificador
18- **alarmaT**: Estado de activación de la alarma de temperatura
19- **alarmaH**: Estado de activación de la alarma de humedad
20- **alarmaG**: Estado de activación de la alarma de gas
21- **pause**: Estado de activación del modo parada

El tipo de dato en cada campo atiende al tamaño del valor a insertar. Por ejemplo, en el caso del campo *16-calefacción*, los valores almacenados serán {0,1}, indicando el estado de activación, por lo que se emplea un tipo *tinyint(1)*. Mientras que para el almacenamiento del valor *4-temperatura,* se esperan valores del siguiente tamaño {25.60, 18.90…}, referentes a ºC; por lo que para garantizar su almacenamiento se emplea el tipo *varchar(10)*.

### 4.2.8. Comunicación Servidor - Android

Además de la labor de comunicación con la tarjeta *Arduino*, la aplicación servidor ha de encargarse de gestionar la atención hacia los usuarios *Android*. Para ello, la aplicación tiene que recurrir a la concurrencia de procesos.

Como se ha visto al principio del capítulo, la función *main* lanza un *hilo* de atención de usuarios llamado **ClientAndroid( )*. El hilo de atención a usuarios se mantiene a la escucha de peticiones y, para que se lleve a cabo una petición, un usuario *Android* tiene que utilizar el método *GET*, basado en el protocolo *HTTP*.

Es un protocolo orientado a transacciones y sigue el esquema *petición-respuesta* entre un cliente y un servidor. Al cliente que efectúa la petición (un navegador web o un spider) se lo conoce como "*user agent*" (agente del usuario). A la información transmitida se la llama recurso y se la identifica mediante un localizador uniforme de recursos (*URL*). El resultado de la ejecución de un programa, una consulta a una base de datos, la traducción automática de un documento, etc.

*HTTP* es un protocolo sin estado, es decir, que no guarda ninguna información sobre conexiones anteriores. El desarrollo de aplicaciones web necesita frecuentemente mantener estado. Para esto se usan las *cookies*, que es información que un servidor puede almacenar en el sistema cliente. Esto le permite a las aplicaciones web instituir la noción de "sesión", y también permite rastrear usuarios ya que las cookies pueden guardarse en el cliente por tiempo indeterminado.

El concepto *GET* es obtener información del servidor. Traer datos que están en el servidor, ya sea en un archivo o base de datos, al cliente. Independientemente de que para ello se tenga que enviar (*request*) algún dato que será procesado para luego devolver la respuesta (*response*) que se espera, como por ejemplo, un identificador para obtener un valor en una base de datos.

El esquema de comunicación es el siguiente *(Ver "Figura 37"):*

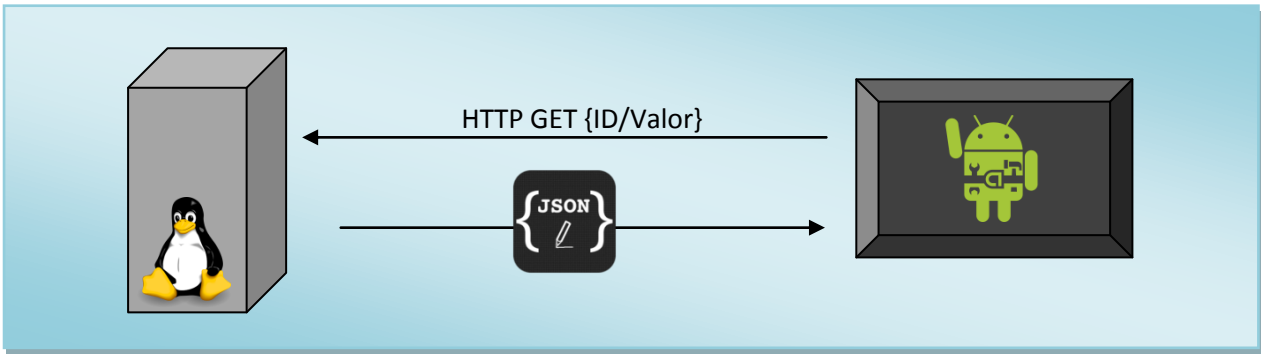



**Figura 37: Comunicación Servidor-Android**

El cliente *Android* realiza una solicitud basada en el método *GET* utilizando una *URL* específica para realizar la petición.

La estructura de la sentencia es la siguiente *(Ver "Figura 38"):*

```
http://ip_servidor:puerto/id/valores
```

**Figura 38: Solicitud GET**

El mensaje comienza con el identificador de protocolo **http://**. Tras ello, se inserta la dirección **IP** del equipo servidor, el cual proporciona los servicios *REST*. Además de señalar la dirección *IP*, es necesario indicar el número de **puerto** a emplear para la conexión. Identificado el equipo, se especifica un **id** para la consulta permitiendo al servidor identificar de alguna forma las necesidades del cliente.

*Por ejemplo, si un cliente quiere conocer la puntuación actual de un partido de futbol, realizará una petición al servidor de la siguiente forma:*

*http://192.168.1.15:8080/puntos*

*El servidor cuya dirección IP sea 192.168.1.15, recibirá el id "puntos" por su puerto 8080. Tras ello, localizará la puntuación del marcador y responderá al cliente con su contenido.*

La parte final de la *URL*, **valor**, permite enviar al servidor un dato para que almacene e interprete.

*Por ejemplo, si un cliente desea conocer el mínimo común múltiplo del número 10, enviaría el siguiente mensaje:* *http://192.168.1.15:8080/mcm/10*

*El servidor recibirá el id "mcm", y a continuación el valor "10". Tras ello, realizará el algoritmo de cálculo con el valor recibido del cliente y le enviará el resultado.*

Utilizando esta técnica, la aplicación *serverICRA* lleva a cabo la labor de operador de comunicaciones con los clientes *Android*.

A continuación se muestra un bloque de código del hilo de servicios *REST*:

```
void *Client Android(void *var){

      int port=8080;
      Server<HTTP> server(port, 4);
      …
      ///GET Monitor
      server.resource["^/monitor/?$"]["GET"]=[](ostream& response,
      shared_ptr<Server<HTTP>::Request> request) {
            string json=icraServer.JsonMonitor();
            response << "HTTP/1.1 200 OK\r\nContent-Length: " <<
            json.length() << "\r\n\r\n" <<json;
      };
      …
}
```

```
{
string json="{temp: ";
json+=t;
json+=", hum: ";
json+=h;
json+=", air: ";
json+=a;
json+=", calef: ";
json+=calef;
json+=", humidi: ";
json+=humidi;
json+=", vent: ";
json+=vent;
json+=", Atemp: ";
json+=panel[0];
json+=", Ahum: ";
json+=panel[1];
json+=", Aair: ";
json+=panel[2];
json+=", switch:
";json+=estado;
json+="}";

return json;
}
```

La primera instrucción que ejecuta el hilo es crear un objeto servidor en el puerto 8080, el cual será el encargado de atender las peticiones *GET*. Seguido se define la petición */monitor/*, responsable de enviar el estado de las variables del invernadero al cliente. En el caso de que un usuario lleve a cabo una solicitud de este tipo:

*http://IP_server:port/monitor/*

el servidor genera un *string* ,denominado *json* cuyo contenido vendrá dado por la respuesta del método *JsonMonitor( )* y contendrá la siguiente información:

- **Temperatura**
- **Humedad**
- **Aire**
- **Estado de calefacción**
- **Estado de humidificador**
- **Nivel de ventilación**
- **Vector de alarmas**
- **Estado del pulsador Android**

El contenido de esta cadena *string* atiende a una estructura particular basada en el uso de elementos ***JSON***.

*JSON* es una notación de objetos basada en *JavaScript*, que utiliza una sintaxis que permite crear objetos de manera rápida y simple. En el siguiente punto, "*4.3 Desarrollo de la aplicación móvil*", se explica con más detalle la estructura y aplicación de estos elementos.

Finalmente se utiliza la siguiente sentencia para realizar la respuesta del mensaje, empleando el protocolo *HTTP.*

```
response << "HTTP/1.1 200 OK\r\nContent-Length: " << json.length() << "\r\n\r\n"
<<json;
```

La siguiente tabla (*Ver "Tabla 6"*) muestra las sentencias *GET* implementadas:

**Tabla 6: Servicios REST serverICRA**

| Definición | Sentencia | Valor de entrada W | Respuesta |
|---|---|---|---|
| **Bit de vida** | …/life | n/a | Código de servidor operativo |
| **Monitor** | …/monitor | n/a | Estado de las variables del sistema |
| **Registro** | …/registro | n/a | Nombre del fichero registro de plantación |
| **Login** | …/login/W | User & Pass | Nivel de privilegios del usuario |
| **Grafica** | …/grafica/W | Parámetros & Rangos temporales | Vector de datos temporales |
| **Rangos Actuadores** | …/rangos | n/a | Valor de los rangos de trabajo de los actuadores. |
| **Panel Alarmas** | …/panel | n/a | Vector de alarmas |
| **Rangos Alarmas** | …/alarmas | n/a | Valor de los rangos de alarmas |
| **Temp. Inf de trabajo** | …/t0/W | Valor de temperatura ºC | Valor de temperatura de trabajo inferior fijado |
| **Temp. Sup de trabajo** | …/t1/W | Valor de temperatura ºC | Valor de temperatura de trabajo superior fijado |
| **Temp. Inf de alarma** | …/t0a/W | Valor de temperatura ºC | Valor de temperatura de alarma inferior fijado |
| **Temp. Sup de alarma** | …/t1a/W | Valor de temperatura ºC | Valor de temperatura de alarma superior fijado |
| **Hum. Inf de trabajo** | …/h0/W | Valor de humedad % | Valor de humedad de trabajo inferior fijado |
| **Hum. Sup de trabajo** | …/h1/W | Valor de humedad % | Valor de humedad de trabajo superior fijado |
| **Hum. Inf de alarma** | …/h0a/W | Valor de humedad % | Valor de humedad de alarma inferior fijado |
| **Hum. Sup de alarma** | …/h1a/W | Valor de humedad % | Valor de humedad de alarma superior fijado |
| **Nivel ventilación** | …/vent/W | Nivel de ventilación | Valor de ventilación fijado |
| **Interruptor remoto** | …/switch/W | Estado del interruptor | Estado del interruptor de parada remoto |

Se procede a estudiar algunas sentencias *GET* relevantes, ya que se salen de la estructura básica vista en el anterior ejemplo.

La solicitud *GET* para el **login** incorpora una consulta a la base de datos *(Ver "Figura 39")* en búsqueda de la existencia del usuario y de la contraseña proporcionados desde el terminal *Android*. Dicha sentencia *GET* incorpora el *user* y *pass* introducidos por el usuario de la siguiente forma:

*http://IP_server:port/**login**/user/pass*

Se extraen los dos parámetros en dos variables auxiliares tipo *string* y se forma una sentencia *MySQL* para realizar una consulta en la *base de datos*. En el caso de encontrar resultado, se extrae el **nivel de permiso** asociado al usuario y se responde a la petición *GET* mediante un objeto *json* que porte el nivel de acceso.

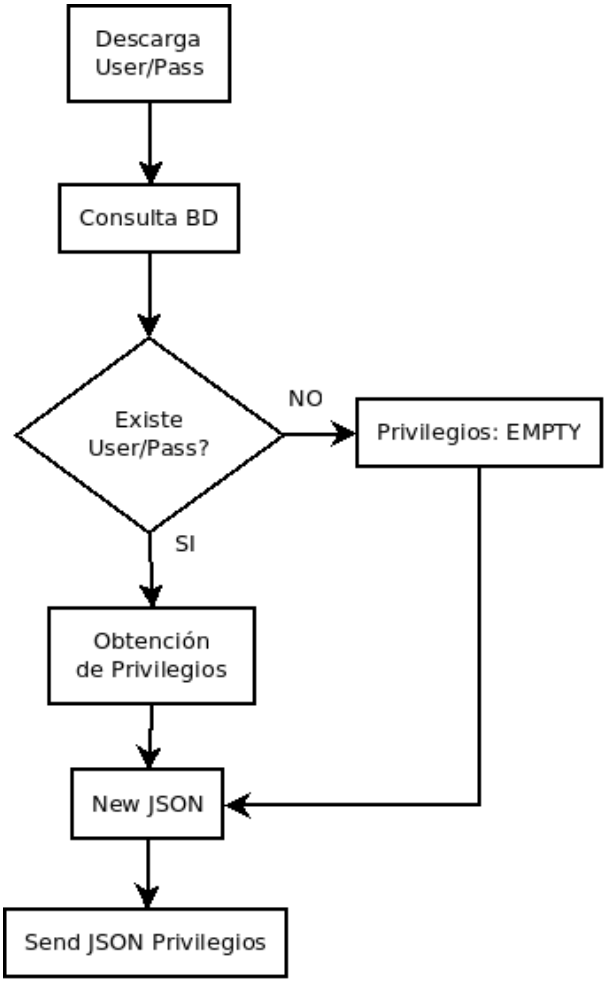



**Figura 39: Diagrama de flujo GET Login**

La gestión de recopilación de datos para **gráficas** sigue una sentencia *GET* con la siguiente estructura:

*http://IP_server:port/**grafica**/param1/param2/fecha0/hora0/fecha1/hora1*

La función de atención *(Ver "Figura 40")* recoge en dos variables los parámetros a trazar (*param1* y *param2)* y los parámetros temporales (*fecha0, hora0, fecha1* y *hora1)* y se utilizan en la función *obtainID(string ,string ,string, string, mysqlpp::Query )* con el objetivo de obtener el *ID* de inicio y el *ID* de final del rango de registros a examinar.

Este rango se emplea en la función *jsonArray(string ,string ,string ,string ,string ,string , mysqlpp::Query*), la cual retorna un *string* con estructura *jsonArray* de la siguiente forma:

```
{"grafica":[
   {"time":"T0", "param1":"A0", "param2":"B0"}, //1- Elemento
   {"time":"T1", "param1":"A1", "param2":"B1"}, //2 - Elemento
  …
   {"time":"Tn", "param1":"An", "param2":"Bn"}//n- Elemento
 ] }
```

Se rellenan las componentes del vector *JSON* a partir de los **datos temporales** y de los valores de las dos **variables** seleccionadas. En el caso de que el usuario solo haya solicitado un parámetro, el segundo parámetro permanecerá vacio.

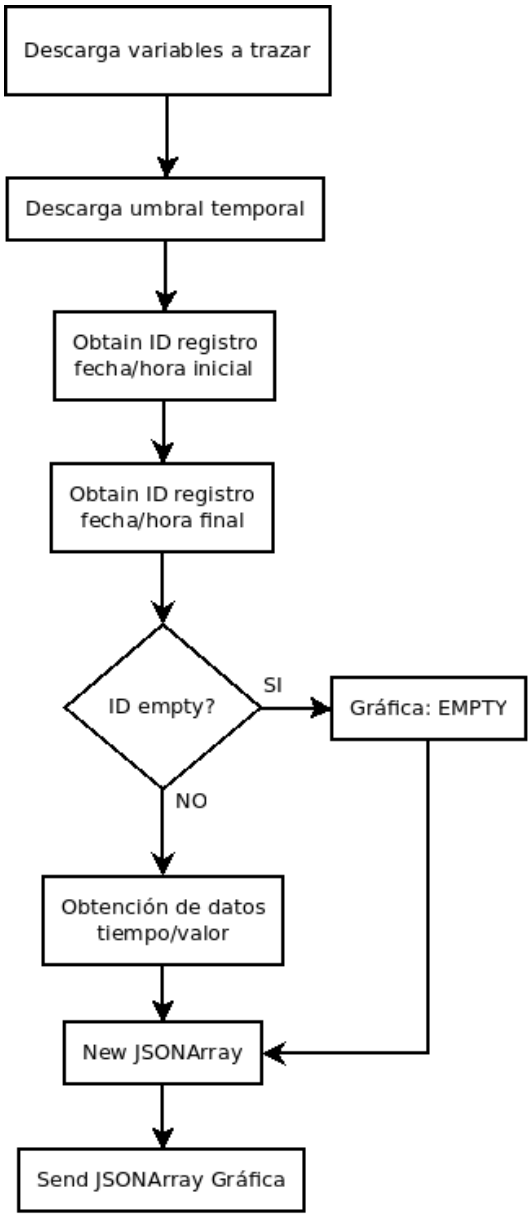



Figura 40: Diagrama de flujo GET Grafica

### 4.2.9. APLICACIÓN USERSICRA

Con el fin de poder gestionar los usuarios del sistema *ICRA*, se ha desarrollado una pequeña aplicación denominada *usersICRA*.

Dicha aplicación ofrece una interfaz de control de registros en la tabla *users*, ubicada dentro de la base de datos a emplear en la medición del sistema. Se muestra el diagrama de flujo de la aplicación *(Ver "Figura 41").*

La función *Inicio()* establece conexión con la base de datos *MySQL* del servidor y solicita al usuario insertar una *base de datos* para trabajar, la cual contendrá la tabla *users* con el registro de usuarios. Tras ello, se muestra el listado de opciones disponibles por pantalla y se evalúa la opción elegida mediante una sentencia *switch-case*.

Es requisito, para poder utilizar la aplicación *ICRApp* el disponer de un usuario registrado en la tabla *users* mediante esta aplicación.

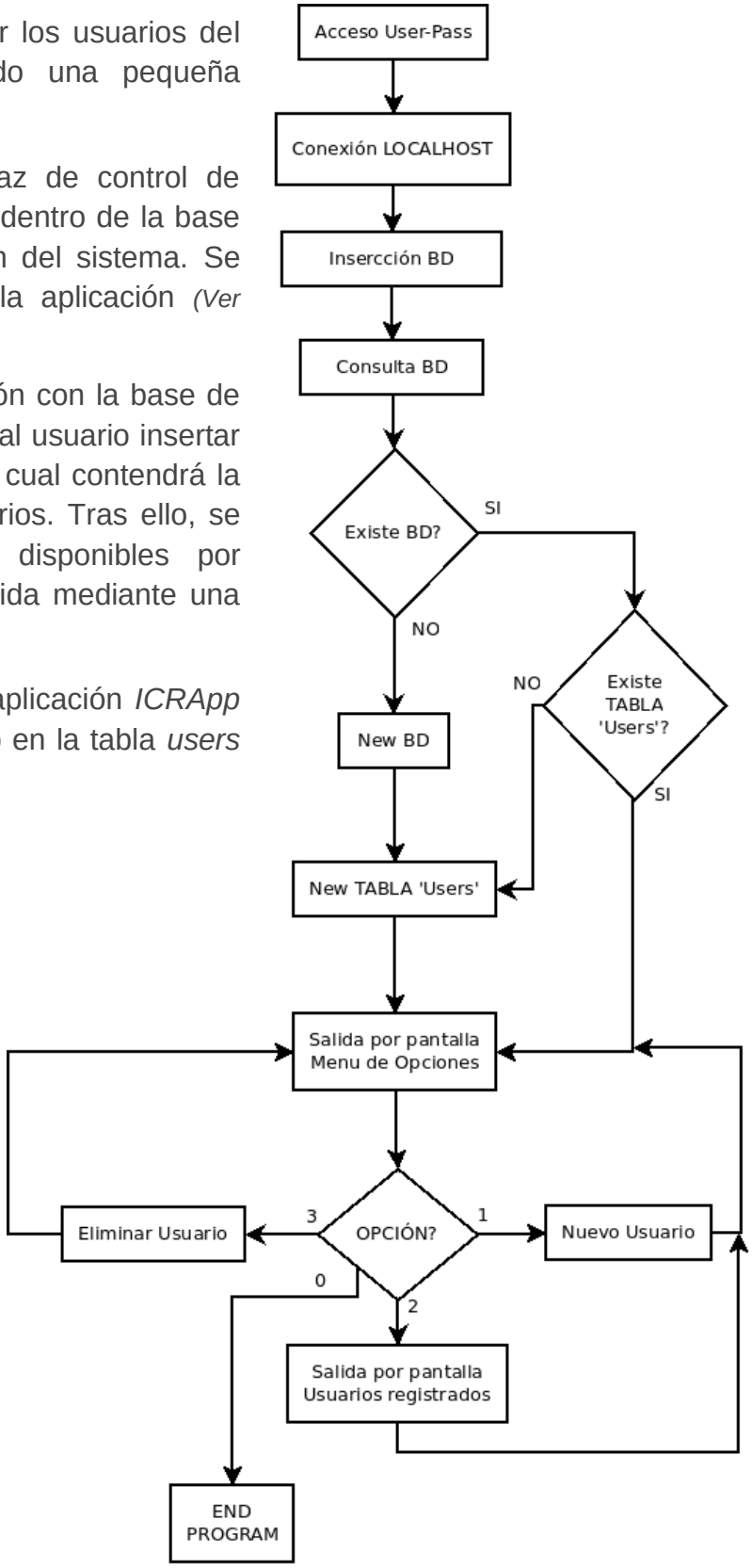
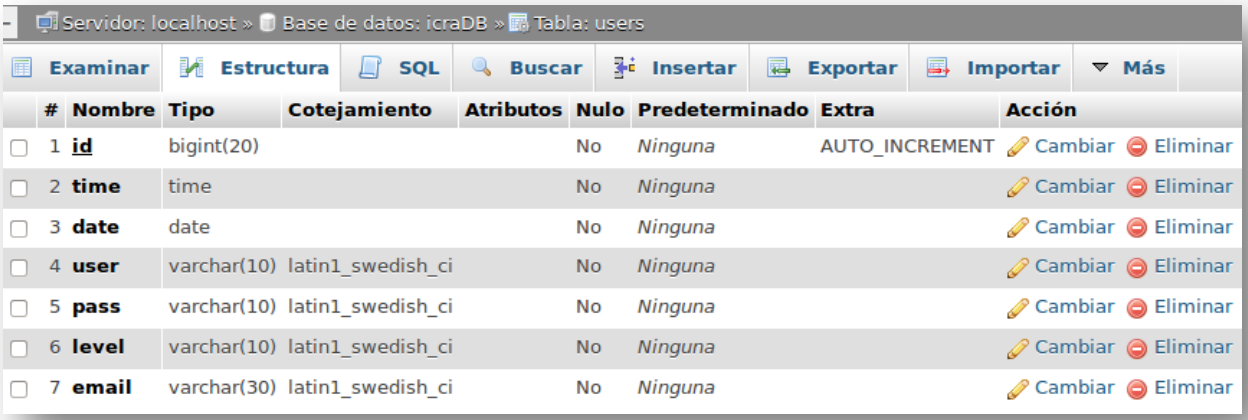



**Figura 41: Diagrama de flujo UsersICRA**

La tabla *users* mantiene una estructura como la mostrada en la *Figura 42.*




| # | Nombre | Tipo | Cotejamiento | Atributos | Nulo | Predeterminado | Extra | Acción |
|---|---|---|---|---|---|---|---|---|
| 1 | id | bigint(20) | | | No | *Ninguna* | AUTO_INCREMENT | Cambiar Eliminar |
| 2 | time | time | | | No | *Ninguna* | | Cambiar Eliminar |
| 3 | date | date | | | No | *Ninguna* | | Cambiar Eliminar |
| 4 | user | varchar(10) | latin1_swedish_ci | | No | *Ninguna* | | Cambiar Eliminar |
| 5 | pass | varchar(10) | latin1_swedish_ci | | No | *Ninguna* | | Cambiar Eliminar |
| 6 | level | varchar(10) | latin1_swedish_ci | | No | *Ninguna* | | Cambiar Eliminar |
| 7 | email | varchar(30) | latin1_swedish_ci | | No | *Ninguna* | | Cambiar Eliminar |

Figura 42: Estructura de tabla users en PhpMyAdmin

Donde cada campo tiene el siguiente significado:

1- **id**: Identificador clave de la tabla
2- **time**: Hora en la que se registro al usuario
3- **date**: Fecha en la que se registro al usuario
4- **user:** Nombre del usuario
5- **pass**: Contraseña del usuario
6- **level**: Nivel de privilegios del usuario
7- **email**: Dirección de correo electrónico del usuario

La información guardada en el campo *email* no se utiliza en este trabajo, pero se ha reservado una columna en la tabla con el fin de emplearlo en futuras ampliaciones del proyecto.

Para más información sobre la aplicación *usersICRA* ver "*ANEXO V: Manual Usuario usersICRA*"

## 4.3. Desarrollo de la aplicación móvil

El diseño y desarrollo de la aplicación de monitorización remota se ha implementado bajo el sistema operativo *Android.*

*Android* es un sistema operativo móvil que se basa en una versión modificada de *Linux*. En 2005, como parte de su estrategia para entrar en el mundo móvil, *Google* compró *Android* y se hizo cargo de su trabajo de desarrollo (así como de su equipo).

*Google* quería que *Android* fuera abierto y libre; de ahí que la mayoría del código *Android* se pusiera disponible bajo licencia *Apache* de código abierto, lo que significa que cualquiera que quiera utilizar *Android* puede hacerlo al descargar su código fuente completo.

La principal ventaja de adoptar *Android* es que ofrece un **enfoque unificado para el desarrollo de aplicaciones.** Los desarrolladores solamente necesitan desarrollar para *Android*, y sus aplicaciones deberían poderse ejecutar en numerosos dispositivos, siempre y cuando los dispositivos utilicen este sistema operativo.

Para entender cómo funciona *Android* ver la *Figura 43*, que muestra los diferentes niveles que componen el sistema operativo. (5)

El sistema operativo se divide en cinco secciones, de cuatro niveles principales.

- **Kernel de Linux**: Contiene los controladores de dispositivo de bajo nivel para los diversos componentes de hardware.
- **Librerías:** Contienen todo el código que proporciona las principales características de un sistema operativo *Android* (SQLite para soporte de base de datos, WebKit proporciona funcionalidades para navegación Web).
- **Tiempo de ejecución Android**: En el mismo nivel que las librerías, permite a los desarrolladores escribir aplicaciones *Android* al utilizar el lenguaje de programación Java. También incluye la máquina virtual Dalvik, que permite que toda aplicación *Android* se ejecute en su propio proceso.
- **Arquitectura de software de aplicación**: Presenta las diferentes posibilidades de uso del sistema operativo *Android* para los desarrolladores de aplicaciones, de modo que pueden hacer uso de ellas en sus aplicaciones.
- **Aplicaciones**: En su capa superior se encuentran las aplicaciones que se distribuyen con el dispositivo *Android*, al igual que aplicaciones que se descargan e instalan desde Google Play.

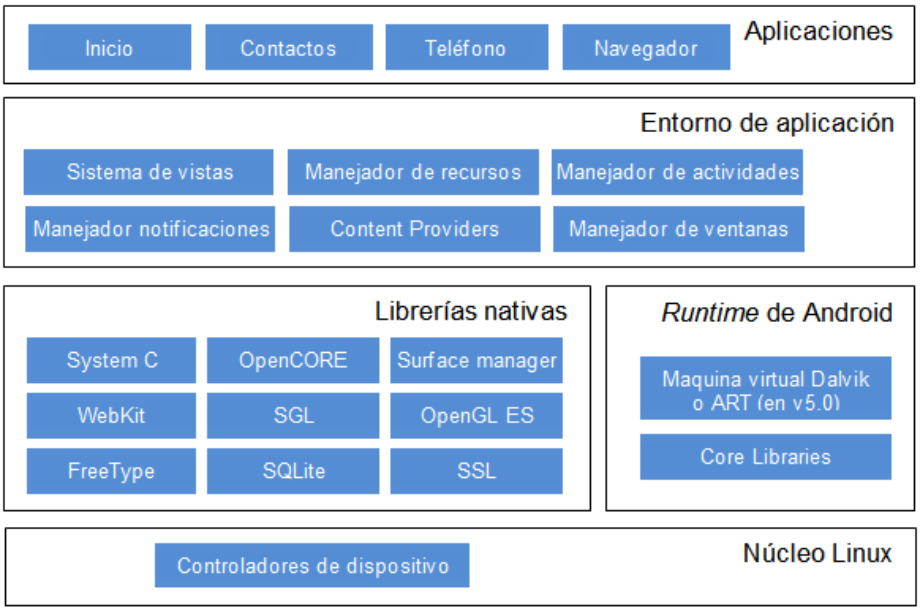



Figura 43: Arquitectura de Android

Para el desarrollo de aplicaciones en fundamental emplear el **Android SDK**, el cual incluye un conjunto de herramientas de desarrollo. Comprende un depurador de código, biblioteca, un simulador de teléfono basado en *QEMU*, documentación, ejemplos de código y tutoriales.

Una vez se procede a desarrollar una aplicación otro aspecto a considerar es la versión de la **API** a emplear. La plataforma *Android* ofrece una *API* (Interfaz de programación de aplicaciones) que las aplicaciones pueden utilizar para interactuar con la base del sistema *Android*. Esta *API* se compone de diversos elementos, tales como un conjunto de paquetes, clases, atributos *XML, Intents*, permisos que las aplicaciones pueden solicitar…

En cada versión sucesiva de la plataforma *Android* *(Ver "Tabla 7")* se incluyen cambios y actualizaciones de la *API*. Estos cambios están diseñados de manera que cada nueva *API* sigue siendo compatible con las versiones anteriores. Es decir, la mayoría de los cambios que se realizan en la *API* son aditivos, introduciendo nuevas funcionalidades. Al actualizar, ciertas partes de la *API* quedan remplazadas por las nuevas, quedan obsoletas, pero aun así, no se borran, por lo que las aplicaciones existentes pueden seguir. (9)

Tabla 7: Versiones de la API de Android

| Versión | Nombre en código | Fecha de distribución | API level | Cuota (3 de noviembre, 2014) |
|---|---|---|---|---|
| **5.1** | *Lollipop* | 06 de abril de 2015 | 22 | 0,7% |
| **5.0** | Lollipop | 3 de noviembre de 2014 | 21 | 9,0% |
| **4.4** | Kit Kat | 31 de octubre de 2013 | 19 | 39,8% |
| **4.3** | Jelly Bean | 24 de julio de 2013 | 18 | 5,5% |
| **4.2** | Jelly Bean | 13 de noviembre de 2012 | 17 | 18,1% |
| **4.1** | Jelly Bean | 9 de julio de 2012 | 16 | 15,6% |
| **4.0** | Ice Cream Sandwich | 16 de diciembre de 2011 | 14 | 5,3% |
| **2.3** | Gingerbread | 9 de febrero de 2011 | 10 | 5,7% |
| **2.2** | Froyo | 20 de mayo de 2010 | 8 | 0,3% |

Existen una serie de elementos clave que resultan imprescindibles de conocer y utilizar para desarrollar aplicaciones en *Android* (15).

- **View**: Las vistas son los elementos que componen la interfaz de usuario de una aplicación. Son por ejemplo, un botón, una entrada de texto,... Todas las vistas van a ser objetos descendientes de la clase *View (Ver "Figura 44"),* y por tanto, pueden ser definidos utilizando código *Java.* Sin embargo, lo habitual va a ser definir las vistas utilizando un fichero *XML* y dejar que el sistema cree los objetos automáticamente a partir de este fichero. Esta forma de trabajar es muy similar a la definición de una página web utilizando código *HTML*.

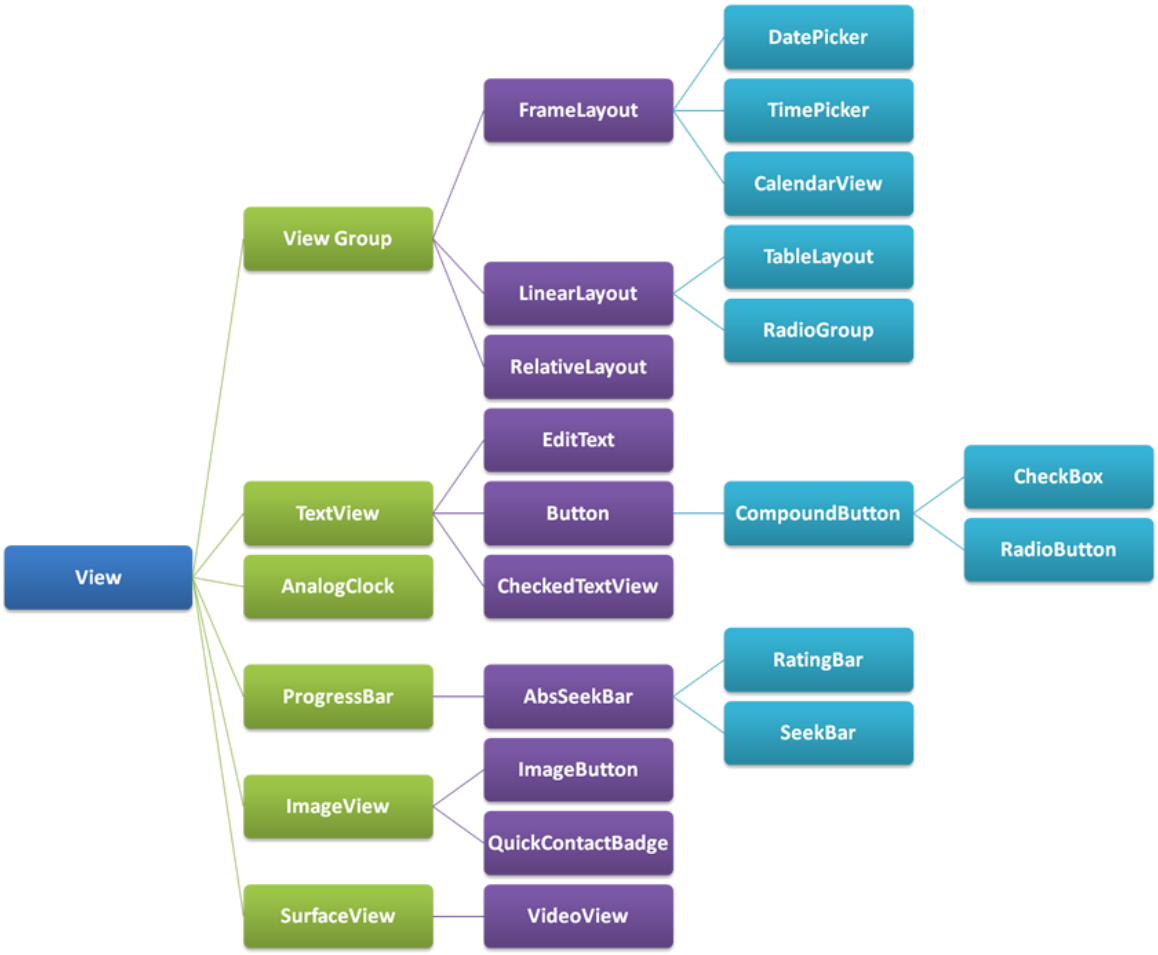



Figura 44: Jerarquía View

- **Layout**: Un *Layout* es un conjunto de vistas agrupadas de una determinada forma. Se dispone de diferentes tipos de *Layouts* para organizar las vistas de forma lineal, en cuadrícula o indicando la posición absoluta de cada vista. Los *Layouts* también son objetos descendientes de la clase *View*, y al igual que las vistas, los *Layouts* pueden ser definidos en código, aunque la forma habitual de definirlos es utilizando código *XML*.

- **Activity**: Una aplicación en *Android* va a estar formada por un conjunto de elementos básicos de visualización, coloquialmente conocidos como pantallas de la aplicación. En *Android* cada uno de estos elementos, o pantallas, se conoce como

*actividad* y su función principal es la creación del *intertaz* de usuario *(Ver "Figura 45")*. Una aplicación suele necesitar varias actividades para crear el interfaz de usuario, siendo independientes entre sí, aunque todas trabajarán para un objetivo común. Toda actividad ha de pertenecer a una clase descendiente de *Activity*.

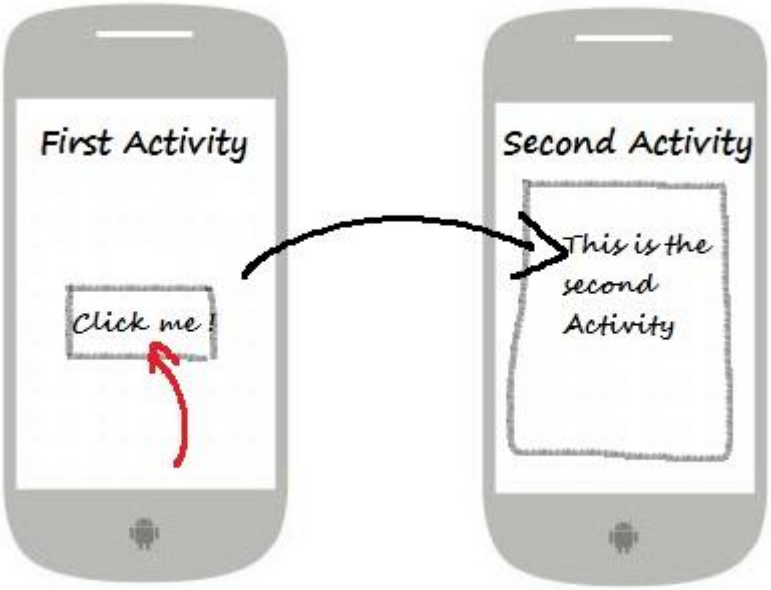



**Figura 45: Concepto de Activity**

Para crear una nueva actividad es necesario crear una clase *Java* que amplíe la clase base *Activity* y declarar esta en el archivo *AndroidManifest.xml*[18]

Hecho esto, la nueva clase definida goza de una serie de métodos propios de una *Activity*, los cuales pueden ser sobrescritos para llevar a cabo tareas específicas *(Ver Figura 46").* Mencionar brevemente el cometido de cada uno:

- *onCreate():* Llamado cuando la actividad se crea por primera vez
- *onStart():* Llamado cuando la actividad se encuentra visible para el usuario
- *onResume():* Llamado cuando la actividad empieza a interactuar con el usuario
- *onPause():* Llamado cuando la actividad actual se detiene y la actividad anterior se reanuda.
- *onStop():* Llamado cuando la actividad ya no es visible para el usuario.
- *onDestroy():* Llamado antes de que el sistema destruya la actividad (tanto manual como automáticamente para conservar memoria).[19]
- *onRestart():* Llamado cuando la actividad se ha detenido y se reinicia de nuevo.

[18] Archivo que especifica los permisos que necesita la aplicación, así como otras características (como filtros)
[19] Una actividad se destruye cuando se hace clic en el botón **Atrás**.

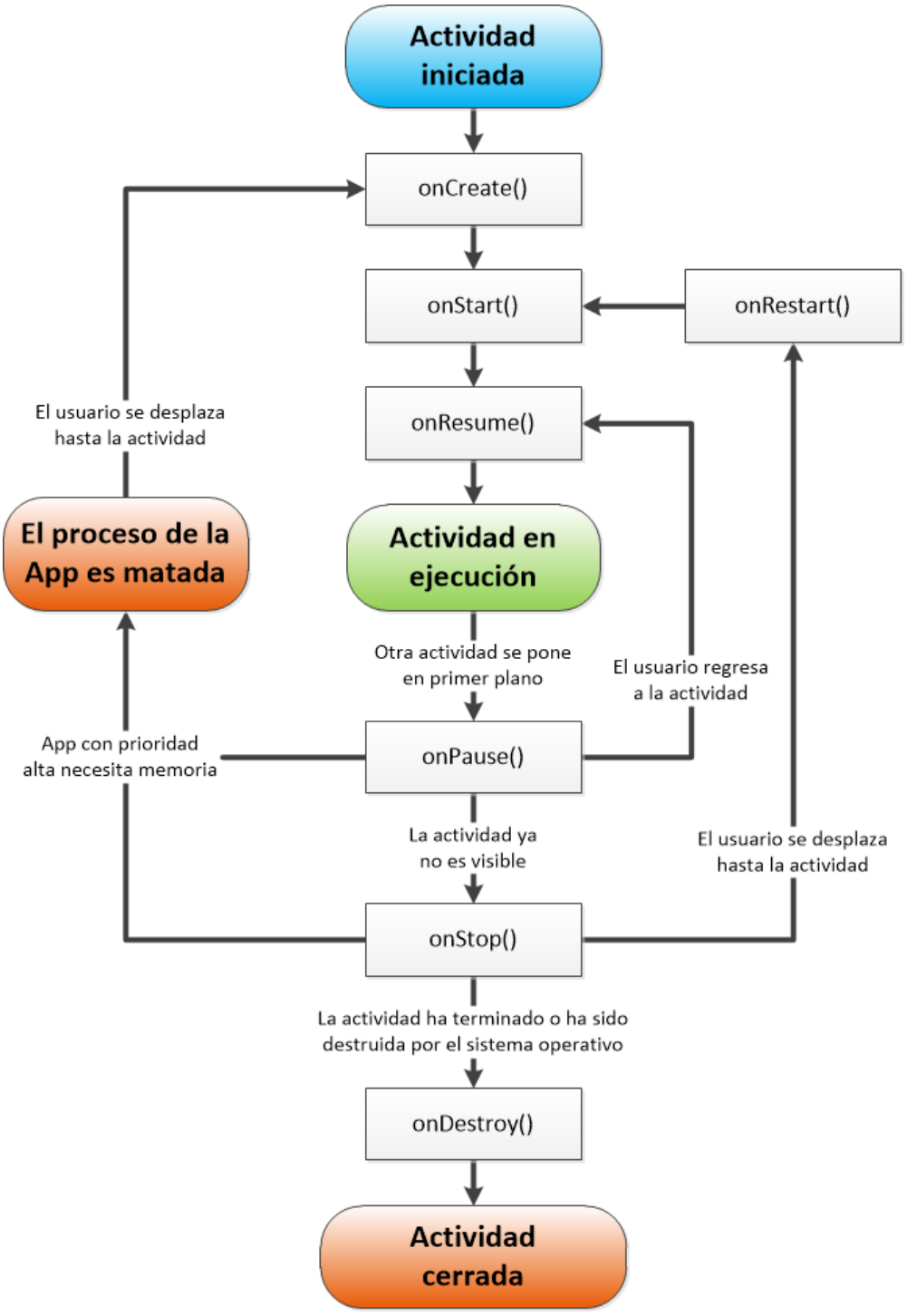



Figura 46: Ciclo de vida de Activity

- **Service**: Un servicio es un proceso que se ejecuta "detrás", sin la necesidad de una interacción con el usuario. Es algo parecido a un demonio en *Unix* o a un servicio en *Windows*. En *Android* se dispone de dos tipos de servicios: servicios locales, que pueden ser utilizados por aplicaciones del mismo terminal y servicios remotos, que pueden ser utilizados desde otros terminales.

- **Intent**: Una intención representa la voluntad de realizar alguna acción; como realizar una llamada de teléfono, visualizar una página web. Se utiliza cada vez que se quiere:
    - Lanzar una actividad.
    - Lanzar un servicio.
    - Lanzar un anuncio de tipo broadcast.
    - Comunicar con un servicio

  Los componentes lanzados pueden ser internos o externos a la aplicación. También se utilizan las intenciones para el intercambio de información entre estos componentes *(Ver "Figura 47").*

  En muchas ocasiones una intención no será inicializada por la aplicación, si no por el sistema, por ejemplo, cuando se pide visualizar una página web. En otras ocasiones será necesario que la aplicación inicialice su propia intención. Para ello se creará un objeto de la clase *Intent*.

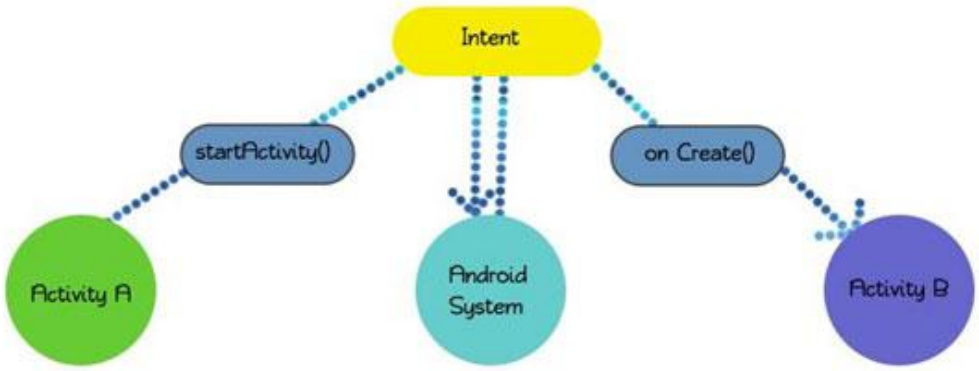



**Figura 47: Concepto de Intent entre Activities**

- **Broadcast Receiver**: Un receptor de anuncios recibe y reacciona ante anuncios de tipo broadcast. Existen muchos originados por el sistema, como por ejemplo *Batería baja*, *Llamada entrante*,... Aunque, las aplicaciones también puede lanzar un anuncio broadcast. No tienen interfaz de usuario, aunque pueden iniciar una actividad para atender a un anuncio.

Una vez descritas y conocidas las herramientas básicas que permiten crear una aplicación *Android*, es necesario emplear un entorno de programación capaz de compilar el código de la *app*, y por consiguiente, obtener el fichero instalador .**apk** de la aplicación.

La aplicación **ICRApp** ha sido desarrollada bajo el software **Eclipse** [20], programa informático compuesto por un conjunto de herramientas de programación de código abierto multiplataforma para desarrollar "*Aplicaciones de Cliente Enriquecido*".

Gracias al plugin **ADT** para el **IDE Eclipse** *(Ver "Figura 48")*, se amplían las capacidades de *Eclipse* para poder configurar rápidamente nuevos proyectos para *Android*, crear una interfaz de usuario de la aplicación, agregar paquetes basados en la *API* de *Android Framework*, depurar aplicaciones usando el *SDK* de *Android Tools*, e incluso exportar con firma (o sin frima) archivos *.apk* con el fin de distribuir la aplicación.


**Figura 48: ADT de Eclipse**

Esta aplicación *Android* se ha desarrollado bajo el nivel de **API 21**, el cual corresponde a la versión de *Android 5.0 LOLLIPOP*, lanzada su versión estable el 3 de noviembre de 2014 *(Ver Figura 49").*

**Figura 49: Android 5.0 LOLLIPOP**

[20] Para más información sobre el software *Eclipse* y el plugin *ADT*, visitar http://developer.*Android*.com

### 4.3.1. INTERFAZ DE USUARIO

Las diferentes interfaces de la aplicación se determinan a partir de **ficheros *XML***, los cuales contienen los diferentes *layout* que conforman las pantallas de la *app*. Como se ha mencionado en el punto anterior, los objetos *layout* permiten integrar diferentes elementos de interacción con el usuario: textos, botones, imágenes…

La siguiente Figura muestra el conjunto de ficheros *XML* ubicados en el directorio *layout* que constituyen las interfaces gráficas:

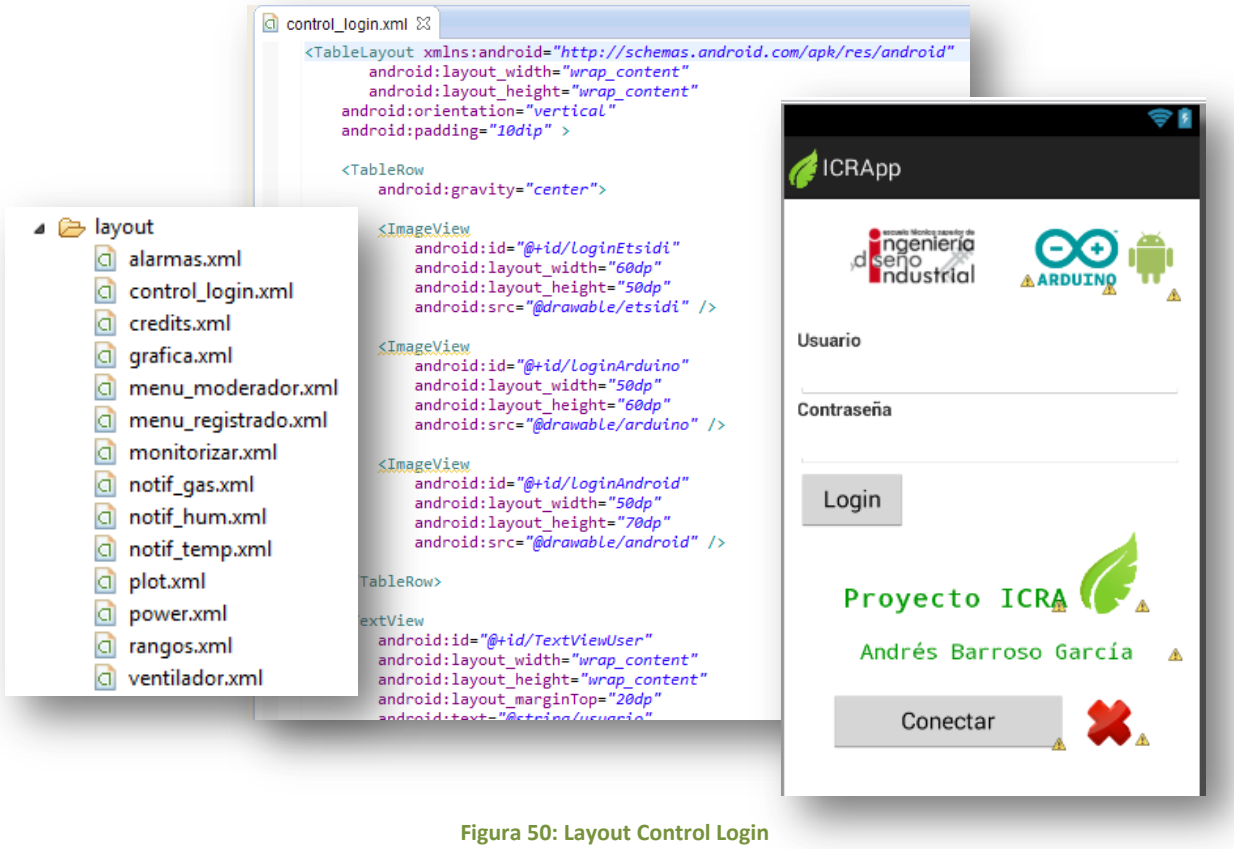



**Figura 50: Layout Control Login**

La "*Figura 50"* muestra el resultado de programar el fichero inicial *control_login.xml*, el cual representa la interfaz de identificación al lanzar la aplicación.

Para comprender mejor el funcionamiento de un fichero de estas características, se muestra un fragmento del código *control_login.xml:*

```
…
<TextView
    Android:id="@+id/TextViewUser"
    Android:layout_width="wrap_content"
    Android:layout_height="wrap_content"
    Android:layout_marginTop="20dp"
    Android:text="@string/usuario"
    Android:textStyle="bold" />
<EditText
    Android:id="@+id/TxtUsuario"
    Android:layout_width="match_parent"
    Android:layout_height="wrap_content"
    Android:inputType="text" />
<TextView
    Android:id="@+id/TextViewPass"
    Android:layout_width="wrap_content"
    Android:layout_height="wrap_content"
    Android:text="@string/contrasena"
    Android:textStyle="bold" />
<EditText
    Android:id="@+id/TxtPassword"
    Android:layout_width="match_parent"
    Android:layout_height="wrap_content"
    Android:inputType="numberPassword"
    Android:maxLength="4" />
<LinearLayout
    Android:layout_width="162dp"
    Android:layout_height="wrap_content"
    Android:orientation="horizontal" >
    <Button
        Android:id="@+id/BtnLogin"
        Android:layout_width="wrap_content"
        Android:layout_height="wrap_content"
        Android:paddingLeft="20dip"
        Android:paddingRight="20dip"
        Android:text="@string/login" />
    <TextView
        Android:id="@+id/LblMensaje"
        Android:layout_width="wrap_content"
        Android:layout_height="wrap_content"
        Android:paddingLeft="10dip"
        Android:textStyle="bold" />
</LinearLayout>
…
```

El primer elemento es un *TextView* que muestra la palabra contenida en la variable *string* denominada *usuario*, en este caso: **Usuario**.

Uno de los atributos fundamentales a tener en cuenta al desarrollar un fichero *XML* es `Android:id` ,el cual permite identificar los elemento desde los ficheros *.java*, necesario para poder adquirir o modificar su contenido de forma dinámica.

El siguiente elemento, *EditText*, permite al usuario introducir una sentencia de caracteres, recogiendo con ello los caracteres insertados mediante el teclado táctil. El contenido de este elemento se interpretará como el nombre del usuario que desee logearse.

El siguiente par de elementos, *TextView* y *EditText* son análogos a los anteriores vistos, pero en esta ocasión para insertar la **Contraseña**.

Apreciar que el segundo elemento *EditText* cuenta con dos atributos diferentes:

```
Android:inputType="numberPassword"
Android:maxLength="4"
```

El primero hace que se muestre un teclado numérico en la pantalla, en lugar del teclado básico compuesto por letras y números, y el segundo atributo restringe la cantidad máxima de cifras insertadas a 4.

Tras estos cuatro elementos, se define un objeto `<LinearLayout>`, ya que se desea insertar dos objetos que se encuentren uno al lado del otro de forma horizontal. Para ello se utiliza la característica `Android:orientation="horizontal`. Estos dos elementos son: un objeto `<Button>` y otro `<TextView>`. El primero servirá como lanzador del evento de evaluación de los parámetros insertados. Hay varias formas de llevar a cabo esta tarea, una de ellas sería emplear un atributo de la clase *Button* denominado `Android:onClick` , el cual lanza la función definida al efectuar una pulsación sobre el objeto desde la pantalla. En este caso se ha optado por otra opción, que es evaluar el estado del *Button* desde un fichero *.java* empleando su atributo `Android:id.` El elemento *TextView* ubicado a la derecha del *Button* sirve como indicador de avisos, mostrando mensajes al usuario del estado de su identificación.

Como se ha mencionado, los **ficheros *XML*** contienen la **estructura visual de la aplicación**, por lo que carecen de lógica o algoritmos de cálculo. Estos se ubican en los ficheros *.java*, los cuales se vinculan con los elementos gráficos a través de sus identificadores. Los **ficheros *.java*** son los responsables de inflar el *layout* que corresponda en cada situación y por tanto, **controlar la ventana de interacción** entre el usuario y el dispositivo.

El siguiente esquema *(Ver Figura 51)* muestra el árbol de salto entre los diferentes *layout* de la *app.*

El próximo punto del capítulo aborda la estructura de las *activies* y su mecanismo interno de funcionamiento, el cual permite llevar a cabo el vínculo entre los diferentes *layout* y la información que estos contienen.

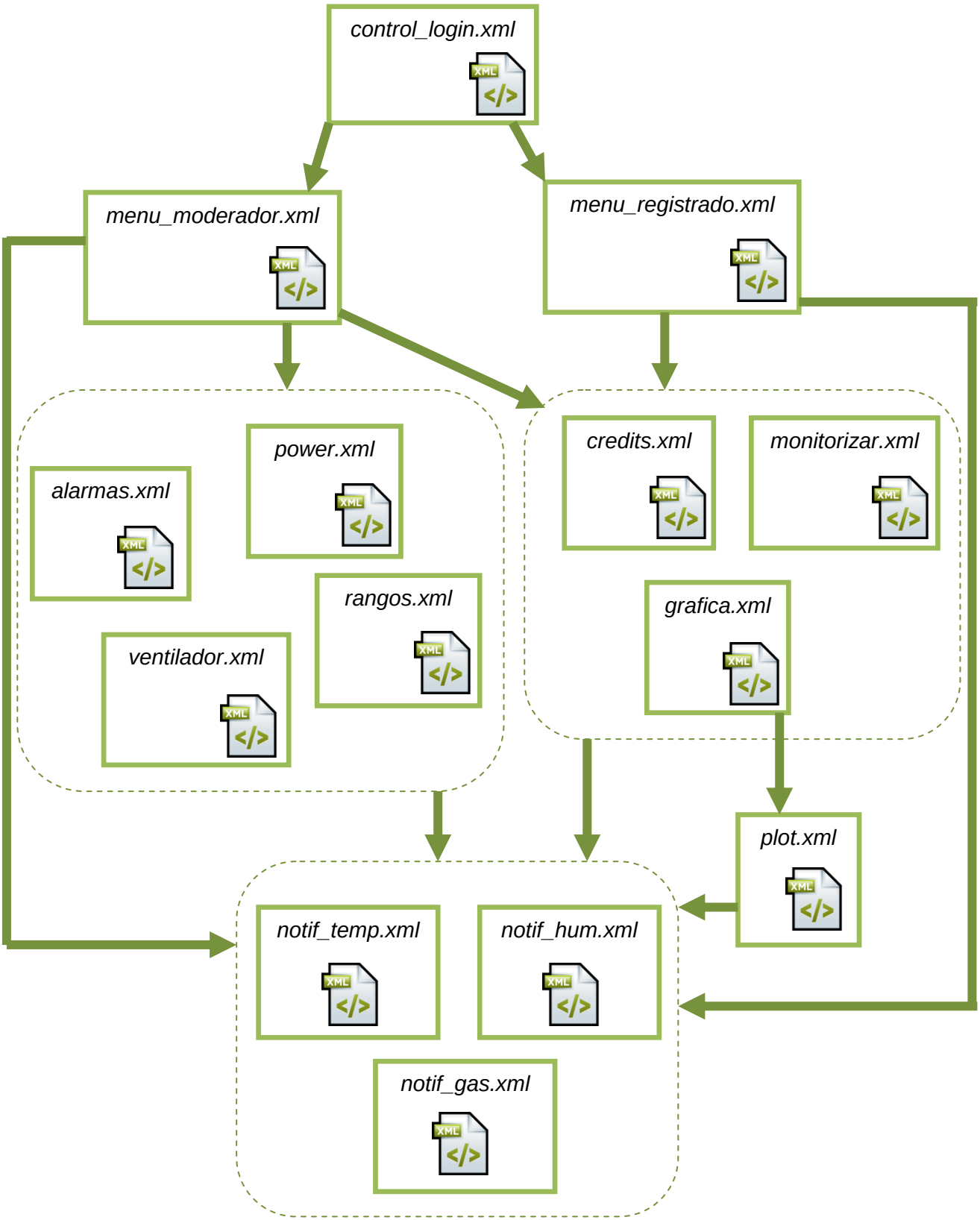



Figura 51: Jerarquía de layout

### 4.3.2. ACTIVITIES

Antes de estudiar cada *activity* del proyecto, es necesario describir un par de conceptos fundamentales para el desarrollo de la aplicación.

La clase ***AsyncTask*** es una clase de ayuda para soportar programación concurrente que requiera interacción con la *IGU*, por lo que un objeto *AsyncTask* ejecuta una operación de forma concurrente al hilo de la *IU* en un hilo en segundo plano *(Ver Figura 52)*. Además, coopera de forma estrecha con el hilo de la *IU* (16):

- Toma parámetro del hilo de la *IU*
- Informa al hilo de la *IU* sobre el progreso de la operación
- Devuelve los resultados al hilo de la *IU*

Esta clase proporciona tres tipos genéricos (es decir, marcadores para los tipos de deben ser reemplazados por tipos reales) para:

- Los parámetros recibidos desde el hilo de la *IU*
- Los valores informando sobre el progreso de la operación
- El resultado devuelto  al hilo de *IU*

Se dispone de métodos para cada paso de ejecución individual, como:

- *doInBackground ( ):* define la operación a ejecutar en segundo plano.
- Otros métodos para controlar operaciones de cooperación entre el hilo de la *IU* y el hilo en segundo plano.

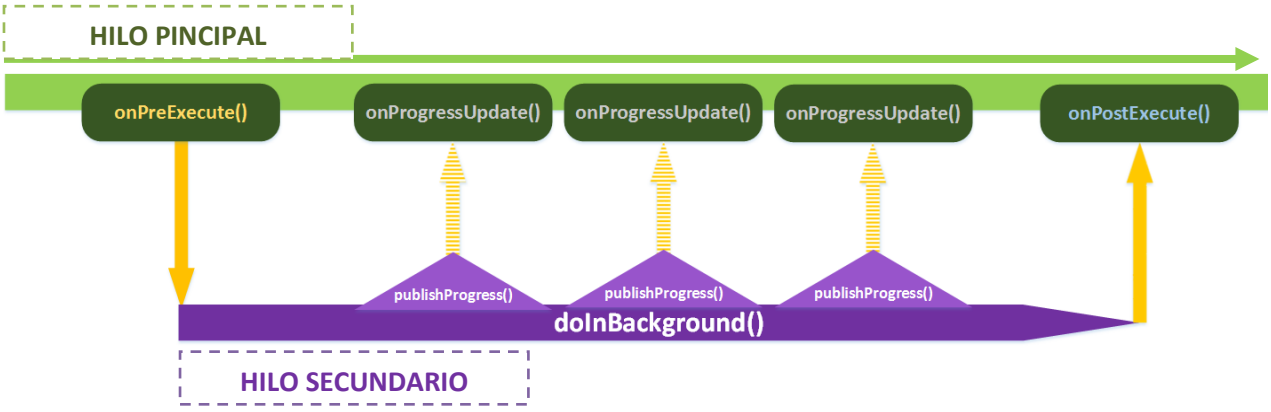



Figura 52: Hilo secundario de proceso

Como se mencionó previamente, en el punto *"4.2.8 Comunicación Servidor-Android ",* para llevar a cabo la comunicación entre el equipo servidor y los clientes *Android* se hace uso de la herramienta *JSON*.

La simplicidad de ***JSON*** ha dado lugar a la generalización de su uso, especialmente como alternativa a *XML* en *AJAX*. Una de las supuestas ventajas de *JSON* sobre *XML,* como formato de intercambio de datos en este contexto, es que es mucho más sencillo escribir un analizador sintáctico (*parser*) de *JSON*.

Una de las mayores ventajas que tiene el uso de *JSON* es que puede ser leído por **cualquier lenguaje de programación**, por lo tanto, puede ser usado para el intercambio de información entre distintas tecnologías. (17)

La estructura de un objeto *JSON* es la siguiente *(Ver Figura 53)*:

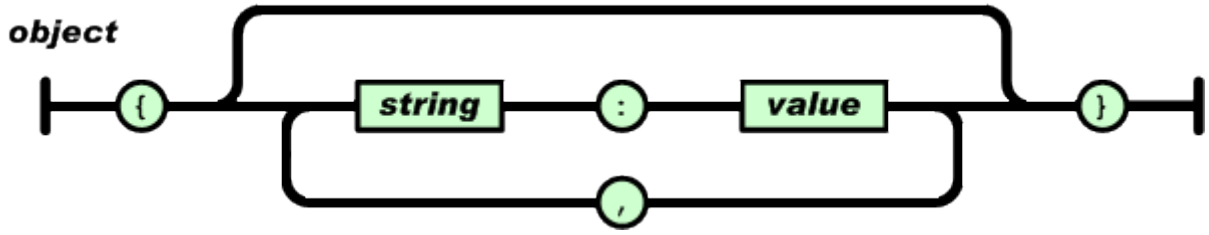



Figura 53: Objeto JSON

Un objeto es un conjunto desordenado de pares nombre / valor. Un objeto comienza con "{" (llave izquierda) y termina con "} "(llave derecha). Cada nombre es seguido por ":" (dos puntos) y el "nombre / pares de valores" están separados por "," (coma).

Es posible coser una *array* de objetos *JSON*, lo cual aumenta las posibilidades de la aplicación ya que permite enviar una gran cantidad de información en una sola variable *(Ver Figura 54)*

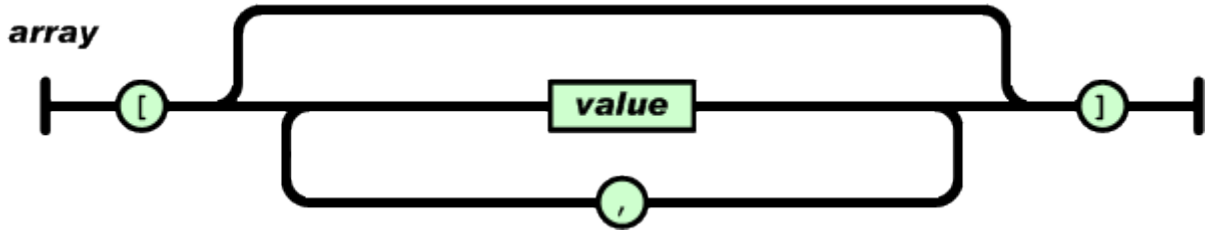



Figura 54: Array JSON

Un *array* es una colección ordenada de valores. Una serie comienza con "[" (corchete izquierdo) y termina con "]" (corchete derecho). Los valores están separados por "," (coma).

Tras esta breve explicación se procede a ver las diferentes *activities* que integran la aplicación *ICRApp*

#### 4.3.2.1. LOGIN DE IDENTIFICACIÓN

La clase *Login* es la encargada de **lanzar la *activity* inicial** de la *app*, por lo que es necesario declarar la siguiente sentencia en el archivo *AndroiManifest.xml*

```
<activity
   Android:name="com.andres.usingintent.Login"
   Android:label="@string/app_name" >
   <intent-filter>
       <action Android:name="android.intent.action.MAIN" />

       <category Android:name="android.intent.category.LAUNCHER" />
   </intent-filter>
 </activity>
```

Toda clase que se declare ha de estar registrada en este fichero de una forma similar a la mostrada para la clase *Login*, a excepción de la sentencia:

```
<category Android:name="android.intent.category.LAUNCHER" />
```

la cual identifica a la *activity* como lanzadora de la aplicación al ser arrancada.

A continuación se analizan algunas de las principales características de la clase:

```
public class Login extends Activity{
       …
       protected void onCreate(Bundle savedInstanceState) {

              super.onCreate(savedInstanceState);
              setContentView(R.layout.control_login);

              …

              }//Fin Login
```

La primera tarea que lleva a cabo la *activity* al ser ejecutada es montar el *layout* correspondiente, en este caso el fichero *control_login.xml* visto en el punto anterior. Esta interfaz permite al usuario interactuar con la *activity*, en este caso con el objetivo de **logearse** en el sistema y tener acceso a la aplicación.

Para ello, la *activity* evalúa el *usuario* y *password* introducidos de la siguiente forma:

```
public class Login extends Activity{
	…
	protected void onCreate(Bundle savedInstanceState) {
		…
	//Gestion de conexion servidor
	TareaBitLife tareaBitLife = new TareaBitLife();
	tareaBitLife.execute();
	//Solicitar identificacion
	BtnLogin.setOnClickListener(new OnClickListener() {
		@Override
		public void onClick(View v) {
			String aux_user=EdtTxtuser.getText().toString();
			String aux_pass=EdtTxtpass.getText().toString();

			if(aux_user.equals("")||aux_pass.equals(""))
				TxtVwLblMensaje.setText("Rellene los campos");

			else{
				TareaLogin tareaLogin = new TareaLogin();
				tareaLogin.execute(aux_user,aux_pass);
			}
		}});
	//Establecer conexion servidor
	BtnConectar.setOnClickListener(new OnClickListener() {
		@Override
		public void onClick(View v) {
			TareaBitLife tareaBtnBitLife = new TareaBitLife();
			tareaBtnBitLife.execute();
		}});
	}//Fin onCreate
}//Fin Login
```

Se crea una instancia de la clase *TareaBitLife*, la cual extiende de la clase *AsyncTask,* y se ejecuta el hilo secundario para **consultar el estado de operatividad del servidor**.

Se muestra brevemente la esencia de esta tarea para comprender el proceso de solicitud *GET* que van a compartir el resto de tareas secundarias en la aplicación:

```
private class TareaBitLife extends AsyncTask<String,Integer,Boolean> {
    protected Boolean doInBackground(String... params) {
	boolean resul = true;
	HttpClient httpClient = new DefaultHttpClient();//Cliente HTTP
	HttpGet del =new HttpGet(getString(R.string.Server_IP)+
		getString(R.string.Server_Port)+"/life");//Solicitud GET
	del.setHeader("content-type", "application/json");
	try{
		HttpResponse resp = httpClient.execute(del);
		String respStr = EntityUtils.toString(resp.getEntity());
		JSONObject respJSON = new JSONObject(respStr);//JSON respuesta
		bit=respJSON.getString("bit");//Extraccion de parametros JSON
		}
	… }//Fin TareBitLife
```

Se declara una variable del tipo *HttpGet* que contenga la *URL* necesaria para llevar a cabo la solicitud *GET*, la cual está formada por la dirección *IP* del servidor, el *puerto* de comunicación y la palabra identificador de la petición, en este caso el termino */life.*

A través del objeto *HttpClient* se lanza la solicitud, y la respuesta obtenida desde el otro lado del canal se almacena en una variable *HttpResponse*. A partir de aquí, se convierte la respuesta en una variable tipo *String* que permita formar un nuevo objeto *JSON*.

Una vez se dispone de un objeto *JSON* para trabajar, se identifica (en este caso) el parámetro *"bit"* en el mensaje y se extrae su contenido. Si el código de conexión corresponde con el esperado, se marca el estado de conexión como válido.

Esta tarea se lleva a cabo al lanzar la aplicación y en el caso de que usuario pulse el botón *Conectar*, de forma que se actualiza el **estado de conexión** por si hubiera habido algún cambio en la conexión con el servidor.

Tras insertar el par de valores usuario y contraseña, y pulsar el botón *Login*, se lanza un hilo secundario de la clase *TareaLogin* encargado de formar una solicitud *GET* con los datos insertados y de recibir el nivel de usuario correspondiente:

```
getString(R.string.Server_IP)+getString(R.string.Server_Port)+"/login/"+user+"/"+_pass)
```

Dicha tarea sigue una estructura análoga al hilo mostrado en el caso anterior, informando al usuario de la respuesta a través de la función final del hilo *onPostExecute():*

```
private class TareaLogin extends AsyncTask<String,Integer,Boolean> {
 …
 protected void onPostExecute(Boolean result) {
       if (result){

             //Limpiar EditText
             EdtTxtuser.setText("");
             EdtTxtpass.setText("");

             if(permiso.equals("empty")==false){
                   Intent i=new Intent("com.andres.usingintent.Menu");
                   i.putExtra("modo", permiso);
                   i.putExtra("user", user);
                   TxtVwLblMensaje.setText("Correcto");
                   startActivityForResult(i,1);
             }
             else   TxtVwLblMensaje.setText("Incorrecto");
       }
       else   TxtVwLblMensaje.setText("Sin conexión");
 }}//Fin TareLogin
```

En el caso de un resultado distinto de *empty*, se crea un nuevo objeto *intent* para lanzar la siguiente *activity*, y se le añade dos parámetros: el **nombre de usuario** y su nivel de **permiso**. Tras ello, se informa al usuario de que se ha logeado correctamente a través del elemento *TxtView* y se lanza la siguiente *activity*.

#### 4.3.2.2. MENÚ DE USUARIO

La clase *Menu* es la encargada de montar la interfaz de acceso a las herramientas de la aplicación.

La prima tarea que recae en esta *activity* es evaluar el parámetro **permiso** enviado por la *activity* previa para decidir que *layout* tiene que inflar.

```
public class Menu extends Activity {
   …
   protected void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);
          String modo=getIntent().getStringExtra("modo");//Nivel de privilegio
          String user=getIntent().getStringExtra("user");//Usuario identificado

          if(modo.equals("ADMINISTRADOR")){

                setContentView(R.layout.menu_moderador);
                ImgVwServer=(ImageView)findViewById
                (R.id.conexionMenuModerador);
          }
          else {

                setContentView(R.layout.menu_registrado);
                ImgVwServer=(ImageView)findViewById
                (R.id.conexionMenuRegistrado);
          }
   … }//Fin Menu
```

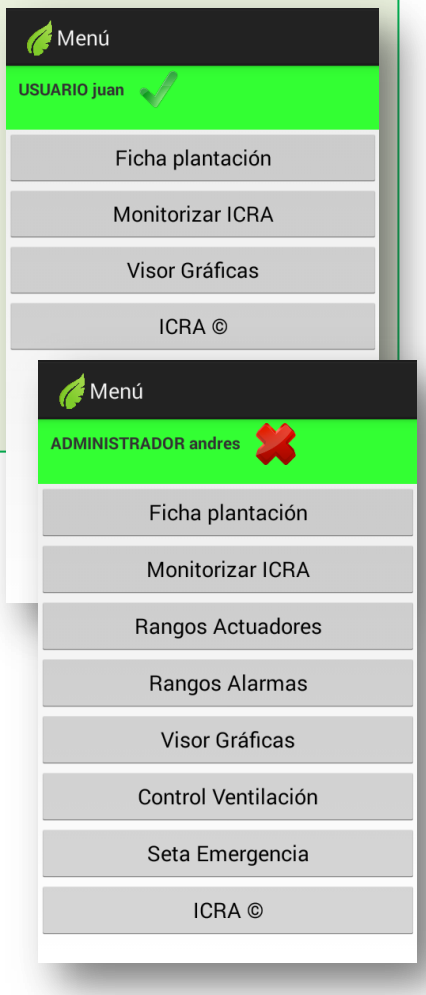



Figura 55: Layout Menus

Si el nivel de permiso es *ADMINISTRADOR*, se carga el *layout* de dicho modo, y en el cualquier otro caso se monta por defecto el *layout* de usuario *USUARIO*.

Se aprecia rápidamente en la *Figura 55* como el *menú de administrador* ofrece más posibilidades que el menú de *usuario*.

Montado el *layout*, la *activity Menu* lanza tres hilos secundarios con tres finalidades distintas:

```
public class Menu extends Activity {
   …
   protected void onCreate(Bundle savedInstanceState) {
   …

          //Gestion de conexion servidor
          TareaBitLife tareaBitLifeMenu = new TareaBitLife();
          tareaBitLifeMenu.execute();
          //Obtencion del Identificador de fichero registro de plantacion
          TareaGetRegister tareaGetRegister=new TareaGetRegister();
          tareaGetRegister.execute();
          //Notificacion de alarmas
          TareaAlarms tareaAlarms = new TareaAlarms();
          tareaAlarms.execute();

   }//Fin OnCreate
   …
   }//Fin Menu
```

El hilo *tareaBitLifeMenu*, al igual que en la *activity* anterior, consultar el estado de **operatividad del servidor**.

El segundo hilo, *tareaGetRegister*, se encarga de obtener la *URL* del **fichero de registro** de la plantación, dirección que se carga en el navegador al clicar sobre el botón *Ficha plantación* en el menú.

El tercer hilo, *tareaAlarms*, es el proceso de **consulta de alarmas**, mediante el cual se solicita información al servidor acerca del estado del panel de alarmas. En el *"punto 4.3.2.8 Notificación de Alarmas" se* aborda el contenido de esta tarea.

Por último, la *activity Menu* contiene las funciones *onClick()* de los botones de su interfaz:

```
   public void OnClickMonitor (View view){startActivity(
          new Intent("com.andres.usingintent.Monitorizacion"));}
   public void OnClickRangos (View view){startActivity(
          new Intent("com.andres.usingintent.Rangos"));}
   public void OnClickAlarmas (View view){startActivity(
          new Intent("com.andres.usingintent.Alarmas"));}
   public void OnClickVentilador (View view){startActivity(
          new Intent("com.andres.usingintent.Ventilacion"));}
   public void OnClickPower (View view){startActivity(
          new Intent("com.andres.usingintent.Power"));}
   public void OnClickCredits (View view){startActivity(
          new Intent("com.andres.usingintent.Credits"));}
   public void OnClickGrafica (View view){startActivity(
          new Intent("com.andres.usingintent.Grafica"));}
```

Cada botón del *layout* tiene asociada una función a su evento *onClick*. Por ejemplo, si se observa el código del botón *Control Ventilación*:

```
<Button
    Android:layout_width="fill_parent"
    Android:layout_gravity="center"
    Android:layout_height="wrap_content"
    Android:text="Control Ventilación"
    Android:onClick="OnClickVentilador" />
```

Al hacer *Click* sobre él invocará a la función *OnClickVentilador()* y se lanzará la *activity Ventilación*.

La diferencia entre el menú para un *administrador* y para un *usuario* reside en los botones a mostrar, y por tanto en la limitación de acceso a ciertas *activities*.

#### 4.3.2.3. MONITORIZAR

El botón *Monitorizar ICRA* lanza la *activity Monitorización* y con ello la carga de un nuevo *layout*, en este caso el fichero *monitorizar.xml*, el cual representa un visor de variables. Se divide en cuatro zonas *(Ver Figura 56)*:

- Sensores: Representación de las tres variables principales del invernadero, **temperatura [ºC]**, **humedad [%]** y **nivel de contaminación**, ilustrando este último mediante un elemento `<ProgressBar/>`, vinculando su longitud al **[%]** de pureza del ambiente.
- Actuadores: Estado de activación del sistema de **calefacción** y de **humidificación** del invernadero, además de representar el nivel de potencia de **ventilación** del recinto.
- Alarmas: Representa el estado del **panel de alarmas** del invernadero, temperatura, humedad o disparo del sensor de gas.
- Refresh: En la parte inferior de la pantalla se ubica un botón que permite realizar un refresco de los valores del visor.

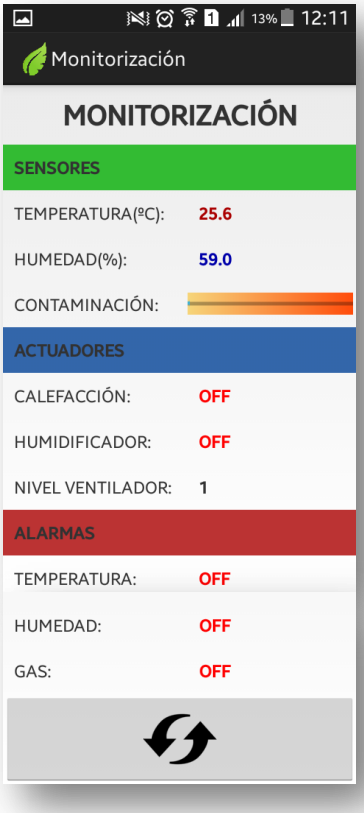



Figura 56: Layout Monitorización

Se observa ahora su clase *.java* asociada:

```
public class Monitorizacion extends Activity{
	…
	public void onCreate(Bundle savedInstanceState){

		super.onCreate(savedInstanceState);
		setContentView(R.layout.monitorizar);
		…
		TareaRefresh tareaRefreshCreate = new TareaRefresh();
		tareaRefreshCreate.execute();

		BtnRefresh.setOnClickListener(new OnClickListener() {
		public void onClick(View v) {

			TareaRefresh tareaRefreshClick = new TareaRefresh();
			tareaRefreshClick.execute();
			}
		});
	}
…
}
```

Tras inflar el *layout monitorizar,* se lanza un hilo secundario, *tareaRefreshCreate* encargado de **obtener el estado actual de las variables** del invernadero.

Dicho hilo, puede relanzarse si se pulsa el botón *Refresh* ubicado en la parte inferior del *layout*, con lo que se consigue actualizar los valores del visor.

Se analiza este hilo secundario:

```
private class TareaRefresh extends AsyncTask<String,Integer,Boolean> {
	…
    protected Boolean doInBackground(String... params) {
	boolean resul = true;
	HttpClient httpClient = new DefaultHttpClient();
	HttpGet del =new HttpGet(getString(R.string.Server_IP)+
	getString(R.string.Server_Port)+"/monitor/");
	del.setHeader("content-type", "application/json");
	try
	  {
		HttpResponse resp = httpClient.execute(del);
		String respStr = EntityUtils.toString(resp.getEntity());
		JSONObject respJSON = new JSONObject(respStr);//JSON respuesta
		temp=respJSON.getString("temp");//Extraccion de parametros JSON
		hum=respJSON.getString("hum");
		air=respJSON.getString("air");
		calef=respJSON.getString("calef");
		humidi=respJSON.getString("humidi");
		vent=respJSON.getString("vent");
		atemp=respJSON.getString("Atemp");
		ahum=respJSON.getString("Ahum");
		agas=respJSON.getString("Aair");     }
…
}
```

El hilo *tareaRefresh* realiza una petición *GET* al servidor solicitando un objeto *JSON* que contenga toda la información necesaria para el monitor.

Recibido este elemento, se extraen los diferentes parámetros en variables *string* auxiliares y se representa cada valor en su elemento *TxtView* presente en el *layout*.

#### 4.3.2.4. RANGOS

Desde el menú de *administrador* se puede acceder a dos *activities* de control, *Rango Actuadores* y *Rango Alarmas*, las cuales permiten modificar la zona de trabajo de los actuadores y de las alarmas *(Ver Figura 57")*.

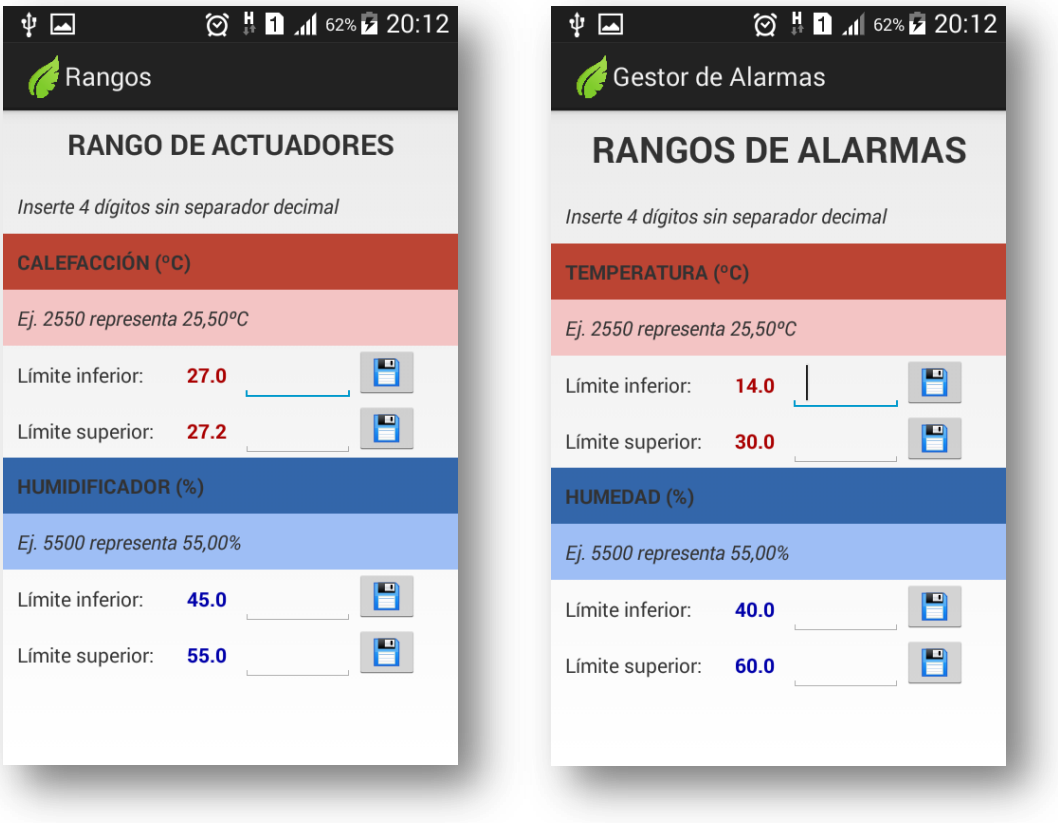



**Figura 57: Layout Rangos**

Las dos *activities* comparten un *layout* muy similar con una misma organización de elementos, y la programación del fichero *.java* sigue una lógica común.

Se describe brevemente el contenido de la clase *Alarms*:

```
public class Alarmas extends Activity{
        …
        public void onCreate(Bundle savedInstanceState){
               super.onCreate(savedInstanceState);
               setContentView(R.layout.alarmas);
               …
               //Obtener limites de rango actuales
               TareaStateGA tareaStateGA = new TareaStateGA();
               tareaStateGA.execute();
               //Notificacion de alarmas
               TareaAlarms tareaAlarms = new TareaAlarms();
               tareaAlarms.execute();

               //TEMPERATURA - Rango Inferior: t0alarma
               ImgBtnGAinftemp.setOnClickListener(new OnClickListener() {
               @Override
               public void onClick(View v) {
                      TareaSaveGA tareaSaveGA = new TareaSaveGA();
                      lim="t0a";
                      tareaSaveGA.execute("/"+lim+"/"+
                      EdtTxtGAinftemp.getText().toString());
               }
               });
               //TEMPERATURA - Rango Superior: t1alarma
               …
        }
 …}
```

Tras inflar el *layout* correspondiente se lanza un hilo secundario, *tareaStateGA,* con el objetivo de **obtener los límites actuales de rango** en el invernadero y mostrarlos en pantalla (además de un segundo hilo, como en el resto de *activities*, encargado de consultar el estado del panel de alarmas). *tareaStateGA* realiza una solicitud *GET* y recupera un objeto *JSON* con los valores de rango fijados en el sistema.

A continuación se establece el método *setOnClickListener()* para cada uno de los cuatros *ImageBoton* del *layout*, responsables de recoger el valor insertado por el usuario en el elemento *EditText* asociado a cada botón, y lanzar un hilo en segundo plano con dicha información.

Este hilo, *tareaSaveGA*, es el encargado de lanzar una petición *GET* con el valor introducido por el usuario y el parámetro al que hace referencia. El servidor responderá con la nueva cantidad fijada como umbral.

En el bloque de código mostrado, se fija un nuevo valor de alarma por *temperatura baja* mediante el método `ImgBtnGAinftemp.setOnClickListener()`.

#### 4.3.2.5. GRÁFICA Y TRAZADO

El botón *Visor Gráficas* lanza la *activity Grafica* y por consiguiente un nuevo *layout* en pantalla, *grafica.xml*.

Esta interfaz permite **seleccionar un rango temporal de trazado**, definiendo la *fecha/hora inicial y final*. El *botón calendario* rellena los campos de *fecha/hora final* automáticamente con los valores actuales del sistema operativo.

La interfaz ofrece la posibilidad de representar dos variables de la misma magnitud, eligiéndolas mediante un par de elementos visuales `<Spinner/>` *(Ver Figura 58).*

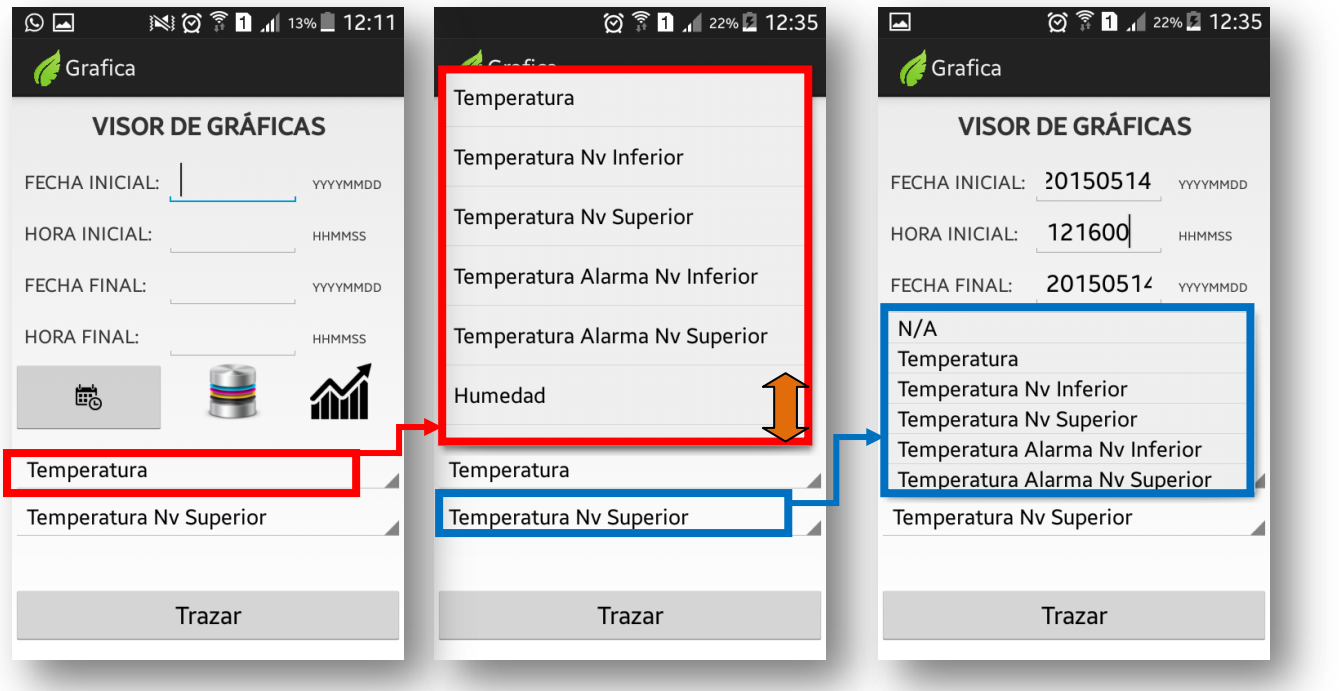



**Figura 58: Layout Visor de Gráficas**

El primer *spinner* permite seleccionar una de las variables del invernadero que están siendo almacenadas en la base de datos *MySQL* *(Ver "Figura 36").*

Una vez escogida una opción, el segundo *spinner* permite elegir otra variable vinculada a la primera, es decir, pueden representarse dos variables que compartan las mismas unidades.

En las *Figura 58* se ha escogido como *variable 1* la *temperatura* y como *variable 2* la *temperatura de cota superior*, ambas pertenecientes a la magnitud **temperatura** [ºC].

Se muestra ahora el código asociado en la clase *.java.*

```
public class Grafica extends Activity{
 …
 public void onCreate(Bundle savedInstanceState){
       super.onCreate(savedInstanceState);
       setContentView(R.layout.grafica);
 …
 final String[] lista =new String[]{"Temperatura","Temperatura Nv Inferior",
 "Temperatura Nv Superior","Temperatura Alarma Nv Inferior","Temperatura Alarma Nv
 Superior","Humedad","Humedad Nv Inferior","Humedad Nv Superior","Humedad Alarma
 Nv Inferior","Humedad Alarma Nv Superior","Calidad aire", "Ventilación",
 "Calefacción", "Humidificador","Alarma Temperatura","Alarma Humedad","Alarma
 Aire",”Parada Sistema"};
 final String[] temp =new String[]{"N/A","Temperatura","Temperatura Nv
 Inferior","Temperatura Nv Superior","Temperatura Alarma Nv Inferior",
 "Temperatura Alarma Nv Superior"};
 …
 ArrayAdapter<String> adaptador1 =new ArrayAdapter<String>
 (this,Android.R.layout.simple_spinner_item, lista);
 ArrayAdapter<String> adaptador2 =new ArrayAdapter<String>
 (this,Android.R.layout.simple_spinner_item, temp);
 cmbOpciones1 = (Spinner)findViewById(R.id.CmbOpciones);
 cmbOpciones2 = (Spinner)findViewById(R.id.CmbOpciones2);
 cmbOpciones1.setAdapter(adaptador1);
 cmbOpciones2.setAdapter(adaptador2);
```

La clase *Grafica* en su método *onCreate()* define varias cadenas de *String* con las diferentes opciones que se pueden mostrar en pantalla. Para que puedan ser representadas mediante los objetos *Spinner*, es necesario definir un objeto *ArrayAdapter* a partir una cadena *String* y cargar este elemento en el objeto *Spinner* deseado, mediante su método *.setAdapter(ArrayAdapter* );

Hecho esto, el elemento *Spinner* muestra todos los elementos del *Array* en pantalla y permite escoger uno de la lista, pudiendo ser este identificado por la *activity*.

Se muestra el método de identificación:

```
 cmbOpciones1.setOnItemSelectedListener(
              new AdapterView.OnItemSelectedListener() {
              public void onItemSelected(AdapterView<?> parent,
              Android.view.View v, int position, long id) {
                     param1=paramGET[position];
                     funcion1=leyenda[position];
                     unidad=unidades[position];
                     …
              }
```

El método *setOnItemSelectedListener()* recoge en la variable *position* la posición del vector que se ha seleccionado, y se emplea como índice para recoger tres datos de la variable: su **identificador *GET*** para la petición al servidor, la **leyenda** empleada en el trazado de la gráfica y las **unidades** que se representarán.

```
final String[] paramGET=new String[]{"temperatura","temp_min","temp_max","Atemp_min",
"Atemp_max", "humedad","hum_min","hum_max","Ahum_min","Ahum_max","aire","ventilador",
"calefaccion","humidificador","alarmaT","alarmaH","alarmaG","pause"};

final String[] leyenda =new String[]{"Temp","Temp.Inf","Temp.Sup","Temp.Alarm.Inf",
"Temp.Alarm.Sup","Hum","Hum.Inf","Hum.Sup","Hum.Alarm.Inf","Hum.Alarm.Sup",
"Calidad.Air","Vent","Calef","Humidi","Alarm.Temp","Alarm.Hum","Alarm.Air",
"Emergency"};
final String[] unidades =new String[]{"[ºC]","[ºC]","[ºC]","[ºC]","[ºC]","[%]",
"[%]","[%]","[%]","[%]","[n/a]","[ON/OFF]","[ON/OFF]","[ON/OFF]","[ON/OFF]",
"[ON/OFF]", "[ON/OFF]","[ON/OFF]"};
```

Los diferentes listados que pueden ser representados por el segundo *spinner* son los siguientes:

```
final String[] temp =new String[]{"N/A","Temperatura","Temperatura Nv Inferior",
"Temperatura Nv Superior","Temperatura Alarma Nv Inferior", "Temperatura Alarma Nv
Superior"};//[ºC]
final String[] hum =new String[]{"N/A","Humedad","Humedad Nv Inferior","Humedad Nv
Superior","Humedad Alarma Nv Inferior","Humedad Alarma Nv Superior"};//[%]
final String[] empty =new String[]{"N/A"};//[n/a]
final String[] perif =new String[]{"N/A","Ventilación","Calefacción",
"Humidificador","Alarma Temperatura","Alarma Humedad","Alarma Aire","Parada
Sistema"};//[ON/OFF]
```

Donde la determinación de cargar uno u otro vendrá dada por el contenido de la variable *unidad*.

Tras completar todos los campos del *layout* y pulsar el botón *Trazado*, se lanza un hilo en segundo planto, *tareaTrazar*, encargado de solicitar las series de datos al servidor. Para ello conforma una solicitud *GET* con la información insertada por el usuario:

```
HttpGet del =new HttpGet(getString(R.string.Server_IP)+getString(R.string.Server_Port)+
"/grafica/"+param1+"/"+param2+"/"+_date0+"/"+_time0+"/"+_date1+"/"+_time1);
```

El servidor retorna un elemento *JSONArray [ ]* formado por los datos temporales y los valores asociados de cada parámetro indicado por usuario. Tras obtener respuesta del servidor, se define un nuevo objeto *intent*:

```
        protected void onPostExecute(Boolean result) {

          if(result){
                TxtVwState.setText("Trazando...");
                Intent i=new Intent("com.andres.usingintent.Trazado");
                //String con JSONArray
                i.putExtra("respStr", respStr);
                //Parametros a trazar
                i.putExtra("param1", param1);
                i.putExtra("param2", param2);
                //Nombre de funcion
                i.putExtra("funcion1", funcion1);
                i.putExtra("funcion2", funcion2);
                //Unidades
                i.putExtra("unidad", unidad);

                startActivityForResult(i,1);
          }
        }
```

Dicho objeto, incluye la información necesaria para llevar a cabo el trazado:

- *String* con estructura *JSONArray [ ]* contenedor de los pares *time/valor*
- Nombre de las variables a trazar
- *Leyenda* de las dos variables
- *Unidades* de las variables

Tras adjuntar esta información, se lanza la *activity* ***Trazado*** encargada del ploteo.

```
public class Trazado extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
          …
          String respStr=getIntent().getStringExtra("respStr");
          param1=getIntent().getStringExtra("param1");
          param2=getIntent().getStringExtra("param2");
          //1 or 2 graphics
          if(!param2.equals("empty"))duo=true;
          else duo=false;
          …
          try{
                //Obtencion JSONObject
                JSONObject json = new JSONObject(respStr);
                //Obtencion JSONArray "grafica"
                JSONArray jArray = json.getJSONArray("grafica");
                int size=jArray.length();
                NmbSeries1=new Number[size*2];//Array {time,valor}
          …}
          }
    …}
```

La clase *Trazado.java* en su método *onCreate()* extrae el *JSONArray [ ]* del *string* adjunto al objeto *intent*, determina su tamaño y define un vector del tipo *Number* [ ] con la dimensión suficiente para almacenar todos los datos.

El *JSONArray [ ]* a desmenuzar sigue una estructura como la siguiente:

```
{"grafica":[
   {"time":"T0", "param1":"A0", "param2":"B0"}, //1- Elemento
   {"time":"T1", "param1":"A1", "param2":"B1"}, //2 - Elemento
    …
   {"time":"Tn", "param1":"An", "param2":"Bn"}, ]//n- Elemento
}
```

En el caso de que solo se haya seleccionado una variable a trazar, el contenido de *param2* es *empty*, estableciendo el bit *duo* a *false* y no considerando el tratamiento de dos gráficas.

Los vectores tipo *Number [ ]* tienen que tener el doble de componentes que el vector *JSONArray [ ]*, de forma que puedan seguir una estructura como la siguiente:

```
NmbSeries1[ ] ={T0, A0, T1, A1, …Tn, An} // 2·n - Elementos

NmbSeries2[ ] ={T0, B0, T1, B1, …Tn, Bn} // 2·n – Elementos
```

El siguiente diagrama de flujo *(Ver "Figura 59")* representa el bucle de extracción de datos desde el *JSONArray [ ]* hacia los vectores *NmbSeries [ ]*. Para garantizar la completa visualización del gráfico, se recoge el máximo, *max*, y mínimo ,*min*, absoluto de la funciones, con el fin de establecer los rangos de ploteado en base a estos valores.

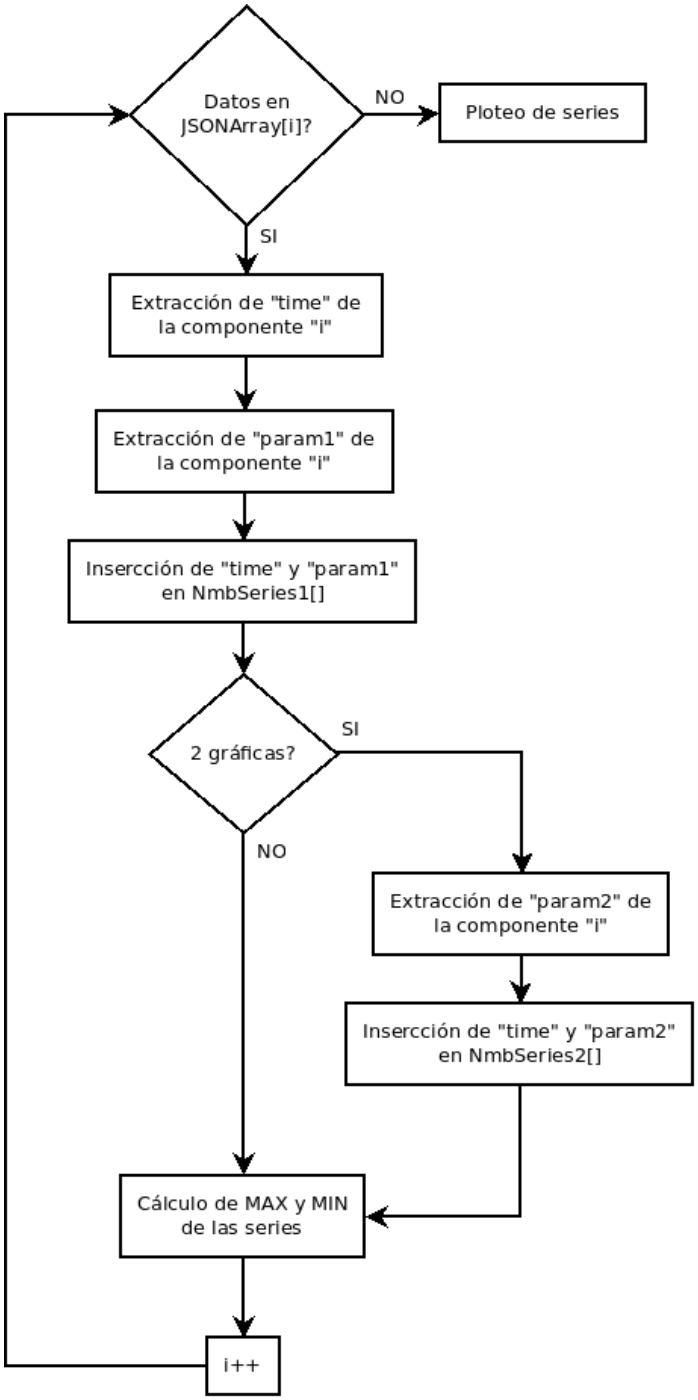



Figura 59: Diagrama de flujo datos gráficas

Una vez recogidos los valores en los vectores *Number [ ]*, se procede a inflar el *layout plot.xml* y a representar las series de datos.

Para llevar a cabo esta tarea se hace uso de una librería libre para ploteado en *Android*, ***Androidplot*****,** la cual incluye las funciones necesarias para llevar a cabo el trazado de series de datos *(Ver "Figura 60")*. Se muestra el código asociado al ploteo:


Figura 60: Librería para gráficas AndroidPlot

```
//Set layout
setContentView(R.layout.plot);
plot = (XYPlot) findViewById(R.id.xyPlot);
//XYSeries
XYSeries series1 = new SimpleXYSeries(Arrays.asList(NmbSeries1),
SimpleXYSeries.ArrayFormat.XY_VALS_INTERLEAVED,funcion1);
//Formato de XYSeries
LineAndPointFormatter s1Format = new LineAndPointFormatter();
s1Format.setPointLabelFormatter(new PointLabelFormatter());
s1Format.configure(getApplicationContext(),R.xml.lpf1);
//Add to Ploteado
plot.addSeries(series1, s1Format);
if(duo){
      XYSeries series2
      …
```

Para realizar el ploteo de una serie de datos, es necesario definir un objeto *XYPlot* en la clase *.java* y vincularlo a un elemento *XYPlot* instanciado en un fihchero *XML*. Dicho objeto goza de ciertos métodos que permiten adjuntarle objetos *XYSeries* y formatos *LineAndPointFormatter*.

La clase *XYSeries* permite definir series de datos a partir de arrays *Number [ ],* y la clase *LineAndPointFormatter* establece propiedades de grosor, color… del trazado a partir de un fichero XML.

Un posible resultado, basándose en el ejemplo mostrado en la *Figura 58*, sería el representado en la *Figura 61*, donde puede apreciarse el aumento del valor de la *temperatura* hasta alcanzar la cota máxima fijada en *27,1ºC*, teniendo lugar este evento en torno a la hora *12:28:00h*, y posteriormente se visualiza como el usuario modifico el valor *máximo de temperatura*, bajándolo hasta *23ºC* en torno a la hora *12:31:00h*.

En el *Anexo VI: Manual de usuario de ICRApp* se muestran varios ejemplos sobre el trazado de gráficas y sus posibilidades.

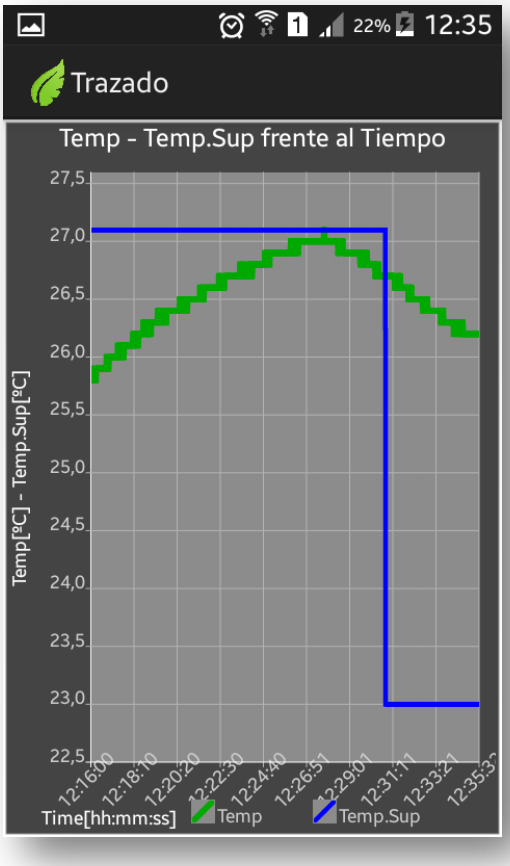



Figura 61: Layout Trazado

#### 4.3.2.6. VENTILACIÓN

La clase *Ventilación* se encarga de **fijar la velocidad del ventilador** en el invernadero, para ello se utiliza un elemento *View* del tipo *RadioGroup* que permite seleccionar una única opción de las listadas *(Ver Figura 62).*

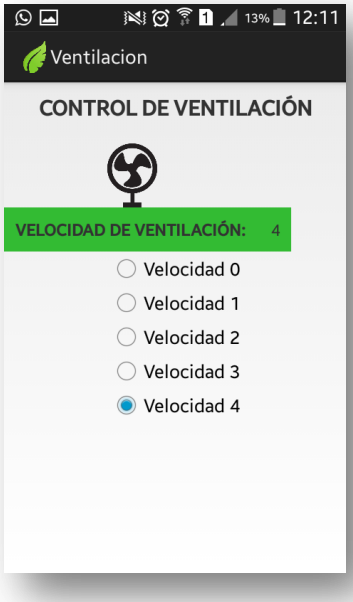



Figura 62: Layout Ventilación

La *activity* nada más ser creada lanza un hilo secundario encargado de recoger el valor actual de ventilación en el recinto, *tareaSetSpeed("I")*, y en su función *onPostExecute()* representa el nivel obtenido en pantalla.

La atención a los eventos del elemento *RadioGroup* sigue la siguiente estructura en el fichero *.java*:

```
public class Ventilacion extends Activity{
   public void onCreate(Bundle savedInstanceState){
          …
          TareaSetSpeed tareaSetSpeed = new TareaSetSpeed();
          tareaSetSpeed.execute("I"); //Obtener Velocidad actual

          BtnVel0.setOnClickListener(new OnClickListener() {
                @Override
                public void onClick(View v) {
                      TareaSetSpeed tareaSetSpeed = new TareaSetSpeed();
                      tareaSetSpeed.execute("0");
                }
          });
          BtnVel1.setOnClickListener(new OnClickListener() {
                @Override
                public void onClick(View v) {
                      TareaSetSpeed tareaSetSpeed = new TareaSetSpeed();
                      tareaSetSpeed.execute("1");
                }
                });

          …
          }

   …
   }
```

Si se detecta una pulsación en alguno de los elementos del *RadioGroup*, se invoca su método asociado. Los métodos *setOnClickListener()* invocan un hilo secundario, el cual recibe como parámetro el identificador del botón seleccionado. Dicho hillo lleva a cabo una solicitud *GET* con el nivel de velocidad deseado y recibe desde el servidor el nuevo valor de ventilación establecido. Tanto el hilo para obtener el estado actual como los hilos para fijar nuevos valores son los mismos, únicamente varia el identificador que recibe como parámetro la función.

#### 4.3.2.7. SETA DE EMERGENCIA

La misión de esta *activity* es la de gestionar el **estado de la seta de emergencia** *Android*, permitiendo su interacción con el usuario y mostrando su estado actual.

Al igual que la anterior *activity* analizada, la clase *Power.java* lanza un hilo secundario al crear la *activity* con el fin de obtener el valor actual del pulsador y de representarlo en pantalla. En el instante que se detecte una pulsación sobre el elemento *ImageView* de la seta de emergencia *(Ver Figura 63),* se invoca al método *setOnClickListener()* asociado:

```
ImgVwSeta.setOnClickListener(new OnClickListener(){
    public void onClick(View v) {
          TareaSetSwitch tareaSetSwitch = new TareaSetSwitch();
          tareaSetSwitch.execute("X");//X:Invert state
    }
```

El método, vinculado a la pulsación sobre la imagen, lanza un hilo con la tarea de invertir el estado actual del pulsador de parada.

```
private class TareaSetSwitch extends AsyncTask<String,Integer,Boolean> {
    …
    if(id.equals("I")){
          del =new HttpGet(getString(R.string.Server_IP)+
          getString(R.string.Server_Port)+ "/monitor");
          del.setHeader("content-type", "application/json");}
    else{
          val=(status.equals("0")==true?"1":"0");
          del =new HttpGet(getString(R.string.Server_IP)+
          getString(R.string.Server_Port)+"/switch/"+val);
          del.setHeader("content-type", "application/json");}
    …}
```

En su método *onPostExecute()* se representa el estado actual del pulsador en el elemento *TxtView* del *layout*.


Figura 63: Layout Power

#### 4.3.2.8. NOTIFICACIÓN DE ALARMAS

Cada una de las *activities* de la aplicación, en su método *onCreate()*, lanza un hilo en segundo plano para **consultar el estado de las alarmas** del invernadero.

Dicha tarea tiene como objetivo evaluar los bits del *panel de alarmas* y en el caso de detectar algún nivel alto, lanzar una *notificación* de alerta al usuario en su dispositivo. Se muestra el contenido del hilo:

```
private class TareaAlarms extends AsyncTask<String,Integer,Boolean> {

    private String atemp,ahum,agas;
    protected Boolean doInBackground(String... params) {
    boolean resul = true;
    HttpClient httpClient = new DefaultHttpClient();//Cliente HTTP
    HttpGet del =new HttpGet(getString(R.string.Server_IP)+
    getString(R.string.Server_Port)+"/panel/");//Solicitud GET
    del.setHeader("content-type", "application/json");
    try{
        HttpResponse resp = httpClient.execute(del);
        String respStr = EntityUtils.toString(resp.getEntity());
        JSONObject respJSON = new JSONObject(respStr);//JSON respuesta
       atemp=respJSON.getString("Atemp");
       ahum=respJSON.getString("Ahum");
       agas=respJSON.getString("Aair");  }
…
}
```

El hilo *tareaAlarms* realiza una solicitud *GET* al servidor pidiendo el estado del *panel de alarmas*, obteniendo como respuesta un elemento *JSON* con la información de los 3 bits. Cada bit se extrae en un variable y se analiza su valor:

```
private class TareaAlarms extends AsyncTask<String,Integer,Boolean> {
      protected void onPostExecute(Boolean result) {

            if (result)
            {
                  if(atemp.equals("1")&&!PanelAlarmas.temperatura){
                              displayNotification(1);
                              PanelAlarmas.temperatura=true;
                  }

                  if(ahum.equals("1")&&!PanelAlarmas.humedad){/
                              displayNotification(2);
                              PanelAlarmas.humedad=true;

                  }

                  if(agas.equals("1")&&!PanelAlarmas.aire){
                              displayNotification(3);
                              PanelAlarmas.aire=true;
                  }
            }
      }//Fin OnPostExecute
```

En el caso de que el servidor informe de una alarma activa y está no se encuentre ya notificada en la aplicación, el hilo setea la alarma correspondiente en su clase global *PanelAlarmas* y lanza una notificación en el dispositivo, utilizando para ello la función *displayNotification(int )*, y pasando como parámetro el *ID* que provoco la *notificación*.

El objetivo de la función *displayNotification(int )* es el de recoger el *identificador* que provoco la *notificación* de alarma, mostrar el texto de usuario en la *barra de notificaciones* y lanzar el evento *NotificationView* con el que el usuario puede interactuar.

Se muestra el contenido de esta función:

```
protected void displayNotification(int _notificationID){
	…
	Intent i=new Intent(this,NotificationView.class);
	i.putExtra("notificationID",_notificationID);
	…
	switch(_notificationID){
		case 1://Alarma TEMPERATURA
			notif=new Notification(R.drawable.alarma_temp,"TEMPERATURA",
			System.currentTimeMillis());
			from="Alarma de Temperatura";
			message="Temperatura crítica en el sistema";
			break;
		case 2://Alarma HUMEDAD
			notif=new Notification(R.drawable.alarma_hum,"HUMEDAD",
			System.currentTimeMillis());
			from="Alarma de Humedad";
			message="Humedad crítica en el sistema";
			break;
		case 3://Alarma GAS
			notif=new Notification(R.drawable.alarma_gas,"GAS",
			System.currentTimeMillis());
			from="Alarma de Gas";
			message="Calidad de aire crítica en el sistema";
			break;
		default:
			notif=new Notification(R.drawable.ic_launcher,"Error",
			System.currentTimeMillis());
			from="Error de notificación";
			message="Error en ID de notificación";
	}
	…
	nm.notify(_notificationID, notif);
}
```

Donde la clase *NotificationView* se encarga de inflar uno de los tres posibles *layout* en función del *ID* que lanzo la notificación. Además, esta clase lanza un hilo secundario con la finalidad de obtener los rangos de alarmas actuales y el valor actual de la variable que provoco la alarma, de modo que el usuario pueda comprobar cómo el valor de ese parámetro se ha desbordado del rango de seguridad establecido.

Una vez ejecutada esta *activity*, el atributo de alarma asociado al objeto global `PanelAlarmas` es desactivado, de manera que la alarma pueda ser relanzada en la aplicación.

La siguiente Figura *(Ver Figura 64)* muestra los tres *layouts* de notificación de alarmas en la aplicación:


Figura 64: Layout NotificationView

# Capítulo 5. Conclusiones y líneas futuras

En este capítulo se presentan las reflexiones sobre los trabajos desarrollados y se evalúan los objetivos conseguidos, además de presentar varias líneas futuras de investigación.

## 5.1. CONCLUSIONES

La maqueta construida ha permitido simular las condiciones ambientales que tienen lugar en el interior de un invernadero, y por consiguiente se han podido llevar a cabo los objetivos propuestos en este trabajo. Se han cubierto todas las fases de desarrollo propuestas y se ha **conseguido el objetivo principal** del proyecto que era conseguir implementar una aplicación remota, capaz de efectuar un control y una monitorización sobre un invernadero.

La programación del **microcontrolador** *Arduino* se ha realizado siguiendo una programación orientada a objetos, con lo que se facilita la incorporación de nuevos periféricos o modificaciones en los ya existentes.

Las tareas principales que lleva a cabo la **aplicación servidor** se han desarrollado empleando funciones y clases, lo que repercute en una aplicación versátil y fácilmente escalable. Propuestas como recibir *una nueva variable* del invernadero, ampliar el *mensaje de comunicación* o crear un *nuevo campo* en la base de datos son fácilmente asumibles.

Se ha desarrollado una **capa de servicios web** que ofrece soporte con independencia del sistema cliente. Esto significa que el servidor está capacitado para atender cualquier número de peticiones cliente, con independencia del sistema operativo que estos tengan instalado y de la aplicación que utilicen para comunicarse. Esto es gracias a las características de los servicios *REST* y de la notación de mensajes *JSON*, los cuales ofrecen una alta flexibilidad.

La **aplicación de gestión de usuarios** ofrece una interfaz rápida y sencilla para regular el alta y baja de usuarios en el sistema, y al igual que la aplicación servidor, ha sido programada de tal forma que pueda ser modificada y ampliada de manera sencilla.

El desarrollo de la **aplicación Android** ha ofrecido un cuadro de mando versátil y fácilmente transportable. Se han conseguido desarrollar tanto las tareas de control como de visión propuestas en este trabajo y, gracias a la estructura basada en *activities,* es posible aumentar el número de controles de la aplicación de una forma fácil y rápida. Sumado a la gran compatibilidad de la aplicación entre todas la versiones del *SO*.

Se ha llevado a cabo un **control de versiones** del software y se ha trabajado con un repositorio online *(Ver "Figura 65")*, al cual se puede acceder para obtener las aplicaciones desarrolladas. El único proceso que ha de realizar el usuario cliente es el de entrar al repositorio https://bitbucket.org/invernadero/, descargar la versión final de *ICRApp* e instalarla en su dispositivo *Android* para poder disfrutar de los servicios implementados. Con ello se garantiza una alta accesibilidad a las aplicaciones y a sus futuras versiones.


Figura 65: Repositorio online Bitbucket

Finalmente se ha podido desarrollar todo el software empleando código **opensource,** lo que otorga una gran ventaja desde el punto de vista económico y desde la posibilidad de encontrar fácilmente soporte a través de la web. Con lo que se contribuye al desarrollo de proyectos de *código abierto,* los cuales cualquier usuario puede probar, modificar, mejorar y publicar.

## 5.2. LÍNEAS FUTURAS

El presente proyecto es un punto de partida en la monitorización y supervisión de invernaderos, pensado para ser integrado en un sistema real. Creado y desarrollado para ser escalado, y todo ello con el objetivo de obtener unos mejores resultados en la calidad de la plantación. Se presentan algunas de las posibles mejoras a implementar:

- Integración del **sistema GCM** (*Google Cloud Messaging*) *(Ver "Figura 66")*, de forma que el usuario *Android* pueda recibir notificaciones de alerta mediante un correo electrónica en el dispositivo *smartphone*, con independencia de tener o no en primer plano la aplicación de monitorización.

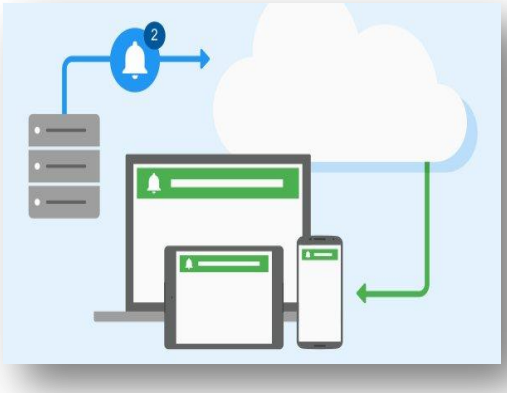



Figura 66: Sistema GCM de Google

- Establecer el protocolo de comunicación **HTTPS** entre la aplicación móvil y el servidor, sustituyendo el protocolo *HTTP* empleado actualmente.
- Implementar una nueva sección en la aplicación *Android* que permite visualizar el **consumo eléctrico de los actuadores**, obteniendo dicha información a partir de los datos temporales de trabajo.
- Integración de una **videocámara** y desarrollo de los métodos necesarios para poder visualizar, e incluso interaccionar, con este elemento desde la aplicación móvil *(Ver "Figura 67").*

Figura 67: Videocámara ICRA

# GLOSARIO

Activity: Representan el componente principal de la interfaz gráfica de una aplicación *Android*. Se puede pensar en una actividad como el elemento análogo a una ventana en cualquier otro lenguaje visual.

APP: Tipo de programa informático diseñado como herramienta para permitir a un usuario realizar uno o diversos tipos de trabajos para dispositivos móviles o tabletas

Bitbucket: Servicio de alojamiento basado en web, para los proyectos que utilizan el sistema de control de revisiones Mercurial y Git. Bitbucket ofrece planes comerciales y gratuitos.

Concurrencia: La computación concurrente es la simultaneidad en la ejecución de múltiples tareas interactivas. Estas tareas pueden ser un conjunto de procesos o hilos de ejecución creados por un único programa.

Handshaking: La traducción de "handshaking" es "apretón de manos" y viene a significar que el procesador y los periféricos intercambian señales de control que les permiten sincronizar sus acciones y "colaborar" conjuntamente en la transferencia de información.

HTTP: Hypertext Transfer Protocol o HTTP (en español protocolo de transferencia de hipertexto) es el protocolo usado en cada transacción de la *World Wide Web*. Es un protocolo orientado a transacciones y sigue el esquema petición-respuesta entre un cliente y un servidor.

IGU: Interfaz gráfica de usuario. Está construida mediante una jerarquía de *View* y objetos *ViewGroup.*

Interrupción: Señal recibida por el procesador de una computadora, para indicarle que debe «interrumpir» el curso de ejecución actual y pasar a ejecutar código específico para tratar esta situación.

IU: Interfaz de usuario. Todo aquello que el usuario puede ver e interactuar.

JSON: acrónimo de JavaScript Object Notation, es un formato ligero para el intercambio de datos. JSON es un subconjunto de la notación literal de objetos de JavaScript que no requiere el uso de XML.

Microcontrolador: Circuito integrado programable, capaz de ejecutar las órdenes grabadas en su memoria. Está compuesto de varios bloques funcionales, los cuales cumplen una tarea específica. Un microcontrolador incluye en su interior las tres principales unidades funcionales de una computadora: unidad central de procesamiento, memoria y periféricos de entrada/salida.

*MySQL*: *MySQL* es un sistema de gestión de bases de datos relacional, multihilo y multiusuario.

Opensource: Código abierto es la expresión con la que se conoce al software o hardware distribuido y desarrollado libremente. Se focaliza más en los beneficios prácticos (acceso al código fuente) que en cuestiones éticas o de libertad que tanto se destacan en el software libre.

Parser: Un analizador sintáctico (o parser) es una de las partes de un compilador que transforma su entrada en un árbol de derivación. El análisis sintáctico convierte el texto de entrada en otras estructuras (comúnmente árboles), que son más útiles para el posterior análisis y capturan la jerarquía implícita de la entrada.

Processing: Lenguaje de programación y entorno de desarrollo integrado de código abierto basado en *Java*, de fácil utilización, y que sirve como medio para la enseñanza y producción de proyectos multimedia e interactivos de diseño digital

Puerto Serie: Interfaz de comunicaciones de datos digitales, frecuentemente utilizado por computadoras y periféricos, donde la información es transmitida bit a bit enviando un solo bit a la vez, en contraste con el puerto paralelo que envía varios bits simultáneamente

PWM: Técnica en la que se modifica el ciclo de trabajo de una señal periódica (una senoidal o una cuadrada, por ejemplo), ya sea para transmitir información a través de un canal de comunicaciones o para controlar la cantidad de energía que se envía a una carga.

Serializar: Proceso de codificación de un objeto en un medio de almacenamiento (como puede ser un archivo, o un buffer de memoria) con el fin de transmitirlo a través de una conexión en red como una serie de bytes o en un formato humanamente más legible como *XML* o *JSON*, entre otros.

Smartphone: El teléfono inteligente o “inteléfono” (en inglés: smartphone) es un tipo teléfono móvil construido sobre una plataforma informática móvil, con una mayor capacidad de almacenar datos y realizar actividades, semejante a la de una minicomputadora, y con una mayor conectividad que un teléfono móvil convencional.

# ANEXO I.

# PRESUPUESTO

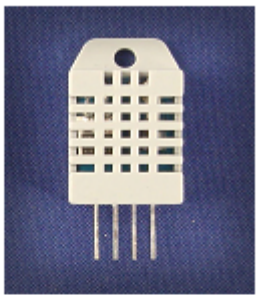
Tabla 8: Presupuesto ICRA

| PRESUPUESTO | | | | |
|---|---|---|---|---|
| **Cantidad** | **Elemento** | **Base** | **Valor (€/base)** | **Valor (€)** |
| **3** | Cuadradillos madera samba 20x20 | Metro | 1,50 € | 4,50 € |
| **0,5** | Listón madera samba 50x50 | Metro | 3,99 € | 2,00 € |
| **1** | Tabla madera DM 240x215 | Unidad | 2,99 € | 2,99 € |
| **1** | Rollo forro plastico (2m) | Unidad | 1,99 € | 1,99 € |
| **1** | Rollo fieltro adhesivo (2m) | Unidad | 2,50 € | 2,50 € |
| **3** | Alambre 1mm ø | Metro | 1,20 € | 3,60 € |
| **2** | Rollo cinta adhesiva color negro | Unidad | 0,99 € | 1,98 € |
| **1** | Rollo cinta adhesiva doble cara | Unidad | 6,00 € | 6,00 € |
| **1** | Rollo celofan | Unidad | 1,00 € | 1,00 € |
| **2** | Tubo super glue | Unidad | 2,00 € | 4,00 € |
| **1** | Tubo pegamento para madera | Unidad | 5,99 € | 5,99 € |
| **1** | Tubo 200ml pintura blanca acrílica | Unidad | 2,99 € | 2,99 € |
| **2** | Hembrilla | Unidad | 0,15 € | 0,30 € |
| **1** | Lámina cartón 20x30 | Unidad | 0,40 € | 0,40 € |
| **1** | Rollo velcro adhesivo (1 m) | Unidad | 2,99 € | 2,99 € |
| **1** | Paquete grapas 23/6 | Unidad | 0,80 € | 0,80 € |
| **1** | Paquete chinchetas | Unidad | 0,80 € | 0,80 € |
| **12** | Cable jumper hembra/hembra | Unidad | 0,50 € | 6,00 € |
| **10** | Cable jumper macho/macho | Unidad | 0,50 € | 5,00 € |
| **3** | Cable cobre 2mm ø | Metro | 0,60 € | 1,80 € |
| **5** | Hilo cobre 1mm ø | Metro | 0,40 € | 2,00 € |
| **1** | Portapilas 3AAA (1 pila) | Unidad | 2,90 € | 2,90 € |
| **1** | Portapilas 2AA (4 pilas) | Unidad | 3,90 € | 3,90 € |
| **1** | Interruptor | Unidad | 2,00 € | 2,00 € |
| **3** | Clema conexión | Unidad | 0,50 € | 1,50 € |
| **1** | Protoboard 400 puntos | Unidad | 8,50 € | 8,50 € |
| **1** | Protoboard 170 puntos | Unidad | 6,20 € | 6,20 € |
| **1** | Pila 12V 3AAA | Unidad | 1,50 € | 1,50 € |
| **1** | Pack 4 Pilas Alcalinas 1,5V 2AAA | Unidad | 2,50 € | 2,50 € |
| **1** | Transformador 220V/6V | Unidad | 18,50 € | 18,50 € |
| **2** | Portabombillas casquillo pequeño | Unidad | 0,50 € | 1,00 € |
| **1** | Cable micro USB – USB | Unidad | 6,90 € | 6,90 € |
| **4** | Diodo LED 3mm Alta luminosidad | Unidad | 1,50 € | 6,00 € |
| **6** | Diodo LED 3mm (varios colores) | Unidad | 0,70 € | 4,20 € |
| **11** | Resistencia 150 | Unidad | 0,10 € | 1,10 € |
| **2** | Resistencia 10k | Unidad | 0,12 € | 0,24 € |
| **2** | Diodo 1N5819 | Unidad | 0,40 € | 0,80 € |
| **1** | Relé 12VDC DPDT 2 C/O 8A | Unidad | 3,90 € | 3,90 € |
| **1** | Transistor 2N2222A | Unidad | 1,20 € | 1,20 € |
| **1** | Interruptor protoboard | Unidad | 2,00 € | 2,00 € |
| **1** | Ventilador 12V PC | Unidad | 5,50 € | 5,50 € |
| **2** | Bombilla 2W casquillo pequeño | Unidad | 1,20 € | 2,40 € |
| **1** | Zumbador piezoelectrico | Unidad | 3,50 € | 3,50 € |
| **1** | Sensor DHT22 *Arduino* | Unidad | 14,90 € | 14,90 € |
| **1** | Sensor MQ2 *Arduino* | Unidad | 15,90 € | 15,90 € |
| **1** | Microcontrolador *Arduino* Leonardo | Unidad | 29,95 € | 29,95 € |
| | | | Subtotal | 206,62 € |
| | | | IVA (21%) | 43,39 € |
| | | | **TOTAL** | **250,00 €** |

# ANEXO II.
# HOJAS DE CARACTERÍSTICAS

# Digital-output relative humidity & temperature sensor/module

# AM2303



Capacitive-type humidity and temperature module/sensor

## 1. Feature & Application:

* Full range temperature compensated
* Relative humidity and temperature measurement
* Calibrated digital signal
*Outstanding long-term stability
*Extra components not needed
* Long transmission distance
* Low power consumption
*4 pins packaged and fully interchangeable

## 2. Description:

AM2303 output calibrated digital signal. It utilizes exclusive digital-signal-collecting-technique and humidity sensing technology, assuring its reliability and stability.Its sensing elements is connected with 8-bit single-chip computer.

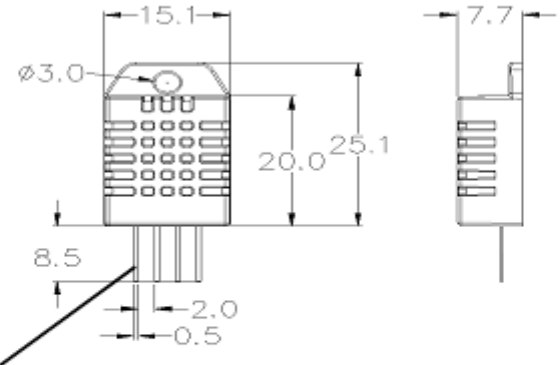
Every sensor of this model is temperature compensated and calibrated in accurate calibration chamber and the calibration-coefficient is saved in type of programme in OTP memory, when the sensor is detecting, it will cite coefficient from memory.
Small size & low consumption & long transmission distance(20m) enable AM2303 to be suited in all kinds of harsh application occasions.

Single-row packaged with four pins, making the connection very convenient.

## 3. Technical Specification:

| | |
|---|---|
| Model | AM2303 |
| Power supply | 3.3-6V DC |
| Output signal | digital signal via single-bus |
| Sensing element | Polymer humidity capacitor & DS18B20 for detecting temperature |
| Measuring range | humidity 0-100%RH; temperature -40~125Celsius |

## 4. Dimensions: (unit----mm)




**Pin sequence number: 1 2 3 4 (from left to right direction).**

| Pin | Function |
|---|---|
| 1 | VDD----power supply |
| 2 | DATA--signal |
| 3 | NULL |
| 4 | GND |

## 5. Operating specifications:

**(1) Power and Pins**

Power's voltage should be 3.3-6V DC. When power is supplied to sensor, don't send any instruction to the sensor within one second to pass unstable status. One capacitor valued 100nF can be added between VDD and GND for wave filtering.

**(2) Communication and signal**

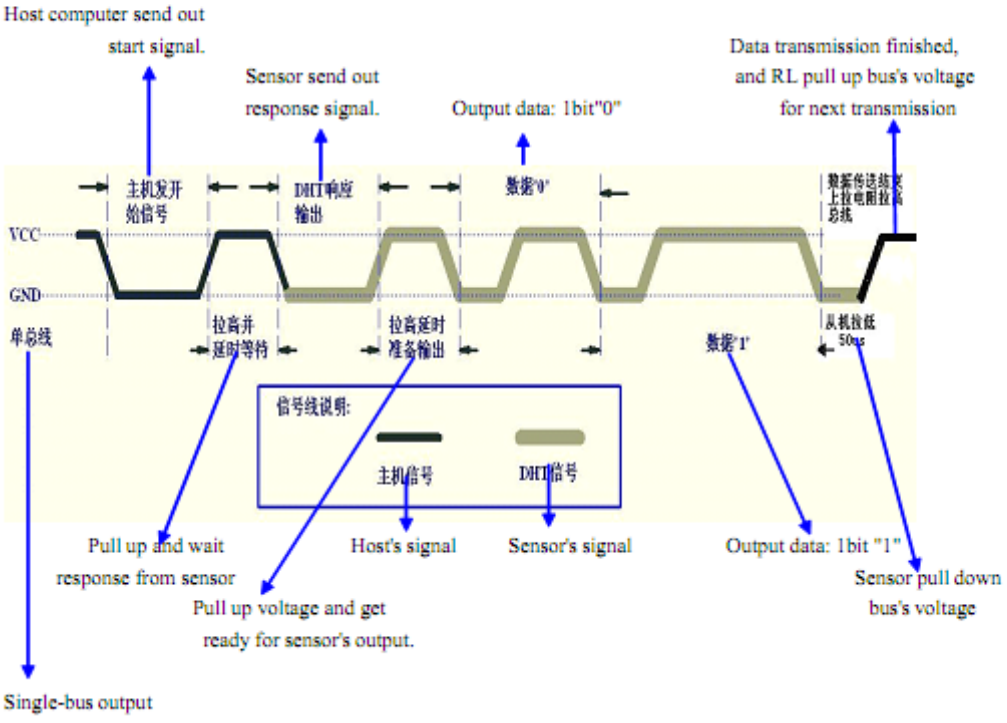
Single-bus data is used for communication between MCU and AM2303, it costs 5mS for single time communication.

Data is comprised of integral and decimal part, the following is the formula for data.
AM2303 send out higher data bit firstly!
DATA=8 bit integral RH data+8 bit decimal RH data+8 bit integral T data+8 bit decimal T data+8 bit check-sum
If the data transmission is right, check-sum should be the last 8 bit of "8 bit integral RH data+8 bit decimal RH data+8 bit integral T data+8 bit decimal T data".

When MCU send start signal, AM2303 change from low-power-consumption-mode to running-mode. When MCU finishs sending the start signal, AM2303 will send response signal of 40-bit data that reflect the relative humidity and temperature information to MCU. Without start signal from MCU, AM2303 will not give response signal to MCU. One start signal for one time's response data that reflect the relative humidity and temperature information from AM2303. AM2303 will change to low-power-consumption-mode when data collecting finish if it don't receive start signal from MCU again.

1) Check bellow picture for overall communication process:

---

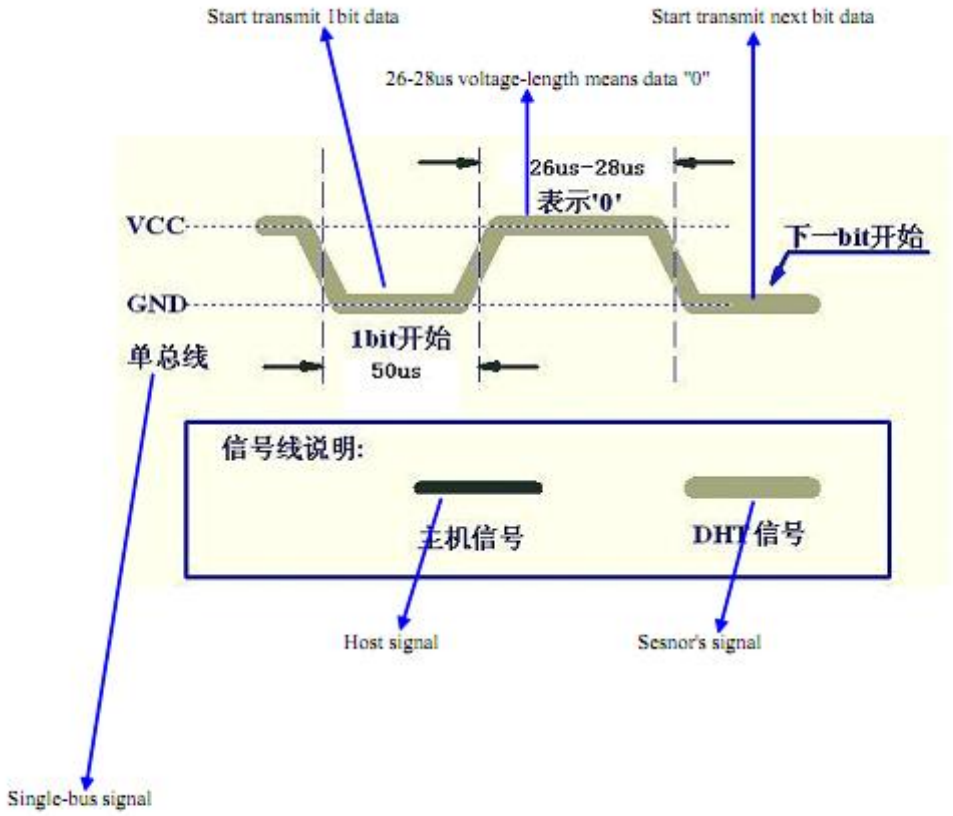



---

2) Step 1: MCU send out start signal to AM2303

Data-bus's free status is high voltage level. When communication between MCU and AM2303 begin, program of MCU will transform data-bus's voltage level from high to low level and this process must beyond at least 18ms to ensure AM2303 could detect MCU's signal, then MCU will wait 20-40us for AM2303's response.

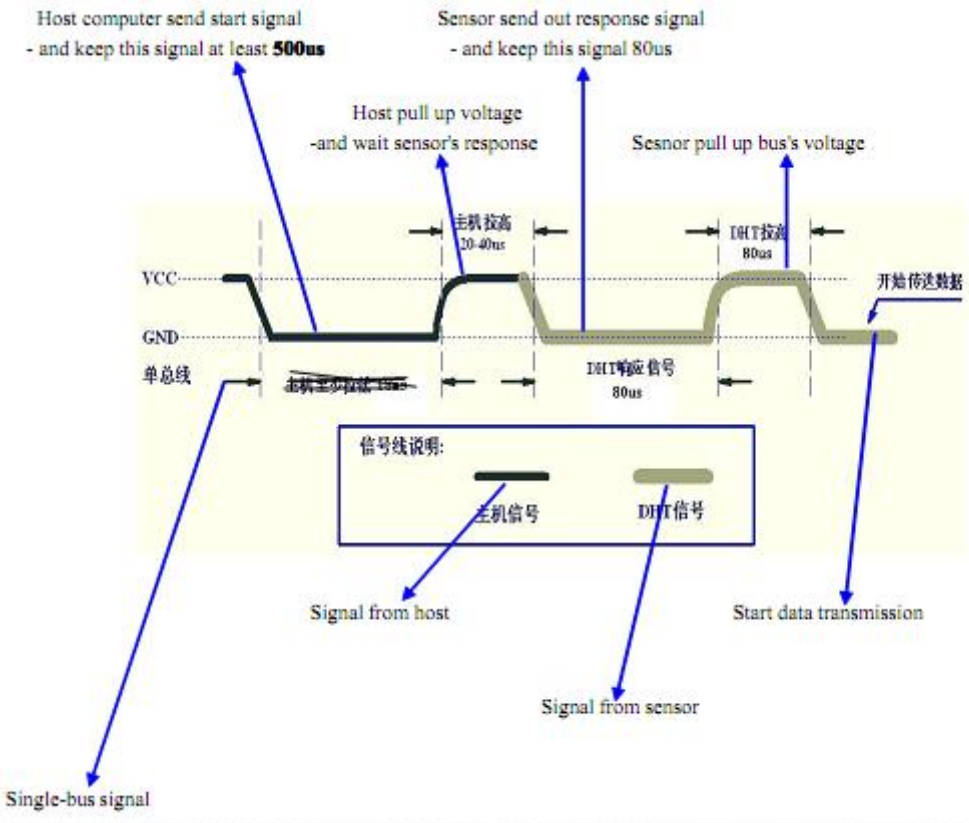
Check bellow picture for step 1:

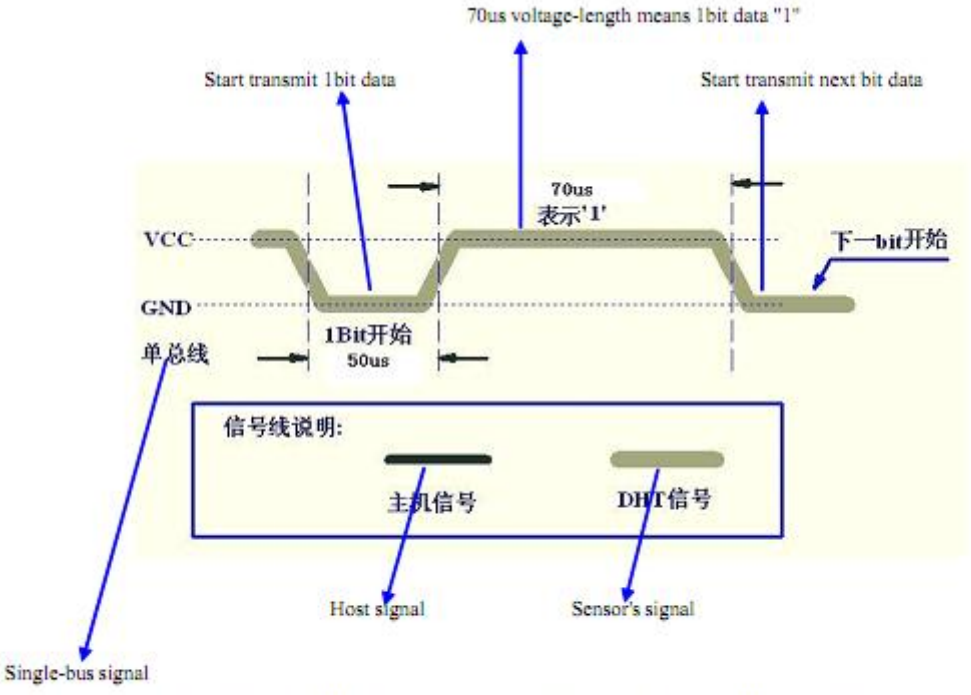



Step 3: AM2303 send data to MCU

When AM2303 is sending data to MCU, every bit's transmission begin with low-voltage-level that last 50us, the following high-voltage-level signal's length decide the bit is "1" or "0".

Check bellow picture for step 3:






---

If signal from AM2303 is always high-voltage-level, it means AM2303 is not working properly, please check the electrical connection status.

## 6. Electrical Characteristics:

| Item | Condition | Min | Typical | Max | Unit |
|---|---|---|---|---|---|
| Power supply | DC | 3.3 | 5 | 5.5 | V |
| Current supply | Measuring | 1.3 | 1.5 | 2.1 | mA |
| | Average | 0.5 | 0.8 | 1.1 | mA |
| Collecting period | Second | 1.7 | | 2 | Second |

*Collecting period should be : >1.7 second.

## 7. Attentions of application:

(1) Operating and storage conditions

We don't recommend the applying RH-range beyond the range stated in this specification. The DHT11 sensor

(2) Attentions to chemical materials

Vapor from chemical materials may interfere AM2303's sensitive-elements and debase AM2303's sensitivity.

(3) Disposal when (1) & (2) happens

Step one: Keep the AM2303 sensor at condition of Temperature 50~60Celsius, humidity <10%RH for 2 hours;

Step two: After step one, keep the AM2303 sensor at condition of Temperature 20~30Celsius, humidity >70%RH for 5 hours.

(4) Attention to temperature's affection

Relative humidity strongly depend on temperature, that is why we use temperature compensation technology to ensure accurate measurement of RH. But it's still be much better to keep the sensor at same temperature when sensing.

AM2303 should be mounted at the place as far as possible from parts that may cause change to temperature.

(5) Attentions to light

Long time exposure to strong light and ultraviolet may debase AM2303's performance.

(6) Attentions to connection wires

The connection wires' quality will effect communication's quality and distance, high quality shielding-wire is recommended.

(7) Other attentions

* Welding temperature should be bellow 260Celsius.
* Avoid using the sensor under dew condition.
* Don't use this product in safety or emergency stop devices or any other occasion that failure of AM2303 may cause personal injury.

# MQ-2 Semiconductor Sensor for Combustible Gas

Sensitive material of MQ-2 gas sensor is $SnO_2$, which with lower conductivity in clean air. When the target combustible gas exist, The sensor's conductivity is more higher along with the gas concentration rising. Please use simple electrocircuit, Convert change of conductivity to correspond output signal of gas concentration.

MQ-2 gas sensor has high sensitity to LPG, Propane and Hydrogen, also could be used to Methane and other combustible steam, it is with low cost and suitable for different application.

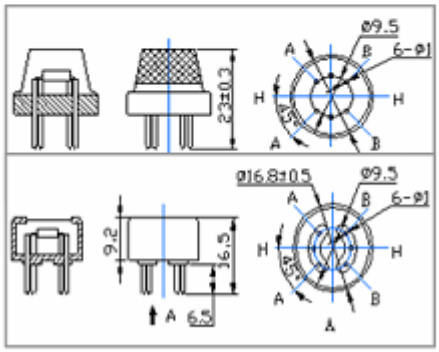
## Character

*Good sensitivity to Combustible gas in wide range
* High sensitivity to LPG, Propane and Hydrogen
* Long life and low cost
* Simple drive circuit

## Application

* Domestic gas leakage detector
* Industrial Combustible gas detector
* Portable gas detector

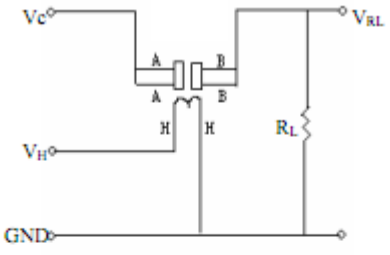
## Configuration




## Technical Data

| | | | |
|---|---|---|---|
| Model No. | | | MQ-2 |
| Sensor Type | | | Semiconductor |
| Standard Encapsulation | | | Bakelite (Black Bakelite) |
| Detection Gas | | | Combustible gas and smoke |
| Concentration | | | 300-10000ppm ( Combustible gas) |
| Circuit | Loop Voltage | $V_c$ | ≤24V DC |
| | Heater Voltage | $V_H$ | 5.0V±0.2V ACorDC |
| | Load Resistance | $R_L$ | Adjustable |
| Character | Heater Resistance | $R_H$ | 31Ω±3Ω（Room Tem.） |
| | Heater consumption | $P_H$ | ≤900mW |
| | Sensing Resistance | $R_s$ | 2KΩ-20KΩ(in 2000ppm $C_4H_{10}$) |
| | Sensitivity | S | Rs(in air)/Rs(1000ppm isobutane)≥5 |
| | Slope | α | ≤0.6($R_{5000ppm}/R_{3000ppm}$ $CH_4$) |
| Condition | Tem. Humidity | | 20℃±2℃；65%±5%RH |
| | Standard test circuit | | Vc:5.0V±0.1V; $V_H$: 5.0V±0.1V |
| | Preheat time | | Over 48 hours |

## Basic test loop




The above is basic test circuit of the sensor. The sensor need to be put 2 voltage, heater voltage（VH） and test voltage（VC）. VH used to supply certified working temperature to the sensor, while VC used to detect voltage (VRL) on load resistance （RL） whom is in series with sensor. The sensor has light polarity, Vc need DC power. VC and VH could use same power circuit with precondition to assure performance of sensor. In order to make the sensor with better performance, suitable RL value is needed:

Power of Sensitivity body(Ps):

$Ps=Vc^2\times Rs/(Rs+RL)^2$

Resistance of sensor(Rs): Rs=(Vc/VRL-1)×RL

**Sensitivity Characteristics**

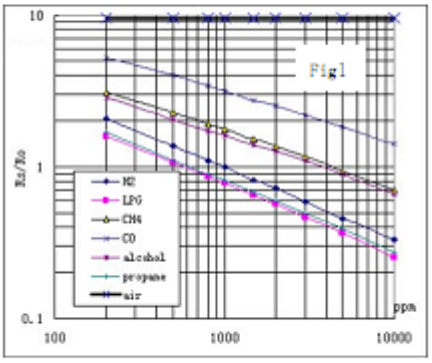



Fig.1 shows the typical sensitivity characteristics of the MQ-2, ordinate means resistance ratio of the sensor (Rs/Ro), abscissa is concentration of gases. Rs means resistance in different gases, Ro means resistance of sensor in 1000ppm Hyrogen. All test are under standard test conditions.

**Influence of Temperature/Humidity**

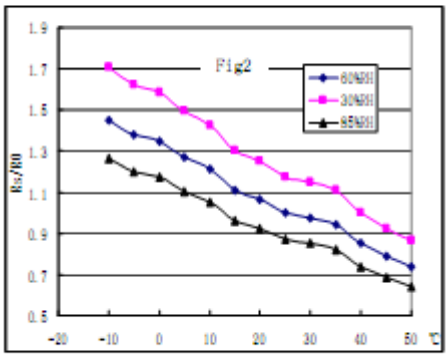
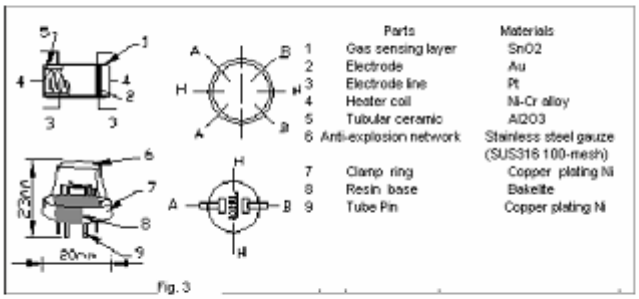



Fig.2 shows the typical temperature and humidity characteristics. Ordinate means resistance ratio of the sensor (Rs/Ro), Rs means resistance of sensor in 1000ppm Butane under different tem. and humidity. Ro means resistance of the sensor in environment of 1000ppm Methane, 20℃/65%RH

**Structure and configuration**




Structure and configuration of MQ-2 gas sensor is shown as Fig. 3, sensor composed by micro AL2O3 ceramic tube, Tin Dioxide (SnO2) sensitive layer, measuring electrode and heater are fixed into a crust made by plastic and stainless steel net. The heater provides necessary work conditions for work of sensitive components. The enveloped MQ-2 have 6 pin, 4 of them are used to fetch signals, and other 2 are used for providing heating current.

**Notification**

**1 Following conditions must be prohibited**

1.1 Exposed to organic silicon steam

Organic silicon steam cause sensors invalid, sensors must be avoid exposing to silicon bond, fixature, silicon latex, putty or plastic contain silicon environment

1.2 High Corrosive gas

If the sensors exposed to high concentration corrosive gas (such as $H_2Sz$, $SO_X$, $Cl_2$, HCl etc), it will not only result in corrosion of sensors structure, also it cause sincere sensitivity attenuation.

1.3 Alkali, Alkali metals salt, halogen pollution

The sensors performance will be changed badly if sensors be sprayed polluted by alkali metals salt especially brine, or be exposed to halogen such as fluorin.

1.4 Touch water

Sensitivity of the sensors will be reduced when spattered or dipped in water.

1.5 Freezing

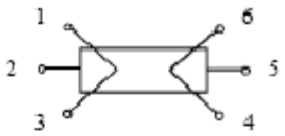
Do avoid icing on sensor'surface, otherwise sensor would lose sensitivity.

1.6 Applied voltage higher

Applied voltage on sensor should not be higher than stipulated value, otherwise it cause down-line or heater damaged, and bring on sensors' sensitivity characteristic changed badly.

1.7 Voltage on wrong pins

For 6 pins sensor, if apply voltage on 1、3 pins or 4、6 pins, it will make lead broken, and without signal when apply on 2、4 pins




**2 Following conditions must be avoided**

2.1 Water Condensation

Indoor conditions, slight water condensation will effect sensors performance lightly. However, if water condensation on sensors surface and keep a certain period, sensor' sensitivity will be decreased.

2.2 Used in high gas concentration

No matter the sensor is electrified or not, if long time placed in high gas concentration, if will affect sensors characteristic.

2.3 Long time storage

The sensors resistance produce reversible drift if it's stored for long time without electrify, this drift is related with storage conditions. Sensors should be stored in airproof without silicon gel bag with clean air. For the sensors with long time storage but no electrify, they need long aging time for stbility before using.

2.4 Long time exposed to adverse environment

No matter the sensors electrified or not, if exposed to adverse environment for long time, such as high humidity, high temperature, or high pollution etc, it will effect the sensors performance badly.

2.5 Vibration

Continual vibration will result in sensors down-lead response then repture. In transportation or assembling line, pneumatic screwdriver/ultrasonic welding machine can lead this vibration.

2.6 Concussion

If sensors meet strong concussion, it may lead its lead wire disconnected.

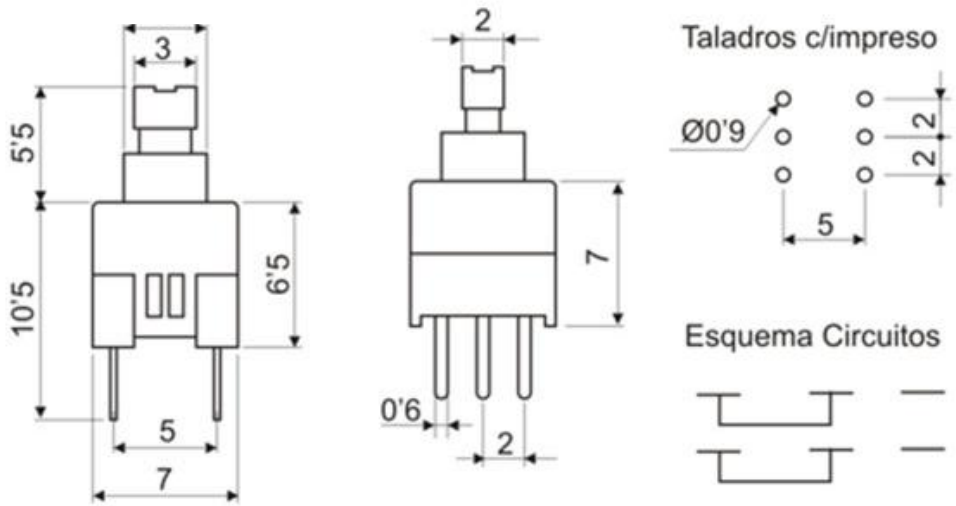
2.7 Usage

For sensor, handmade welding is optimal way. If use wave crest welding should meet the following conditions:

2.7.1 Soldering flux: Rosin soldering flux contains least chlorine
2.7.2 Speed: 1-2 Meter/ Minute
2.7.3 Warm-up temperature：100±20℃
2.7.4 Welding temperature：250±10℃
2.7.5 1 time pass wave crest welding machine

If disobey the above using terms, sensors sensitivity will be reduced.

- SWITCH



Tyco Electronics — General Purpose Relays — SCHRACK

# Power PCB Relay RT1

- **1 pole 12 / 16 A, 1 CO or 1 NO contact**
- **DC- or AC-coil**
- **Sensitive coil 400 mW**
- **5 kV / 10 mm coil-contact, reinforced insulation**
- **Ambient temperature 85°C (DC-coil)**
- **WG version: Product in accordance to IEC60335-1**

F0144-C

**Applications**

**Boiler control, timers, garage door control, POS automation, interface modules**

## Approvals

VDE REG.-Nr. 6106, cULus E214025, CSA 14385, SEAR C0786
Technical data of approved types on request

## Contact data

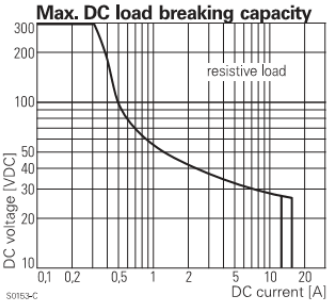
| | | |
|---|---|---|
| Contact configuration | 1 CO or 1 NO contact | |
| Contact set | single contact | |
| Type of interruption | micro disconnection | |
| Rated voltage / max. switching voltage AC | 250 / 400 VAC | |
| Rated current | 12 A | 16 A |
| Limiting continuous current | 12 A | 16 A, UL: 20 A |
| Maximum breaking capacity AC | 3000 VA | 4000 VA |
| Limiting making capacity, max 4 s, df 10% | 25 A | 30 A |
| Contact material | AgNi 90/10, AgNi 90/10 gold plated | |
| Rated frequency of operation with / without load | DC coil: 6 / 1200 min$^{-1}$<br>AC coil: 6 / 600 min$^{-1}$ | |
| Operate- / release time DC coil | max 8 / 6 ms | |
| Bounce time DC coil, NO / NC contact | max 4 / 6 ms | |

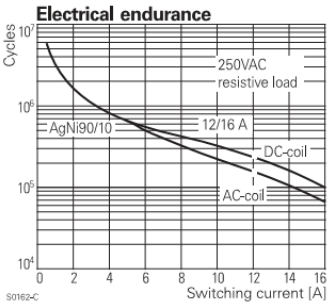
## Contact ratings

| Type | Contact | Load | Ambient temp. [°C] | Cycles |
|---|---|---|---|---|
| **IEC 61810** | | | | |
| RT314 DC-coil | NO | 16 A, 250 VAC, cosφ=1 | 85°C | 30x10$^3$ |
| RT314 DC-coil | CO | 16 A, 250 VAC, cosφ=1 | 85°C | 10x10$^3$ |
| RT314 DC-coil | NO | 10 A, 400 VAC, cosφ=1 | 85°C | 150x10$^3$ |
| RT114 DC-coil | NO | 12 A, 250 VAC, cosφ=1 | 85°C | 50x10$^3$ |
| RT114 AC-coil | NO | 12 A, 250 VAC, cosφ=1 | 70°C | 100x10$^3$ |
| **UL 508** | | | | |
| RT314 | NO / NC | 20 A, 250 VAC, general purpose | 85°C | 6x10$^3$ |
| RT334 | NO | 16 A, 250 VAC, general purpose | 85°C | 50x10$^3$ |
| RT314 | NO | 1 hp, 240 VAC | 40°C | 1x10$^3$ |
| **EN60947-5-1** | | | | |
| RT314 DC-coil | NO / NC | 2 A, 24 VDC DC13 | | 6.050 |
| **EN60730-1** | | | | |
| RT314 DC-coil | NO | 12(2) A, 250 VAC | 85°C | 100x10$^3$ |

## Coil data

| | |
|---|---|
| Rated coil voltage range DC coil | 5...110 VDC |
| AC coil | 24...230 VAC |
| Operative range to IEC 61810 | 2 |
| Coil insulation system according UL1446 | class F |




Datasheet Rev. HK1
Issued 2008/11
www.tycoelectronics.com
www.schrackrelays.com

Dimensions are in mm unless otherwise specified and are shown for reference purposes only.

Product specification according to IEC 61810-1. Product data, technical parameters, test conditions and processing information only to be used together with the 'Definitions' section in the catalogue or at schrackrelays.com in the 'Schrack' section.

Specifications subject to change.

## Power PCB Relay RT1 (Continued)

**Coil versions, DC-coil**

| Coil code | Rated voltage VDC | Operate voltage VDC | Release voltage VDC | Coil resistance Ohm | Rated coil power mW |
|---|---|---|---|---|---|
| 005 | 5 | 3.5 | 0.5 | 62±10% | 403 |
| 006 | 6 | 4.2 | 0.6 | 90±10% | 400 |
| 009 | 9 | 6.3 | 0.9 | 200±10% | 400 |
| 012 | 12 | 8.4 | 1.2 | 360±10% | 400 |
| 024 | 24 | 16.8 | 2.4 | 1440±10% | 400 |
| 048 | 48 | 33.6 | 4.8 | 5520±10% | 417 |
| 060 | 60 | 42.0 | 6.0 | 8570±12% | 420 |
| 110 | 110 | 77.0 | 11.0 | 28800±12% | 420 |

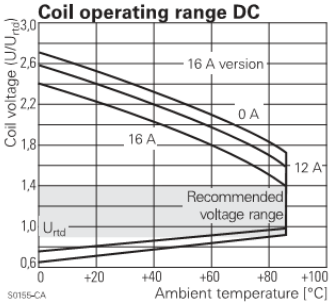
All figures are given for coil without preenergization, at ambient temperature +23°C
Other coil voltages on request

**Coil versions, AC-coil 50Hz**

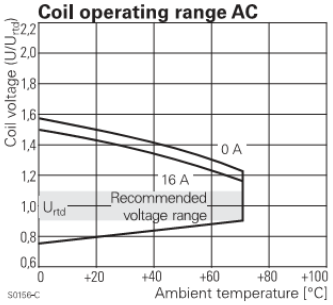
| Coil code | Rated voltage VAC | Operate voltage 50 Hz VAC | Release voltage 50 Hz VAC | Coil resistance Ohm | Rated coil power 50 Hz VA |
|---|---|---|---|---|---|
| 524 | 24 | 18.0 | 3.6 | 350±10% | 0.76 |
| 615 | 115 | 86.3 | 17.3 | 8100±15% | 0.76 |
| 620 | 120 | 90.0 | 18.0 | 8800±15% | 0.75 |
| 700 | 200 | 150.0 | 30.0 | 24350±15% | 0.76 |
| 730 | 230 | 172.5 | 34.5 | 32500±15% | 0.74 |

All figures are given for coil without preenergization, at ambient temperature +23°C

**Coil operating range DC**




**Coil operating range AC**




**Insulation**

| | | |
|---|---|---|
| Dielectric strength coil-contact circuit | | 5000 $V_{rms}$ |
| open contact circuit | | 1000 $V_{rms}$ |
| Clearance / creepage coil-contact circuit | | ≥ 10 / 10 mm |
| Material group of insulation parts | | IIIa |
| Tracking index of relay base | | PTI 250 V |
| Insulation to IEC 61810-1 | | |
| Type of insulation coil-contact circuit | | reinforced |
| open contact circuit | | micro disconnection |
| Rated insulation voltage | | 250 V |
| Pollution degree 12 A version | 3 | 3 |
| 16 A version | 3 | 2 |
| Rated voltage system | 240 V | 230 / 400 V |
| Overvoltage category | | III |

**Other data**

| | |
|---|---|
| Mechanical endurance DC coil | > 30 x $10^6$ cycles |
| AC coil | > 10 x $10^6$ cycles |
| Material | |
| RoHS - Directive 2002/95/EC | compliant as per product date code 0413 |
| Resistance to heat and fire, WG version | according EN60335-1, par.30 |
| Environment | |
| Ambient temperature range DC coil | -40...+85°C |
| AC coil | -40...+70°C |
| Vibration resistance (function), NO / NC cont. | 20 / 5 g, 30 ... 500 Hz |
| Shock resistance (destruction) | 100 g |
| Category of protection | RTII - flux proof, RTIII - wash tight |
| Processing | |
| Mounting | pcb or on socket |
| Mounting position | any |
| Mounting distance DC / AC coils | ≥ 0 / ≥ 2.5 mm |
| Resistance to soldering heat flux-proof version | 270°C / 10 s |
| wash-tight version | 260°C / 5 s |
| Relay weight | 14 g |
| Packaging unit | 20 / 500 pcs |

**Accessories**

| | |
|---|---|
| For details see datasheet | Accessories Power Relay RT |

Datasheet Rev. HK1
Issued 2008/11
www.tycoelectronics.com
www.schrackrelays.com

Dimensions are in mm unless otherwise specified and are shown for reference purposes only.

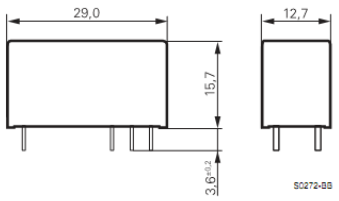
Product specification according to IEC 61810-1. Product data, technical parameters, test conditions and processing information only to be used together with the 'Definitions' section in the catalogue or at schrackrelays.com in the 'Schrack' section.

Specifications subject to change.

Tyco Electronics — General Purpose Relays — SCHRACK

## Power PCB Relay RT1 (Continued)

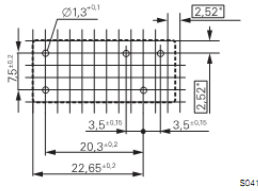
**Dimensions**



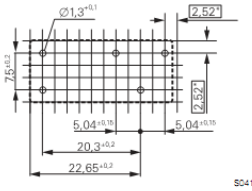

**PCB layout / terminal assignment**
Bottom view on solder pins

*) With the recommended PCB hole sizes a grid pattern from 2.5 mm to 2.54 mm can be used.

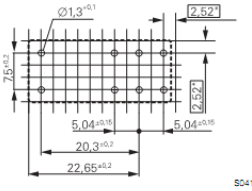
12 A, pinning 3.5 mm



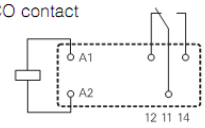

12 A, pinning 5 mm



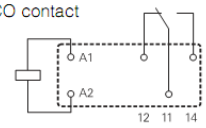

16 A, pinning 5 mm



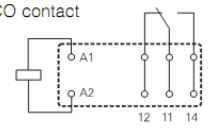

1 CO contact



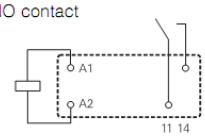

1 CO contact



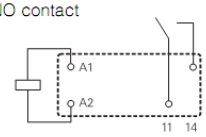

1 CO contact



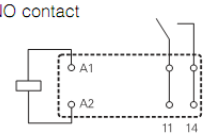

1 NO contact




1 NO contact




1 NO contact




**Product key** — Typical product key: **RT 3 1 4 024**

**Type**
**RT** Power PCB Relay RT1

**Version**

| | | | |
|---|---|---|---|
| **1** | **12 A, pinning 3.5 mm, flux proof** | **B** | 12 A, pinning 3.5 mm, wash tight |
| **2** | 12 A, pinning 5 mm, flux proof *) | | |
| **3** | **16 A, pinning 5 mm, flux proof** | **D** | 16 A, pinning 5 mm, wash tight |

**Contact configuration**

| | | | |
|---|---|---|---|
| **1** | 1 CO contact (1 form C) | **3** | 1 NO contact (1 form A) |

**Contact material**

| | | | |
|---|---|---|---|
| **4** | AgNi 90/10 | **5** | AgNi 90/10 gold plated (for type RT31.) |

**Coil**
Coil code: please refer to coil versions table

**Version**
**Blank** Standard version
**WG** Product in accordance with IEC 60335-1 (domestic appliances)

Preferred types in bold print
*) Wash tight version on request

Datasheet Rev. HK1
Issued 2008/11
www.tycoelectronics.com
www.schrackrelays.com

Dimensions are in mm unless otherwise specified and are shown for reference purposes only.

Product specification according to IEC 61810-1. Product data, technical parameters, test conditions and processing information only to be used together with the 'Definitions' section in the catalogue or at schrackrelays.com in the 'Schrack' section.

Specifications subject to change.

Tyco Electronics — General Purpose Relays — SCHRACK

## Power PCB Relay RT1 (Continued)

| Product key | Version | Contacts | Cont. material | Coil | Coil | Part number |
|---|---|---|---|---|---|---|
| RT114005 | 12 A pinning 3.5 mm flux proof | 1 CO contact | AgNi 90/10 | DC-coil | 5 VDC | 1393239-7 |
| RT114006 | | | | | 6 VDC | 1393239-8 |
| RT114012 | | | | | 12 VDC | 1419108-1 |
| RT114024 | | | | | 24 VDC | 1-1393239-3 |
| RT114048 | | | | | 48 VDC | 1-1393239-4 |
| RT114110 | | | | | 110 VDC | 1-1393239-6 |
| RT114524 | | | | AC-coil | 24 VAC | 1-1393239-7 |
| RT114615 | | | | | 115 VAC | 1-1393239-8 |
| RT114730 | | | | | 230 VAC | 1-1393239-9 |
| RT134012 | | 1 NO contact | | DC-coil | 12 VDC | 2-1393239-6 |
| RT134024 | | | | | 24 VDC | 3-1393239-0 |
| RT214012 | 12 A, pinning 5mm flux proof | 1 CO contact | | | 12 VDC | 5-1393239-4 |
| RT214024 | | | | | 24 VDC | 5-1393239-5 |
| RT314005 | 16 A pinning 5 mm flux proof | | | | 5 VDC | 9-1393239-1 |
| RT314006 | | | | | 6 VDC | 9-1393239-3 |
| RT314012 | | | | | 12 VDC | 9-1393239-5 |
| RT314024 | | | | | 24 VDC | 9-1393239-8 |
| RT314048 | | | | | 48 VDC | 1393240-1 |
| RT314060 | | | | | 60 VDC | 1393240-2 |
| RT314110 | | | | | 110 VDC | 1393240-3 |
| RT314524 | | | | AC-coil | 24 VAC | 1393240-4 |
| RT314615 | | | | | 115 VAC | 1393240-6 |
| RT314730 | | | | | 230 VAC | 1393240-7 |
| RT315012 | | | AgNi 90/10 gold plated | DC-coil | 12 VDC | 1-1393240-1 |
| RT315024 | | | | | 24 VDC | 1-1393240-4 |
| RT315730 | | | | AC-coil | 230 VAC | 1-1419108-1 |
| RT334012 | | 1 NO contact | AgNi 90/10 | DC-coil | 12 VDC | 4-1393240-5 |
| RT334024 | | | | | 24 VDC | 4-1393240-8 |
| RT334048 | | | | | 48 VDC | 5-1393240-0 |
| RTB14005 | 12 A pinning 3.5 mm wash tight | 1 CO contact | | | 5 VDC | 1-1393238-2 |
| RTB14012 | | | | | 12 VDC | 1-1393238-5 |
| RTB14024 | | | | | 24 VDC | 1-1393238-9 |
| RTB14048 | | | | | 48 VDC | 2-1393238-1 |
| RTD14005 | 16 A pinning 5 mm wash tight | | | | 5 VDC | 5-1393238-9 |
| RTD14006 | | | | | 6 VDC | 6-1393238-0 |
| RTD14012 | | | | | 12 VDC | 6-1393238-2 |
| RTD14015 | | | | | 15 VDC | 6-1393238-4 |
| RTD14024 | | | | | 24 VDC | 6-1393238-8 |
| RTD14048 | | | | | 48 VDC | 6-1393238-9 |
| RTD34005 | | 1 NO contact | | | 5 VDC | 8-1393238-3 |
| RTD34012 | | | | | 12 VDC | 3-1419108-5 |
| RTD34024 | | | | | 24 VDC | 3-1419108-8 |

Referencias Electrónica Embajadores:

(1 circuito conmutado de 12A - 5 pines)

| | |
|---|---|
| RT114005: RL3A405 | 5 VCC |
| RT114006: RL3A406 | 6 VCC |
| RT114012: RL3A412 | 12 VCC |
| RT114024: RL3A424 | 24 VCC |
| RT114048: RL3A448 | 48 VCC |
| RT114060: RL3A460 | 60 VCC |
| RT114110: RL3A461 | 110 VCC |
| RT114524: RL3A470 | 24 VAC |
| RT114615: RL3A471 | 110 VAC |
| RT114730: RL3A473 | 220 VAC |

Referencias Electrónica Embajadores:

(1 circuito conmutado de 16A - 8 pines)

| | |
|---|---|
| RT314005: RL3A305 | 5 VCC |
| RT314006: RL3A306 | 6 VCC |
| RT314012: RL3A312 | 12 VCC |
| RT314024: RL3A324 | 24 VCC |
| RT314048: RL3A348 | 48 VCC |
| RT314060: RL3A360 | 60 VCC |
| RT314110: RL3A361 | 110 VCC |
| RT314524: RL3A370 | 24 VAC |
| RT314615: RL3A371 | 110 VAC |
| RT314730: RL3A373 | 220 VAC |

Datasheet Rev. HK1
Issued 2008/11
www.tycoelectronics.com
www.schrackrelays.com

Dimensions are in mm unless otherwise specified and are shown for reference purposes only.

Product specification according to IEC 61810-1. Product data, technical parameters, test conditions and processing information only to be used together with the 'Definitions' section in the catalogue or at schrackrelays.com in the 'Schrack' section.

Specifications subject to change.

## NPN SILICON PLANAR SWITCHING TRANSISTORS

**2N2221**
**2N2222**


**TO-18**
**Metal Can Package**

**Switching and Linear Application DC and VHF Amplifier Applications**

**ABSOLUTE MAXIMUM RATINGS (Ta=25°C unless specified otherwise)**

| DESCRIPTION | SYMBOL | 2N2221, 22 | UNIT |
|---|---|---|---|
| Collector Emitter Voltage | $V_{CEO}$ | 30 | V |
| Collector Base Voltage | $V_{CBO}$ | 60 | V |
| Emitter Base Voltage | $V_{EBO}$ | 5 | V |
| Collector Current Continuous | $I_C$ | 800 | mA |
| Power Dissipation @Ta=25°C | $P_D$ | 500 | mW |
| Derate Above 25°C | | 2.28 | mW/°C |
| Power Dissipation @ Tc=25°C | $P_D$ | 1.2 | W |
| Derate Above 25°C | | 6.85 | mW/°C |
| Operating and Storage Junction Temperature Range | $T_j$, $T_{stg}$ | -65 to +200 | °C |

**ELECTRICAL CHARACTERISTICS (Ta=25°C unless specified otherwise )**

| DESCRIPTION | SYMBOL | TEST CONDITION | VALUE MIN | VALUE MAX | UNIT |
|---|---|---|---|---|---|
| Collector Emitter Breakdown Voltage | $BV_{CEO}$ | $I_C$=10mA,$I_B$=0 | 30 | | V |
| Collector Base Breakdown Voltage | $BV_{CBO}$ | $I_C$=10µA,$I_E$=0 | 60 | | V |
| Emitter Base Breakdown Voltage | $BV_{EBO}$ | $I_E$=10µA, $I_C$=0 | 5 | | V |
| Collector Leakage Current | $I_{CBO}$ | $V_{CB}$=50V, $I_E$=0 | | 10 | nA |
| | | $V_{CB}$=50V, $I_E$=0 Ta=150 ° C | | 10 | µA |
| Collector Emitter Saturation Voltage | $V_{CE(Sat)}$* | $I_C$=150mA,$I_B$=15mA | | 0.4 | V |
| | | $I_C$=500mA,$I_B$=50mA | | 1.6 | V |
| Base Emitter Saturation Voltage | $V_{BE(Sat)}$* | $I_C$=150mA,$I_B$=15mA | 0.6 | 1.3 | V |
| | | $I_C$=500mA,$I_B$=50mA | | 2.6 | V |

**ELECTRICAL CHARACTERISTICS (Ta=25°C unless specified otherwise)**

| DESCRIPTION | SYMBOL | TEST CONDITION | 2221 MIN | 2221 MAX | 2222 MIN | 2222 MAX | UNIT |
|---|---|---|---|---|---|---|---|
| DC Current Gain | $h_{FE}$ | $I_C$=0.1mA,$V_{CE}$=10V* | 20 | | 35 | | |
| | | $I_C$=1mA,$V_{CE}$=10V | 25 | | 50 | | |
| | | $I_C$=10mA,$V_{CE}$=10V* | 35 | | 75 | | |
| | | $I_C$=150mA,$V_{CE}$=1V* | 20 | | 50 | | |
| | | $I_C$=150mA,$V_{CE}$=1V* | 40 | 120 | 100 | 300 | |
| | | $I_C$=500mA,$V_{CE}$=10V* | 20 | | 30 | | |
| **DYNAMIC CHARACTERISTICS** | | | | | | | |
| Transition Frequency | $f_T$ | $I_C$=20mA, $V_{CE}$=20V f=100MHz | 250 | | 250 | | MHz |
| Output Capacitance | $C_{ob}$ | $V_{CB}$=10V, $I_E$=0 f=100KHz | | 8 | | 8 | pF |
| Input Capacitance | $C_{ib}$ | $V_{EB}$=0.5V, $I_C$=0 f=100kHz | | 30 | | 30 | pF |
| **SWITCHING CHARACTERISTICS** | | | | | | | |
| Delay time | $t_d$ | | | | | 10 | ns |
| | | $I_C$=150mA,IB1=15mA | | | | | |
| Rise time | $t_r$ | $V_{CC}$=30V,$V_{BE(off)}$=0.5V | | | | 25 | ns |
| Storage time | $t_s$ | | | | | 225 | ns |
| | | $I_C$=150mA, IB1=15mA | | | | | |
| Fall time | $t_f$ | IB2=15mA, $V_{CC}$=30V | | | | 60 | ns |

*Pulse Condition: Pulse Width ≤300µs, Duty Cycle ≤2%

# ANEXO III.

# MANUAL DE INSTALACIÓN

Este pequeño manual es una guía de instalación de los recursos necesarios para la puesta en marcha del sistema.

- ## INSTALACIÓN DEL SERVIDOR APACHE Y MYSQL

Para llevar a cabo la instalación del servidor *Apache* y la base de datos *MySQL* en *Ubuntu 14.04*, abrir la aplicación *terminal* y escribir los siguientes comandos[21]:

INSTALACIÓN APACHE2

```
sudo apt-get update
```

```
sudo apt install apache2
```

INSTALACIÓN PHP5

```
sudo apt install php5 libapache2-mod-php5
```

```
sudo /etc/init.d/apache2 restart
```

```
sudo chmod -R 755 /var/www/html
```

```
nano /var/www/html/info.php
```

```
(dentro del fichero escribir <?php phpinfo();?>)
```

Dirigirse a la dirección *http://localhost/info.php* para comprobar el éxito de la instalación.

INSTALACIÓN *MYSQL*

```
sudo apt install MySQL-server MySQL-client
```

```
sudo /etc/init.d/apache2 restart
```

[21] Para más información sobre la instalación visitar http://lignux.com/guia-instalacion-de-apache2-en-ubuntu-14-04

## • INSTALACIÓN DE HERRAMIENTAS DE COMPILACIÓN EN LINUX

COMPILADOR G++

Primero de todo es necesario instalar el conjunto de herramientas de compilación de *C++*, bajo la herramienta **g++**.

Para ello abrir el *terminal* de *Linux* y teclear:

```
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
```

```
sudo apt-get update
```

```
sudo apt-get install g++-4.9
```

Una vez instalado el compilador, se procede a instalar el resto de dependencias necesarias para compilar el proyecto *icra*.

COMUNICACIÓN SERIAL PORT

```
sudo apt-get install libserial-dev
```

CONEXIÓN *MYSQL*

```
sudo apt-get install lib MySQL++ lib MySQL++-dev
```

SERVICIOS REST

Para poder llevar a cabo los servicios REST en el equipo, es necesario descargar e instalar el paquete *Boost C++* del siguiente enlace: *http://www.boost.org/*

Instalar el fichero "*boost_1_57_0*" en el directorio *usr/include.*

El fichero descargado incluye una guía de instalación en el sistema Ubuntu.

Tras cumplir todos los pasos, ejecutar el siguiente comando:

```
sudo apt-get install g++-4.8 g++ git make libboost1.54-all-dev libssl-dev cmake
```

Al finalizar el proceso, reiniciar el equipo.

## • INSTALACIÓN DE LA PLACA ARDUINO

A partir de la versión *12.02* de *Ubuntu*, la instalación del soporte software de *Arduino* es muy sencilla.

Abrir la aplicación *terminal* y ejecutar la siguiente línea de código:

```
sudo apt-get update && sudo apt-get install Arduino Arduino-core
```

Una vez instalada la versión estable, clicar en el siguiente enlace para descargar el paquete software con el programa

http://arduino.cc/en/Main/Software

Descomprimir el fichero en un directorio del equipo y ejecutar el archivo *arduino*.[22]

Una vez abierta la aplicación de *Arduino*, ir a *Sketch->Importar Librería->Add Library…*Buscar en la ventana que se abrirá el fichero *DHT.zip* ubicado en la carpeta *ProyectoIcra / Arduino / Recursos* en el CD del proyecto *Icra*. Es necesario importar la librería del sensor *DHT* para poder llevar a cabo la compilación del código.

Por último, pulsar *Ctrl+O* y abrir el fichero del proyecto *icra.ino* ubicado en la carpeta *ProyectoIcra / Arduino / icra*

Pulsar en el botón de *Verificar* para compilar el código.

## • INSTALACIÓN ICRAPP

Conectar el dispositivo *Android* al *PC* mediante el puerto *USB.* Activar la conexión *USB*. Copiar el fichero *(Ver "Tabla 9")* **ICRApp.apk** ubicado en el directorio */ProyectoIcra /Android* del *CD* en el *Smartphone*. Deshabilitar la comunicación *USB*.

Activar la opción "*Orígenes desconocidos*" ubicada en *Ajustes->Aplicaciones* dentro del dispositivo móvil[23]. Con ello se permite instalar aplicaciones *Android* ajenas al servicio *Google Play Store*.

Ejecutar el fichero *ICRApp.apk*. Aceptar e instalar en las siguientes ventanas para completar el proceso.

**Tabla 9: ICRApp requisitos sistema**

| Tamaño instalador | Tamaño app instalada | Acceso de dispositivo | Versión Android |
|---|---|---|---|
| 2,7 Mb | 3,95 Mb | Acceso a Internet sin límites<br>Vibrador de control | 2.2 o superior |

[22] Para más información sobre la instalación visitar http://playground.*Arduino*.cc/Linux/Ubuntu
[23] La ubicación de la opción "Orígenes desconocidos" puede variar en función del modelo y versión del dispositivo

## • COMPILACIÓN SERVERICRA

Una vez instaladas las herramientas de compilación, se puede llevar a cabo la construcción del ejecutable **serverICRA**.

Para ello es necesario copiar la carpeta */ProyectoIcra/Servidor/serverICRA* del *CD* del proyecto en el directorio del equipo *var/www/*

Tras ello, ejecutar la aplicación *terminal* de *Linux* y utilizar la instrucción *cd* para desplazarse al directorio del servidor: *cd /…/serverICRA*

Ejecutar las siguientes instrucciones:

```
g++ -std=C++11 -Iserial -I /usr/include/MySQL++/ -I /usr/include/MySQL -c
-o serverICRA.o serverICRA.cpp
```

```
g++ -o serverICRA serverICRA.o -lserial -pthread -lMySQLpp -lboost_system
```

Una vez finalizadas, teclear el ejecutable creado:

```
./serverICRA
```

## • COMPILACIÓN USERSICRA

Una vez instaladas las herramientas de compilación, se puede llevar a cabo la construcción del ejecutable **usersICRA**.

Para ello es necesario copiar la carpeta */ProyectoIcra/Servidor/usersICRA* del *CD* del proyecto en un directorio del equipo (preferentemente próximo al directorio raíz).

Tras ello, ejecutar la aplicación *terminal* de *Linux* y utilizar la instrucción *cd* para desplazarse al directorio del servidor: *cd /…/usersICRA*

Ejecutar las siguientes instrucciones:

```
g++ -I /usr/include/MySQL++/ -I /usr/include/MySQL -c -o usersICRA.o
usersICRA.cpp
```

```
g++ -o usersICRA usersICRA.o -lMySQLpp
```

```
./usersICRA
```

# ANEXO IV.

# MANUAL DE USUARIO SERVER ICRA

Este pequeño manual es una guía de uso de la aplicación *serverICRA* instalada en el equipo PC servidor, la cual ofrece los servicios de adquisición de datos y comunicación con los dispositivos móviles.

Para iniciar la aplicación *Servidor* de *ICRA* ha de seguirse los siguientes pasos:

1- Pegar la carpeta */ProyectoIcra/Servidor/serverICRA* ubicada en el disco óptico , en el directorio *var/www* del equipo.
2- Ejecutar la aplicación *terminal* de *Linux*
3- Utilizar el comando *cd* para desplazarse por los directorios del equipo hasta llegar a la carpeta *serverICRA* copiada en el paso 1.
4- Ejecutar el comando *./serverICRA*. En caso de no estar compilado el proyecto, ver el *"ANEXO III.Instalación de herramientas de compilación en Linux"*.

Una vez arrancada la aplicación *serverICRA* aparecerá la siguiente interfaz *(Ver "Figura 68"):*


Figura 68: serverICRA

Al ejecutar la aplicación se muestra la cabecera de presentación del proyecto. El cursor aguarda la inserción del usuario y contraseña para conectar con la base de datos local del equipo.

Tras insertar el usuario y la contraseña la aplicación permite seleccionar o crear (en caso de no existir la base de datos tecleada) la base de datos a emplear[24] *(Ver "Figura 69"):*

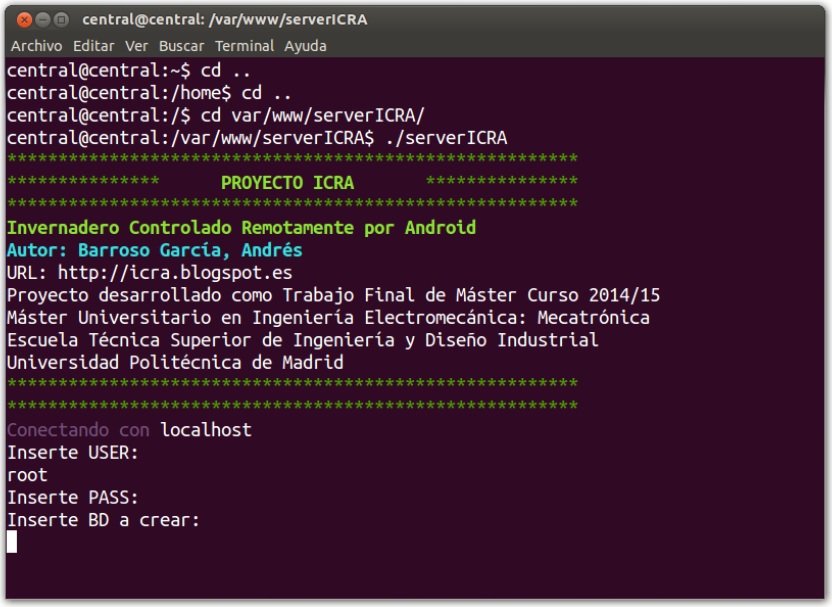



**Figura 69: serverICRA: Creación BD**

Definida la base de datos, la aplicación solicita la tabla a utilizar *(Ver "Figura 70").*

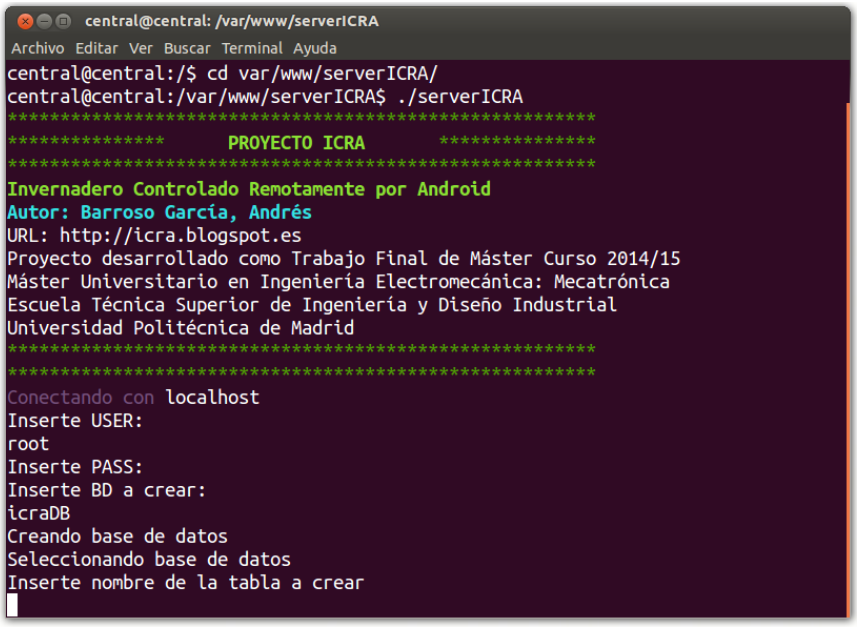



**Figura 70: serverICRA: Creación Tabla**

En el caso de existir la tabla introducida, la aplicación mostrará los últimos valores registrados por esta y pedirá al usuario insertar una tabla nueva.

[24] El *usuario* y *pass* a utilizar serán los que se hayan indicado al instalar la base de datos *MySQL* + Apache en el equipo PC.

Tras ser creada la base de datos y la tabla a utilizar, el siguiente punto es introducir una descripción sobre la muestra a controlar y monitorizar *(Ver "Figura 71").*


Figura 71: serverICRA: Descripción muestra

El texto insertado es guardado en un **fichero de texto *.txt*** en el directorio del *serverICRA.* El identificador del fichero es el nombre de la *BD* y de la *tabla* usada *(Ver "Figura 72").*


Figura 72: serverICRA: Cerrar registro

Presionar la tecla *ENTER* para pasar al siguiente paso.

Definida la base de datos y tabla a utilizar, y una vez que se ha creado el fichero de registro, el siguiente punto es indicar el puerto de comunicación con *Arduino*. Para ello es

necesario asegurarse de que, antes de insertar el puerto a utilizar, el equipo *Servidor* está conectado físicamente mediante *USB* con el microcontrolador *Arduino*.

Teclear el puerto de comunicación[25] (en este caso */dev/ttyACM0*) y presionar la tecla ENTER *(Ver "Figura 73").*

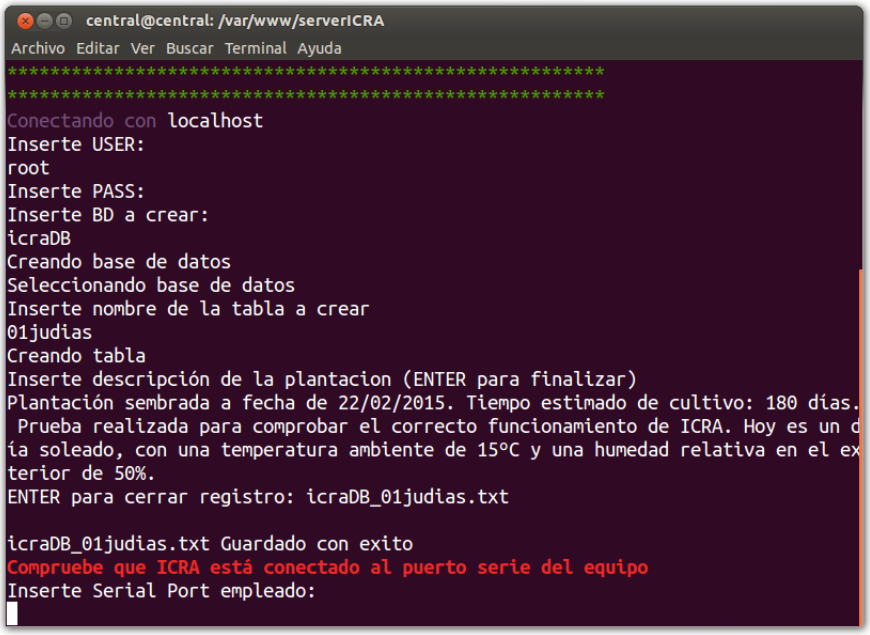



**Figura 73: serverICRA: Puerto comunicación**

Insertado el puerto serie de comunicación, el equipo comenzará a comunicar con la placa *Arduino* y a recibir información proveniente del invernadero *(Ver "Figura 74")*.

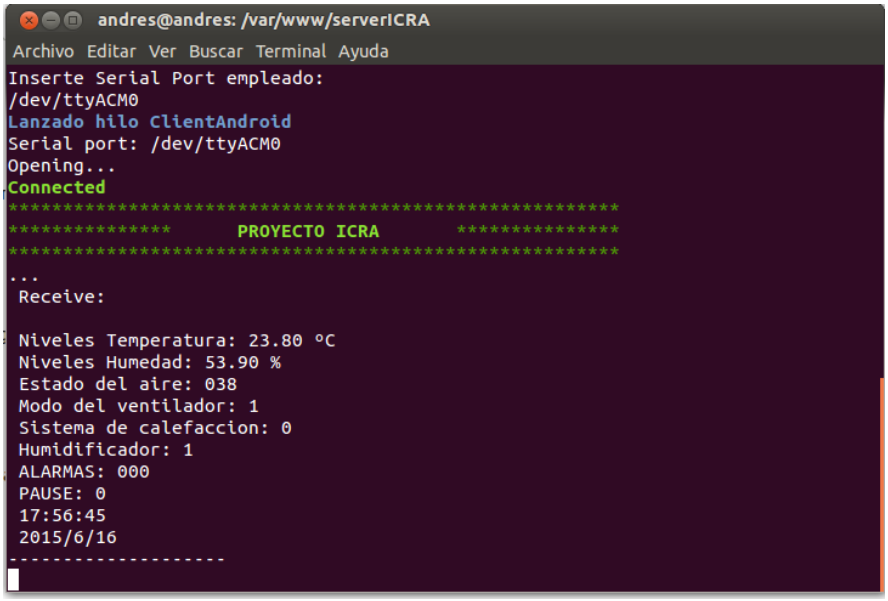



**Figura 74: serverICRA: Comunicación Arduino**

[25]Para acceder al puerto serie COMx en Linux, la instrucción utilizada es: /dev/ttyACMx. Utilizar el software de *Arduino* para detectar el puerto en el que está conectada la tarjeta.

# ANEXO V.

# MANUAL DE USUARIO usersICRA

Este pequeño manual es una guía de uso de la aplicación *usersICRA* encargada de la gestión de usuarios en el sistema.

Seguir los siguientes pasos:

1- Pegar la carpeta */ProyectoIcra/Servidor/usersICRA* ubicada en el disco óptico, en el directorio *home* del equipo.
2- Ejecutar la aplicación *terminal* de *Linux*
3- Utilizar el comando *cd* para desplazarse por los directorios del equipo hasta llegar a la carpeta *usersICRA* copiada en el paso 1.
4- Ejecutar el comando *./usersICRA*. En caso de no estar compilado el proyecto, ver la sección *ANEXO III: Instalación de herramientas de compilación en Linux*.

Una vez arrancada la aplicación *usersICRA* aparecerá la siguiente interfaz *(Ver "Figura 75"):*

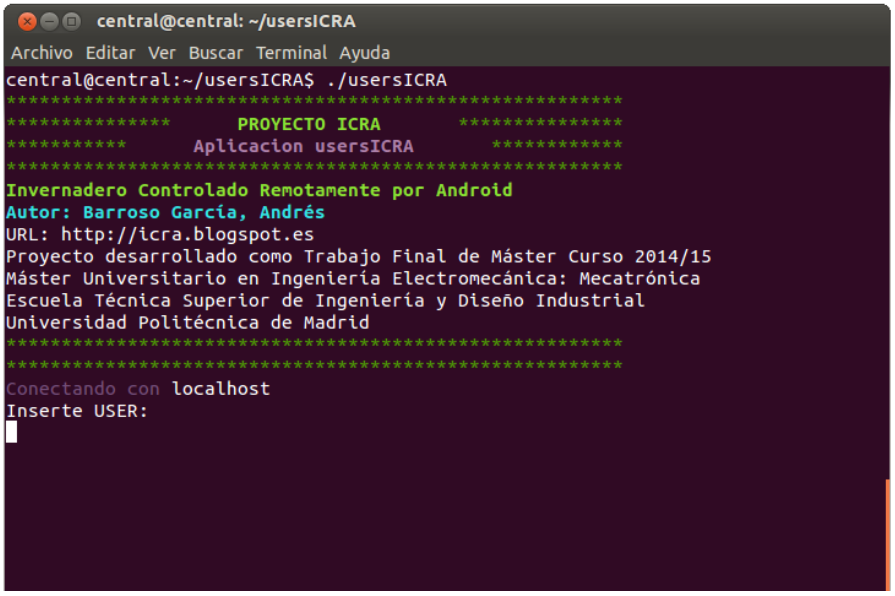



Figura 75: usersICRA: Pantalla inicial

La aplicación solicita el usuario y password de acceso al *localhost*. Tras insertar los datos se lanza el siguiente mensaje *(Ver "Figura 76"):*

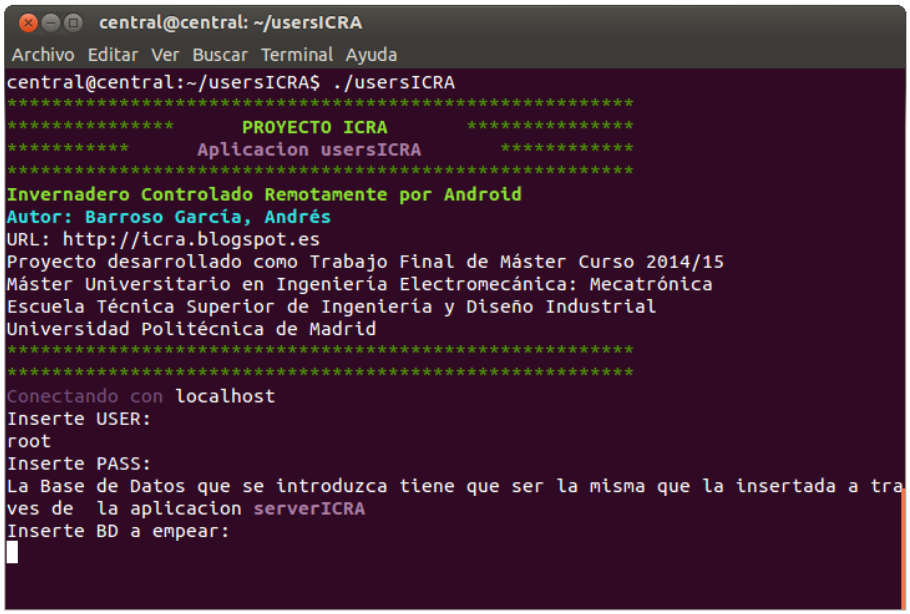



Figura 76: usersICRA: Insertar BD

La Base de Datos insertada ha de ser la misma que la empleada en la aplicación *serverICRA* para el correcto funcionamiento del sistema. Tras insertar la base de datos se muestra el menú principal *(Ver "Figura 77"):*

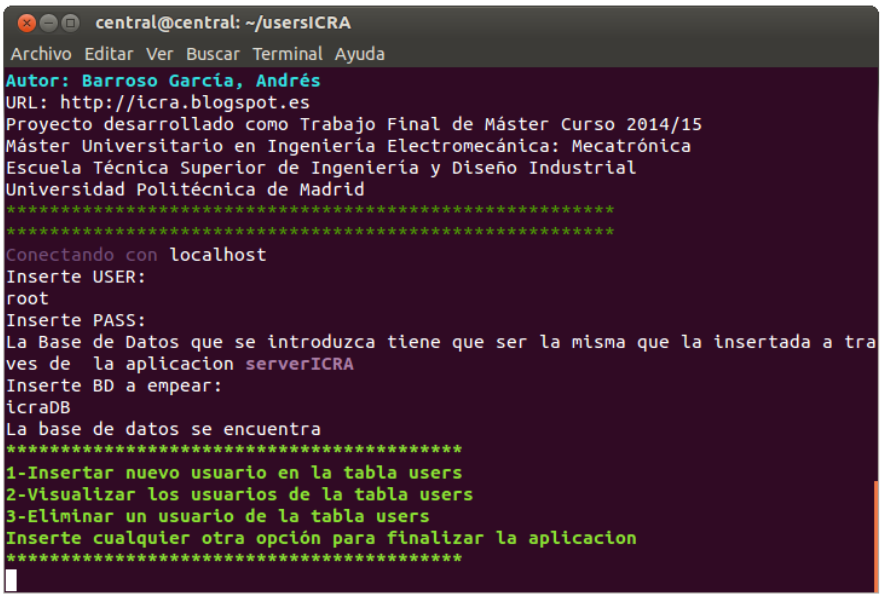



Figura 77: usersICRA: Menú Principal

A través del teclado numérico se pueden seleccionar las diferentes opciones de la aplicación. En el caso de querer insertar un nuevo usuario, pulsar la tecla "1" y rellenar los siguientes campos *(Ver "Figura 78"):*


Figura 78: usersICRA: Nuevo Usuario

La información a cumplimentar es:

- Nombre de usuario
- Contraseña de usuario, formada por 4 dígitos
- Nivel de usuario, existiendo dos modelos actualmente
    - ADMINISTRADOR: Funciones de super-usuario
    - USUARIO: Funciones de visualizador
- Email del usuario

Una vez finalizado el registro, pueden visualizarse todos los usuarios registrados en la tabla pulsando la tecla “2” en el menú principal *(Ver “Figura 79”):*

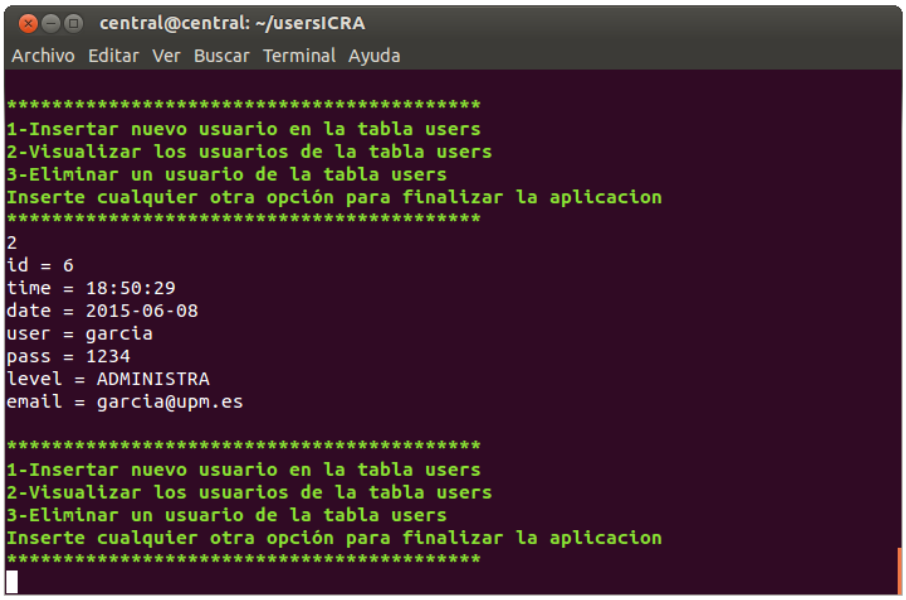



Figura 79: usersICRA: Visualizar usuarios

# ANEXO VI.

# MANUAL DE USUARIO ICRAPP

El siguiente manual es una pequeña guía de uso de la aplicación *Android ICRApp,* ilustrando una posible configuración del invernadero, procurando reflejar todas las posibles operaciones que pueden llevarse a cabo con este software.

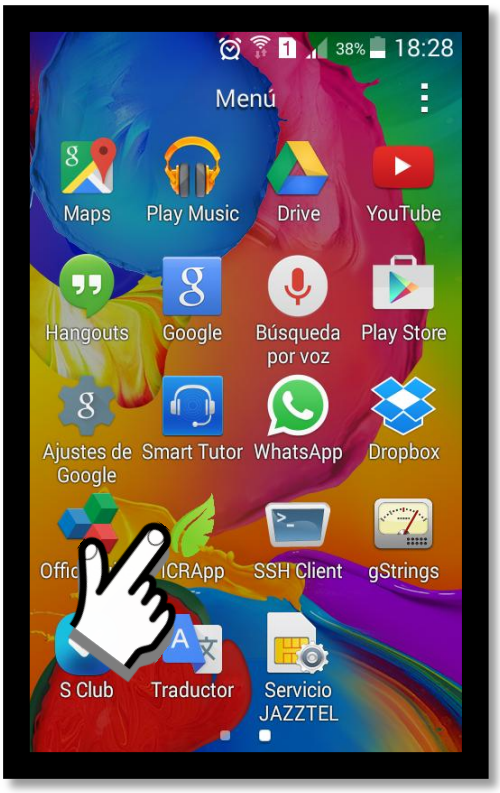



Figura 80: ICRApp: Lanzador de app


Figura 81: ICRApp: Login

1 Para comenzar a utilizar la *app* dirigirse al menú de aplicaciones del terminal y buscar el lanzador de aplicación denominado **ICRApp**

2 Introducir los datos de **usuario** y **password** y pulsar el botón **Login**. Pulsar el botón **Conectar** en caso de no estar conectado al servidor


Figura 82: ICRApp: Menu

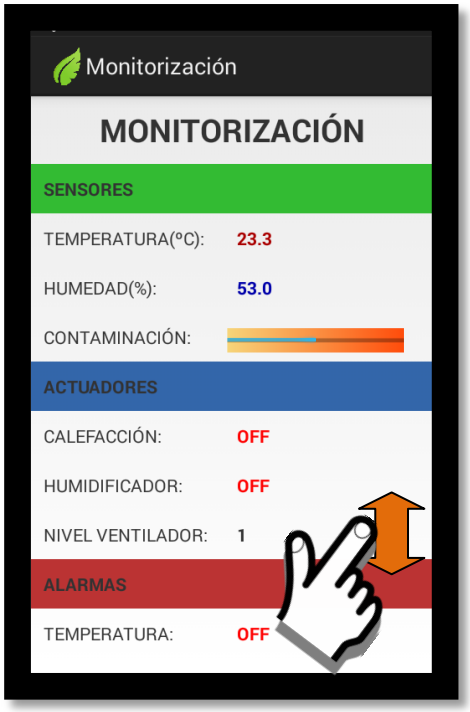



Figura 83: ICRApp: Monitorización

3 El menú permite el acceso a todos los gadgets de la app. Pulsar en **Monitorizar** para ver el actual estado del sistema

4 Esta pantalla muestra el estado de sensores, actuadores y alarmas del invernadero. Pulsar el botón *Return* del terminal para regresar al menú.


Figura 84: ICRApp: Rango Actuadores

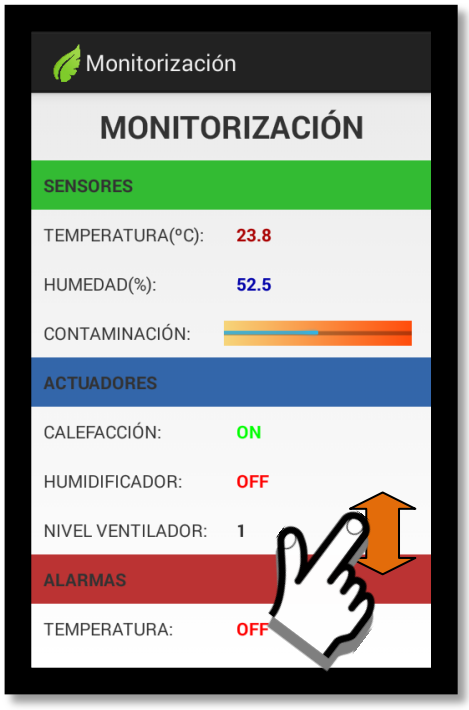



Figura 85: ICRApp: Monitorización (2)

5 Seleccionar **Rangos Actuadores** para modificar la temperatura. Insertar 24.00ºC y 25.00º como umbral de calefacción. Pulsar en los botones de disquete para guardar.

6 Regresar a la pantalla **Monitorización** para comprobar el estado activo de la calefacción.

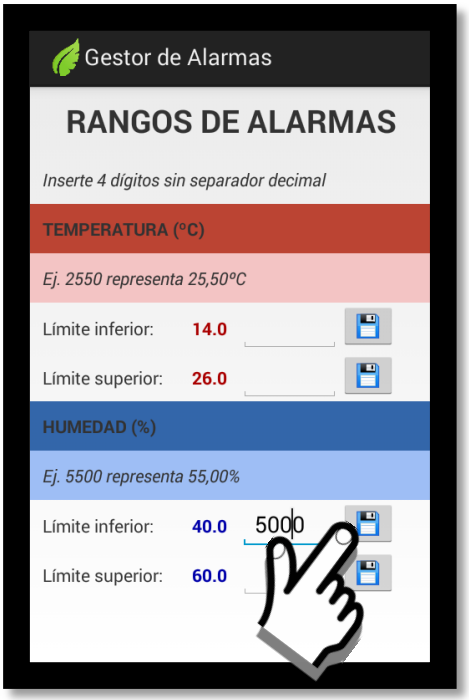



Figura 86: ICRApp: Rango de Alarmas


Figura 87: ICRApp: Control Ventilación

7 Se procede ahora a modificar el rango de alarma asociado a la humedad relativa. Para ello pulsar en el botón **Rango Alarmas** en el menú, y modificar el límite inferior a 50.00% de humedad relativa.

8 La siguiente modificación será fijar una nueva velocidad de ventilación. Para ello pulsar en el botón **Control Ventilación** en el menú. Fijar una **Velocidad 3** de ventilación.


Figura 88: ICRApp: Visor Gráficas

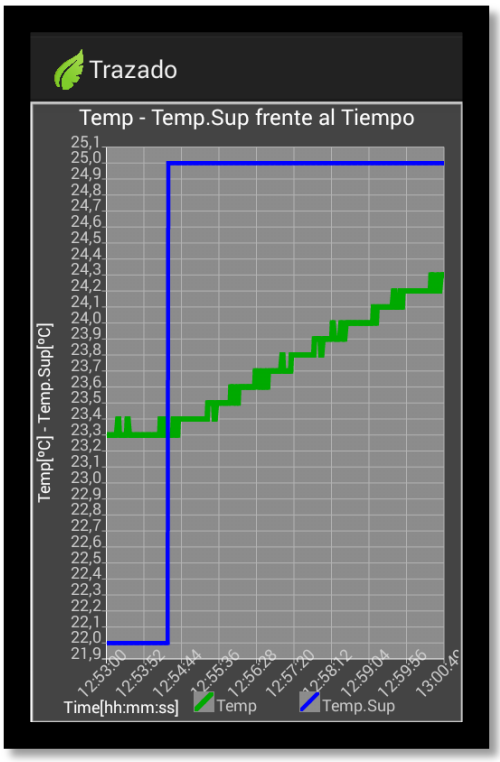



Figura 89: ICRApp: Trazado Temp-Temp.Sup

10 Acceder a **Visor de Gráficas** desde el menú para ver la evolución de variables. Insertar fecha/hora inicial y pulsar el botón de **calendario/ reloj**. Seleccionar la variable **Temperatura** y **Temperatura Nv Superior**. Pulsar el botón **Trazar**.

11 Se puede visualizar en pantalla el escalón provocado en la variable **Temp.Sup**, y como la **Temperatura** del recinto asciende hacia el valor superior establecido.

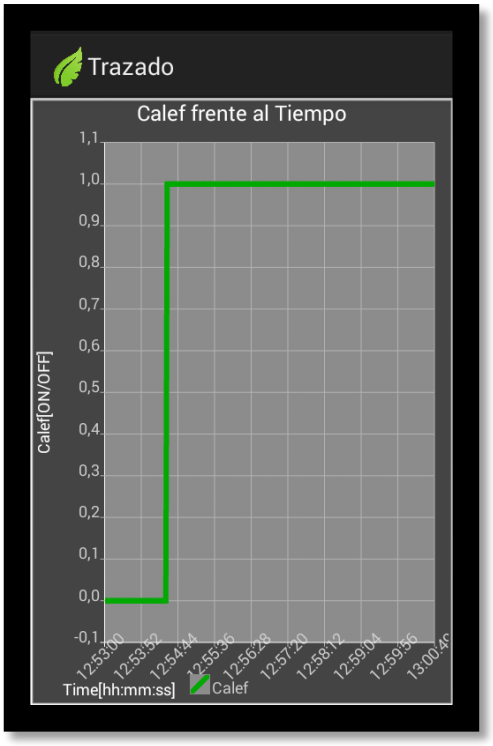



Figura 90: ICRApp: Trazado Calef.


Figura 91: ICRApp: Seta Emergencia

12 Pulsando el botón *Return* del terminal se regresa a la pantalla de selección de variables. Pueden seleccionarse diferentes parámetros y verse la evolución de estos.

13 Regresar al menú y pulsar el botón **Seta Emergencia**. Pulsar sobre el icono del pulsador de parada. El estado cambiará a **ACIVADA.**

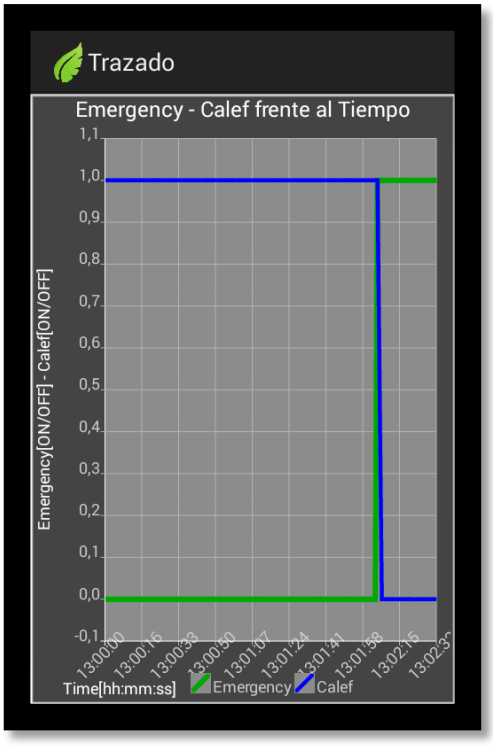



Figura 92: ICRApp: Trazado Calef.-Parada

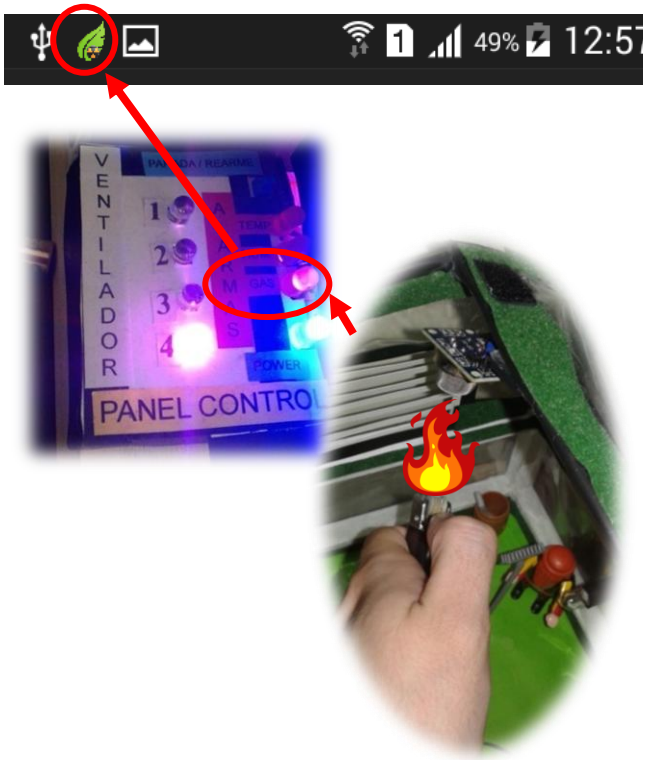



Figura 93: ICRApp: Notif Alarma Gas

14 Regresar al **Visor de Gráficas**, seleccionar un intervalo de tiempo que comprenda el evento de activación de la seta. Seleccionar las variables **Parada** y **Calefacción**. Ver el resultado.

15 El sistema ha registrado un exceso de contaminación ambiental, por lo que aparece una **notificación** de alarma en la barra superior del dispositivo.

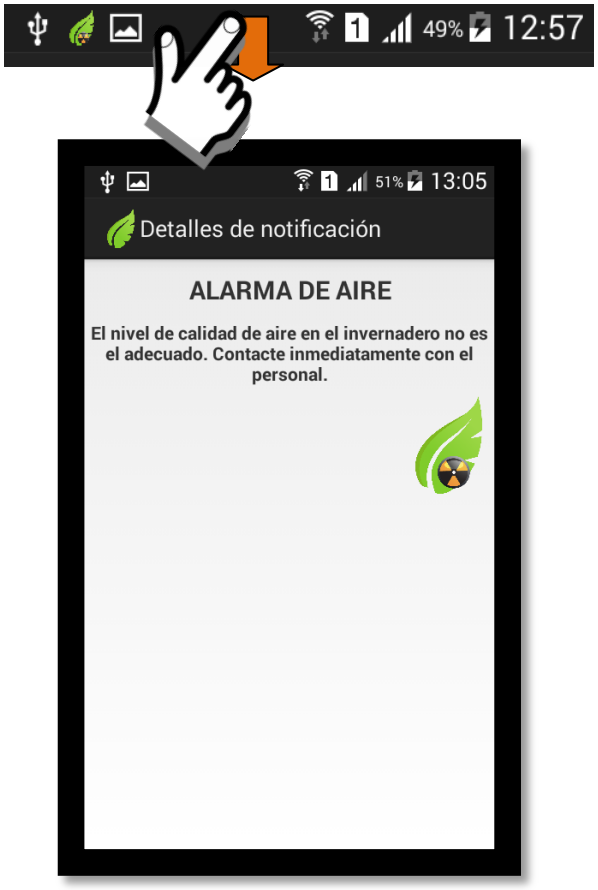



Figura 94: ICRApp: Detalle Notificación

16 Deslizar la barra de notificaciones y pulsar sobre la notificación de **ICRApp**. La información de la alarma activada se muestra en pantalla.

# ANEXO VII.

# MONTAJE DE ICRA

Este documento ilustra varias instantáneas que muestran el proceso de montaje de la maqueta *ICRA*, desde los primeros bloques de madera hasta su conexión final con el equipo servidor.

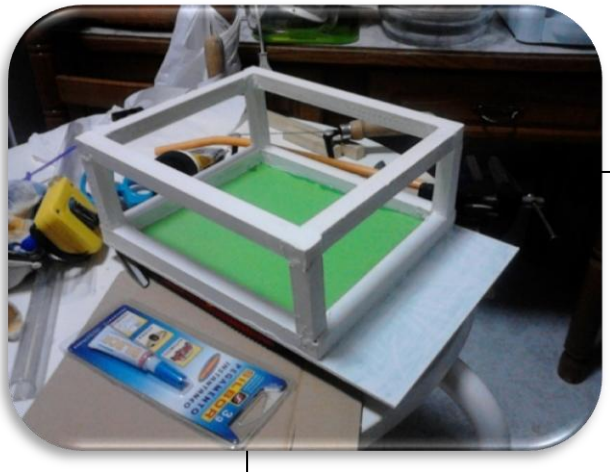


Figura 95: Estructura de madera

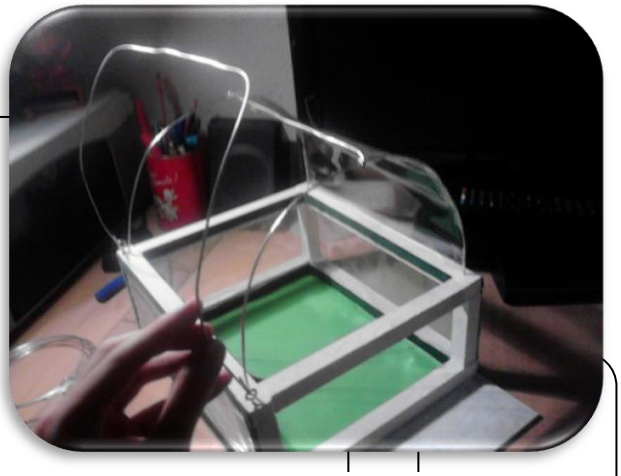


Figura 96: Compuerta abatible

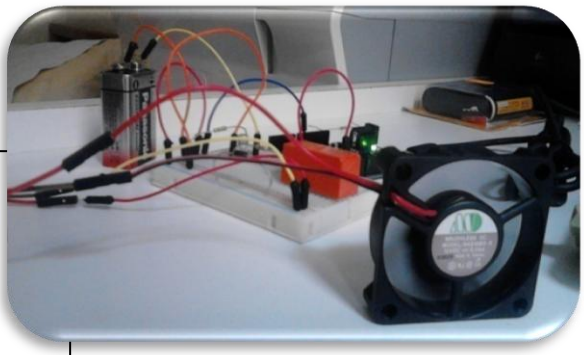


Figura 97: Prueba Ventilador

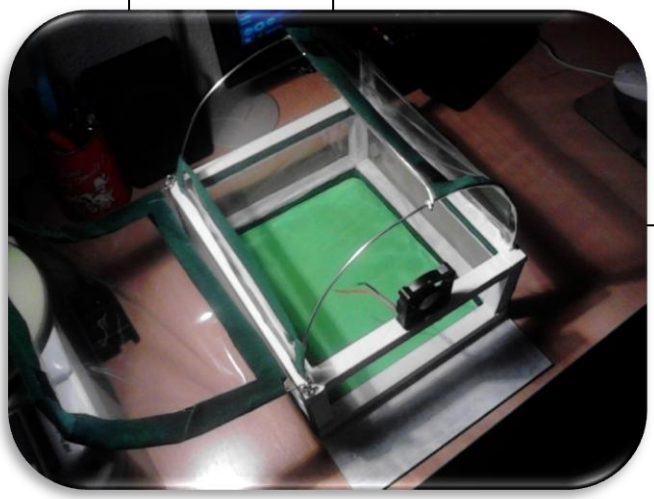


Figura 98: Embellecedores de paredes

Figura 99: Rejilla trasera

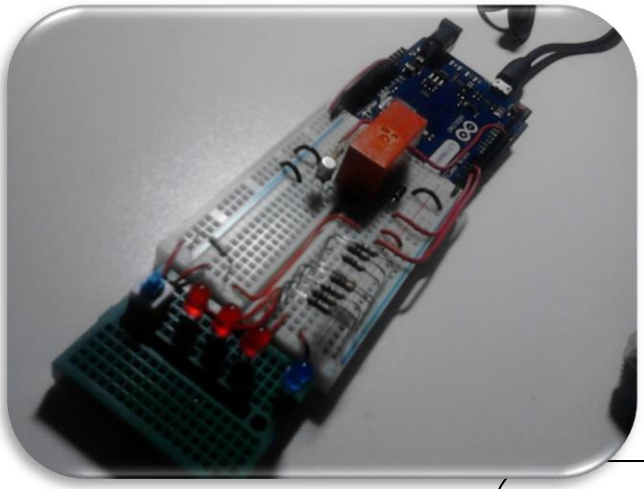


Figura 100: Conexiones eléctricas

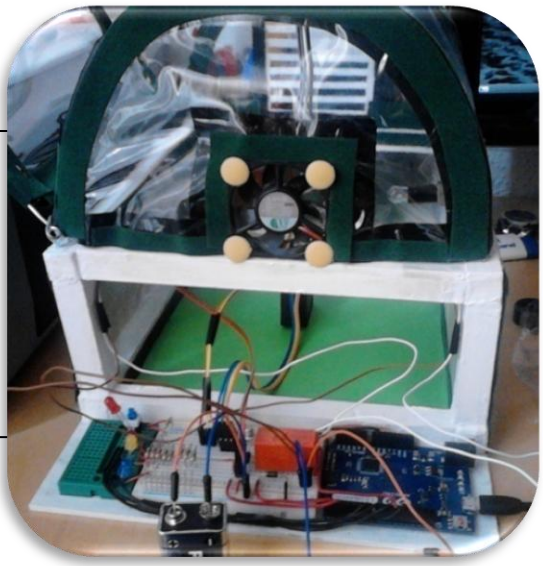


Figura 101: Instalación de placas

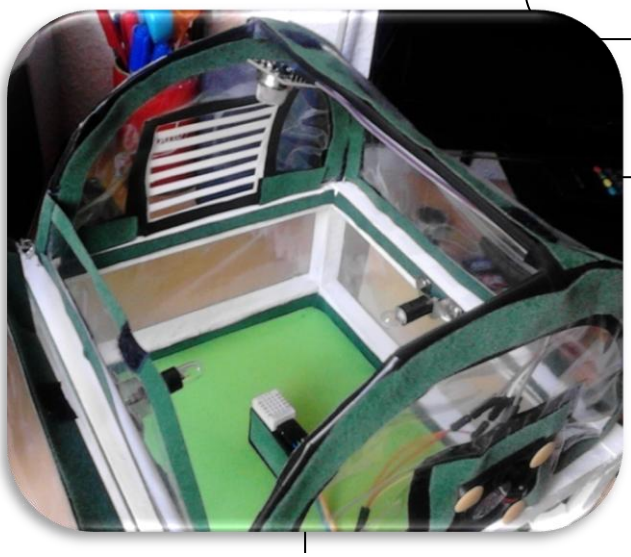


Figura 102: Sensores y Actuadores

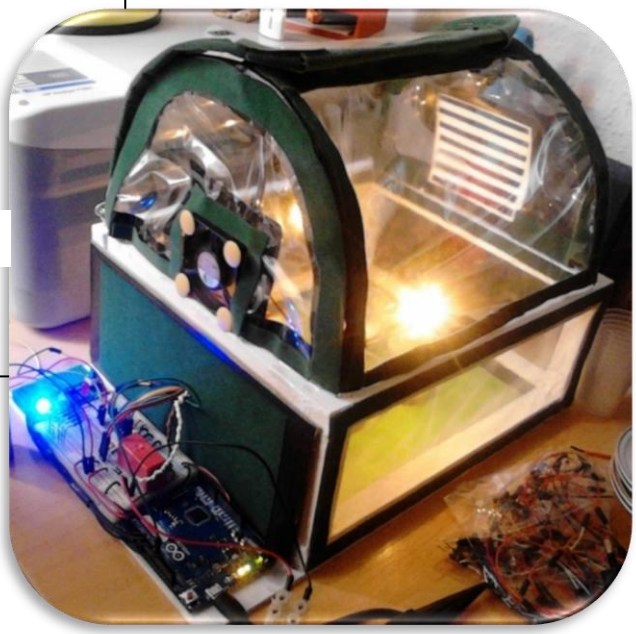


Figura 103: Primer arranque

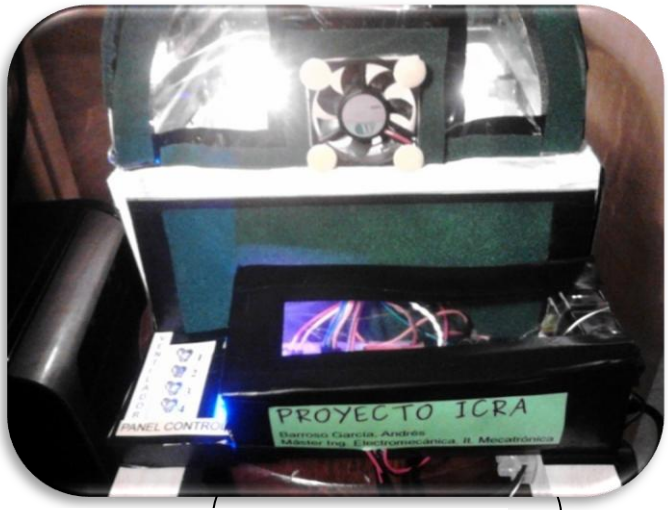



Figura 104: Cubierta de paneles

Figura 105: Primer experimento con vida

Figura 106: Etiquetado de cables

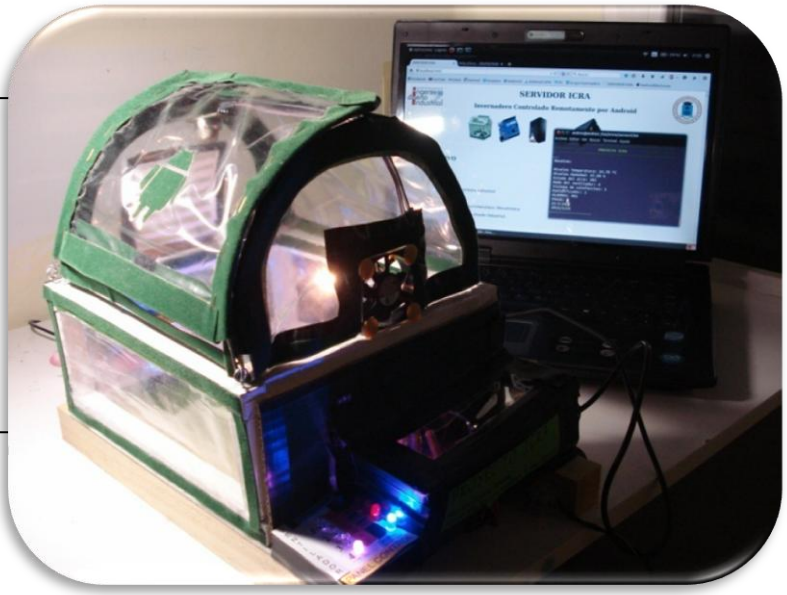



Figura 107: ICRA conectado al PC Servidor

# Bibliografía

1. **Aranda, Daniel Francisco Campos.** *Agroclimatología: Cuantitativa de Cultivos.* México : Trillas, 2005.

2. **Torrente Artero, Oscar.** *ARDUINO Curso práctico de formación.* 2013.

3. *A Low Cost Architecture for Remote Control and Monitoring of Greenhouse Fields.* **Daniel Moga, Dorin Petreus, Nicoleta Stroia.** Cluj-Napoca, Romania : Technical University of Cluj-Napoca, 2011.

4. INTA Tecnología adaptada a tu cultivo. [En línea] [Citado el: 12 de 11 de 2014.] http://www.inta.com.es/index.php/es/.

5. **Lee, Wei-Meng.** *Android 4 Desarrollo de aplicaciones.* Primera. s.l. : ANAYA, 2012.

6. International Data Corporation (IDC). [En línea] [Citado el: 30 de 12 de 2014.] http://www.idc.com/.

7. *Remote Control and Automation of Agriculture Devices Using Android Technology.* **Mrs. V.R. Waghmare, Ajit Bande, Komal Bhalerao, Tushar Deshmukh and Mukund Jadhav.** MMIT, Pune, India : s.n., 2013.

8. *A Gadget-Based Information Management System for Environmental Measurement and Control in Greenhouses .* **Takehiko Hoshi, Ryosuke Ohata, Katsuyoshi Watanabe and Ryuji Osuka.** Tokyo, Japan : Waseda University, 2011.

9. **Android Open Source Project.** Android Developers. [En línea] [Citado el: 15 de 06 de 2015.] developer.android.com/develop/.

10. **Hernando, Miguel.** *Programación C++.* s.l. : Servicio Publicaciones EUITI-UPM, 2005.

11. **Carretero, Jesús, y otros.** *Sistemas operativos: Una visión aplicada.* s.l. : McGraw-Hill, 2007.

12. Wikipedia. [En línea] [Citado el: 16 de 2 de 2015.] http://es.wikipedia.org/wiki/Servicio_web.

13. **Aransay, Alberto Los Santos.** *Metodologías para el Desarrollo de Servicios en la Web.* Universidad de Vigo. 2009.

14. **Gilfillan, Ian.** *La biblia de MySQL.* s.l. : Anaya, 2003.

15. **Oliver, Salvador Gómez.** *Curso Programación Android.* 2011.

16. **Girones, Jesús Tomás, y otros.** *El gran libro de Android avanzado.* Primera Edición. s.l. : Marcombo, 2014.

17. Introducción a JSON. [En línea] [Citado el: 15 de 04 de 2015.] http://json.org/json-es.html.

18. *Artemisa: An eco-driving assistant for Android Os.* **V.Corcoba Magaña, M. Muñoz-Organero.** Leganes, Spain : Universidad Carlos III, 2011.

19. *Green House Environment Monitor Technology Implementation Based on Android Mobile Platform.* **Wei Ai, Cifa Chen.** Yichang,Hubei Province,China : School of Computer and Information,Three - Gorges university, 2011.

20. *Apps in the Greenhouse.* **Tambascio, Sara.** s.l. : ProQuest Agricultural Science Collection, 2012.

21. *A Monitoring and Control System for Aquaculture via Wireless Network and Android Platform .* **Juan Huan, Xingqiao Liu, Hui Li, Hongyuan Wang, Xiaowei Zhu.** Changzhou, China : Changzhou University, 2014.

22. **Dawes, Beman y Abrahams, David.** Boost C++ Libraries. [En línea] [Citado el: 20 de 01 de 2015.] http://www.boost.org/.

23. **Arduino.** Arduino Proyect. [En línea] [Citado el: 1 de 12 de 2014.] http://arduino.cc.

24. **González Gómez, Juan.** Wiki-Robotics. [En línea] 2010. [Citado el: 12 de 12 de 2014.] http://www.iearobotics.com/wiki/index.php?title=Tutorial:_Puerto_serie_en_Linux_en_C%2B%2B.

25. **Göransson, Andreas y Cuartielles Ruiz, David.** *Android Open Accessory Programming with Arduino.* Indianapolis : Wiley, 2013.