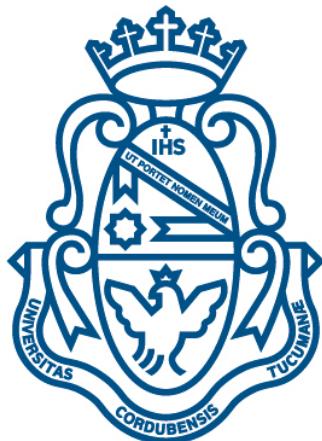


Universidad Nacional de Córdoba



Proyecto Integrador de Ingeniería en Computación

Cliente Android para control asíncrono de dispositivo IoT

Autor:

Esteban Andrés Morales

*Matrícula:*35.104.714

Director Docente:

Mg.Ing. Miguel Solinas

Facultad de Ciencias Exáctas, Físicas y Naturales

Laboratorio de Arquitectura de Redes y Computadoras.

3 de mayo de 2024

Índice general

Contenido	I
Lista de Figuras	IV
Lista de Tablas	VI
1. Introducción	1
1.1. Motivación	1
1.2. Dominio del problema	1
1.3. Objetivos	1
1.3.1. General	1
1.3.2. Objetivos Particulares	1
1.3.3. Objetivos Específicos	1
1.4. Metodología de Trabajo	2
2. Marco Teórico	3
2.1. Cerramientos Eléctricos	3
2.1.1. Principio de Funcionamiento	3
2.1.1.1. Solenoide y Perno	4
2.1.1.2. Solenoide y Clavija	4
2.1.1.3. Electroimán	5
2.1.1.4. Motor Eléctrico	6
2.2. Controladores	6
2.2.1. Sensores	7
2.2.1.1. Sensores de Fin de Carrera	7
2.2.1.2. Encoder Rotativo	8
2.2.2. Monitoreo y Alerta	8
2.2.2.1. Monitoreo Remoto	9
2.2.3. Llaves Electrónicas y Control Remoto	9
2.2.3.1. Llaves RFID	10
2.2.3.2. Control Remoto RF	10
2.2.3.3. Un problema de seguridad	11
2.3. Cerramientos Electrónicos IoT	12
2.3.1. Smart Alarms	12
2.3.2. Smart Door Locks	13
2.3.3. Smart Garage Door	14
2.4. Conclusión	14

3. Análisis de Requerimientos	16
3.1. Dominio	16
3.1.1. Conexión con los controladores	16
3.2. Funcionalidades de un Sistema de Acceso	18
3.3. Casos de Uso	18
3.4. Requerimientos	20
3.4.1. Requerimientos de Sistema	22
3.4.2. Modulo Electrónico	22
3.4.3. Aplicación Móvil	23
3.4.4. Identificación de Riesgos	23
3.4.5. Análisis de Riesgo	24
3.4.6. Planificación de Riesgos	25
4. Diseño	28
4.1. Arquitectura de Software	29
4.1.1. Clean Architecture	29
4.1.1.1. Dominio Transparente	29
4.1.1.2. Regla de Dependencias	30
4.1.1.3. Principio de Abstracción	31
4.1.1.4. Comunicación entre Capas	31
4.2. Diseño de Capas	32
4.2.1. Capa de Presentación: Model View Presenter (MVP)	33
4.2.2. Capa de Dominio: Patrón Command	34
4.2.3. Capa de Datos: Patrón Repository	37
4.3. Programación Reactiva	40
4.4. Interfaz Módulo - Aplicación	41
4.4.1. Protocolos de Comunicación	41
4.4.1.1. MDNS DNS-SD	41
4.4.1.2. HTTP y Digest Authentication	44
4.4.1.3. MQTT	46
4.4.2. RPC API	47
4.5. Estructura del Proyecto	50
4.5.1. Unidades Funcionales	50
4.5.1.1. Plantillas de Arquitectura	51
4.5.2. Planificación de las Iteraciones	52
5. Desarrollo: Iteración I	54
5.1. Introducción	54
5.2. Requerimientos	54
5.2.1. Casos de Uso	54
5.2.1.1. Configuración Inicial	54
5.2.1.2. Accionamiento	54
5.3. Validación	54
5.3.1. Pruebas Unitarias	54
5.3.2. Pruebas de Sistema	54
5.4. Conclusión	54

6. Desarrollo: Iteración II	55
6.1. Introducción	55
6.2. Requerimientos	55
6.2.1. Casos de Uso	55
6.2.1.1. Creación de Usuario	55
6.2.1.2. Ver listado de Usuarios	55
6.2.1.3. Editado de Usuario	55
6.2.1.4. Eliminación de Usuario	55
6.3. Validación	55
6.3.1. Pruebas Unitarias	55
6.3.2. Pruebas de Sistema	55
6.4. Conclusión	55
7. Desarrollo: Iteración III	56
7.1. Introducción	56
7.2. Requerimientos	56
7.2.1. Casos de Uso	56
7.2.1.1. Estado de Apertura	56
7.2.1.2. Notificaciones	56
7.2.1.3. Actualización Inalámbrica (OTA)	56
7.3. Validación	56
7.3.1. Pruebas Unitarias	56
7.3.2. Pruebas de Sistema	56
7.4. Conclusión	56
8. Conclusiones y Trabajos Futuros	57
8.1. Conclusión	57
8.2. Trabajos Futuros	58
Bibliografía	59

Índice de figuras

2.1. Cerramientos de perno	4
2.2. Cerramientos de perno	5
2.3. Cerradura Electroimán	5
2.4. Motor Portones Automatizados	6
2.5. Controladores de Acceso	7
2.6. Sensores de fin de carrera	8
2.7. Encoder Rotativo	9
2.8. Llaves RFID	10
2.9. Controles RF	11
2.10Smart Alarms	13
2.11Smart Locks	13
2.12Smart Garage Gates	14
3.1. Controladoras Comparativa	17
3.2. Diagrama Bloques Controlador	19
3.3. Diagrama Bloques Internos Controlador	20
3.4. Diagrama de Componentes Conexión al Controlador	20
3.5. Diagrama de Casos de Uso	21
3.6. Diagrama de Casos de Uso	21
3.7. Diagrama de Casos de Uso	22
3.8. Diagrama de Casos de Uso	22
4.1. Principio de Dependencias	30
4.2. Abstraction Principle	31
4.3. Layer Communication	32
4.4. Dependencia de Módulos	33
4.5. MVP Components	34
4.6. MVP Sequence	35
4.7. Commander Classes	35
4.8. MVP Components	36
4.9. Commander Review	37
4.10Repository Pattern Class Diagram	38
4.11Repository Pattern Detailed Class Diagram	39
4.12Modified Repository Pattern Class Diagram	40
4.13Connection Diagram	42
4.14Connection Diagram	42
4.15Connection Diagram	44
4.16Connection Diagram	45

4.17 Connection Diagram	46
4.18 Connection Diagram	47
4.19 Connection Diagram	50
4.20 Connection Diagram	51
4.21 Diagrama de clases de una unidad funcional.	52
4.22 Subconjuntos de Casos de Uso para cada Iteración.	53

Índice de cuadros

3.1. Requerimientos de Sistema	23
3.2. Requerimientos del Módulo Electrónico	24
3.3. Requerimientos de la Aplicación	25
3.4. Riesgos Identificados	26
3.5. Planificación de Riesgos	27
4.1. Protocolos remotos proporcionados por el firmware del modulo	49

Capítulo 1

Introducción

1.1. Motivación

sadasasd

1.2. Dominio del problema

asdasd

1.3. Objetivos

asdasd

1.3.1. General

asdasd

1.3.2. Objetivos Particulares

asdasd

1.3.3. Objetivos Específicos

asdasd

1.4. Metodología de Trabajo

asdasd

Capítulo 2

Marco Teórico

2.1. Cerramientos Eléctricos

Cuando hablamos de cerramientos eléctricos se hace referencia a todos los dispositivos de funcionamiento electromecánico destinados a ofrecer la función de cerradura para aperturas de instalaciones domiciliarias, comerciales e industriales. Se listan a continuación ejemplos de estos dispositivos.

- Cerraduras por electro-imán.
- Cerraduras de perno.
- Portones automatizados.
- Barreras automáticas.
- Pasadores eléctricos.
- Traba-pestillo eléctrico.
- Pestillo eléctrico para cajones o guarda equipajes.
- Puertas de Ascensor con control de acceso.

2.1.1. Principio de Funcionamiento

Aunque todos los cerramientos eléctricos comparten la misma funcionalidad existe una variedad acotada de principios de funcionamiento, diseño industrial o factor de forma entre los cuales podemos mencionar:

- Solenoide y Perno

- Solenoide y Clavija
- Electro-imán
- Motor Eléctrico

2.1.1.1. Solenoide y Perno

Un solenoide es enrollado alrededor de un eje cilíndrico dieléctrico hueco que a su vez envuelve un perno también cilíndrico pero metálico. Este solenoide está conectado en un circuito de corriente continua. Cuando se alimenta dicho circuito el campo magnético inducido en el núcleo del solenoide genera corrientes de Foucault sobre el perno que es desplazado por la fuerza del campo magnético, generando así la acción de traba o bloqueo sobre el herraje de la abertura. Este tipo de principio de funcionamiento es el que emplean dispositivos tales como los pestillos eléctricos y las cerraduras de perno que se muestran en la figura 2.1.

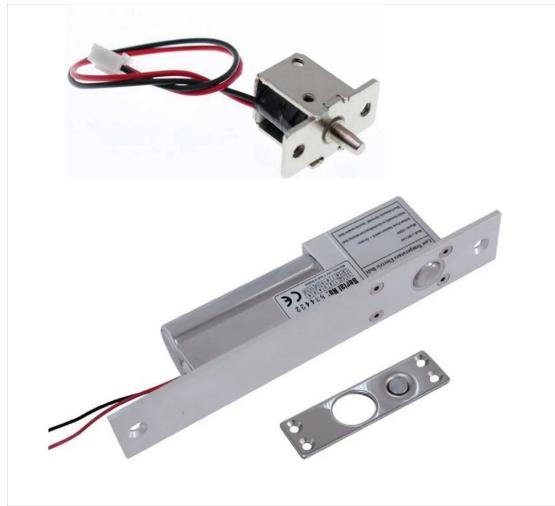


Figura 2.1: Se muestran ejemplos de cerramientos eléctricos que funcionan con solenoide y perno.

2.1.1.2. Solenoide y Clavija

De una forma similar a la descripta para el caso del perno, un solenoide se utiliza para formar un pequeño electroimán en el interior del dispositivo. La fuerza inducida deforma de manera elástica una clavija o chapa metálica delgada en forma de ele. Esta deformación libera el mecanismo que bloquea el movimiento del pestillo. Este es el caso del traba-pestillo eléctrico o del pasador eléctrico cuyos ejemplos se muestran en la figura 2.2.



Figura 2.2: Se muestran ejemplos de cerramientos eléctricos que funcionan con solenoide y clavija.

2.1.1.3. Electroimán

En esta forma de funcionamiento se construye un electro-imán de dimensiones considerables capaz de ejercer una fuerza de atracción magnética cercana a los 300 kgF sobre una chapa metálica que también se incluye como parte del cerramiento. Por lo general incluyen en su circuito una etapa de compensación de factor de potencia para remanencia cero. Este es el caso de la cerradura por electro-imán que se muestra en la figura 2.3.



Figura 2.3: Se muestran un ejemplo de cerradura eléctrica que funcionan por electro-imán.

2.1.1.4. Motor Eléctrico

Para el caso de grandes portones de garaje o acceso vehicular se utilizan motores de corriente alterna o brushless de corriente continua con su respectivos controladores. Estos motores interactúan mecánicamente con otras interfaces mecánicas instaladas en las aperturas para posibilitar la tarea de apertura o cierre. Adicionalmente es necesario la incorporación de sensores que indiquen el estado del sistema y aseguran su correcto funcionamiento. Todos los tipos de automatización de portones así como los sistemas de apertura con cortinas metálicas comparten el mismo principio de funcionamiento y utilizan motores similares al que se muestra en la figura 2.4.



Figura 2.4: Motor empleado en sistemas de automatización de portones de garaje o accesos vehiculares.

2.2. Controladores

Todos los cerramientos eléctricos mencionados anteriormente no funcionan por sí mismos sino que requieren de un dispositivo controlador. Algunos ejemplos se muestran en la figura 2.5.

Estos controladores son circuitos electrónicos que se encargan de generar las señales de activación y/o control de los cerramientos eléctricos. El usuario podrá actuar sobre el cerramiento eléctrico a través de diversos modos de accionamiento, se mencionan los más habituales:

- Portero telefónico
- Contraseña por teclado
- Lector biométrico

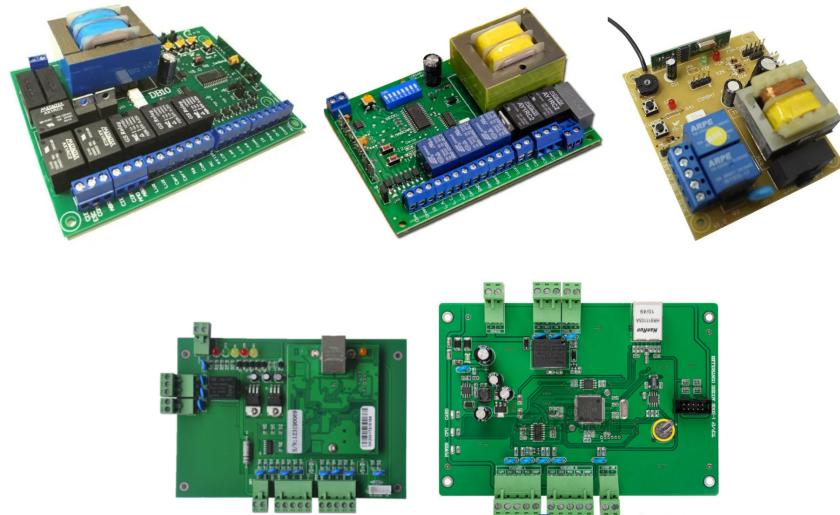


Figura 2.5: Algunos ejemplos de controladores de cerramientos eléctricos.

- Llave electrónica
- Botón externo

Para dar soporte a una o múltiples formas de accionamiento el controlador ofrece una interfaz de configuración y los componentes necesarios para su operación. En el caso de los cerramientos a motor es necesario incluir sensores de apertura y funcionamiento.

2.2.1. Sensores

Por lo general solo son empleados por los controladores para cerramientos a motor. Existen varios tipos de sensores para estos controladores se mencionan los más utilizados:

- Sensores de Fin de Carrera
- Encoder Digital

2.2.1.1. Sensores de Fin de Carrera

Se utiliza el plural para este caso de sensado porque se utilizan de a pares. Se trata de un par de sensores binarios que indican apertura o cierre total del cerramiento.

Suelen venir en dos presentaciones con modo de funcionamiento distinto.

Por proximidad magnética: En este caso cada uno de los sensores está compuesto por un imán permanente y un switch encapsulado como se muestra en la figura 2.6(a). Este encapsulado contiene en su interior un filamento de material ferromagnético flexible que al aproximarse lo suficiente al imán se desplaza y cierra el circuito.

Por choque mecánico: Estos sensores son switches mecánicos muy sensibles que se instalan en los límites del marco de la abertura y al mínimo contacto con la parte móvil cierran sus circuitos. En la figura 2.6(b) se muestran ejemplos de estos dispositivos.

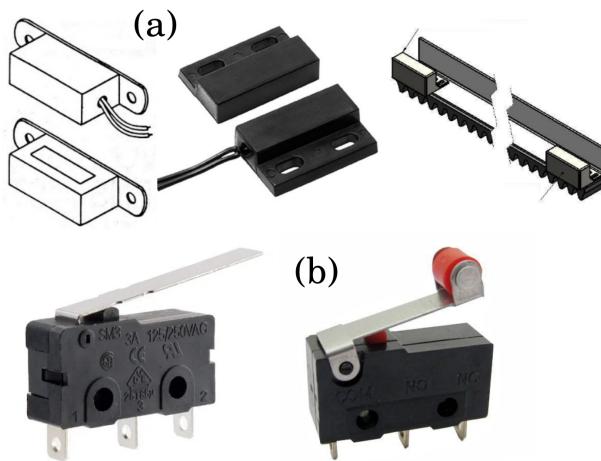


Figura 2.6: Ejemplos de sensores de fin de carrera.(a) Sensores por proximidad magnética, (b) Sensores por choque mecánico.

2.2.1.2. Encoder Rotativo

Un encoder es un sensor de movimiento mecánico que genera señales digitales en respuesta al movimiento y puede proveer información sobre la posición, la velocidad y la dirección del movimiento. En particular el encoder rotativo responde al movimiento de rotación de un eje. Por lo general los controladores de cerramientos eléctricos a motor utilizan encoders rotativos incrementales que generan un tren de pulsos que se puede utilizar para determinar la posición y la velocidad del eje del motor. En la figura 2.7 se observa la forma del tren de pulsos que genera el encoder.

2.2.2. Monitoreo y Alerta

La mayoría de los controladores poseen interfaces que permiten la conexión local de luminarias testigo, así como bocinas de alerta. Algunos modelos incluyen un

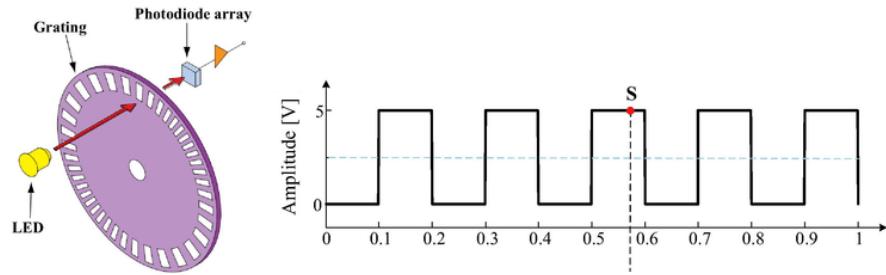


Figura 2.7: Ilustración simplificada de un encoder rotativo con salida de onda cuadrada TTL.

puerto serie (RJ45, RJ14) capaz de comunicarse mediante comandos estándar con los tradicionales sistemas de alarma.

2.2.2.1. Monitoreo Remoto

Al momento de iniciar el presente proyecto integrador no existían controladoras de acceso con la funcionalidad de monitoreo remoto incorporada de fábrica. La alternativa para tener información del cerramiento eléctrico y el estado de la abertura que opera consiste en instalar un sensor de apertura y conectarlo a una central de alarma. Estos sensores son baratos y su principio de funcionamiento coincide con el descripto para los sensores de fin de carrera y que se puede consultar en la sección [2.2.1.1](#). Las centrales de alarma suelen estar conectadas por cableado telefónico a un servicio de monitoreo que lanza las notificaciones pertinentes según se hayan configurado. Una basta mayoría de proveedoras de servicio de monitoreo de alarmas ahora ofrecen plataformas web o aplicaciones móviles que permiten el monitoreo de las centrales en tiempo real.

2.2.3. Llaves Electrónicas y Control Remoto

Actualmente la mayoría de los controladores de acceso soportan algún tipo de llave electrónica. Una llave electrónica es un dispositivo electrónico que le permite al usuario operar el cerramiento eléctrico. Existen diversos tipos de llaves electrónicas a continuación se mencionan las más utilizadas.

- Llaves RFID
- Control Remoto RF

2.2.3.1. Llaves RFID

RFID (Radio-frequency identification) es un sistema de almacenamiento y recuperación de datos a distancias cortas por emisión de radiofrecuencia de baja potencia. Una etiqueta RFID consiste en un pequeño circuito receptor y transmisor de radio. Cuando es activada por un pulso de interrogación electromagnético emitido por un dispositivo lector RFID cercano, la etiqueta transmite datos digitales de vuelta al lector, generalmente un número identificador. Estas etiquetas o circuitos RFID son tan diminutos que pueden incluirse en objetos de tamaño reducido como llaveros y tarjetas, algunos ejemplos se muestran en la figura 2.8. El usuario del sistema de acceso aproximará esta llave al lector RFID incluido en el controlador y se producirá el intercambio del número identificatorio. En caso de que ese dato esté registrado en el controlador el usuario podrá controlar el cerramiento eléctrico.



Figura 2.8: Llaveros y tarjetas con etiquetas RFID.

2.2.3.2. Control Remoto RF

RF es el acrónimo de Radio-Frequency y hace referencia a las tecnologías que emplean sistemas de comunicación por emisión de ondas electromagnéticas. Por lo general se utilizan anchos de bandas libres sin restricciones gubernamentales con baja potencia de transmisión. En el caso del control remoto para por RF un código numérico identificatorio se transmite al controlador que incluye un circuito receptor comúnmente en las frecuencias que van de los 200 MHz a los 500 Mhz. El rango de acción de estos dispositivos es de aproximadamente 100 mts en línea de visión. El controlador de acceso permite la actualización del código identificatorio en caso de que ya esté empleado al momento de la configuración. En algunas implementaciones este código es rotatorio y cambia con el uso. En la figura 2.9 se muestran algunos ejemplos.



Figura 2.9: Controles remotos RF en diferentes presentaciones.

2.2.3.3. Un problema de seguridad

Tanto el empleo de llaves RFID como controles RF presentan un problema de seguridad para estos sistemas de acceso electrónico ya que conceden acceso a quien porta llave sin importar si se trata del dueño real. En ningún caso se utilizan protocolos para la identificación del solicitante. Así mismo tampoco se emplean algoritmos de encripción de datos por lo que el código de identificación se transmite en texto plano, por lo menos este es el caso para la mayoría de las implementaciones.

Por esta razón existen métodos de hacking ya documentados que involucran dos técnicas principales.

Clonado por lectura RFID: En este caso simplemente basta con tener un lector RF y acceso a la etiqueta RFID empleada por la llave a ser clonada. Una vez leído el código se crea una etiqueta con el mismo código y se obtiene acceso.

Clonado por intercepción RF: Ésta técnica implica un ataque por sniffing del código de identificación. Teniendo en cuenta que la frecuencia y el tipo de modulación empleado por los sistemas actuales son conocidos, resulta en una tarea sencilla empleando un transeptor RF y un decodificador por software.

Inhibición de Señal: Esta técnica es quizás la mas sencilla de todas. Se utiliza para vulnerar los sistemas de acceso que emplean controles remotos RF. Implica la utilización de un transmisor que contamina con ruido el ancho de banda de operación del control remoto. De esta forma se impide la correcta comunicación con el controlador del cerramiento al momento de enviar la señal de cierre, evitando así que la abertura pueda ser cerrada.

2.3. Cerramientos Electrónicos IoT

IoT (Internet of Things) describe la red de objetos físicos que incorporan circuitos con microcontroladores, software embebido, sensores y componentes de comunicación con el propósito de intercambiar datos con otros objetos conectados y servicios en linea a través de internet. Teniendo en cuenta que las aberturas de un edificio son objetos físicos y que los cerramientos eléctricos ofrecen un modo de accionamiento electrónico para estos objetos, es posible concebir la adaptación de los mismos para convertirlos en implementaciones IoT. Con el ánimo de ilustrar el panorama de productos IoT orientados a la seguridad y al control de acceso se mencionan las categorías más relevantes.

- Smart Alarms
- Smart Door Locks
- Smart Garage Doors

2.3.1. Smart Alarms

Estos productos proponen la instalación y configuración de una red de sensores (comúnmente inalámbricos) en el interior y alrededor del hogar o cualquier edificio en general. Algunos ejemplos de estos sensores se listan a continuación:

- Sensores de Humo
- Sensores de gases peligrosos (CO₂, CO, propano, butano, metano).
- Sensores de presencia IR.
- Sensores de apertura para puertas y ventanas.

Estos sensores son sistemas embebidos de tamaño reducido con poca capacidad de cómputo y de muy bajo consumo eléctrico incluso en algunas versiones alimentados a baterías. Por esta razón se emplean protocolos de comunicación acorde a las capacidades de estos dispositivos tales como *zigbee* y *Thread* ambos basados en la especificación IEEE 802.15.4 por lo que sus respectivos stacks de software presentan footprints de memoria adecuados para la aplicación en particular.

Como parte fundamental de la red se incluye un dispositivo electrónico denominado *concentrador*, *gateway* ó *central* que posee al menos dos interfaces de comunicación principales, una para conectarse a la red de sensores y otra para acceder a internet. Este "gateway" recibe las actualizaciones de estado de los sensores y las reenvía por internet a un servicio de monitoreo mantenido por el mismo proveedor.

El usuario puede acceder a los datos almacenados por el servicio a través de un cliente web o aplicación móvil. Adicionalmente estos clientes ofrecen una GUI para la configuración tanto de la central como para los sensores. En la figura 2.10 se muestran ejemplos de los productos ofrecidos por los fabricantes más populares.

Honeywell



Figura 2.10: Familia de productos de los fabricantes mas conocidos.

2.3.2. Smart Door Locks

Esta categoría de productos comprenden dispositivos electromecánicos que reemplazan o se adaptan a las tradicionales cerraduras de tambor radial comúnmente utilizadas en puertas residenciales estadounidenses. Estos dispositivos incluyen el sistema embebido capaz de conectarse a internet a través de WiFi o Bluetooth (mediante el uso de un gateway WiFi como en el caso de SESAME). Para todos los casos se ofrece una aplicación móvil que hace de llave virtual y permite el acceso a estos cerramientos inteligentes. Adicionalmente el usuario administrador podrá agregar y remover usuarios autorizados a operar el dispositivo. En la figura 2.11 se pueden observar ejemplos reales de productos disponibles en el mercado.



Figura 2.11: Modelos más comercializados de cerraduras inteligentes.

2.3.3. Smart Garage Door

Para el caso de los portones automatizados de garaje existen diversas implementaciones que agregan estos objetos a la familia de productos IoT.

La mayoría de los controladores ofrecen al menos una interfaz para conexión de un botón externo de apertura y cierre. Por esta razón diversas startups desarrollaron implementaciones de dispositivos a modo de accesorios compatibles con una familia de controladores. Este es el caso de ISmartGate, Nexx NXG-100 y Garadget.

Luego los fabricantes más importantes de controladores y motores para portones de garaje de estados unidos realizaron sus propias implementaciones y las incorporaron a sus controladores como una funcionalidad built-in. Este es el caso de la empresa Genie con su linea de productos Aladdin. Por otro lado la empresa Chamberlane optó por el enfoque de agregar la misma funcionalidad mediante el uso de accesorios que ofrecen con su línea myQ.

Mas recientemente, en el transcurso del año 2019 PPA, la empresa más grande de Sudamérica especializada en automatización de aberturas residenciales e industriales. Presentó un gateway IoT (SPIRIT) compatible con una amplia familia de sensores y actuadores que también fabrican. Entre ellos un módulo que permite el accionamiento y monitoreo remoto de portones automatizados. Todas las implementaciones ofrecen una aplicación móvil que se usará para el monitoreo, control y configuración del producto. En la figura 2.12 se muestran ejemplos de estos productos.



Figura 2.12: Modelos más comercializados de automatización inteligente para portones de garaje.

2.4. Conclusión

Los cerramientos eléctricos presentan una interesante gama de dispositivos para los que pueden plantearse la extensión de funcionalidad con el objetivo de convertirlos en productos IoT. Algunas de las funcionalidades que se entienden deseables para este tipo de dispositivos son el monitoreo remoto, el control a distancia y la

configuración remota.

Adicionalmente aparece una oportunidad de mejora al plantear los problemas de seguridad que tienen las implementaciones actuales y que podrían ser mitigados al utilizar protocolos de comunicación modernos.

En este capítulo se intentó mostrar el panorama actual de los cerramientos eléctricos y sus controladores tradicionales. También se mencionaron desarrollos IoT para algunos tipos de cerramientos. Cualquiera de las alternativas actuales ofrecen una aplicación móvil para el control y la configuración de estos productos. Se hace necesario entonces mencionar algunos detalles sobre el desarrollo de aplicaciones móviles modernas en cuanto a las tecnologías empleadas y los modos de implementación.

En este sentido la vasta mayoría todos los equipos de desarrollo optan por el uso de patrones de arquitectura con el fin de facilitar la colaboración y favorecer la mantenibilidad del código. Las alternativas de patrones de arquitectura y diseño fueron previamente estudiadas como parte de mi práctica profesional supervisada.

Con esto en mente y teniendo en cuenta que los pormenores de estos tópicos son extensos e independientes se desarrollarán con amplitud en el siguiente capítulo.

Capítulo 3

Análisis de Requerimientos

El presente capítulo tiene como objetivo principal encontrar el factor común de diseño de los sistemas de acceso que se ofrecen en el mercado con el propósito de elucidar el conjunto mínimo de funcionalidades que debería cumplir una implementación moderna e incluir las mejoras de la propuesta de valor. Los resultados serán formalizados en diagramas de caso de uso y la especificación de los requerimientos de software.

Finalmente, se expondrán los riesgos asociados con los requerimientos que podrían entorpecer el desarrollo del proceso de software y los posibles caminos para eludir estos obstáculos.

3.1. Dominio

Como se detalló en la Sección [2.2](#) todos los cerramientos eléctricos no funcionan por sí solos sino que necesitan de un controlador. Estos controladores se comercializan en diferentes presentaciones con distintos diseños, desde simples interfaces eléctricas para un botón pulsador externo como se muestra en la Figura [3.1\(a\)](#) hasta complejas centrales de acceso electrónico como la que se muestra en la Figura [3.1\(b\)](#) que incluyen interfaces para sistemas de identificación de usuarios o soporte para llaves electrónicas como las descritas en la Sección [2.2.3](#).

3.1.1. Conexión con los controladores

Del análisis y la comparación de los distintos tipos de controladores se pudo inferir el diseño general de estos sistemas así como sus componentes principales y accesorios que se listan a continuación:

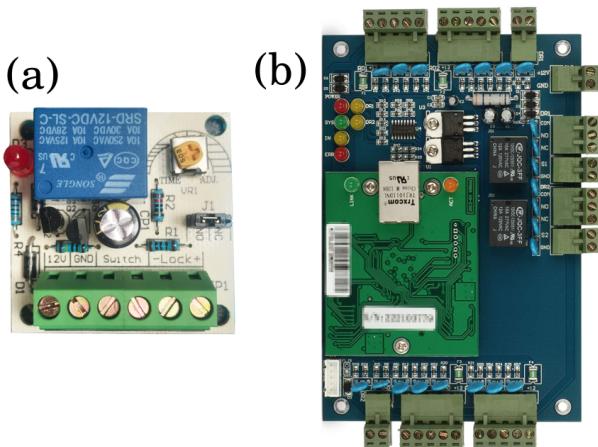


Figura 3.1: Controladoras de cerramientos eléctrico, a la izquierda un simple adaptador de voltaje a la derecha una central de acceso con múltiples puertos.

PRINCIPALES

- Etapa de Potencia y Adaptación de Voltajes
- Driver Señal de Cerramiento
- Borneras para Cerramientos
- Bornera Pulsador Externo
- Microcontrolador y Firmware

ACCESORIOS

- Bornera Luz Testigo Externa
- Bornera Barrera Infrarroja
- Bocina
- LEDs de Funcionamiento
- Pantalla
- Teclado
- Comutadores Deslizantes de Configuración (DIP Switch)
- Jumpers de Configuración
- Pulsadores de Configuración

- Sensores Funcionamiento y Estado
- Módulo portero telefónico o Intercom
- Módulo RX RF para control remoto
- Lector etiquetas RFID
- Sensor Huella dactilar

En la Figura 3.2 se puede observar un diagrama de definición por bloques del diseño generalizado para las controladoras de cerramientos eléctricos. Se resalta en violeta el componente correspondiente al pulsador físico externo.

En el diagrama de bloques interno del controlador de la Figura 3.3 la señal de activación del pulsador es procesada por el módulo disparador de acciones que, en la mayoría de los casos, genera la señal de apertura o cierre según sea el estado del cerramiento.

Después de realizar el análisis de los diseños se pudo evidenciar que la bornera para pulsador o botón externo es un factor común entre todos los modelos. Teniendo en cuenta esta particularidad se decidió adoptar esa interfaz de accionamiento como el modo de conexión universal de la solución a ser implementada. En la Figura 3.4 se muestra un diagrama de componentes de alto nivel dónde se ilustra tal decisión.

3.2. Funcionalidades de un Sistema de Acceso

Estos sistemas ofrecen la posibilidad de limitar el acceso a ciertas áreas de un inmueble mediante el control de cerramientos eléctricos utilizando llaves electrónicas u otros modos de autenticación de usuarios como sensores biométricos o el ingreso de credenciales por teclados. Para poder llevar a cabo esta tarea es primordial que estos controladores implementen una interfaz de configuración que permita entre otras operaciones:

- Permitir la configuración del sistema utilizando una clave maestra.
- Dar de alta llaves electrónicas, nuevas credenciales biométricas o por teclado.
- Dar de baja llaves electrónicas, credenciales biométricas o por teclado.
- Resetear el sistema a sus valores de fábrica.

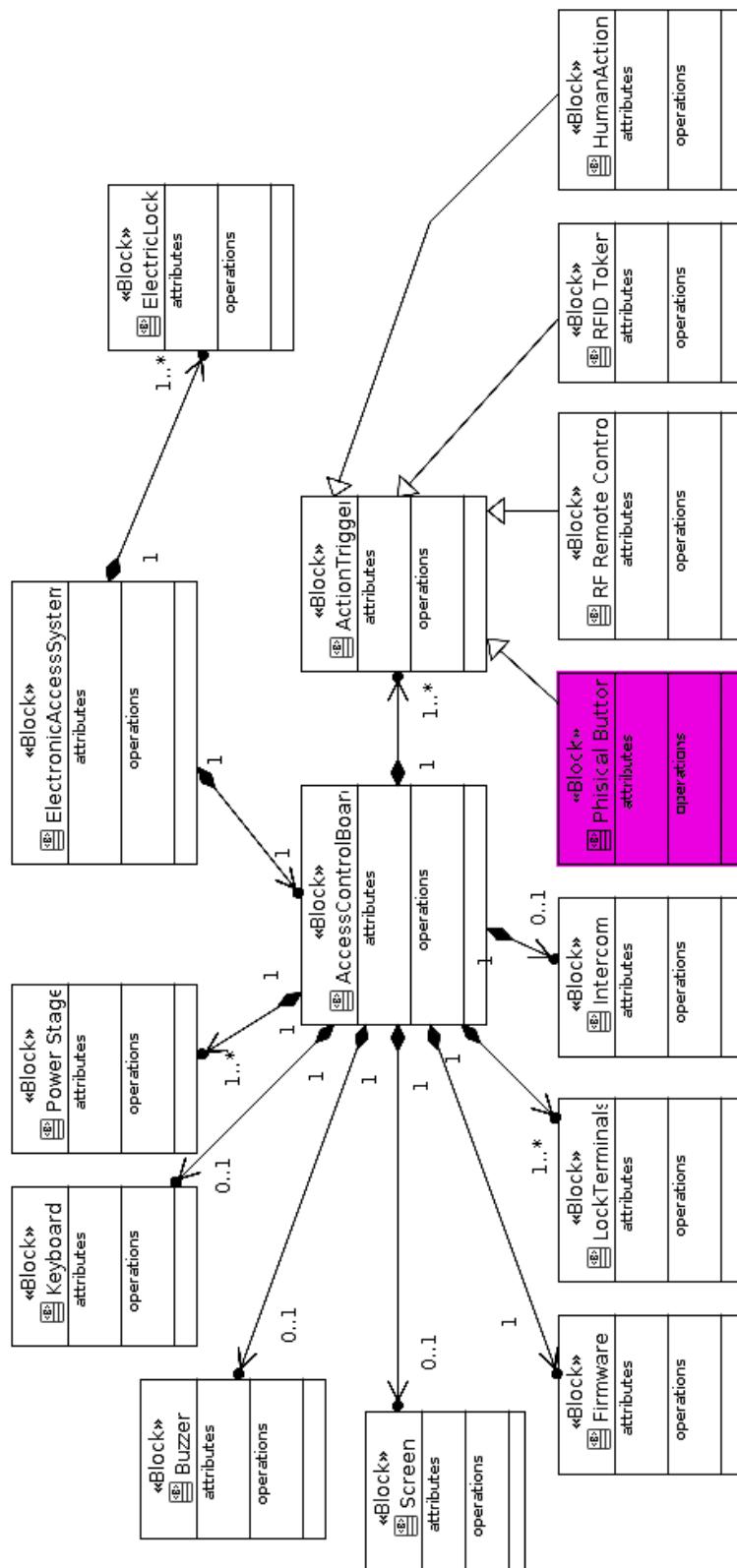


Figura 3.2: Diagrama de definición por bloques para la generalización del diseño de los controladores.

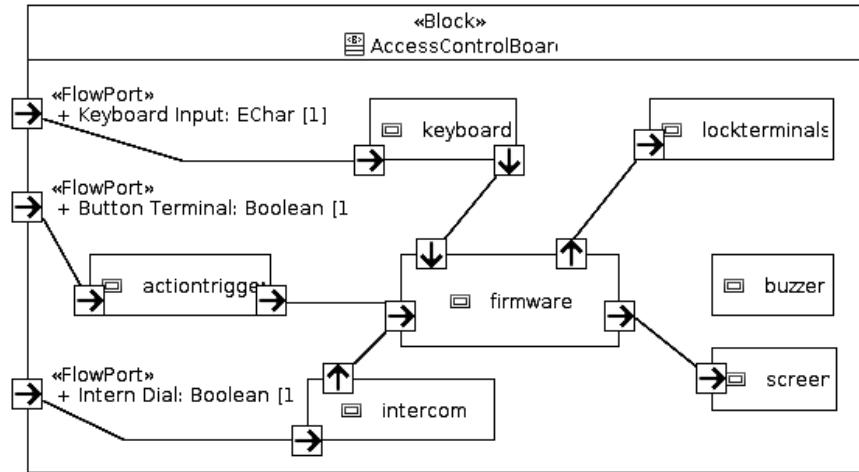


Figura 3.3: Diagrama de bloques internos del diseño generalizado de un controlador de cerramientos eléctricos.

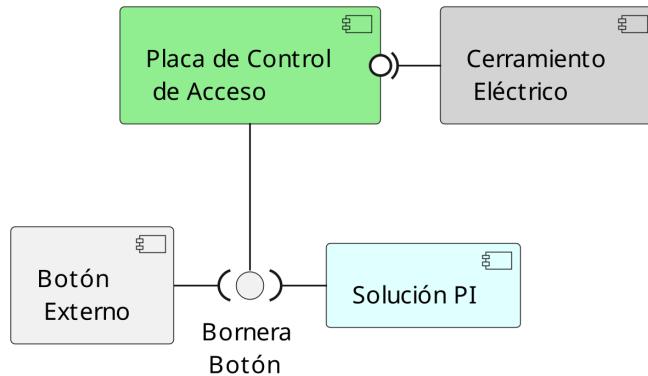


Figura 3.4: Diagrama de componentes de alto nivel muestra la conexión propuesta para la solución de hardware.

3.3. Casos de Uso

Teniendo en cuenta las funcionalidades básicas de un sistema de acceso tradicional y su adaptación a la propuesta de valor del producto se plantean los actores y casos de uso. Se identifican 5 actores que intervienen en el uso del producto: el *Usuario* y sus 3 subtipos: *Administrador*, *Autorizado* e *Invitado* y el *Módulo Electrónico* que ejecutará los comandos recibidos previa autenticación y autorización. En las figuras 3.5, 3.6, 3.7 y 3.8 se muestran los diagramas de casos de uso separados en paquetes para facilitar su lectura.

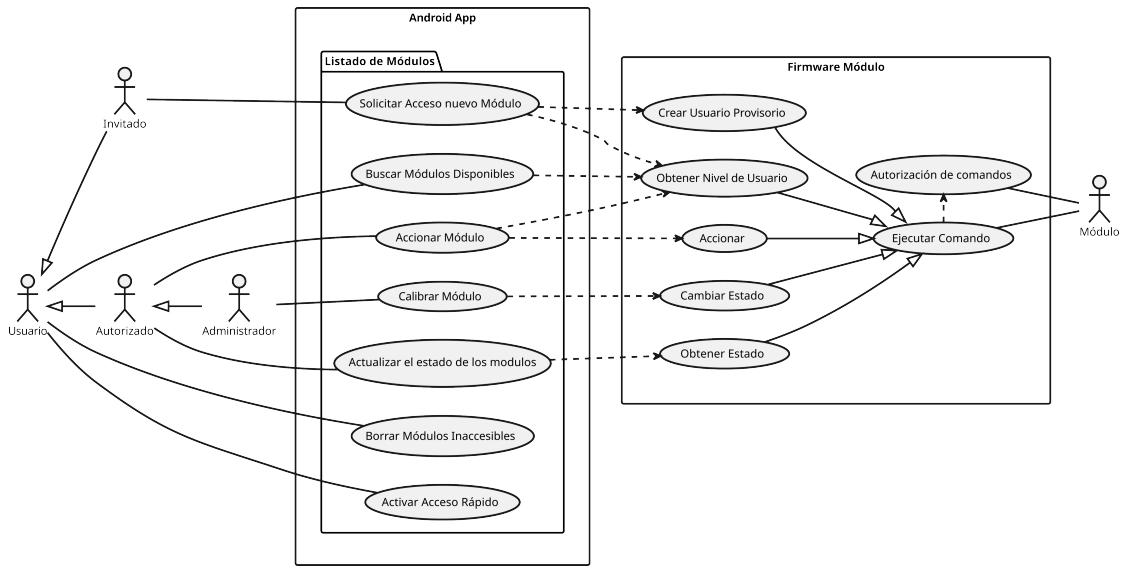


Figura 3.5: Diagrama de casos de uso relacionados con el descubrimiento y control de módulos.

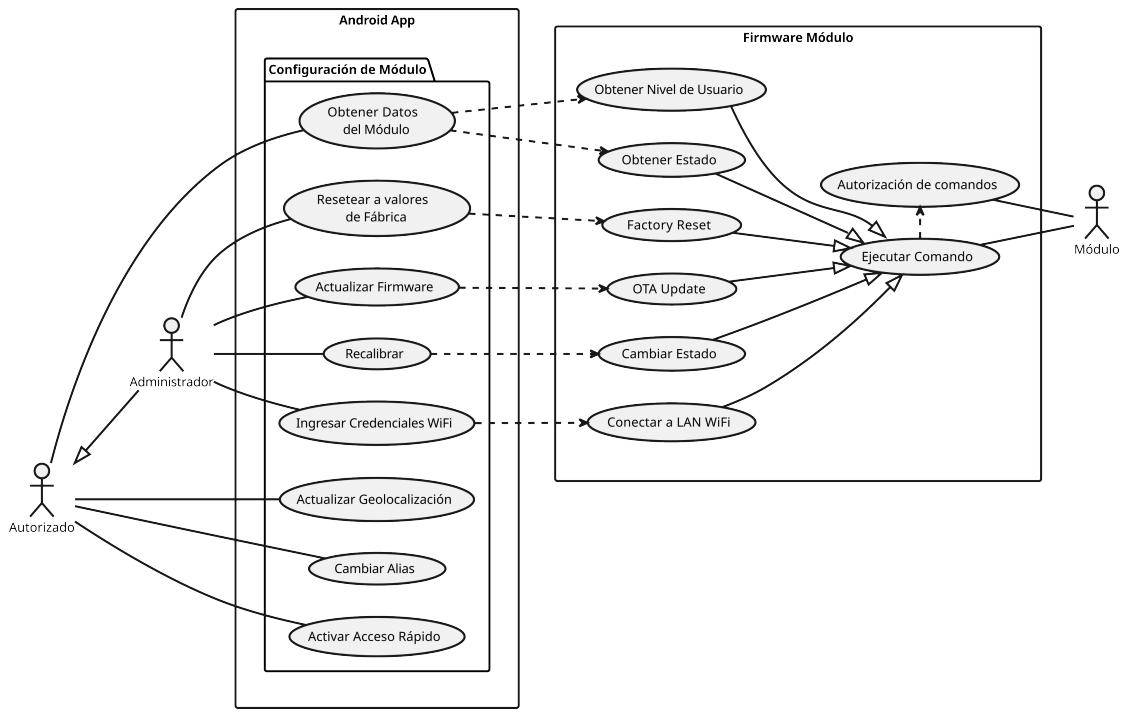


Figura 3.6: Diagrama de casos de uso relacionados con la configuración de un módulo.

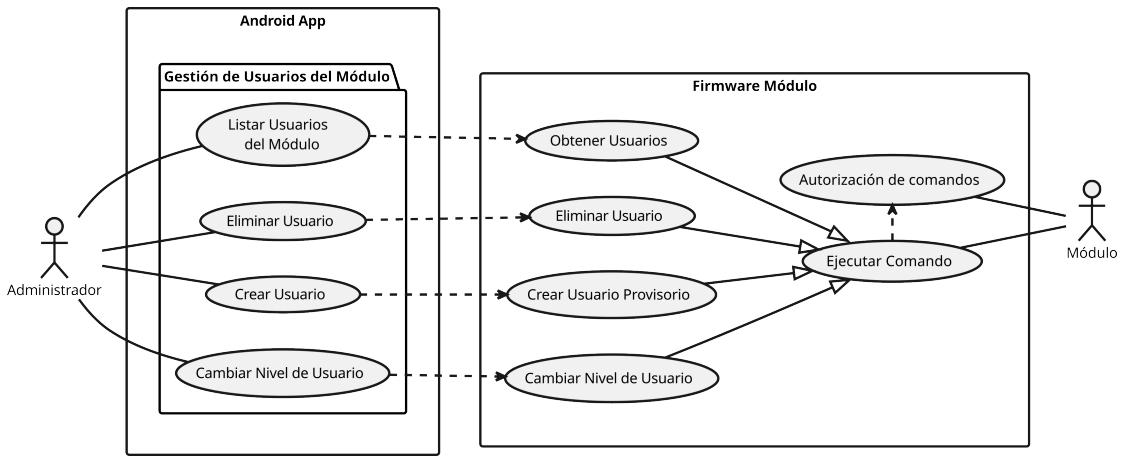


Figura 3.7: Diagrama de casos de uso relacionados con el descubrimiento y control de módulos.

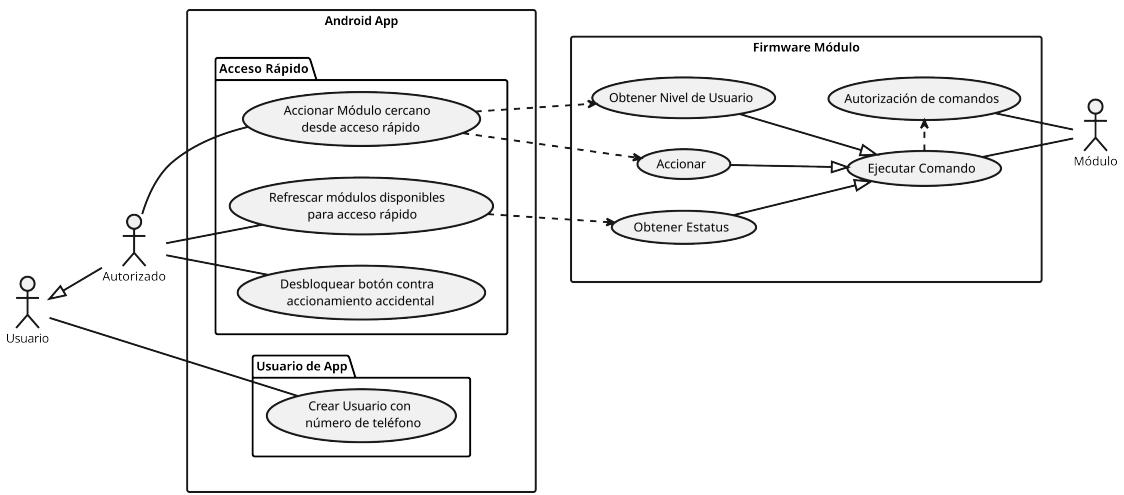


Figura 3.8: Diagrama de casos de uso relacionados con el control rápido y el registro del usuario de la aplicación.

3.4. Requerimientos

Considerando los casos de uso y el modo de conexión universal elegido se listan a continuación los requerimientos a nivel sistema de la solución propuesta.

3.4.1. Requerimientos de Sistema

A nivel de sistema se listan los requerimientos identificados en la Tabla 3.1.

ID	Descripción
RS00	El sistema debe accionar cerramientos eléctricos.
RS01	El sistema debe permitir que un mismo usuario tenga acceso a múltiples cerramientos.
RS02	El sistema debe establecer conexión a la red wifi del inmueble donde será instalado.
RS03	El sistema debe mantenerse operativo offline de manera local.
RS04	El sistema debe funcionar de manera remota a través de internet.
RS05	El sistema debe admitir alta de nuevos usuarios para un cerramiento.
RS06	El sistema debe permitir la baja de usuarios.
RS07	El sistema debe permitir el cambio de privilegios de los usuarios.
RS08	El sistema debe admitir la conexión y calibración de sensores (fin de carrera, encoder y de apertura magnética).
RS09	El sistema debe permitir su restablecimiento a valores de fábrica.
RS10	El sistema debe permitir actualizar su versión de software de manera sencilla.

Cuadro 3.1: Tabla de requerimientos a nivel de sistema para la solución propuesta.

3.4.2. Módulo Electrónico

Los requerimientos encontrados para el Módulo Electrónico se listan en la Tabla 3.2. Estos requerimientos fueron implementados por un colaborador del equipo quién se encargó principalmente del desarrollo del firmware por lo que los detalles del diseño y la implementación del software embebido exceden los alcances de este informe.

3.4.3. Aplicación Móvil

En la tabla 3.3 se listan los requerimientos encontrados para la aplicación móvil.

3.4.4. Identificación de Riesgos

En esta sección se expone el panorama de incertidumbres del proyecto y se hace foco en las eventualidades que pueden surgir durante su ejecución. Estas pueden alterar, de alguna manera, la planificación y duración del proyecto. La gestión temprana de riesgos facilita la identificación de los mismos y permite la creación de planes de contingencia para minimizar el efecto en caso de ocurrencia.

Las situaciones que puedan constituir una amenaza a la realización del proyecto se pueden clasificar por el objeto de impacto o por su causalidad.

Categorización por objeto de impacto.

ID	Descripción
RME00	El módulo electrónico debe alimentarse con 220v AC.
RME01	El módulo electrónico debe conectarse al controlador por la bornera de pulsador externo
RME02	El módulo electrónico debe conectarse a la red wifi del inmueble donde será instalado.
RME03	El módulo electrónico debe almacenar los usuarios registrados autorizados para
RME04	El módulo electrónico debe funcionar de manera remota a través de internet.
RME05	El módulo electrónico debe funcionar de manera local offline en caso de que la conexión a internet se vea interrumpida.
RME06	El módulo electrónico debe recibir comandos desde la aplicación cliente y devolver una respuesta según se detalla en la API.
RME07	El módulo electrónico debe contar con puertos para los distintos tipos de sensores.
RME08	El módulo electrónico debe contar con luces indicadoras de funcionamiento.
RME09	El módulo electrónico debe incluir botones para configuración y operación.
RME10	El módulo electrónico debe poder actualizar su versión de firmware de manera inalámbrica.

Cuadro 3.2: Tabla de requerimientos del Módulo Electrónico.

- Al Proyecto: afectan la planificación y los recursos del proyecto.
- Al Producto: afectan la calidad o el rendimiento del producto.
- Al Negocio: afectan a la organización que desarrolla el producto.

Por otro lado se pueden clasificar los riesgos respecto de su origen o causa.

- **Riesgo Tecnológico** (TECNO): Estrechamente relacionado con los aspectos técnicos del proyecto se evidencian en las herramientas utilizadas para la implementación, evaluación y ejecución del trabajo. En el caso del presente proyecto existirán riesgos de hardware y software.
- **Riesgo de Personal** (PER) : Relacionados a las personas involucradas en la ejecución del proyecto. En este caso el equipo de desarrollo.
- **Riesgo Organizacional**(ORG): Derivan del entorno dónde se está realizando el trabajo. En este caso el proyecto está vinculado a un emprendimiento tecnológico.

ID	Descripción
RA00	La aplicación debe permitir el registro inicial del usuario con su número de teléfono.
RA01	La aplicación debe generar una clave única y secreta para el envío de comandos.
RA02	La aplicación debe descubrir y listar módulos electrónicos nuevos y ya configurados.
RA03	La aplicación debe permitir enviar comandos al módulo y recibir la respuestas según se detalla en la API.
RA04	La aplicación debe permitir configurar los módulos nuevos.
RA05	La aplicación debe permitir solicitar acceso a los módulos electrónicos encontrados ya configurados.
RA06	La aplicación debe permitir el accionamiento de los cerramientos eléctricos.
RA07	La aplicación debe permitir cambiar el alias de los módulos.
RA08	La aplicación debe ofrecer un modo de accionamiento con pantalla bloqueada.
RA09	La aplicación debe ofrecer distintas configuraciones según el privilegio del usuario.
RA10	La aplicación debe permitir al usuario administrador restablecer el módulo a valores de fábrica.
RA11	La aplicación debe permitir al usuario administrador calibrar el sensor.
RA12	La aplicación debe permitir al usuario administrador invitar a nuevos usuarios.
RA13	La aplicación debe permitir al usuario administrador convertir usuarios a administradores.
RA14	La aplicación debe permitir al usuario administrador eliminar cualquier usuario.
RA15	La aplicación debe permitir al usuario administrador actualizar la versión del firmware del módulo.

Cuadro 3.3: Tabla de requerimientos de la Aplicación Cliente.

3.4.5. Análisis de Riesgo

Para poder tener un panorama completo de los riesgos del proyecto se elabora una tabla con los riesgos identificados y se pondera de manera cualitativa los mismos. Los parámetros de esta ponderación serán la probabilidad de ocurrencia y el impacto de su efecto.

La probabilidad de ocurrencia se estimará en tres valores: Improbable (0.3), Probable (0.6) y Muy Probable (0.9).

El impacto se estimará también en tres valores: Bajo (1), Moderado(10), Alto(100) Realizando el producto aritmético entre estos parámetros se calcula la importancia de los riesgos identificados como se puede observar en la Tabla 3.4. Aunque el valor de importancia obtenido es ilustrativo ofrece una alternativa numérica que permite el ordenamiento y con esto un inmediato orden de prioridad.

ID	Descripción	Objeto de Impacto	Causa	Probabilidad	Impacto	Importancia
R00	Problemas con el entendimiento de la arquitectura elegida para la implementación	PROY	PER	0.3	100	30
R01	Implementación de la arquitectura elegida	PROY	PER	0.6	100	60
R02	Entendimiento del paradigma de programación reactiva utilizando RxJava	PROY	PER	0.6	10	6
R03	Cambio de versiones de librerías	PROY	TEC	0.6	1	0.6
R04	Falta de librerías de comunicación	PROY	TEC	0.3	100	30
R05	Librerías de comunicación no adaptadas a programación reactiva	PROY	TEC	0.6	10	6
R06	Falta de documentación de las librerías	PROY	TEC	0.9	10	9
R07	Retraso o complicaciones en el desarrollo del módulo electrónico	PROY	PER	0.9	100	90
R08	Subestimación de tiempo de desarrollo	PROY	ORG	0.9	10	9
R09	Cambio de requerimientos durante el desarrollo	PROY	NEG	0.9	100	90

Cuadro 3.4: Tabla de riesgos identificados y la ponderación para el cálculo de su importancia.

3.4.6. Planificación de Riesgos

Con el objetivo de minimizar los efectos de las posibles eventualidades riesgosas se plantean tres tipos de estrategias que se detallan a continuación.

- De Prevención: Acciones preventivas orientadas a reducir la probabilidad de ocurrencia.
- De Mitigación: Acciones preventivas orientadas a atenuar el impacto en caso de ocurrencia.
- De Contingencia: Acciones curativas en caso de ocurrencia.

Para los 5 riesgos identificados más importantes se plantearon las estrategias y se exponen en la Tabla 3.5.

ID	Descripción	Estrategia		
		De Prevención	De Mitigación	De Contingencia
R06	Retraso o complicaciones en el desarrollo del módulo electrónico	Se iniciará el desarrollo de ambos componentes una vez que se hayan establecido las tecnologías para ambos.	Se establecerá con claridad una API de comunicación a modo de contrato y con una versión definida.	Con las definiciones de la API se puede simular el comportamiento del módulo.
R08	Cambio de requerimientos durante el desarrollo	Se celebraran entrevistas con los posibles clientes y usuarios para anticipar cambios y requerimientos futuros.	Se realizará una revisión de los requerimientos periódica.	Se planificará la modificación o adición de funcionalidad para la próxima versión. Intentando mantener siempre un entregable funcional.
R01	Implementación de la arquitectura elegida	Se buscarán ejemplos de implementaciones similares y documentación respaldatoria.	Se contará con el apoyo de algún contacto técnico externo con experiencia para poder consultar.	Se agregarán la resolución de conflictos a la planificación de tareas ordinarias para tener en cuenta el impacto y el progreso en ese aspecto.
R00	Problemas con el entendimiento de la arquitectura elegida para la implementación	Se recolectará bibliografía diversa sobre los conceptos introducidos con la arquitectura.	Se podrá realizar consultas inter-equipo para despejar dudas y evitar retrasos.	Se pondrá en marcha un pequeño análisis de impacto al reemplazar el asunto bloqueante por una alternativa más sencilla y quedará documentado.
R03	Falta de librerías de comunicación	Se escogerá un framework de desarrollo lo suficientemente maduro como para anticipar la búsqueda de las librerías necesarias. Y asegurar la existencia de las mismas.	Se empleará un modo de implementación por interfaces a modo Mock, de manera que se pueda postergar el uso de la biblioteca sin retrasar el desarrollo de las funcionalidades	Se buscarán alternativas a los protocolos originalmente propuestos y se evaluará el impacto sobre la implementación. En caso de ser un cambio viable se dejará documentado.

Cuadro 3.5: Tabla con la planificación de los riesgos más importantes.

Capítulo 4

Diseño

Este capítulo se dedica a explorar los aspectos más relevantes relacionados con el diseño de la solución.

En primera instancia se plantea la necesidad de elegir y emplear un patrón de arquitectura de software para llevar a cabo la implementación de la aplicación. Brevemente se introducen los beneficios que motivaron la decisión de encuadrar las tareas de codificación bajo los lineamientos de la arquitectura seleccionada.

Se mencionan cada uno de los principios de diseño propuestos por la arquitectura y las repercusiones que deberían tener tanto en la estructura de la implementación como en su proceso. Como parte de la descripción técnica se listan las partes constituyentes propuestas, se detallan las características más relevantes, sus responsabilidades y la relación entre ellas.

Quizás la propiedad más crítica de un diseño como el sugerido es la comunicación entre sus componentes. Para su puesta en práctica se propuso utilizar el paradigma de programación reactiva. Se conoce a priori que esta nueva forma de pensar el software lleva adjunta una pronunciada curva de aprendizaje que implica una reformulación transversal del modo de codificación y resolución de los algoritmos en general. Teniendo en cuenta el impacto de esta decisión de diseño se hace necesario incluir una reseña de sus características principales.

Dado que el producto final incluye un modulo electrónico y un servicio online es imperativo definir con anticipación una interfaz de comunicación entre los subsistemas.

Así mismo, debido a la naturaleza de los requerimientos definidos en el capítulo ?? deben incluirse como parte del diseño diversos protocolos de comunicación, se mencionan sus características principales y se justifica su empleo en el funcionamiento del producto.

4.1. Arquitectura de Software

Para realizar la implementación de la aplicación cliente se eligió la plataforma de desarrollo para dispositivos android, más precisamente teléfonos inteligentes y tabletas.

El objetivo principal de emplear una estructura fija para la implementación del proyecto es utilizar un único "lenguaje arquitectónico" que resulte familiar a los integrantes de un posible equipo de desarrollo y que sea transversal tanto para la implementación android, iOS o cualquier otra plataforma que pueda aparecer durante la vida útil del producto. De esta manera no es necesario pagar un costo demasiado alto al incluir una implementación del mismo sistema para una plataforma distinta. Los equipos de cada una de estas implementaciones podrán discutir aspectos de diseño, validar reglas de negocio y evacuar dudas sin tener en cuenta los detalles de las plataformas, así mismo será más fácil conservar coherencia y mostrar armonía entre las implementaciones nativas para dichas plataformas.

4.1.1. Clean Architecture

También conocida como arquitectura de capas (Onion Architecture). Su característica distintiva es que coloca la lógica de negocio, también conocido como dominio, al centro del diseño, es decir justo al medio entre las entradas y las salidas del sistema[1].

El **Principio Fundamental** de esta arquitectura puede resumirse en una frase: "*Las capas internas contienen lógica de negocios, las capas externas detalles de implementación*".

4.1.1.1. Dominio Transparente

Al listarse los directorios de un proyecto que cumple con los lineamiento de esta arquitectura, con tan solo leer el nombre de las carpetas debería ser posible, casi de inmediato, tener una idea de qué se trata esta aplicación, independientemente de la tecnología. Todo lo demás es un *detalle de implementación*[2].

Esta arquitectura propone un conjunto de características que debería seguir el proyecto que la implementa:

- Regla de dependencia
- Abstracción

- Comunicación entre Capas

4.1.1.2. Regla de Dependencias

Las capas externas deben depender de las capas internas. Permaneciendo en el centro las entidades del dominio inmediatamente seguidas por los objetos que encapsulan la lógica de negocio y que tienen acceso a tal dominio. En la Figura 4.1 una ilustración con una flecha que representa el sentido de las dependencias.

En lugar de emplear el verbo “depende”, tal vez sea mejor usar términos como “ve”, “conoce” o “está consciente de...”. En estos términos, las capas externas *ven, conocen y son conscientes* de las capas internas, pero el recíproco está prohibido como regla de diseño. Como se mencionó anteriormente, las capas internas contienen lógica de negocio y las externas los detalles de implementación. Combinado con el Principio Fundamental 4.1.1, se deduce que la lógica de negocio no ve, ni conoce detalles de implementación.

No existe una única forma de implementar esta regla. Una estrategia consiste en colocar las clases de cada capa en paquetes diferentes, poniendo especial cuidado en no importar paquetes “externos” en paquetes “internos”. Sin embargo, si algún programador del equipo no es consciente del principio de dependencias, nada le impediría incumplirlo. Otro enfoque un tanto más sofisticado consiste en separar las capas en diferentes módulos de construcción independiente, y ajustar las dependencias en el archivo de construcción para que la capa interna simplemente no pueda utilizar la capa externa, sin embargo este enfoque implica un exhaustivo conocimiento de la herramienta de construcción de la plataforma para la que se está desarrollando. Por esta razón para el presente proyecto se optó por la primera alternativa.



Figura 4.1: Esquema de dependencias para una arquitectura en capas.

4.1.1.3. Principio de Abstracción

El principio de abstracción ya se ha insinuado antes. Postula que, a medida que se recorre el diagrama de la arquitectura de la Figura 4.2 a lo largo del radio en dirección del centro, las implementaciones se vuelven más abstractas, agnósticas de plataformas y frameworks. Como se mencionó en la sección anterior, el círculo interno contiene lógica de negocios mientras que el exterior comprende los detalles de implementación.

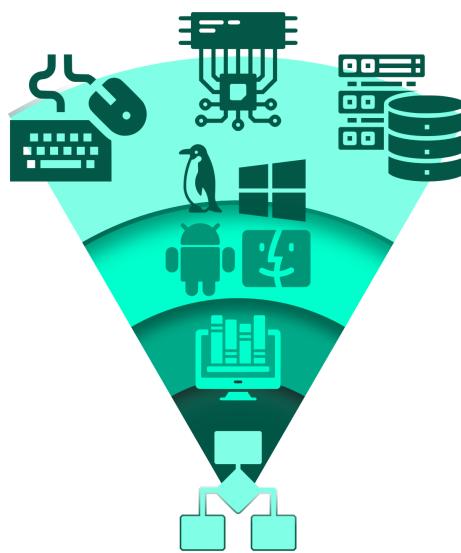


Figura 4.2: Principio de Abstracción en una arquitectura por capas.

4.1.1.4. Comunicación entre Capas

Dado que en esta arquitectura la lógica de negocio se encuentra en el centro del diseño, ésta debe mediar entre los componentes periféricos de entrada y salida. Aquí se presenta un inconveniente paradójico introducido por el empleo de la regla de dependencias: la capa con lógica de negocios ni siquiera conoce que los componentes en la periferia existen. Esto representa un desafío ya que necesitamos que los datos sean capaces de fluir desde las capas externas a las internas y viceversa, esto sin violar la regla de dependencias.

Se propone entonces un método para resolver el problema de comunicación entre capas. En primera instancia se define los objetos *Caso de Uso* como las entidades principales de la Lógica de Negocios. Se conceptualizan dos puertos para cada caso de uso, uno de entrada y otro de salida. Mediante el puerto de entrada se reciben

parámetros para la ejecución del caso de uso mientras que el puerto de salida permite la extracción de sus resultados. En la práctica, la definición de estos puertos (interfaces) se realiza dentro de la capa más interna mientras que su implementación e instancia en la capa más externa. Esto satisface la regla de dependencias y puede visualizarse en la Figura 4.3.

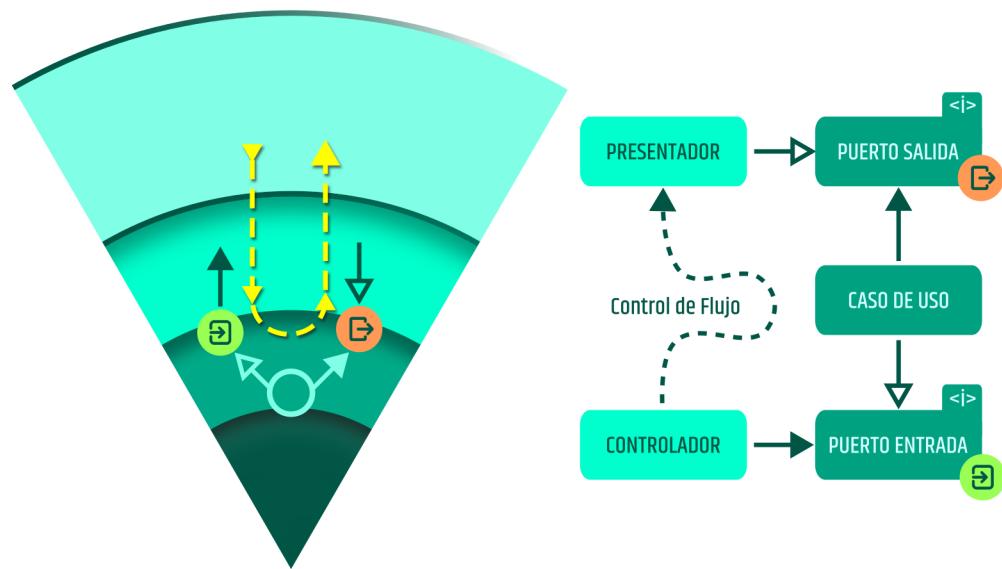


Figura 4.3: Comunicación entre capas.

4.2. Diseño de Capas

Como parte del proyecto de PPS se investigaron implementaciones reales de esta arquitectura en proyectos android. Las dos mejores disponibles a código abierto y con buena documentación pertenecen a un desarrollador argentino Fernando Cejas [3] (SoundCloud) y al repositorio de Blueprints Arquitectónicos de Google[4].

Ambas propuestas dividen la implementación en tres capas principales, una capa de presentación, una capa de dominio y la capa de datos. Cada capa tiene una responsabilidad bien definida y se comunica con una única capa vecina respetando la regla de dependencias. La cadena de comunicación puede apreciarse en la figura 4.4. La capa de presentación transmite las acciones del usuario a la capa de dominio, la cual efectúa solicitudes de información a la capa de datos. Una vez resueltas, la capa de datos enviará los resultados de vuelta a la capa de dominio que ejecutará la lógica de negocios correspondiente. Esto genera una respuesta que, a su vez, produce los efectos deseados en la capa de presentación.

A continuación se describe brevemente las responsabilidades de cada capa.

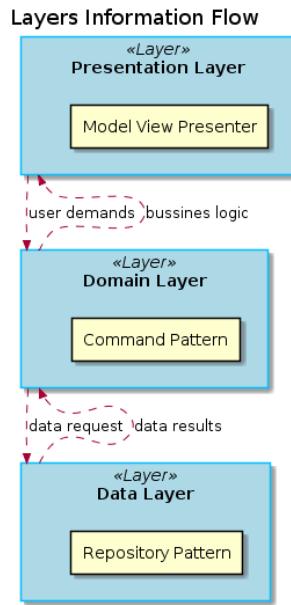


Figura 4.4: Esquema de dependencias para una arquitectura en capas.

- Capa de Presentación: Esta capa se encarga de presentar la interfaz de usuario, esto es, mostrar por pantalla los objetos visuales correspondientes y recibir los eventos de interacción que realiza el usuario. Para la implementación se recomienda el empleo del patrón de diseño conocido como **MVP (Model View Presenter)**.
- Capa de Dominio: Esta capa contiene toda la lógica de negocio. La capa de dominio contiene las clases denominadas casos de uso o interactores según la literatura. Estos objetos encapsulan los escenarios contemplados por la lógica de negocio y son ejecutados por la capa de presentación. Estos casos de uso representan todas las acciones posibles admitidas por el sistema y que pueden ser compuestas en la implementación por los desarrolladores siempre desde la capa de presentación. Para la implementación de estos casos de uso se sugiere la utilización del patrón de diseño conocido como **Command Pattern**.
- Capa de Datos: Esta capa administra la adquisición de datos y es capaz de utilizar diferentes orígenes de datos, así como la lógica de cache o persistencia temporal. Esta capa se suele implementar utilizando el patrón de diseño conocido como **Repository Pattern**.

4.2.1. Capa de Presentación: Model View Presenter (MVP)

El patrón de arquitectura que se utiliza en la capa de presentación se conoce como Modelo-Vista-Presentador. La idea detrás del patrón es concentrar la interacción

con el usuario en una entidad conocida como presentador, las operaciones directamente relacionadas con la manipulación de objetos gráficos y la captura de acciones de usuario están delegadas a la entidad Vista, finalmente la adquisición de datos y la ejecución de los algoritmos que encapsulan la lógica de negocio forman parte de las entidades modelo en el patrón [5]. El diagrama de componentes [4.5](#) describe la relación entre los objetos principales.

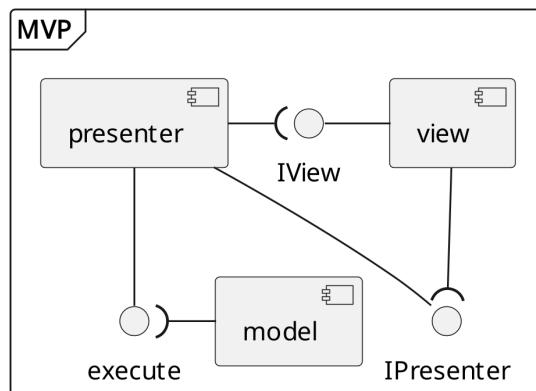


Figura 4.5: Diagrama de componentes del patrón.

Es posible deducir el modo en que interactúan los componentes. El presentador se comunica de manera bidireccional con la vista y de manera unidireccional con el modelo.

Una convención para la implementación del patrón es tratar de generar vistas completamente ajena de cualquier lógica operativa y agnósticas del estado de la aplicación. Esto las convierte en un mero instrumento de interfaz entre lo que percibe el usuario y sus reacciones.

Otra de las convenciones sugiere utilizar objetos modelo-vista en la comunicación entre el presentador y la vista para estandarizar el tipo de mensaje y el proceso de actualización de la vista.

En el caso de las implementaciones antes mencionadas la interfaz con el modelo es satisfecha mediante el uso de objetos casos de uso ó interactor, ambos términos suelen utilizarse de manera intercambiable.

La secuencia de mensajes que son intercambiados entre los objetos del patrón se ilustran en el diagrama de secuencias [4.6](#).

4.2.2. Capa de Dominio: Patrón Command

El patrón de diseño *Command* se utiliza para abstraer la ejecución de procedimientos mediante la implementación de entidades comando [6]. Estos objetos ejecutan

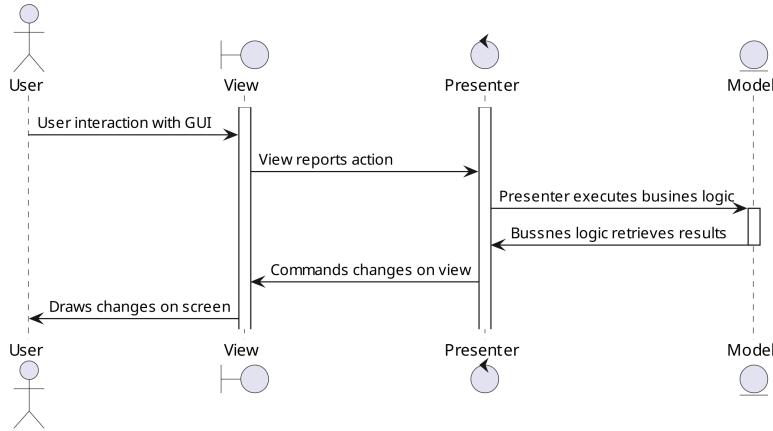


Figura 4.6: Diagrama de secuencia para una interacción con el usuario utilizando MVP.

un único algoritmo y encapsulan la lógica de negocio de la aplicación o sistema. Originalmente el diseño contempla 4 entidades principales que se pueden apreciar en el diagrama de clases de la figura 4.7:

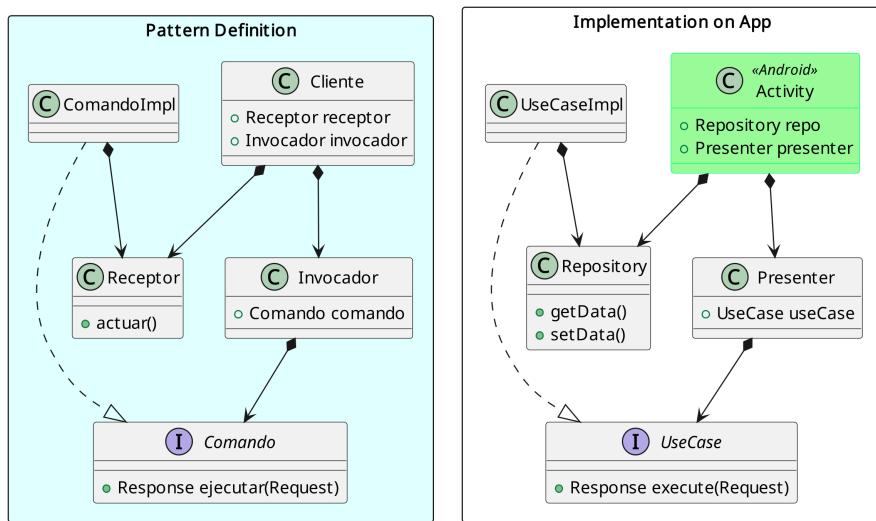


Figura 4.7: Diagrama de clases para el planteo inicial del patrón Command y su adaptación para este proyecto.

1. **Cliente:** Este componente se encarga de crear las instancias de cada comando y distribuirlas entre los correspondientes invocadores.
2. **Receptor:** Es la entidad que se ve afectada por la ejecución de un comando. Puede ser compartida por varios comandos o bien un único comando puede interactuar con varios receptores en su ejecución.
3. **Comando:** Es el objeto que contiene la implementación del algoritmo o lógica de ejecución.

4. Invocador: Se encarga de ejecutar instancias de comandos.

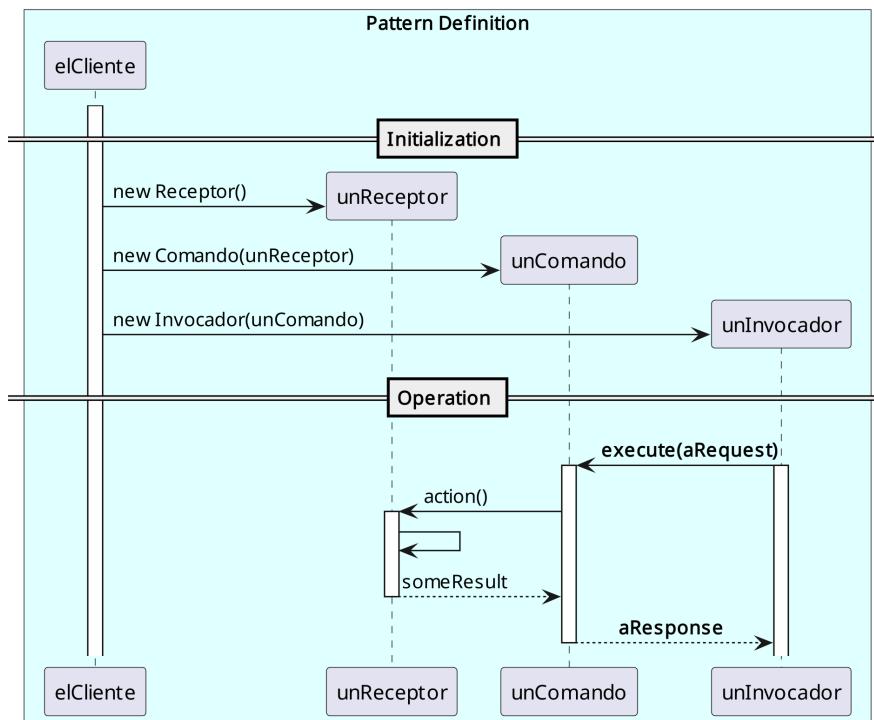


Figura 4.8: Diagrama de secuencia para el patrón Commander.

La mayoría de los ejemplos de aplicación de este patrón implementan un comando por cada una de las operaciones soportadas por el sistema o aplicación. Sin embargo en sistemas suficientemente grandes la diversidad de funcionalidades soportadas es tal que el diseño propuesto se vuelve impráctico. Para mitigar este problema se suele implementar de manera adicional una modificación que permite la ejecución paramétrica de los comandos para reducir al máximo la cantidad de comandos implementados. Esta modificación permite diversas alternativas de implementación pero la más utilizada incorpora conceptos del patrón Request-Response donde el comando es utilizado como un objeto tipo caja negra que admite Solicitudes y emite Respuestas estandarizadas para cada caso.

- **Solicitud (Request):** Un objeto que contiene el conjunto de parámetros de entrada que deben ser satisfechos para poder realizar la ejecución de la rutina del comando.
- **Respuesta (Response):** Un objeto que contiene los valores que se obtuvieron de la ejecución del algoritmo del comando.

Teniendo en cuenta estos detalles se describe el flujo de operación y ejecución 4.9 de un comando:

1. El cliente crea instancias de comandos y sus correspondientes invocadores.
2. El invocador crea e inicializa los objetos solicitud necesarios para ejecutar cada comando.
3. El invocado ejecuta los comandos llamando al método "ejecutar" implementado por cada comando pasando como parámetro la solicitud previamente creada.
4. El invocador observa los resultados en espera activa implementando el patrón Observer o mediante algún esquema de callback.

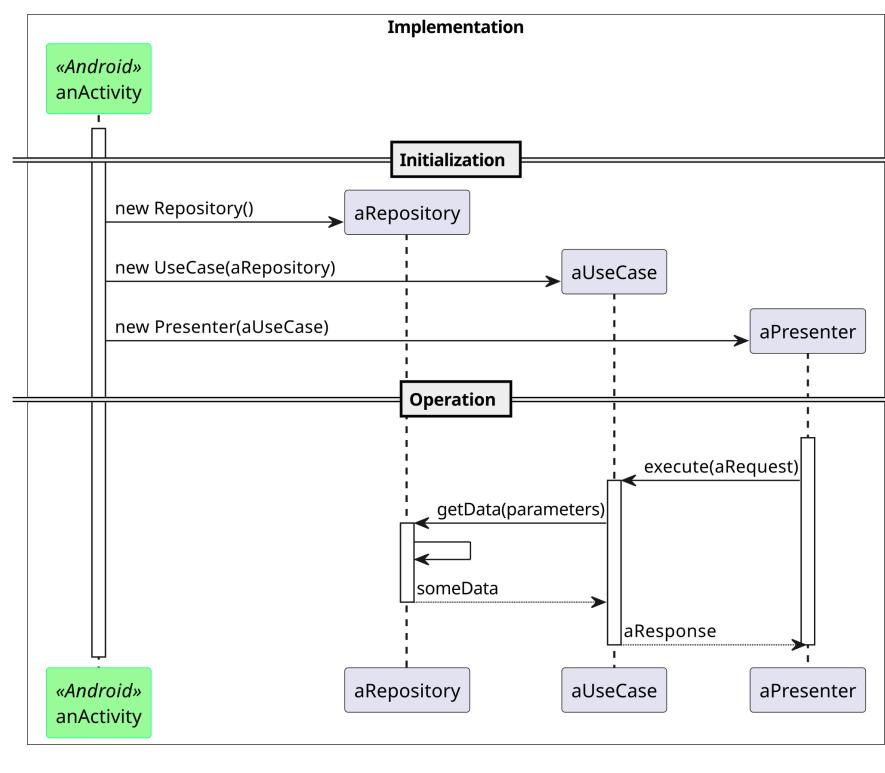


Figura 4.9: Diagrama de secuencia para el diseño revisado.

Siguiendo los lineamientos de la arquitectura propuesta los autores denominan a los comandos: Casos de Uso, ó Interactores.

Como una nota relevante de implementación se recomienda ejecutar las rutinas de los comandos en un hilo/proceso separado para evitar bloquear el proceso principal de la aplicación.

4.2.3. Capa de Datos: Patrón Repository

En la capa de datos se propone la implementación de un patrón de diseño conocido como Repository(Repositorio). Originalmente se concibe a este diseño como una

forma de estandarizar la implementación y el uso de los objetos DAO (Data Access Object) comúnmente utilizados para mapear objetos entidad con las persistencias en la base de datos [7]. Adicionalmente este patrón encapsula en la clase repositorio todos los métodos particulares de filtrado, procesamiento calculado y ordenamiento de entidades. Sin embargo se define una interfaz genérica que deberá ser respetada por todas las implementaciones de repositorios para todo el sistema independientemente de la entidad que atienda. En el diagrama de clases de la figura 4.10 se puede apreciar el diseño original del patrón.

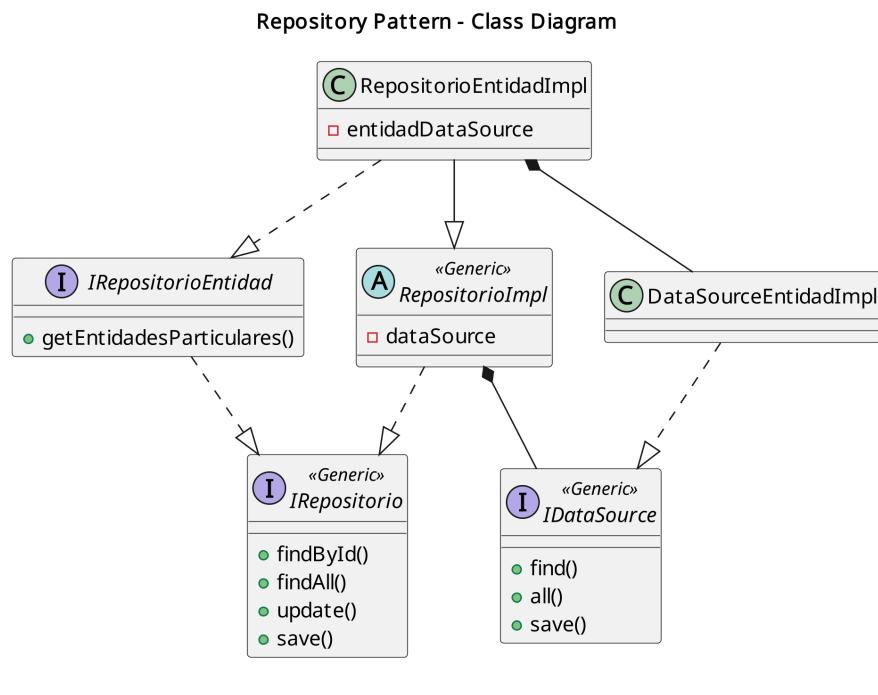


Figura 4.10: Diagrama de clases del patrón Repository.

Como puede apreciarse en el diagrama se definen:

- IRepositorio: es una interfaz genérica que establece el contrato básico que deben respetar todas las implementaciones de repositorios.
- RepositoryImpl: es una clase genérica que establece la interacción con una fuente de datos genérica.
- IRepositorioEntidad: es la interfaz que *Especifica* la interfaz genérica de repositorio y establece el contrato o métodos particulares que deberá cumplir la implementación concreta de repositorio para esta Entidad en particular.
- RepositoryEntidadImpl: es la clase que *Especifica* la implementación genérica de repositorio e implementa los métodos particulares para esta Entidad en particular.

En una repaso más detallado del diagrama puede observarse que existen dos interfaces genéricas para acceso de datos IRepository y IDatasource, esto podría generar confusión y duplicación de código. Adicionalmente en cualquier implementación moderna de sistemas con persistencia es prácticamente mandatario el empleo de frameworks que soportan ORM (Object Relational Mapping) out-of-the-box. En la figura 4.11 se puede observar la modificación sobre la propuesta original del patrón de diseño.

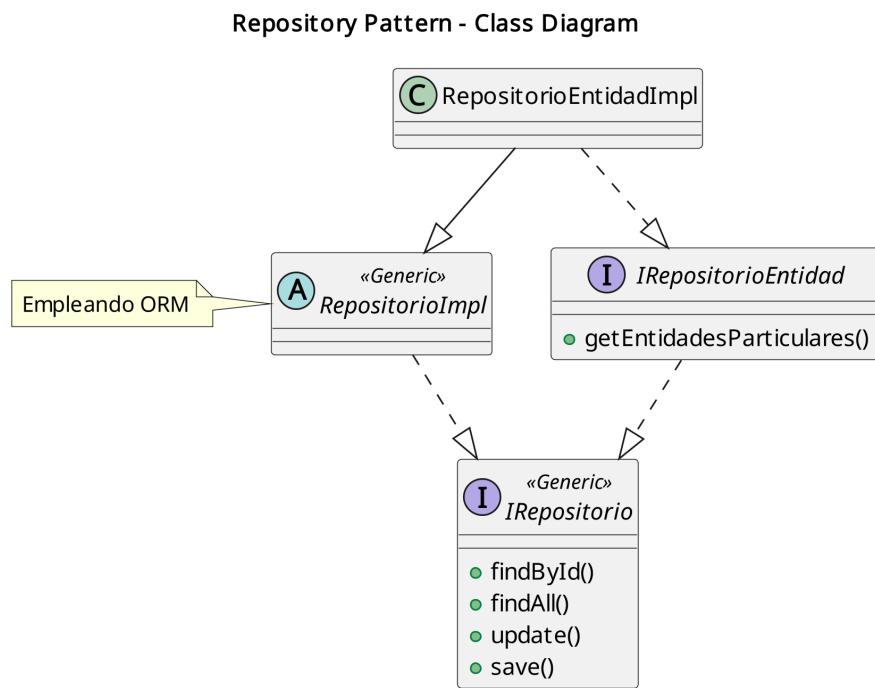


Figura 4.11: Diagrama de clases reducido del patrón Repository.

En las implementaciones mencionadas se reemplaza la interfaz del repositorio por la interfaz fuente de datos. Coloquialmente es fácil de entender ya que un repositorio definitivamente es una fuente de datos. Si además se quita la estandarización por genéricos se consigue un diseño más sencillo y que genera menos código estructural o scaffold manteniendo un único contrato o interfaz de acceso a los datos. En la figura 4.12 se puede observar la modificación mencionada y el diseño final propuesto para la implementación del patrón.

Principalmente orientado a encapsular la manipulación, selección, priorización y mantenimiento de diversas fuentes u orígenes de datos, este diseño de repositorios modificado permite que el peticionario se comunique con una única interfaz para solicitar operaciones sobre datos permaneciendo agnóstico del origen sobre el cual tendrán impacto. Implementar una política de caching local se convierte en una tarea sencilla de implementar y mantener.

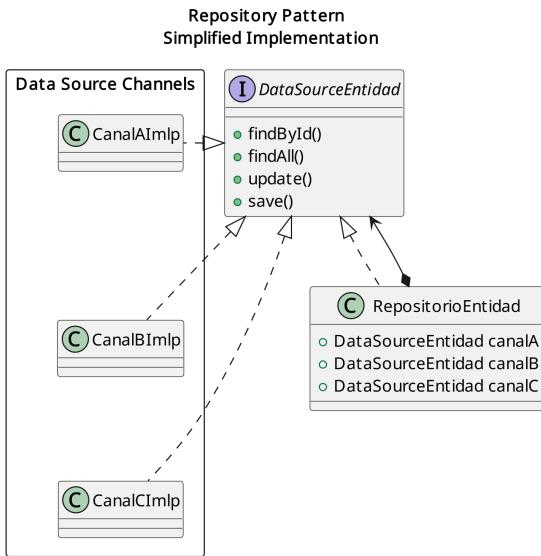


Figura 4.12: Diagrama de clases del patrón Repository modificado.

4.3. Programación Reactiva

Se trata de un paradigma de programación cuya máxima sostiene que el código debe observar y reaccionar a estímulos internos y externos como eventos, interrupciones, resultados, etc. Estos estímulos por lo general se manifiestan como unidades de datos y tienen como característica principal su naturaleza asíncrona o de sucesión impredecible. Por lo que el código deberá observar de manera continua la ocurrencia de estos y reaccionar consecuentemente. En la jerga, la sucesión de estímulos de una misma fuente se denomina flujo.

Se definen tres entidades fundamentales para este paradigma: los **Estímulos**, sus **Emisores** y quienes los **Observan**. También se admite la concatenación de este trío de entidades lo que produce operadores que podrán, a su vez, generar el mismo flujo o uno modificado.

En términos meramente lingüísticos al adaptar un lenguaje de programación procedural u orientado a objetos para que soporte este paradigma se le confieren atributos que originalmente eran propios de la programación funcional como lo son la composición de funciones y el tratamiento de datos como flujos o secuencias, lo que permite un abordaje más natural a problemas que implican manejo de eventos, concurrencia y asincronía. Así mismo la programación reactiva promueve la inmutabilidad y la pureza funcional, especialmente en la manipulación de datos a través de operadores. Los cuales en los flujos reactivos suelen aplicar transformaciones inmutables a los datos, lo que garantiza que los originales no se modifiquen y que las operaciones sean consistentes y predecibles.

En términos arquitectónicos este paradigma implica la implementación del patrón *Observer* para los objetos que observan los estímulos y el patrón *Iterator* para los emisores. Será necesario también convertir a las funciones en entidades del lenguaje con estatus de *First Class Citizen* para admitir su composición de manera genérica.

Afortunadamente, y de la mano de el proyecto ReactiveX, estas adaptaciones para lenguajes no funcionales se ofrecen como librerías que implementan las tres entidades, una colección de operadores y el manejo de las primitivas de concurrencia. En particular para Java existe la librería RxJava que es la que utilizaré en el proyecto.

4.4. Interfaz Módulo - Aplicación

El monitoreo y control de los módulos se realiza utilizando la aplicación móvil.

Tanto el módulo como la aplicación son sistemas de software independientes que se ejecutan en dispositivos de arquitecturas distintas. Dado que, por requerimientos del producto, ambos sistemas deben interactuar entre si, es necesario definir las tecnologías, protocolos y definiciones que faciliten la comunicación.

Para la configuración inicial el módulo genera un Access Point Wi-Fi, el teléfono android se conectará a éste generando una LAN IPv4 con solo estos dos dispositivos en conexión punto-a-punto. Utilizando esta red la aplicación enviará las credenciales del router WiFi para su operación regular.

La operación regular del producto requiere que el módulo electrónico y el teléfono con la aplicación estén conectados al mismo router Wi-Fi. De esta manera ambos dispositivos pertenecen a la misma LAN IPv4. El módulo anunciará su presencia en la red para que la aplicación pueda encontrarlo y registre su dirección IP.

Ya sea para la configuración inicial o la operación regular el intercambio de mensajes se realiza utilizando el protocolo HTTP.

En una segunda etapa se extenderá el modo de comunicación entre el módulo y la aplicación móvil a través de internet empleando el protocolo MQTT. En la figura 4.13 se puede observar una representación gráfica de los modos de operación soportados por el producto.

4.4.1. Protocolos de Comunicación

4.4.1.1. MDNS DNS-SD

La aplicación móvil podrá descubrir todos los módulos disponibles en la LAN.

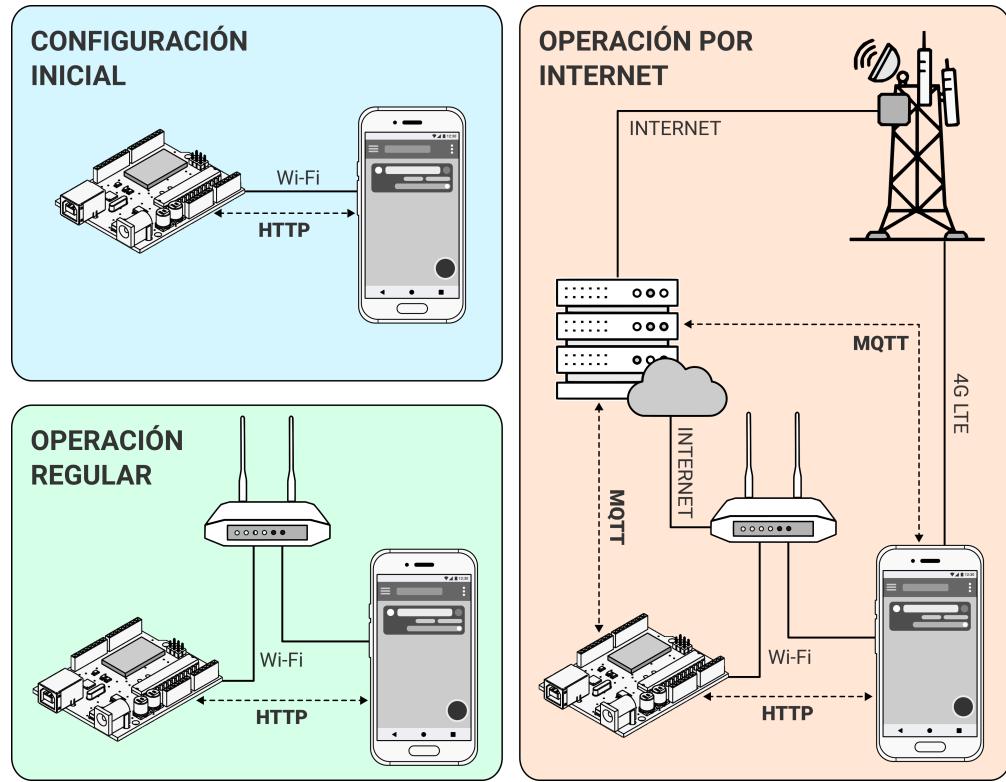


Figura 4.13: Diagrama de comunicación entre módulo y aplicación.

Para conseguir esto se optó por utilizar un enfoque Zeroconf que emplee DNS-SD y mDNS para resolver los hostnames de los módulos compatibles y sus direcciones IP. Incluso en la literatura existe una confusión entre estos términos por lo que será debidamente aclarada antes de continuar.

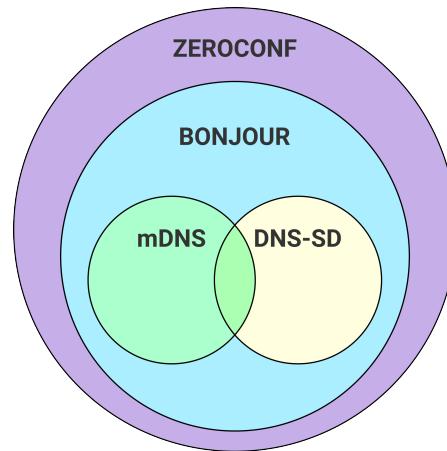


Figura 4.14: Diagrama de Venn que ilustra la relación entre los términos.

Zeroconf (Zero Configuration Networking) no es un estándar sino un término acuñado en la década de 1990 para denotar a la colección de protocolos y tecnologías

que habilitan la configuración automática de redes y el descubrimiento de servicios en redes locales.

mDNS es un protocolo de capa de aplicación (RFC 6762) y se presenta como una extensión sobre *DNS unicast* para la resolución de nombres en redes de área local. Este protocolo resuelve exclusivamente nombres de host que terminan con .local como dominio raíz. A diferencia de DNS la información de resolución se almacena localmente en cada dispositivo de la red y el intercambio de mensajes se realiza a una dirección multicast dónde cada dispositivo responde directamente a las consultas.

DNS-SD es un estándar de capa de aplicación (RFC 6763) que permite a clientes DNS descubrir instancias nombradas de un servicio utilizando registros DNS (RR). Una instancia de servicio en DNS-SD se describe utilizando los siguientes RR: SRV, TXT, PTR y A/AAAA.

- PTR proporciona el mapeo entre tipos de servicio e instancias de servicio que lo proveen.
- SRV contiene información de la instancia de servicio, tipo de servicio (protocolo) peso, puerto y endpoint. Los parámetros de prioridad y peso dan preferencia cuando el mismo servicio es proporcionado por múltiples instancias. El parámetro endpoint respeta el formato de nombre de host.
- TXT contiene los metadatos del servicio en forma de pares [clave]:[valor]. El contenido exacto depende del protocolo utilizado.
- A/AAAA Resuelve una dirección IP a partir de un nombre de host.

En mDNS/DNS-SD, todos los RRs que describen una instancia de servicio se almacenan en el nodo que proporciona tal servicio. Un ejemplo de intercambio de mensajes mediante el cual se realiza el descubrimiento de un servicio se muestra en la figura 4.15.

Bonjour es una implementación zeroconf desarrollada por Apple. Emplea mDNS como protocolo de comunicación y DNS-SD para el descubrimiento y descripción de los servicios.

Android ofrece una API para el descubrimiento de dispositivos *Network Service Discovery API* pero ésta no soporta la consulta de registros TXT. Google ofrece también la *Google Nearby API* una librería de similares características pero que es de código cerrado y no proporciona el nivel de granularidad adecuado para este proyecto. Por estos motivos se optó por emplear la librería *RxDNSSD* de andriydruk [8] que hace de wrapper reactivo sobre la implementación oficial de Bonjour.

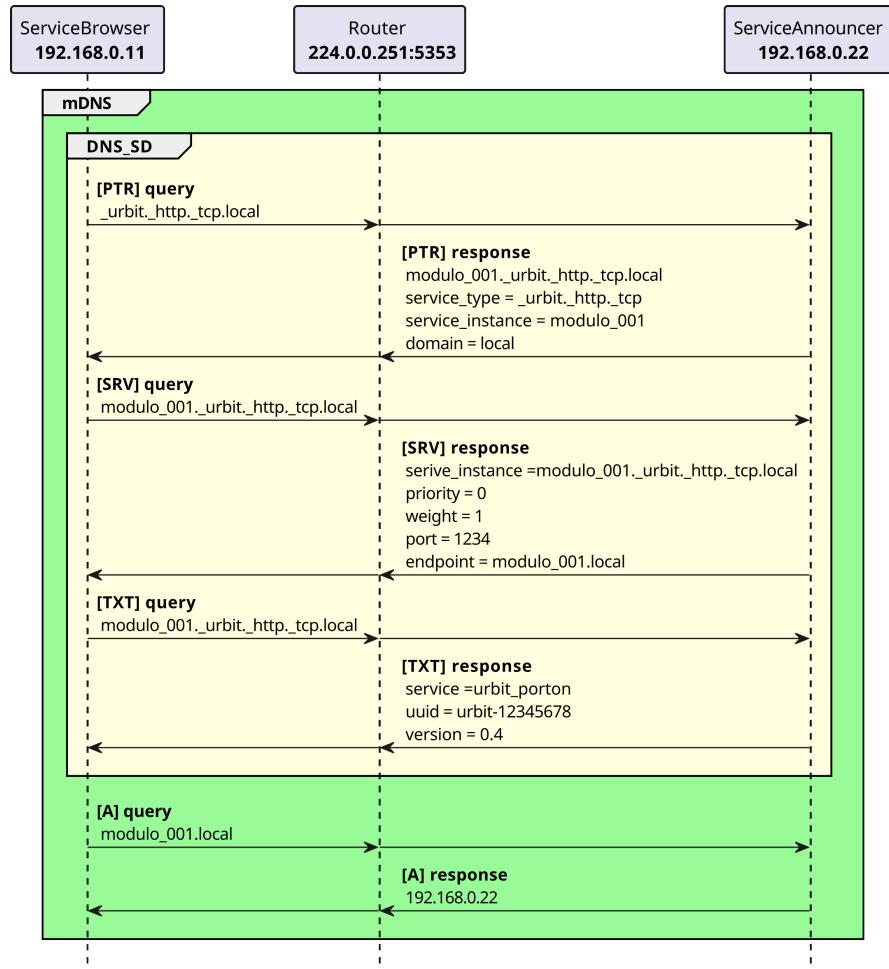


Figura 4.15: Secuencia del descubrimiento de servicios.

4.4.1.2. HTTP y Digest Authentication

El módulo y la aplicación se comunican por defecto utilizando HTTP.

El uso normal del producto implica la autenticación y la autorización de los usuarios por parte del módulo. Estas operaciones requieren el envío y la corroboración de las credenciales de usuario. El enfoque más básico *Basic Authentication* realiza el intercambio de las credenciales en texto plano lo que implica un gran riesgo de seguridad. Se evaluaron diversas alternativas para securitizar esta tarea y se optó por emplear *Digest Authentication*.

Digest Authentication es un esquema de autenticación (RFC 2617) para comunicaciones HTTP cuyo propósito es proveer un método de autenticación de acceso que evite las fallas más serias de Basic Authentication. Este esquema requiere la realización de un intercambio inicial entre el servidor y el cliente para verificar las credenciales de usuario. El procedimiento se describe a continuación, un ejemplo detallado se muestra en la figura 4.16.

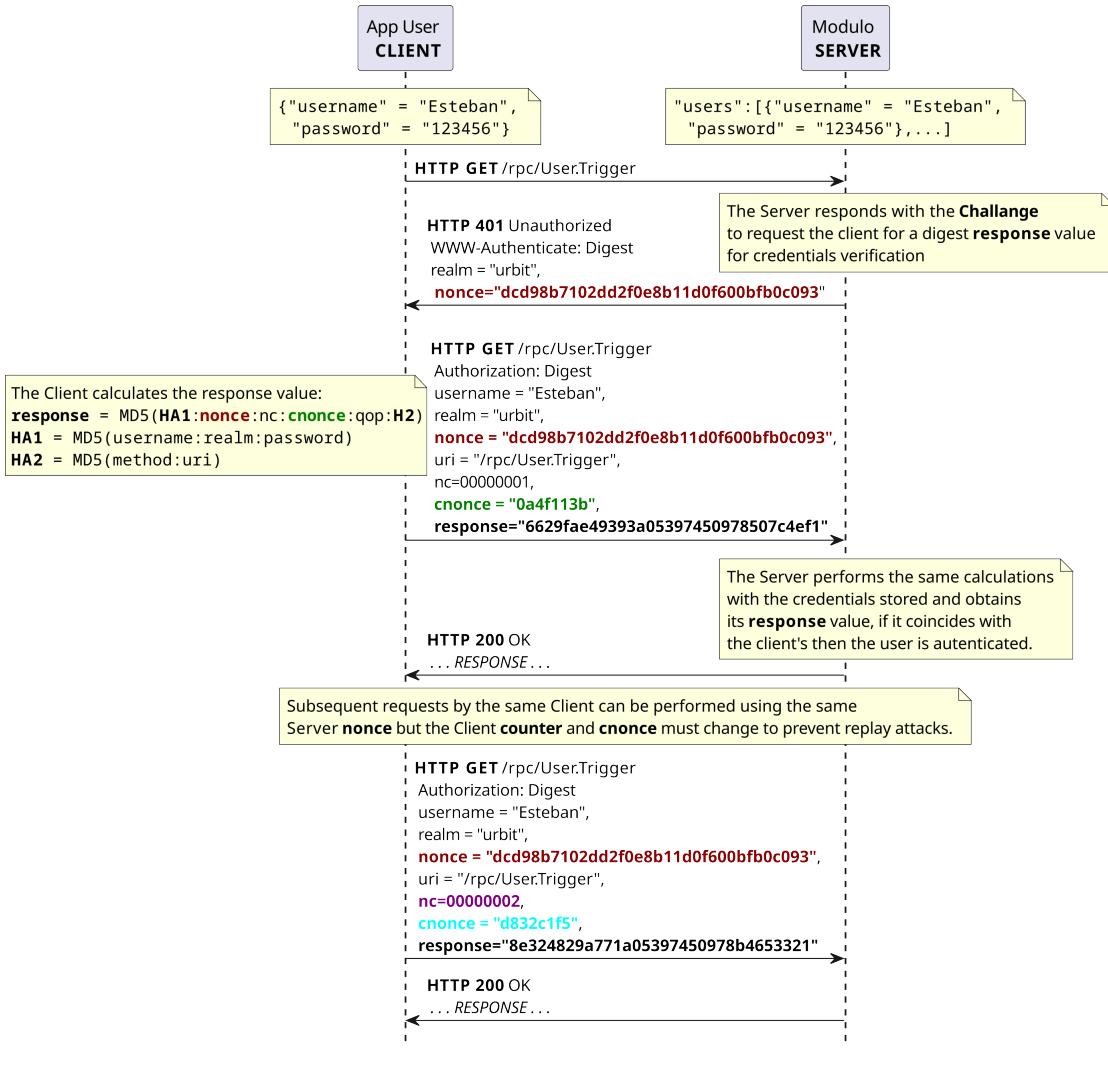


Figura 4.16: Secuencia de la autenticación utilizando Digest Authentication.

1. El cliente hace una solicitud con parámetros de autenticación incorrectos o ausentes.
2. El Servidor responde con el challenge solicitando que el cliente calcule y envíe el valor **response**.
3. El cliente genera todos los argumentos necesarios, calcula el valor **response** y replica la solicitud original con estos datos adjuntos como HEADER.
4. El Servidor hace el mismo cálculo y compara su resultado con el recibido del cliente. Si coinciden entonces se autentica al usuario y se procede con la consulta.

Una vez autenticado, el cliente puede realizar nuevas solicitudes utilizando el mismo **nonce** del servidor para calcular el valor **response**. Sin embargo es necesario que

el cliente actualice **cnonce** y el contador **nc** para prevenir ataques de tipo *replay attack*.

4.4.1.3. MQTT

Uno de los atractivos del producto es la posibilidad de monitorear y accionar módulos de manera remota.

Para que esto sea posible es necesario que tanto el módulo como la aplicación móvil tengan acceso a internet ???. Luego de discutir alternativas el equipo decidió que la comunicación por internet se realizará utilizando MQTT.

MQTT es un protocolo de capa de aplicación (RFC 9431) diseñado para el intercambio de mensajes machine-to-machine (M2M) en redes de bajo ancho de banda y alta latencia. Este protocolo de comunicación se construye utilizando la arquitectura publish-suscribe mediante la que proporciona un servicio de colas de mensajes. Su implementación en los clientes es liviana por lo que se puede utilizar en sistemas embebidos de bajos recursos de hardware, razón por la cual es ideal para este proyecto.

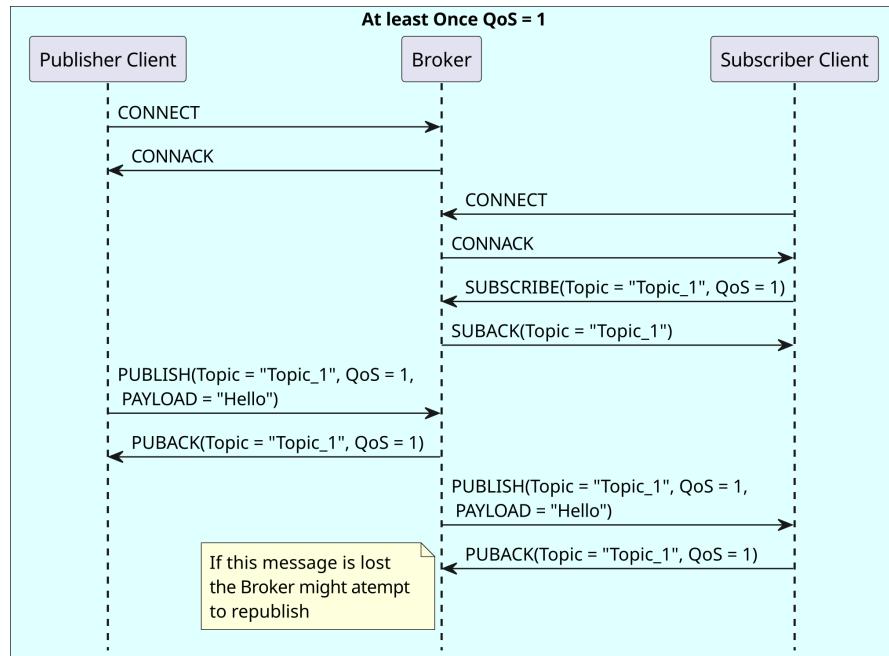


Figura 4.17: Secuencia del envío de mensajes par *QoS* = 1.

Para su funcionamiento es fundamental el rol de un agente que se denomina Broker. Este hace de intermediario entre los clientes conectados que envían mensajes y quienes los reciben. Los clientes pueden suscribirse a múltiples Tópicos para recibir los mensajes que se publican en cada uno de ellos. Así mismo los clientes pueden publicar mensajes en Tópicos existentes o crear nuevos. En consecuencia el Broker

debe mantener un registro de los clientes, tópicos y suscripciones. Un ejemplo de envío de mensajes se muestra en la figura 4.17.

Todos los clientes envían de forma periódica el mensaje KEEP_ALIVE al Broker con el propósito de mantener actualizadas sus direcciones IP y el estado de las sesiones.

Haciendo uso de estas características el protocolo consigue establecer un canal de comunicación continuo y persistente entre los clientes del servicio.

Los pormenores de la adaptación de este protocolo a los requerimientos del proyecto se detallaran más adelante.

4.4.2. RPC API

Como se expuso al inicio de la sección 4.4 el módulo y la aplicación deben interactuar entre sí empleando diversos protocolos de comunicación. Si bien se detallaron los medios y los procedimientos, aún falta definir el formato de los mensajes que intercambiarán para garantizar la ejecución de las funcionalidades.

Con este propósito se analizaron diversas alternativas utilizadas en la industria y se optó por especificar una API RPC.

RPC (Remote Procedure Call) o ejecución de procedimientos remotos es un protocolo de mensajes (RFC 1057, 5531) que permite a un sistema cliente ejecutar funciones en otro sistema servidor de manera transparente. *El cliente llama al procedimiento como si fuera local y la implementación RPC subyacente se encarga de los detalles de comunicación.*

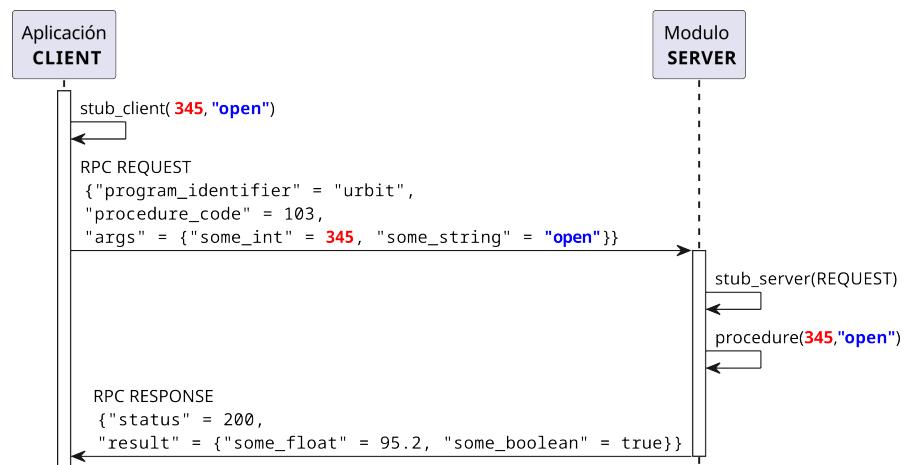


Figura 4.18: Secuencia de ejecución de un procedimiento remoto.

El protocolo implementa el patron Request-Response que se compone de “stubs” o “pasamanos” de los procedimientos tanto en el cliente como en el servidor. Estos

stubs son funciones que no ejecutan la lógica del procedimiento sino que se encargan de: construir estructuras de datos para solicitudes y respuestas, convertir estos objetos de memoria en otros con formatos de red, y los pormenores de su transmisión.

Elementos de una Solicitud (Request):

- Identificador del programa
- Número del procedimiento
- Argumentos

Elementos de una Respuesta (Response):

- Resultado o valor de retorno.
- Información de estado (Ejecución Exitosa, Error).

Por legibilidad y estandarización, solicitudes y respuestas tendrán formato JSON. En la figura 4.18 se muestra un ejemplo de RPC.

El formato definitivo para Solicitudes:	El formato definitivo para Respuestas exitosas:	El formato definitivo para Respuestas con error:
<pre>{ "id": 103, "method": "method_name", "params":{ "un_parametro": ..., "otro_parametro": ..., ... } }</pre>	<pre>{ "id": 103, "result":{ "un_resultado": ..., "otro_resultado": ..., ... } }</pre>	<pre>{ "id": 103, "error":{ "code": 404, "message": "... no existe", ... } }</pre>

En una de las labores más tediosas del diseño se revisaron los casos de uso y a partir de ellos se definieron 17 procedimientos que la aplicación podrá utilizar para ejecutar en un módulo. La especificación completa se puede consultar en la tabla 4.1. La invocación de los procedimientos se realizará enviando consultas HTTP siempre utilizando el método POST.

Código	Protocolo	Descripción	Argumentos	Respuesta	Errores
100	Admin.FactoryReset	Elimina las configuraciones y los usuarios registrados.	{}	{"message":string}	401 Unauthorized 403 Forbidden {"code":int, "message":string}
101	Admin.SetWiFiAP	Conecta el módulo a un Access Point WiFi.	{"ssid": "string", "password": "string"}	{"message":string}	400 Bad Request [ssid and password required] [wrong ssid or password]
102	Admin.GetUsers	Devuelve la lista de usuarios.	{}	{"users": [{"user_name":string, "user_type":int}]}	400 Bad Request [username or type incorrect] 404 Not Found [user not found]
103	Admin.UpdateUser	Cambia el nivel de privilegio del usuario pasado como parametro	{"user_name":string, "user_type":int}	{"message": "user updated"}	400 Bad Request [user not found]
104	Admin.DeleteUser	Elimina un usuario del módulo	{"user_name":string}	{"message": "user deleted"}	404 Not Found [user not found] 500 Internal Error [user not deleted from server]
105	Admin.Trigger	Acciona el módulo	{}	{"message": "Triggering gate"}	500 Internal Error [user changed type]
106	Admin.CreateUser	Crea un usuario provisorio, pendiente de aprobación.	{"user_name":string}	{"user_type":int}	400 Bad Request [credentials parameter is required] [credentials error] [user already exists] 412 Precondition Failed [user's max qty reached]
107	Admin.EnableUpdate	Habilita el proceso de actualización del firmware del módulo	{}	{"message": "OTA Update Enabled"}	
108	Admin.SetGateStatus	Cambia el estado de apertura	{"status_code":int}	{"status_code":int}	
109	Admin.GetGateStatus	Devuelve el estado de apertura	{}	{"status_code":int, "opening_percentage":int}	
200	Guest.CreateUser	Crea una solicitud de acceso	{"credentials":string}	{"user_type":int}	IDEM 106 No Aplican Errores 401 y 403
201	Guest.UserStatus	Devuelve el nivel del usuario solicitado	{"user_name":string}	{"user_name":string, "user_type":int}	No Aplican Errores 401 y 403 400 Bad Request [User Name is required] 404 Not Found [user not found]
203	Sys.GetInfo	Devuelve información de sistema del módulo	{}	{"app":string, "fw_version":string, "fw_id":string, "mac":string, "uptime":int}	
300	User.Trigger	IDEM 105			
301	User.GetGateStatus	IDEM 109			
	/update	Envía el nuevo firmware como un POST multipart fijando el parámetro commit_timeout=75 (segundos).			
	/update/commit	Se confirma la correcta actualización ejecutando el RPC OTA.Commit con un POST para evitar el rollback por timeout.			

Cuadro 4.1: Protocolos remotos proporcionados por el firmware del módulo

4.5. Estructura del Proyecto

4.5.1. Unidades Funcionales

Se realizó un análisis de experiencia de usuario en el que se definieron las pantallas que se implementaran en la aplicación móvil. En la figura 4.19 se puede observar las vistas y las conexiones entre ellas.

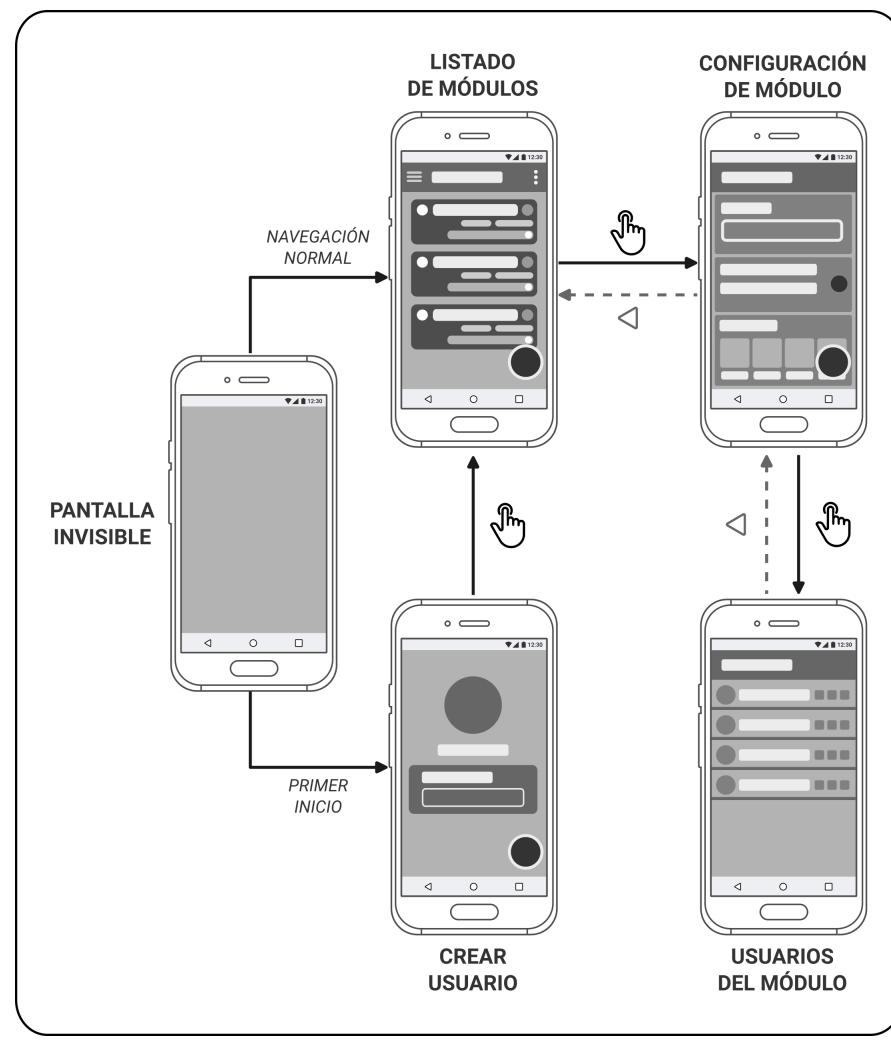


Figura 4.19: Diagrama de flujo de usuario para la aplicación móvil.

Se plantea la definición de unidades funcionales como un conjunto de funcionalidades de la aplicación que están estrechamente relacionadas. Utilizar las pantallas pareciera una buena estrategia para definir estas unidades.

- **Creación de Usuario:** El usuario ingresa su número de teléfono y se crea el objeto App User.

- **Listado de Módulos:** Se buscan todos los Módulos disponibles en los distintos canales de comunicación y se muestran en una lista para permitir su manipulación.
- **Configuración del Módulo:** Permite modificar parámetros del módulo seleccionado.
- **Usuarios del Módulo:** Muestra el listado de usuarios de un Módulo y permite su administración.

4.5.1.1. Plantillas de Arquitectura

Siguiendo los lineamientos de la arquitectura cada unidad respetará el formato arquitectónico ilustrado en la figura 4.20, como puede observarse todos los patrones de diseño están represados en el diagrama de clases.

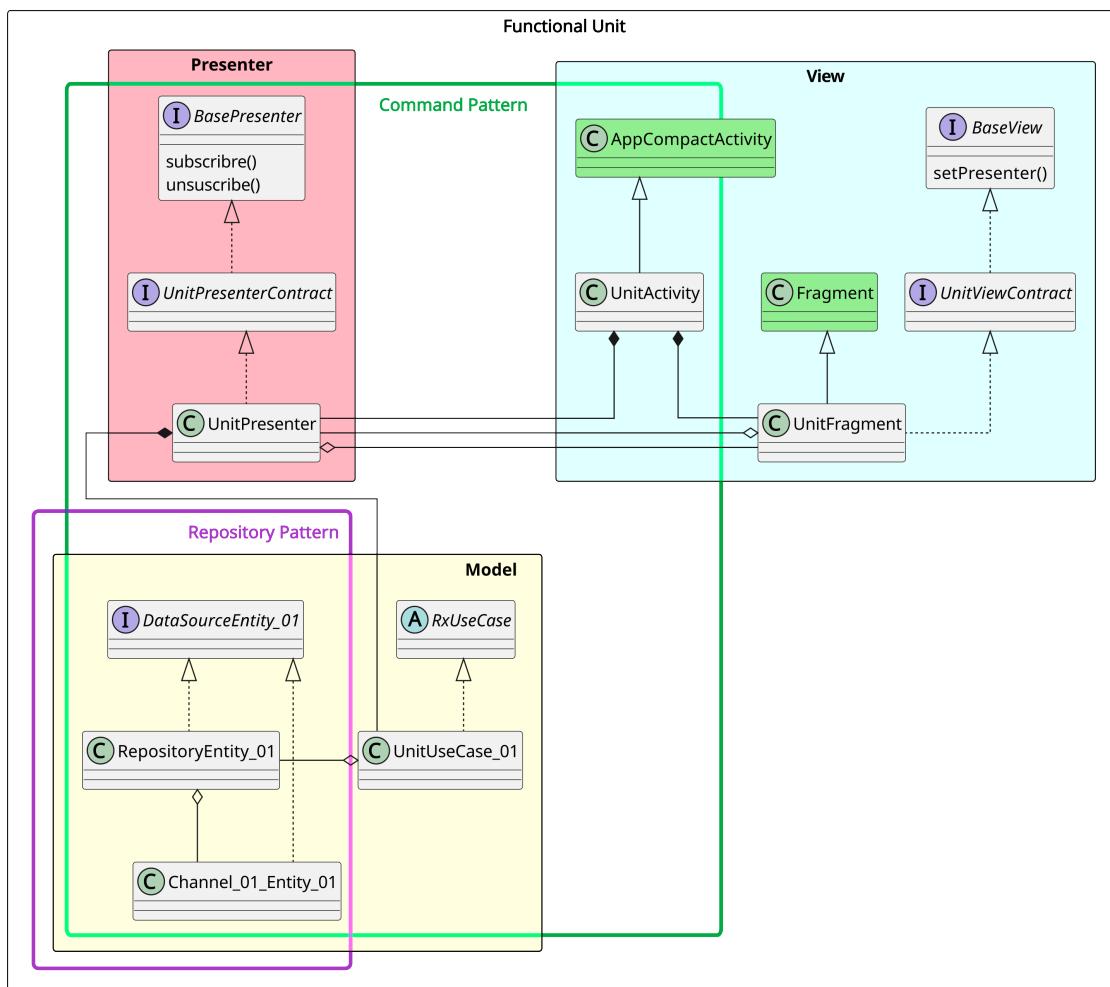


Figura 4.20: Diagrama de clases de una unidad funcional.

Al implementar esta plantilla se observa también que la organización del código fuente produce una jerarquía de directorios similar para cada unidad funcional. En la figura 4.21 se puede observar cómo debería verse cada unidad en el proyecto.

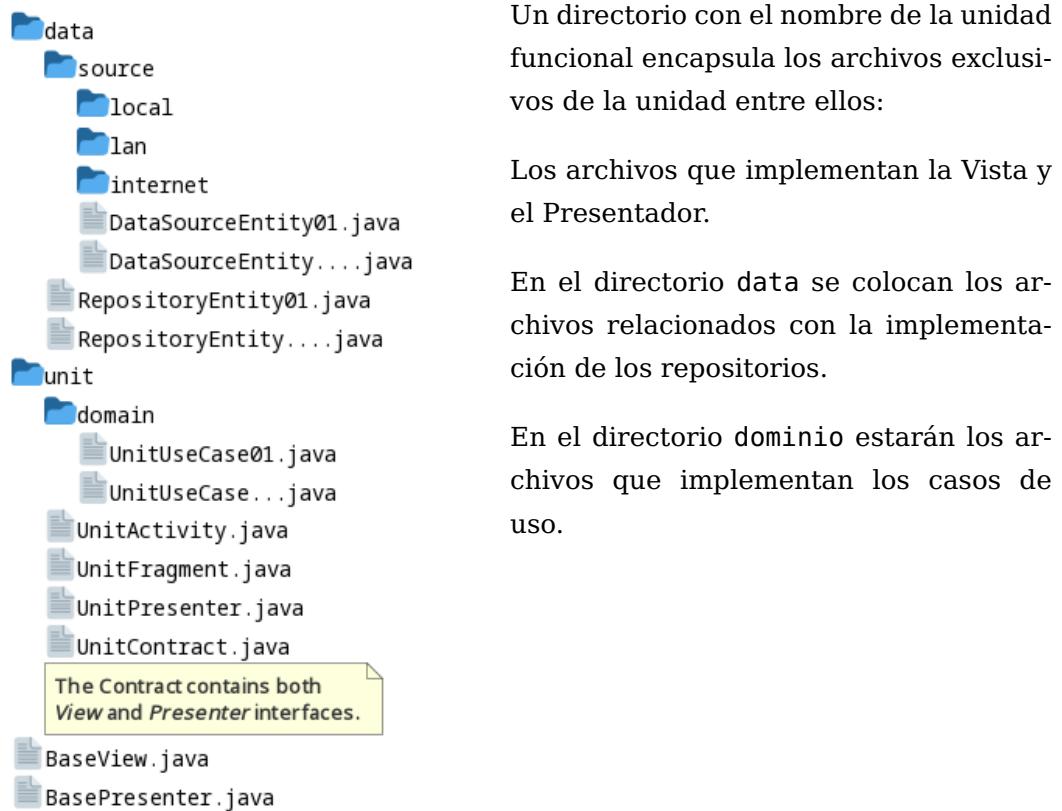


Figura 4.21: Diagrama de clases de una unidad funcional.

4.5.2. Planificación de las Iteraciones

Para garantizar un prototipo mínimo y plantear las subsecuentes mejoras se proponen tres iteraciones de desarrollo acotadas, consecutivas e incrementales. Con este propósito se definieron tres subconjuntos 4.22 sobre los casos de uso documentados en la sección ???. Para cada iteración se codificará todo el andamiaje necesario para construir las unidades funcionales con las primitivas de la arquitectura, los repositorios con sus canales correspondientes, el diseño gráfico de las vistas y los algoritmos de los casos de uso.

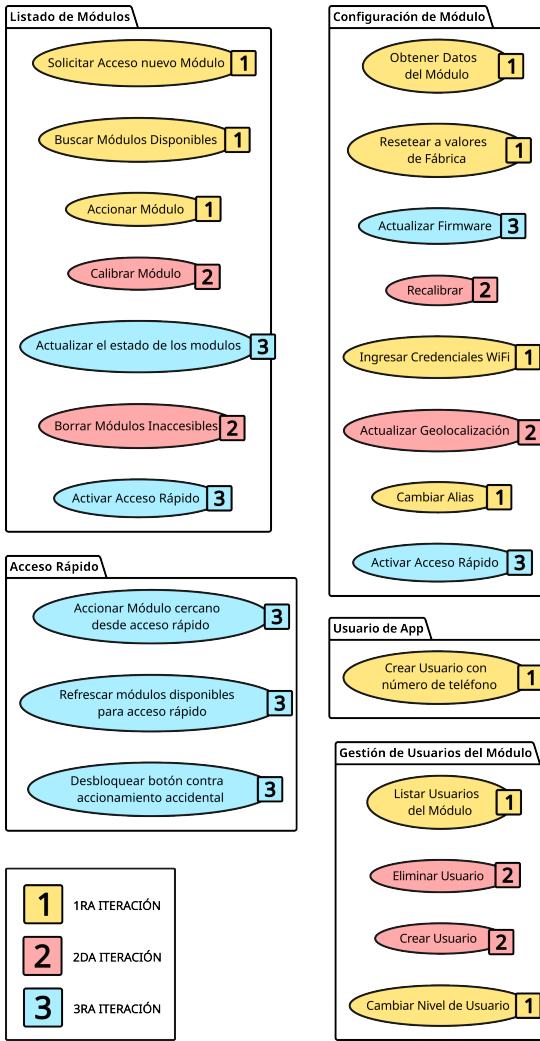


Figura 4.22: Subconjuntos de Casos de Uso para cadaad Iteración.

Se describe una síntesis del alcance de cada iteración:

ITERACIÓN I

- Se plantea y comenta la implementación reactiva y genérica de un Caso de Uso. Se expondrá sobre el uso y funcionamiento de los Schedulers.
- Casos de Uso contemplados para la primera iteración.
- El repositorio de los objetos Módulo. Canales de Persistencia local y Comunicación LAN.

ITERACIÓN II

- Casos de uso contemplados para la segunda iteración.
- El repositorio para obtener la ubicación geográfica del teléfono.
- Se introducen las pruebas unitarias al código base.
- Se configuran, codifican y ejecutan pruebas funcionales automáticas.

ITERACIÓN III

- Motivación e implementación del “Control Rápido”.
- Casos de uso contemplados para la tercera iteración.
- Se documenta la propuesta de comunicación a través de internet.
- Wrapper reactivo sobre una librería de terceros.
- La lógica de selección de canales disponibles para un mismo repositorio.
- Seguridad sobre el canal MQTT.

Capítulo 5

Desarrollo: Iteración I

5.1. Introducción

5.2. Requerimientos

5.2.1. Casos de Uso

5.2.1.1. Configuración Inicial

5.2.1.2. Accionamiento

5.3. Validación

5.3.1. Pruebas Unitarias

5.3.2. Pruebas de Sistema

5.4. Conclusión

Capítulo 6

Desarrollo: Iteración II

6.1. Introducción

6.2. Requerimientos

6.2.1. Casos de Uso

6.2.1.1. Creación de Usuario

6.2.1.2. Ver listado de Usuarios

6.2.1.3. Editado de Usuario

6.2.1.4. Eliminación de Usuario

6.3. Validación

6.3.1. Pruebas Unitarias

6.3.2. Pruebas de Sistema

6.4. Conclusión

Capítulo 7

Desarrollo: Iteración III

7.1. Introducción

7.2. Requerimientos

7.2.1. Casos de Uso

7.2.1.1. Estado de Apertura

7.2.1.2. Notificaciones

7.2.1.3. Actualización Inalámbrica (OTA)

7.3. Validación

7.3.1. Pruebas Unitarias

7.3.2. Pruebas de Sistema

7.4. Conclusión

Capítulo 8

Conclusiones y Trabajos Futuros

8.1. Conclusión

Durante el desarrollo de la presente práctica se entendieron las ventajas de implementar un sistema de software utilizando un patrón de arquitectura. La inspección de código escrito por profesionales del área introdujo conceptos de programación Java avanzados tales como el uso de clases anónimas [9] y la implementación de clases y métodos genéricos[10]. Se pudieron observar las técnicas empleadas para realizar las pruebas sobre el código implementado y fue necesario invertir tiempo en la investigación de las librerías y framework para pruebas (UnitTest, Integration Tests) tales como Mockito [11] (creación de objetos mock(maquetas), stubs(comportamiento forzado) y spies (espías))y Espresso [12] (Simulación e interacción con objetos del framework android).

Luego de estudiar las implementaciones se fueron evidenciando los beneficios de aplicar una arquitectura de estas características. La modularización en componentes con responsabilidades reducidas y bien definidas permite seguir el flujo de ejecución del código con facilidad y como consecuencia directa el rastreo de bugs reduce el radio de ubicación del código involucrado a unas pocas líneas en muy poco tiempo. Dado que la lógica de negocios está encapsulada en la capa de dominio, su ejecución es completamente independiente de los componentes del framework ofreciendo la posibilidad de exportar/traducir la lógica a otros lenguajes, frameworks y sistemas operativos. Tanto los presentadores como las vistas tienen contratos que deberían respetarse en cualquier plataforma por lo que el planteo inicial de la capa de presentación permite el desarrollo en paralelo de implementaciones nativas. El uso de una abstracción de repositorios en la capa de datos permite la inclusión de diversos orígenes de datos o canales que ofrecen mayor flexibilidad al momento de establecer los niveles de redundancia soportados y los esquemas de actualización

disponibles. En una buena implementación, la organización del código en directorios y paquetes debería facilitar la identificación de los componentes de arquitectura y la discriminación de funcionalidades. De manera indirecta se observó que la implementación introduce un procedimiento de trabajo repetitivo tanto para la adición de nuevas funcionalidades como para la remoción de errores y la inspección del código en general.

Como una desventaja notoria se menciona la empinada curva de aprendizaje para la inclusión de nuevos miembros en un hipotético equipo de desarrollo. Así mismo se hizo evidente que todos los conceptos de abstracción que fueron introducidos se traducen en un aumento notable en la cantidad de líneas de código meramente dedicadas a mantener la estructura del diseño pero que no proveen una funcionalidad concreta al sistema.

Finalmente, se observaron inconsistencias entre el planteo teórico de la arquitectura y la implementación real del software mayormente por la dificultad técnica y concesiones que se tuvieron en cuenta para disminuir la verbosidad de algunos componentes o interacciones (e.g. la violación de la regla de dependencias en los casos de uso).

8.2. Trabajos Futuros

Bibliografía

- [1] Robert C. Martin (Uncle Bob). The clean architecture, 2012. URL <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>.
- [2] Tomislav Homan (FIVE). Android architecture who-le series, 2016. URL <https://five.agency/android-architecture-part-1-every-new-beginning-is-hard/>.
- [3] Fernando Cejas. Architecting android...the clean way?, 2014. URL <https://fernandocejas.com/2014/09/03/architecting-android-the-clean-way/>.
- [4] Android Team. Android architecture blueprints [beta] - mvp + clean architecture, 2015. URL <https://github.com/android/architecture-samples>.
- [5] Antonio Leiva. Mvp for android: how to organize the presentation layer, 2018. URL <https://antonioleiva.com/mvp-android/>.
- [6] James Sugrue. Java anonymous class, 2010. URL <https://dzone.com/articles/design-patterns-command>.
- [7] Wolfgang Ofner. Repository and unit of work pattern, 2018. URL <https://www.programmingwithwolfgang.com/repository-and-unit-of-work-pattern/>.
- [8] Andriy Druk. Bonjour in android applications, 2015. URL <https://andriydruk.com/post/mdnsresponder/>.
- [9] Pankaj. Java anonymous class, 2018. URL <https://www.journaldev.com/12534/java-anonymous-class>.
- [10] Cecilio Álvarez Caules. Uso de java generics, 2014. URL <https://www.arquitecturajava.com/uso-de-java-generics/>.
- [11] Fabian Pfaff (Vogella) Lars Vogel. Unit tests with mockito, 2018. URL <http://www.vogella.com/tutorials/Mockito/article.html>.
- [12] Maksim Akifev. Android automation testing: Getting started with espresso, 2018. URL <https://medium.com/@akifev/android-automation-testing-getting-started-with-espresso-a6f8cb50746a>.