

TRABAJO ESPECIAL DE GRADO

DESARROLLO DE UNA APLICACIÓN PARA MONITOREO BAJO EL SISTEMA OPERATIVO ANDROID

Presentado ante la ilustre
Universidad Central de Venezuela
por el Br. José M. Pires D.
para optar al título de
Ingeniero Electricista.

Caracas, 2015

TRABAJO ESPECIAL DE GRADO

DESARROLLO DE UNA APLICACIÓN PARA MONITOREO BAJO EL SISTEMA OPERATIVO ANDROID

TUTOR ACADÉMICO: José Alonso

Presentado ante la ilustre
Universidad Central de Venezuela
por el Br. José M. Pires D.
para optar al título de
Ingeniero Electricista.

Caracas, 2015

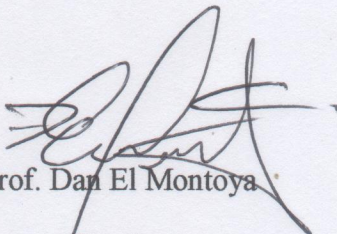
CONSTANCIA DE APROBACIÓN

Caracas, 04 de junio de 2015

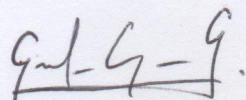
Los abajo firmantes, miembros del Jurado designado por el Consejo de Escuela de Ingeniería Eléctrica, para evaluar el Trabajo Especial de Grado presentado por el Bachiller José Manuel Pires Dias, titulado:

“DESARROLLO DE UNA APLICACIÓN PARA MONITOREO BAJO EL SISTEMA OPERATIVO ANDROID”

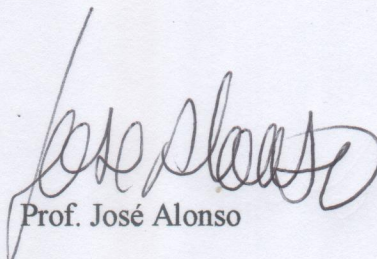
Consideran que el mismo cumple con los requisitos exigidos por el plan de estudios conducente al Título de Ingeniero Electricista en la mención de Electrónica y Control, y sin que ello signifique que se hacen solidarios con las ideas expuestas por el autor, lo declaran **APROBADO.**



Prof. Dan El Montoya
Jurado



Prof. Gerlis Caropresse
Jurado



Prof. José Alonso
Profesor guía

A mi madre, quien me ha dado todo.

RECONOCIMIENTOS Y AGRADECIMIENTOS

En primer lugar a mi madre, María Ynés, quien es responsable de todo lo que soy y de todo lo que he logrado.

A mi tío José Luis y mi tío Manuel. Por mi formación personal.

A mi hermano, Carlos David, que es mi imagen y reflejo.

A Rafael Lara, quien ha estado en las buenas y en las malas.

Vanessa Izaguirre, quien ha sido mi apoyo en incontables oportunidades.

A el profesor José Alonso y a José Carrasquel, quienes fueron mentores y guías, fundamentales en este proyecto.

A el profesor Ebert Brea, la profesora Mercedes Arocha, al profesor Luis Osorio y a todos los profesores que colaboraron en mi formación.

Alejandro Chitty, Omar Hokche, Alejandro Temprano, José Colmenares y Mauro Pinto, compañeros que colaboraron a lo largo de todo este camino.

A Andrea, Brenda y Claudia, tres personas que siempre están para lo que sea.

Gracias permanentes a todos ustedes porque son piezas fundamentales tanto en mi formación como en mi rumbo, gracias por todas las experiencias, por todo el aprendizaje y por todas las cosas, buenas y malas.

José M. Pires D.

DESARROLLO DE UNA APLICACIÓN PARA MONITOREO BAJO EL SISTEMA OPERATIVO ANDROID

Tutor Académico: José Alonso. Trabajo de grado. Caracas, Universidad Central de Venezuela. Facultad de Ingeniería. Escuela de Ingeniería Eléctrica. Mención: Electrónica, computación y control. Año 2015, 130h + anexos.

Palabras Claves: Android, Android Studio, Modbus, Modbus4j, Telemetría

Resumen.- En este trabajo se realiza una introducción a la programación de aplicaciones bajo el sistema operativo Android, adicionalmente, se mencionan algunos conceptos básicos de programación en Java. Luego de manejar los conceptos de programación en Android, se hará una breve introducción a conceptos de control supervisorio y el protocolo de comunicación Modbus. Conociendo todos los conceptos anteriores se realizará la elección de un ambiente de desarrollo y base de datos para finalmente desarrollar una aplicación en Android sobre monitoreo de variables importantes utilizando el protocolo de comunicación Modbus implementado con la librería *Modbus4j*. Esta aplicación permite monitorear variables dentro de un rango de valores determinado por el usuario y en un tiempo de muestreo personalizado, enviando alarmas vía mensaje de texto y correo electrónico ante una eventual medición que se encuentre fuera de rango durante el funcionamiento de la aplicación.

ÍNDICE GENERAL

CONSTANCIA DE APROBACIÓN	III
RECONOCIMIENTOS Y AGRADECIMIENTOS	V
RESUMEN	IX
ÍNDICE GENERAL	X
LISTA DE FIGURAS	XIV
LISTA DE TABLAS	XVI
LISTA DE ACRÓNIMOS	XVII
INTRODUCCIÓN	1
CAPÍTULO I	3
PLANTEAMIENTO DEL PROBLEMA	3
1.1. Objetivos	4
1.1.1. Objetivo general	4
1.1.2. Objetivos específicos	4
CAPÍTULO II	5
MARCO HISTÓRICO	5
CAPÍTULO III	9

MARCO TEÓRICO	9
3.1. Conceptos fundamentales de Android.	9
3.1.1. Android software development kit	10
3.1.2. Componentes de una aplicación	12
3.1.3. Archivo <i>AndroidManifest</i>	14
3.1.4. Interfaz de usuario	15
3.1.5. Librerías de soporte	17
3.1.6. Bases de datos	18
3.2. Programación en Java	19
3.2.1. Objetos	20
3.2.2. Clases	20
3.2.3. Paquetes	21
3.2.4. Excepciones	21
3.3. Comunicaciones	22
3.3.1. Protocolo TCP	24
3.3.2. Socket	24
3.3.3. Puerta de enlace	27
3.4. Conceptos de control supervisorio	27
3.4.1. Unidades remotas	27
3.4.2. Interfaz hombre-máquina	28
3.4.3. Modbus	28
 CAPÍTULO IV	 34
 MARCO METODOLÓGICO	 34
4.1. Metodología	34

4.2. Financiamiento y recursos requeridos	35
CAPÍTULO V	37
PROTOCOLO DE COMUNICACIÓN, AMBIENTE DE DESARROLLO Y DIAGRAMA DEL SISTEMA	37
5.1. Elección del protocolo de comunicación	37
5.2. Elección del ambiente de desarrollo	38
5.2.1. Estructura de archivos y proyectos	40
5.2.2. Rendimiento y depuración	41
5.3. Elección de base de datos	43
5.4. Equipos físicos y diagrama funcional del sistema	45
5.4.1. Dispositivo Android	45
5.4.2. Computador y programas	46
5.4.3. Diagrama funcional del sistema	46
CAPÍTULO VI	48
DESARROLLO DE LA APLICACIÓN	48
6.1. Diagrama interno de la aplicación y sus componentes	48
6.2. Creación de un nuevo proyecto en Android Studio	51
6.3. Archivo <i>AndroidManifest.xml</i>	54
6.4. Librerías externas	56
6.4.1. <i>Modbus4j.jar</i>	56
6.4.2. <i>mail.jar</i>	57
6.5. Paquetes internos destacados	59
6.5.1. <i>SharedPreferences</i>	60

6.5.2. <i>Timer</i> y <i>TimerTask</i>	60
6.5.3. <i>Handler</i>	60
6.5.4. <i>AsyncTask</i>	61
6.6. Desarrollo de las actividades y clases.	61
6.6.1. <i>MainActivity.java</i>	61
6.6.2. <i>SettingsActivity.java</i>	88
6.6.3. <i>AlarmConfig.java</i>	94
6.6.4. Creación y manejo de la base de datos	100
CAPÍTULO VII	110
RESULTADOS Y PRUEBAS	110
7.1. Pruebas realizadas	111
7.1.1. Pruebas sobre las funciones de Modbus	112
7.1.2. Prueba de alarmas	118
7.1.3. Prueba en un periodo largo de tiempo	119
7.1.4. Errores de entrada	121
CONCLUSIONES	123
RECOMENDACIONES	125
REFERENCIAS	127
ANEXOS	131

LISTA DE FIGURAS

Figura 3.1. Ilustración jerárquica de la disposición de interfaz (Android Developers, s.f.-e).	16
Figura 3.2. Flujo de eventos en una sesión de <i>socket</i> (IBM Knowledge, s.f.).	25
Figura 3.3. Ubicación del <i>socket</i> en el modelo OSI(IBM Knowledge, s.f.).	26
Figura 5.1. Vista de proyecto Android. (Android Developers, s.f.-a) . .	40
Figura 5.2. Monitor de memoria y CPU en Android Studio (Android Developers, s.f.-a)	42
Figura 5.3. Modo diseño de la interfaz en Android Studio (Android Developers, s.f.-a)	43
Figura 5.4. Diagrama funcional del sistema.	47
Figura 6.1. Diagrama interno de la aplicación.	49
Figura 6.2. Creación de nuevo proyecto en Android Studio.	52
Figura 6.3. Selección del <i>form factor</i> y el API en Android Studio. . . .	53
Figura 6.4. Selección del nombre de la actividad en Android Studio. .	53
Figura 6.5. Vista por defecto y actividad principal en Android Studio.	54
Figura 6.6. Interfaz gráfica de la actividad principal.	62
Figura 6.7. Interfaz gráfica de la actividad de configuración.	88
Figura 6.8. Interfaz gráfica de la actividad de configuración de alarmas.	95
Figura 7.1. Configuración de conexión de <i>Modbus Slave</i>	112
Figura 7.2. Configuración de <i>Modbus Slave</i>	113

Figura 7.3. Funcionamiento de <i>Modbus Slave</i> en la función 01	114
Figura 7.4. Introducción de parámetros en el dispositivo móvil.	114
Figura 7.5. Aplicación funcionando en prueba de la función 01.	115
Figura 7.6. Configuración de <i>Modbus Slave</i>	116
Figura 7.7. Funcionamiento de <i>Modbus Slave</i> en la función 03	117
Figura 7.8. Aplicación funcionando en prueba de la función 03.	117
Figura 7.9. Configuración de las alarmas	118
Figura 7.10. Medición antes de disparar alarma.	119
Figura 7.11. Pantalla luego de envío de la alarma.	120
Figura 7.12. Esclavo de Modbus simulado por más de ocho horas.	120
Figura 7.13. Resultado de las mediciones luego de ocho horas.	121

LISTA DE TABLAS

Tabla 3.1. Diferencias entre Modbus RTU y Modbus ASCII (RTA Automation, s.f.)	31
Tabla 6.1. Ejemplo de la estructura de la tabla para valores medidos. .	101
Tabla 6.2. Ejemplo de la estructura de la tabla para valores de alarma.	102

LISTA DE ACRÓNIMOS

IDE: Integrated Development Environment.

ADT: Android Development Tools.

SCADA: Supervisory Control and Data Acquisition .

SMS: Short Messaging Service.

RTU: Remote Terminal Unit.

IEEE: Institute of Electrical and Electronic engineers.

PLC: Programmable Logic Controller.

HMI: Human-Machine Interface.

SDK: System Development Kit.

APK: Android Package.

API: Application Program Interface.

UI: User Interface.

DBMS: DataBase Management System.

OSI: Open System Interconnection.

TCP: Transmission Control Protocol.

HAXM: Hardware Accelerated Execution Manager.

PDU: Protocol Data Unit.

CRUD: Create, Read, Update, Delete.

INTRODUCCIÓN

Este proyecto se ubica en el marco del desarrollo de aplicaciones para Android, además de ubicarse en el marco anterior, se ubica en el marco de la puesta en marcha de un protocolo de comunicación de calidad industrial como lo es Modbus. Para lograr implementar esta aplicación es necesario conocer los conceptos básicos de la programación en Android y conocer cómo funciona el protocolo Modbus, además, se requiere relacionar adecuadamente los diferentes elementos del sistema a implementar para poder proceder de forma correcta con todos los pasos que conlleva el desarrollo del proyecto.

En el primer capítulo de este trabajo se presenta una reseña histórica de los inicios de Android en el mercado de la telefonía móvil y su estado actual en el mercado de la tecnología. Además de los avances de Android desde su origen, se presenta una introducción a los sistemas SCADA y cómo forman parte en el origen del monitoreo de variables, formando un sistema de comunicación estable con un protocolo estándar y el uso de las unidades remotas como dispositivos para adquisición de datos. Se presentan los orígenes del protocolo Modbus y su presencia actual en el desarrollo de aplicaciones en dispositivos móviles para control supervisorio.

El marco teórico incluye diferentes conceptos fundamentales para el desarrollo de aplicaciones bajo el sistema operativo Android y algunos conceptos necesarios del lenguaje de programación Java, utilizado ampliamente en Android. Otros conceptos incluidos en el marco teórico son los relacionados a las comunicaciones, abarcando el modelo OSI, el protocolo TCP, la presencia de *Sockets* en las comunicaciones y el concepto de puerta de enlace como dispositivo. Adicionalmente a

los conceptos previos, se introducen conceptos básicos de control supervisorio y la explicación de cómo funciona el protocolo Modbus.

El capítulo referente al marco metodológico abarca el planteamiento general del problema a resolver en este proyecto, los objetivos generales que llevan a la solución del problema y los objetivos específicos necesarios para cumplir con el objetivo general, adicionalmente, se explica la metodología con la que se va a proceder para cumplir con los objetivos generales.

En el capítulo 5 se justifica la elección del protocolo de comunicación, la selección del ambiente de desarrollo y del paquete de bases de datos a implementar en el proyecto. Conociendo los elementos anteriores, se listan los equipos físicos a utilizar y se presenta un diagrama general del sistema con la finalidad de dar a conocer cómo se relacionan los elementos físicos que forman parte del proyecto.

En el capítulo 6 se explica todo el desarrollo de la aplicación, incluyendo un diagrama interno de su forma, puesta en marcha del ambiente de desarrollo seleccionado, explicación de las librerías y paquetes más relevantes y posteriormente la codificación de los elementos más importantes en la aplicación. Adicionalmente en el capítulo 7 se presentan los resultados del desarrollo de la aplicación y algunas pruebas realizadas para verificar el correcto funcionamiento de la aplicación.

Finalmente en el capítulo 8 se realizan las conclusiones del proyecto y posteriormente algunas recomendaciones a futuro para la vida del proyecto y mejora del trabajo realizado.

CAPÍTULO I

PLANTEAMIENTO DEL PROBLEMA

Existen diferentes problemas generados por la falta de monitoreo de variables específicas, que pueden requerir atención particular frente a situaciones que no se manejan diariamente, como la avería de equipos, la descomposición de un material refrigerado o inclusive la pérdida de un punto ideal en una variable para realizar un trabajo.

A partir de dicho problema, y trabajando en función de una solución viable e innovadora en el área de ingeniería, se plantea realizar una aplicación bajo el sistema operativo Android que permita realizar el monitoreo en tiempo real de variables críticas, de forma desasistida y que, ante cambios significativos de las variables observadas, se realicen notificaciones vía SMS y/o correo electrónico de forma inmediata al personal encargado del mantenimiento, con la finalidad de evitar el problema nombrado.

Conociendo los cambios en tiempo real de una variable, y manejando una base de datos como elemento necesario para guardar y manipular la información obtenida, se puede plantear además, la posibilidad de presentar dichos cambios como fuente de futuros estudios estadísticos que finalmente se traducen en el conocimiento cuantitativo del evento presentado, permitiendo entonces plantear mejoras que sin el previo estudio nombrado, difícilmente pudieran ser realizadas.

1.1. Objetivos

1.1.1. Objetivo general

Desarrollar una aplicación bajo el ambiente del sistema operativo Android, que permita monitorear variables críticas de un sistema, con la finalidad de identificar y notificar variaciones significativas de forma oportuna, ya sea vía SMS o e-mail.

1.1.2. Objetivos específicos

1. Definir mediante un diagrama funcional la topología del sistema a implementar, relacionando las diferentes partes que integran el sistema.
2. Recopilar información para el desarrollo de aplicaciones bajo el sistema operativo Android y elegir un ambiente de desarrollo adecuado para el proyecto.
3. Implementar el software descrito, partiendo del diagrama funcional y un ambiente de desarrollo acorde a las necesidades y el alcance del proyecto.
4. Realizar pruebas exhaustivas con dispositivos en tiempo real, con la finalidad de probar y depurar el software desarrollado.
5. Elaborar un informe final, que incluya características, recomendaciones y planteamientos a futuro en vista de mejoras para el software planteado.

CAPÍTULO II

MARCO HISTÓRICO

La compañía **Android, Inc.** fundada en el año 2003 por andy Rubin en Palo Alto, California, tenía la idea de explotar el potencial existente en el desarrollo de dispositivos móviles inteligentes, atentos a la ubicación y preferencias del usuario (Amadeo, 2014). Fue adquirida en 2005 por **Google, Inc.** (Elgin, 2005) tan solo explicando, según uno de sus voceros: “Adquirimos Android por el talento de los ingenieros y la gran tecnología, estamos emocionados de tenerlos aquí.”

En 2007, la “**Open Handset Alliance**” un consorcio de compañías de tecnología, incluyendo Google, anuncia en alianza el gol común de llevar innovación a dispositivos móviles y complacer tanto al desarrollador como al usuario final con una experiencia que tiene un nuevo nivel de apertura que permitiría el desarrollo y trabajo colaborativo, acelerando el ritmo en el cual se realizan aplicaciones y servicios móviles para los consumidores. Android es el proyecto que llevaría esta alianza y sus metas a la realidad (Open handset alliance, 2007).

Android es actualmente el sistema operativo móvil más utilizado en *smartphones*, para septiembre de 2013 había más de un billón de dispositivos activados en total, y más de un millón de aplicaciones publicadas para descargar (Pichai, 2013).

Para finales del 2014, el *IDE* oficial para Android era Eclipse, apoyado por la extensión *ADT*. Adicionalmente, la plataforma IntelliJ IDEA es otro IDE que

soporta el desarrollo de aplicaciones de Android junto con NetBeans IDE, todos estos ambientes permiten incluir el kit de desarrollo de Android entre sus fuentes de trabajo y estructurar el proyecto de una aplicación de forma ordenada. Para el año 2015 Android desarrolla su propio IDE llamado Android Studio, con apoyo de IntelliJ que posee algunas innovaciones frente a Eclipse, como es la generación de múltiples instancias de una aplicación en desarrollo o la importación de librerías y dependencias para el desarrollo de aplicaciones. (Wikipedia, 2014a)

Además de Android como sistema operativo, la información histórica respecto a monitoreo de variables se remonta a las fechas de 1965 cuando se desarrollaron los primeros sistemas SCADA, donde ya era posible utilizar computadores para implementar estaciones maestras de control. Los primeros equipos fueron desarrollados por WestingHouse y General Electric, las funciones implementadas para esa fecha incluían escaneo de datos, estado de variables, y alarmas debido a cambios drásticos en valores. La mayoría de los sistemas SCADA funcionaban como un maestro, enviando peticiones de datos a unidades remotas mediante escaneo continuo de todas las unidades con información (Russell, 2009).

Las RTU tomaron esta nomenclatura durante la implementación de los sistemas SCADA, ya que los vendedores de sistemas SCADA eran los principales desarrolladores de unidades terminales remotas, estas unidades tenían la necesidad de trabajar inclusive ante fallas eléctricas generales, por lo tanto estaban siempre conectadas a baterías, además era importante que fueran de respuesta rápida a operaciones de control en caso de algún evento inesperado, por lo tanto el protocolo de comunicación debía ser tanto eficiente como seguro. Durante los años 70 la mayoría de las RTU estaban sujetas al protocolo de comunicación implementado por su fabricante, es decir, bajo un software propietario. Gracias a la necesidad de implementar diferentes RTU's en un solo sistema SCADA, se realizó el esfuerzo de estandarizar protocolos bajo la IEEE. La estructura básica de una RTU consistía

entonces en de una interfaz de comunicación, un controlador lógico central, un sistema de entradas/salidas con entradas analógicas, control de salidas digitales y algunas veces control de salidas analógicas. Un sistema SCADA de gran tamaño podía contener varias centenas de RTU's(Russell, 2009).

En el año 1979 el fabricante de PLC's Modicon publica Modbus, un protocolo de comunicación de red basado en una arquitectura maestro/esclavo, es un estándar abierto que describe la estructura que debe tener el mensaje para poder comunicar diferentes unidades y equipos en la industria (Siemens, s.f.). Se convirtió en el protocolo estándar más popular de la historia, con la mayor cantidad de implementaciones debido a su confianza y robustez, actualmente existen diferentes implementaciones de Modbus que parten del estándar oficial, llevándolo entonces a un nivel más alto de popularidad y establecimiento frente a otros protocolos de comunicación. Los formatos utilizados de Modbus actualmente son los siguientes: Modbus RTU, Modbus ASCII y Modbus TCP, utilizados en equipos modernos con el mismo protocolo implementado por Modicon (PLCDev, s.f.).

Los primeros sistemas SCADA donde utilizaron paneles de control cableados, con botones para seleccionar puntos y realizar operaciones de control, y a finales de los años 60 los tubos de rayos catódicos fueron implementados en la HMI (Russell, 2009). Hoy en día es común encontrar dispositivos de HMI sumamente modernos, con entornos gráficos que permiten interacción directa entre variables y el usuario-hombre.

Actualmente existen algunas soluciones dentro de Android basadas en Modbus y sistemas SCADA, como lo es el desarrollo de Modbus-Droid o TeslaSCADA en Rusia, que plantean soluciones de Modbus dentro de dispositivos móviles en la actualidad y desarrollan aplicaciones funcionales.

En el marco de este proyecto, se presenta la posibilidad de implementar una HMI moderna, basada en el desarrollo de una aplicación bajo el sistema operativo Android y la presencia de Modbus como estándar de comunicación efectivo con las variables que se desea monitorear, representando el desarrollo de un sistema SCADA a pequeña escala, que permita el monitoreo de variables, del mismo modo que ha sido implementado tanto en sus inicios como en la actualidad.

CAPÍTULO III

MARCO TEÓRICO

En este capítulo se definirán todos los conceptos y herramientas necesarias para desarrollar una aplicación bajo Android, empezando por el ambiente de desarrollo, conceptos de Java, librerías, y el uso de bases de datos. Adicionalmente se abarcarán los conceptos que se deben manejar para implementar una aplicación de monitoreo, incluyendo una breve introducción a los sistemas SCADA y protocolos de comunicación.

3.1. Conceptos fundamentales de Android.

Android es un sistema operativo para dispositivos móviles basado en un núcleo Linux.(Android Developers, s.f.-b) El código fuente de este sistema operativo es liberado por Google bajo licencias de código abierto, lo que permite el desarrollo de software con una mezcla entre código abierto y software propietario. Es un sistema operativo de bajo costo que permite la puesta en marcha de dispositivos de alta tecnología que requieren un software personalizable para funcionar, la naturaleza abierta de este sistema ha llevado a desarrolladores y entusiastas a utilizar el código fuente como base para cualquier tipo de proyectos, como aplicaciones y sistemas para otro tipo de dispositivos, como el Raspberry Pi(Calin, 2014).

Las aplicaciones (“*apps*”) de Android están escritas en el lenguaje de programación Java (Android Developers, s.f.-b). Esto implica que se deben conocer algunos conceptos de Java y además se deben conocer los componentes que forman

una aplicación dentro de lo que es formalmente un ambiente de desarrollo para aplicaciones en Android. Adicionalmente a los conceptos anteriores, se definirá lo que es un SDK y cómo existen las aplicaciones para Android dentro del sistema operativo.

3.1.1. Android software development kit

El Android SDK, devkit o “*Software development kit*” es un paquete de herramientas proporcionado por los desarrolladores de Android que contiene todas las herramientas y datos necesarios para compilar el código de una aplicación. Las herramientas que proporciona el SDK de Android son independientes de una plataforma y permiten al usuario la libertad de elegir una plataforma o ambiente de desarrollo de su preferencia (Android Developers, s.f.-d). Las herramientas proporcionadas por el SDK de Android son instaladas con el paquete inicial del SDK, estas herramientas son las mínimas necesarias si se desea desarrollar una aplicación para Android. Algunas de las herramientas más importantes son las siguientes:

- *Virtual Device Tools*: Contiene herramientas que permiten realizar instancias virtuales de dispositivos Android para probar aplicaciones, simular bahías de memoria externa y hasta simular dispositivos particulares.
- *Development Tools*: Kit de herramientas que permite manejar, en general, el SDK. Algunas de las herramientas permiten optimizar el código de una aplicación y mantenerlo limpio, también contiene herramientas para manejar la versión del SDK y poder cambiarla e inclusive contiene los archivos de SQLite para manejar bases de datos directamente con el ambiente de desarrollo utilizado.
- *Debugging Tools*: Las herramientas para *debugging* son las que permiten

la comunicación e interacción entre el usuario y una instancia de prueba de una aplicación o simulación. Entre las herramienta que posee este kit se encuentran logs sobre todas las acciones ejecutadas dentro de la prueba de una aplicación, tanto de forma gráfica como en texto, herramientas de análisis de la aplicación dentro del contexto del sistema operativo y así como herramientas para visualizar todos los archivos generados.

- *Build Tools*: Son las herramientas que realizan la creación del archivo final de la aplicación. Se encarga de optimizar código luego de que una aplicación es funcional, reducir el espacio ocupado por todos los archivos que forman la aplicación y de encriptar código en caso de que sea necesario. El archivo final de una aplicación de Android tiene como extensión APK (*Android package*).
- *Image Tools*: Estas herramientas son las necesarias para el manejo de imágenes en el desarrollo de aplicaciones, maneja formatos para compresión, captura de imágenes y creación de gráficos para el resto de las herramientas del SDK.

A partir de las herramientas anteriores, existen diferentes plataformas para manejar el SDK, como son los IDE o ambientes de desarrollo integrado. Los ambientes de desarrollo más completos contienen compiladores precargados, con herramientas dispuestas a su uso, en el caso del desarrollo de una aplicación para Android, el compilador que se debe cargar al ambiente de desarrollo es el SDK de Android.

Es parte de este trabajo definir el uso de un ambiente de desarrollo para realizar el proyecto, basado en la información de *Android Developers*, en los próximos capítulos se elegirá un ambiente de desarrollo adecuado para integrar el SDK y disponer de las herramientas para desarrollar una aplicación en Android.

Para el momento en el que una aplicación de Android está instalada en un dispositivo, ésta vive en su propia caja de seguridad, con las siguientes características:

- Cada aplicación es un usuario personal diferente en un sistema operativo multi-usuarios.
- El sistema operativo asigna a cada aplicación un *user ID* único
- Cada proceso tiene su propia máquina virtual, evitando mezclar el código de una aplicación con otra.
- Cuando se ejecuta la aplicación ella es su propio proceso, y es finalizado cuando se cierra la misma, evitando utilizar memoria innecesaria.

Sin embargo, existen formas para que una aplicación comparta datos con otra aplicación, o para acceder a servicios particulares de otra aplicación. Una aplicación puede pedir permiso para utilizar los contactos del usuario, enviar SMS, enviar e-mails, utilizar la cámara, *bluetooth*, etc. Todos los permisos deben ser garantizados por el usuario al momento de la instalación.

Conociendo los términos en los que una aplicación subsiste en el sistema operativo, se pueden entonces cubrir los componentes básicos que definen una aplicación en Android.

3.1.2. Componentes de una aplicación

Existen cuatro tipos diferentes de componentes. Cada uno tiene un propósito distinto y una forma de existir dentro de la aplicación. Los cuatro tipos de componentes son:

- *Activities*: Una actividad representa una sola pantalla con una interfaz de usuario. Por ejemplo, una aplicación de e-mail puede tener una actividad para mostrar una lista de correos, otra actividad para redactar un correo, y otra actividad para leer correos. A pesar de que las actividades trabajan juntas para crear una experiencia de usuario en su aplicación de e-mail, cada actividad es independiente de las otras.
- *Services*: Un servicio es un componente que corre en un segundo plano, realizando operaciones extensas o trabajos de procesos remotos. Un servicio no tiene una interfaz de usuario. Por ejemplo, un servicio puede reproducir música de fondo mientras el usuario está en una aplicación diferente.
- *Content providers*: Un proveedor de contenido maneja toda la data de una aplicación que debe ser compartida. Se puede guardar toda la data en la web, archivos SQLite, o cualquier lugar de almacenamiento persistente al que pueda acceder la aplicación. A través del *content provider* otras aplicaciones pueden ver o hasta modificar los datos, si el proveedor lo permite.
- *Broadcast receivers*: Es un componente que responde a anuncios del sistema. Por ejemplo, un mensaje global de que la pantalla se apaga, la batería está baja, o que se realizó una captura de pantalla. Algunas aplicaciones también pueden realizar mensajes globales, para indicar que se descargó algún dato o que la aplicación está lista para utilizarse.

Un aspecto particular del sistema operativo Android, es que cualquier aplicación puede iniciar componentes de otras aplicaciones. Por ejemplo, si el usuario quiere capturar una foto con la cámara del dispositivo, existe otra aplicación que utiliza la cámara y el usuario puede iniciarla desde otra aplicación que pueda iniciar la actividad de tomar fotos, sin necesidad de crear una actividad para iniciar la cámara en cada aplicación que requiera uso de ella.

El uso de componentes que pertenecen a otras aplicaciones, sin embargo, debe ser declarado como un permiso por el usuario al momento de instalar la aplicación, de tal forma que el sistema tenga permiso de comunicar los componentes deseados para que una aplicación particular funcione.

3.1.3. Archivo *AndroidManifest*

Para que el sistema operativo pueda inicializar un componente de otra aplicación el sistema debe saber que el componente existe, esto lo hace mediante el archivo *AndroidManifest.xml* de la aplicación. Se deben declarar todos los componentes en este archivo, que debe estar en el directorio raíz del proyecto de la aplicación.

El archivo *AndroidManifest.xml* hace algunas cosas adicionales además de declarar los componentes de una aplicación, como por ejemplo:

- Identificar los permisos que requiere una aplicación, como el acceso a internet.
- Declarar el nivel mínimo del API que requiere una aplicación.
- Declarar el hardware y software necesario o requerido por la aplicación, como la cámara, servicios o servicios de *bluetooth*.
- Librerías a las que la aplicación debe estar conectada o que deben ser importadas.

El concepto de *API level* se refiere simplemente a un valor entero que identifica el marco de la versión API que ofrece la plataforma de Android. Cada versión sucesiva de la plataforma de Android incluye actualizaciones al marco de la versión

API, y la idea de estas actualizaciones es que el nuevo API se mantenga compatible con versiones anteriores(Android Developers, s.f.-f).

Finalmente, el sistema operativo responde a una versión específica del API, y los desarrolladores se encargan de que las nuevas funciones en Android se integren de forma aditiva al sistema, evitando eliminar componentes importantes y manteniendo cada versión de Android funcional para su nivel de API.

El diseño de una aplicación debe ir dirigido a un nivel específico de API, asegurando así el funcionamiento de la aplicación en la versión elegida de Android y las posteriores. Esto permite evitar problemas de compatibilidad entre la aplicación y el sistema operativo en su versión mínima elegida, según el nivel de API.

3.1.4. Interfaz de usuario

Todos los elementos de la interfaz de usuario (“UI”)en una aplicación de Android son creados utilizando objetos *View* y *ViewGroup*. Un objeto *View* dibuja algo en la pantalla con lo que el usuario puede interactuar, mientras que un objeto *ViewGroup* está encargado de relacionar los objetos *View* y otros *ViewGroup* para definir el *Layout* de la interfaz. En la figura 3.1 se puede observar un ejemplo de la jerarquía entre los elementos *Viewgroup* y *View*.

Layout

La disposición o *layout* son los elementos no visuales destinados a controlar la distribución, posición y dimensiones de los controls que se insertan en su interior. Estos componentes extienden la clase base *ViewGroup*, como muchos otros componentes contenedores, es decir, capaces de contener a otros controles. Existen varios tipos de *Layout* entre ellos están los siguientes:

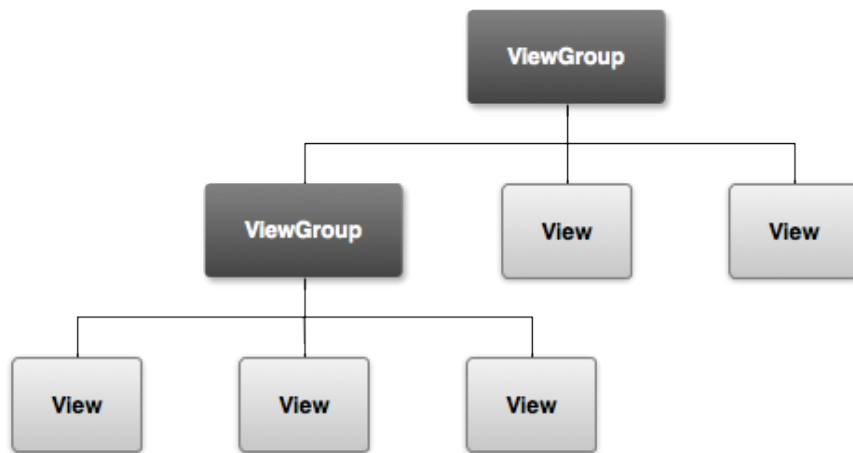


Figura 3.1. Ilustración jerárquica de la disposición de interfaz (Android Developers, s.f.-e).

- *Frame Layout*: Este layout coloca todos sus elementos alineados a su esquina superior izquierda, de forma que el control quede ubicado de la forma más simple posible.
- *Linear Layout*: Es un *ViewGroup* que alinea todos sus inferiores en una sola dirección, vertical u horizontal.
- *Relative Layout*: Es un *ViewGroup* que muestra sus inferiores en posiciones relativas. La posición de cada *View* puede ser especificada tan relativa como se quiera respecto a otros elementos.

Existen otros tipos de *Layouts* con otros beneficios y características, como lo son el *List View* o el *Grid View*, también útiles para definir elementos fijos o relativos en disposiciones particulares.(Android Developers, s.f.-c)

Input Controls

Los *Input Controls* o controles de entrada, son los componentes interactivos en la interfaz de usuario. Android provee una amplia variedad de controles que

se pueden utilizar en la *UI*, como botones, campos de texto, barras de búsqueda, *checkboxes*, botones de acercamiento y otros. Según *Android developers* los *Input Control* más comunes son:

- *Button*: Es un pulsador que puede ser presionado o clickeado por el usuario para realizar una acción.
- *Text field*: Un campo de texto editable.
- *Checkbox*: Un switch *on/off* que puede ser fijado por el usuario. Se puede utilizar en un grupo con opciones no mutuamente excluyentes.
- *Radio button*: Similar a las *checkbox* a excepción de que solo se puede seleccionar una opción en un grupo.
- *Toggle button*: Un botón *on/off* con un indicador de luz.
- *Spinner*: Una lista *drop-down* que permite al usuario elegir un valor de una lista
- *Pickers*: Un diálogo para que el usuario seleccione un valor para una lista utilizando botones o gestos en la pantalla

Cada elemento de *Input control* tiene un kit específico de manejo de eventos, como por ejemplo lo es inicialización del teclado cuando es necesaria introducción de texto.

3.1.5. Librerías de soporte

Android posee un paquete que contiene librerías para incluir en las aplicaciones que se desean desarrollar. Cada una de estas librerías soporta un rango dentro del *API Level* de la aplicación.

Para incluir librerías primero se debe nombrar el *Gradle*, éste es un kit que provee herramientas para manejar lógica de implementación de código, la configuración para implementar está definida en el archivo *build.gradle* y es allí donde se indican las librerías que el desarrollador desea importar e implementar dentro de su proyecto.

Incluir una librería, con o sin recursos, dentro de un proyecto, es necesario incluir los pasos que describa el ambiente de desarrollo utilizado para el proyecto.

3.1.6. Bases de datos

Hoy en día las bases de datos son herramientas que utilizan las corporaciones en todos los niveles. Mantienen todos sus datos importantes en bases de datos, un DBMS es una herramienta poderosa para crear y manejar grandes cantidades de información eficientemente, y permitiéndola subsistir a través del tiempo. Estos sistemas son sumamente complicados a nivel de software, entre las labores de una DBMS se encuentran las siguientes:

- Permitir al usuario crear nuevas bases de datos y especificar su estructura lógica, utilizando un lenguaje especial.
- Dar a los usuarios la habilidad para consultar los datos y modificarlos, utilizando un lenguaje de manipulación de datos.
- Soportar el almacenamiento de grandes cantidades de datos, a través de largos periodos de tiempo, permitiendo acceso eficiente a la consulta de datos.
- Permitir la durabilidad. Recuperación de los datos en caso de fallas o mal uso de los mismos.

- Controlar el acceso a los datos para varios usuarios a la vez, sin permitir interacción inesperada entre usuarios y sin permitir acciones que modifiquen los datos parcialmente.

Originalmente, los DBMS eran sistemas grandes de software que funcionaban en grandes computadores. Hoy en día, cientos de *gigabytes* de información caben en una pequeña memoria portátil, y es mucho más sencillo implementar una DBMS en una computadora personal o inclusive un *smartphone*, es un hecho que las DBMS basadas en un sistema relacional se encuentren disponibles en pequeños dispositivos, y son una herramienta común en aplicaciones de cualquier índole.

Las bases de datos actuales son presentadas al usuario con una vista de los datos organizados en tablas llamadas relaciones. Detrás de las tablas existe una compleja estructura para que la base de datos funcione como es requerido, sin embargo, el administrador o creador de la base de datos no debe preocuparse por la estructura en la que se almacenan, ya que este proceso es manejado detrás de la tabla relacional. Las consultas y los comandos de trabajo están expresados en lenguaje de alto nivel, lo que incrementa la eficiencia en la programación y creación de base de datos (J. D. Ullman, 1988).

3.2. Programación en Java

Basado en la sección 3.1 donde se indica que las aplicaciones y código de Android están basadas en el lenguaje de programación Java, es conveniente conocer algunos conceptos básicos de este lenguaje orientado a objetos, con fuente en “The Java Tutorials” (Oracle, 2007) se introducen algunos:

3.2.1. Objetos

Los objetos son clave para entender la programación orientada a objetos. En nuestro alrededor se encuentran diferentes tipos de objetos reales: un perro, una bicicleta o un televisor.

Los objetos reales comparten dos características: Todos tienen un estado y un comportamiento. Los perros tienen estado (nombre, color, raza) y comportamiento (ladrar, sentarse, mover la cola). Es importante definir el estado y comportamiento de los objetos en este tipo de programación.

Los objetos de software son conceptualmente similares a los objetos reales: también consisten en un estado y un comportamiento que los relaciona. Un objeto almacena su estado en campos (variables en otros lenguajes de programación) y exponen su comportamiento mediante métodos (funciones en otros lenguajes de programación). Los métodos operan el estado interno de un objeto y sirven como el mecanismo primario para comunicación objeto-objeto.

3.2.2. Clases

En el mundo real, es normal encontrar objetos individuales del mismo tipo. Pueden existir miles de bicicletas en el mundo real, todos de la misma marca y modelo. Cada bicicleta fue realizada desde el mismo plano, y por lo tanto contienen los mismos componentes. En el tipo de programación en cuestión, se puede decir que una bicicleta particular es una instancia de la clase de objetos llamada bicicletas. Una clase es un plano o plantilla de la cual se crea un objeto particular.

3.2.3. Paquetes

Un paquete es un espacio que organiza una serie de clases e interfaces relacionadas. Conceptualmente los paquetes son similares las carpetas en un computador. Se pueden guardar archivos de un tipo en una carpeta, imágenes en otra y ordenarlos como se desee. Basado en que el software escrito en Java puede ser realizado a partir de cientos de miles de clases individuales tiene sentido organizar clases relacionadas dentro de paquetes.

La plataforma de Java provee una enorme clase de librerías (grupo de paquetes) que pueden ser utilizados en aplicaciones individuales. Basado en librerías que aporta Java y otros usuarios, se establece el API mencionado anteriormente, representando las tareas más comunes y siempre necesarias en un área particular donde se implemente un proyecto.

3.2.4. Excepciones

A pesar de que las excepciones no son un concepto básico en Java, las excepciones son importantes para el manejo de eventos particulares durante el funcionamiento de una aplicación o programa.

Una excepción es un evento que ocurre durante la ejecución de un programa, y este interrumpe el flujo normal de las instrucciones del mismo.

Cuando un error ocurre dentro de un método, éste crea un objeto y se lo pasa al sistema de ejecución. El objeto, llamado objeto de excepción, contiene información sobre el error, incluyendo el tipo de error y el estado del programa cuando el error ocurre. Crear dicho objeto y pasarlo al sistema de ejecución es llamado “lanzar una excepción”.

Cuando un método lanza una excepción, el sistema de ejecución intenta encontrar algo que lo maneje. El grupo de posibles elementos que manejan la excepción están ordenados y pueden ser llamados dentro del método donde ocurre el error.

Si el sistema logra encontrar un elemento que maneje la excepción, se dice que el sistema “atrapa la excepción”. En el caso contrario de que el sistema no logre manejar la excepción, el programa es terminado. El uso de excepciones es una herramienta que permite el manejo de errores y su solución dentro del desarrollo de una aplicación y su funcionamiento cotidiano.

3.3. Comunicaciones

El sistema de comunicaciones, en general, utiliza como marco de referencia el modelo OSI de interconexión de sistemas. El modelo OSI posee siete capas, empezando por la de menor jerarquía (capa física) hasta la de mayor jerarquía (aplicación) las capas son las siguientes:

- Aplicación.
- Presentación.
- Sesión.
- Transporte.
- Red.
- Enlace de datos.
- física.

La **capa física** es la que se ocupa de la transmisión y recepción de datos en el medio físico. Describe la interfaz, ya sea física, eléctrica, mecánica o funcional,

y lleva las señales a el resto de las capas.

En la capa de **enlace de datos** se transfieren los datos entre nodos de forma libre de errores, permitiendo que las capas superiores obtengan la información completa y controlando la capa física a nivel de conexiones y permisos entre nodos.

La capa de **red** controla las operaciones de redes internas, decidiendo cuál es el camino que deben recorrer los datos en el camino físico, decide las prioridades en el servicio y otros factores como enrutamiento o mapeo de direcciones.

La capa de **transporte** se asegura de que los mensajes sean enviados libres de errores, en secuencia y sin pérdidas o duplicados. El tamaño y la complejidad del protocolo de transporte depende del tipo de servicio que se obtenga de la capa de red.

La capa de **sesión** establece conexiones entre procesos que se comunican a través de la red. Incluye reconocimiento de nombres, seguridad, inicio de sesión y procesos de autenticación.

La capa de **presentación** es la que le da el formato a los datos para ser presentados a la capa de aplicación, se puede ver como el traductor para la red, esta capa debe traducir datos desde el formato usado en la capa de aplicación a un formato más común en la capa de transporte.

Finalmente, la capa de **aplicación** es la ventana para los usuarios y procesos que utilizan los servicios de la red. Entre sus funciones está el acceso a los archivos, manejo de las redes, terminales virtuales o procesos internos de comunicación (Microsoft Support, 2014).

3.3.1. Protocolo TCP

Un protocolo es un formato de mutuo acuerdo para realizar una acción entre dos partes. Respecto a la comunicación digital, se refiere a un grupo de reglas o estándar que permite a los computadores o dispositivos conectarse y transmitir datos entre sí.

El protocolo TCP es orientado a conexiones, lo que significa que establece y mantiene conexiones virtuales entre anfitriones, de manera que éstos realicen el intercambio de información que requieran y luego terminen la conexión. Divide el mensaje para ser transmitido en paquetes, los enumera y luego los envía a una dirección IP de programa. Aunque los mensajes vayan dirigidos a la misma dirección IP, no tienen porque llegar en el orden en el que fueron enviados.

El protocolo utiliza corrección de errores y técnicas de control de datos para asegurarse de que los paquetes lleguen a su destino sin corromper su contenido y en la secuencia correcta, haciendo la conexión prácticamente libre de errores.

El protocolo TCP opera en la **capa de transporte** dentro del modelo OSI. Es el protocolo más utilizado por aplicaciones que requieren transmisión de datos confiable.

3.3.2. Socket

Los *socket* son utilizados normalmente para interacción entre cliente y servidor. La configuración típica coloca al servidor en una máquina y los clientes en otro grupo de dispositivos. Los clientes se conectan al servidor, intercambian información y luego se desconectan.

Un *socket* tiene un flujo estándar de eventos. En un modelo de conexión

cliente-servidor, el *socket* en el servidor procesa las peticiones del cliente, para hacer esto, el servidor debe establecer una dirección para ser ubicado. Cuando la dirección existe, el servidor espera a los clientes por la petición de un servicio. A partir de la existencia de la petición existe el intercambio de datos entre cliente y servidor a través del *socket*. El servidor realiza la acción requerida por el cliente y luego le contesta. En la figura 3.2 se puede observar el flujo típico de eventos entre la conexión e intercambio entre un cliente y servidor a través de un *socket*(IBM Knowledge, s.f.).

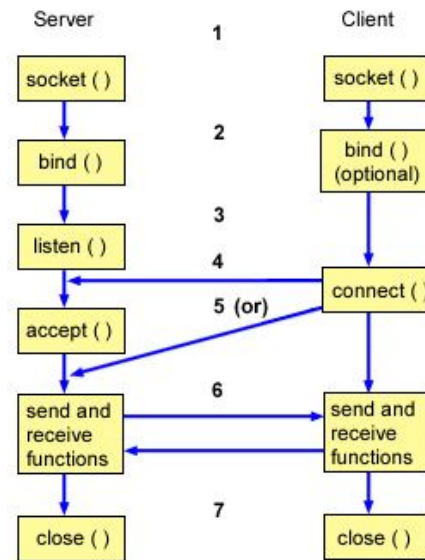


Figura 3.2. Flujo de eventos en una sesión de *socket*(IBM Knowledge, s.f.).

Explicando el flujo de eventos, se tiene la siguiente sucesión:

- La llamada a *socket()* crea un punto de comunicación.
- Cuando una aplicación obtiene el punto, se le asigna un nombre único al *socket*. De este modo es accesible en la red.
- la llamada de *listen()* indica que existe la posibilidad de aceptar peticiones de un cliente. Es necesario que esta conexión ocurra luego de asignar un nombre al *socket*.

- El cliente se conecta al servidor
- El servidor acepta la petición a conexión del cliente, basándose en el nombre adecuado y la llamada de *listen()*.
- Luego de que se establece la conexión, se puede realizar el intercambio de información dependiendo de la disponibilidad de interacción: *send()*, *recv()*, *read()*, *write()* o funciones particulares.
- Cuando se desea finalizar la conexión, el *socket* debe ser cerrado y liberar los recursos que utiliza el sistema para este proceso.

Es complicado ubicar al *socket* dentro de alguna capa del modelo OSI, ya que entre la estructura del *socket* y la interacción que tiene mediante el resto de las capas, incluyendo la conexión TCP, se cumplen varios procesos referentes al modelo (transporte, sesión, red y aplicación) En la figura 3.3 se puede apreciar la relación entre las capas y los protocolos implementados en el modelo OSI.

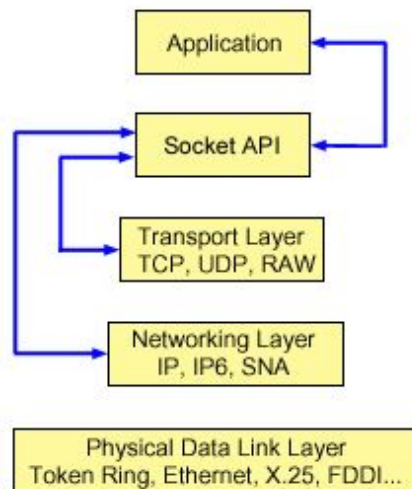


Figura 3.3. Ubicación del *socket* en el modelo OSI(IBM Knowledge, s.f.).

3.3.3. Puerta de enlace

Una puerta de enlace o *gateway* ocupa la labor de comunicar redes que utilizan diferentes protocolos de comunicación. Una puerta de enlace puede contener dispositivos necesarios para proveer interoperabilidad entre sistemas, como puede ser convertidores de protocolo, transformadores de señales y procedimientos para establecer procedimientos administrativos entre redes(Wikipedia, 2014b).

3.4. Conceptos de control supervisorio

Un sistema de control supervisorio permite obtener los datos o señales, y realizar ajustes de manera remota a un controlador para que actúe sobre un proceso de la manera deseada.

Los sistemas SCADA nacen bajo la necesidad de supervisar y controlar señales en procesos de alta peligrosidad y donde intervengan una gran cantidad de elementos distribuidos en lugares de grandes dimensiones, que impliquen grandes distancias a recorrer. Actualmente, se están implementando a niveles industriales sistemas SCADA web, los cuales permiten acceder al proceso de control desde un dispositivo con conexión a internet. Esto es de gran utilidad e importancia debido a que el operador puede acceder al sistema de control, recibir notificaciones de alarmas o realizar supervisión del sistema vía web, aún cuando éste no se encuentre en la planta Berbis (2012).

3.4.1. Unidades remotas

Una RTU es un dispositivo electrónico que se encarga llevar variables físicas a un sistema maestro, con la finalidad de realizar mediciones de las variables o de controlarlas mediante mensajes del mismo maestro. la comunicación con

una RTU se puede realizar de varias formas, ya sea a partir de un medio físico, como ethernet, RS-232, RS-485 o por Wi-Fi. Puede monitorear o controlar parámetros tanto digitales como analógicos, una RTU puede tener un protocolo de comunicación predefinido por los medios previamente nombrados, o también puede ser configurable mediante el establecimiento del medio físico de transporte de la información, o mediante la implementación de un gateway para generar compatibilidad con dispositivos o redes particulares.

3.4.2. Interfaz hombre-máquina

El HMI es comúnmente usado para referirse a la interfaz que controla un sistema mecánico, en la interacción entre humanos y computadores, la interfaz de usuario se refiere a la información gráfica, textual y auditiva que el programa le presenta al usuario, y las secuencias de control que el usuario emplea para controlar un programa. Para trabajar con un sistema, el usuario debe ser capaz de controlar el sistema. La idea de una interfaz hombre-máquina es que sea lo más sencilla posible, fácil de entender con la finalidad de minimizar errores en su uso (Ravi, 2009).

Existen ahora términos modernos para interfaces que solo controlan dispositivos a nivel de software, como lo es una interfaz táctil en los dispositivos Android actuales, por lo tanto la interfaz se lleva su propio concepto dentro del desarrollo de aplicaciones para dispositivos móviles.

3.4.3. Modbus

El protocolo Modbus es una estructura de mensajes utilizada para establecer comunicación maestro-esclavo/cliente-servidor entre dispositivos inteligentes. Es un estándar *de facto*, abierto y el más utilizado en la industria. Puede ser utilizado para transferir datos discretos, analógicos, de entrada/salida o registros en

dispositivos de control (Modbus Organization, 2005).

Entre las características del protocolo Modbus se puede destacar:

- Existe un solo maestro.
- Los nodos esclavos no exceden de 247.
- La comunicación es iniciada por el maestro.
- Los nodos esclavos nunca transmiten sin haber recibido una petición del maestro.
- Los nodos esclavos nunca se comunican entre ellos
- El protocolo es público, lo que permite implementarlo en diversos dispositivos y fabricantes.

Un mensaje de Modbus enviado por el maestro al esclavo consta de varios elementos:

- Dirección del esclavo.
- El comando o pregunta a realizar.
- Los datos a enviar.
- Un chequeo de errores y verificación de integridad

La pregunta

Modbus se puede encontrar en diferentes variantes, como lo es Modbus RTU, Modbus ASCII o Modbus TCP. Según RTA Automation (s.f.):

Modbus RTU/ASCII

Modbus RTU es un protocolo serial (RS-232 o RS-485) empleando la misma arquitectura maestro-esclavo nombrada anteriormente. El protocolo Modbus está considerado en el protocolo OSI en la capa de aplicación (capa 7).

Para que exista la comunicación, el cliente debe iniciar una transacción de Modbus. Es, entonces, a partir de la función que se le informa al servidor sobre qué acción debe realizar. El código de la función está introducido en solo un byte del mensaje, solo códigos entre el rango de 1 a 255 son considerados válidos, con el rango de 128-255 reservado para respuestas de excepción. Las tramas de Modbus RTU están diseñadas solo para enviar datos; no tienen la posibilidad de enviar parámetros como nombres, resolución o unidades.

Existen ciertas diferencias del protocolo Modbus entre sus versiones ASCII y RTU, básicamente la diferencia entre estos dos modos es la forma en que el mensaje está codificado. Los mensajes en el formato ASCII son legibles, mientras que el mensaje en formato RTU son transmitidos en código binario. El costo de tener un mensaje en ASCII es que las tramas son más largas, y por lo tanto se transmiten menos datos en más tiempo. Es imposible comunicar Modbus RTU con Modbus ASCII.

Los mensajes en Modbus ASCII son codificados con valores hexadecimales, representados en ASCII (0-9 y A-F) por cada byte de información. Por cada byte de información, se deben enviar dos bytes de comunicaciones. En el modo RTU de Modbus, cada byte de información representa directamente su valor en comunicación.

Los mensajes en Modbus no son entregados de manera directa o plana. Estos son contruidos de una forma que permite al receptor detectar el principio y el final

	Modbus ASCII	Modbus RTU
Caracteres	ASCII/HEX	Binario
Chequeo de errores	LRC	CRC
Inicio de la trama	':'	Silencio de 3.5 char
Fin de la trama	CR/LF	Silencio de 3.5 char
Espacios en el mensaje	1 seg	Silencio de 1.5 char
Bit de inicio	1	1
Bits de datos	7	8
Paridad	Par/impar (o ninguna)	Par/impar (o ninguna)
Bits de parada	1 (o 2)	1 (o 2)

Tabla 3.1. Diferencias entre Modbus RTU y Modbus ASCII (RTA Automation, s.f.)

de una trama. El caracter que inicia la trama en el modo ASCII es el ':' y el que finaliza la trama es un CR/LF. Modbus RTU utiliza un método diferente. Cada mensaje es construido midiendo espacios de silencio en la línea de comunicación. Antes de cada mensaje, debe haber un mínimo de 3.5 char de espacio vacío. Para que un mensaje termine, el receptor debe tener 3.5 char de espacio vacío al final de la trama. Esto permite que la comunicación sea continua y los mensajes sean recibidos y diferenciados entre datos en blanco. En la tabla 3.1 se representan algunas diferencias entre Modbus RTU y Modbus ASCII:

Modbus TCP

Modbus TCP es otra de las variantes del protocolo Modbus, ésta cubre el uso de Modbus en redes que están implementadas bajo protocolos TCP/IP. El mayor uso de esta variación está destinada a la comunicación con dispositivos Ethernet, como PLC's, módulos I/O o gateways que conviertan los datos.

Una conexión TCP permite llevar el control de las transacciones individuales de Modbus, llevándola en una trama identificada y supervisada, que no requiere acciones adicionales a las que ya realiza el maestro o el esclavo. Esto le da al

sistema tolerancia a cambios en el rendimiento del sistema, permite el uso de seguridad como firewalls o proxies sin relacionarse directamente con el uso de Modbus.

Los mensajes en Modbus TCP no son más que el mensaje de Modbus envuelto en una trama TCP/IP, con ciertas diferencias. El mensaje de Modbus RTU/ASCII tiene cálculos del CRC/LRC para garantizar la integridad del mensaje, los cálculos del CRC y LRC no son necesarios dentro de la trama de Modbus TCP, ya que el mismo protocolo posee su propia verificación de integridad a nivel de TCP. Una diferencia adicional es el uso de identificador en el mensaje de Modbus RTU/ASCII, ya que para comunicarse con un esclavo, la trama de Modbus debe tener el ID del esclavo en cuestión. Para casos generales de Modbus TCP, el maestro se comunica directamente con una dirección IP y un puerto, que basta para identificar el esclavo con el cual se desea comunicar el maestro.

Un detalle adicional que posee la variación TCP de Modbus, es que cualquier maestro que se conecte a la red y tenga conocimiento de la dirección IP de un esclavo, se podrá comunicar y realizar peticiones/transacciones dentro de la red, a diferencia de la estructura de Modbus RTU/ASCII, donde solo existe una unidad maestra en la red (RTA Automation, 2010).

Modbus RTU encapsulado en TCP/IP

La implementación de Modbus RTU encapsulado en TCP/IP no es más que el mensaje original de Modbus RTU o ASCII introducido íntegramente dentro de una trama de TCP/IP, incluyendo los cálculos del CRC y la identificación del esclavo con el cual se desea comunicar el maestro. El uso de Modbus RTU encapsulado en TCP involucra, la mayoría de las veces, el uso de un *gateway* que pueda “desencapsular” el mensaje original de Modbus RTU, y llevarlo de la red Ethernet/LAN a la red RS-485 donde existe el protocolo clásico Modbus RTU

(RTA Automation, 2010).

Funciones de Modbus

Con fuente en Modbus Tools (2002), Las diferentes funciones que puede enviar un maestro como petición a un esclavo son, para el caso general, las siguientes:

Función 01. *Read Coils*: Lee el estado *ON/OFF* de bobinas discretas en el esclavo.

Función 02. *Read Discrete Inputs*: Lee el estado *ON/OFF* de entradas discretas en el esclavo.

Función 03. *Read Holding Register*: Lee el contenido binario de un registro en el esclavo.

Función 04. *Read Input Register*: Lee el contenido binario de un registro de entrada en el esclavo.

Función 05. *Write Single Coil*: Escribe en una bobina el estado *OFF* o el estado *ON*.

Función 06. *Write Single Register*: Escribe un valor en un registro del esclavo.

Función 15. *Write Multiple Coils*: Escribe en múltiples bobinas en secuencia el estado *OFF* o el estado *ON*.

Función 16. *Write Multiple Registers*: Escribe múltiples valores de registros en secuencia.

CAPÍTULO IV

MARCO METODOLÓGICO

Para el desarrollo de este proyecto, se plantean diferentes conceptos que permitirán enmarcar la metodología del trabajo que se realiza. Partiendo del planteamiento del problema, el objetivo general y de los objetivos específicos a cumplir se enumera de la misma forma la metodología según la cual se realiza cada objetivo, y finalmente una breve reseña de los recursos y equipos disponibles para el proyecto.

4.1. Metodología

1. Inicialmente se plantearán los equipos físicos que van a formar parte del sistema, así como su relación con el componente de programación a implementar, permitiendo entonces realizar el diagrama funcional que explique visualmente la topología del proyecto.
2. Conociendo la topología del sistema, se tomarán en cuenta diferentes ambientes de desarrollo para aplicaciones bajo Android, lo que permitirá tener una visión amplia de las posibilidades de programación presentes, y que entonces, según las ventajas y desventajas planteadas, se podrá elegir adecuadamente un ambiente que cumpla con las características para implementar el software como se espera. Se recopilará información sobre estándares de comunicación que permitan al dispositivo Android recibir la información de las variables necesaria para completar la base de datos del programa, y posteriormente seleccionar el más adecuado según las exigencias

del ambiente de desarrollo y las posibilidades planteadas por los equipos disponibles. Finalmente, se plantearán al menos dos elecciones de bases de datos dentro del ambiente de desarrollo, con la finalidad de elegir la más adecuada dentro de las necesidades del proyecto y facilitar el aprendizaje y manejo de sus características.

3. A partir del diagrama funcional, del ambiente de desarrollo planteado y de los estándares de comunicación conocidos, se procederá con la implementación del programa, que incluye la relación de los componentes que establecen la comunicación con las variables, los elementos de base de datos y una interfaz hombre máquina que permita manipular y configurar la aplicación final.
4. Finalizado el desarrollo de la aplicación, se realizarán pruebas en tiempo real de la misma, con equipos e instalaciones que permitan conocer las capacidades de la aplicación y que den pie a una evaluación de su funcionamiento en un ambiente físico real. Según los resultados de las pruebas realizadas, y las necesidades encontradas, se realizarán posibles ajustes al programa desarrollado, con la finalidad de finiquitar, pulir y depurar procesos con posibles deficiencias.
5. A partir de la información recopilada con las pruebas realizadas, se redactará un informe que permita a un usuario final conocer la aplicación desarrollada, su función principal, limitaciones y posibles configuraciones necesarias para su funcionamiento.

4.2. Financiamiento y recursos requeridos

Para realizar el proyecto se dispone de todo el material necesario:

- Equipos para realizar pruebas en tiempo real, tanto unidades de comunicación como diferentes dispositivos Android para distintos planteamientos.
- Recursos de software como pueden ser los programas de bases de datos, ambientes de desarrollo de aplicaciones y programas de software libre necesarios.
- Computador apto para el desarrollo, pruebas e instalación de programas.
- Artículos de oficina que se puedan requerir durante el avance del proyecto.

CAPÍTULO V

PROTOCOLO DE COMUNICACIÓN, AMBIENTE DE DESARROLLO Y DIAGRAMA DEL SISTEMA

En este capítulo se mostrarán tanto los equipos físicos como parte del software que formará parte del proyecto. Partiendo de la selección del ambiente de desarrollo y el protocolo de comunicación, se realizará un diagrama donde se relacionen los diferentes equipos y elementos que forman parte del sistema. Adicionalmente se hace la introducción del motor de bases de datos a implementar en el proyecto y su relación con el sistema operativo.

5.1. Elección del protocolo de comunicación

Actualmente en el ambiente de aplicaciones para Android, no se ha desarrollado de manera extensa la implementación de protocolos de comunicación utilizados en la industria de automatización, por lo tanto la elección de un protocolo para comunicarse con dispositivos de adquisición de datos se vuelve complicada.

En los proyectos actualmente desarrollados, tomando como referencia inicial *Modbus-Droid*, el protocolo Modbus es el más desarrollado actualmente, con diferentes librerías implementadas y funcionales que permiten el desarrollo de aplicaciones y programas basados en Modbus. Estas librerías vienen desde el entorno de programación de Java, y son utilizadas con éxito dentro del desarrollo de aplicaciones de Android ya que son librerías sumamente completas que pueden lidiar con casos tan exigentes como los que puede afrontar un programa

desarrollado para un computador.

Entre otras opciones para los protocolos de comunicación, los desarrolladores de *TeslaSCADA* implementaron la comunicación con dispositivos Siemens y Allen Bradley para una cantidad de equipos limitada y con protocolos como Siemens ISO/TCP o Ethernet/IP.

La información de librerías ISO/TCP de Siemens o Ethernet/IP no es de acceso común como lo es la información y conocimiento sobre las librerías de Modbus en cualquiera de sus versiones. Adicionalmente a las dificultades de disponibilidad de protocolos privados, los equipos disponibles para realizar el proyecto actual no están en capacidad de realizar pruebas con los protocolos mencionados.

Entre las diferentes modalidades del protocolo Modbus, las librerías de Modbus4J, Jamod o J2mod pueden ser configuradas en la modalidad que se desee implementar, abriendo la posibilidad de que se lleve a cabo el proyecto utilizando el protocolo Modbus dentro de un dispositivo Android de manera exitosa. Tomando en cuenta las librerías disponibles para Modbus, se considera éste como la opción más apropiada para implementar la comunicación entre un dispositivo Android y equipos con la capacidad de medir variables reales y entregarlas con seguridad a través de un protocolo estándar.

5.2. Elección del ambiente de desarrollo

Durante el desarrollo del proyecto será necesario elegir un único ambiente de desarrollo. Actualmente existen diferentes opciones al momento de desarrollar un proyecto para Android, sin embargo, para diciembre de 2014 Google anuncia que está por terminar el desarrollo de su propio IDE, Android Studio, y que dejará de soportar su plugin de ADT para el ambiente de desarrollo Eclipse, que para momentos del anuncio era el ambiente más utilizado por los desarrolladores

de aplicaciones para Android. El lanzamiento de Android Studio proveerá de muchas ventajas a las que ofrece actualmente Eclipse en su ambiente de desarrollo (Duckett, 2014).

El ambiente de desarrollo Android Studio fue anunciado en 2013 en el evento *Google I/O* y a partir de su lanzamiento Google lo considera su “IDE oficial”, y recomienda a todos los desarrolladores de Android migrar desde cualquier ambiente que utilicen al que ellos desarrollan. Entre las características iniciales que ofrece Android Studio se encuentran las siguientes:

- Un instalador que prepara el SDK adecuado y un ambiente de desarrollo con emulador optimizado y plantillas de trabajo.
- Todas las herramientas de codificación implementadas por otros ambientes como *IntelliJ IDEA*.
- Vista previa del *Layout* de forma dinámica, permitiendo agregar y remover elementos de la interfaz vía *Drag&Drop* para múltiples versiones de la API.
- Monitoreo del desempeño de la memoria de los dispositivos.
- un sistema constructor basado en *Gradle* integrado con el ambiente de desarrollo, pero que es independiente en versión y actualizaciones para garantizar estabilidad.
- Se pueden generar diferentes versiones de una aplicación utilizando el mismo código fuente.
- Integración con la plataforma de *Google Cloud*.
- Herramientas para lidiar con problemas de compatibilidad de dispositivos, versiones y usabilidad de una aplicación.
- Posibilidades de seguridad y firma de aplicaciones integrada.

Basado en el lanzamiento de Android Studio la elección del ambiente de desarrollo se vuelve sencilla, ya que las herramientas que proporciona este ambiente de desarrollo cumplen prácticamente con todas las que ofrece cualquier ambiente de desarrollo, y es el ambiente de desarrollo que será soportado a través del tiempo por los desarrolladores de Google. Tomando en cuenta la elección de Android Studio como ambiente de desarrollo a utilizar para este proyecto, vale la pena revisar con mayor profundidad sus características, con fuente en la página web de desarrolladores de Google.

5.2.1. Estructura de archivos y proyectos

Por defecto, los proyectos en Android Studio se visualizan desde la vista Android, es una vista simplificada de los archivos de proyecto, constructores y recursos, se pueden manejar los archivos básicos con acceso simplificado. En la figura 5.1 se puede observar un ejemplo de la vista en cuestión.

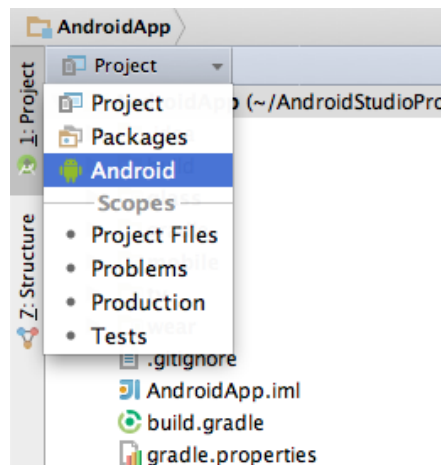


Figura 5.1. Vista de proyecto Android. (Android Developers, s.f.-a)

Adicionalmente a la vista Android existen otros tipos de vista para los archivos de un proyecto. En general, la vista tipo proyecto permite una visión más clásica de los archivos, con una estructura de carpetas organizada más familiar a la utilizada comúnmente en Eclipse. Finalmente, el ambiente de desarrollo Android

Studio permite realizar vistas de los archivos de proyecto personalizadas según el usuario y que se orienten a las características requeridas para el manejo deseado del proyecto.

5.2.2. Rendimiento y depuración

Android Virtual Device (AVD)

El gerente del AVD posee pantallas actualizadas que permiten elegir los dispositivos más recientes y las configuraciones más populares, con resoluciones reales para previsualizar las aplicaciones.

Tiene la posibilidad de crear dispositivos personalizados, basados en procesadores específicos y propiedades particulares. Android Studio instala el HAXM y crea un emulador predeterminado para probar las aplicaciones.

Depuración entre líneas

Se puede utilizar la depuración entre líneas para mejorar el código al momento de ser depurado, se pueden verificar referencias, expresiones y valores de las variables. La depuración entre líneas incluye:

- Valores de variables entre líneas.
- Referencia de objetos que referencian a objetos seleccionados.
- Valores que regresen métodos.
- Expresiones *Lambda* y operadores.
- Información sobre herramientas.

Monitor de memoria y CPU

Android Studio provee un monitor de memoria y CPU para poder llevar control del desempeño de las aplicaciones, ubicar filtraciones de memoria, llevar control del uso de la memoria y uso del CPU. La vista de monitoreo posee una pestaña en la que se puede verificar el uso de memoria y CPU al momento de emular o probar las aplicaciones.

Las herramientas del SDK, como el *logcat*, *SysTrace* o *Traceview* generan datos de desempeño y depuración para realizar análisis detallados. En la figura 5.2 Se observa un ejemplo del monitor de memoria y CPU.

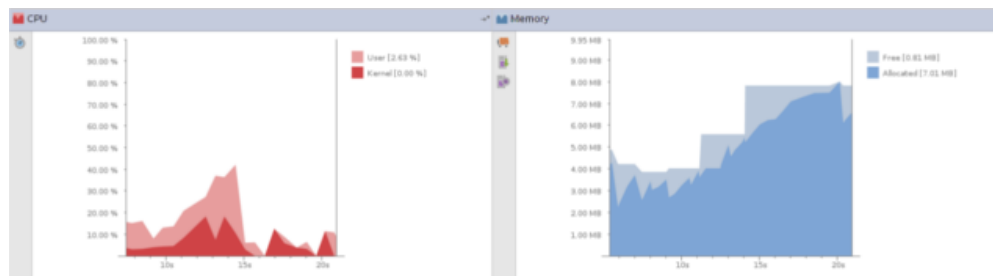


Figura 5.2. Monitor de memoria y CPU en Android Studio (Android Developers, s.f.-a)

Previsualización dinámica del *Layout*

El ambiente de desarrollo permite trabajar con un *Layout* en modo de texto y diseño, se puede intercambiar fácilmente entre ambos modos. El modo texto funciona similar al desarrollo de una interfaz HTML, mientras que el modo de diseño permite integrar elementos mediante *Drag&Drop* desde una paleta con diferentes elementos disponibles. Se puede observar una imagen del modo diseño en la figura 5.3

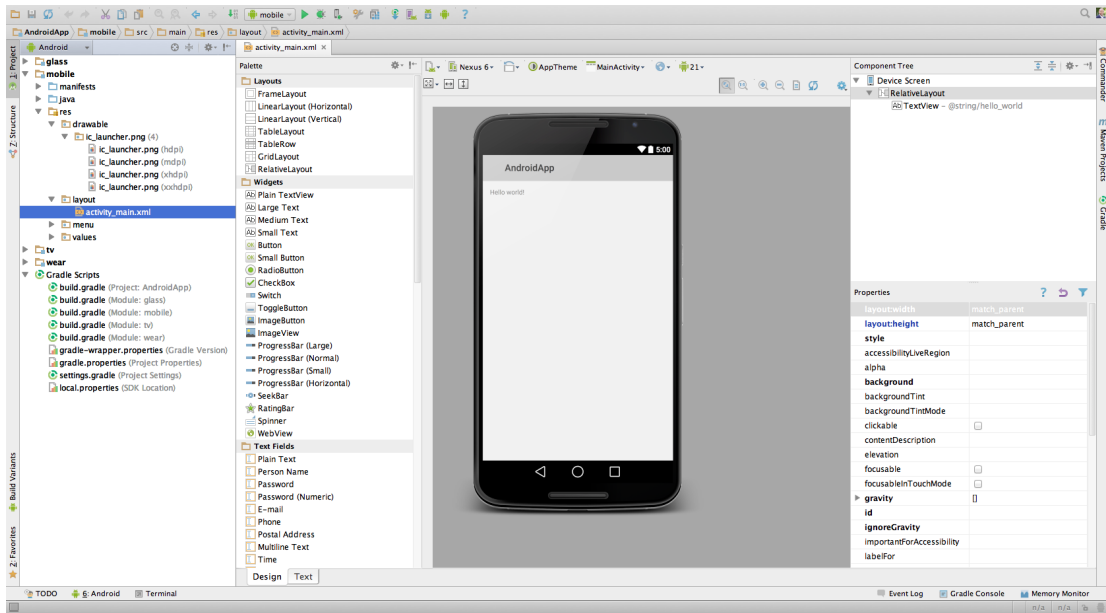


Figura 5.3. Modo diseño de la interfaz en Android Studio (Android Developers, s.f.-a)

5.3. Elección de base de datos

Entre las elecciones que se debe realizar para implementar la aplicación en Android, es necesario elegir una librería que funcione como soporte para bases de datos. Actualmente la opción recomendada por los desarrolladores de Android, en la página de Android Developers, es SQLite. SQLite posee un paquete de clases directamente entre las características de Android, por lo tanto se puede considerar prácticamente un paquete nativo para Android. SQLite es un motor de base de datos que es *self-contained*, *serverless*, *zero-config* y *transactional*.

Un paquete como SQLite, *self-contained* se refiere a que no requiere librerías externas del sistema operativo para ser implementada. Lo que lo hace ideal para sistemas embebidos que no requieren una infraestructura de servidores o computadores personales para funcionar. Esto completa también el concepto de *serverless*, es un paquete que no requiere servidores, ya que sus procesos son particulares dentro del sistema y son accesibles para el resto del sistema opera-

tivo. Cuando los desarrolladores de SQLite se refieren a un paquete como *zero-configuration*, hablan de un paquete que no requiere instalación o proceso de puesta en marcha, configuración o encendido del mismo. Un paquete de bases de datos *transactional* es aquella en la cual los cambios y consultas son basados en una serie de propiedades particulares, como son las siguientes:

Atomicity: La atomicidad se refiere a que todas las transacciones deben ser “todo o nada”. Si una parte de una transacción falla, la transacción falla en su totalidad y no cambia el estado de la base de datos.

Consistency: La consistencia es una propiedad que asegura que cualquier transacción va a llevar a la base de datos de un estado válido a otro estado válido. Cualquier dato escrito en la base de datos debe ser válido de acuerdo a las reglas establecidas por el administrador.

Isolation: La propiedad de aislamiento es aquella que se asegura de que la ejecución concurrente de transacciones resulte en un estado del sistema correcto mientras todas las transacciones sean realizadas exitosamente en forma serial, es decir, una después de la otra.

Durability: La durabilidad se refiere a que una transacción, una vez que sea realizada, se mantenga realizada ante cualquier evento inesperado, así sea errores, terminaciones súbitas de un proceso o simplemente una falla de alimentación al sistema.

SQLite es actualmente el motor de bases de datos más utilizado alrededor del mundo, y los recursos de programación son totalmente de dominio público y libre. (SQLite, 2009)

Otro motor de bases de datos que se puede plantear es MongoDB. Un motor basado en data flexible, a diferencia de SQL, donde se deben determinar tablas

antes de introducir datos en una base de datos. Sin embargo, debido a la fácil implementación y puesta en marcha de SQLite dentro de Android, es la elección a utilizar en este proyecto.

5.4. Equipos físicos y diagrama funcional del sistema

Entre los equipos físicos a utilizar, se deberá conocer el dispositivo con Android a utilizar, un computador personal capaz de manejar software como el ambiente de desarrollo y simuladores de protocolos para realizar pruebas del sistema. Adicionalmente se nombrarán los equipos físicos de los que se dispone para realizar pruebas reales de la aplicación que se realiza.

5.4.1. Dispositivo Android

Se dispone de un *smartphone* Samsung Galaxy S3 (GT-I9300) para realizar la aplicación y pruebas de software en Android. Es un dispositivo moderno y posee características físicas adecuadas para implementar software. Entre sus características tiene:

- Pantalla de 4.8' con una resolución de 720x1280 píxeles
- Procesador Quad core de 1.4GHz Cortex-A9
- 1GB de memoria RAM
- Wif-Fi 802.11 a/b/g/n
- Android 4.4.4 KitKat

5.4.2. Computador y programas

Se posee un computador personal tipo *desktop* con las características suficientes para utilizar programas relacionados al desarrollo de aplicaciones en Android, y programas necesarios para simular protocolos de comunicación.

Modbus Slave

Uno de los programas importantes a utilizar es *Modbus Slave*, realizado por *ModbusTools* (www.modbustools.com). Este programa es una herramienta para simular hasta 32 dispositivos esclavos dentro de 32 ventanas, esto permite realizar pruebas de la aplicación durante el desarrollo del proyecto. Soporta las funciones 01, 02, 03, 04, 05, 06, 15, 16, 22 y 23 de Modbus y otras características importantes como es la configuración del esclavo (ID, puntos, variación de Modbus). Se puede editar la data de los esclavos en Excel o configurarlo en su propia ventana.

5.4.3. Diagrama funcional del sistema

A partir de los equipos disponibles, y Modbus como protocolo de comunicación, una topología basada en un gateway, con una RTU y una red LAN, se puede implementar el sistema mostrado en la figura 5.4.

El sistema representa una RTU con diferentes sensores de cualquier tipo, que se pueda comunicar utilizando el protocolo Modbus RTU encapsulado en TCP/IP mediante un gateway que esté conectado a una red local de Wi-Fi. A partir de la red Wi-Fi es posible conectarse mediante el dispositivo Android y así funcionar como maestro de la RTU, con la finalidad de recibir los valores que esté midiendo. Tomando en cuenta que la finalidad del sistema está en recibir o monitorear los valores que mide una RTU, solo se implementarán las primeras 4

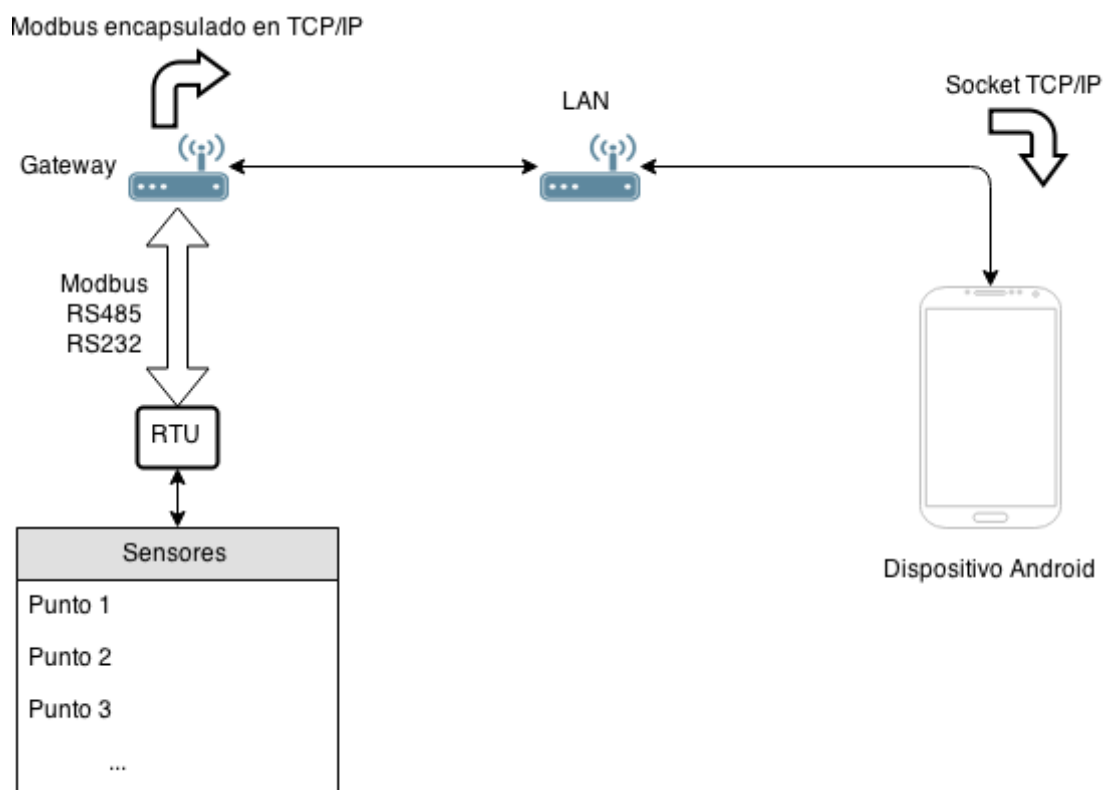


Figura 5.4. Diagrama funcional del sistema.

funciones del protocolo Modbus, que son funciones de lectura. En los próximos capítulos se presentará un diagrama interno del funcionamiento de la aplicación, que permita comprender el funcionamiento de la misma y su integración con el sistema presentado.

CAPÍTULO VI

DESARROLLO DE LA APLICACIÓN

En este capítulo esencialmente se describirá todo el proceso realizado para crear la aplicación en Android utilizando Android Studio. Adicionalmente a la descripción de todos los segmentos de código implementados, se pondrá en marcha el ambiente de desarrollo, se explicarán las librerías más relevantes utilizadas, tanto nativas como externas al sistema operativo, el desarrollo de la base de datos y las diferentes interfaces implementadas en el proyecto.

6.1. Diagrama interno de la aplicación y sus componentes

Antes de poner en marcha el ambiente de desarrollo y de realizar avances significativos con el programa, es conveniente plantear un diagrama en el cual se relacionen los componentes de la aplicación y cómo se comunican entre sí. En el diagrama de la figura 6.1 se muestra cómo se relacionan algunos elementos clave que existen dentro de la aplicación.

Las librerías de Modbus deben primero recibir los parámetros de comunicación con la unidad remota, introducidos por el usuario antes de iniciar la medición de variables. Los parámetros que el usuario debe introducir son los siguientes:

- Dirección IP.
- Puerto.

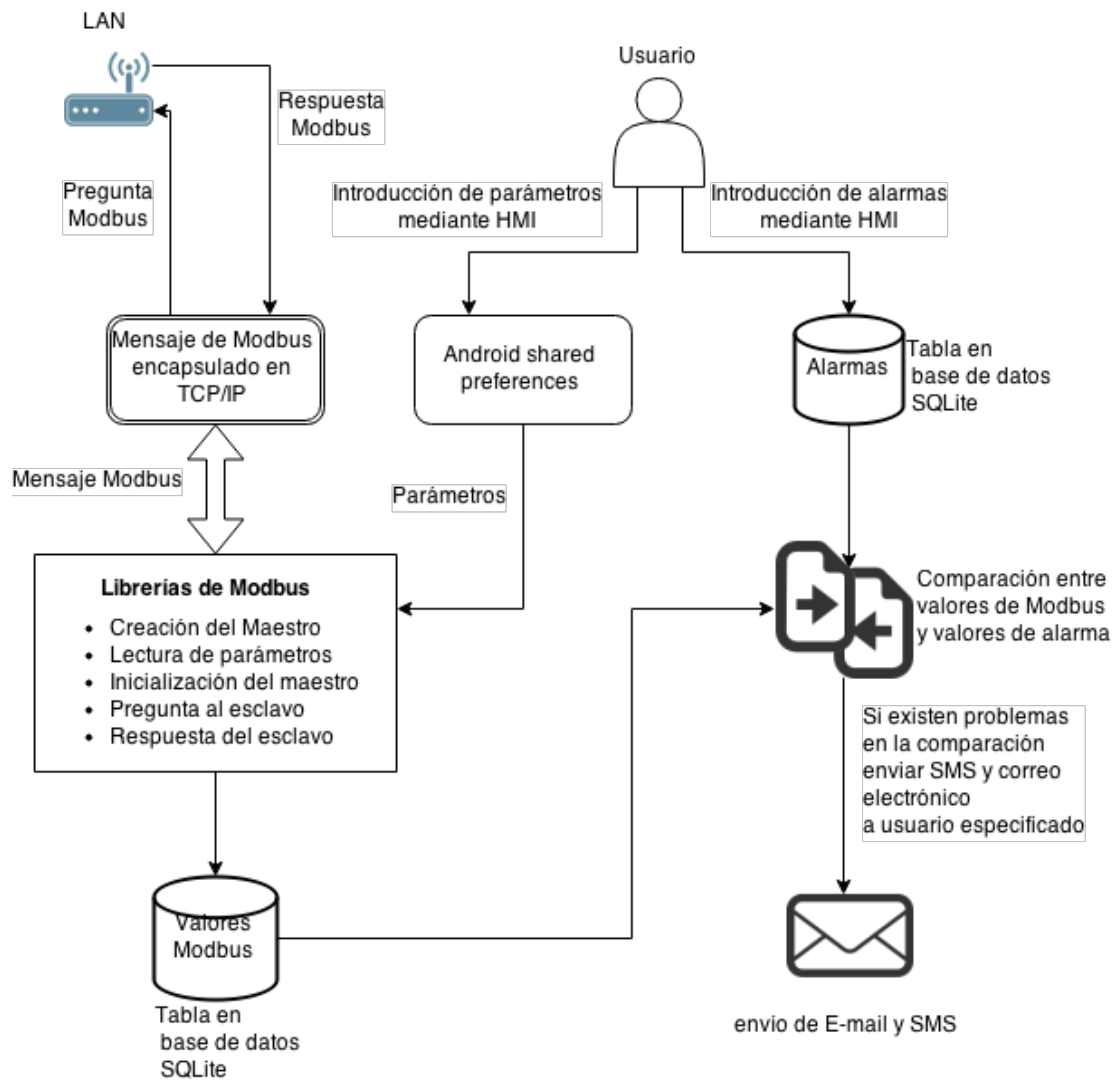


Figura 6.1. Diagrama interno de la aplicación.

- ID del esclavo.
- Punto o sensor inicial.
- Número de puntos a medir.
- Tiempo de muestreo en segundos.
- Función de Modbus con la que se va a trabajar.

Estos parámetros se guardan no en una tabla de la base de datos sino en las *SharedPreferences*, un grupo de métodos que sirven para guardar, acceder y modificar preferencias o valores que el usuario desea manejar a lo largo de toda la aplicación o en actividades particulares dentro de la misma. Estas actividades, y las librerías más importantes, están explicadas en la sección 6.6, posterior a la sección actual.

Del mismo modo que el usuario introduce los parámetros de la unidad remota, debe introducir los parámetros de alarma, valores que van a una tabla particular de la base de datos, donde podrán ser utilizados al momento de comparar con los valores de Modbus que recibe el dispositivo. Los parámetros de alarma que debe introducir el usuario son los siguientes:

- ID del esclavo
- Número del sensor o punto
- Valor mínimo aceptable para ese punto
- Valor máximo aceptable para ese punto
- E-mail del administrador o encargado del servicio
- Número celular del administrador o encargado del servicio

Mediante dichos parámetros se espera poder monitorear individualmente cada punto de una o varias unidades remotas en una misma red local, según un tiempo de muestreo fijado en los parámetros de comunicación. Actualmente los parámetros de comunicación permiten el uso de solo **una unidad remota** para el proyecto por términos de simplicidad, sin embargo, los parámetros de alarma están pensados de una forma global y más genérica en función de mejorar con el tiempo el estado de la aplicación en desarrollo.

Cuando se reciben los parámetros, y si estos parámetros son válidos, el dispositivo Android permite iniciar el envío y la recepción de mensajes de Modbus encapsulados en una trama TCP/IP que será recibida por el esclavo, que genera una respuesta enviada de vuelta al dispositivo Android. Cuando el mensaje es recibido por el dispositivo, éste se encarga de guardar los valores en otra tabla particular de la base de datos. A medida de que los valores son medidos y guardados, éstos son comparados con el rango aceptable (valores mínimos y máximos) de la tabla de alarmas, y en caso de que cualquier valor esté fuera del rango establecido previamente, se llamará a los métodos encargados de enviar tanto un correo electrónico como un SMS al encargado del servicio o administrador de la red de Modbus.

6.2. Creación de un nuevo proyecto en Android Studio

Antes de crear un nuevo proyecto en Android Studio es necesario instalar dicho programa, Android Studio puede ser descargado desde la web de Android Developers, en el siguiente link:

- <http://developer.android.com/sdk/index.html>

Luego de instalar el programa según sus instrucciones, es posible crear un nuevo proyecto en el menú principal de Android Studio mediante la selección de “Start a new Android Studio project” como se puede observar en la figura 6.2.

Luego de la selección anterior, se le debe colocar el nombre a la aplicación, en el proceso se puede elegir también el nombre del paquete a utilizar a lo largo del desarrollo del proyecto. En el caso actual se colocó como nombre de la aplicación “ModbusMaster” y como nombre del paquete “com.example.josempd.modbusmaster” como sugiere el programa.

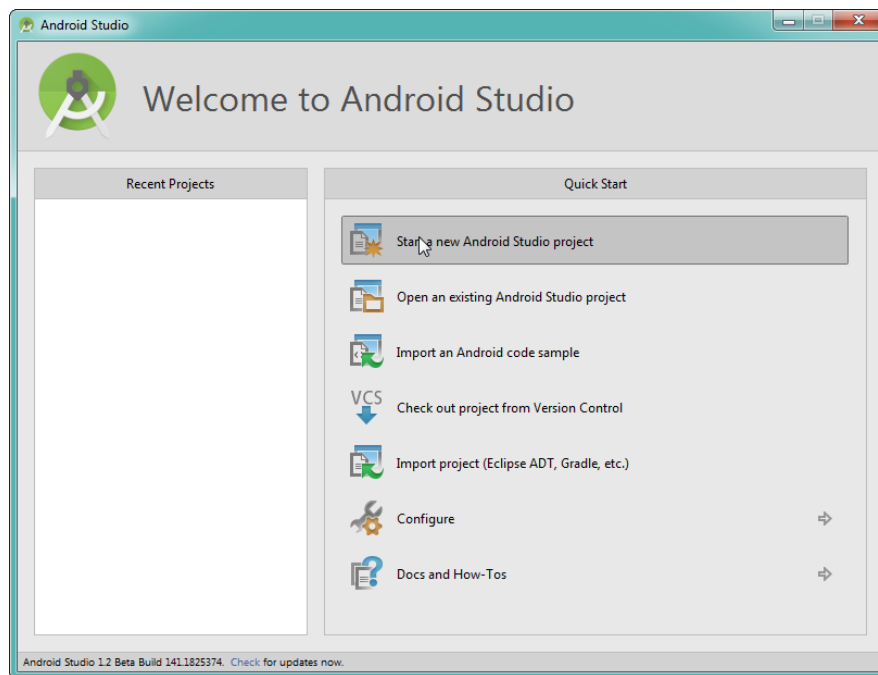


Figura 6.2. Creación de nuevo proyecto en Android Studio.

El siguiente menú pide al usuario elegir un *form factor* en el cual correrá la aplicación, para el caso actual, se elige solo la opción *Phone and Tablet* bajo el API 15, que soporta versiones anteriores de Android hasta la 4.0.3 (IceCream-Sandwich). Seleccionar el API 15 permite, según el programa, llegar al 90,4 % de los dispositivos activos en *Google Play Store* lo que se considera un número elevado de dispositivos compatibles con la aplicación a desarrollar. En la figura 6.3 se observa una captura de la selección del factor de forma y el API sobre el cual se trabaja.

Al seleccionar el factor de forma y API de la aplicación, el programa nos presentará una serie de plantillas base para iniciar el proyecto, en el caso del proyecto actual se desarrolló en base a una *blank activity*. Posterior a la selección de la plantilla de la actividad principal, el programa permite colocar un nombre para la nueva actividad, como se muestra en la figura 6.4. Para este proyecto se eligió el nombre por defecto, “MainActivity” y en esa actividad se desarrollará

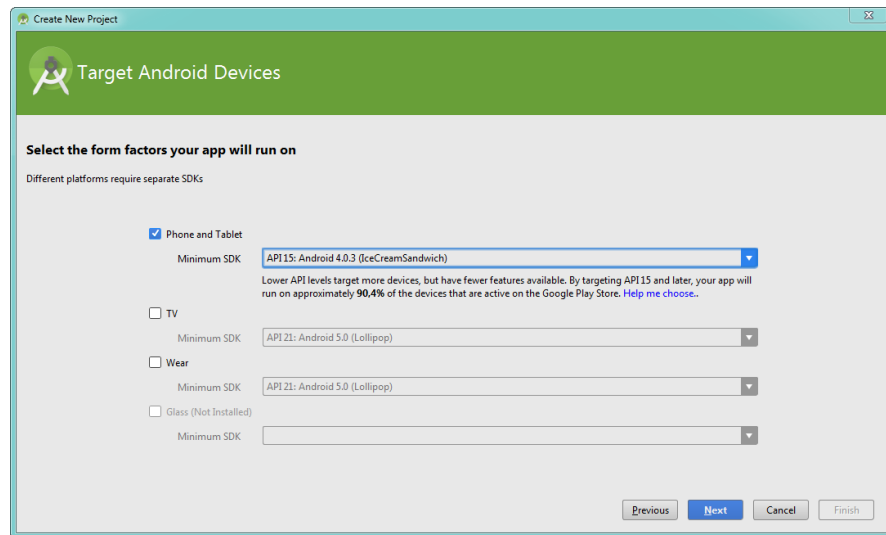


Figura 6.3. Selección del *form factor* y el API en Android Studio.

gran parte de la aplicación.

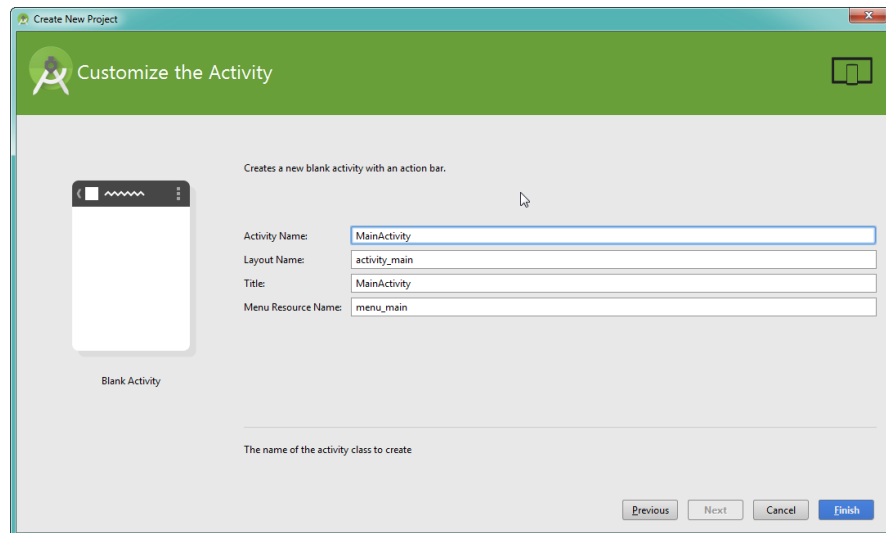


Figura 6.4. Selección del nombre de la actividad en Android Studio.

Finalmente el programa estará entonces en su ventana principal, como se observa en la figura 6.5, donde se tiene vista en la clase principal de *MainActivity.java*

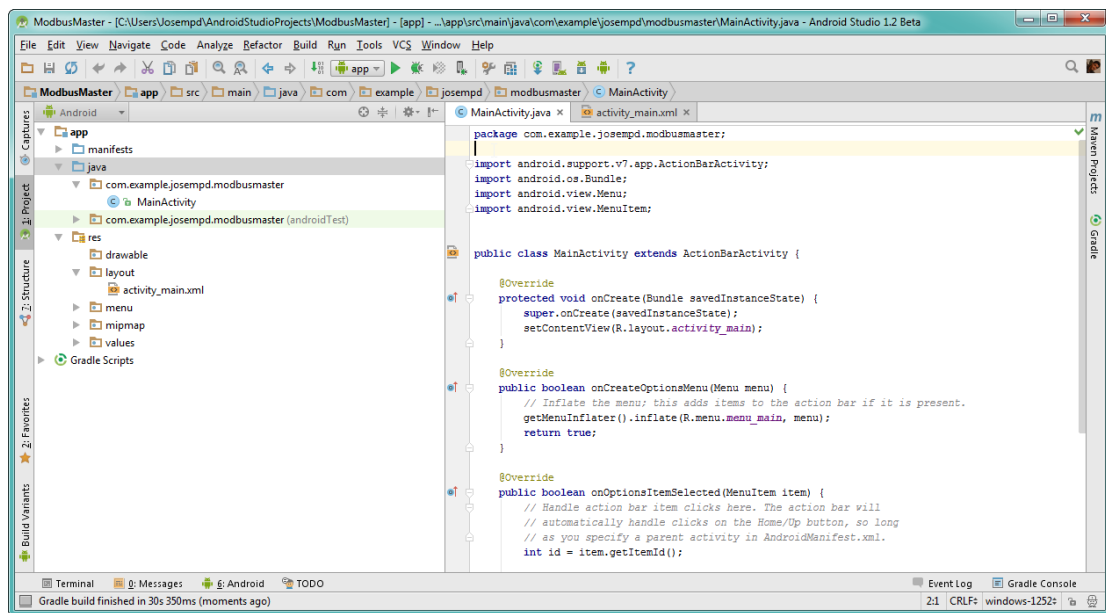


Figura 6.5. Vista por defecto y actividad principal en Android Studio.

6.3. Archivo *AndroidManifest.xml*

El archivo *AndroidManifest.xml* contiene el siguiente código, explicado en los comentarios

```
<?xml version="1.0" encoding="utf-8"?>
//nombre del paquete
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.josempd.modbusmaster" >

    //Permisos asignados por Android a la aplicacion
    //Acceso a internet y envio de SMS
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.SEND_SMS" />

    //Informacion general de la aplicacion
    //respaldo, icono, nombre y tema.
```

```

//Icono y tema son valores por defecto de Android Studio
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >

    //Primera actividad, MainActivity, es la principal
    //Se inicia con el arranque de la aplicacion
    <activity
        android:name="com.example.josempd.modbusmaster.MainActivity"
        android:label="@string/app_name" >

        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

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

    //Segunda actividad, SettingsActivity
    <activity
        android:name="com.example.josempd.modbusmaster.SettingsActivity"
        android:label="@string/title_activity_settings" >
    </activity>

    //Tercera y ultima actividad, AlarmConfig, para las alarmas
    <activity
        android:name="com.example.josempd.modbusmaster.AlarmConfig"
        android:label="@string/title_activity_alarm_config"

```

```
        android:windowSoftInputMode="adjustPan">
    </activity>
</application>

</manifest>
```

El archivo *AndroidManifest.xml* es base para la aplicación y contiene información útil para el sistema operativo en materia de identificación de la aplicación, permisos y estructura general.

6.4. Librerías externas

Para el desarrollo de la aplicación, es necesario el uso de algunas librerías ajenas al sistema operativo Android, la mayoría de estas librerías provienen de desarrolladores que poseen suma experiencia en el desarrollo de programas y aplicaciones en el lenguaje **Java**. Para introducir las librerías externas dentro del proyecto, es necesario crear una carpeta llamada “libs” en el directorio del proyecto. En el caso particular de este proyecto, el directorio es:

- C:/Users/Josempd/AndroidStudioProjects/ModbusMaster

Luego de crear dicho directorio, se pueden importar las librerías, pegándolas dentro del directorio creado y agregándolas dentro del ambiente de desarrollo. Las librerías externas utilizadas en el proyecto se presentan a continuación:

6.4.1. *Modbus4j.jar*

Esta librería, según su descripción, es una implementación del protocolo Modbus, de alto rendimiento y fácil uso, escrita en el lenguaje Java por *Serotonin*

Software. Soporta comunicación ASCII, RTU, TCP y UDP. Se puede configurar como esclavo o como maestro, peticiones automáticas y manejo gramatical de datos. (SourceForge.net, 2008)

Las librerías trabajan bajo una licencia GPLv3 (GNU General Public License) disponibles en el paquete de la librería.

Entre la información disponible del paquete descargado, se encuentra toda la documentación de clases y métodos que posee la librería. Adicionalmente incluye los paquetes *RXTXcomm.jar* y *SeroUtils.jar* utilizados por la librería de forma interna, para manejo de acciones particulares del paquete o transmisión/recepción de datos.

6.4.2. *mail.jar*

El envío de correos electrónicos de forma desasistida en Android es más complicado que el envío común, por lo tanto, se utilizó una clase creada por el desarrollador Jon Simon (2010). Las características de la clase creada son las siguientes:

- Inicialización de propiedades y valores por defecto de envío de correos
- Inicialización de javamail para manejo de correos bajo *Handler*.
- Envío de correo según las propiedades utilizadas
- método para añadir archivos adjuntos
- Uso de los servidores de Google para envío del correo

Ya que la clase creada es de uso particular, fue creado un usuario de correo electrónico para realizar la acción de envío de correos en la aplicación, el correo

electrónico creado para esta acción, junto con su clave:

Correo: alarmasmodbus@gmail.com

Clave: AlarmaModbus

Con la finalidad de no vulnerar correos personales introducidos en la aplicación.

La librería *mail.jar* utiliza los paquetes *additionnal.jar* y *activation.jar* para procesos internos del envío de correos electrónicos.

Con la lista de paquetes en el directorio donde deben ubicarse, solo resta indicar al ambiente de desarrollo la presencia de éstas librerías, esto se realiza colocando el directorio de cada paquete en el archivo *build.gradle* del módulo de la aplicación, como se muestra en el segmento de código:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 21
    buildToolsVersion "21.1.2"

    defaultConfig {
        applicationId "com.example.josempd.modbusmaster"
        minSdkVersion 15
        targetSdkVersion 21
        versionCode 1
        versionName "1.0"
    }
    buildTypes {
        release {
```



```

        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android.txt'),
            'proguard-rules.pro'
    }
}
}

dependencies {
    compile fileTree(include: ['*.jar'], dir: 'libs')
    compile 'com.android.support:appcompat-v7:22.0.0'
    compile
        files('C:/Users/Josempd/AndroidStudioProjects/ModbusMaster/libs/seroUtils.jar')
    compile
        files('C:/Users/Josempd/AndroidStudioProjects/ModbusMaster/libs/modbus4J.jar')
    compile
        files('C:/Users/Josempd/AndroidStudioProjects/ModbusMaster/libs/RXTXcomm.jar')
    compile
        files('C:/Users/Josempd/AndroidStudioProjects/ModbusMaster/libs/additionnal.jar')
    compile
        files('C:/Users/Josempd/AndroidStudioProjects/ModbusMaster/libs/mail.jar')
    compile
        files('C:/Users/Josempd/AndroidStudioProjects/ModbusMaster/libs/activation.jar')
}

```

6.5. Paquetes internos destacados

Existen algunos paquetes nativos de Android útiles para el desarrollo del proyecto, de los cuales se requiere cierto conocimiento antes de ser implementados. Toda la información de estos paquetes proviene de la fuente brindada en la página

web de *Android Developers*.

6.5.1. *SharedPreferences*

La clase *SharedPreferences* provee un marco general que permite guardar y utilizar valores clave e información valiosa de forma persistente. Se puede utilizar para guardar cualquier tipo de data primitiva: booleanos, flotantes, enteros, longs, y strings. Esta información persiste a través de una sesión, inclusive si se cierra la aplicación. Esta clase tiene varios métodos utilizados en este proyecto para manipular las variables, llamarlas y guardarlas a lo largo de la aplicación.

6.5.2. *Timer* y *TimerTask*

el uso de esta clase es básico, permite hacer tareas recurrentes de forma sencilla, cada *Timer* particular tiene su propio hilo en el que se ejecuta su tarea de forma secuencial, es posible que existan tareas de menor prioridad que la de un *Timer* y puedan ser diferidas por la presencia del mismo.

La clase *TimerTask* no es más que la posibilidad del uso de un *Timer* en un momento específico, ya sea para un uso particular o repetitivo.

6.5.3. *Handler*

Handler es una clase pública nativa, que permite encolar mensajes dentro de un mismo proceso, con la finalidad de que los eventos dentro del mismo ocurran de forma ordenada y sin interferir de manera negativa en el proceso principal. Cuando se crea un *Handler*, éste se liga al proceso en el cual funciona, por lo tanto se pueden realizar acciones particulares dentro del mismo, permitiendo mayor versatilidad, por ejemplo, dentro de un *Timer*.

6.5.4. *AsyncTask*

La clase *AsyncTask* activa el uso apropiado de tareas en un segundo plano, asíncronas al proceso principal de la aplicación, acompañando al *Handler* y a los *Timer*. Una tarea asíncrona de la *AsyncTask* puede publicar resultados en el proceso principal solo al terminar su tarea asignada. Existen métodos, utilizados en esta aplicación, para publicar el progreso en partes específicas de la tarea en segundo plano. Es necesario el uso de la *AsyncTask* para la comunicación de Modbus como el envío de correos electrónicos de alarma, ya que la apertura de un socket TCP en el proceso principal de una aplicación de Android está prohibido, con la finalidad de evitar tareas repetitivas dentro del proceso principal de una aplicación.

6.6. Desarrollo de las actividades y clases.

La aplicación final tiene tres actividades, como indica el archivo *AndroidManifest.xml*. En esta sección se explicarán las actividades y se mostrará la interfaz gráfica que posee cada una.

6.6.1. *MainActivity.java*

La primera actividad, *MainActivity.java*, es donde se realiza la mayor parte de la aplicación. Aquí es donde se inician los *Timer*, se llaman las librerías de Modbus, se actualiza la base de datos y se envían los SMS y correos electrónicos de las alarmas. La disposición de la interfaz para esta actividad se puede observar en la figura 6.6. Como se observa en la figura, en esta actividad no se introducen los parámetros, solo se puede leer sus valores, en el campo *TextView* “Parámetros”. El siguiente *TextViewes* “Inicio”, en este campo se muestra la fecha y hora en la que se inicia la medición de valores de Modbus. En el tercer campo se van a escribir

los valores de Modbus según la disposición en la unidad remota y la tabla creada en la base de datos, explicada en la sección 6.6.4. El último campo es referente al Aviso de alarma, en este campo se indica en texto de color rojo cuando una alarma ha sido enviada. Finalmente el botón “ConexButton”, que se muestra en su estado **OFF** es un botón tipo *toggle*, éste debe tocarse para pasar al estado **ON**, en el que se inicia la medición de valores de Modbus.

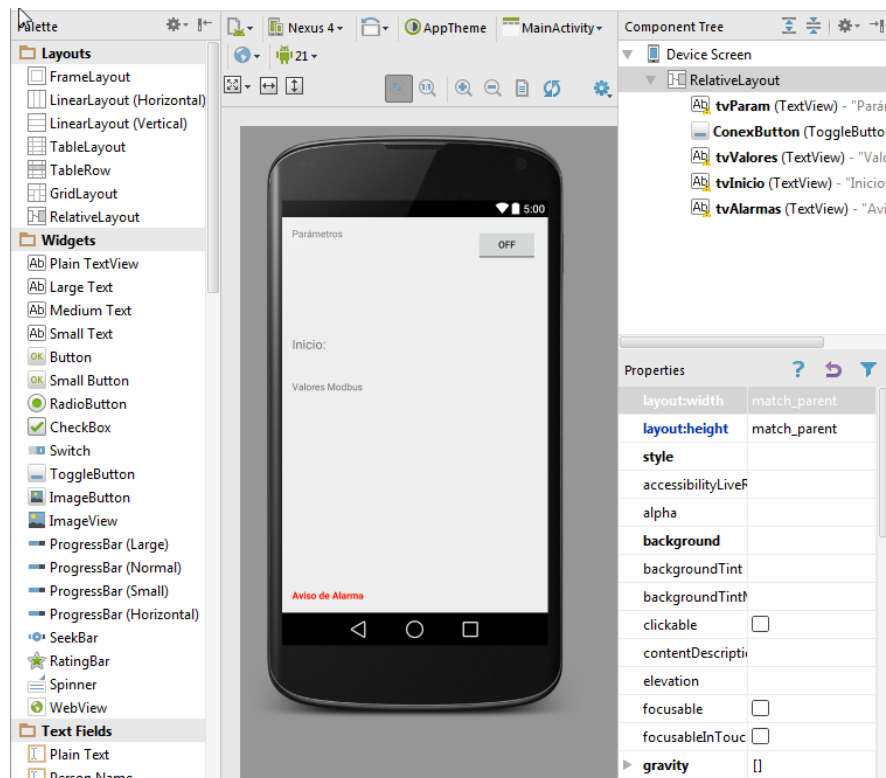


Figura 6.6. Interfaz gráfica de la actividad principal.

Siguiendo entonces con el código de la actividad, algunos segmentos de código contienen comentarios entre líneas, con la finalidad de explicar partes específicas del código, el resto de las particularidades serán explicadas en esta sección. El código completo de esta actividad estará en el anexo 1.1.

Declaración de variables globales

El primer segmento importante de la clase *MainActivity.java* viene siendo la declaración de las variables globales, el llamado a una variable de bases de datos, y la conexión entre los elementos de la interfaz con sus variables respectivas en el código.

```
public class MainActivity extends ActionBarActivity {  
    //Variable para manejo de la base de datos  
    DatabaseHandler db = new DatabaseHandler(this);  
  
    //Declaración de los TextView de la actividad  
    TextView ParamView;  
    TextView ValoresView;  
    TextView TVInicio;  
    TextView tvAlarmas;  
  
    //String globales para escribir los TextView:  
    String ParamText = "";  
    String ValoresText = "";  
  
    //String globales para las SharedPreferences:  
    String IPK;//IP  
    String PortK;//Puerto  
    String SlaveK;//Direccion del esclavo  
    String SensorK;//Punto inicial de los sensores  
    String CountK;//Numero de sensores  
    String ModbusK;//Funcion de Modbus  
    String Tdm;//Tiempo de muestreo  
  
    //Declaración del boton toggle
```

```

ToggleButton ONOFF;

//Variable global, booleana para saber si se envia una alarma
boolean AlarmSent;

//Strings que muestran la fecha y hora actual.
static Date d = new Date();
static CharSequence Fecha = android.text.format.DateFormat.format
    ("EEE, dd MMM, HH:mm:ss", d.getTime());

//Creacion del timer para realizar el ciclo de mediciones
static Timer timer = new Timer();

```

La clase de una actividad siempre va a partir, o es una extensión, de la clase madre *ActionBarActivity*. Esto hace que por defecto, todas las actividades posean las características base de una actividad para Android, incluyendo la *action bar* en el tope de cada pantalla de las aplicaciones modernas. La fecha y hora se realiza con un formato personalizado, en este caso: “EEE, dd MMM, HH:mm:ss”, donde:

EEE: Indica con tres caracteres el día de la semana.

dd: Indica con dos caracteres el día del mes.

MMM: Indica con tres caracteres el mes del año

HH: Indica con dos caracteres la hora.

mm: Indica con dos caracteres los minutos.

ss: Indica con dos caracteres los segundos.

Clase *OnCreate*

Luego de declarar las variables globales de esta actividad, se llama a la primera clase, presente en todas las actividades de Android:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    //Se relacionan las variables globales con
    //Los elementos de la interfaz grafica
    setContentView(R.layout.activity_main);
    ParamView = (TextView) findViewById(R.id.tvParam);
    ValoresView = (TextView) findViewById(R.id.tvValores);
    TVInicio = (TextView) findViewById(R.id.tvInicio);
    tvAlarmas = (TextView) findViewById(R.id.tvAlarmas);

    //Esta linea permite que los valores de modbus
    //Se puedan mediante una barra de scrolling
    //Si hay mas sensores que lineas disponibles para escritura
    ValoresView.setMovementMethod(new ScrollingMovementMethod());

    ONOFF = (ToggleButton) findViewById(R.id.ConexButton);

    //Se coloca un valor inicial para el booleano de la alarma
    AlarmSent = false;

    //Aca se llaman los valores de las SharedPreferences
    //Provenientes de otra actividad
    //Si no existen los valores, se colocan los sugeridos
    SharedPreferences MisPreferencias =
        getSharedPreferences("MisPreferencias",
            Context.MODE_PRIVATE);
```

```

        IPK = MisPreferencias.getString("IPKey", "0.0.0.0");
        PortK = MisPreferencias.getString("PortKey", "0000");
        SlaveK = MisPreferencias.getString("SlaveKey", "1");
        SensorK = MisPreferencias.getString("SensorKey", "1");
        CountK = MisPreferencias.getString("CountKey", "1");
        ModbusK = MisPreferencias.getString("FuncKey", "04");
        Tdm = MisPreferencias.getString("TiempoKey", "100");

        //Se escriben los parametros, si ya existen.
        ParamText = ParamText + "Los par\u00e1metros son:" + "\n" +
            "IP:" + IPK + "\n" +
            "Puerto:" + PortK + "\n" +
            "ID del esclavo:" + SlaveK + "\n" +
            "primera direcci\u00f3n:" + SensorK + "\n" +
            "N\u00famero de direcciones:" + CountK + "\n" +
            "Funci\u00f3n de Modbus:" + ModbusK + "\n" +
            "Tiempo de muestreo: " + Tdm + " seg";

        ParamView.setText(ParamText);
    }

```

Esta clase “OnCreate” es llamada cada vez que se inicia su actividad, y es la que trae al frente los parámetros por defecto de la misma. Si existe un estado previo de esta actividad, éste puede ser guardado en un *Bundle* y obtener el estado previo al cargar la clase.

La presencia de *@Override* en una clase indica que la clase original debe ser reemplazada por la que está escrita en esta actividad. La clase *OnCreate* existe de forma nativa en Android, y es generalmente reemplazada por los valores que desee el desarrollador en la actividad particular. Sin embargo, la línea de comando “super.onCreate(savedInstanceState);” permite traer el contenido por defecto de

la clase original y, por lo tanto, lo que se incluya en la declaración dentro de la actividad es adicional al contenido por defecto de esta clase.

Clase para el uso del botón *toggle*

El siguiente segmento es el segmento referente a iniciar la medición de valores de Modbus, dependiendo del botón *toggle* y su cambio de estado al realizar la acción de tocarlo.

```
public void onToggleClicked(View view) {
    //Si el boton se enciende, entonces se cumple la condicion
    boolean on = ((ToggleButton) view).isChecked();
    if (on) {
        //Se coloca el inicio de la medicion, con la hora
        //en el formato previamente realizado
        TVInicio.setText("Medici\u00f3n iniciada en: " + Fecha);
        //Se crea un Handler
        final Handler handler = new Handler();
        //Se inicializa el timer creado previamente
        timer = new Timer();
        //Se crea una tarea para el timer
        TimerTask doAsynchronousTask = new TimerTask() {
            //Dentro de la tarea del timer, se corre el
            //Handler iniciado previamente
            @Override
            public void run() {
                handler.post(new Runnable() {
                    @Override
                    public void run() {
                        //Aca se llama la clase de Modbus
                        //pasandole los parametros
                    }
                });
            }
        };
        timer.schedule(doAsynchronousTask, 0);
    }
}
```

```

        try {
            ModbusTask myClientTask = new
                ModbusTask
                    (IPK,
                    Integer.parseInt(PortK),
                    Integer.parseInt(SlaveK),
                    Integer.parseInt(SensorK),
                    Integer.parseInt(CountK),
                    Integer.parseInt(ModbusK));
            myClientTask.execute();
        } catch (Exception e) {
            e.printStackTrace();
            ValoresView.setText(e.toString());
        }
    }
});

}

};

//En este segmento se configura el timer
//0 ms de retardo, Cada Tdm*1000ms
timer.schedule(doAsynchronousTask, 0,
    Integer.parseInt(Tdm)*1000);
} else {
    //Si se vuelve a tocar el boton toggle
    //Se cancela el timer
    //Se reemplaza el texto de modbus
    //Se limpian los valores de alarma
    //La actividad pasa a un estado de espera por
    //presionar de nuevo el boton
    timer.cancel();
}

```

```
        ValoresView.setText("no se est\u00e1 midiendo actualmente.");
        tvAlarmas.setText("");
        AlarmSent = false;
    }
}
```

Los parámetros de Modbus son pasados dentro de un bloque *try*, este bloque permite atrapar cualquier excepción que lance en su interior, y enviando un mensaje con el error en el campo de los valores medidos, evitando que se cierre la aplicación de forma inesperada. Cuando se pasan los parámetros de Modbus, en esa clase se realizan el resto de acciones importantes de esta actividad.

Declaración de la *AsyncTask* y librerías de Modbus

En el siguiente segmento se implementará la clase que lleva todo el proceso de mensajes de Modbus, el guardado de los valores en la base de datos, la comparación con los valores de alarma y los ciclos para enviar las alarmas, tanto por SMS como por correo electrónico. Esta clase llamada *ModbusTask* es una extensión de la clase nativa de Android *AsyncTask*.

Conociendo la *AsyncTask*, se expondrá en varios segmentos para simplificar el contenido. El primer segmento incluye la declaración de la clase y el constructor para llamarla desde cualquier parte del código:

```
public class ModbusTask extends AsyncTask<Void, String, String> {
    //Variables de la clase
    String Resp;
    String IPMod;
    int PortMod;
    int slaveId;
```

```

int RefMod;
int CountMod;
int Func;

//Arreglos para manipular valores de Modbus
private int[] ModbusArray;
private int[] SensorArray;
private boolean[] ArregloModbus01;
private boolean[] ArregloModbus02;
private short[] ArregloModbus03;
private short[] ArregloModbus04;

//Constructor
ModbusTask(String addr, int port, int ID, int ref, int count, int
    func) {
    IPMod = addr;
    PortMod = port;
    slaveId = ID;
    RefMod = ref;
    CountMod = count;
    Func = func;
}

```

Teniendo el constructor y las variables globales para la clase, se puede declarar la tarea en segundo plano que realiza la *AsyncTask*

```

@Override
protected String doInBackground(Void... arg0) {

    ModbusFactory factory = new ModbusFactory();//Nueva tarea
    Modbus

```

```

IpParameters params = new IpParameters();//Se crea la
    variable que
// maneja los parametros para el maestro de modbus
params.setHost(IPMod);//Se asigna el IP del esclavo
params.setPort(PortMod);//Se asigna el puerto del esclavo
params.setEncapsulated(true);//Se asigna el valor True a la
    encapsulacion del
// mensaje dentro de una trama TCP
ModbusMaster master = factory.createTcpMaster(params, true);
    //Se crea un maestro de
// Modbus segun los parametros y se mantiene abierto con
    'keepalive = true'
master.setTimeout(500);//Se coloca un timeout de 500ms para
    enviar la trama
master.setRetries(2);//Se colocan 2 reintentos por fallo en
    la conexion

```

Luego de introducidos los parámetros, se implementan en el segmento “DoIn-Background” las funciones de Modbus (01-02-03-04) construidas con la librería *Modbus4j*:

- A partir de un bloque “switch” se elige un segmento según la función de Modbus
- Se crea un arreglo para la respuesta del esclavo, dependiendo de la función elegida
- Luego el arreglo es transformado a un arreglo genérico de tipo entero que será utilizado para guardar los valores obtenidos dentro de la base de datos

La forma general de las funciones de Modbus implementadas es la siguiente:

- Inicialización del maestro de Modbus, abriendo el socket TCP
- Se genera la petición o pregunta al esclavo.
- Según los datos recibidos se genera un mensaje de respuesta
- Si el mensaje de respuesta es un error, se arroja la excepción.
- Si el mensaje de respuesta es correcto, se genera el arreglo con los valores de Modbus que entrega el esclavo, según la función elegida.

El segmento de código, dentro de la tarea *AsyncTask* es el siguiente:

```
try {

    master.init(); //Se inicializa el maestro de ModBus
    switch (Func) {
        case 01:// Se genera la petición de la función 01
            ReadCoilsRequest request01 = new
                ReadCoilsRequest(slaveId,
                    RefMod, CountMod);
            // Se genera la respuesta de la función 01
            ReadCoilsResponse response01 =
                (ReadCoilsResponse) master.send(request01);
            if (response01.isException()) {
                Resp = "Exception response: " +
                    "message=" +
                        response01.getExceptionMessage();
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        ValoresView.setText(Resp);
                    }
                });
            }
    }
}
```

```

    } else {
        //Se guarda la respuesta en un arreglo
        ArregloModbus01 =
            response01.getBooleanData();
        //En el ciclo while se convierte
        //El arreglo en arreglo de enteros
        int i = 0;
        ModbusArray = new int[CountMod];
        while (i < CountMod) {
            ModbusArray[i] = ArregloModbus01[i] ? 1
                : 0;
            i++;
        }
    }
}

case 02:// Se genera la peticion de la funcion 02
ReadDiscreteInputsRequest request02 =
    new ReadDiscreteInputsRequest(slaveId,
        RefMod, CountMod);
// Se genera la respuesta de la funcion 02
ReadDiscreteInputsResponse response02 =
    (ReadDiscreteInputsResponse)
        master.send(request02);
if (response02.isException()) {
    Resp = "Exception response: message=" +
        response02.getExceptionMessage();
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            ValoresView.setText(Resp);
        }
    })
}

```

```

    });
} else {
    //Se guarda la respuesta en un arreglo
    ArregloModbus02 =
        response02.getBooleanData();
    //En el ciclo while se convierte
    //El arreglo en arreglo de enteros
    int j = 0;
    ModbusArray = new int[CountMod];
    while (j < CountMod) {
        ModbusArray[j] = ArregloModbus02[j] ? 1
            : 0;
        j++;
    }
}

case 03:// Se genera la peticion de la funcion 03
    ReadHoldingRegistersRequest request03 =
        new
            ReadHoldingRegistersRequest(slaveId,
                RefMod, CountMod);
    // Se genera la respuesta de la funcion 03
    ReadHoldingRegistersResponse response03 =
        (ReadHoldingRegistersResponse)
            master.send(request03);

    if (response03.isException()) {
        Resp = "Exception response: message=" +
            response03.getExceptionMessage();
        runOnUiThread(new Runnable() {
            @Override

```



```

        public void run() {
            ValoresView.setText(Resp);
        }
    });
} else {
    //Se guarda la respuesta en un arreglo
    ArregloModbus03 = response03.getShortData();
    //En el ciclo while se convierte
    //El arreglo en arreglo de enteros
    int k = 0;
    ModbusArray = new int[CountMod];
    while (k < CountMod) {
        ModbusArray[k] = (int)
            ArregloModbus03[k];
        k++;
    }
}

case 04:// Se genera la peticion de la funcion 04
ReadInputRegistersRequest request04 =
    new ReadInputRegistersRequest(slaveId,
        RefMod, CountMod);
// Se genera la respuesta de la funcion 04
ReadInputRegistersResponse response04 =
    (ReadInputRegistersResponse)
        master.send(request04);
if (response04.isException()) {
    Resp = "Exception response: message=" +
        response04.getExceptionMessage();
    runOnUiThread(new Runnable() {
        @Override

```

```

        public void run() {
            ValoresView.setText(Resp);
        }
    });
} else {
    //Se guarda la respuesta en un arreglo
    ArregloModbus04 = response04.getShortData();
    //En el ciclo while se convierte
    //El arreglo en arreglo de enteros
    int l = 0;
    ModbusArray = new int[CountMod];
    while (l < CountMod) {
        ModbusArray[l] = (int)
            ArregloModbus04[l];
        l++;
    }
}

break;
}

```

Todos los mensajes de Excepción en el segmento de código anterior están dentro de un bloque “runOnUiThread” con la finalidad de escribir en los campos de la interfaz gráfica cualquier error que ocurra durante la tarea en segundo plano. Los arreglos de la respuesta del esclavo son guardados en su variable particular dependiendo de la función:

- Los arreglos de la función 01 son guardados en un arreglo booleano
- Los arreglos de la función 02 son guardados en un arreglo booleano

- Los arreglos de la función 03 son guardados en un arreglo short
- Los arreglos de la función 04 son guardados en un arreglo short

Luego de que son guardados dependiendo de la función elegida, son llevados a un arreglo tipo entero, que permitirá guardar cualquier valor dentro de la base de datos sin importar qué tipo de variable era en un principio.

En el siguiente y último segmento de la *AsyncTask* se ejecutan las acciones restantes de esta tarea, que son las siguientes:

- Se inicializa el texto para escribir en la pantalla principal
- Se inicia un ciclo para guardar valores en la base de datos
- Dentro del ciclo iniciado, se escriben los valores en un String que permitirá actualizar la pantalla principal al finalizar el guardado de valores
- Se compara cada valor del arreglo de Modbus con los valores de alarmas, si existen
- Si hay una alarma, se escriben los mensajes de alarma, y se envía la notificación
- Finalmente se actualiza la pantalla principal, se destruye el maestro y finaliza la *AsyncTask*

La comparación se realiza extrayendo de la base de datos los valores guardados que corresponden a la RTU y el sensor que circula por el ciclo en ese momento. Si la condición de alarma se cumple entonces se extraen también los valores para enviar el correo electrónico y el SMS al destinatario asignado los siguientes mensajes de alarma:

para el SMS: ALERTA: Valor del sensor “X” fuera de rango aceptable.

para el E-mail: ALERTA: Valor del sensor “X” fuera de rango aceptable:

“Datos”

Donde “X” representa al número del sensor en cuestión y “Datos” representa todos los valores escritos en esa medición hasta el punto donde se encontró la alarma. En la sección 7.1.2 se encuentra un ejemplo de cómo es el mensaje en el caso del e-mail y en el caso del SMS enviados por una alarma. Los mensajes escritos para cada caso deben ser distintos, ya que en el caso del SMS el mensaje es limitado a 160 caracteres, y por lo tanto se envía solo la información del sensor que falla. Mientras tanto en el correo electrónico es posible enviar la información del sensor que falla y adicionalmente enviar la información del valor de los sensores que se han medido hasta el punto de la alarma.

```
//Se inicializa el texto de respuesta
ValoresText = "";
//Inicia un ciclo while para escribir los valores en la
    base de datos
int i = 0;
SensorArray = new int[Integer.parseInt(CountK)];
while (i < Integer.parseInt(CountK)) {
    SensorArray[i] = i + 1;
    Log.d("Insertar datos: ", "Insertando ..");
    //Se agregan los valores en la base de datos
    db.addValue(new ValorModbus(1, SensorArray[i],
        ModbusArray[i]));
    db.close();
    //Se escriben en un String que luego actualiza la
        pantalla principal
```

```

ValoresText = ValoresText + "ID: " + db.LastValor() +
    ";RTU: " +
        "1" + ";Sensor:" + SensorArray[i] + ";Valor: " +
        ModbusArray[i] + " \n";
//Se extraen los valores de rango de las alarmas de la
    base de daots
//Segun la remota y el sensor actuales en el ciclo
String rangomin = db.ValorminValue(1, SensorArray[i]);
String rangomax = db.ValormaxValue(1, SensorArray[i]);
//Existe una condicion para funcionar si no hay alarmas
if(!rangomin.matches("") || !rangomax.matches("")){
    int rangominint = Integer.parseInt(rangomin);
    int rangomaxint = Integer.parseInt(rangomax);
    //Si hay alarmas, comparar el sensor actual del
        ciclo
    //Con los valores de alarma
    if ((ModbusArray[i] > rangomaxint ||
        ModbusArray[i] < rangominint) &&
        !AlarmSent) {
        //Si hay una alarma y no se ha enviado una antes
        //Se extrae el correo electronico y celular del
        //Administrador, guardados en la base de datos
        String MailAlarma = db.MailValue(1,
            SensorArray[i]);
        String CelAlarma = db.CelValue(1,
            SensorArray[i]);
        //Escribir los mensajes de alarma
        String MensajeAlarmaMail = "ALERTA: Valor del
            sensor " +
                String.valueOf(SensorArray[i] +

```

```

        " fuera de rango aceptable:" + "\n"
        + ValoresText);

String MensajeAlarmaCel = "ALERTA: Valor del
    sensor " +
        String.valueOf(SensorArray[i] +
            " fuera de rango aceptable");
//Enviar SMS y correo electronico de alarma
sendSMS(CelAlarma, MensajeAlarmaCel);
sendMail(MailAlarma, MensajeAlarmaMail);
//Se coloca verdadera la variable de 'alarma
    enviada'
AlarmSent = true;
    }
}
i++;
}

runOnUiThread(new Runnable() {
    @Override
    public void run() {
        ValoresView.setText(ValoresText);
        //Si se envio una alarma, activar texto de alarmas
        if(AlarmSent){
            tvAlarmas.setText("ALARMA ENVIADA. Para
                reactivar " +
                    "las alarmas debe reiniciar la
                    medici\u00f3n");
        }
    }
});
} catch (ModbusInitException | ModbusTransportException e) {

```

```

        e.printStackTrace();
        Resp = "Error: " + e.toString();
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                ValoresView.setText(Resp);
            }
        });
    } finally {
        //Se destruye el maestro de modbus
        //Que se inicializa en el proximo ciclo
        master.destroy();
    }
    return ValoresText;
}

@Override
//Se actualiza la pantalla principal con los valores guardados
//previamente
protected void onProgressUpdate(String... values) {
    ValoresView.setText(ValoresText);
}
}

```

Los mensajes de alarma y cómo funcionan de forma detallada serán explicados en la sección 6.6.3, donde se aborda la configuración de las alarmas.

Métodos para enviar SMS y correo electrónico

Las líneas de comando utilizado en el segmento anterior son solo llamados a los métodos que tiene la clase para enviar un SMS y/o correo electrónico, para el caso

de un mensaje de texto, se utiliza la librería *Android.telephony.SmsManager*, nativa del sistema operativo y que permite el uso de mensajes de texto directamente desde código, sin la interacción del usuario.

El segmento de la clase implementado realiza varios pasos para enviar un mensaje de texto y recibir la confirmación, tal como lo hacen las aplicaciones actuales de mensajería. El envío de mensajes de texto se basa principalmente en dos procedimientos, el uso de *Broadcast receivers*, explicados en la sección 3.1, y el uso de *Intents*. Un Intent, según Android developers, es un procedimiento utilizado para comunicarse con otros servicios que hacen vida dentro del dispositivo Android, ya sea mediante *Broadcast Receivers* u otras actividades, los *Intent* pueden llamar procesos que ya están corriendo, o iniciar procesos para utilizarlos en su contexto de aplicación. Comprendiendo esto, el procedimiento para enviar mensajes de texto dentro de la aplicación ocurre de la siguiente forma:

- Se declara una clase interna al envío de mensajes de texto para el momento en el cual el mensaje es enviado con el *Broadcast Receiver* y los resultados que arroja, generando respuestas para los diferentes casos que se pueden presentar, incluyendo envío exitoso del mensaje o errores de servicio.
- Existe otra clase del mismo tipo, realizada para cuando el mensaje, además de enviado, fue entregado al destinatario, lo que permite saber si el mensaje fue enviado con éxito o no.
- Finalmente, con el contenido del número telefónico, y el mensaje incluido, se envía el mensaje de acuerdo a las condiciones y las clases creadas (enviado y entregado)

El código de la clase para enviar SMS es el siguiente:

```
//Enviar SMS a destinatario
```



```

public void sendSMS(String phoneNumber, String message)
{
    //Strings para el broadcast receiver
    String SENT = "SMS_SENT";
    String DELIVERED = "SMS_DELIVERED";
    //Creacion de intents para mensaje enviado y entregado
    //sin flags, ni codigos particulares de envio
    PendingIntent sentPI = PendingIntent.getBroadcast(this, 0,
        new Intent(SENT), 0);

    PendingIntent deliveredPI = PendingIntent.getBroadcast(this, 0,
        new Intent(DELIVERED), 0);

    //Luego de enviar el mensaje.
    //Respuestas a los diferentes casos enviados
    registerReceiver(new BroadcastReceiver() {
        @Override
        public void onReceive(Context arg0, Intent arg1) {
            switch (getResultCode()) {
                case Activity.RESULT_OK:
                    Toast.makeText(getBaseContext(),
                        "Mensaje de alarma enviado.",
                        Toast.LENGTH_SHORT).show();
                    break;
                case SmsManager.RESULT_ERROR_GENERIC_FAILURE:
                    Toast.makeText(getBaseContext(),
                        "Falla generica al enviar SMS",
                        Toast.LENGTH_SHORT).show();
                    break;
                case SmsManager.RESULT_ERROR_NO_SERVICE:

```

```

        Toast.makeText(getBaseContext(),
                        "Sin servicio, sms no enviado.",
                        Toast.LENGTH_SHORT).show();

        break;

    case SmsManager.RESULT_ERROR_NULL_PDU:
        Toast.makeText(getBaseContext(),
                        "Mensaje no enviado, receptor no
                        disponible",
                        Toast.LENGTH_SHORT).show();

        break;

    case SmsManager.RESULT_ERROR_RADIO_OFF:
        Toast.makeText(getBaseContext(),
                        "Antena no disponible, mensaje no enviado.",
                        Toast.LENGTH_SHORT).show();

        break;

    }

}

}, new IntentFilter(SENT));

//Cuando el mensaje fue entregado
//Respuestas al caso de mensaje entregado
registerReceiver(new BroadcastReceiver() {

    @Override

    public void onReceive(Context arg0, Intent arg1) {

        switch (getResultCode()) {

            case Activity.RESULT_OK:

                Toast.makeText(getBaseContext(),
                                "Mensaje de alarma entregado",
                                Toast.LENGTH_SHORT).show();

                break;

        }

    }

});

```

```

        case Activity.RESULT_CANCELED:
            Toast.makeText(getBaseContext(),
                "Mensaje de alarma enviado, pero no
                entregado",
                Toast.LENGTH_SHORT).show();
            break;
        }
    }
}, new IntentFilter(DELIVERED));

//Envio del SMS, a partir del contenido del mensaje
//y el numero celular.
SmsManager sms = SmsManager.getDefault();
sms.sendTextMessage(phoneNumber, null, message, sentPI,
    deliveredPI);
}

```

Luego, el llamado a la clase para enviar correos electrónicos, es más complicado y por lo tanto tiene una clase entera en su archivo particular, que será expuesto en el anexo 1.9. El código para llamar la clase mencionada es el siguiente:

```

//Se introducen como parametros el mensaje a enviar
//Y el correo electrónico del destinatario
public void sendMail(String mail, String message){
    String Mail = mail;
    String Message = message;
    //Es necesario introducir el correo y clave
    //Desde donde se envía el correo
    Mail m = new Mail("alarmasmodbus@gmail.com", "AlarmaModbus");
    String[] toArr = {Mail};
}

```

```

        //Valores para enviar el correo
        //Destinatario, asunto y envio
        m.setTo(toArr);
        m.setFrom("alarmasmodbus@gmail.com");
        m.setSubject("Alerta");
        m.setBody(Message);
        try {
            //Se envia el correo en un bloque de excepcion
            m.send();
        } catch (Exception e) {
            Log.e("MailApp", "No se pudo enviar el E-mail", e);
        }
    }
}

```

Los dos segmentos restantes de código en el archivo *MainActivity.java* son colocados con la creación de la actividad en el ambiente de desarrollo, estas dos clases se encargan de controlar el menú que tiene la *Action Bar* en el tope de la pantalla, desde ese menú se llama al resto de las actividades en la aplicación, y se transita a través de ellas por toda la aplicación. El código introducido por el ambiente de desarrollo es el siguiente:

```

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is
    // present.
    getMenuInflater().inflate(R.menu.menu_main, menu);
    return true;
}

@Override

```

```

public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();

    //noinspection SimplifiableIfStatement
    //Aca se llama a la actividad SettingsActivity
    if (id == R.id.action_settings) {
        Intent i = new Intent(this, SettingsActivity.class);
        ONOFF.setChecked(false);
        timer.cancel();
        ValoresView.setText("no se est\u00e1 midiendo actualmente.");
        startActivity(i);
        return true;
    }

    //Aca se llama a la actividad AlarmConfig
    if (id == R.id.alarm_settings) {
        Intent i = new Intent(this, AlarmConfig.class);
        ONOFF.setChecked(false);
        timer.cancel();
        ValoresView.setText("no se est\u00e1 midiendo actualmente.");
        startActivity(i);
        return true;
    }

    return super.onOptionsItemSelected(item);
}

```

Con este segmento de código finaliza la actividad *MainActivity.java*, que viene

siendo la actividad más compleja de toda la aplicación, en el anexo 1.2 se encuentra el código completo, incluyendo el código XML de la interfaz gráfica.

6.6.2. *SettingsActivity.java*

En esta segunda actividad, la actividad de configuración, se introducen los parámetros de comunicación para Modbus, además se incluye el tiempo de muestreo con el que se va a realizar el envío y recepción de mensajes. Adicionalmente se implementó un botón para borrar los valores de las mediciones que se guardan en la base de datos, con fines de prueba. La interfaz para esta actividad se presenta en la figura 6.7.

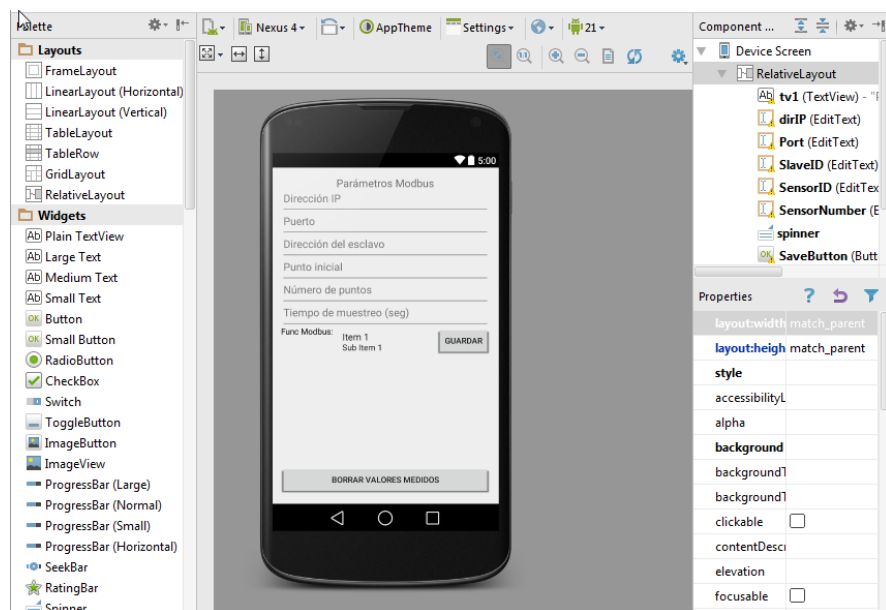


Figura 6.7. Interfaz gráfica de la actividad de configuración.

La actividad incluye 6 *TextFields* para introducir los datos, un *TextView* para indicar dónde se introduce la función de Modbus, un *Spinner*, donde se selecciona entre las funciones 01-02-03-04. Un botón de guardar que al ser tocado lleva a la actividad principal, invitando al usuario a iniciar las mediciones. Adicionalmente se implementó el botón de borrado de la base de datos mencionado anteriormente.

Para acceder a esta actividad, se parte de la actividad principal de la aplicación, y en las opciones de menú (botón menú del dispositivo) o mediante un toque en las opciones de la *ActionBar*, se despliega el menú inflable que puede llevar a la actividad de configuración o *Settings*, dependiendo del idioma del dispositivo.

Declaración de variables globales

Al igual que en la actividad anterior, en esta actividad se declaran ciertas variables globales, las primeras, para relacionar los botones y entradas de texto de la interfaz gráfica con variables en la actividad como tal. Además de las variables globales, se declaran las variables que forman parte de las *SharedPreferences*. Yendo al caso particular de éstas variables, se declaran las variables cuyos valores son introducidos en esta actividad (Todos los parámetros de Modbus y el tiempo de muestreo), estas variables se guardan con un nombre particular y un título para todas ellas, luego en segmentos siguientes del código, la clase *SharedPreferences* es llamada , guardando los valores de las variables introducidas bajo el título “Mis-Preferencias” y llevándolas a otras actividades (En este caso *MainActivity.java*) guardándolas a través del tiempo sin necesidad de una base de datos. El uso de las *SharedPreferences* para valores menos críticos que mediciones importantes simplifica el desarrollo del código, y extiende el uso de las herramientas proporcionadas por Android. El código de la declaración de variables globales y de *SharedPreferences* es el siguiente:

```
public class SettingsActivity extends ActionBarActivity {  
  
    DatabaseHandler db = new DatabaseHandler(this);  
  
    //Se declaran los EditText  
    EditText DirIP;  
    EditText Port;
```

```

EditText SlaveID;
EditText SensorID;
EditText SensorNumber;
EditText Tdm;

//Se declara el Spinner de la funcion de modbus
Spinner FuncModbus;

//Se declaran los botones
Button SaveButton;
Button DeleteAllButton;

//Se declaran las Shared Preferences
SharedPreferences sharedPreferences;

public static final String MisPreferencias = "MisPreferencias";
public static final String IPK = "IPKey";
public static final String PortK = "PortKey";
public static final String SlaveK = "SlaveKey";
public static final String SensorK = "SensorKey";
public static final String CountK = "CountKey";
public static final String ModbusK= "FuncKey";
public static final String TiempoK = "TiempoKey";

```

Clase *OnCreate*

La clase *OnCreate* de esta actividad es más sencilla, conteniendo la relación directa entre la interfaz y la clase, como detalle en esta clase, está el desarrollo del *Spinner*, mediante un arreglo de String que contiene los números de las funciones de Modbus implementadas. El código que implementa esta clase:

```

@Override
protected void onCreate(Bundle savedInstanceState) {

```



```

super.onCreate(savedInstanceState);

setContentView(R.layout.activity_settings);

//Se asignan los EditText a variables de la activity
DirIP = (EditText) findViewById(R.id.dirIP);
Port = (EditText) findViewById(R.id.Port);
SlaveID = (EditText) findViewById(R.id.SlaveID);
SensorID = (EditText) findViewById(R.id.SensorID);
SensorNumber = (EditText) findViewById(R.id.SensorNumber);

Tdm = (EditText) findViewById(R.id.Tdm);

SaveButton = (Button) findViewById(R.id.SaveButton);
DeleteAllButton = (Button) findViewById(R.id.DeleteAllButton);

//Se inicializa el spinner
FuncModbus = (Spinner) findViewById(R.id.spinner);
String []opciones={"01", "02", "03", "04"};
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
    android.R.layout.simple_spinner_item, opciones);
FuncModbus.setAdapter(adapter);

//Funcion del boton save
SaveButton.setOnClickListener(buttonSaveOnClickListener);

DeleteAllButton.setOnClickListener(buttonDeleteOnClickListener);

//Se inicializan las preferencias
//El nombre de las preferencias
//El contexto privado limita las preferencias a esta app

```

```
        sharedPreferences = getSharedPreferences(MisPreferencias,
            Context.MODE_PRIVATE);
    }
```

Clase *OnClickListener*

Esta clase es implementada a partir del uso del botón guardar, es presentada para guardar las variables introducidas dentro de las *SharedPreferences*. La única particularidad en el desarrollo de esta clase es la presencia de un condicional para verificar si se introduce algún valor vacío o no válido dentro de los campos presentados, esto evitará errores en la aplicación, por ejemplo si el ID del esclavo es colocado "0", no se podrán introducir los parámetros, ya que el esclavo debe ser propiamente identificado. El código para esta clase es el siguiente:

```
View.OnClickListener buttonSaveOnClickListener = new
    View.OnClickListener() {

    @Override
    public void onClick(View arg0){
        //Se verifica la validez de los parametros
        if (DirIP.getText().toString().matches("") ||
            SensorNumber.getText().toString().matches("") ||
            SlaveID.getText().toString().matches("") ||
            Port.getText().toString().matches("") ||
            SensorID.getText().toString().matches("") ||
            SensorNumber.getText().toString().matches("0") ||
            Tdm.getText().toString().matches("0") ||
            Tdm.getText().toString().matches("") ||
            SlaveID.getText().toString().matches("0")) {
            //Mensaje en pantalla si los parametros
```

```

        //Son invalidos
        Toast.makeText(getApplicationContext(), "Existen par\u00e1metros
        inv\u00e1lidos", Toast.LENGTH_SHORT).show();
    }
    else {
        //Se guardan los Strings en las preferencias.
        SharedPreferences.Editor editor =
            sharedPreferences.edit();
        editor.putString(IPK, DirIP.getText().toString());
        editor.putString(PortK, Port.getText().toString());
        editor.putString(SlaveK, SlaveID.getText().toString());
        editor.putString(SensorK, SensorID.getText().toString());
        editor.putString(CountK,
            SensorNumber.getText().toString());
        editor.putString(TiempoK, Tdm.getText().toString());
        editor.putString(ModbusK,
            FuncModbus.getSelectedItem().toString());
        editor.apply();
        //Se llama a la main activity
        Intent i = new Intent(getApplicationContext(),
            MainActivity.class);
        onNewIntent(i);
        startActivity(i);
    }
}
};

```

El código completo de esta clase se presenta en el anexo 1.3 donde se encuentran las clases de *ActionBar* y el botón de borrado de la base de datos, que contienen código sumamente sencillo.

6.6.3. *AlarmConfig.java*

La tercera y última actividad de la aplicación está relacionada a la configuración y manejo de las alarmas en la aplicación. En esta actividad se introducen las alarmas en una tabla de la base de datos, con la finalidad de manejarlas y relacionarlas a la otra tabla presente en la base de datos, donde se guardan los valores de Modbus.

Para el caso particular de este proyecto, las alarmas funcionan de la siguiente forma:

- Se puede introducir una alarma por cada sensor de todo el circuito
- Se puede monitorear un valor mínimo del cual no se desea bajar en las mediciones
- Se puede monitorear un valor máximo del cual no se desea subir en las mediciones
- La alarma será enviada tanto al correo electrónico como al número celular (SMS) del administrador colocado en la alarma

El mensaje de alarma es configurado en la actividad principal de la aplicación, sin embargo, el resto de los parámetros es introducido en esta actividad, mediante la interfaz presentada en la figura 6.8

Para acceder a esta actividad, se parte de la actividad principal de la aplicación, y en las opciones de menú (botón menú del dispositivo) o mediante un toque en las opciones de la *ActionBar*, se despliega el menú inflable que puede llevar a la actividad de Config. de alarmas.

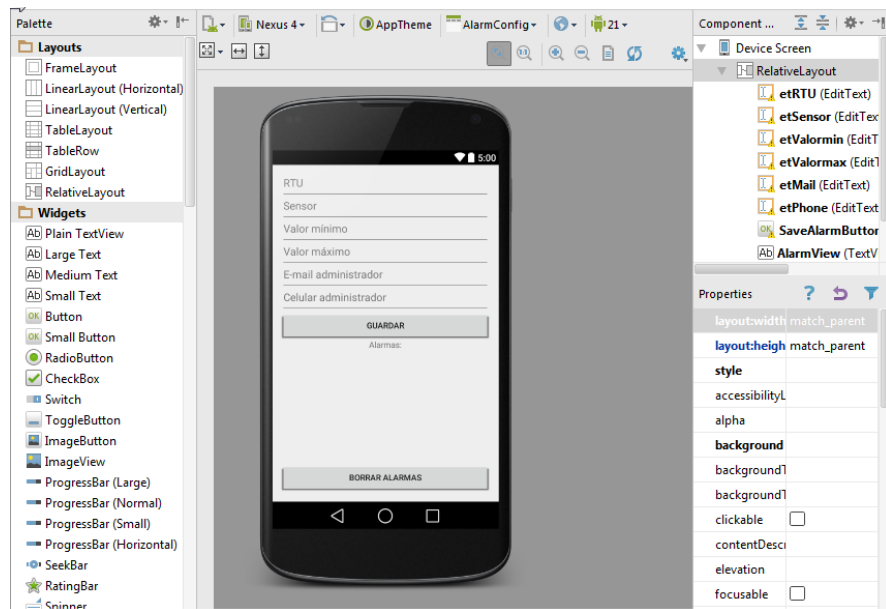


Figura 6.8. Interfaz gráfica de la actividad de configuración de alarmas.

Esta actividad posee 6 campos de *EditText*, dos campos de *TextView* y dos botones, uno para guardar las alarmas y otro para borrarlas.

Los valores que introduce el usuario para guardar una alarma son los siguientes:

RTU: El número de la unidad remota a la que se refiere la alarma

Sensor: El número de sensor que se va a monitorear

Valor mínimo: El mínimo valor al que puede llegar el sensor antes de enviar una alarma

Valor máximo: El máximo valor al que puede llegar el sensor antes de enviar una alarma

E-mail administrador: correo electrónico al que se le enviará el mensaje de alarma

Celular administrador: Número celular al que se le enviará el mensaje de alarma

El mensaje de alarma puede ser editado en la actividad principal, dentro del ciclo donde se envían los mensajes y se realizan las comparaciones de un sensor con sus valores límite.

Luego de que el usuario introduce los valores y presiona guardar, se mostrará la alarma guardada con todos los valores introducidos, y cada vez que guarde una nueva alarma, se mostrará debajo de la anterior. Es posible guardar una alarma para cada sensor en el sistema.

Mediante el botón “BORRAR ALARMAS” el usuario puede borrar todas las alarmas introducidas.

Declaración de variables globales

Las variables globales en esta actividad son reducidas en comparación a las otras dos actividades, solo se incluyen las variables referentes a la interfaz gráfica de la actividad, la unidad remota, los sensores, valores límites de una alarma y la variable para manejar la base de datos, el segmento de variables globales se presenta en el anexo 1.5.

Clase *OnCreate*

La clase *OnCreate* solo incluye la relación de las variables de la actividad con los campos de la interfaz gráfica. El código se presenta en el anexo 1.5.

SaveAlarmOnClickListener

Esta clase es la que se encarga de guardar los valores de alarma introducidos por el usuario al presionar el botón de “GUARDAR”, del mismo modo en el que se guardan los parámetros de Modbus, esta clase contiene una condición que revisa si se intentan introducir parámetros en blanco, acción que no es posible dentro de la base de datos. Luego de la verificación, se guardan los valores escritos dentro de una tabla en la base de datos, y posteriormente se extraen y son escritos en un *TextView*, comprobando que se guardaron de forma exitosa. Las clases y métodos para manejar la base de datos serán explicados en una sección posterior. El código de la clase es el siguiente:

```
View.OnClickListener SaveAlarmOnClickListener = new
    View.OnClickListener() {
        @Override
        public void onClick(View arg0){
            String alarm = "";
            //Condicion que revisa la validez de los parametros
            if (etRTU.getText().toString().matches("") ||
                etSensor.getText().toString().matches("") ||
                etValorminimo.getText().toString().matches("") ||
                etValormaximo.getText().toString().matches("") ||
                etMail.getText().toString().matches("") ||
                etPhone.getText().toString().matches("")) {
                //Mensaje que indica si se introdujeron
                //Parametros errados
                Toast.makeText(getApplicationContext(), "Existen par\u00e1metros
                    inv\u00e1lidos", Toast.LENGTH_SHORT).show();
            }
        }
    }
```

```

else {
    //las variables se pasan a numeros enteros
    //o texto.

    intRTU = Integer.parseInt(etRTU.getText().toString());
    intSensor =
        Integer.parseInt(etSensor.getText().toString());
    intValormin =
        Integer.parseInt(etValorminimo.getText().toString());
    intValormax =
        Integer.parseInt(etValormaximo.getText().toString());
    strMail = etMail.getText().toString();
    strPhone = etPhone.getText().toString();
    // se llama el metodo para guardar valores de alarma
    db.addAlarma(new AlarmasModbus(intRTU,
        intSensor,
        intValormin,
        intValormax,
        strMail,
        strPhone));

    //Se verifican en el log del
    //ambiente de desarrollo

    List<AlarmasModbus> Alarmas = db.getAllAlarmas();

    for (AlarmasModbus cn : Alarmas) {
        alarm = alarm + "ID:" + cn.getID() + " ;RTU:" +
            cn.getRTU()
            + " ;Sensor:" + cn.getSensor() + " ;Valormin:"
            + cn.getValormin()
            + " ;Valormax:" + cn.getValormax() + " ;Mail:"
            + cn.getMail()
    }
}

```



```

        + " ;Cel:" + cn.getCel() + "\n";
        //Se escriben los valores en el TextView
        AlarmView.setText(alarm);
    }
}
};

```

DeleteAlarmsOnClickListener

En esta clase se borran los valores de alarma introducidos por la clase anterior, ante cualquier error o intención de quitar las alarmas, se puede utilizar este botón, borrando las alarmas de la base de datos. El código es el siguiente:

```

View.OnClickListener DeleteAlarmsOnClickListener = new
    View.OnClickListener() {

        @Override
        public void onClick(View arg0){
            //Borrado de alarmas
            db.clearAlarms();
            //Verificacion del borrado en el ambiente de desarrollo
            List<AlarmasModbus> Alarmas = db.getAllAlarmas();

            for (AlarmasModbus cn : Alarmas) {
                String alarm = "Id: " + cn.getID() + " ,RTU: " +
                    cn.getRTU()
                    + " ,Sensor:" + cn.getSensor() + " ,Valormin:" +
                    cn.getValormin()
                    + " ,Valormax:" + cn.getValormax() + " ,Mail:" +

```

```

        cn.getMail()
        + " ,Cel:" + cn.getCel();
    AlarmView.setText("");
    //Mensaje de borrado exitoso
    Toast.makeText(getBaseContext(), "Alarmas borradas",
        Toast.LENGTH_SHORT).show();
}
}
};

```

El código completo de la clase *AlarmConfig.java* se encuentra en el anexo 1.5.

6.6.4. Creación y manejo de la base de datos

La implementación de la base de datos es relativamente sencilla, requiriendo solo organización en su desarrollo. Para la aplicación se utilizan dos tablas, una tabla para guardar las mediciones de Modbus y otra para guardar los valores de alarma.

Estructura de la tabla para valores de Modbus

La tabla de valores medidos tiene una estructura fija, creada específicamente para esta aplicación, y que tiene como finalidad adicional a almacenar los valores de forma organizada y comprensible. La estructura creada para la tabla de valores medidos se presenta en la tabla ejemplo 6.1.

La estructura de la tabla 6.1 es la siguiente:

- el campo ID representa su identificación en la base de datos, cada vez que se inserte una línea, se inserta un nuevo ID para cada valor.

Tabla 6.1. Ejemplo de la estructura de la tabla para valores medidos.

ID	RTU	Sensor	Valor
1	1	1	50
2	1	2	35
3	1	3	60
4	2	1	100
5	2	2	40
6	2	3	90
7	2	4	20
...

- El campo RTU representa la unidad remota donde se mide un valor, en el proyecto solo se implementó una unidad remota.
- El sensor representa el punto de la unidad remota donde se toma la medición
- El campo Valor representa la medición como tal, de un punto que pertenece a una remota particular.

Con la estructura anterior la tabla que se presenta es sencilla y se puede utilizar como plantilla para la escritura en pantalla de los datos a medida que se realizan las mediciones. En la clase que maneja la base de datos, esta tabla tiene el nombre de “valores”

Estructura de la tabla para alarmas

A partir de la tabla para medir valores de Modbus se crea una tabla para guardar los valores de alarma. La idea de esta tabla es que tenga valores en común con la tabla de valores, con la finalidad de que se puedan relacionar lo más posible, y sea fácil filtrar los valores requeridos mediante la RTU y el sensor de una medición. La estructura de la tabla de alarmas se presenta en la tabla 6.2

Como se observa en la tabla 6.2, se puede colocar un rango particular para cada

Tabla 6.2. Ejemplo de la estructura de la tabla para valores de alarma.

ID	RTU	Sensor	Valormin	Valormax	Mail	Cel
1	1	1	0	100	admin@dominio.com	04141234567
2	1	2	20	50	admin@dominio.com	04141234567
3	1	3	60	120	admin@dominio.com	04141234567
4	2	1	35	50	admin@dominio.com	04141234567
5	2	2	35	50	admin@dominio.com	04141234567
6	2	3	35	50	admin@dominio.com	04141234567
7	2	4	35	50	admin@dominio.com	04141234567
...

sensor en el sistema, y los valores de RTU y sensor se ubican en ambas tablas, lo que permite asignar entonces a un valor medido de Modbus, un rango (Valormin-Valormax), y los datos de un administrador que pueda recibir una eventual alarma. Se pueden colocar correos y números telefónicos particulares para cada sensor. Del mismo modo, cada alarma agregada tiene su propio identificador en la tabla. El nombre de esta tabla en el archivo que maneja la base de datos es “alarmas”

Creación de las clases *ValorModbus.java* y *AlarmasModbus.java*

Las clases presentadas en esta sección son realizadas para construir y asignar valores a las dos tablas de la base de datos, mediante estas clases, se pueden crear objetos para introducir en una tabla, llamar valores que existen en la tabla, borrar valores, etc.

La clase para crear objetos que van a la tabla “valores” es la siguiente:

```
package com.example.josempd.modbusmaster;

/**
 * Created by Josempd on 15/04/2015.
 */
public class ValorModbus {
```

```

//variables privadas
int _id;
int _RTU;
int _idSensor;
int _valor;

// Constructor vacio
public ValorModbus(){

}

// constructor
public ValorModbus(int id, int RTU, int sensor, int valor){
    this._id = id;
    this._RTU = RTU;
    this._idSensor = sensor;
    this._valor = valor;
}

// constructor para valores sin ID
public ValorModbus(int RTU, int sensor, int valor){
    this._RTU = RTU;
    this._idSensor = sensor;
    this._valor = valor;
}

// obteniendo ID
public int getID(){
    return this._id;
}

```

```

// colocando ID
public void setID(int id){
    this._id = id;
}

// obteniendo RTU
public int getRTU(){
    return this._RTU;
}

// colocando RTU
public void setRTU(int RTU){
    this._RTU = RTU;
}

// obteniendo Sensor
public int getSensor(){
    return this._idSensor;
}

// colocando Sensor
public void setSensor(int sensor){
    this._idSensor = sensor;
}

// obteniendo valor
public int getValor(){
    return this._valor;
}

```

```
// colocando valor
public void setValor(int valor){
    this._valor = valor;
}
}
```

La clase para construir alarmas tiene exactamente la misma estructura, con campos adicionales para el rango y los datos del administrador. Se encuentra disponible en el anexo 1.8 bajo el nombre de *AlarmasModbus.java*.

Creación de la clase *DatabaseHandler.java*

Según el tutorial realizado en AndroidHive por Ravi Tamada (2011), lo más conveniente para el manejo de la base de datos es una clase particular, donde se encuentran las sub clases *OnCreate* y *OnUpgrade*, para crear y renovar la base de datos. En el marco de esta aplicación, nunca se utiliza un *Upgrade* de la base de datos, en cualquier caso, se borra la base de datos actual y se genera una nueva con la misma estructura.

La creación de las columnas de una tabla, y la creación de la base de datos, están comentados en el archivo *DatabaseHandler.java*. presentado en segmentos a continuación:

```
/**
 * Created by Josempd on 15/04/2015.
 */
public class DatabaseHandler extends SQLiteOpenHelper {
```

```

// Todas las variables Static
// Version de la base de datos
private static final int DATABASE_VERSION = 1;

// Nombre de la base de datos
private static final String DATABASE_NAME = "MonitoreoModbus";

// Nombre de las tablas
private static final String TABLE_VALORES = "valores";
private static final String TABLE_ALARMAS = "alarmas";

// Nombre de las columnas comunes entre tablas
private static final String KEY_ID = "id";
private static final String KEY_RTU = "RTU";
private static final String KEY_SENSOR = "Sensor";

//Columnas exclusivas de la tabla Valores
private static final String KEY_VALOR = "Valor";

//Columnas exclusivas de la tabla Alarmas
private static final String KEY_VALORMIN = "Valormin";
private static final String KEY_VALORMAX = "Valormax";
private static final String KEY_MAIL = "Mail";
private static final String KEY_CEL = "Cel";

// Creando las tablas
//Creando la tabla de valores
private static final String CREATE_TABLE_VALORES = "CREATE TABLE "
    + TABLE_VALORES + "(" + KEY_ID + " INTEGER PRIMARY KEY," +
    KEY_RTU + " INTEGER," + KEY_SENSOR + " INTEGER," + KEY_VALOR +

```



```

        " INTEGER" + ")";

//Creando la tabla de alarmas
private static final String CREATE_TABLE_ALARMAS = "CREATE TABLE "
    + TABLE_ALARMAS + "(" + KEY_ID + " INTEGER PRIMARY KEY," +
    KEY_RTU + " INTEGER," + KEY_SENSOR + " INTEGER," +
    KEY_VALORMIN +
    " INTEGER," + KEY_VALORMAX + " INTEGER," + KEY_MAIL + "
    TEXT," +
    KEY_CEL + " TEXT" + ")";

//Constructor de la clase
public DatabaseHandler(Context context) {
    super(context, DATABASE_NAME, null, DATABASE_VERSION);
}

//Crear tablas con las string declaradas previamente
public void onCreate(SQLiteDatabase db){
    db.execSQL(CREATE_TABLE_VALORES);
    db.execSQL(CREATE_TABLE_ALARMAS);
}

```

Se crean ambas tablas utilizando *Strings* que luego son pasados como un comando al método *execSQL*. Este método permite utilizar *Strings* de texto para pasar comandos a las librerías de SQLite en Android, por lo tanto se torna sencillo la implementación de una base de datos en el sistema operativo.

Es necesario crear el nombre de las tablas, las columnas que comparten, columnas particulares y un *String* que contenga todo el comando para crear la base de datos, utilizando los valores creados previamente.

Posterior a la creación de las tablas, en la clase se presentan las operaciones *CRUD* (crear, leer, actualizar, borrar) de la base de datos. La mayoría de estos métodos están basadas en el uso de los constructores de *ValorModbus.java* y *AlarmasModbus.java*, con la finalidad de manejar y manipular los valores que se utilizan.

En el código del archivo *databaseHandler.java* presentan las siguientes operaciones de base de datos:

- Agregar un valor a la tabla “valores” mediante el uso de *ContentValues*, clase nativa de Android utilizada para guardar grupos pequeños de valores o campos.
- Obtener el último valor introducido en la tabla “valores” utilizando un cursor para moverse dentro de la tabla.
- Obtener los valores deseados de la tabla “alarmas” (Valormin, Valormax, Mail, Cel) mediante un cursor que filtra la tabla utilizando el valor de la RTU y el sensor de la tabla “valores”.
- Obtener todos los valores de la tabla “valores” para usos de revisión de la base de datos
- Contar la cantidad de valroes obtenidos, para revisión de la base de datos.
- Agregar un valor en la tabla “alarmas” del mismo modo que para la tabla “valores”, mediante *ContentValues*.
- Borrar todos los valores de la tabla “alarmas”
- Borrar todos los valores de la tabla “valores”
- Escribir todos los valores de la tabla “alarmas”

Entonces, a nivel de código, las operaciones *CRUD* se presentan comentadas en el anexo 1.8.

Con el manejo de la base de datos presentado, se cubre todo el contenido de la aplicación realizada, basada en el diagrama de la figura 6.1 con total funcionalidad.

CAPÍTULO VII

RESULTADOS Y PRUEBAS

Con el código implementado en el capítulo anterior, se obtiene una aplicación funcional de acuerdo a los diagramas de las figuras 5.4 y 6.1.

Un factor importante dentro del proyecto es que, a pesar de que los objetivos comprendidos en el marco de este trabajo se consideran cumplidos, lo que se tiene finalmente es una base sólida para continuar desarrollando y mejorando la aplicación. Es un trabajo que a partir del punto actual puede llevarse a otros puntos específicos deseados, y no es un proyecto con límites estrictos fijados a largo plazo. En estos términos, la aplicación se puede considerar como una potencial herramienta a futuro que puede contribuir con su funcionamiento en diferentes aspectos, conocidos y por plantear, dentro de algún proyecto en el campo laboral.

La aplicación presentada representa un funcionamiento genérico del protocolo en un dispositivo móvil con fines académicos dentro del marco de este proyecto, sin embargo, las posibilidades de configuración presentes a nivel de código prestan la versatilidad nombrada anteriormente. Como aspectos concretos del punto anterior, se pueden tomar en cuenta algunas consideraciones a agregar en la aplicación basadas en el proyecto presente:

- Uso de unidades en la presentación de mediciones y envío de alarmas.
- Implementación de un ciclo que recorra una serie de unidades remotas en

una ubicación específica.

- Implementación de diferentes niveles de alarma según exigencias de las mediciones.
- Cambio en el formato de presentación de datos o mensajes de alarma, tanto SMS como correos electrónicos.

Los puntos anteriores se pueden implementar para casos particulares sin mayor complicación con bases en el proyecto actual y en vía de la particularización del código dependiendo de las necesidades de un proyecto real.

A lo largo de este capítulo se realizarán algunas pruebas a la aplicación desarrollada, con la finalidad de verificar su funcionamiento y explorar las posibilidades de uso en toda la extensión del código.

7.1. Pruebas realizadas

La mayoría de las pruebas a realizar de la aplicación se basan en el uso de *Modbus Slave* para simular la presencia de un esclavo en una red local. Con la presencia de un esclavo basado en Modbus RTU encapsulado en TCP/IP se pueden realizar todas las pruebas a la aplicación, con la finalidad de comprobar su adecuado funcionamiento.

Las pruebas que se pretenden realizar en este capítulo son:

- Comprobar la aplicación bajo las 4 funciones de Modbus configuradas.
- Envío de una alarma cuando existe un valor fuera de un rango establecido.
- Funcionamiento de la aplicación ante situaciones no convencionales: Esclavo desconectado o función errada.

- Reiniciar el funcionamiento de la aplicación luego de haberse enviado una alarma
- Funcionamiento de la aplicación por un periodo de tiempo extenso.

7.1.1. Pruebas sobre las funciones de Modbus

Se planteará para cada caso un esclavo de Modbus, como lo permite el proyecto, con un máximo de 15 puntos a medir en esta unidad. Esta prueba se repetirá para cada función de Modbus implementada. En la figura 7.1 Se observa la configuración de una conexión RTU sobre TCP/IP, en el puerto 1337.

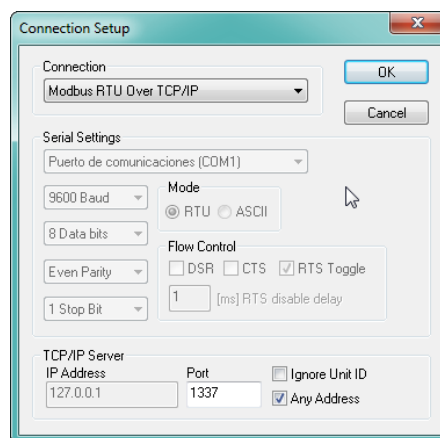


Figura 7.1. Configuración de conexión de *Modbus Slave*.

Prueba para la función 01: *Read coil status*

Luego de tener la conexión del esclavo configurada, se configuran los parámetros particulares del esclavo:

Slave ID: Es el ID del esclavo, en este caso 1.

Function: Función de Modbus a probar, en este caso 01.

Address: Es el primer punto del esclavo conectado.

Quantity: El número de punto o sensores en el esclavo.

Se puede observar la configuración del programa en la figura 7.2.

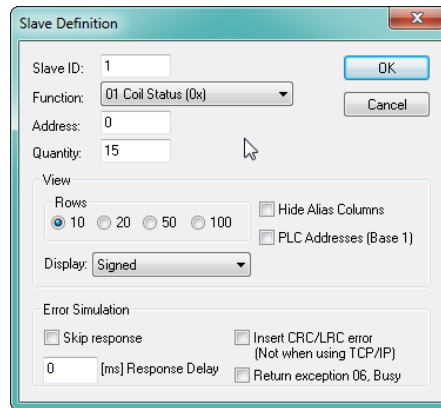


Figura 7.2. Configuración de *Modbus Slave*.

Finalmente, con el esclavo configurado, se colocan algunos valores para cada punto a medir, tal como en la figura 7.3.

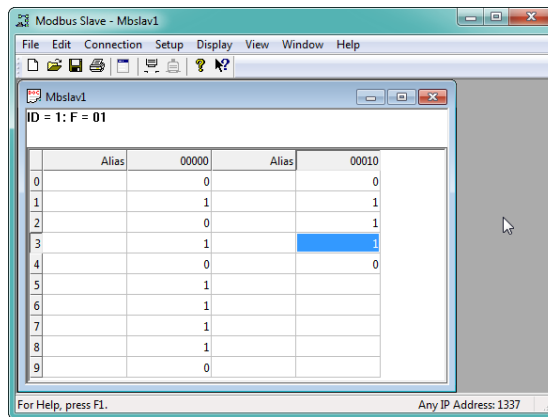


Figura 7.3. Funcionamiento de *Modbus Slave* en la función 01

Con el esclavo configurado, se pueden entonces introducir los parámetros en el dispositivo móvil, como se muestra en la figura 7.4.

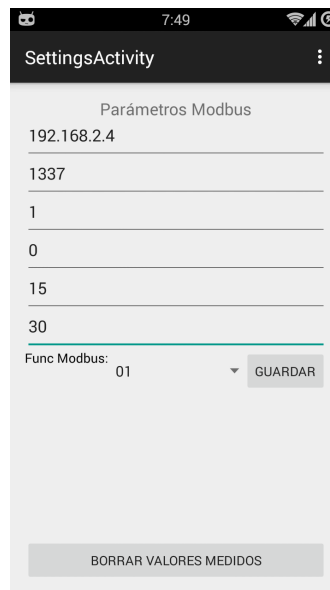


Figura 7.4. Introducción de parámetros en el dispositivo móvil.

Con los parámetros introducidos correctamente, la aplicación se puede mostrar funcionando en la actividad *MainActivity*, luego de presionar el botón *toggle*, como se observa en la figura 7.5.

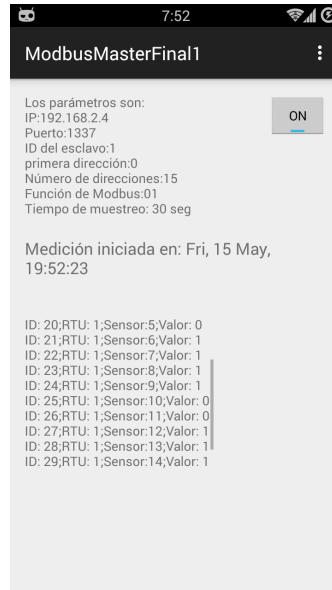


Figura 7.5. Aplicación funcionando en prueba de la función 01.

Se puede observar en la figura 7.5 como se visualizan los sensores si hay más de 10 líneas escritas en una sola medición. Es posible usar una barra de *scroll* para moverse a través de las mediciones como se aprecia en la figura.

Ya que la estructura de la función 02: *Read discrete input* es la misma que la de la función 01, la siguiente prueba se realizará en la función 03: *Read holding register*.

Prueba para la función 03: *Read holding register*

Con la misma configuración de conexión en *Modbus Slave* que para la función 01, solo es necesario configurar el esclavo, como se observa en la figura 7.6.

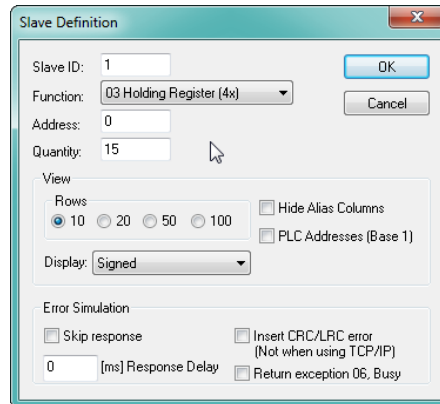


Figura 7.6. Configuración de *Modbus Slave*.

Configurado el esclavo, se colocan algunos valores para medir y probar la función 03, tal como en la figura 7.7. Luego de introducir los parámetros del mismo

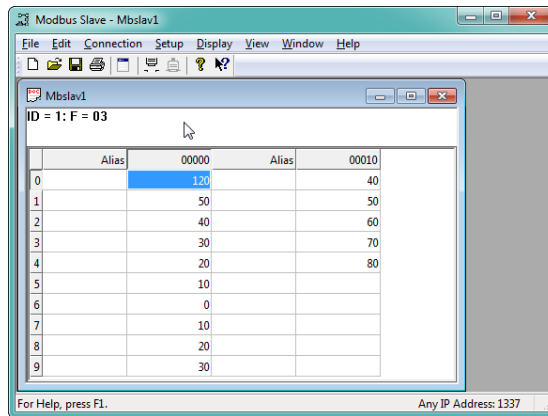


Figura 7.7. Funcionamiento de *Modbus Slave* en la función 03

modo que para la función anterior, se puede observar el dispositivo recibiendo valores de Modbus en la figura 7.8. De acuerdo al mensaje de la función 03 de

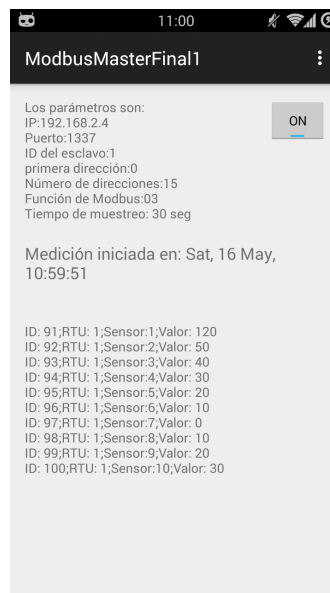


Figura 7.8. Aplicación funcionando en prueba de la función 03.

Modbus, se miden los valores y se comprueba el funcionamiento de la aplicación para esta función. Ya que la estructura de las funciones 03 y 04 de Modbus son similares, no se realizará la misma prueba para la función 04.

7.1.2. Prueba de alarmas

Para comprobar el funcionamiento de las alarmas se colocarán tres puntos en la función 04, y uno de ellos estará fuera del rango aceptado por los valores de alarma, posteriormente se enviará un correo electrónico y un mensaje de texto al dispositivo, comprobando el funcionamiento de las alarmas en la aplicación. En la figura 7.9 se puede observar la actividad de configuración de alarma.



Figura 7.9. Configuración de las alarmas

Luego de configurar las alarmas e iniciar la medición, en la figura 7.10 se tiene el valor de los puntos medidos antes de dispararse la alarma. El sensor número dos incrementa su valor en 1 cada segundo que pasa, esperando que en próximas mediciones se active la alarma.

Luego de algunas mediciones, el punto número dos supera el valor máximo aceptable para el que está configurado (100), y por lo tanto es enviada la alarma. En la figura 7.11 se puede observar el texto de aviso, indicando que se envió la alarma y además, en la parte superior de la pantalla, las notificaciones de un correo electrónico y un mensaje de texto recibidos.

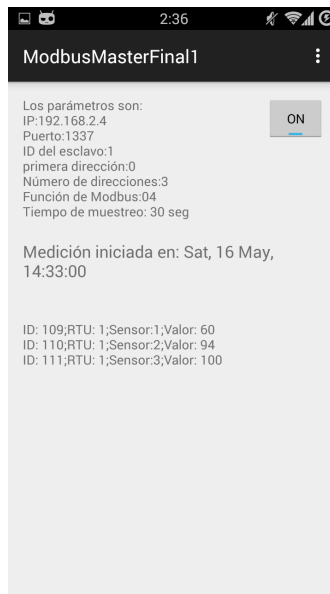


Figura 7.10. Medición antes de disparar alarma.

Recibidos entonces el SMS y el mensaje de correo electrónico, se tiene el siguiente texto:

Para el SMS: ALERTA: Valor del sensor 2 fuera de rango aceptable

Para el E-mail: ALERTA: Valor del sensor 2 fuera de rango aceptable: ID: 112;RTU: 1;Sensor:1;Valor: 60 ID: 113;RTU: 1;Sensor:2;Valor: 123

Posterior a el envío de la alarma, el dispositivo sigue midiendo pero no envía más mensajes de alarma hasta que se reinicie el proceso de medición.

7.1.3. Prueba en un periodo largo de tiempo

Adicionalmente a las pruebas anteriores se realizaron mediciones de la función 04 de Modbus durante más de ocho horas con la finalidad de comprobar la estabilidad de la aplicación a través del tiempo. En la figura 7.12 se puede observar el estado del esclavo simulado en *Modbus Slave*, con 15 puntos funcionando durante

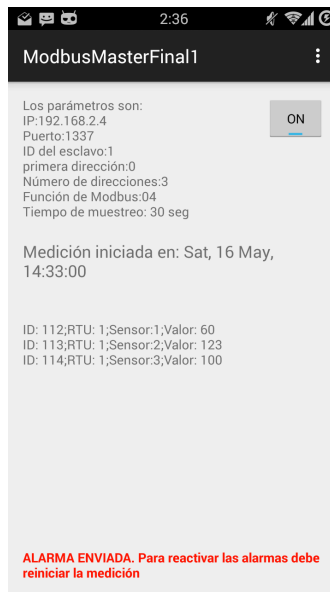


Figura 7.11. Pantalla luego de envío de la alarma.

29277 segundos, la medición fue iniciada en cero para todos los puntos y ocho de ellos incrementaron su valor en una unidad por cada segundo que pasó.

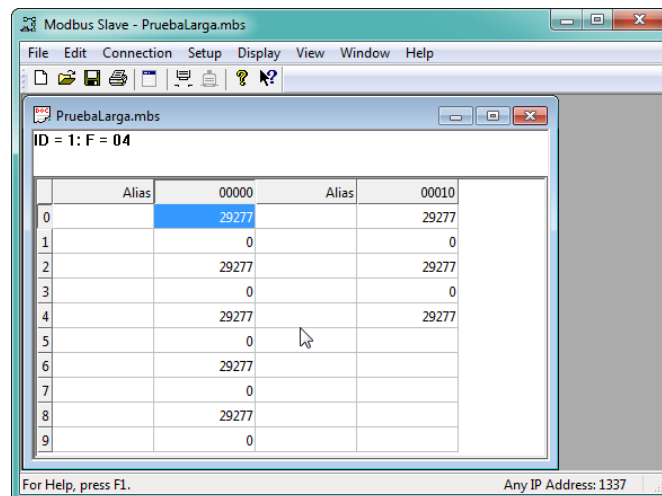


Figura 7.12. Esclavo de Modbus simulado por más de ocho horas.

Luego, en la figura 7.13, se puede observar el resultado de las mediciones, comprobando el funcionamiento adecuado de la aplicación durante un periodo relativamente largo de tiempo.

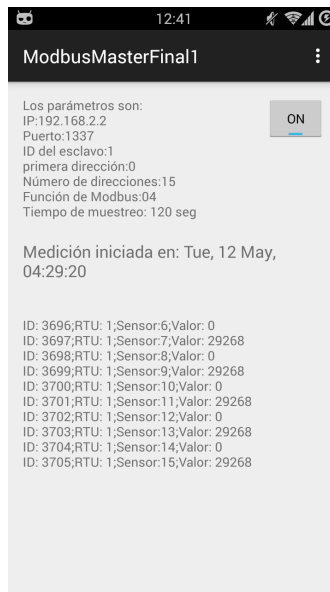


Figura 7.13. Resultado de las mediciones luego de ocho horas.

A pesar de que ocho horas no es un tiempo demasiado extenso para mediciones bajo un protocolo de comunicación de calidad industrial se puede comprobar el correcto funcionamiento de la aplicación, y su estabilidad a través del tiempo.

7.1.4. Errores de entrada

Si el usuario llega a colocar valores errados en el campo de **IP** o **Puerto** o si simplemente el dispositivo móvil y la unidad remota no se logran conectar, al intentar iniciar la medición en la pantalla principal se mostrará un texto indicando el fallo de la conexión, el texto que se genera, gracias a la excepción de *timeout* es el siguiente:

“Error:com.serotonin.modbus4j.exception.ModbusInitExceptionjava.net.

SocketTimeoutException: failed to connect to /192.168.2.4 (port 1337) after 500ms”

Donde la dirección IP indicadas y el puerto indican el lugar donde se intentó

conectar el dispositivo, mientras que los 500ms son el tiempo que intentó el dispositivo en realizar la conexión.

Adicionalmente si el usuario selecciona una función de Modbus errada al momento de introducir los parámetros el dispositivo no podrá recibir los valores, y por lo tanto, el mensaje que se escribirá en la pantalla es uno de los siguientes:

- Error: java.lang.NullPointerException
- Error: message = illegal function

En el caso particular de una función errada, si el protocolo logra reconocer que es una función incorrecta, entrará el mensaje de función ilegal, en otro caso el arreglo que se genera al recibir la respuesta estará vacío y enviará un excepción de puntero nulo. Del mismo modo ocurre si el usuario introduce mal el número de puntos que tiene una unidad remota o el número de identificación de una unidad remota, obtendrá un error del tipo “NullPointerException” o un error de las librerías *Modbus4j*, pero en ningún caso se espera una terminación súbita del programa.

CAPÍTULO VIII

CONCLUSIONES

Finalizado el proyecto se cumple con la totalidad de los objetivos planteados, considerados esenciales para el desarrollo de todo el trabajo realizado, cada punto específico en los objetivos fue necesario para poder cumplir el objetivo general y resolver el problema planteado.

El desarrollo de aplicaciones para el sistema operativo Android es una herramienta importante en la actualidad no solo tomando en cuenta la programación como herramienta fundamental en la ingeniería, sino como representación de la modernidad en la implementación de proyectos competitivos que puedan tener presencia tanto en la industria como en un ambiente de oficina o de cotidianidad como el hogar.

Es necesario nombrar lo significativo que es la implementación de un protocolo de comunicación como lo es Modbus para el desarrollo de aplicaciones en general, no solo en la actualidad sino a lo largo de la historia del control supervisorio. Este protocolo brinda robustez y seguridad al proyecto realizado, generando confianza en las comunicaciones y evitando posibles problemas entre dispositivos.

El uso de bases de datos en el proyecto lleva también otro punto de robustez en el desarrollo de la aplicación, ya que garantiza una estructura de datos fija y su preservación a través del tiempo, otro factor que brinda seguridad al momento de visualizar la aplicación como elemento de control. A pesar de que la implemen-

tación de una base de datos no es estrictamente necesaria, puede brindar a futuro beneficios para el análisis estadístico de datos, esto puede ofrecer un valor agregado como puede ser la optimización del sistema general o encontrar la presencia de problemas que sin un análisis de datos y del sistema no es posible determinar.

La aplicación finalmente desarrollada es también un punto de partida para otros proyectos que solo se pueden visualizar entrando al campo desarrollando proyectos. A partir de esta implementación se pueden plantear variantes como lo es diferentes protocolos de comunicación o la introducción a otro tipo de dispositivos que puedan aportar de otra forma a un sistema como el presentado en este proyecto. Los beneficios generales de este proyecto no se quedan solo en la implementación de una aplicación funcional, sino que aporta ampliamente al conocimiento de ingeniería y abre la visión personal de lo que se puede llegar a desarrollar.

CAPÍTULO IX

RECOMENDACIONES

Cumplidos los objetivos del trabajo actual, no se puede considerar a la aplicación como un proyecto terminado, sino como una base para el desarrollo de la misma. Durante la implementación de este proyecto, basado en los objetivos que se plantearon, surgieron múltiples ideas para implementar a futuro en el marco de la misma aplicación. Entre algunas mejoras que se pueden plantear para este proyecto se pueden nombrar las siguientes:

- Mejora en la presentación de los mensajes de error.
- Implementación de diferentes variantes del protocolo de comunicación, como *Modbus TCP*.
- Incluir una etiqueta de la hora para cada medición, elemento que no se puede implementar con uso de *Timers*.
- Posibilidad de exportar la base de datos como archivo adjunto por correo electrónico.
- Generar solicitud de envío de últimas mediciones como método de consulta remota.
- optimización y mejora general del código

Estas mejoras y recomendaciones pueden representar avances significativos para la vida de la aplicación, que si es implementada de forma adecuada puede convertirse en un producto de uso comercial, basado en librerías de uso libre y un código de dominio público que pueda contribuir a la comunidad de desarrolladores e incrementar el nivel general de programación en la ingeniería local.

REFERENCIAS

- Amadeo, R. (2014). *The history of android*. (Disponible en <http://arstechnica.com/gadgets/2014/06/building-android-a-40000-word-history-of-googles-mobile-os/> y obtenido en Diciembre de 2014)
- Android Developers. (s.f.-a). *Android studio overview*. (Disponible en <http://developer.android.com/tools/studio/index.html> y consultado en febrero de 2015)
- Android Developers. (s.f.-b). *Application fundamentals*. (Disponible en <http://developer.android.com/guide/components/fundamentals.html> y obtenido en Enero de 2015)
- Android Developers. (s.f.-c). *Layouts*. (Disponible en <http://developer.android.com/intl/es/guide/topics/ui/declaring-layout.html> y obtenido en Enero de 2015)
- Android Developers. (s.f.-d). *Tools help*. (Disponible en <http://developer.android.com/tools/help/index.html> y obtenido en Enero de 2015)
- Android Developers. (s.f.-e). *Ui overview*. (Disponible en <http://developer.android.com/guide/topics/ui/overview.html> y obtenido en Enero de 2015)
- Android Developers. (s.f.-f). *<uses-sdk>*. (Disponible en <http://developer.android.com/guide/topics/manifest/uses-sdk-element.html#ApiLevels> y obtenido en Enero de 2015)
- Berbis, J. (2012). *Diseño e implementación de un sistema de control para la dosificación automática de reactivos de proceso en una planta de tratamiento de aguas blancas*.

- Calin, D. (2014). *Raspberry pi and android - guides and resources*. (Disponible en <http://www.intorobotics.com/raspberry-pi-android-guides-resources/> y obtenido en Enero de 2015)
- Duckett, C. (2014). *Google releases android studio, kills off eclipse adt plugin*. (Disponible en <http://www.zdnet.com/article/google-releases-android-studio-kills-off-eclipse-adt-plugin/> y consultado en febrero de 2015)
- Elgin, B. (2005). *Google buys android for its mobile arsenal*. (Disponible en <http://www.webcitation.org/5wk7sIvVb> y obtenido en Diciembre de 2014)
- IBM Knowledge. (s.f.). *How sockets work*. (Disponible en http://www-01.ibm.com/support/knowledgecenter/ssw_ibm_i_71/rzab6/howdosockets.htm y consultado en enero de 2015)
- J. D. Ullman. (1988). Principles of Database and Knowledge-Base Systems. En *Computer science press* (Vols. I,II).
- Jon Simon. (2010). *Sending emails without user intervention (no intents) in android*. (Disponible en http://www.jondev.net/articles/Sending_Emails_without_User_Intervention_%28no_Intents%29_in_Android y obtenido en febrero de 2015)
- Microsoft Support. (2014). *The osi model's seven layers defined and functions explained*. (Disponible en <https://support.microsoft.com/en-us/kb/103884/> y consultado en enero de 2015, revisión 2.1)
- Modbus Organization. (2005). *What is modbus protocol?* (Disponible en <http://www.modbus.org/faq.php> y consultado en febrero de 2015)
- Modbus Tools. (2002). *Modbus slave help index*. (obtenido en Diciembre de 2014)
- Open handset alliance. (2007). *Industry leaders announce open platform for mobile devices*. (Disponible en http://www.openhandsetalliance.com/press_110507.html y obtenido en Diciembre de 2014)
- Oracle. (2007). *The java tutorials*. (Disponible en

- <http://docs.oracle.com/javase/tutorial/java/concepts/index.html> y obtenido en Diciembre de 2014)
- Pichai, S. (2013). *1,000,000,000 activations*. (Disponible en <https://plus.google.com/+VicGundotra/posts/8CVJ79nPQwN> y obtenido en Diciembre de 2014)
- PLCDev. (s.f.). *Modbus FAQ*. (Disponible en http://www.plcdev.com/modbus_faq y obtenido en Diciembre de 2014)
- Ravi. (2009). *User interfaces of hmi*. (Disponible en <http://ravi-softwares.blogspot.com/2009/08/user-interfaces-of-hmi.html> y obtenido en Diciembre de 2014)
- Ravi Tamada. (2011). *Android sqlite database tutorial*. (Disponible en <http://www.androidhive.info/2011/11/android-sqlite-database-tutorial/> y obtenido en febrero de 2015)
- RTA Automation. (s.f.). *Modbus rtu*. (Disponible en <http://www.rtaautomation.com/technologies/modbus-rtu/> y consultado en febrero de 2015)
- RTA Automation. (2010). *All you need to know about modbus tcp*. (Disponible en <https://www.youtube.com/watch?v=E1nsgukeKKA> y obtenido en Diciembre de 2014)
- Russell, J. (2009). *Scada/ems history*. (Disponible en <http://scadahistory.com/index.html> y obtenido en Diciembre de 2014)
- Siemens. (s.f.). *History of the Modbus protocol*. (Disponible en http://w3.usa.siemens.com/us/internet-dms/btlv/CircuitProtection/MoldedCaseBreakers/docs_MoldedCaseBreakers/Modbus%20Information.doc y obtenido en Diciembre de 2014)
- SourceForge.net. (2008). *Modbus4j*. (Disponible en <https://sourceforge.net/projects/modbus4j/> y obtenido en febrero de 2015)

SQLite. (2009). *About sqlite*. (Disponible en <https://www.sqlite.org/about.html> y obtenido en febrero de 2015)

Wikipedia. (2014a). *Android software development*. (Disponible en http://en.wikipedia.org/wiki/Android_software_development y obtenido en Diciembre de 2014)

Wikipedia. (2014b). *Gateway (telecommunications)*. (Disponible en [http://en.wikipedia.org/wiki/Gateway_\(telecommunications\)](http://en.wikipedia.org/wiki/Gateway_(telecommunications)) y consultado en enero de 2015)

ANEXOS

ARCHIVOS Y CÓDIGO DE LA APLICACIÓN

1.1. Archivo *MainActivity.java*

```
package com.example.josempd.modbusmaster;

import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.app.PendingIntent;
import android.content.IntentFilter;
import android.content.SharedPreferences;
import android.os.AsyncTask;
import android.os.Handler;
import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.telephony.SmsManager;
import android.text.method.ScrollingMovementMethod;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.TextView;
import android.widget.Toast;
import android.widget.ToggleButton;

import com.serotonin.modbus4j.ModbusFactory;
```

```

import com.serotonin.modbus4j.ModbusMaster;
import com.serotonin.modbus4j.exception.ModbusInitException;
import com.serotonin.modbus4j.exception.ModbusTransportException;
import com.serotonin.modbus4j.ip.IpParameters;
import com.serotonin.modbus4j.msg.ReadCoilsRequest;
import com.serotonin.modbus4j.msg.ReadCoilsResponse;
import com.serotonin.modbus4j.msg.ReadDiscreteInputsRequest;
import com.serotonin.modbus4j.msg.ReadDiscreteInputsResponse;
import com.serotonin.modbus4j.msg.ReadHoldingRegistersRequest;
import com.serotonin.modbus4j.msg.ReadHoldingRegistersResponse;
import com.serotonin.modbus4j.msg.ReadInputRegistersRequest;
import com.serotonin.modbus4j.msg.ReadInputRegistersResponse;

import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;

public class MainActivity extends ActionBarActivity {

    //Variable para manejo de la base de datos
    DatabaseHandler db = new DatabaseHandler(this);

    //Declaracion de los TextView de la actividad
    TextView ParamView;
    TextView ValoresView;
    TextView TVInicio;
    TextView tvAlarmas;

    //String globales para escribir los TextView:
    String ParamText = "";

```

```

String ValoresText = "";

//String globales para las SharedPreferences:
String IPK;
String PortK;
String SlaveK;
String SensorK;
String CountK;
String ModbusK;
String Tdm;

//Declaracion del boton toggle
ToggleButton ONOFF;

//Variable global, booleana para saber si se envio una alarma
boolean AlarmSent;

//Strings que muestran la fecha y hora actual.
static Date d = new Date();
static CharSequence Fecha = android.text.format.DateFormat.format
    ("EEE, dd MMM, HH:mm:ss", d.getTime());

//Creacion del timer para realizar el ciclo de mediciones
static Timer timer = new Timer();

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

```

```

setContentView(R.layout.activity_main);
ParamView = (TextView) findViewById(R.id.tvParam);
ValoresView = (TextView) findViewById(R.id.tvValores);
TVInicio = (TextView) findViewById(R.id.tvInicio);
tvAlarmas = (TextView) findViewById(R.id.tvAlarmas);
ValoresView.setMovementMethod(new ScrollingMovementMethod());

ONOFF = (ToggleButton) findViewById(R.id.ConexButton);

AlarmSent = false;

SharedPreferences MisPreferencias =
    getSharedPreferences("MisPreferencias",
        Context.MODE_PRIVATE);
IPK = MisPreferencias.getString("IPKey", "0.0.0.0");
PortK = MisPreferencias.getString("PortKey", "0000");
SlaveK = MisPreferencias.getString("SlaveKey", "1");
SensorK = MisPreferencias.getString("SensorKey", "1");
CountK = MisPreferencias.getString("CountKey", "1");
ModbusK = MisPreferencias.getString("FuncKey", "04");
Tdm = MisPreferencias.getString("TiempoKey", "100");

ParamText = ParamText + "Los par\u00e1metros son:" + "\n" +
    "IP:" + IPK + "\n" +
    "Puerto:" + PortK + "\n" +
    "ID del esclavo:" + SlaveK + "\n" +
    "primera direcci\u00f3n:" + SensorK + "\n" +
    "N\u00famero de direcciones:" + CountK + "\n" +
    "Funci\u00f3n de Modbus:" + ModbusK + "\n" +
    "Tiempo de muestreo: " + Tdm + " seg";

```

[illegible]

```

        Integer.parseInt(SlaveK),
        Integer.parseInt(SensorK),
        Integer.parseInt(CountK),
        Integer.parseInt(ModbusK));
        myClientTask.execute();
    } catch (Exception e) {
        e.printStackTrace();
        ValoresView.setText(e.toString());
    }
}

});

}

};

//En este segmento se configura el timer
//0 ms de retardo, Cada Tdm*1000ms
timer.schedule(doAsynchronousTask, 0,
        Integer.parseInt(Tdm)*1000);
} else {
    //Si se vuelve a tocar el boton toggle
    //Se cancela el timer
    //Se reemplaza el texto de modbus
    //Se limpian los valores de alarma
    //La actividad pasa a un estado de espera por
    //presionar de nuevo el boton
    timer.cancel();
    ValoresView.setText("no se est\u00e1 midiendo actualmente.");
    tvAlarmas.setText("");
    AlarmSent = false;
//}
}

```

```
}
```

```
@Override
```

```
public boolean onCreateOptionsMenu(Menu menu) {  
    // Inflate the menu; this adds items to the action bar if it is  
    // present.  
    getMenuInflater().inflate(R.menu.menu_main, menu);  
    return true;  
}
```

```
@Override
```

```
public boolean onOptionsItemSelected(MenuItem item) {  
    // Handle action bar item clicks here. The action bar will  
    // automatically handle clicks on the Home/Up button, so long  
    // as you specify a parent activity in AndroidManifest.xml.  
    int id = item.getItemId();  
  
    //noinspection SimplifiableIfStatement  
    if (id == R.id.action_settings) {  
        Intent i = new Intent(this, SettingsActivity.class);  
        ONOFF.setChecked(false);  
        timer.cancel();  
        ValoresView.setText("no se est\u00e1 midiendo actualmente.");  
        startActivity(i);  
        return true;  
    }
```

```
    if (id == R.id.alarm_settings) {  
        Intent i = new Intent(this, AlarmConfig.class);  
        ONOFF.setChecked(false);
```



```

        timer.cancel();
        ValoresView.setText("no se est\u00e1 midiendo actualmente.");
        startActivity(i);
        return true;
    }

    return super.onOptionsItemSelected(item);
}

//Enviar SMS a destinatario
public void sendSMS(String phoneNumber, String message)
{
    String SENT = "SMS_SENT";
    String DELIVERED = "SMS_DELIVERED";

    PendingIntent sentPI = PendingIntent.getBroadcast(this, 0,
        new Intent(SENT), 0);

    PendingIntent deliveredPI = PendingIntent.getBroadcast(this, 0,
        new Intent(DELIVERED), 0);

    //Luego de enviar el mensaje.
    registerReceiver(new BroadcastReceiver() {
        @Override
        public void onReceive(Context arg0, Intent arg1) {
            switch (getResultCode()) {
                case Activity.RESULT_OK:
                    Toast.makeText(getBaseContext(),
                        "Mensaje de alarma enviado.",
                        Toast.LENGTH_SHORT).show();

```

```

        break;
    case SmsManager.RESULT_ERROR_GENERIC_FAILURE:
        Toast.makeText(getBaseContext(),
            "Falla generica al enviar SMS",
            Toast.LENGTH_SHORT).show();
        break;
    case SmsManager.RESULT_ERROR_NO_SERVICE:
        Toast.makeText(getBaseContext(),
            "Sin servicio, sms no enviado.",
            Toast.LENGTH_SHORT).show();
        break;
    case SmsManager.RESULT_ERROR_NULL_PDU:
        Toast.makeText(getBaseContext(),
            "Mensaje no enviado, receptor no
            disponible",
            Toast.LENGTH_SHORT).show();
        break;
    case SmsManager.RESULT_ERROR_RADIO_OFF:
        Toast.makeText(getBaseContext(),
            "Antena no disponible, mensaje no enviado.",
            Toast.LENGTH_SHORT).show();
        break;
    }
}

}, new IntentFilter(SENT));

//Cuando el mensaje fue entregado
registerReceiver(new BroadcastReceiver() {
    @Override
    public void onReceive(Context arg0, Intent arg1) {

```

```

        switch (getResultCode()) {
            case Activity.RESULT_OK:
                Toast.makeText(getBaseContext(),
                    "Mensaje de alarma entregado",
                    Toast.LENGTH_SHORT).show();

                break;

            case Activity.RESULT_CANCELED:
                Toast.makeText(getBaseContext(),
                    "Mensaje de alarma enviado, pero no
                    entregado",
                    Toast.LENGTH_SHORT).show();

                break;
        }
    }

    }, new IntentFilter(DELIVERED));

    SmsManager sms = SmsManager.getDefault();
    sms.sendTextMessage(phoneNumber, null, message, sentPI,
        deliveredPI);
}

public void sendMail(String mail, String message){
    String Mail = mail;
    String Message = message;
    Mail m = new Mail("alarmasmodbus@gmail.com", "AlarmaModbus");
    String[] toArr = {Mail};
    m.setTo(toArr);
    m.setFrom("alarmasmodbus@gmail.com");
    m.setSubject("Alerta");
    m.setBody(Message);
}

```

```

    try {
        m.send();
    } catch(Exception e) {
        Log.e("MailApp", "No se pudo enviar el E-mail", e);
    }
}

}

public class ModbusTask extends AsyncTask<Void, String, String> {

    String Resp;
    String IPMod;
    int PortMod;
    int slaveId;
    int RefMod;
    int CountMod;
    int Func;

    //Arreglos para manipular valores de Modbus
    private int[] ModbusArray;
    private int[] SensorArray;
    private boolean[] ArregloModbus01;
    private boolean[] ArregloModbus02;
    private short[] ArregloModbus03;
    private short[] ArregloModbus04;

    ModbusTask(String addr, int port, int ID, int ref, int count, int
        func) {
        IPMod = addr;
        PortMod = port;
        slaveId = ID;
        RefMod = ref;

```

```

        CountMod = count;

        Func = func;
    }

@Override
protected String doInBackground(Void... arg0) {

    ModbusFactory factory = new ModbusFactory();
    IpParameters params = new IpParameters();//Se crea la
        variable que
    // maneja los parametros para el maestro de modbus
    params.setHost(IPMod);//Se asigna el IP del esclavo
    params.setPort(PortMod);//Se asigna el puerto del esclavo
    params.setEncapsulated(true);//Se asigna el valor True a la
        encapsulacion del
    // mensaje dentro de una trama RTU
    ModbusMaster master = factory.createTcpMaster(params, true);
        //Se crea un maestro de
    // Modbus segun los parametros y se mantiene abierto con
        'keepalive = true'
    master.setTimeout(500);//Se coloca un timeout de 500ms para
        enviar la trama
    master.setRetries(2);//Se colocan 2 reintentos por fallo en
        la conexion

    try {
        master.init(); //Se inicializa el maestro de ModBus
        switch (Func) {
            case 01:// Se genera la peticion de la funcion 01

```

```

ReadCoilsRequest request01 = new
    ReadCoilsRequest(slaveId,
        RefMod, CountMod);
// Se genera la respuesta de la funcion 01
ReadCoilsResponse response01 =
    (ReadCoilsResponse) master.send(request01);
if (response01.isException()) {
    Resp = "Exception response: " +
        "message=" +
            response01.getExceptionMessage();
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            ValoresView.setText(Resp);
        }
    });
} else {
    ArregloModbus01 =
        response01.getBooleanData();
    int i = 0;
    ModbusArray = new int[CountMod];
    while (i < CountMod) {
        ModbusArray[i] = ArregloModbus01[i] ? 1
            : 0;
        i++;
    }
}
case 02:// Se genera la peticion de la funcion 02
    ReadDiscreteInputsRequest request02 =

```

```

        new ReadDiscreteInputsRequest(slaveId,
            RefMod, CountMod);
    // Se genera la respuesta de la funcion 02
    ReadDiscreteInputsResponse response02 =
        (ReadDiscreteInputsResponse)
            master.send(request02);
    if (response02.isException()) {
        Resp = "Exception response: message=" +
            response02.getExceptionMessage();
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                ValoresView.setText(Resp);
            }
        });
    } else {
        ArregloModbus02 =
            response02.getBooleanData();
        int j = 0;
        ModbusArray = new int[CountMod];
        while (j < CountMod) {
            ModbusArray[j] = ArregloModbus02[j] ? 1
                : 0;
            j++;
        }
    }
}

case 03:// Se genera la peticion de la funcion 03
    ReadHoldingRegistersRequest request03 =
        new
            ReadHoldingRegistersRequest(slaveId,

```

```

        RefMod, CountMod);

// Se genera la respuesta de la funcion 03
ReadHoldingRegistersResponse response03 =
    (ReadHoldingRegistersResponse)
        master.send(request03);

if (response03.isException()) {
    Resp = "Exception response: message=" +
        response03.getExceptionMessage();
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            ValoresView.setText(Resp);
        }
    });
} else {
    ArregloModbus03 = response03.getShortData();
    int k = 0;
    ModbusArray = new int[CountMod];
    while (k < CountMod) {
        ModbusArray[k] = (int)
            ArregloModbus03[k];
        k++;
    }
}

case 04:// Se genera la peticion de la funcion 04
ReadInputRegistersRequest request04 =
    new ReadInputRegistersRequest(slaveId,
        RefMod, CountMod);

// Se genera la respuesta de la funcion 04

```



```

ReadInputRegistersResponse response04 =
    (ReadInputRegistersResponse)
        master.send(request04);
if (response04.isException()) {
    Resp = "Exception response: message=" +
        response04.getExceptionMessage();
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            ValoresView.setText(Resp);
        }
    });
} else {
    ArregloModbus04 = response04.getShortData();
    int l = 0;
    ModbusArray = new int[CountMod];
    while (l < CountMod) {
        ModbusArray[l] = (int)
            ArregloModbus04[l];
        l++;
    }
}

break;

}

//Se inicializa el texto de respuesta
ValoresText = "";

//Inicia un ciclo while para escribir los valores en la
    base de datos
int i = 0;

```

```

SensorArray = new int[Integer.parseInt(CountK)];
while (i < Integer.parseInt(CountK)) {
    SensorArray[i] = i + 1;
    Log.d("Insertar datos: ", "Insertando ..");
    //Se agregan los valores en la base de datos
    db.addValue(new ValorModbus(1, SensorArray[i],
        ModbusArray[i]));
    db.close();
    //Se escriben en un String que luego actualiza la
        pantalla principal
    ValoresText = ValoresText + "ID: " + db.LastValor() +
        ";RTU: " +
        "1" + ";Sensor:" + SensorArray[i] + ";Valor: " +
        ModbusArray[i] + " \n";
    //Se extraen los valores de rango de las alarmas
    //Segun la remota y el sensor actuales en el ciclo
    String rangomin = db.ValorminValue(1, SensorArray[i]);
    String rangomax = db.ValormaxValue(1, SensorArray[i]);
    //Existe una condicion para funcionar si no hay alarmas
    if(!rangomin.matches("") || !rangomax.matches("")){
        int rangominint = Integer.parseInt(rangomin);
        int rangomaxint = Integer.parseInt(rangomax);
        String rango1 = rangomin;
        String rango2 = rangomax;
        Log.d("Modbus", rango1);
        Log.d("Modbus", rango2);
        //Si hay alarmas, comparar el sensor actual del
            ciclo
        //Con los valores de alarma

```

```

        if ((ModbusArray[i] > rangomaxint ||
            ModbusArray[i] < rangominint) &&
            !AlarmSent) {
            //Si hay una alarma y no se ha enviado una antes
            //Escribir los mensajes de alarma
            String MailAlarma = db.MailValue(1,
                SensorArray[i]);
            String CelAlarma = db.CelValue(1,
                SensorArray[i]);
            String MensajeAlarmaMail = "ALERTA: Valor del
                sensor " +
                String.valueOf(SensorArray[i] +
                    " fuera de rango aceptable:" + "\n"
                    + ValoresText);
            String MensajeAlarmaCel = "ALERTA: Valor del
                sensor " +
                String.valueOf(SensorArray[i] +
                    " fuera de rango aceptable");
            //Enviar SMS y correo electronico de alarma
            sendSMS(CelAlarma, MensajeAlarmaCel);
            sendMail(MailAlarma, MensajeAlarmaMail);
            //Se coloca verdadera la variable de 'alarma
            enviada'
            AlarmSent = true;
        }
    }
    i++;
}

runOnUiThread(new Runnable() {
    @Override

```

```

        public void run() {
            ValoresView.setText(ValoresText);

            //Si se envio una alarma, activar texto de alarmas
            if(AlarmSent){
                tvAlarmas.setText("ALARMA ENVIADA. Para
                                reactivar " +
                                "las alarmas debe reiniciar la
                                medici\u00f3n");
            }
        }
    });

    } catch (ModbusInitException | ModbusTransportException e) {
        e.printStackTrace();
        Resp = "Error: " + e.toString();
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                ValoresView.setText(Resp);
            }
        });
    } finally {
        //Se destruye el maestro de modbus
        //Que se inicializa en el proximo ciclo
        master.destroy();
    }

    return ValoresText;
}

@Override
//Se actualiza la pantalla principal con los valores guardados
previamente

```

```
protected void onProgressUpdate(String... values) {  
    ValoresView.setText(ValoresText);  
}  
}  
}
```

1.2. Archivo de *Layout: activity_main.xml*

En este archivo se muestra el código del *Layout* de la actividad principal

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
    android:layout_height="match_parent"
        android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
        tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/tvParam"
        android:layout_alignParentTop="true"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_toLeftOf="@+id/ConexButton"
        android:layout_toStartOf="@+id/ConexButton"
        android:text="Parámetros" />

    <ToggleButton
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/ConexButton"
        android:layout_alignParentTop="true"
```

```
android:layout_alignParentRight="true"
android:layout_alignParentEnd="true"
android:checked="false"
android:onClick="onToggleClicked" />
```

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceSmall"
    android:text="Valores Modbus"
    android:id="@+id/tvValores"
    android:layout_below="@+id/tvInicio"
    android:layout_alignParentLeft="true"
    android:layout_alignParentStart="true"
    android:layout_marginTop="42dp"
    android:maxLines = "10"
    android:scrollbars = "vertical"/>
```

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceMedium"
    android:text="Inicio:"
    android:id="@+id/tvInicio"
    android:layout_below="@+id/ConexButton"
    android:layout_alignParentLeft="true"
    android:layout_alignParentStart="true"
    android:layout_marginTop="108dp" />
```

```
<TextView
```

```
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:id="@+id/tvAlarmas"
android:layout_alignParentLeft="true"
android:layout_alignParentStart="true"
android:layout_alignParentBottom="true"
android:layout_alignRight="@+id/ConexButton"
android:layout_alignEnd="@+id/ConexButton"
android:textColor="#ffff1300"
android:textStyle="bold" />
```

```
</RelativeLayout>
```

1.3. Archivo *SettingsActivity.java*

```
package com.example.josempd.modbusmaster;

import android.content.Context;
import android.content.Intent;
import android.content.SharedPreferences;
import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Spinner;
import android.widget.Toast;

public class SettingsActivity extends ActionBarActivity {

    DatabaseHandler db = new DatabaseHandler(this);

    //Se declaran los EditText
    EditText DirIP;
    EditText Port;
    EditText SlaveID;
    EditText SensorID;
    EditText SensorNumber;
    EditText Tdm;

    //Se declara el Spinner de la funcion de modbus
```

```

Spinner FuncModbus;

//Se declaran los botones
Button SaveButton;
Button DeleteAllButton;


//Se declaran las Shared Preferences
SharedPreferences sharedPreferences;

public static final String MisPreferencias = "MisPreferencias";
public static final String IPK = "IPKey";
public static final String PortK = "PortKey";
public static final String SlaveK = "SlaveKey";
public static final String SensorK = "SensorKey";
public static final String CountK = "CountKey";
public static final String ModbusK= "FuncKey";
public static final String TiempoK = "TiempoKey";


@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_settings);


    //Se asignan los EditText a variables de la activity
    DirIP = (EditText) findViewById(R.id.dirIP);
    Port = (EditText) findViewById(R.id.Port);
    SlaveID = (EditText) findViewById(R.id.SlaveID);
    SensorID = (EditText) findViewById(R.id.SensorID);
    SensorNumber = (EditText) findViewById(R.id.SensorNumber);


    Tdm = (EditText) findViewById(R.id.Tdm);

```

```

SaveButton = (Button) findViewById(R.id.SaveButton);
DeleteAllButton = (Button) findViewById(R.id.DeleteAllButton);

//Se inicializa el spinner
FuncModbus = (Spinner) findViewById(R.id.spinner);
String []opciones={"01", "02", "03", "04"};
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
    android.R.layout.simple_spinner_item, opciones);
FuncModbus.setAdapter(adapter);

//Funcion del boton save
SaveButton.setOnClickListener(buttonSaveOnClickListener);

DeleteAllButton.setOnClickListener(buttonDeleteOnClickListener);

//Se inicializan las preferencias
sharedpreferences = getSharedPreferences(MisPreferencias,
    Context.MODE_PRIVATE);
}

View.OnClickListener buttonSaveOnClickListener = new
View.OnClickListener() {

@Override
public void onClick(View arg0){
    //Se verifica la validez de los parametros
    if (DirIP.getText().toString().matches("") ||
        SensorNumber.getText().toString().matches("") ||
        SlaveID.getText().toString().matches("") ||
        Port.getText().toString().matches("")) ||

```

```

        SensorID.getText().toString().matches("") ||
        SensorNumber.getText().toString().matches("0") ||
        SlaveID.getText().toString().matches("0")) {
    Toast.makeText(getBaseContext(), "Existen par\u00e9metros
        inv\u00edlidos", Toast.LENGTH_SHORT).show();
}
else {
    //Se guardan los Strings en las preferencias.
    SharedPreferences.Editor editor =
        sharedPreferences.edit();
    editor.putString(IPK, DirIP.getText().toString());
    editor.putString(PortK, Port.getText().toString());
    editor.putString(SlaveK, SlaveID.getText().toString());
    editor.putString(SensorK, SensorID.getText().toString());
    editor.putString(CountK,
        SensorNumber.getText().toString());
    editor.putString(TiempoK, Tdm.getText().toString());
    editor.putString(ModbusK,
        FuncModbus.getSelectedItem().toString());
    editor.apply();
    //Se llama a la main activity
    Intent i = new Intent(getBaseContext(),
        MainActivity.class);
    onNewIntent(i);
    startActivity(i);
}
}
};

```

```

View.OnClickListener buttonDeleteOnClickListener = new
    View.OnClickListener() {

        @Override
        public void onClick(View arg0){
            db.clearValores();
            Toast.makeText(getBaseContext(), "Valores borrados",
                Toast.LENGTH_SHORT).show();
        }
    };

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is
        // present.
        getMenuInflater().inflate(R.menu.menu_settings, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();

        //noinspection SimplifiableIfStatement
        if (id == R.id.action_settings) {
            Intent i = new Intent(this, SettingsActivity.class);
            startActivity(i);
            return true;
        }
    }

```

```
    }  
  
    return super.onOptionsItemSelected(item);  
  }  
}
```

1.4. Archivo de *Layout: activity_settings.xml*

En este archivo se muestra el código del *Layout* de la actividad de configuración de parámetros.

```
<RelativeLayout

    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent"
    android:layout_height="match_parent"

    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    tools:context="com.example.josempd.modbusmaster.SettingsActivity">

    <TextView

        android:layout_width="wrap_content"
        android:layout_height="wrap_content"

        android:textAppearance="?android:attr/textAppearanceMedium"
        android:text="Parámetros Modbus"
        android:id="@+id/tv1"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true" />

    <EditText

        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/dirIP"
        android:layout_below="@+id/tv1"
        android:layout_alignParentLeft="true"
```

```
android:layout_alignParentStart="true"
android:layout_alignParentRight="true"
android:layout_alignParentEnd="true"
android:hint="Dirección IP"
android:singleLine="true" />
```

```
<EditText
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/Port"
    android:layout_below="@+id/dirIP"
    android:layout_alignParentLeft="true"
    android:layout_alignParentStart="true"
    android:layout_alignRight="@+id/dirIP"
    android:layout_alignEnd="@+id/dirIP"
    android:hint="Puerto"
    android:inputType="number" />
```

```
<EditText
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/SlaveID"
    android:layout_below="@+id/Port"
    android:layout_alignParentLeft="true"
    android:layout_alignParentStart="true"
    android:layout_alignParentRight="true"
    android:layout_alignParentEnd="true"
    android:hint="Dirección del esclavo"
    android:inputType="number" />
```



```
<EditText
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/SensorID"
    android:layout_below="@+id/SlaveID"
    android:layout_alignParentLeft="true"
    android:layout_alignParentStart="true"
    android:layout_alignRight="@+id/SlaveID"
    android:layout_alignEnd="@+id/SlaveID"
    android:hint="Punto inicial"
    android:inputType="number" />
```

```
<EditText
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/SensorNumber"
    android:layout_below="@+id/SensorID"
    android:layout_alignParentLeft="true"
    android:layout_alignParentStart="true"
    android:layout_alignRight="@+id/SensorID"
    android:layout_alignEnd="@+id/SensorID"
    android:inputType="number"
    android:hint="Número de puntos" />
```

```
<Spinner
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/spinner"
    android:spinnerMode="dropdown"
    android:layout_toRightOf="@+id/TextMB"
```

```
        android:layout_toLeftOf="@+id/SaveButton"
        android:layout_toStartOf="@+id/SaveButton"
        android:layout_alignTop="@+id/TextMB"
        android:layout_alignBottom="@+id/TextMB" />
```

<Button

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Guardar"
        android:id="@+id/SaveButton"
        android:layout_below="@+id/Tdm"
        android:layout_alignParentRight="true"
        android:layout_alignParentEnd="true" />
```

<TextView

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Func Modbus:"
        android:id="@+id/TextMB"
        android:textColor="#000"
        android:layout_alignBottom="@+id/SaveButton"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_alignTop="@+id/SaveButton"
        android:textAlignment="center" />
```

<EditText

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/Tdm"
```

```
android:layout_below="@+id/SensorNumber"
android:layout_alignParentLeft="true"
android:layout_alignParentStart="true"
android:hint="Tiempo de muestreo (seg)"
android:inputType="number"
android:layout_alignRight="@+id/DeleteAllButton"
android:layout_alignEnd="@+id/DeleteAllButton" />
```

```
<Button
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Borrar valores medidos"
    android:id="@+id/DeleteAllButton"
    android:layout_alignParentBottom="true"
    android:layout_alignLeft="@+id/TextMB"
    android:layout_alignStart="@+id/TextMB"
    android:layout_alignRight="@+id/SaveButton"
    android:layout_alignEnd="@+id/SaveButton" />
```

```
</RelativeLayout>
```

1.5. Archivo *AlarmConfig.java*

```
package com.example.josempd.modbusmaster;

import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.text.method.ScrollingMovementMethod;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
import android.widget.Toast;

import java.util.List;

public class AlarmConfig extends ActionBarActivity {

    DatabaseHandler db = new DatabaseHandler(this);

    EditText etRTU;
    EditText etSensor;
    EditText etValorminimo;
    EditText etValormaximo;
    EditText etMail;
    EditText etPhone;

    int intRTU;
    int intSensor;
    int intValormin;
```

```

int intValormax;

String strMail;
String strPhone;

TextView AlarmView;

Button SaveAlarm;
Button DeleteAlarms;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_alarm_config);

    etRTU = (EditText) findViewById(R.id.etRTU);
    etSensor = (EditText) findViewById(R.id.etSensor);
    etValorminimo = (EditText) findViewById(R.id.etValormin);
    etValormaximo = (EditText) findViewById(R.id.etValormax);
    etMail = (EditText) findViewById(R.id.etMail);
    etPhone = (EditText) findViewById(R.id.etPhone);

    AlarmView = (TextView) findViewById(R.id.AlarmView);
    AlarmView.setMovementMethod(new ScrollingMovementMethod());

    SaveAlarm = (Button) findViewById(R.id.SaveAlarmButton);
    DeleteAlarms = (Button) findViewById(R.id.DeleteAlarmButton);

    SaveAlarm.setOnClickListener(SaveAlarmOnClickListener);
    DeleteAlarms.setOnClickListener(DeleteAlarmsOnClickListener);

```

```
}
```

```
View.OnClickListener SaveAlarmOnClickListener = new  
    View.OnClickListener() {  
        @Override  
        public void onClick(View arg0){  
            String alarm = "";  
            if (etRTU.getText().toString().matches("") ||  
                etSensor.getText().toString().matches("") ||  
                etValorminimo.getText().toString().matches("") ||  
                etValormaximo.getText().toString().matches("") ||  
                etMail.getText().toString().matches("") ||  
                etPhone.getText().toString().matches("")) {  
                Toast.makeText(getApplicationContext(), "Existen par\u00e1metros  
                    inv\u00e1lidos", Toast.LENGTH_SHORT).show();  
            }  
            else {  
                intRTU = Integer.parseInt(etRTU.getText().toString());  
                intSensor =  
                    Integer.parseInt(etSensor.getText().toString());  
                intValormin =  
                    Integer.parseInt(etValorminimo.getText().toString());  
                intValormax =  
                    Integer.parseInt(etValormaximo.getText().toString());  
                strMail = etMail.getText().toString();  
                strPhone = etPhone.getText().toString();  
  
                db.addAlarma(new AlarmasModbus(intRTU,  
                    intSensor,  
                    intValormin,
```

```

        intValormax,
        strMail,
        strPhone));
List<AlarmasModbus> Alarmas = db.getAllAlarmas();

for (AlarmasModbus cn : Alarmas) {
    alarm = alarm + "ID:" + cn.getID() + " ;RTU:" +
        cn.getRTU()
        + " ;Sensor:" + cn.getSensor() + " ;Valormin:"
        + cn.getValormin()
        + " ;Valormax:" + cn.getValormax() + " ;Mail:"
        + cn.getMail()
        + " ;Cel:" + cn.getCel() + "\n";
    AlarmView.setText(alarm);
}
}
};

```

```

View.OnClickListener DeleteAlarmsOnClickListener = new
    View.OnClickListener() {

        @Override
        public void onClick(View arg0){
            db.clearAlarms();

            List<AlarmasModbus> Alarmas = db.getAllAlarmas();

            for (AlarmasModbus cn : Alarmas) {
                String alarm = "Id: " + cn.getID() + " ,RTU: " +
                    cn.getRTU()

```

```

        + " ,Sensor:" + cn.getSensor() + " ,Valormin:" +
            cn.getValormin()
        + " ,Valormax:" + cn.getValormax() + " ,Mail:" +
            cn.getMail()
        + " ,Cel:" + cn.getCel();
    AlarmView.setText("");
    Toast.makeText(getBaseContext(), "Alarmas borradas",
        Toast.LENGTH_SHORT).show();
}
}
};

```

```

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is
    // present.
    getMenuInflater().inflate(R.menu.menu_alarm_config, menu);
    return true;
}

```

```

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();

    //noinspection SimplifiableIfStatement
    if (id == R.id.action_settings) {
        return true;
    }
}

```



```
    }  
    return super.onOptionsItemSelected(item);  
  }  
}
```

1.6. Archivo de *Layout: activity_alarm_config.xml*

En este archivo se muestra el código del *Layout* de la actividad para configurar alarmas.

```
<RelativeLayout

    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"

        android:layout_width="match_parent"
    android:layout_height="match_parent"

        android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    tools:context="com.example.josempd.modbusmaster.AlarmConfig">

    <EditText

        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/etRTU"
        android:layout_alignParentTop="true"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_alignParentRight="true"
        android:layout_alignParentEnd="true"
        android:hint="RTU"
        android:inputType="number" />

    <EditText

        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
android:id="@+id/etSensor"
android:layout_below="@+id/etRTU"
android:layout_alignParentLeft="true"
android:layout_alignParentStart="true"
android:layout_alignRight="@+id/etRTU"
android:layout_alignEnd="@+id/etRTU"
android:inputType="number"
android:hint="Sensor" />
```

<EditText

```
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:id="@+id/etValormin"
android:layout_below="@+id/etSensor"
android:layout_alignParentLeft="true"
android:layout_alignParentStart="true"
android:layout_alignParentRight="true"
android:layout_alignParentEnd="true"
android:inputType="number"
android:hint="Valor mínimo" />
```

<EditText

```
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:id="@+id/etValormax"
android:layout_below="@+id/etValormin"
android:layout_alignParentLeft="true"
android:layout_alignParentStart="true"
android:layout_alignParentRight="true"
android:layout_alignParentEnd="true"
```

```
android:hint="Valor máximo"  
android:inputType="number" />
```

```
<EditText  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:id="@+id/etMail"  
    android:layout_below="@+id/etValormax"  
    android:layout_alignParentLeft="true"  
    android:layout_alignParentStart="true"  
    android:layout_alignParentRight="true"  
    android:layout_alignParentEnd="true"  
    android:hint="E-mail administrador"  
    android:inputType="textEmailAddress" />
```

```
<EditText  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:inputType="phone"  
    android:ems="10"  
    android:id="@+id/etPhone"  
    android:layout_below="@+id/etMail"  
    android:layout_alignParentLeft="true"  
    android:layout_alignParentStart="true"  
    android:layout_alignParentRight="true"  
    android:layout_alignParentEnd="true"  
    android:hint="Celular administrador" />
```

```
<Button  
    android:layout_width="wrap_content"
```

```
android:layout_height="wrap_content"
android:text="Guardar"
android:id="@+id/SaveAlarmButton"
android:layout_below="@+id/etPhone"
android:layout_alignParentLeft="true"
android:layout_alignParentStart="true"
android:layout_alignParentRight="true"
android:layout_alignParentEnd="true" />
```

<TextView

```
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:id="@+id/AlarmView"
android:layout_alignParentLeft="true"
android:layout_alignParentStart="true"
android:layout_alignParentRight="true"
android:layout_alignParentEnd="true"
android:layout_above="@+id/DeleteAlarmButton"
android:layout_below="@+id/tvalarmas" />
```

<Button

```
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Borrar alarmas"
android:id="@+id/DeleteAlarmButton"
android:layout_alignParentBottom="true"
android:layout_alignParentRight="true"
android:layout_alignParentEnd="true"
android:layout_alignParentLeft="true"
android:layout_alignParentStart="true"
```

```
        android:layout_alignWithParentIfMissing="false"  
        android:enabled="true" />
```

```
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Alarmas:"  
    android:id="@+id/tvalarmas"  
    android:layout_below="@+id/SaveAlarmButton"  
    android:layout_centerHorizontal="true"  
    android:maxLines = "8"  
    android:scrollbars = "vertical"/>
```

```
</RelativeLayout>
```

1.7. Archivos para configurar la barra de menú

Estos son los archivos que configuran la *ActionBar* para ir de una actividad a otra

1.7.1. *menu_main.xml*

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:tools="http://schemas.android.com/tools"
      tools:context=".MainActivity">
    <item android:id="@+id/action_settings"
          android:title="@string/action_settings"
          android:orderInCategory="100" app:showAsAction="never" />
    <item android:id="@+id/alarm_settings" android:title="Config. de
          alarmas"
          android:orderInCategory="100" app:showAsAction="never" />
</menu>
```

1.7.2. *menu_settings.xml*

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:tools="http://schemas.android.com/tools"
      tools:context="com.example.josempd.modbusmaster.SettingsActivity">
    <item android:id="@+id/action_settings"
          android:title="@string/action_settings"
          android:orderInCategory="100" app:showAsAction="never" />
</menu>
```

1.7.3. *menu_alarm_config.xml*

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:tools="http://schemas.android.com/tools"
      tools:context="com.example.josempd.modbusmaster.AlarmConfig">
    <item android:id="@+id/action_settings"
          android:title="@string/action_settings"
          android:orderInCategory="100" app:showAsAction="never" />
</menu>
```

1.8. Archivos para la base de datos

1.8.1. *ValorModbus.java*

En este archivo se crea la clase para los objetos que representan las mediciones de Modbus dentro de la base de datos.

```
package com.example.josempd.modbusmaster;

/**
 * Created by Josempd on 15/04/2015.
 */
public class ValorModbus {

    //variables privadas
    int _id;
    int _RTU;
    int _idSensor;
    int _valor;
```



```
// Constructor vacio
public ValorModbus(){

}

// constructor
public ValorModbus(int id, int RTU, int sensor, int valor){
    this._id = id;
    this._RTU = RTU;
    this._idSensor = sensor;
    this._valor = valor;
}

// constructor para valores sin ID
public ValorModbus(int RTU, int sensor, int valor){
    this._RTU = RTU;
    this._idSensor = sensor;
    this._valor = valor;
}

// obteniendo ID
public int getID(){
    return this._id;
}

// colocando ID
public void setID(int id){
    this._id = id;
}

// obteniendo RTU
```

```
public int getRTU(){
    return this._RTU;
}

// colocando RTU
public void setRTU(int RTU){
    this._RTU = RTU;
}

// obteniendo Sensor
public int getSensor(){
    return this._idSensor;
}

// colocando Sensor
public void setSensor(int sensor){
    this._idSensor = sensor;
}

// obteniendo valor
public int getValor(){
    return this._valor;
}

// colocando valor
public void setValor(int valor){
    this._valor = valor;
}
}
```

1.8.2. *AlarmasModbus.java*

En este archivo se crea la clase para los objetos donde se manipulan las alarmas en la base de datos.

```
package com.example.josempd.modbusmaster;

/**
 * Created by Josempd on 23/04/2015.
 */
public class AlarmasModbus {

    //variables privadas
    int _id;
    int _RTU;
    int _idSensor;
    int _valormin;
    int _valormax;
    String _email;
    String _cel;

    // Constructor vacio
    public AlarmasModbus(){

    }

    // constructor
    public AlarmasModbus(int id, int RTU, int sensor, int valormin, int
        valormax,
        String email, String cel){

        this._id = id;
        this._RTU = RTU;
```

```

        this._idSensor = sensor;
        this._valormin = valormin;
        this._valormax = valormax;
        this._email = email;
        this._cel = cel;
    }

    // constructor para valores de alarma sin ID
    public AlarmasModbus(int RTU, int sensor, int valormin, int valormax,
                        String email, String cel){
        this._RTU = RTU;
        this._idSensor = sensor;
        this._valormin = valormin;
        this._valormax = valormax;
        this._email = email;
        this._cel = cel;
    }

    // obteniendo ID
    public int getID(){
        return this._id;
    }

    // colocando ID
    public void setID(int id){
        this._id = id;
    }

    // obteniendo RTU

```

```
public int getRTU(){
    return this._RTU;
}

// colocando RTU
public void setRTU(int RTU){
    this._RTU = RTU;
}

// obteniendo Sensor
public int getSensor(){
    return this._idSensor;
}

// colocando Sensor
public void setSensor(int sensor){
    this._idSensor = sensor;
}

// obteniendo valormin
public int getValormin(){
    return this._valormin;
}

// colocando valormin
public void setValormin(int valor){
    this._valormin = valor;
}

// obteniendo valormax
```

```
public int getValormax(){
    return this._valormax;
}

// colocando valormax
public void setValormax(int valor){
    this._valormax = valor;
}

// obteniendo Mail
public String getMail(){
    return this._email;
}

// colocando Mail
public void setMail(String mail){ this._email = mail; }

// obteniendo Celular
public String getCel(){
    return this._cel;
}

// colocando Celular
public void setCel(String cel){ this._cel = cel; }
}
```

1.8.3. Archivo *DatabaseHandler.java*

```
package com.example.josempd.modbusmaster;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

import java.util.ArrayList;
import java.util.List;

/**
 * Created by Josempd on 15/04/2015.
 */
public class DatabaseHandler extends SQLiteOpenHelper {

    // Todas las variables Static
    // Version de la base de datos
    private static final int DATABASE_VERSION = 1;

    // Nombre de la base de datos
    private static final String DATABASE_NAME = "MonitoreoModbus";

    // Nombre de las tablas
    private static final String TABLE_VALORES = "valores";
    private static final String TABLE_ALARMAS = "alarmas";

    // Nombre de las columnas comunes entre tablas
    private static final String KEY_ID = "id";
```

```

private static final String KEY_RTU = "RTU";
private static final String KEY_SENSOR = "Sensor";

//Columnas exclusivas de la tabla Valores
private static final String KEY_VALOR = "Valor";

//Columnas exclusivas de la tabla Alarmas
private static final String KEY_VALORMIN = "Valormin";
private static final String KEY_VALORMAX = "Valormax";
private static final String KEY_MAIL = "Mail";
private static final String KEY_CEL = "Cel";

// Creando las tablas
//Creando la tabla de valores
private static final String CREATE_TABLE_VALORES = "CREATE TABLE "
    + TABLE_VALORES + "(" + KEY_ID + " INTEGER PRIMARY KEY," +
    KEY_RTU + " INTEGER," + KEY_SENSOR + " INTEGER," + KEY_VALOR +
    " INTEGER" + ")";

//Creando la tabla de alarmas
private static final String CREATE_TABLE_ALARMAS = "CREATE TABLE "
    + TABLE_ALARMAS + "(" + KEY_ID + " INTEGER PRIMARY KEY," +
    KEY_RTU + " INTEGER," + KEY_SENSOR + " INTEGER," +
    KEY_VALORMIN +
    " INTEGER," + KEY_VALORMAX + " INTEGER," + KEY_MAIL + "
    TEXT," +
    KEY_CEL + " TEXT" + ")";

//Constructor de la clase
public DatabaseHandler(Context context) {

```



```

        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    //Crear tablas con las string declaradas previamente
    public void onCreate(SQLiteDatabase db){
        db.execSQL(CREATE_TABLE_VALORES);
        db.execSQL(CREATE_TABLE_ALARMAS);
    }

    // Updateando las tablas
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int
        newVersion) {
        // Si existe una tabla vieja, abandonarla
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_VALORES);
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_ALARMAS);

        // Crear de nuevo la tabla
        onCreate(db);
    }

    /**
     * Operaciones CRUD(Create, Read, Update, Delete)
     */

    // Agregando nuevo valor
    void addValor(ValorModbus Valor) {
        //Se llama la base de datos
        SQLiteDatabase db = this.getWritableDatabase();
        //Se guardan los valores mediante contentValues
    }

```

```

        ContentValues values = new ContentValues();
        values.put(KEY_RTU, Valor.getRTU());
        values.put(KEY_SENSOR, Valor.getSensor());
        values.put(KEY_VALOR, Valor.getValor());

        // Insertando fila
        db.insert(TABLE_VALORES, null, values);
        db.close(); // se cierra la conexion de la db
    }

    //Obteniendo el ultimo valor introducido:
    public String LastValor(){
        //Uso de el cursor en una consulta para devolver el valor
        String selectQuery= "SELECT * FROM " + TABLE_VALORES+" ORDER BY "
            + KEY_ID +
            " DESC LIMIT 1";
        SQLiteDatabase db = this.getWritableDatabase();
        Cursor cursor = db.rawQuery(selectQuery, null);
        String str = "";
        if(cursor.moveToFirst())
            str = cursor.getString( cursor.getColumnIndex(KEY_ID) );
        cursor.close();
        return str;
    }

    //Obteniendo el Valormin de alarma segun valor de RTU y sensor
    public String ValorminValue(int rtu, int sensor){
        String selectQuery= "SELECT " + "*" + " FROM " + TABLE_ALARMAS+"
            WHERE " +
            KEY_RTU + " =? AND " + KEY_SENSOR + " =?";
    }

```

```

        SQLiteDatabase db = this.getWritableDatabase();
        Cursor cursor = db.rawQuery(selectQuery, new String[]
            {String.valueOf(rtu), String.valueOf(sensor)});
        String intv = "";
        if(cursor.moveToFirst())
            intv = cursor.getString(cursor.getColumnIndex(KEY_VALORMIN));
        cursor.close();
        return intv;
    }

```

//Obteniendo el Valormax de alarma segun valor de RTU y sensor

```

public String ValormaxValue(int rtu, int sensor){
    String selectQuery= "SELECT " + "*" + " FROM " + TABLE_ALARMAS+"
        WHERE " +
            KEY_RTU + " =? AND " + KEY_SENSOR + " =?";
    SQLiteDatabase db = this.getWritableDatabase();
    Cursor cursor = db.rawQuery(selectQuery, new String[]
        {String.valueOf(rtu), String.valueOf(sensor)});
    String intv = "";
    if(cursor.moveToFirst())
        intv = cursor.getString( cursor.getColumnIndex(KEY_VALORMAX));
    cursor.close();
    return intv;
}

```

//Obteniendo el mail de alarma segun valor de RTU y sensor

```

public String MailValue(int rtu, int sensor){
    String selectQuery= "SELECT " + "*" + " FROM " + TABLE_ALARMAS+"
        WHERE " +
            KEY_RTU + " =? AND " + KEY_SENSOR + " =?";

```

```

        SQLiteDatabase db = this.getWritableDatabase();
        Cursor cursor = db.rawQuery(selectQuery, new String[]
            {String.valueOf(rtu), String.valueOf(sensor)});
        String intmail = "";
        if(cursor.moveToFirst())
            intmail = cursor.getString(cursor.getColumnIndex(KEY_MAIL));
        cursor.close();
        return intmail;
    }

```

```

//Obteniendo el Celular de alarma segun valor de RTU y sensor
public String CelValue(int rtu, int sensor){
    String selectQuery= "SELECT " + "*" + " FROM " + TABLE_ALARMAS+"
        WHERE " +
            KEY_RTU + " =? AND " + KEY_SENSOR + " =?";
    SQLiteDatabase db = this.getWritableDatabase();
    Cursor cursor = db.rawQuery(selectQuery, new String[]
        {String.valueOf(rtu), String.valueOf(sensor)});
    String intcel = "";
    if(cursor.moveToFirst())
        intcel = cursor.getString(cursor.getColumnIndex(KEY_CEL));
    cursor.close();
    return intcel;
}

```

```

//Borrando un valor
//public void ClearValor(ValorModbus Valor){
//    SQLiteDatabase db = this.getWritableDatabase();

    //db.delete(TABLE_VALORES, KEY_ID + " = ?",

```

```

        //      new String[]{ String.valueOf(Valor.getID())});
    //}

    // Obteniendo todos los valores
    public List<ValorModbus> getAllValores() {
        List<ValorModbus> ValorList = new ArrayList<ValorModbus>();
        // Consultando todos
        String selectQuery = "SELECT * FROM " + TABLE_VALORES;

        SQLiteDatabase db = this.getWritableDatabase();
        Cursor cursor = db.rawQuery(selectQuery, null);

        // loop a traves de todas las filas
        if (cursor.moveToFirst()) {
            do {
                ValorModbus valorModbus = new ValorModbus();
                valorModbus.setID(Integer.parseInt(cursor.getString(0)));
                valorModbus.setRTU(Integer.parseInt(cursor.getString(1)));
                valorModbus.setSensor(Integer.parseInt(cursor.getString(2)));
                valorModbus.setValor(Integer.parseInt(cursor.getString(3)));

                // Anadiendo valores a la lista
                ValorList.add(valorModbus);
            } while (cursor.moveToNext());
        }

        // Return de la lista
        return ValorList;
    }

```

```

// Conteo de valores obtenidos
public int getContactsCount() {
    String countQuery = "SELECT * FROM " + TABLE_VALORES;
    SQLiteDatabase db = this.getReadableDatabase();
    Cursor cursor = db.rawQuery(countQuery, null);
    cursor.close();

    // return del count
    return cursor.getCount();
}

//Creando un valor de alarma
// Agregando nuevo valor
void addAlarma(AlarmasModbus Alarma) {
    SQLiteDatabase db = this.getWritableDatabase();
    //Uso de ContentValues para pasar los valores
    //De la clase creada a la base de datos
    ContentValues values = new ContentValues();
    values.put(KEY_RTU, Alarma.getRTU());
    values.put(KEY_SENSOR, Alarma.getSensor());
    values.put(KEY_VALORMIN, Alarma.getValormin());
    values.put(KEY_VALORMAX, Alarma.getValormax());
    values.put(KEY_MAIL, Alarma.getMail());
    values.put(KEY_CEL, Alarma.getCel());

    // Insertando fila
    db.insert(TABLE_ALARMAS, null, values);
    db.close(); // Closing database connection
}

```

```

//Borrando un valor de alarma
void clearAlarms(){
    SQLiteDatabase db = this.getWritableDatabase();
    db.delete(TABLE_ALARMAS, null, null);
}

//Borrando los valores medidos
void clearValores(){
    SQLiteDatabase db = this.getWritableDatabase();
    db.delete(TABLE_VALORES, null, null);
}

//Escribiendo todos los valores de alarmas
public List<AlarmasModbus> getAllAlarmas() {
    List<AlarmasModbus> ValorList = new ArrayList<AlarmasModbus>();
    // Consultando todos
    String selectQuery = "SELECT * FROM " + TABLE_ALARMAS;

    SQLiteDatabase db = this.getWritableDatabase();
    Cursor cursor = db.rawQuery(selectQuery, null);

    // loop a traves de todas las filas
    if (cursor.moveToFirst()) {
        do {
            AlarmasModbus AlarmasModbus = new AlarmasModbus();
            AlarmasModbus.setID(Integer.parseInt(cursor.getString(0)));
            AlarmasModbus.setRTU(Integer.parseInt(cursor.getString(1)));
            AlarmasModbus.setSensor(Integer.parseInt(cursor.getString(2)));
            AlarmasModbus.setValormin(Integer.parseInt(cursor.getString(3)));
            AlarmasModbus.setValormax(Integer.parseInt(cursor.getString(4)));

```

```

        AlarmasModbus.setMail(cursor.getString(5));
        AlarmasModbus.setCel(cursor.getString(6));

        // Anadiendo valores a la lista
        ValorList.add(AlarmasModbus);
    } while (cursor.moveToNext());
}

// Return de la lista
return ValorList;
}
}

```

1.9. Archivo *Mail.java*

En este archivo se configura el envío de correo electrónico desasistido

```

package com.example.josempd.modbusmaster;

/**
 * Created by Josempd on 29/04/2015.
 */
import java.util.Date;
import java.util.Properties;
import javax.activation.CommandMap;
import javax.activation.DataHandler;
import javax.activation.DataSource;
import javax.activation.FileDataSource;
import javax.activation.MailcapCommandMap;
import javax.mail.BodyPart;

```



```
import javax.mail.Multipart;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeBodyPart;
import javax.mail.internet.MimeMessage;
import javax.mail.internet.MimeMultipart;

public class Mail extends javax.mail.Authenticator {
    private String _user;
    private String _pass;

    private String[] _to;
    private String _from;

    private String _port;
    private String _sport;

    private String _host;

    private String _subject;
    private String _body;

    private boolean _auth;

    private boolean _debuggable;

    private Multipart _multipart;
```

```

public Mail() {
    _host = "smtp.gmail.com"; // default smtp server
    _port = "465"; // default smtp port
    _sport = "465"; // default socketfactory port

    _user = ""; // username
    _pass = ""; // password
    _from = ""; // email sent from
    _subject = ""; // email subject
    _body = ""; // email body

    _debuggable = false; // debug mode on or off - default off
    _auth = true; // smtp authentication - default on

    _multipart = new MimeMultipart();

    // There is something wrong with MailCap, javamail can not find a
    // handler for the multipart/mixed part, so this bit needs to be
    // added.
    MailcapCommandMap mc = (MailcapCommandMap)
        CommandMap.getDefaultCommandMap();
    mc.addMailcap("text/html;;
        x-java-content-handler=com.sun.mail.handlers.text_html");
    mc.addMailcap("text/xml;;
        x-java-content-handler=com.sun.mail.handlers.text_xml");
    mc.addMailcap("text/plain;;
        x-java-content-handler=com.sun.mail.handlers.text_plain");
    mc.addMailcap("multipart/*;;

```

```

        x-java-content-handler=com.sun.mail.handlers.multipart_mixed");
mc.addMailcap("message/rfc822;;
        x-java-content-handler=com.sun.mail.handlers.message_rfc822");
CommandMap.setDefaultCommandMap(mc);
}

public Mail(String user, String pass) {
    this();

    _user = user;
    _pass = pass;
}

public boolean send() throws Exception {
    Properties props = _setProperties();

    if(!_user.equals("") && !_pass.equals("") && _to.length > 0 &&
        !_from.equals("") && !_subject.equals("") &&
        !_body.equals("")) {
        Session session = Session.getInstance(props, this);

        MimeMessage msg = new MimeMessage(session);

        msg.setFrom(new InternetAddress(_from));

        InternetAddress[] addressTo = new InternetAddress[_to.length];
        for (int i = 0; i < _to.length; i++) {
            addressTo[i] = new InternetAddress(_to[i]);
        }
        msg.setRecipients(MimeMessage.RecipientType.TO, addressTo);
    }
}

```

```

        msg.setSubject(_subject);
        msg.setSentDate(new Date());

        // setup message body
        BodyPart messageBodyPart = new MimeBodyPart();
        messageBodyPart.setText(_body);
        _multipart.addBodyPart(messageBodyPart);

        // Put parts in message
        msg.setContent(_multipart);

        // send email
        Transport.send(msg);

        return true;
    } else {
        return false;
    }
}

public void addAttachment(String filename) throws Exception {
    BodyPart messageBodyPart = new MimeBodyPart();
    DataSource source = new FileDataSource(filename);
    messageBodyPart.setDataHandler(new DataHandler(source));
    messageBodyPart.setFileName(filename);

    _multipart.addBodyPart(messageBodyPart);
}

```

```

@Override

public PasswordAuthentication getPasswordAuthentication() {
    return new PasswordAuthentication(_user, _pass);
}

private Properties _setProperties() {
    Properties props = new Properties();

    props.put("mail.smtp.host", _host);

    if(_debuggable) {
        props.put("mail.debug", "true");
    }

    if(_auth) {
        props.put("mail.smtp.auth", "true");
    }

    props.put("mail.smtp.port", _port);
    props.put("mail.smtp.socketFactory.port", _sport);
    props.put("mail.smtp.socketFactory.class",
        "javax.net.ssl.SSLSocketFactory");
    props.put("mail.smtp.socketFactory.fallback", "false");

    return props;
}

// the getters and setters
public String getBody() {
    return _body;
}

```

```
}

public void setBody(String _body) {
    this._body = _body;
}

public void setTo(String[] toArr) {
    this._to = toArr;
}

public void setFrom(String string) {
    this._from = string;
}

public void setSubject(String string) {
    this._subject = string;
}
}
```
