UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE ELECTRÓNICA
VALPARAÍSO - CHILE

"DESARROLLO DE APLICACIÓN MÓVIL DE ADQUISICIÓN DE SIGNOS VITALES PARA SISTEMA DE MONITOREO REMOTO"

NICOLÁS FELIPE ANTONIO MIRANDA VALENCIA

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELECTRÓNICO MENCIÓN COMPUTADORES

PROFESOR GUIA: **AGUSTIN GONZÁLEZ V.**

PROFESOR CORREFERENTE: **RUDY MALONNEK W.**

JULIO-2025

CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título; ☐ Tesis de Postgrado;

Título del trabajo: Desarrollo de aplicación móvil de adquisición de signos vitales para sistema de monitoreo remoto

Nombre del candidato(a): Nicolás Felipe Antonio Miranda Valencia

Carrera / Grado: Ingeniería Civil Electrónica

Campus: Casa Central Valparaiso**;** **Departamento**: Electrónica

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Agustín J. González Valenzuela, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL

☒ El trabajo **NO contiene información que amerite confidencialidad** y puede ser publicado de inmediato en repositorio con acceso abierto.

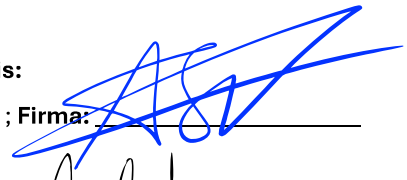
☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (embargo) por:

☐ 6 meses; ☐ 12 meses; ☐ 2 años; ☐ 3 años; ☐ 5 años; ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 07.10.2025 ; **Firma:** 

Estudiante o Candidato(a):

Fecha: 07.10.2025 ; **Firma:** 

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

Desarrollo de aplicación móvil de adquisición de signos vitales para sistema de monitoreo remoto

Miranda Valencia Nicolás Felipe

Memoria de titulación para optar al título de Ingeniero Civil Electrónico mención Computadores

Universidad Técnica Federico Santa María

Profesor Guía: Agustín González V.

Julio 2025

Resumen

En casos de accidentes, la disponibilidad del estado de salud de un paciente mientras es trasladado a un centro de atención es de gran importancia para un tratamiento de emergencia exitoso. Por lo que sistemas responsables de recoger, guardar y distribuir datos de signos vitales en tiempo real son fundamentales para una medicina moderna.

En este desarrollo se presenta una actualización y modernización de la aplicación ERPHA Mobile creada hace años, para la adquisición y comunicación de signos vitales, esta trabaja en conjunto con la plataforma ERPHA Server y ERPHA Monitoring, que también fueron modernizadas por otros estudiantes, para ofrecer un servicio de monitoreo de signos vitales en tiempo real.

La aplicación ERPHA Mobile se encarga de comunicar y leer mediciones en tiempo real provenientes de tres sensores médicos inalámbricos: presión, oximetría de pulso y ECG, para comunicarlas al servidor ERPHA Server. Además, la aplicación tiene la capacidad de enviar ubicación continua, fotografías e información básica de un paciente.

Palabras Claves: Sensores médicos, Aplicación móvil, Telemedicina, Signos vitales

Development of mobile application for vital signs acquisition for remote monitoring system

Miranda Valencia Nicolás Felipe

Final Project Report towards the partial fulfillment of the requirements of the Electronic Engineering Degree, majoring in Computers

Universidad Técnica Federico Santa María

Advising Professor: Agustín González V.

July 2025

Abstract

In cases of accidents, the availability of information about a patient while they are being transported is of great importance for successful emergency treatment. Therefore, systems that are responsible for collecting, storing and distributing measurements such as vital signs in real time are essential for modern medicine.

The work developed exists as an update and modernization of the ERPHA Mobile application created years ago for the acquisition and communication of vital signs. This works in conjunction with the ERPHA Server and ERPHA Monitoring platforms, which were also modernized by other students to offer a vital signs monitoring service in real time.

The ERPHA Mobile application is responsible for communicating and reading measurements in real time from three wireless medical sensors to communicate the data to the ERPHA Server. In addition, the application has the ability to send continuous location, photographs and basic information about a patient.

Keywords: Medical sensors, Mobile application, Telemedicine, Vital signs

Glosario

ERPHA : Emergency Remote Pre-Hospital Assistance (Asistencia Remota Pre-Hospitalaria para Emergencias).

ECG : Electrocardiograma.

UI : User Interface (Interfaz de usuario).

IDE : Integrated development environment (Entorno de desarrollo integrado).

TCP : Transmission Control Protocol (Protocolo de control de transmisión).

HTTP : Hypertext Transfer Protocol (Protocolo de transferencia de hipertexto).

MQTT : Message Queuing Telemetry Transport. Un protocolo de mensajería ligera.

AMQP : Advanced Message Queuing Protocol (Protocolo de cola de mensajes avanzado).

MVVM : Model-View-ViewModel. Estructura de software recomendada para aplicaciones móviles complejas.

Bluetooth : especificación industrial para redes inalámbricas de área personal.

SPP : Serial Port Profile (Perfil de puerto serie). Perfil estándar de comunicaciones a través de Bluetooth

Índice general

Índice de figuras

1. INTRODUCCIÓN

Al momento de una emergencia médica en que es necesaria la presencia de paramédicos para el traslado del paciente al centro médico disponible más apropiado, la existencia de información sobre los signos vitales en tiempo real es crítica para el tratamiento de emergencia y una eventual recuperación exitosa. Por esta razón existen distintos servicios y tecnologías de telemedicina.

En el año 2013 se completó el proyecto sistema ERPHA[1] como una alternativa para monitorear signos vitales de forma remota. Este sistema consta de tres componentes: ERPHA Mobile para recoger y enviar mediciones de signos vitales, ERPHA Server para recibir, guardar y dar acceso a los datos, y ERPHA Monitoring para solicitar la información a ERPHA Server y visualizar los signos vitales e información de un paciente en tiempo real.

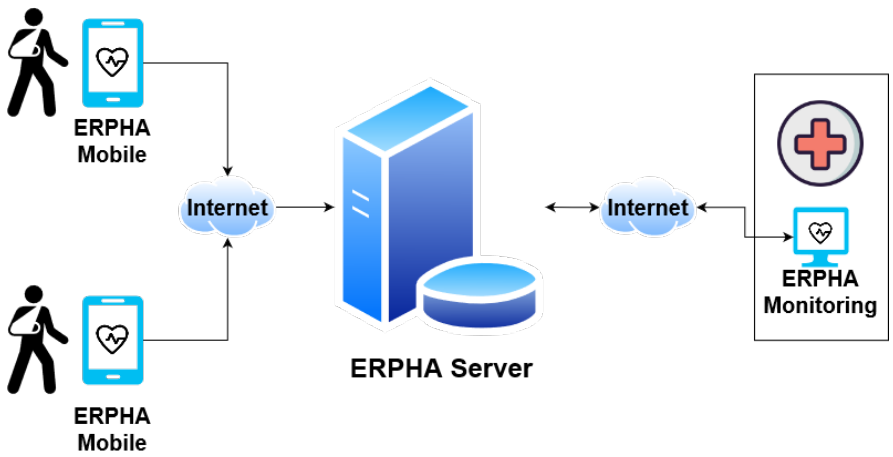


Fig. 1.1: Estructura general sistema ERPHA.

Actualmente casi 10 años después, existe el problema de que muchas de las tecnologías utilizadas originalmente para el sistema están obsoletas o ya no tienen soporte, haciendo que los tres componentes del sistema necesiten una actualización para trabajar con las plataformas modernas. Este trabajo tiene como objetivo modernizar la aplicación móvil ERPHA Mobile para poder utilizarse con las herramientas modernas disponibles, y la nueva versión, también actualizada, de ERPHA Server.

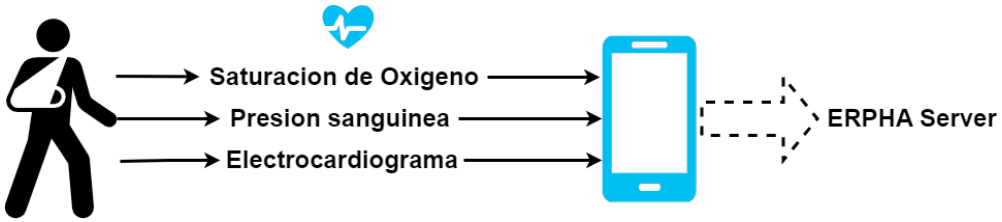


Fig. 1.2: Estructura general ERPHA Mobile.

2. DISEÑO DE APLICACIÓN MÓVIL DE ADQUISICIÓN DE SIGNOS VITALES

El objetivo de este trabajo es desarrollar una aplicación móvil capaz de utilizar ciertos sensores médicos inalámbricos para recoger mediciones periódicas de signos vitales de un paciente. Todo esto con la finalidad de enviar los datos recogidos a un servidor dedicado a distribuir la información, para lograr una mejor atención de una emergencia médica.

La aplicación móvil para la adquisición de signos vitales del sistema de monitoreo remoto tiene los siguientes requisitos mínimos de funcionamiento para trabajar en conjunto con el servidor y el equipo disponible:

- Utilizar el sensor de presión sanguínea **"boso-medicus prestige"** para obtener mediciones de presión **diastólica** y **sistólica** en tiempo real y comunicar los resultados al servidor correspondiente con un formato específico.
- Utilizar el sensor de saturación de oxígeno **"nonin oximeter 4100"** para obtener mediciones periódicas de la **saturación de oxígeno** y **frecuencia cardiaca** en tiempo real y comunicar los resultados al servidor correspondiente con un formato específico.
- Utilizar el sensor de electrocardiograma (ECG) **"corscience bt3/6"**, para obtener dos canales de mediciones de la actividad eléctrica del corazón denominadas: ***Derivación II*** y ***Derivación III*** en tiempo real y comunicar los resultados al servidor correspondiente con un formato específico. Estas señales representan la diferencia de potencial eléctrico entre puntos de la caja torácica.

(a) Sensor de presión sanguínea, **boso-medicus prestige**.

(b) Sensor de saturación de oxigeno, **nonin oximeter 4100**.

(c) Sensor ECG, **corscience bt 3/6**.

Fig. 2.1: Sensores médicos para aplicación móvil

- Enviar ubicación geográfica del equipo móvil, **latitud** y **longitud**, periódicamente al servidor correspondiente.
- Permitir al usuario tomar una fotografía con el equipo móvil y enviar el archivo al servidor correspondiente.
- Permitir al usuario ingresar información básica de un paciente, nombre, número de identificación y sexo, para enviar al servidor correspondiente.

2.1. Trabajo previo al desarrollo de la aplicación

El desarrollo de una aplicación para plataforma móvil necesita de decisiones previas al diseño de clases e implementación en código. En este caso, la aplicación requiere de un dispositivo con módulos de **Locación**, **Cámara**, **Bluetooth** y acceso a **Internet**, por lo que la plataforma ideal sigue siendo el teléfono móvil, el Smartphone.

Dentro de la familia de teléfonos móviles modernos se debe elegir el sistema operativo donde programar, donde hay que tomar en cuenta el protocolo de comunicación con el servidor, el entorno de uso y la accesibilidad a recursos para la programación.

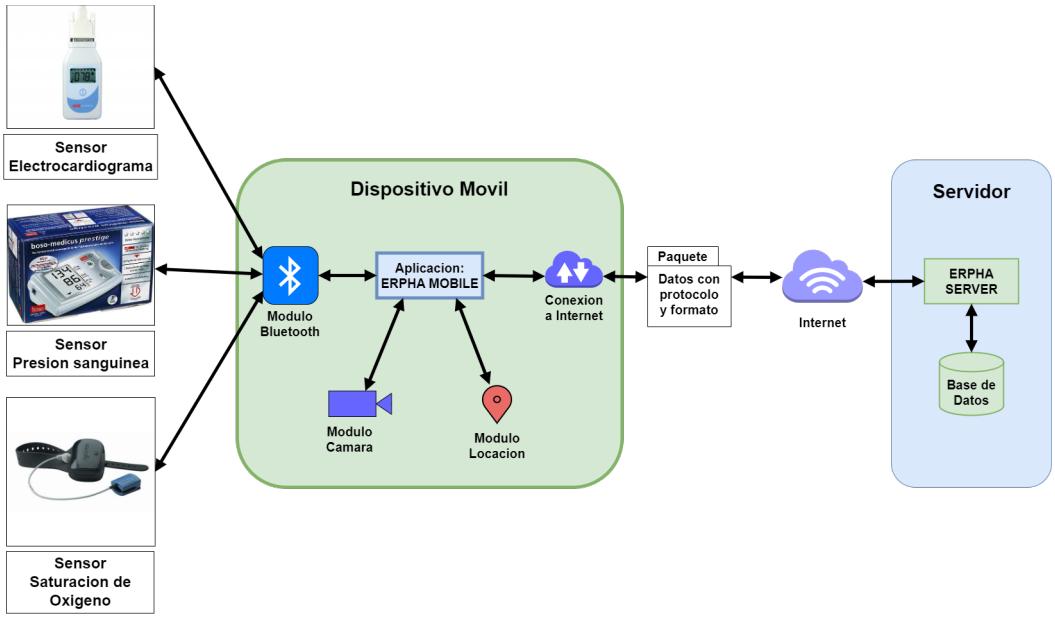


Fig. 2.2: Diagrama de componentes de la aplicación.

2.1.1. Sistema operativo

Dentro los teléfonos móviles disponibles existen dos grandes familias populares con capacidades similares, las de sistema operativo **iOS** y las de sistema operativo **Android**. Ambas opciones juntas conforman la gran mayoría del mercado de celulares en el mundo y dependiendo del país existe una distinta distribución de los usuarios entre los sistemas operativos, pero específicamente en Chile, año 2024, la mayoría de los usuarios utilizan el sistema **Android**[2].

Para el desarrollo de esta aplicación a pesar de que ambas familias tienen todas las capacidades necesarias, el sistema **Android** es opción preferida. Este es más accesible para desarrolladores y no tiene restricciones para obtener o usar el software de desarrollo Android.

Dentro del ecosistema de Android también es necesario definir la API mínima de la aplicación. Esto significa definir que tan antiguos son los teléfonos permitidos que

podrán ejecutar la aplicación sin ningún problema. Teniendo en cuenta los teléfonos móviles disponibles para pruebas y la gran presencia actual de estos equipos[3], se impuso la **API 30**, o en otras palabras **Android 11.0** como mínima. De esta forma todavía se tiene acceso a posibles redes 5G (Android 10.0 mínimo[4]).

Utilizando las recomendaciones oficiales para desarrolladores de Android se puede usar la **IDE Android Studio**, que permite crear fácilmente el archivo ejecutable **.apk** y manejar todas las dependencias. También existe el lenguaje de programación alternativo a JAVA y sus librerías, **Kotlin** que permite tener la posibilidad de utilizar su herramienta oficial para desarrollo de interfaz de usuario en Android, **Jetpack Compose**.

2.1.2. Protocolo de comunicación con el servidor

La conexión entre la aplicación y servidor debe garantizar la llegada de los paquetes, no es deseable perder paquetes con información de signos vitales, ya que todo dato histórico de un paciente es beneficioso, ya sea para el tratamiento de emergencia frente a un accidente o posteriormente para lograr una recuperación exitosa. Debido a esto, se tiene como opciones el **Protocolo de Control de Transmisión (TCP)[5]**, **Protocolo de transmisión de control de flujo (SCTP)[6]** y **Conexiones UDP rápidas en Internet (QUIC)[7]**.

- **TCP**: Protocolo de capa de transporte que permite una comunicación segura entre máquinas y asegura la llegada de paquetes al receptor, pero tiene dificultades al intentar enviar múltiples flujos de datos a través de una sola conexión, esto provoca posibles retrasos en los paquetes debido al problema llamado **bloqueo de cabecera de línea**. TCP es uno de los protocolos más populares en la internet.

- **SCTP**: Protocolo de capa de transporte creado como alternativa para solucionar problemas de TCP. Tiene la capacidad extra de permitir **paquetes fuera de orden** al poder crear distintos flujos independientes dentro de una misma conexión,

también permite dirigir los paquetes a múltiples destinos IP. A pesar de su utilidad, su adopción se limita a ciertas aplicaciones específicas de la industria de comunicaciones.

- **QUIC**: Protocolo de capa de transporte basado en las características del **Protocolo de datagramas de usuario (UDP)**. Apareció como un reemplazo de TCP para mejorar el tiempo de comunicación de las aplicaciones web y permitir múltiples flujos de datos en una sola conexión. Actualmente, es parte del protocolo **HTTP/3**, permitiendo la comunicación de múltiples elementos de una página web de forma simultánea, rápida y con la seguridad de TCP.

Entre las alternativas, el protocolo TCP otorga una mayor cantidad de beneficios si se utiliza múltiples conexiones TCP independientes para solucionar el **bloqueo de cabecera de línea** a costo de rendimiento. Como beneficio se tiene acceso a múltiples bibliotecas externas para implementaciones de protocolos de capa de aplicación, tanto para ERPHA Mobile como para ERPHA Server, ya probados y a prueba de errores.

Otro punto a tomar en cuenta es los estándares modernos de seguridad, es importante al menos tener un método de identificación de usuario antes de poder comunicar las mediciones al servidor (para evitar o poder enfrentar conexiones maliciosas). Igualmente, la tasa de transmisión de datos es un factor a tomar en cuenta para preocupaciones como uso de batería y planes de datos, especialmente en los que casos que se envían paquetes de forma periódica con pequeñas cantidades información, por esto un protocolo de capa de aplicación con menor encabezado es vital para una menor transmisión de datos.

Tomando en cuenta todas las necesidades de comunicación con el servidor, ERPHA Server, las principales alternativas de protocolos de comunicación estudiadas para el envío frecuente de paquetes con datos de mediciones a través del internet, resultó con protocolos orientados a IoT (internet de las cosas) como: **HTTP/1.1**[8], **MQTT**[9] y

AMQP[10].

De las opciones disponibles se considera **MQTT** como la mejor opcion a seleccionar, este protocolo está pensado específicamente para aplicaciones de bajo ancho de banda, utilizando un encabezado pequeño, con una estructura de implementación más ligera y simple que AMQP. Ya que en esta aplicación la comunicación es casi enteramente unidireccional, las opciones disponibles más avanzadas para el esquema cliente-servidor de AMQP no son necesarias y el esquema cliente-corredor, figura 2.3, de MQTT es suficiente.

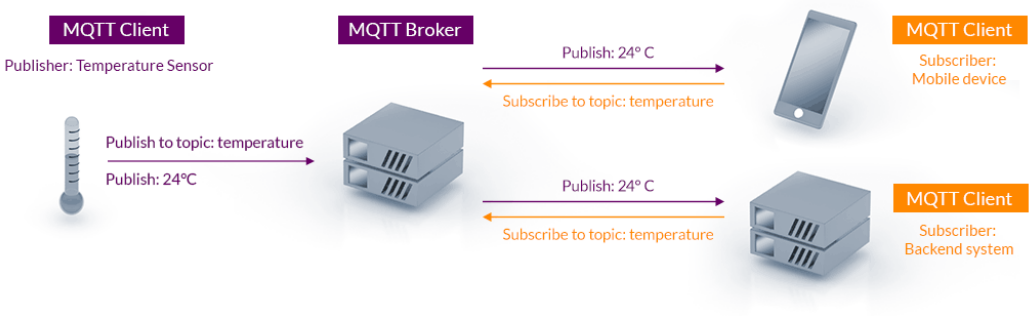


Fig. 2.3: Arquitectura general publicar/suscribir de MQTT[9]

Para la implementación de un cliente MQTT en sistema Android existen dos opciones confiables de uso libre para desarrolladores, **Eclipse Paho MQTT**[11] y **HiveMQ MQTT**[12]. Ambas tienen soporte para la implementación de clientes o corredores, pero **HiveMQ** se destaca debido a que tiene soporte para utilizar la versión MQTT 5.0 con opciones utiles como *time-to-live* y mensajes de despedida al desconectar un cliente. Además, en esta versión las sesiones de los clientes no se cierran automáticamente al momento de la desconexión, sino que se espera un tiempo determinado antes de descartar la sesión. Esto permite que la aplicación se vuelva a conectar rápidamente frente a problemas de red, por ejemplo el traslado de un paciente transita por un sector con conexión inestable.

2.1.3. Herramientas para interfaz de usuario

El desarrollo de aplicaciones en sistema Android pone mucha importancia en la interfaz de usuario (UI) y cómo el usuario es capaz de interactuar fácilmente con sus manos. Es por eso que existen diversas herramientas para ayudar a los desarrolladores a organizar y manipular los elementos visuales de la aplicación.

La opción más conocida y simple para comenzar es el **lenguaje de marcado extensible (XML)** que permite declarar todos los elementos de la UI y su organización en un archivo externo, para luego ser manipulados con el código de la aplicación (en este caso **Kotlin**). Pero si se quiere aprovechar el hecho de utilizar el lenguaje de programación **Kotlin**, existe su paquete de herramientas llamado **Jetpack Compose**[13].

Jetpack Compose es una herramienta oficial de *Google* que utiliza **Kotlin** para declarar los elementos de UI en el código utilizando funciones con la etiqueta *@Composable*. Esto permite manipular fácilmente los elementos visuales a medida que se ejecuta la aplicación debido a que todos los componentes son funciones dentro del mismo codigo.

Con esta herramienta se dispone de múltiples componentes previamente construidos para organizar o mostrar información, y al usarse en conjunto se pueden crear nuevos bloques (funciones) que son capaces de reutilizarse en múltiples secciones del código y con distintos datos de entrada. Además, se tiene acceso a otros tipos de herramientas para controlar la UI, como cambio fácil entre pantallas y lo más importante, compatibilidad con clases nativas para utilizar arquitecturas de software recomendadas como Modelo–vista–modelo de vista (MVVM).

2.2. Flujo de funcionamiento de la aplicación

El flujo de funcionamiento de la aplicación en una situación normal desde la perspectiva del **usuario** es:

1: **Usuario** abre la aplicación.
2: **Usuario** inicia el descubrimiento de los sensores a través de Bluetooth con un **botón**.
3: **Usuario** ingresa por pantalla la información requerida del paciente.
4: **Usuario** inicia conexión y transmisión al servidor con un **botón**.
5: **Usuario** ingresa credenciales del servidor por pantalla.
6: **Aplicación** envía la información del paciente al servidor.
7: **Aplicación** inicia el módulo de locación y comienza el envío periódico de coordenadas latitud-longitud al servidor.
8: **Aplicación** comienza conexión con sensores y comienza la retransmisión de mediciones al servidor.
9: **Usuario** utiliza la cámara y hace el envío de una fotografía mediante un **botón**.
10: **Usuario** detiene la comunicación con el servidor y los sensores con un **botón**.

Internamente, la aplicación separa los procesos de retransmisión de mediciones en distintas *corrutinas asincrónicas*, estas actúan como una versión propia de hilos en Kotlin. Además, antes de comenzar la transmisión de datos la aplicación por temas de seguridad siempre necesita asegurarse de la disponibilidad de permisos para la utilización de los modulos de: **Camara**, **Locación** y **Bluetooth**.

El funcionamiento interno de la aplicación tomando en cuenta sus módulos internos se describe con la siguiente figura 2.4.

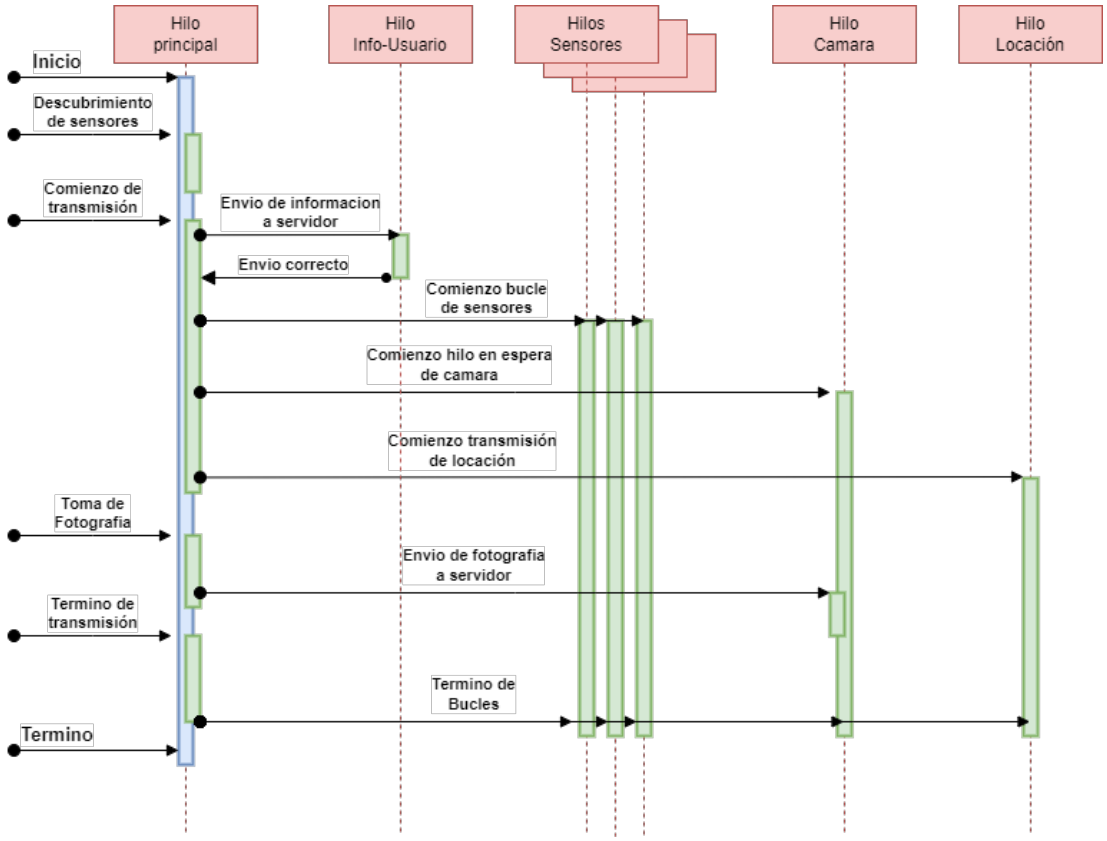


Fig. 2.4: Diagrama de procesos de la aplicación.

2.2.1. Estructura de clases

Al hacer uso del lenguaje **Kotlin** con su *toolkit* **Jetpack Compose**, tanto la UI como la lógica interna de la aplicación serán archivos Kotlin y por tanto clases en este caso.

Siguiendo la estructura de software **Modelo-Vista-Modelo de Vista** (MVVM), la aplicación se puede dividir en tres grandes secciones:

1. Modelo (*Model*): Se refiere a todas las clases dentro de la aplicación que hacen el trabajo, es decir producen o guardan datos. Este trabajo incluye comunicaciones con los sensores, servidor, cámara, servicios internos de Bluetooth y los servicios de locación.

2. Vista (*View*): Son las clases que describen la UI y solo esta, esto no incluye la información variable a mostrar por pantalla. Estas clases conforman las pantallas disponibles a mostrar y todos los elementos que muestran u organizan los datos relevantes.

3. Modelo de vista (*View-model*): Esta clase controla la comunicación entre las otras dos secciones. Reacciona frente a presionados de botones originados por la Vista e indica que rutina la capa de Modelo debe ejecutar, o en caso contrario comunica los datos originados por la capa de Modelo hacia los componentes de la Vista.

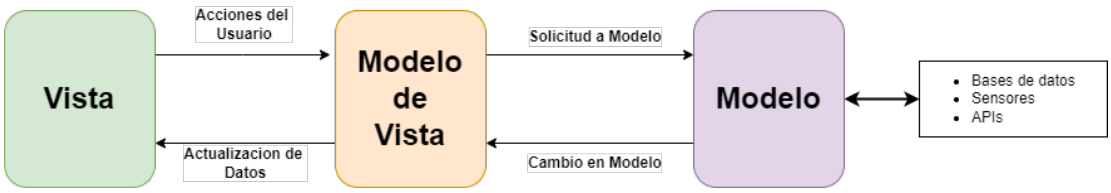


Fig. 2.5: Diagrama de funcionamiento para arquitectura MVVM.

La implementación de esta estructura es posible debido a la clase nativa de *Kotlin*, **ViewModel()**. Esta clase permite el uso de *StateFlow* (una clase para guardar datos) que es capaz de informar inmediatamente a componentes de la UI que estén usando sus valores cualquier cambio de forma automática, esto provoca que se recalculen las funciones de *Compose* y, por tanto, todos sus datos a mostrar por pantalla.

De forma específica las clases personalizadas que son necesarias para la aplicación según cada sección son las siguientes:

- **Modelo (*Model*):**

    1. ***Bluetooth_Handler***: Clase que se encarga de **descubrir** y **emparejar** el teléfono movil con los sensores inalámbricos bluetooth. Necesita los servicios de bluetooth y de locación activados.

2. ***Patient_Info***: Clase que contiene un cliente-publicador MQTT personal, con una sola conexión TCP dedicada, que se conecta y comunica con el servidor al momento de iniciar la transmisión de datos para enviar la información del paciente ingresada por el usuario a través de la UI.

3. ***Location***: Clase que contiene un cliente-publicador MQTT personal y que comunica periódicamente las coordenadas geográficas en formato decimal del equipo móvil que ejecuta la aplicación.

4. ***Sensor_ECG***: Clase que se conecta y comunica con el sensor inalámbrico de ECG, **corscience bt3/6**, para obtener y reenviar al servidor todas las mediciones mediante su propio cliente-publicador MQTT.

5. ***Sensor_presión***: Clase que se conecta y comunica con el sensor inalámbrico de presión sanguínea, **boso-medicus prestige**, para obtener y reenviar al servidor las mediciones mediante su propio cliente-publicador MQTT.

6. ***Sensor_oxigeno***: Clase que se conecta y comunica con el sensor inalámbrico de saturación de oxígeno, **nonin oximeter 4100**, para obtener y reenviar al servidor todas las mediciones mediante su propio-publicador MQTT.

7. ***Camara***: Clase que recibe la ubicación de archivo de la fotografía recién tomada por el usuario. Usa su propio cliente-publicador MQTT para enviar la foto por partes, ya que resoluciones modernas crean archivos de fácilmente 10 MegaBytes que pueden superar el tamaño máximo de un paquete MQTT para configuraciones estándar de un servidor y acaparar los recursos del sistema por mucho tiempo para una sola tarea ininterrumpida.

8. ***MQTT_Client***: Clase utilizada por los demás módulos transmisores. Crea y contiene un cliente MQTT con ciertas credenciales, es capaz de conectarse y publicar datos a los tópicos correspondientes indicados. No tiene necesidad de suscribirse o recibir paquetes desde el servidor MQTT.

- **Vista (*View*):** Una sola clase que incluye distintas pantallas que puede mostrar la aplicación. La navegación entre estas es gracias a la herramienta **navControler** que permite a botones creados o al botón nativo **"Atrás"** poder alternar las pantallas que ve el usuario.

  Esta clase también se encarga de revisar los permisos necesarios al momento de iniciar alguna tarea y mostrar ventanas de diálogo para pedirlos al usuario, o informarles que no es posible ejectar la tarea por falta de permisos.

  1. ***Pantalla Inicial***: Vista de inicio que muestra:
     - El estado de conexión de los sensores y sus mediciones instantáneas.
     - Botón para comenzar la comunicación de datos con el servidor.
     - Botón para utilizar la cámara.
     - Botón para descubrir y emparejar los sensores.
     - Campos de texto para ingresar información del paciente.

     Estos botones se activan y se desactivan dependiendo del estado de la aplicación:

     1. Sensores no-descubiertos.
     2. Descubriendo sensores.
     3. Esperando comienzo de transmisión.
     4. Conectando al servidor.
     5. Comunicando datos al servidor.

Fig. 2.6: Prototipo de pantalla inicial de UI.

2. ***Pantalla de cámara***: Al presionar el botón de cámara se cambia la vista para tomar una fotografía. Solo se muestra el contenido visto por la cámara y un botón para tomar la fotografía.
3. ***Pantalla de confirmación***: Vista que aparece luego de tomar una fotografía. En esta sección se muestra la foto tomada y se tiene botones para, ya sea enviar la fotografía al servidor y volver al inicio, o usar la cámara otra vez para tomar otra foto.

- **Modelo de vista (*ViewModel*):** Corresponde a una clase que contiene el estado de la aplicación y dirige las acciones posibles que puede tomar el usuario. Utiliza una clase llamada *StateFlow* para implementar un **UIState** que contenga múltiples variables para guardar, modificar y comunicar a toda la aplicación el:
  - Estado de emparejamiento de los sensores.
  - Estado de conexión de los sensores.

- Estado de activación de los botones. Estos valores dependen del estado total calculado en la aplicación.
- Últimos valores de medición de los sensores.

El *ViewModel* además contiene todas las funciones que puede llamar el usuario mediante el presionado de los botones. Estas funciones utilizan las clases de *Modelo* para:

- Comenzar proceso de descubrimiento de sensores llamando a *Bluetooth_Handler* en su propia *corrutina asincrónica* (siempre se debe evitar bloquear el hilo de *Vista*). Luego espera los resultados y actualizar el estado de los sensores.
- Comenzar la comunicación con el servidor creando nuevas *corrutinas asincrónicas* e iniciando los procesos de locación, información de paciente, cámara y sensores. Estos procesos se mantienen en un bucle hasta ser detenidos.
- Enviar señales para detener todos las *corrutinas asincrónicas* en bucle que producen datos.
- Responder frente a la captura de una nueva fotografía. Esta foto debe ser guardada en la memoria reservada para la aplicación.
- En caso de que se confirme la fotografía tomada y se quiera enviar, debe invocar a la clase *Cámara* para publicar la foto anteriormente guardada.

3. IMPLEMENTACIÓN DE MÓDULOS DEL SISTEMA

3.1. Comunicación con servidor

La comunicación con el servidor hace uso del protocolo *Message Queuing Telemetry Transport*(MQTT) sobre TCP, con múltiples conexiones simultaneas. La implementación de los clientes MQTT utiliza la implementación de HiveMQ para Android.

Cada fuente de datos o sensor crea y maneja un cliente propio con la clase interna **MQTTClientHiveMQ**. De forma interna cada cliente MQTT se crea con la siguiente función:

```
1   mqtt5Client = Mqtt5Client.builder()
2                   .identifier(ID_cliente)
3                   .serverHost(direccion_broker)
4                   .serverPort(puerto_broker)
5                   .sslWithDefaultConfig()
6                   .simpleAuth()
7                   .username(nombre_de_usuario)
8                   .password(contrasena.toByteArray())
9                   .applySimpleAuth()
10                  .build();
```

Listing 3.1: Codigo para contruir cliente MQTT 5.0 con HiveMQ.

- **.identifier():** corresponde a un identificador personal para el cliente al conectarse con el servidor. Cada módulo que se comunica con el servidor tiene una ID distinta que incluye su función y el identificador único y seguro **Settings.Secure.ANDROID_ID**.

Este identificador se crea combinando llaves de cifrado del teléfono móvil y de la aplicación[17] de modo que no pueda ser modificado por el usuario.

- **.serverHost():** la dirección del servidor, permite formato *link*.
- **.serverPort():** puerto del servidor. Por defecto se usa el puerto **8883**, o *8884* si se quiere utilizar la convención para un cliente web.
- **.sslWithDefaultConfig():** declara que se utilizara el protocolo TLS para poder autenticar el servidor y encriptar los mensajes.
- **.simpleAuth():** declara que la conexión usara el método simple de autenticación, este necesita las credenciales de **nombre de usuario** y **contraseña**.

La aplicación solo necesita que el usuario ingrese la información de **nombre de usuario** y **contraseña**. Esta credenciales se compartirán con todos las clases de la categoria Modelos 2.2.1 que proveen datos al servidor, toda la demás información necesaria para la conexión se crea o rescata de forma interna sin necesidad del usuario.

Los clientes MQTT creados en la aplicación móvil solo tienen la capacidad de publicar información, estos procesos se hacen de forma bloqueante dentro de sus *corrutinas asincrónicas*. La publicación además necesita un **Tópico** y una **Carga útil (*payload*)**.

- **Carga útil:** Contiene la información a transmitir, puede ser los Bytes de una fotografía, o un archivo en formato **JSON** para comunicar los campos de información necesarios. El unico campo presente en todos los archivos **JSON** a transmitir es **timestamp_ms** que corresponde a la marca de tiempo en milisegundos en formato UNIX de la creación de la carga útil.

```
{
    "timestamp_ms" : UNIX_timestamp,
    "latitude"  : -33.02062,
    "longitude" : -71.35432
}
```

Listing 3.2: Paquete JSON de ejemplo para módulo de Locación, con saltos de linea para lectura.

- **Topico:** El envió del paquete necesita una dirección de recepción dentro del mismo servidor. El tópico es una colección de *Strings* normalmente separados por el carácter *"/"* de la misma forma que la ruta de un archivo. Esta es una ventaja que en esta situación permite clasificar e identificar los paquetes recibidos en el servidor diferenciando entre **pacientes**, **equipos móvile**s y **fuentes de datos**.

```
topico_locacion = "11111111-1/${ANDROID_ID}/location"
```

Listing 3.3: Tópico de ejemplo para módulo de Locación.

3.2. Implementación de módulos con conexión Bluetooth

El sistema operativo Android tiene capacidad para utilizar Bluetooth desde sus primeras versiones publicadas. La tecnología de Bluetooth en general tiene distintas versiones, pero para esta aplicación los sensores disponibles se comunican mediante *Classic Bluetooth* a través un sistema de canales RFCOMM simulado llamado el *Perfil de puerto serie* (SPP).

La comunicación a través de SPP significa que al abrir conexión con un sensor, se obtiene un *Bluetooth Socket* donde se envían y reciben *bytes* a través de las funciones típicas de comunicación, *write() y read()*. Cada dispositivo tiene una lista de instrucciones y opciones personales para su configuración haciendo necesario conocer

su documentación o en este caso tambien se tiene disponible la implementación de los sensores en el trabajo previo del Sistema ERPHA[14][15][16] como ejemplo.

3.2.1. Módulo Bluetooth

Previamente a establecer una conexión Bluetooth SPP el dispositivo móvil necesita obtener una dirección y ser reconocido por el sensor inalámbrico objetivo. estos dos pasos son llamados **Descubrimiento** y **Emparejamiento** respectivamente.

Al momento que el usuario presiona el botón *"conectar sensores"* se inicia la rutina para descubrir y emparejar. Comenzando desde la clase *Vista*, donde existe el botón, esta se asegura de pedir los permisos al usuario por pantalla según corresponda antes de delegar el resto de la tarea al *Modelo de Vista* que a su vez utiliza *Bluetooth Handler* de la sección *Model*.

Dentro de la clase *Bluetooth Handler*, se utiliza la clase nativa de Android, *BluetoothAdapter*, para manejar el módulo de Bluetooth y, rescatar los dispositivos ya emparejados, descubrir los sensores en caso de ser necesario y hacer el emparejamiento para poder iniciar la posible conexión más adelante.

- **Descubrimiento:** La rutina de descubrimiento es un proceso asincrónico iniciado directamente por el objeto *BluetoothAdapter* que se controla registrando una clase *BroadcastReceiver()* para poder recibir una variedad de señales llamadas *Intents* enviadas por la rutina y así dependiendo del tipo de señal se configura la clase para responder con distintas funciones personalizadas.

  El *BroadcastReceiver()* registrado actua frente a los *Intents* enviados al **comienzo-término** del proceso de descubrimiento, para actualizar el estado de la aplicación, y al recibir el *Intent* resultante del **descubrimiento de un dispositivo cercano**.

Entre los dispositivos descubiertos, los sensores son identificados basándose en los nombres incluidos en el objeto *BluetoothDevice* representativo del nuevo dispositivo. Estos nombres se comparan con los nombres de fabrica de los sensores previamente guardados en un archivo de configuración, cada uno de los *BluetoothDevice* correspondientes a los sensores objetivos se guarda para la siguiente etapa de emparejado.

- **Emparejado:** Este es un proceso asincronico iniciado por el telfono movil para cada sensor una vez que se tiene sus *BluetoothDevice*, con la finalidad de poder habilitar una comunicación segura. Este proceso también necesita la ayuda de un *BroadcastReceiver()* para configurar el comportamiento frente a los *Intent* de: **inicio de la solicitud de emparejamiento** y **cambio de estado de emparejamiento**.

  Ya que este proceso necesita de la participación de cada sensor se debe esperar el *Intent* correspondiente al inicio real del emparejamiento. Luego se configura un PIN de seguridad perteneciente a cada sensor según su documentación, para finalmente esperar la actualización del estado de emparejamiento de todos los sensores descubiertos.

3.2.2. Módulo sensor ECG

El sensor ECG es la fuente de datos correspondientes a la actividad eléctrica del corazón. Entrega voltajes de solo las **dos ultimas** derivaciones del corazón, dejando la posibilidad de calcular la primera en base a los demás datos:

- La **derivación I** mide la diferencia de potencial entre el electrodo del brazo derecho y el izquierdo.
- La **derivación II** mide la diferencia de potencial entre el electrodo del brazo derecho a la pierna izquierda.

- La **derivación III** mide la diferencia de potencial entre el electrodo del brazo izquierdo a la pierna izquierda.

Estos datos son importantes ya que es posible estimar las derivaciones de otros puntos llamados: **aVR**, **aVL** y **aVF**, o *Vector aumentado derecho*, *Vector aumentado izquierdo* y *Vector aumentado pie* respectivamente. Así se obtiene una imagen más completa del corazón con datos de múltiples ubicaciones.

Este módulo trabaja de forma asincronica al hilo principal de la aplicación por medio de corrutinas para extraer los datos de forma continua por medio de un bucle infinito. Este *bucle principal* es creado y controlado por la clase *ViewModel*, y su funcionamiento paso por paso es:

```
1 Bucle principal del sensor de ECG:
2     Intento de conexion del Cliente MQTT con el servidor.
3     Intento de conexion con el sensor de presion para obtener el
    BluetoothSocket.
4     Si hay conexion con el servidor y el sensor:
5         Comenzar bucle de lectura de datos del sensor mientras
            existan las conexiones.
```

Listing 3.4: Pseudocodigo de funcionamiento general, módulo sensor ECG.

El *bucle de lectura* es el encargado de utilizar el protocolo propio de comunicación correspondiente al sensor ECG. Esta comunicación es mediante el intercambio de los siguientes paquetes 3.1:

Start Flag	Packet Number	Command	Payload	Checksum	End Flag
0xFC	1 byte	2 byte	x byte	2 byte	0xFD

Fig. 3.1: Diagrama de Bytes para paquetes con sensor ECG.

- ***Start Flag y End Flag*:** Los valores 0xFC y 0xFD, siempre indican el comienzo y final de un paquete. Estos bytes no se deben usar en el resto de los campos del paquete para evitar confusión, debido a eso en caso de ser necesario se usa la tecnica llamada *Octet stuffing* para reemplazar los bytes problemáticos con una combinación especifica, para esto es necesario reservar el byte 0xFE (byte de escape) y usarlo como indicador que el siguiente byte fue manipulado y es el resultado de la operación: ***(Byte_Reservado) EXOR (0x20)***.

Byte para el paquete	Byte cambiado a:
0xFC	0xFE 0xDC
0xFD	0xFE 0xDD
0xFE	0xFE 0xDE

Fig. 3.2: Tabla de valores a cambiar con *Octet stuffing*.

- ***Packet Number:*** es un numero desde 0 a 255 que se aumenta consecutivamente a los largo de la comunicación de forma cíclica.

- ***Command*:** Comando de dos bytes que se lee comenzando por el byte menos significativo. Estos pueden utilizar el campo de *payload* para más información de la intrucción.

- ***Payload*:** Bytes con los datos, tiene un largo variable y su interpretación depende del comando.

- ***Checksum*:** Bytes de seguridad, se usan para asegurar la integridad del paquete. Se utiliza el algoritmo **CRC16 (CCITT)**[18] con el polinomio 0x1021, valor de inicio 0xFFFF y desde bit menos significativo.

  El calculo para **preparar** un paquete se hace sin los campos *Start Flag*, *End Flag* y *Checksum*, y antes de utilizar *Octet stuffing*.

  En el caso de **recibir** un paquete el *Checksum* se vuelve a calcular sin la *Start Flag* y *End Flag*, y luego de revertir el *Octet stuffing*. El resultado de la operación

debe ser **cero** para asegurar que el contenido del paquete se recibió correctamente.

Dentro del *bucle de lectura*, donde se recibe y maneja los datos provenientes del sensor, solomente la primera ejecución se vez se envía la secuencia estándar recomendada de paquetes para iniciar la transmisión:

```
1        //Secuencia de paquetes comunicados al sensor
2        Envio de comando para solicitud de ID.
3        Envio de comando para solicitud de protocolo registrado.
4        Envio de comando para configuracion modos del ECG.
5        Envio de comando para comienzo de transmision de datos.
6
7        Comenzar bucle de lectura:
```

Listing 3.5: Secuencia de paquetes para el inicio de transmsión, sensor ECG.

Etapa	Bytes *Command*	Bytes *Payload*	Explicación
Solicitud de ID	0x0800	0x0500	Se consulta la ID del dispositivo.
Solicitud de protocolo	0x0800	0x0100	Se consulta el protocolo utilizado.
Configuración de ECG	0x0901	0x0101	Se utilizara el sensor de 4 cables a una frecuencia de 100Hz
Comienzo de transmisión	0x0927	0x0001	Comenzar la transmisión en modo avanzado, con información de marca de tiempo.

Fig. 3.3: Comandos para paquetes de secuencia de inicio.

Una vez que terminada la etapa de inicio de transmisión, se comienza el manejo del contenido transmitido por el sensor mediante su *BluetoothSocket* siguiendo la rutina de 3.2.2 en un bucle infinito hasta que este sea detenido por el usuario. En caso de alguna desconexión siempre se intenta una reconexión de forma automatica para poder reiniciar la rutina.

```
1  buffer_entrada = bytes recibidos por el BluetoothSocket.
2  buffer_paquete = bytes de UN paquetes del sensor.
3
4  Inicio bucle de lectura, sensor ECG:
5     Si hay conexion se hace lectura completa del socket a buffer_entrada.
6         Se lee cada byte del buffer_entrada.
7             Si byte es igual a:
8                 StartFlag -> Se limpia buffer_paquete y se agrega la
9                              StartFlag.
10                EndFlag ->  Se agrega EndFlag al buffer_paquete.
11                            Se verifica el checksum del buffer_paquete.
12                            Si es el command correcto se procesan
13                            los datos del buffer_paquete.
14                Otro ->    Se agrega byte al buffer_paquete.
```

Listing 3.6: Pseudocodigo bucle de lectura, sensor ECG.

El *command* esperado para un paquete con datos de ECG es **0x0727** e indica un **Paquete avanzado de transmisión** con la estructura de la figura 3.4.

Start Flag	Packet Number	Command	Timestamp	Pulse	Monitor	Signal Detection	Payload	Checksum	End Flag
0xFC	1 byte	0x0727	4 byte	1 byte	1 byte	1 byte	x byte	2 byte	0xFD

Fig. 3.4: Diagrama de Bytes para paquetes avanzado de mediciones, sensor ECG.

En el segmento *payload* se encuentran múltiples mediciones de las *Derivadas* en forma comprimida. Se encuentran datos con signo de la *Derivada II* y *Derivada III* de

forma alternada con valores en **Complemento 2** de largo 1 o 2 bytes. Para diferenciar el largo del dato se reserva el primer *bit* del primer byte recibido, con 0 si el valor es un solo byte o con 1 si el valor usara 2 bytes.

Byte 1								Byte 2							
Sig-no	B_{13}	B_{12}	B_{11}	B_{10}	B_9	B_8	1	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0

Fig. 3.5: Valor de 2 bytes, sensor ECG.

Byte 1							
Signo	B_5	B_4	B_3	B_2	B_1	B_0	0

Fig. 3.6: Valor de 1 bytes, sensor ECG.

Todos las mediciones recibidas en un paquete se separan en arreglos de mismo largo, **Lead_II[] y Lead_III[]**, y además se rescata el campo *Signal Detection* para indicar si se detectó una onda R, el valor periódico máximo de la señal e importante para el diagnóstico de trastornos ventriculares. Con estos datos se prepara y envía el paquete al servidor.

```
{
    "timestamp_ms" : UNIX_timestamp,
    "lead_2"     : Lead_II[],
    "lead_3"     : Lead_III[],
    "r_wave"     : existe_onda_R
}
```

Listing 3.7: Paquete JSON de ejemplo para módulo ECG.

```
topico_modulo_ecg = "11111111-1/${ANDROID_ID}/ecg"
```

Listing 3.8: Tópico de ejemplo para paquete del sensor ECG.

Si fuera necesario gráficar, con cada par de derivadas obtenidas se puede calcular los datos de la primera faltante, según la ecuación 3.1. Ademas cada tupla de mediciones obtenidas en un paquete tiene sus datos espaciados temporalmente por *10ms*, según la frecuencia configurada en el sensor al inicio de la transmisión.

$$I = II - III \tag{3.1}$$

3.2.3. Módulo sensor de presión

El sensor de presión es capaz de obtener mediciones de presión diastólica y sistólica mediante una abrazadera que se ubica alrededor del *bicep*. Este proceso ocurre solo con el presionado de un botón directamente en el sensor y solo toma **un** par de mediciones. Estos datos son guardados en la memoria del dispositivo mientras este prendido esperando a ser solicitados por otro dispositivo.

De la misma forma que el modulo del sensor ECG, este módulo vive en una *corrutina asincronica* con un bucle principal que se encarga de mantener una conexión con el servidor y sensor de presión, y un segundo bucle para hacer la lectura y manejo de datos:

```
Bucle principal del sensor de presion:
    Intento de conexion del Cliente MQTT con el servidor.
    Intento de conexion con el sensor de presion para obtener el
   BluetoothSocket.
    Si hay conexion con el servidor y el sensor:
        Comenzar bucle de lectura de datos del sensor mientras
         existan las conexiones.
```

Listing 3.9: Pseudocodigo de funcionamiento general, módulo sensor de presión.

La lectura de datos en este módulo esta limitada por los datos disponibles por el sensor de presión, por lo que en el *bucle de lectura* solo se hace una solicitud de datos cada cierta cantidad de segundos y el resto del tiempo se mantiene dormida la corrutina. La secuencia de instrucciones para obtener los datos sigue el siguiente pseudocodigo:

```
Bucle de lectura:
    Envio de paquete para Solicitud de datos.
    Lectura de buffer de entrada.
    Inversion de Octet Stuffing y revision de checksum.
    Si la seccion command de la respuesta representa:
        ID -> Break, para hacer otra solicitud de datos.
        Datos -> Se extraen mediciones y envian al servidor.
                 Corrutina duerme por unos segundos.
        No mas datos -> Corrutina duerme por unos segundos.
```

Listing 3.10: Pseudocodigo bucle de lectura, módulo sensor de presión.

El protocolo para comunicarse con el sensor medidor de presión es uno especializado, pero muy similar al sensor ECG con la misma estructura de bytes (3.1), con *Octet stuffing* y un *Checksum* tipo **CRC16 (CCITT)**. Las diferencias se encuentran en las instrucciones disponibles y respuestas posibles.

Para hacer la solicitud de datos, el paquete debe tener los campos de la tabla 3.7.

Función	*Command*	*Payload*
Solicitar: Datos	0x0800	0x0607

Fig. 3.7: Tabla de valores para solicitar datos, sensor presión.

Luego las posibles respuestas del sensor se distinguen según su campo *command* y se tiene tres posibilidades relevantes según la tabla 3.8.

Función	*Command*	*Payload*
ID del dispositivo	0x0500	ID del dispositivo, productor y el numero de serie.
Datos	0x0706	Datos de medición de presión.
No existen más datos	0x07FA	Vacio.

Fig. 3.8: Tabla de valores para paquetes de respuesta, sensor presión.

Los paquetes recibidos con datos de presión siguen la estructura de la tabla 3.9, donde los datos de interés son ambos **valores de presión**.

Start Flag	**Packet Number**	**Com-mand**	**Times-tamp**	**Unidad**	**Presión Sistólica**	**Presión Diastólica**	**Pulso**	**End Flag**
0xFC	1 byte	0x0706	6 byte	1 byte	2 byte	1 byte	1 byte	0xFD

Fig. 3.9: Diagrama de Bytes para paquetes con datos de presión, sensor presión.

Ambos **valores de presión** se interpretan según la unidad de medida indicada en el campo *Unidad*, este por defecto siempre será **mmHg** (milímetros de mercurio). Además se tiene el campo *pulso* como un valor único del periodo que se tomó la medición, y el *timestamp* según la hora configurada en el sensor de forma manual en formato **Año: Mes: Día: Hora: Minuto: Segundo** con cada sección correspondiendo a uno de los 6 bytes del campo.

Para rescatar la información de la medición es necesario saber que la presión diastólica muestra su valor de forma explícita en un solo byte, pero la presión sistólica puede utilizar dos bytes que deben ser sumados cuando el más significativo sea mayor a cero.

Finalmente con toda los datos extraidos, el paquete a publicar con el cliente MQTT tiene la siguiente *payload* y tópico:

```
{
    "timestamp_ms" : UNIX_timestamp,
    "sys"          : presion_sistolica,
    "dia"          : presion_diastolica
}
```

Listing 3.11: Paquete JSON de ejemplo para paquete módulo presión.

```
topico_modulo_presion = "11111111-1/${ANDROID_ID}/pressure"
```

Listing 3.12: Tópico de ejemplo para paquete módulo presión.

3.2.4. Módulo sensor de oximetría

El sensor de oximetría es un pequeño dispositivo que se ubica en el dedo índice para tomar mediciones de pulso cardiaco y de saturación de oxígeno. El dispositivo utiliza su propio protocolo de comunicación sobre SPP, donde se tiene la posibilidad de configurar el modo de funcionamiento.

Se tiene 4 modos de funcionamiento con dos tipos de condiciones de encendido (con detección de dedo o mientras esté conectado). Cada modo cambia la estructura y frecuencia de los paquetes a enviar:

- **Modo #1** : Envía paquetes de 3 bytes a una tasa de 1 por segundo. Contiene la información del estado del sensor, pulso cardiaco y saturación de oxígeno.

- **Modo #2** : Envía un paquete que consiste de 25 segmentos de 5 bytes, 3 veces por segundo. Cada segmento contiene un byte de *status* y un byte con un valor de *Plethysmographic pulse* (corresponde a un pulso con amplitud). Además, cada segmento contiene distintos valores de saturación de oxígeno y pulso, calculados como un promedio de 4 u 8 latidos y con distintas velocidades de actualización para ser mostrados en pantallas que quieren cambios de valor en intervalos definidos.

- **Modo #7** : Mismo tamaño y tasa de paquetes que Modo #2, pero utiliza más bytes para mejores valores de *Plethysmographic pulse*.

- **Modo #8** : Similar a Modo #1, pero envía 4 bytes por paquete y un paquete por segundo. Utiliza el byte extra para comunicar información sobre el estado de la batería.

El **modo #1** es el indicado para proporcionar los datos requeridos por el sistema de monitoreo remoto, ya que el servidor no utiliza datos de *Plethysmographic pulse*, o las versiones de los datos con una tasa de refresco para una pantalla.

Los paquetes recibidos en **modo #1** tienen la estructura de la figura 3.10 según la documentación del dispositivo:

Byte 1 – STATUS							
BIT7	**BIT6**	**BIT5**	**BIT4**	**BIT3**	**BIT2**	**BIT1**	**BIT0**
1	SNSD	OOT	LPRF	MPRF	ARTF	HR8	HR7

*Note: Bit 7 is always set

Byte 2 - HEART RATE							
BIT7	**BIT6**	**BIT5**	**BIT4**	**BIT3**	**BIT2**	**BIT1**	**BIT0**
0	HR6	HR5	HR4	HR3	HR2	HR1	HR0

*Note: Bit 7 is always clear

Byte 3 - SPO2							
BIT7	**BIT6**	**BIT5**	**BIT4**	**BIT3**	**BIT2**	**BIT1**	**BIT0**
0	SP6	SP5	SP4	SP3	SP2	SP1	SP0

*Note: Bit 7 is always clear

Fig. 3.10: Estructura de bytes para paquete de sensor de oximetría, según documentación [19]

El byte 1 contiene el *status* y los bits más significativos del pulso cardiaco.

- SNSD: Desconexión del sensor cuando hay valor “1”.
- OOT: Existen pulsos consecutivos con mala señal cuando hay valor “1”.
- LPRF: Existe mala señal cuando hay valor “1”.
- MPRF: Existe una señal con calidad media cuando hay valor “1”.
- ARTF: Existe una condición de artefacto cuando hay valor “1”.
- HR8-HR0: Bits del pulso cardiaco calculado con el promedio de 4 latidos. En caso de ausencia de datos se envía el valor 511.
- SP6-SP0: Saturación de oxígeno calculada con el promedio a lo largo de 4 latidos. En caso de ausencia de datos se envía el valor 127.

El módulo de oximetría, al igual que los demás, funciona de forma asincronica con el bucle:

```
Bucle principal del sensor de oximetria:
    Intento de conexion del Cliente MQTT con el servidor.
    Intento de conexion con el sensor de oximetria para obtener el
   BluetoothSocket.
    Si hay conexion con el servidor y el sensor:
        Comenzar bucle de lectura de datos del sensor mientras
         existan las conexiones.
```

Listing 3.13: Pseudocodigo de funcionamiento general, módulo sensor de oximetría.

El paquete de configuración necesario para utilizar el **Modo #1** antes de comenzar el *bucle de lectura* consiste en tan solo escribir al *BluetoothSocket*, ya conectado, los bytes **(0x44, 0x41)** que corresponden a las letras "D" y "A" en ASCII. El carácter "D" indica un comando y la letra "A" es la instrucción, en este caso corresponde a la última opción que aparece en la tabla 3.11, para activar el **Modo #1** con la opción de encendido con detección de dedo.

Configuration Command	Serial Data Format	Turn-on Mode
1	1 (3 bytes 1 per/sec)	Sensor
2	2 (5 bytes, 75 per/sec, 8-bit pleth)	Sensor
7	7 (5 bytes, 75 per/sec, 16-bit pleth)	Sensor
8	8 (4 bytes, 1 per/sec)	Sensor
A	1 (3 bytes 1 per/sec)	Spot-check
B	2 (5 bytes, 75 per/sec, 8-bit pleth)	Spot-check
C	7 (5 bytes, 75 per/sec, 16-bit pleth)	Spot-check
D	8 (4 bytes, 1 per/sec)	Spot-check

Fig. 3.11: Modos de funcionamiento para sensor oximetría según documentación [19]

Una vez configurado el modo de funcionamiento se comienza el *bucle de lectura* para armar los paquete recibido.

```
Bucle de lectura:
    Configuracion del sensor.
    Lectura del buffer de entrada:
        Contador = 0.
        Lectura de byte del buffer de entrada en orden:
            Si BIT7 de byte == 1:
                Si Contador == 0:
                    Se agrega el byte a buffer de paquete.
                Otro:
                    Contador = 0.
                    Se agrega el byte a buffer de paquete.
            Otro:
                Si Contador es:
                    (1) ->  Se agrega el byte a buffer de paquete.
                            Contador = 2.
                    (2) ->  Se agrega el byte a buffer de paquete.
                            Contador = 0.
                            Se extraen y envian los datos
                            del buffer de paquete.
```

Listing 3.14: Pseudocodigo de bucle de lectura, módulo sensor de oximetría.

La extracción del *pulso* y *saturación de oxigeno* se hace de forma directa separando los bytes según 3.10. Luego el paquete JSON hacia el servidor tiene la siguiente estructura.

```
{
    "timestamp_ms" : UNIX_timestamp,
    "pulse"      : pulso_cardiaco,
    "o2_sat"     : saturacion_de_oxigeno
}

```

Listing 3.15: Paquete JSON de ejemplo para módulo de oximetría.

```
 topico_modulo_oximetria =
↪ "11111111-1/${ANDROID_ID}/oximeter"

```

Listing 3.16: Tópico de ejemplo para módulo de oximetría.

3.3. Módulo de locación

Para informar la posición geográfica del paciente en tiempo real, el dispositivo móvil usa su servicio de locación. En el sistema Android se aprovecha la API de servicios *Google Play* para solicitar periodicamente y bajo ciertas circunstancias la locación del dispositivo.

El objeto con la configuración para las solicitudes es **LocationRequest()**. Este tiene posibilidad para ajustar:

- **Prioridad:** define si se requiere de: maxima precisión, consumo medio de energia o ahorro de energia.
- **Intervalo:** el intervalo de tiempo, en *ms*, objetivo entre las solicitudes de locación. No hay garantía absoluta de que se cumplirá perfectamente.
- **Mínima Distancia:** distancia minima, en *metros*, entre dos solicitudes de locación. Ayuda a ahorrar energía.
- **Granularidad:** establece si las locaciones serán de alta o baja fidelidad. Esta variable también se puede establecer según el permiso que entrega el usuario.
- **Esperar locación precisa:** esperar un tiempo extra en caso de que el servicio pueda tener disponible una ubicación más precisa.

Para la aplicación móvil se tiene como configuración:

- Prioridad alta
- Intervalo de 5 segundos
- Mínima Distancia de 50 metros
- Granularidad según permiso del usuario. Alta fidelidad por defecto.

- Siempre esperar la locación precisa.

```
LocationRequest.Builder(Priority.PRIORITY_HIGH_ACCURACY,
    timeInterval_ms)
    .apply {
        setMinUpdateDistanceMeters(minDistance)
        setGranularity(Granularity.GRANULARITY_PERMISSION_LEVEL)
        setWaitForAccurateLocation(true)
    .build()


```

Fig. 3.12: Construcción de objeto LocationRequest().

Una vez existe el objeto *LocationRequest()*, que define las solicitudes de ubicación, se crea un objeto cliente 3.13 para monitorear esas solicitudes continuamente en un **bucle interno**, para responder frente a cada nueva actualización llamando funciones del objeto que son **sobrecargadas** dentro del Módulo de locación.

```
locationClient =
    LocationServices.getFusedLocationProviderClient(context)
locationClient.requestLocationUpdates(request, this@Location,
    Looper.getMainLooper())


```

Fig. 3.13: Construcción de objeto cliente LocationRequest().

Al recibir una nueva locación, el módulo ejecuta la función ***onLocationResult()*** donde se extrae las coordenadas geográficas, **latitud** y **longitud** en **formato decimal** resultantes de la solicitud de locación. Inmediatamente, se envía el nuevo paquete correspondiente al servidor a través del cliente MQTT conectado en la creación de la tarea

y mantenido en el *bucle principal* del módulo, similar a los sensores, pero sin un *bucle de lectura*.

```
{
    "timestamp_ms" : UNIX_timestamp,
    "latitude"  : -33.02062,
    "longitude" : -71.35432
}

```

Listing 3.17: Paquete JSON de ejemplo para módulo de Locación.

```
 topico_modulo_locacion =
↪ "11111111-1/${ANDROID_ID}/location"

```

Listing 3.18: Tópico de ejemplo para módulo de Locación.

3.4. Módulo de cámara

El módulo de cámara funciona con un solo bucle que se encarga de mantener el cliente MQTT conectado, y una función que recibe la dirección de archivo de una nueva foto para enviarla al servidor.

Para enviar la fotografía a través del cliente MQTT se divide la tarea en dos etapas, así se evita las dificultades debido a el gran posible tamaño de las imagenes y las usuales perdidas de conexión al estar en movimiento.

Primero se envía un primer paquete con un archivo JSON con solo el propósito de informar la cantidad de fragmentos y el tamaño de la fotografía proximamente a transmitir.

Segundo se rescata el *bitmap* de la imagen en formato **JPEG** para un crear un *ByteArray* que represente la foto. Este arreglo se divide en multiples fragmentos de hasta **500 Kbytes** siempre tomando en cuenta que puede existir un ultimo fragmento más pequeño que representa el resto de la división.

```
{
    "timestamp_ms"    : UNIX_timestamp,
    "photo_size_bytes"   : ByteArray_largo,
    "n_fragments" : cantidad_de_fragmentos
}
```

Listing 3.19: Paquete JSON de ejemplo para información de foto, módulo cámara.

El paquete de información se publica a un tópico específico de la imagen dentro de la sección del módulo de cámara. Se usa el *timestamp* como nombre para identificar la foto.

Luego los fragmentos se crean dividiendo el *ByteArray* en los segmentos correspondientes (largo máximo de **500 Kbytes**), y se envian de forma **secuencial** al servidor con un tópico que indique el número del fragmento.

```
topico_modulo_camara
    =
↪ "11111111-1/${ANDROID_ID}/camera/${UNIX_timestamp}.jpg"

topico_modulo_camara_fragmento_i
    =
↪ "11111111-1/${ANDROID_ID}/camera/${UNIX_timestamp}.jpg/${i}"
```

Listing 3.20: Tópicos de ejemplo para módulo de cámara.

Como seguridad, en caso de que un fragmento no se haya transmitido correctamente, se reintenta el mismo segmento nuevamente antes de seguir al siguiente.

3.5. Interfáz gráfica y modelo de vista

La interfaz gráfica de la aplicación se encarga de:

- Mostrar datos transmitidos por los sensores.
- Mostrar estado de conexión de los sensores.
- Ingresar el ID, nombre y sexo de un paciente.
- Iniciar el descubrimiento y emparejamiento de los sensores a través de un botón.
- Iniciar y detener la transmisión de datos al servidor a través de un botón.
- Tomar una fotografía con la cámara del dispositivo.
- Revisar la fotografía tomada y confirmar su transmisión.
- Cambiar entre pantallas de la aplicación.

Esto es posible mediante la asistencia del *ViewModel*, la clase que se comunica con las fuentes de datos y la UI. La interfaz gráfica, o *Vista*, contiene una instancia del *ViewModel* que le permite acceder a las funciones necesarias al presionar botones y al objeto *UIstate* que contiene las variables que se leen continuamente o modifican a través de la *Vista*.

La interfaz gráfica por sí sola es capaz de navegar por sus tres pantallas, **Pantalla inicial**, **Pantalla cámara** y **Pantalla de confirmación**. La navegación funciona manteniendo un historial de los movimientos para saber donde ir al usar el botón *atrás* del dispositivo móvil, por esto en cada cambio de pantalla provocado al ejecutar una nueva tarea se debe tener en cuenta a donde volverá el usuario cuando retroceda.
Las tres pantallas tienen los siguientes posibles movimientos para mantener el historial

acotado y no ocurran situaciones en que se deba presionar atrás múltiples veces, recorriendo las mismas pantallas numerosas veces.

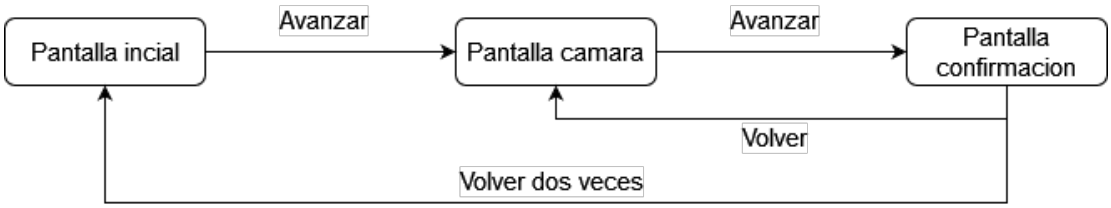


Fig. 3.14: Movimiento entre pantallas al ejecutar tareas.

Cada pantalla se define como una sola función *@Composable* que contiene todos los elementos gráficos necesarios que se muestran bajo un objeto **título** siempre posicionado en la sección superior de la pantalla.

- **Pantalla principal:** sigue el diagrama descrito en la figura 3.15. Esta pantalla está conformada por tres secciones apiladas dentro de un elemtento *columna*.

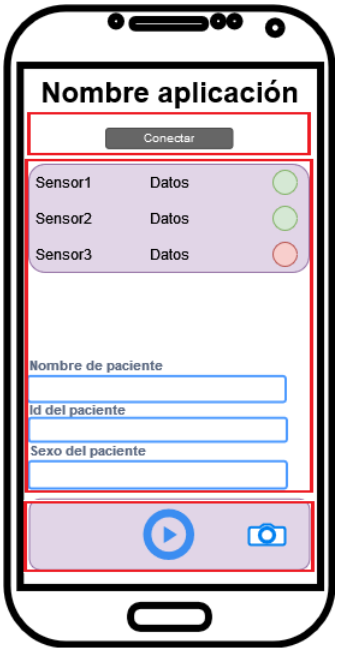


Fig. 3.15: Diagrama de pantalla principal dividido en tres secciones.

1. La primera sección es un solo elemento *botón* que desencadena la rutina de

descubrimiento mediante una función del *ViewModel*.

2. La segunda sección es una *columna* desplazable de forma vertical que apila: elementos *fila* con la información de sensores, los elementos **campos de texto** necesitados para ingresar la información de ID o nombre del paciente, y un elemento **menú desplegable** para elegir una opción de sexo.
   Los elementos *fila* de cada sensores contienen dos **textos** y un **icono dinámico** para representar el estado de conexión.
3. La tercera sección sitúa dos **botones** en una *fila*, uno para comenzar la transmisión y otro para utilizar la cámara.

Para controlar las posibles acciones que puede hacer el usuario en la **Pantalla principal**, el *ViewModel* define 5 estados posibles usando variables de *UIstate* según la figura 3.16.

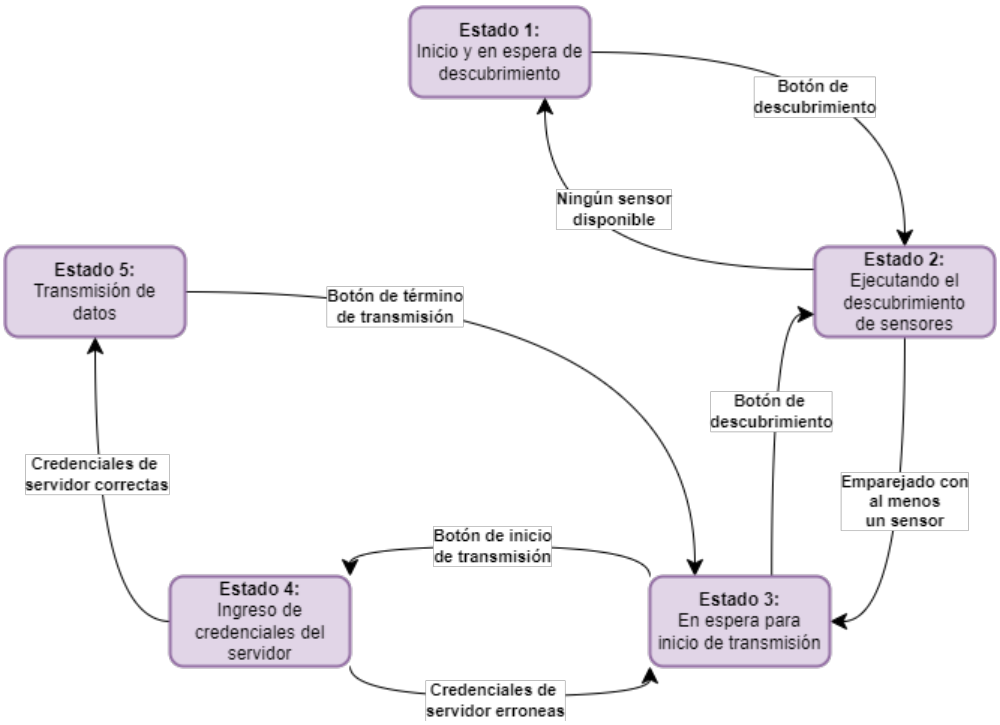


Fig. 3.16: Diagrama de estados para *pantalla principal*.

1. **Estado 1:** Solo se permite el funcionamiento del **botón para descubrir los sensores**, los demás botones están deshabilitados. En este estado los iconos de estado de los sensores se muestran desconectados y está habilitado el **ingreso de información del paciente**.
2. **Estado 2:** Ocurre al presionar el botón para ejecutar el descubrimiento, en este estado todos los **botones se deshabilitan** y es necesario esperar hasta que se termine el proceso de descubrimiento o se halla logrado emparejamiento con los sensores médicos del sistema.
3. **Estado 3:** En caso de haber al menos un sensor descubierto y emparejado, la aplicación pasa el estado 3. Aquí solo es posible **descubrir otra vez**, o si ya se ingresó la información del paciente necesaria, **comenzar la transmisión**.
4. **Estado 4:** Al presionar comenzar la transmisión se pide ingresar las **credenciales de acceso al servidor** (en un *diálogo de alerta*) antes de poder cambiar al estado 4. En este estado todos los **botones y campos de texto se deshabilitan** y se mantiene hasta confirmar una respuesta del servidor.
5. **Estado 5:** En caso de una conexión exitosa con el servidor se cambia al estado 5. Aquí se permite iniciar la **cámara** y **detener la transmisión** usando nuevamente el botón para comenzar la transmisión.

- **Pantalla cámara:** Al presionar el botón para iniciar la cámara en el **estado 5** se navega hacia la siguiente pantalla. Se utiliza la API de Android llamada **CameraX** para interactuar con el *hardware*.

  La pantalla está compuesta por un botón para tomar la foto y la misma imagen que está siendo capturada por la cámara. Para obtener la vista previa de la cámara se debe mostrar el objeto **PreviewView** que contenga un **ProcessCameraProvider()**. Este último debe configurarse con un objeto que reciba la imagen capturada, **ImageCapture()**, y una selección de cámara (delantera o trasera).

```
        val cameraProvider = context.getCameraProvider()
        cameraProvider.unbindAll()
        cameraProvider.bindToLifecycle(lifecycleOwner,
            cameraxSelector, preview, imageCapture)
        preview.setSurfaceProvider(previewView.surfaceProvider)

```

Listing 3.21: Código para contruir vista previa de cámara.

Al activar el botón de captura, *ViewModel* utiliza la clase **imageCapture** para almacenar la imagen en la sección de memoria privada de la aplicación (*cacheDir*), con el fin de no necesitar solicitar permisos adicionales.

- **Pantalla confirmación:** luego de capturar y guardar una fotografía se cambia de pantalla. En esta pantalla se sitúan la imagen capturada y dos botones, uno positivo y uno negativo, para confirmar si se envía la imagen capturada al servidor o si se debe intentar de nuevo volviendo a la pantalla anterior.

4. RESULTADOS DE PRUEBA

4.1. Entorno de prueba

El entorno de prueba se compone de tres dispositivos según el esquema 4.1:

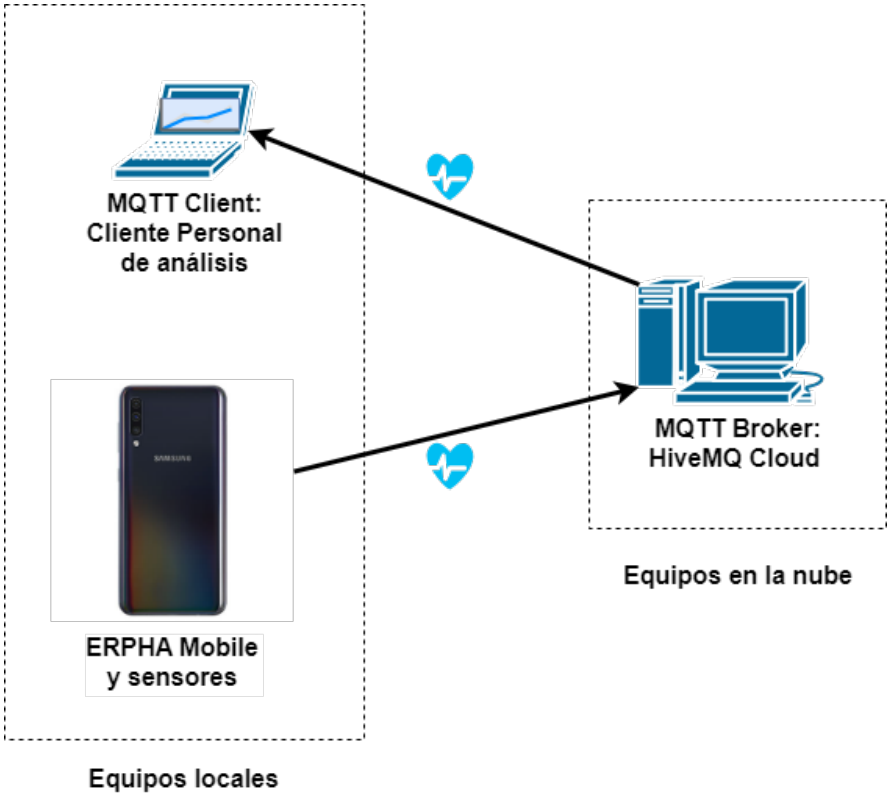


Fig. 4.1: Esquema para entorno de prueba.

- ERPHA Mobile: aplicación instalada en un telefono de prueba adecuado y los tres sensores medicos correspondientes.
- MQTT Broker: un servidor MQTT como destino de los datos producidos por la

aplicación. Este servidor corresponde a un servicio en la nube completamente funcional.

- Cliente MQTT de lectura: un cliente MQTT local que rescata todos los datos recibidos en el servidor y es capaz de visualizar las mediciones en tiempo real.

El móvil de prueba disponible corresponde a un teléfono **Samsung Galaxy A50** con un **Sistema Android 11**, es decir una **API 30**.

El **MQTT Broker** utiliza la versión gratis del servicio *cloud*, **HiveMQ Cloud**, provisto por **AWS** para tener una dirección fija.

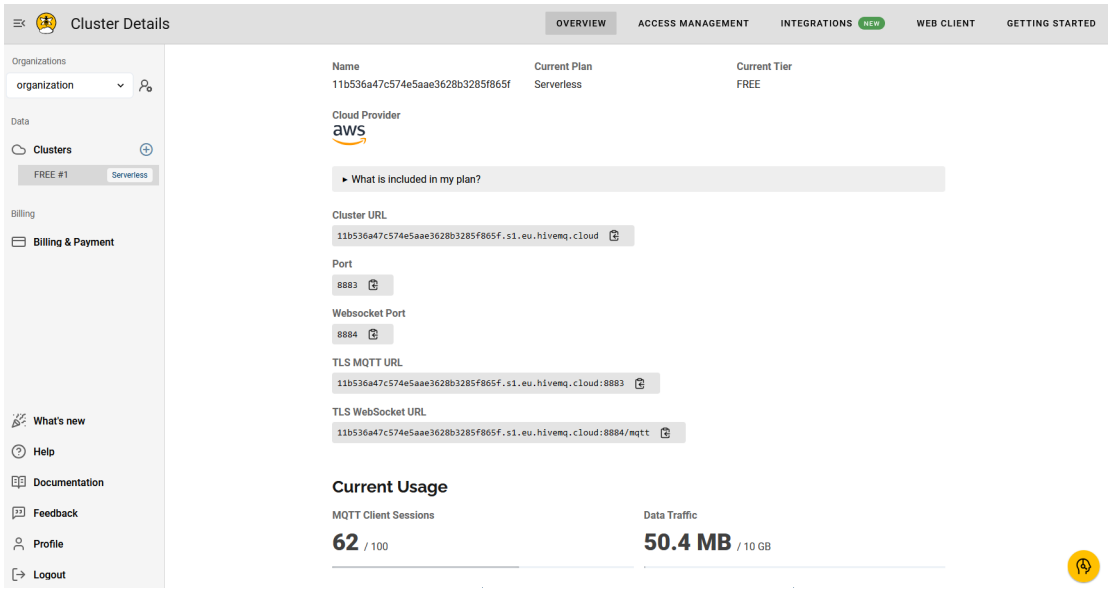


Fig. 4.2: Página web de control para servicio HiveMQ Cloud.

Para utilizar el **MQTT Broker** se crea un par de credenciales de seguridad (usuario-contraseña) en la sección ***Access management***, uno para la aplicación con solo permiso para publicar, y uno para el servidor con permisos para recibir.

El **Cliente MQTT de lectura** tiene como objetivo recibir los datos de la aplicación en tiempo real y hacer una vizualización de los datos a medida que se reciben. La comunicación con el servidor utiliza el cliente **Python** provisto por HiveMQ para solicitar

al **MQTT Broker** todos los paquetes de datos provenientes de la aplicación. El análisis de los datos utiliza el paquete **Dash-python** para graficar todas las señales de interés en un solo panel o *Dashboard* incluyendo una curva calculada del uso de ancho de banda (**KBytes/s**) en el tiempo.

Dentro de este cliente MQTT se suscribe a todos los tópicos posibles con el tópico "#", así se recibe todos los datos de todos los dispositivos conectados para crear un solo *Dashboard*, por lo que solo funciona correctamente con un teléfono móvil transmitiendo datos. El cliente funciona en un *loop* con una función de recepción modificada para tratar los paquetes según su tópico, se guarda todos los datos recibidos en distintos archivos de texto dependiendo del origen.

La aplicación está programada en **Android Studio**, donde es posible generar el archivo ejecutable APK y utilizar el mismo dispositivo móvil en **modo desarrollador** para ver la información de funcionamiento y los mensajes de **log** producidos por la aplicación a través de la IDE.

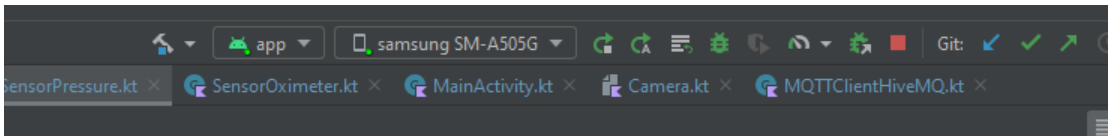


Fig. 4.3: Programa Android Studio vinculado con el dispositivo móvil.

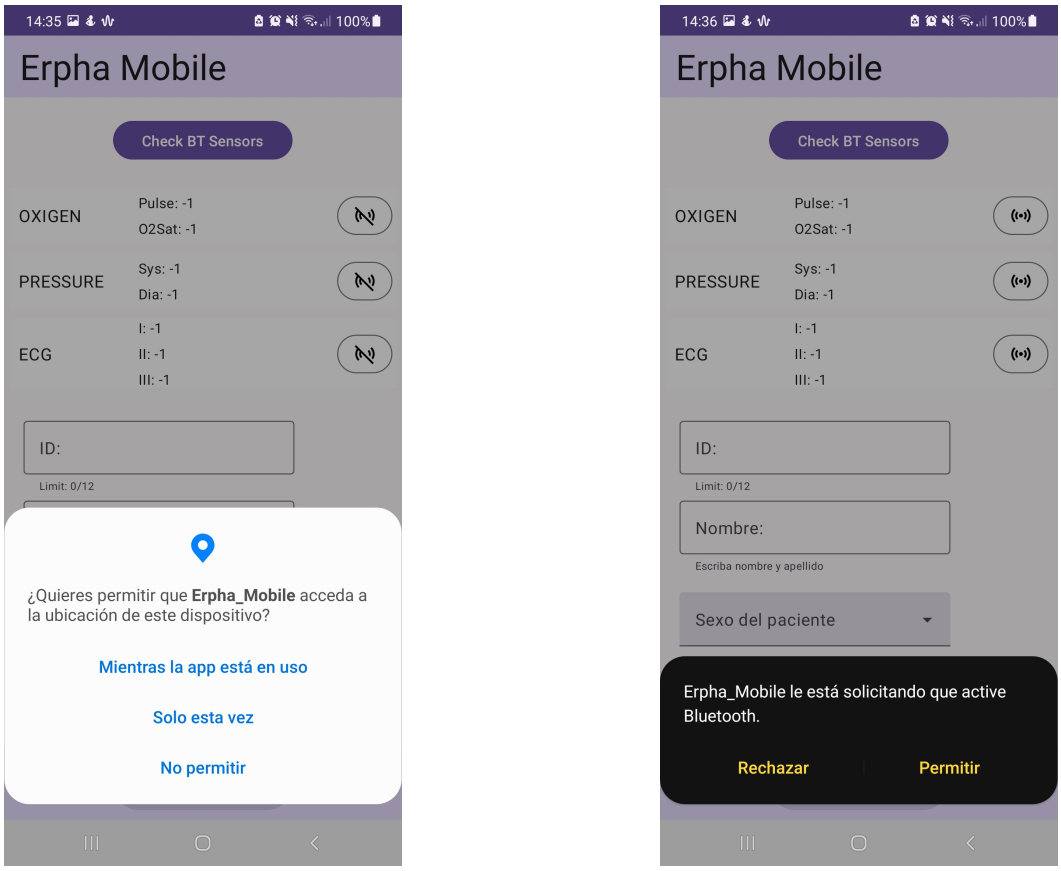
4.2. Resultados

Al construir y cargar la aplicación con el botón siguiente al nombre del dispositivo en la figura 4.3, la aplicación comienza la ejecución en su **Pantalla inicial**:

Fig. 4.4: Pantalla de inicio de aplicación.

En estado inicial, todos los textos de los datos de sensores se inician en "-1" con los iconos mostrando la ausencia de sensores. El único botón habilitado es para descubrir los dispositivos.

Al comenzar el descubrimiento de los sensores por primera vez la aplicación requiere que el usuario otorgue permisos para el *descubrimiento* y *locación*. Una vez otorgados es necesario habilitar las funcionalidades de **locación** y **Bluetooth** para poder proseguir.



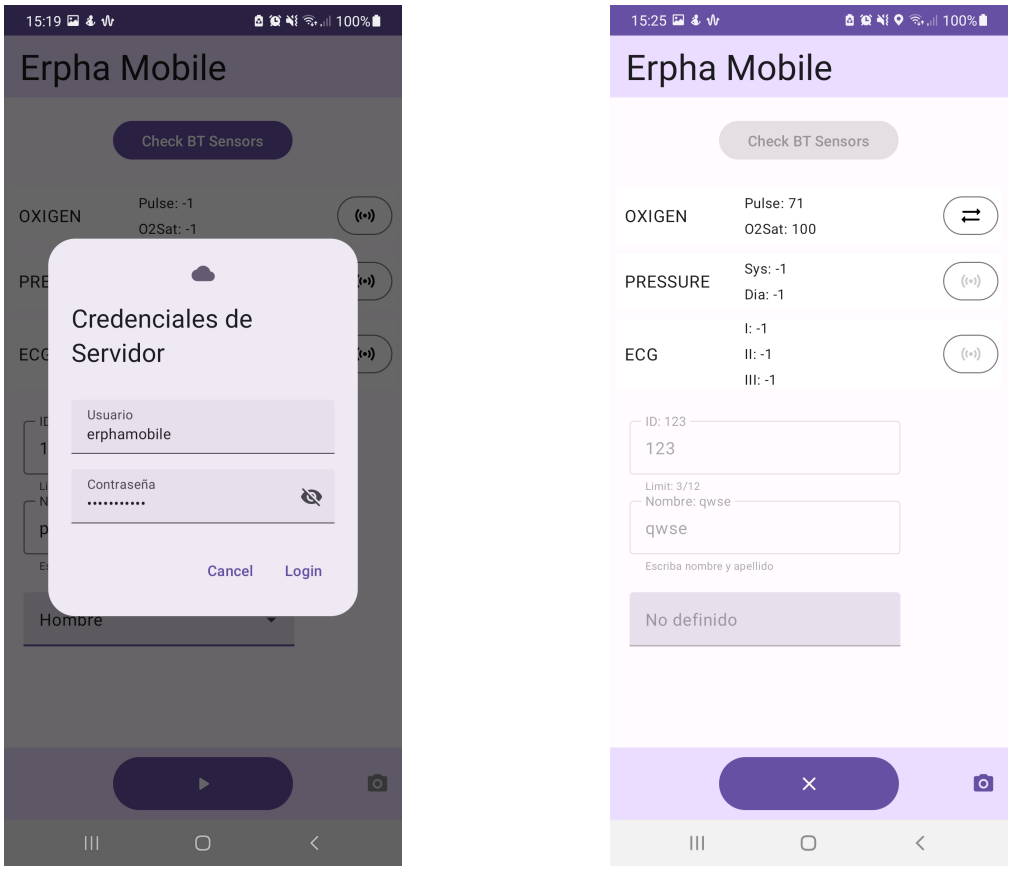
(a) Solicitud de permiso de Locación. *(b)* Solicitud de habilitación de servicios.

Fig. 4.5: Alertas de diálogo para requerimientos.

Iniciando la tarea de descubrimiento, la aplicación trabaja por aproximadamente un minuto máximo y muestra los resultados de los sensores vinculados por medio de iconos. La vinculación crea una ventana emergente para ingresar un código PIN, pero luego de un retardo, esta desaparece cuando la aplicación eventualmente establece las credenciales de forma programática. En caso de que todos los sensores estén vinculados y emparejados previamente, la aplicación termina la tarea de forma temprana.

Fig. 4.6: Resultado de comprobación de los sensores, con oxímetro no vinculado.

Para comenzar la transmisión es necesario ingresar un ID del paciente para distinguir los datos entre distintas transmisiones del teléfono móvil. Al presionar el botón de inicio de transmisión aparece una ventana de diálogo para ingresar las credenciales del servidor antes de intentar conectar los sensores disponibles.



(a) Ventana emergente para ingresar credenciales del servidor.

(b) Aplicación con solo sensor de oximetría transmitiendo.

Fig. 4.7: Ejemplo de inicio de transmisión de datos en aplicación, con solo oxímetro disponible.

En el **Cliente MQTT de lectura** se observa por la consola *python*, según la figura 4.8, todos los mensajes de bienvenida pertenecientes a los módulos de la aplicación al hacer sus conexiones individuales al *MQTT Broker*. Además, se recibe al mismo tiempo el primer mensaje JSON que incluye la información del paciente seguido del paquete periódico del modulo de locación según la figura 4.9.

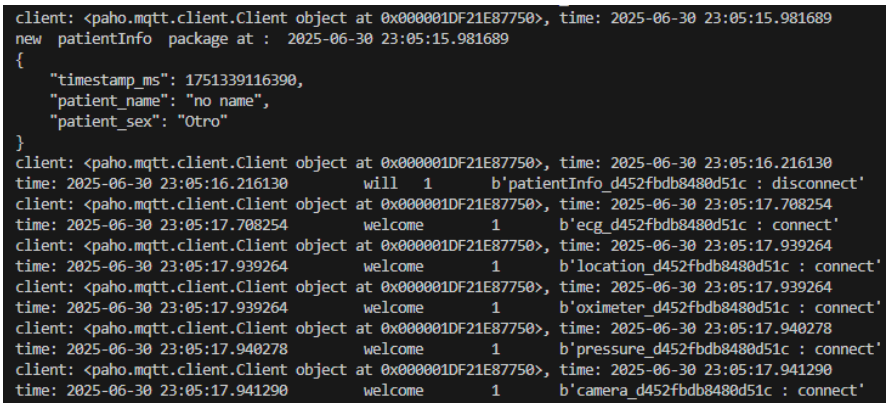


Fig. 4.8: Consola de **Cliente MQTT de lectura** al inicio de transmisión.

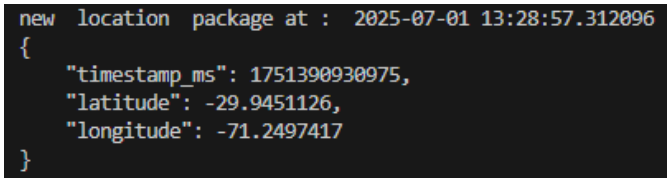


Fig. 4.9: Consola de **Cliente MQTT de lectura**, paquete de locación.

Las mediciones de presión se pueden confirmar por el unico paquete enviado por la aplicación al terminar una toma de presión. Resultados se ven en la figura 4.10.

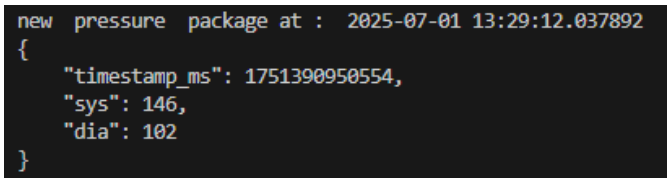


Fig. 4.10: Consola de **Cliente MQTT de lectura**, paquete de presión.

En el *Dashboard* creado por el **Cliente MQTT de lectura.** se tiene:

- Información general: fecha, estado de conexión, contador de paquete MQTT, KBytes totales recibidos en la sesion del cliente y nombre del paciente.
- Gráfica de *KBytes/s* leidos desde el servidor en el tiempo.
- Gráficas en el tiempo de O2 y pulso pertenecientes al sensor de oximetria.

- Gráficas en el tiempo de *Lead I*, *Lead II*, *Lead III* y pulso pertenecientes al sensor ECG.

- Gráfica en el tiempo de las mediciones individuales de presión Sistólica y Diastólica pertenecientes al sensor de presión.

Las curvas en el tiempo de los sensores se pueden observar en la figura 4.11. Aqui ambas curvas de pulso están algo desfasadas en el tiempo debido a un retraso al momento de conexión y una diferencia de escala debido a las cantidades de datos, pero ambas mantienen figuras similares. La curva de pulso perteneciente al ECG es altamente dependiente del movimiento de los electrodos y cualquier movimiento brusco del cuerpo se visualiza como una perturbación del gráfico.

Fig. 4.11: Curvas de datos de sensores en el tiempo real.

Por otro lado, curvas pertenecientes a las *Leads* del sensor ECG toman la forma esperada de un electrocardiograma con cierta periodicidad de valores máximos.

También se encuentra la gráfica para el monitoreo de ancho de banda. Al hacer una

transmisión, sin cámara, de 120 segundos se obtienen la figuras 4.12 y 4.13.
El calculo aproximado de *KBytes/s* se hace sumando todos los bytes recibidos en pequeños intervalos fijos de tiempo y luego escalandolos a un segundo. En este caso se nota valores maximos calculados de 12KBytes en un segundo.

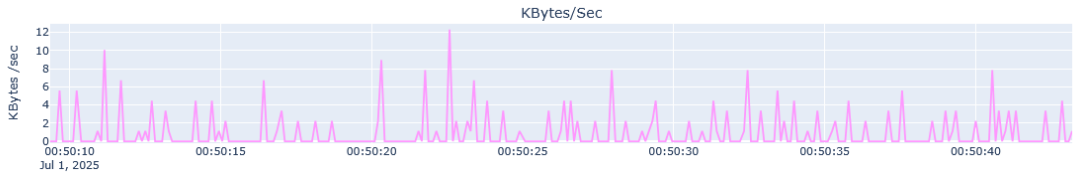


Fig. 4.12: Gráfica de KBytes/s en el tiempo, con un rango de 30 segundos, sin cámara.

Fig. 4.13: Información de número de paquetes y KBytes enviados en una sesión de 120 segundos.

Viendo la figura 4.13 se puede concluir las siguientes estadísticas.

- La aplicación tiene una tasa de aproximadamente 296 paquetes en 120 segundos o **6,63 paquetes por segundo** en funcionamiento normal.
- La cantidad de KBytes totales enviados por la aplicación en una sesión normal es de 362,096KB en 120 segundos, o **3,01 KBytes por segundo**.

Para la prueba del módulo de cámara, el paquete de información recibido por el cliente corresponde a la figura 4.14. La fotografía tomada tiene un tamaño de 7,49 MBytes y demoró aproximadamente un minuto en enviar y recibir todos los 15 fragmentos, todo este proceso mientras los demás sensores emitían datos.

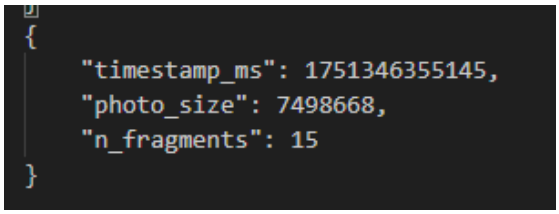


Fig. 4.14: Paquete JSON de información recibido para módulo camara.

5. CONCLUSIONES

La aplicación desarrollada resulta ser una herramienta útil para nuevos sistemas con base en tecnologías modernas, pero con la ventaja de ser capaz de comunicarse y aprovechar dispositivos utilizados en sistemas antiguos con sensores de perfil SPP, a diferencia de opciones con nuevos perfiles de Bluetooth para comunicación con sensores, como HDP[20], que ya se encuentran implementados en Android con una API de alto nivel. Por otro lado, la utilización de *Bluetooth Classic* es perfectamente compatible para dispositivos que manejan un gran flujo de datos y sin grandes restricciones de batería, a pesar de la existencia de la nueva tecnología *Bluetooth Low Energy (BLE)* que está orientado a sistemas con sensores que prefieren funcionar sin intervención por largos periodos de tiempo.

Como dificultad, los sensores y sus protocolos personalizados están pensados para lenguajes de programación de más bajo nivel, donde es necesario tratar la trama mediante operaciones *bit-a-bit* en bytes individuales, cosa que no está soportada completamente en **Kotlin** para variables que no sean explícitamente de clase *int*.

El resultado final logra cumplir con los requisitos de funcionamiento, pero con espacio para mejoras de rendimiento debido a la gran curva de aprendizaje de Android y la constante adición de nuevas funcionalidades o estándares con mejoras de rendimiento. Al término del trabajo se ve la posibilidad de mejorar la experiencia de uso para el usuario a través de **Tareas en segundo plano** y **WorkManager**, haciendo posible el funcionamiento de la aplicación mientras se utiliza el dispositivo móvil para otras tareas. Actualmente, la aplicación es capaz de funcionar estando minimizada, pero na-

da garantiza que se seguirá ejecutando, cualquier nueva tarea que el sistema Android considere más importante tomará control de los recursos de la aplicación sin devolverlos, necesitando que estos se soliciten nuevamente abriendo la aplicación para resumir el funcionamiento desde el comienzo. Por otro lado, también existe la posibilidad de disminuir el ancho de banda transmitido hacia el servidor por medio de otros formatos orientados a comunicación entre sistemas como *Binary JSON*[21] o *Protobuf*[22], o también métodos más simples como el uso de CSV, pero cualquiera de estos cambios significa una implementación y adaptación de estos nuevos formatos en el lado de la aplicación móvil y el servidor.

Bibliografía

[1] CARRASCO Guerra, Cristian Rodrigo. Integración de servicio de medición de signos vitales en dispositivo móvil con conectividad 3g y su despliegue en una aplicación web. Memoria(Ing. Civil Electrónica). Valparaíso UTFSM. Departamento de electrónica. 2013 79h

[2] StatCounter "Mobile Operating System Market Share Chile" '
(en linea) https://gs.statcounter.com/os-market-share/mobile/chile [Consulta Agosto. 19, 2024].

[3] StatCounter "Android API Levels"
(en linea) https://apilevels.com/ [Consulta Agosto. 19, 2024].

[4] Android Developers "Android Versions"
(en linea) https://developer.android.com/about/versions [Consulta Agosto. 19, 2024].

[5] Transmission Control Protocol (TCP) - RFC9293
(en linea) https://datatracker.ietf.org/doc/html/rfc9293 [Consulta Febrero. 20, 2025].

[6] An Introduction to the Stream Control Transmission Protocol (SCTP) - RFC3286
(en linea) https://datatracker.ietf.org/doc/html/rfc3286 [Consulta Febrero. 20, 2025].

[7] QUIC: A UDP-Based Multiplexed and Secure Transport - RFC9000
(en linea) https://datatracker.ietf.org/doc/html/rfc9000 [Consulta Febrero. 20, 2025].

[8] Hypertext Transfer Protocol – HTTP/1.1 - RFC2616
(en linea) https://datatracker.ietf.org/doc/html/rfc2616 [Consulta Febrero. 20, 2025].

[9] MQTT: The Standard for IoT Messaging
(en linea) https://mqtt.org/ [Consulta Febrero. 20, 2025].

[10] AMQP Advanced Message Queuing Protocol
(en linea) https://www.amqp.org/ [Consulta Febrero. 20, 2025].

[11] Eclipse Foundation "Eclipse Paho Android Service"'
(en linea) https://eclipse.dev/paho/ [Consulta Agosto. 19, 2024].

[12] HiveMQ "HiveMQ MQTT Client"'
(en linea) https://hivemq.github.io/hivemq-mqtt-client// [Consulta Agosto. 19, 2024].

[13] Android Developers "Jetpack Compose" '
(en linea) https://developer.android.com/compose [Consulta Agosto. 19, 2024].

[14] VALÍN Muñoz, Guillermo Andrés. Monitoreo remoto de señales de electrocardiograma en pacientes mediante dispositivos móviles 3g provistos de conexión bluetooth. . Memoria(Ing. Civil Electrónica). Valparaíso UTFSM. Departamento de electrónica. 2012 47h

[15] HERNÁNDEZ Caro, Cristian Antonio. Desarrollo de un sistema móvil de registro, transmisión y despliegue visual en una página web de señales de presión arterial. Memoria(Ing. Civil Electrónica). Valparaíso UTFSM. Departamento de electrónica. 2012 90h

[16] MERELLO Pinilla, Alejandro Lorenzo. Desarrollo de un sistema móvil de registro de signos vitales y almacenamiento en base de datos en internet. Memoria(Ing. Civil Electrónica). Valparaíso UTFSM. Departamento de electrónica. 2011 59h

[17] Android Developers: Settings.Secure.ANDROID_ID
(en línea) https://developer.android.com/reference/android/provider/Settings.Secure#ANDROID_ID [Consulta Agosto. 29, 2024].

[18] CRC16-CCITT
(en línea) https://srecord.sourceforge.net/crc16-ccitt.html [Consulta Agosto. 29, 2024].

[19] Nonin-4100, 2006
(en línea)https://web.archive.org/web/20061113114717/http://www.nonin.com/documents/4100%20Specifications.pdf [Consulta Junio. 29, 2025].

[20] Bluetooth Health Device Profile
(en línea) https://www.bluetooth.com/wp-content/uploads/2019/03/HDP-Implementation_WP_V10.pdf [Consulta Junio. 29, 2025].

[21] Binary JSON
(en línea) https://bsonspec.org/ [Consulta Junio. 29, 2025].

[22] Protocol Buffers
(en línea) https://protobuf.dev/ [Consulta Junio. 29, 2025].