Módulo de comunicación SmolComm

Jeison Ortiz Samboni

Informe Practica académica para optar al título de Ingeniero Electrónico

Asesor

Augusto Enrique Salazar Jiménez, Doctor (PhD)

Universidad de Antioquia

Facultad de ingeniería

Pregrado en Ingeniería Electrónica

Medellín

2022

Tabla de contenido

Resumen

La empresa Intelligent Electronic Solutions (IES) se encarga de facilitar el software de gestión de la información de las máquinas a casinos, en una actualización del sistema se requería desarrollar un módulo de comunicación que funcione como una aplicación stand-alone, para permitir la comunicación entre las máquinas y el sistema de gestión. La implementación se hizo en el lenguaje de programación java, haciendo uso del framework spring boot, se implementaron dos conexiones que usan el modelo publicador/suscriptor, del lado de la máquina se usó la tecnología Mqtt con el broker mosquitto, y del lado del servidor, bajo este mismo modelo se usó una base de datos redis, logrando hacer que la información que las máquinas publican en mosquitto llegue a la base de datos redis, y de la misma manera, la información que publica el sistema general en redis, llegue a la máquina correspondiente, logrando así una comunicación bidireccional.

1. Introducción

Las herramientas administrativas son un conjunto de instrumentos modernos y efectivos que permiten a los directivos de pequeñas y medianas organizaciones, tomar decisiones cruciales y oportunas tanto ante alguna dificultad como en el día a día y, además, gestionar y planificar proyectos [1].

Los proyectos exitosos son los que tienen una estrategia de administración de recursos efectiva, pero lograr esto requiere tiempo y quizás represente riesgos cuando no se tiene un software de gestión [2].

La empresa Intelligent Electronic Solutions (IES) es una compañía colombiana de tecnología, especializada en el desarrollo de aplicaciones web y móviles, con énfasis en soluciones de alto impacto para la gestión, generación de la información y el conocimiento [3].

Como empresa tiene la necesidad de brindar a sus clientes un producto que cada vez tenga mejores características, para ello requiere estar en una constante actualización de su sistema, de donde se desprende la necesidad de desarrollar el módulo de comunicación SmolComm, que implementa nuevas tecnologías para mejorar la experiencia del usuario y la modularidad dentro del sistema. Para ello, se planteó implementar este módulo como una aplicación stand-alone que se ejecute de manera independiente.

Su desarrollo implicó conocer el sistema ya existente, para tener una idea del negocio y hacer el planteamiento de los requerimientos, con esto listo y teniendo en cuenta las buenas prácticas seguidas por la empresa, se plantea una solución que cumple con estos. Esta solución se divide en cuatro etapas, la primera y la segunda, determinan las conexiones con el broker **mosquitto** y la base de datos **redis** respectivamente, en la tercera se hace la manipulación de los mensajes para redirigirlos al lugar de destino correcto, y en la última etapa, se implementa el control de comunicación con las máquinas. Teniendo la implementación anterior se valida el correcto funcionamiento de la aplicación cumpliendo satisfactoriamente el objetivo planteado.

2. Objetivos

2.1 Objetivo general

Diseñar e implementar un módulo de comunicación, como una aplicación stand-alone mediante el framework Spring boot de java, de tal manera que permita mantener el flujo de información bidireccional entre la tarjeta SMIB y una base de datos, permitiendo al sistema general (SMOL) gestionar e interpretar la información proveniente de las máquinas.

2.2 Objetivos específicos

- Definir los requerimientos funcionales y de prueba para la aplicación, donde se determinen los comandos que será capaz de recibir, interpretar y enviar tanto desde el lado de la tarjeta como del SMOL.
- Diseñar los diagramas para los topics del broker MQTT y las clases que permitan integrar las funcionalidades definidas, de tal manera que se tenga un bajo acoplamiento y una alta cohesión entre cada clase.
- Implementar el diseño planteado en cada uno de los diagramas mediante código fuente, teniendo en cuenta las buenas prácticas de programación.
- Validar mediante pruebas el correcto funcionamiento parcial de cada comunicación y de la implementación general, simulando una máquina y el sistema general.

3. Marco Teórico

Aplicación Stand-alone

Una aplicación *stand-alone* es aquella que se puede ejecutar sin necesidad de ningún elemento soporte (por ejemplo, un navegador) [4]. Es decir, es un programa que puede trabajar de manera autónoma, que se puede instalar y ejecutar, o simplemente ejecutar, en un sistema sin necesidad de nada más [5].

Modelo de comunicación publicador subscriptor

La mensajería de publicación/suscripción, es una forma de comunicación asíncrona de servicio a servicio utilizada en arquitecturas sin servidor y de microservicios [6]. Con el modelo de comunicación de publicación/suscripción, las aplicaciones no están relacionadas con socios específicos. Los sistemas de publicación/suscripción manejan datos y no presentan requisitos específicos para los destinatarios o los orígenes de los mensajes. El modelo de

publicación/suscripción separa el proveedor de la información de los consumidores de dicha información.

El proveedor de información se denomina publicador. Los publicadores proporcionan información sobre un asunto. El consumidor de la información se denomina suscriptor. Existe un intermediario entre ambos.
La información se envía en un mensaje y el asunto de la información se identifica mediante un tópico. El publicador específico el tópico donde se publica la información. El suscriptor específica los tópicos sobre la información deseada. Al suscriptor sólo se le envía la información a la que se ha suscrito [7].

En este modelo, todos los suscriptores de un tópico reciben inmediatamente cualquier mensaje publicado en este tópico. Está mensajería se puede utilizar para habilitar arquitecturas basadas en eventos o para desacoplar aplicaciones a fin de aumentar el rendimiento, la confiabilidad y la escalabilidad [6].

Broker (intermediario)

Un bróker es el servidor con el que se comunican los clientes: recibe comunicaciones de unos y se las envía a otros. Los clientes no se comunican directamente entre sí, sino que se conectan con el bróker. Cada cliente puede ser un publicador, un suscriptor o ambos [8].

Mqtt

MQTT es un protocolo de transporte de mensajería de publicación/suscripción del servidor del cliente. Es liviano, abierto, simple y diseñado para que sea fácil de implementar. Estas características lo hacen ideal para su uso en muchas situaciones, incluidos entornos limitados, como la comunicación en contextos de máquina a máquina (M2M) e Internet de las cosas (IoT), donde se requiere una huella de código pequeña y/o el ancho de banda de la red es un bien escaso.

El protocolo se ejecuta sobre TCP/IP, o sobre otros protocolos de red que proporcionan conexiones bidireccionales ordenadas y sin pérdidas. Sus características incluyen:

- Uso del patrón de mensajes de publicación/suscripción que proporciona distribución de mensajes de uno a muchos y desacoplamiento de aplicaciones.

- Un transporte de mensajería que es independiente del contenido de la carga útil.

- Tres calidades de servicio para la entrega de mensaje:

"**A lo sumo una vez"**, donde los mensajes se entregan de acuerdo con los mejores esfuerzos del entorno operativo. Puede ocurrir la pérdida de mensajes. Este nivel podría usarse, por ejemplo, con datos de sensores ambientales en los que no importa si se pierde una lectura individual, ya que la siguiente se publicará poco después.

"**Al menos una vez"**, donde se asegura que los mensajes llegarán, pero pueden ocurrir duplicados.

"Exactamente una vez", donde se asegura que el mensaje llegue exactamente una vez. Este nivel podría usarse, por ejemplo, con sistemas de facturación en los que los mensajes duplicados o perdidos podrían dar lugar a la aplicación de cargos incorrectos.

- Una sobrecarga de transporte pequeña y los intercambios de protocolo minimizados para reducir el tráfico de red [9].

Topics en mqtt

Como hemos dicho, los Broker MQTT aplican un filtrado a los mensajes que son recibidos desde los publicadores, para discriminar a qué clientes suscritos es entregado.

En MQTT este filtro se denomina Topic, y simplemente consiste en una cadena de texto UTF-8, y una longitud máxima de 65536 caracteres (aunque lo normal es que sea mucho menor). Se distingue entre mayúsculas y minúsculas [10].

Eclipse Mosquitto

Eclipse Mosquitto es un broker de mensajes de código abierto (con licencia EPL/EDL) que implementa las versiones 5.0, 3.1.1 y 3.1 del protocolo MQTT. Mosquitto es liviano y es adecuado para su uso en todos los dispositivos, desde computadoras de placa única de bajo consumo hasta servidores completos [11].

Redis

Redis es un almacén de estructura de datos de valores de clave en memoria rápido y de código abierto. Redis incorpora un conjunto de estructuras de datos en memoria versátiles que le permiten crear con facilidad diversas aplicaciones personalizadas. Entre los casos de uso principales de Redis se encuentran el almacenamiento en caché, la administración de sesiones, pub/sub y las clasificaciones. Es el almacén de valores de clave más popular en la actualidad. Tiene licencia BSD, está escrito en código C optimizado y admite numerosos lenguajes de desarrollo. Redis es el acrónimo de REmote DIctionary Server (servidor de diccionario remoto).

Gracias a su velocidad y facilidad de uso, Redis es una opción popular para aplicaciones web, móviles, de juegos, de tecnología publicitaria y de IoT que requieren el mejor desempeño de su clase [12].

Redisson

Redisson es un cliente Redis seguro para subprocesos para el lenguaje de programación Java. Le permite utilizar todas las colecciones y estructuras de datos de Java conocidas además de Redis, como Lista, Mapa, Cola, Bloqueo,

Semáforo y muchas más [13], en comparación con otros clientes como Jedis, cuenta con muchas más funcionalidades y es mucho más completo.

Spring Boot

Java Spring Boot (Spring Boot) es una herramienta que hace que el desarrollo de aplicaciones web y microservicios con Spring Framework sea más rápido y fácil a través de tres capacidades principales:

1. Autoconfiguración
2. Un enfoque obstinado de la configuración
3. La capacidad de crear aplicaciones independientes

Java Spring Framework (Spring Framework) es un marco popular, de código abierto y de nivel empresarial para crear aplicaciones independientes de nivel de producción que se ejecutan en la máquina virtual de Java (JVM) [14].

Cliente Eclipse Paho Java

Paho Java Client es una biblioteca de cliente MQTT escrita en Java para desarrollar aplicaciones que se ejecutan en JVM u otras plataformas compatibles con Java, como Android.

El cliente Java de Paho proporciona dos API: MqttAsyncClient proporciona una API completamente asíncrona en la que se notifica la finalización de las actividades a través de devoluciones de llamadas registradas. MqttClient es un contenedor sincrónico alrededor de MqttAsyncClient donde las funciones aparecen sincrónicas con la aplicación [15].

Docker

Docker es una plataforma de software que permite crear, probar e implementar aplicaciones rápidamente. Docker empaqueta software en unidades estandarizadas llamadas contenedores que incluyen todo lo necesario para que el software se ejecute, incluidas bibliotecas, herramientas de sistema, código y tiempo de ejecución. Con Docker, puede implementar y ajustar la escala de aplicaciones rápidamente en cualquier entorno con la certeza de saber que su código se ejecutará.

Docker es un sistema operativo para contenedores. De manera similar a cómo una máquina virtual virtualiza (elimina la necesidad de administrar directamente) el hardware del servidor, los contenedores virtualizan el sistema operativo de un servidor. Docker se instala en cada servidor y proporciona comandos sencillos que puede utilizar para crear, iniciar o detener contenedores [16].

Docker compose

Docker Compose es una herramienta de la plataforma dedicada a la orquestación local de dockers, es decir, se utiliza con el objetivo de definir y ejecutar aplicaciones Docker de varios contenedores de forma fácil y rápida.
Esta definición y orquestación se lleva a cabo de forma local al interior de los containers, quienes, además, se encontrarán unidos a través de una red de Docker.
Para su funcionamiento, Docker Compose emplea un archivo tipo YAML que le permite realizar la configuración de los diferentes servicios pertenecientes a la aplicación. Después de esto, un comando cumplirá la función de crear e iniciar estos servicios a partir de sus ajustes [17].

Volumes en Docker compose

Son simplemente archivos y directorios en el host que esté corriendo docker. Estos son utilizados por los contenedores para persistir y/o leer datos [18].

Testing de software

El testing de software o software QA, es un proceso para verificar y validar la funcionalidad de un programa o una aplicación de software con el objetivo de garantizar que el producto de software esté libre de defectos. La intención final es que coincida con los requisitos esperados para entregar un producto de calidad. Implica la ejecución de componentes de software o sistema utilizando herramientas manuales o automatizadas para evaluar una o más propiedades de interés.

El testing de software es un proceso paralelo al desarrollo de software cuyas tareas deben ir realizándose a medida que se construye el producto para evitar problemas en la funcionalidad de manera previa a su lanzamiento [19].

Mockito

Mockito es un marco de simulación, una biblioteca basada en JAVA que se utiliza para pruebas unitarias efectivas de aplicaciones JAVA. Mockito se usa para simular interfaces para que se pueda agregar una funcionalidad ficticia a una interfaz simulada que se puede usar en pruebas unitarias [20].

4. Metodología

Para el desarrollo de este apartado fue necesario el aporte y acompañamiento de otras áreas de la empresa involucradas en lo que tiene que ver el proyecto.

- En principio, el área con la que se tuvo mayor interacción fue con el equipo SMOL, encargado de dar soporte del sistema que existe

actualmente, con ellos se contextualiza el funcionamiento y la implementación existente.

- Posteriormente con el equipo de desarrollo se tuvo el apoyo para diseñar la aplicación, y tener retroalimentación en cuanto a la implementación de las buenas prácticas adoptadas por la empresa.
- Finalmente, con el área del hardware, encargada del mantenimiento y actualización de las tarjetas de las máquinas, de quien se recibió apoyo para conocer funcionalidades necesarias en la implementación de algunos requerimientos.

Con ayuda de estos equipos se logró tener un contexto suficiente para proceder a realizar los objetivos planteados.

Primero, conociendo el funcionamiento del sistema existente y las nuevas necesidades de implementar un módulo totalmente nuevo, con ayuda de los equipos smol y de hardware se procede a hacer un levantamiento de requerimientos para la comunicación con la tarjeta, que posteriormente se evaluaron y aprobaron por el equipo de desarrollo, dichos requerimientos se describen a continuación.

- La aplicación debe desarrollarse en el framework Spring Boot, con la versión 11 de java.
- Esta debe implementar el modelo publicador suscriptor, por tanto, debe ser capaz de suscribirse al servidor **mosquitto** (*broker mqtt*) y escuchar cualquier mensaje que llegue de las tarjetas, procesarlo y hacer que llegue al destino correspondiente. También debe cumplir la función de un cliente y poder publicar mensajes en mosquitto en cualquiera de los tópicos, de tal manera que la información llegue a la tarjeta indicada.

- La aplicación debe llevar el control de la comunicación con las tarjetas, por lo que debe informar al sistema general si alguna de ellas se desconecta.

Por otro lado, la aplicación debe comunicarse con el sistema general (SMOL), para ello se usa una base de datos intermediaria **Redis**, que funciona bajo el mismo modelo de publicador subscriptor, estableciendo de este lado los siguientes requerimientos.

- La aplicación debe usar un cliente **redisson** para publicar y suscribirse a la base de datos **redis**.
- La aplicación debe publicar cualquier mensaje que provenga de las tarjetas en el tópico correspondiente en la base de datos.
- También debe suscribirse y escuchar cualquier mensaje que provenga del sistema general y redirigirlo a la máquina destino.

Teniendo estos requerimientos establecidos se procedió a plantear la solución para cumplir con ello.

Con ayuda del equipo de desarrollo se establece el flujo de interacción de la aplicación con los demás módulos, en la figura 1 se observa este flujo donde se evidencia dos caminos, uno para la información que va desde la tarjeta de la máquina hasta el Smol y el otro desde el Smol hasta la tarjeta.

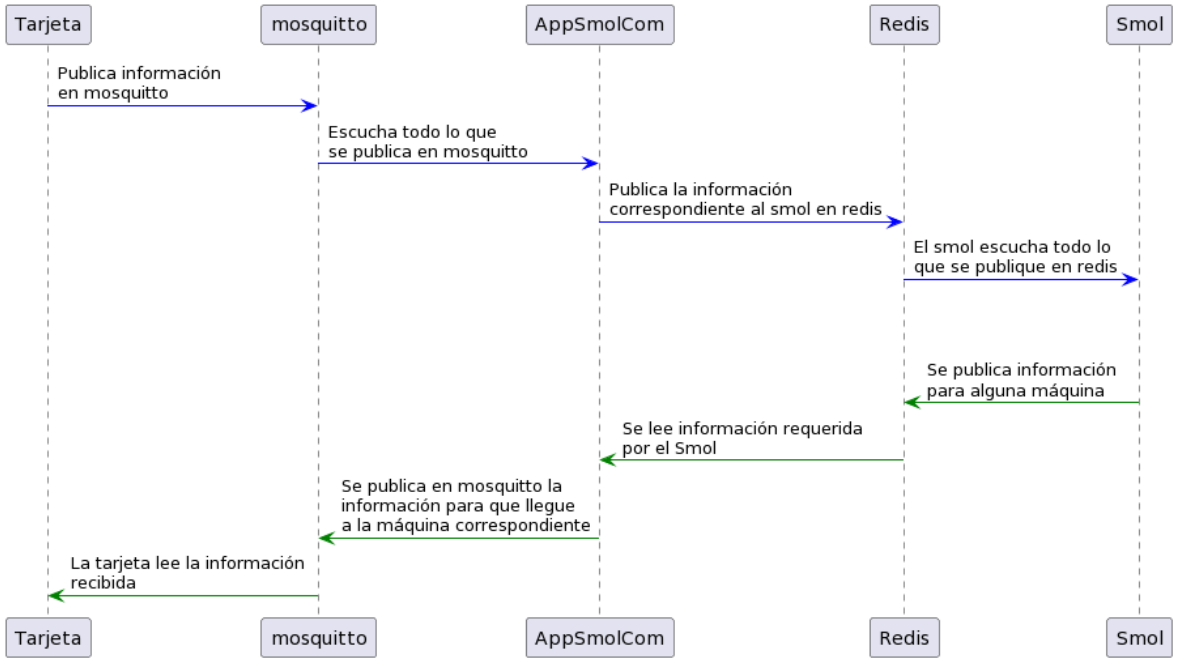

Figura 1: Flujo de interacción de la aplicación con la tarjeta y el smol

Establecido el flujo se procedió a definir los tópicos en los que la tarjeta, la aplicación y el Smol se suscriben y publican, dichos tópicos se definieron de la siguiente manera.

casino/macMaquina/enviada

casino/macMaquina/recibida

Como se puede observar los dos tópicos a implementar son similares, ambos tienen el subtópico *casino*, que hace referencia al nombre del casino donde este corriendo el sistema, el segundo corresponde a la mac de la máquina, tópico único para cada una de estas, finalmente el último subtópico que diferencia a los dos tópicos, corresponde a "*enviada*" cuando la tarjeta envía información, es decir, pública en **mosquitto** en ese tópico específico, y "*recibida*" cuando llega un mensaje a ese tópico que es donde se encuentra suscrita.

En la figura 2 se muestran los módulos y cada tópico en el que se suscribe o pública cada uno de ellos.

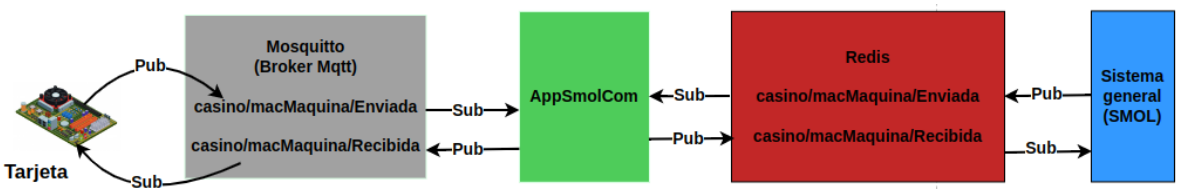

Figura 2: Módulos, tópicos y flujo de suscripción y publicación de cada componente

Teniendo claro lo anterior se hizo un diseño de diagrama de clases de manera general, donde solo se muestran las clases principales, las demás clases y métodos que se necesitaron fueron incluidos durante su implementación, esto teniendo siempre en cuenta el bajo acoplamiento y la alta cohesión entre las clases, en la figura 3 se observan dicho diseño.

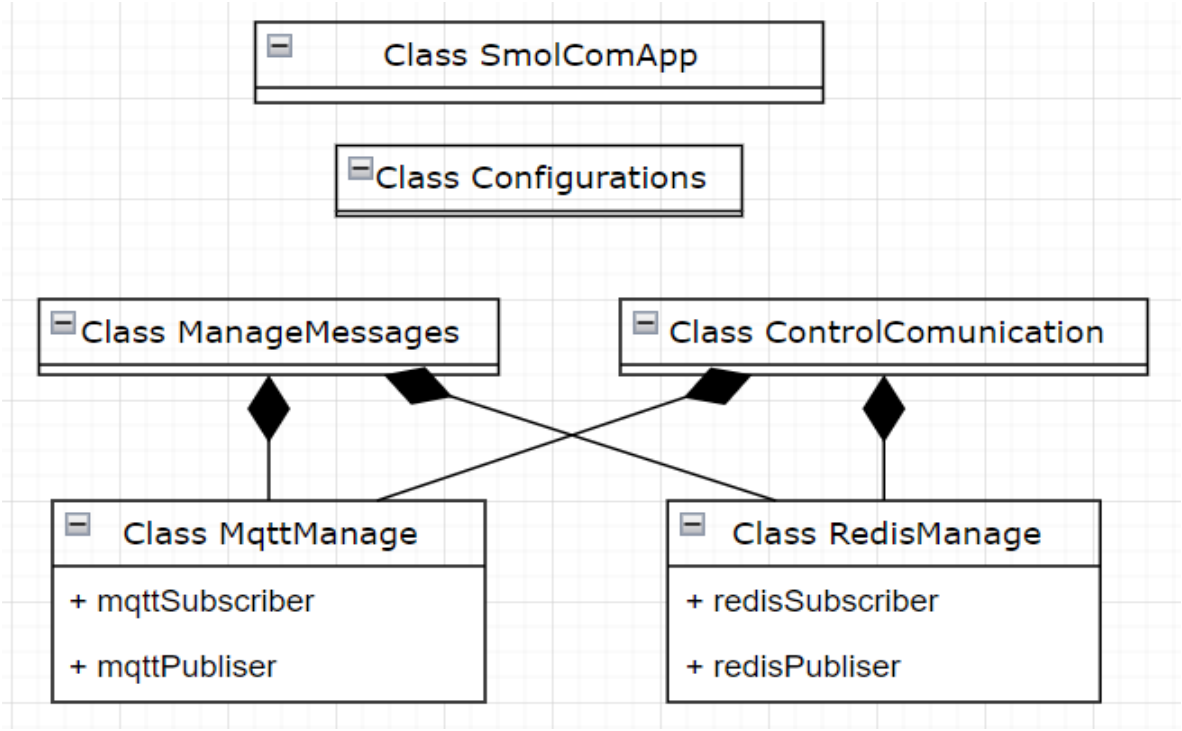

Figura 3: Diseño de diagrama general de clases para la aplicación

Class SmolComApp: Clase principal del proyecto, dentro de ella solo está el main y no se implementó ninguna lógica.

Class Configuration: Implementa las configuraciones para redis y mosquitto

Class ManageMessages: Manipula y hace llegar los mensajes al destino correspondiente, ya sea que provengan de las tarjetas o del SMOL.

Class ControlComunication: Verifica la comunicación con las tarjetas, si alguna de estas se desconecta, publica en redis para informal al smol sobre esta desconexión.

Class MqttManage: Encargada de implementar el publicador y suscriptor para el servidor **mosquitto**.

Class RedisManage: Encargada de implementar el publicador y suscriptor para redis.

Planteado el diseño y teniendo en cuenta que cada una de estas clases puede encapsular otras para cumplir con lo requerido, se procede a hacer la implementación en código fuente, a continuación, se describe este proceso.

Primero se preparó el entorno de desarrollo, se descargó **mosquitto** y **redis** en su última versión, el computador con el que se trabajó ya tenía eclipse para el desarrollo del código en java, por lo que no fue necesario hacer la descarga. Teniendo esto listo se dividió el proceso en cuatro etapas.

Etapa 1: comunicación del lado de la tarjeta.

Etapa 2: Comunicación del lado del smol.

Etapa 3: Unión de las dos etapas anteriores e implementación de la manipulación de los mensajes.

Etapa 4: Control de comunicación con las tarjetas.

Etapa 1: Para la comunicación con la tarjeta se procedió a hacer la respectiva configuración, haciendo que la aplicación escuche cualquier mensaje que llega a **mosquitto**, esto se logró usando el libreria paho de java que facilita crear un cliente *Mqtt*, el cual con el carácter especial "#" de **mosquitto**, puede escuchar todos los tópicos, logrando que cada vez que haya una publicación se pueda obtener el mensaje y el tópico en el que se publicó, para posteriormente procesar a conveniencia esta información.

Luego se implementa el publicador, que como parámetros recibe el mensaje y el tópico donde se debe publicar, esto se hizo creando un cliente mqtt con ayuda de la librería *paho*, el cual se conecta, envía el mensaje y posteriormente se desconecta.

Etapa 2: Para la comunicación con el smol, también fue necesario crear una configuración de la misma manera como se hizo del lado de la tarjeta, de tal manera que la aplicación quedó escuchando cualquier mensaje que llegue a **redis**, en este caso se usó el carácter especial "*" y se implementó la clase *redisMesageSuscriber*, que ejecuta la acción correspondiente cuando llega un mensaje. De todo esto es importante destacar que se usó un cliente ***redisson***, por ser uno de los que tiene mejores características de funcionalidad respecto a los demás.

Etapa 3: Con las conexiones listas y logrando publicar tanto en **mosquitto** como en **redis**, se implementó la lógica para la gestión de los mensajes. En esta etapa fue necesario tener en cuenta las prácticas sugeridas por la empresa, que corresponden a separar la lógica de negocio y la parte de servicios, poniéndolas en dos paquetes diferentes, *domain* y *service* respectivamente. Con esto en

mente se puede modularizar las clases que ya se habían implementado, siendo necesario crear paquetes que permiten encapsular las clases correspondientes.

La clase de configuración se divide en dos clases, una para **mqtt** y otra para **redis** (*MqttConfig* y *RedisConfig*), de igual manera la implementación de **redis**, se divide en publicador y suscriptor (*RedisMessagePublisher* y *RedisMessageSubscriber*).

Con esto listo se empezó a hacer la manipulación de los mensajes que como es de imaginarse, hacen uso de las clases de Mqtt y de **redis** como lo requieran.

Teniendo en cuenta que al terminar la implementación el código debía testearse, entonces se crean interfaces para la comunicación entre las clases, con el fin de facilitar el proceso de testeo.

La clase principal en este caso es *ManageMessageMachinesImpl* que implementa la interfaz *IManageMessageMachines*, que contiene como método principal *classifyMessageReceived(),* encargado de verificar de donde proviene el mensaje, identificar de qué tipo es y de acuerdo a ello mira si es un mensaje de confirmación de conexión, pedido de configuración de la máquina o un mensaje cualquiera, para el mensaje de confirmación de conexión, guarda como activa la mac de esa máquina en un mapa que contiene las máquinas conectadas, en el caso de que sea un pedido de configuración, agrega esa máquina al mapa de máquinas conectadas y publica en **redis** la solicitud de configuración de esa máquina, en el caso de que no sea ninguno de los anteriores, simplemente pública en **redis** en el tópico correspondiente sin enterarse de lo que contiene el mensaje.

Dado que se requiere estar manipulando mucho el mapa de macs de las máquinas, se creó la clase *ManageMacsMachinesImpl*, esta implementa la

interfaz *IManageMacsMachines*, donde se definieron las acciones que se requiere en el mapa, como agregar, eliminar, modificar, buscar y demás que implican la manipulación de este.

Es importante aclarar que la manipulación de los mensajes se hizo más por parte de la información que proviene de las tarjetas, puesto que la información que proviene del sistema general (Smol), al ser escuchada se procede a publicarla en **mosquitto** sin la necesidad de procesarla.

Finalizando esta etapa se logró tener el flujo requerido, donde un mensaje que publica la tarjeta llega hasta la base de datos **redis**, donde es recibido por el smol, y de la misma manera, un mensaje publicado por el smol en **redis** llega a la tarjeta de destino.

Etapa 4: Para el control de comunicación con las tarjetas se implementó una clase que no tiene que ver con el flujo de las etapas anteriores, esta tiene una configuración especial para que se ejecute cada minuto, dicha configuración se implementa en la clase *ScheduledConfig*, para su funcionamiento hace uso del publicador mqtt y del mapa de macs que manipula la clase ManageMacsMachinesImpl, con el fin de publicar un mensaje a las máquinas para que estas le respondan en caso de que estén conectadas, y de esta manera llevar el control de la comunicación e informar al sistema general (SMOL), en caso de que alguna deje de responder.

En la figura 4 se observa el empaquetado que se mencionó anteriormente, donde vemos que el paquete **config** contiene las tres configuraciones necesarias para **Mqtt**, **redis** y **Scheduled**, el paquete de **domain,** contiene al paquete *core*, el cual a su vez contiene dos interfaces y el paquete *impl,* que corresponde a la implementación de esas dos interfaces. Por otro lado, se tiene el paquete **service** que contiene al paquete *core*, que a la vez contiene dos interfaces que

corresponden a la gestión de las macs y de los mensajes de las máquinas, también tiene el paquete *impl* que es donde se implementan las interfaces, por último, se tiene los paquetes *comunicationmachine* y *comunicationsmol* que contiene las clases encargadas de publicar y suscribirse tanto en **mosquitto** como en **redis**.

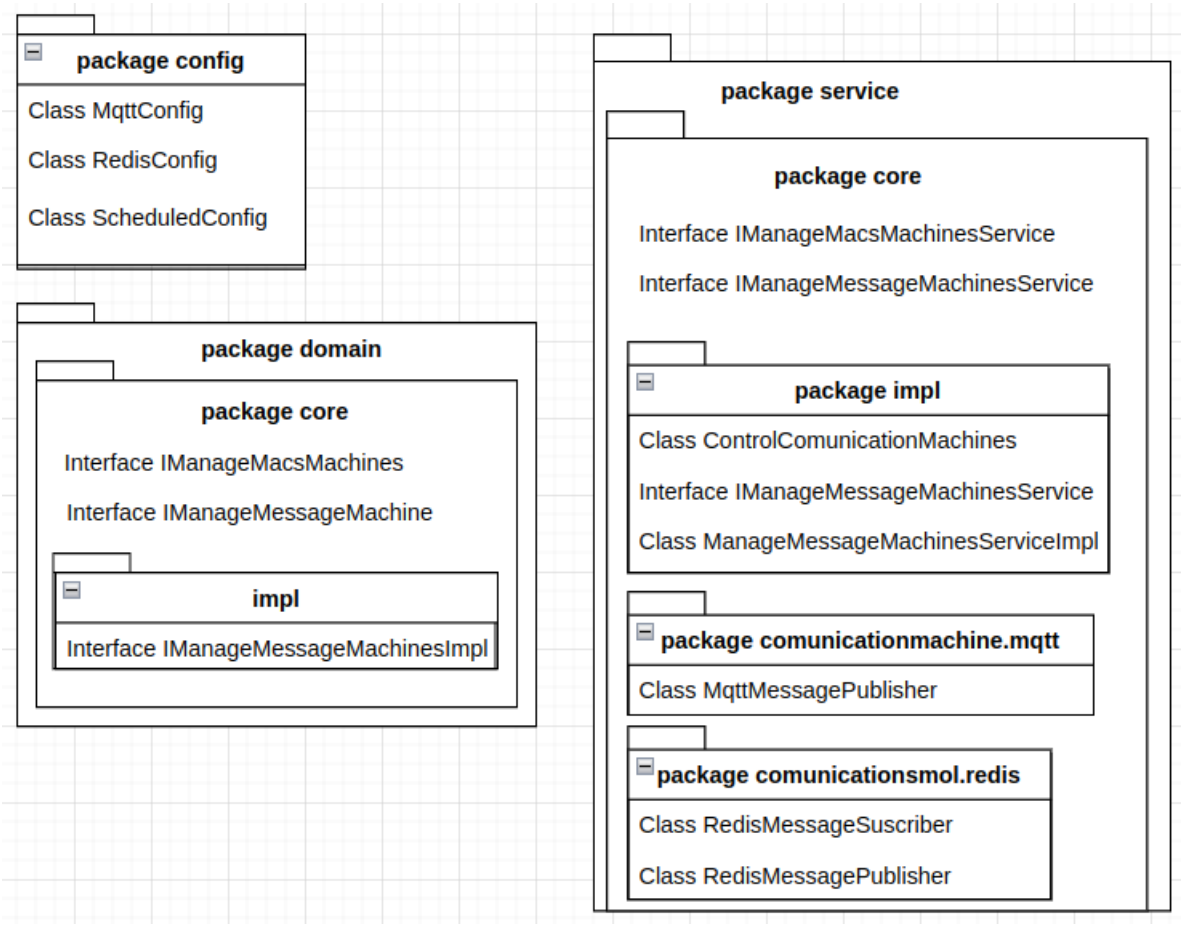

Figura 4: Estructura de implementación de paquetes y clases de la aplicación

En la figura 5 se observa en detalle la relación entre las clases mencionadas anteriormente, donde se ve las interfaces que cada una implementa y la conexión que tiene con las demás.

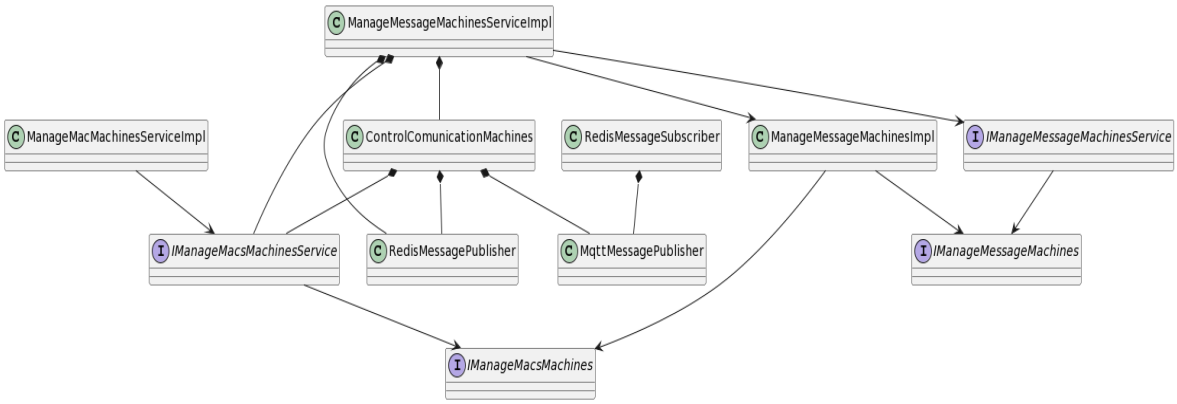

Figura 5: Diagramas de clases de la aplicación

La manera como se distribuyeron las clases y los paquetes se hizo pensando en facilitar el entendimiento de la implementación, de tal manera que la encapsulación nos permita tener un bajo acoplamiento y facilitar la modificación del proyecto en un futuro.

Teniendo la implementación funcional de la aplicación, se procedió a hacer una prueba unitaria al método más importante que corresponde a *classifyMessageReceived()* de la clase *ImanageMessageMachine,* para validar el funcionamiento del flujo a la hora de clasificar un mensaje que se recibe desde la tarjeta. Para esto se usó la librería *Mockito* de java, que permite simular los parámetros que recibe el método y la respuesta de algunos métodos que se llaman dentro de este.

Finalmente se implementa en docker tanto **mosquitto** como **redis**, para verificar que la aplicación corre en este entorno, porque es donde finalmente estará corriendo después de implementar muchas más funcionalidades correspondientes a otros módulos. Para esto se configuró un docker compose que permite conectar la aplicación con **mosquitto** y **redis**, en este se especificó los puertos por donde se comunican y algunos *volumes* que fueron necesarios para

mapear direcciones locales a direcciones dentro del contenedor, correspondientes a configuraciones de **mosquitto**.

5. Resultados

Después de implementar todas las etapas anteriores se muestran los resultados parciales y finales de la aplicación, validando los objetivos establecidos.

De acuerdo con los objetivos, se plantearon requerimientos que fueron implementados bajo un buen diseño de diagramas de clases, con un bajo acoplamiento y alta cohesión, que posteriormente se escribieron en líneas de código fuente para dar solución a lo requerido.

A medida que se fue desarrollando el código se verificaron los resultados parciales de la aplicación, el primero corresponde a la conexión con la tarjeta. Para validar esto se creó un cliente que publica y se suscribe en un tópico de **mosquitto**, para ello se hace uso de dos consolas, donde una hace las veces de publicador y la otra de suscriptor, en las figuras 6 y 7 se observa las características de cada uno de estos, donde:

- mosquitto_pub: Comando para publicar en un tópico de **mosquitto**
- mosquitto_sub: Comando para suscribirse a un tópico de **mosquitto**

-h localhost: La url del broker

-t TopicEscuchaApp: El tópico donde se publica o suscribe

-m "Prueba de conexión": Mensaje a enviar

-u ies: Usuario

-P smol: Contraseña

-p 8883: Puerto

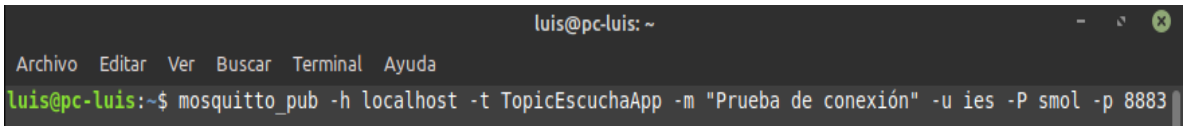

Figura 6: creación del cliente que simula a la tarjeta cuando publica

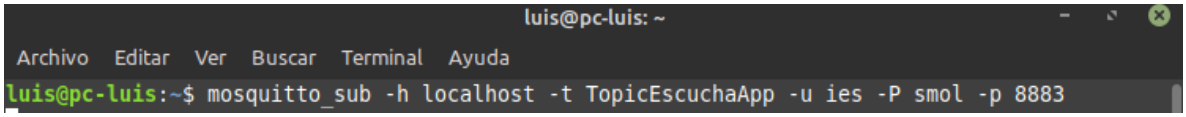

Figura 7: creación del cliente que se suscribe para simular la tarjeta

Teniendo el cliente listo se agregan logs en el código para mostrar que el mensaje que se publica llega a la aplicación, y un mensaje que se publica desde la aplicación llega a la consola que se encuentra suscrita, este resultado se observa en la figura 8, donde vemos que el mensaje que se publica llega a la aplicación, conociendo tanto el tópico como el mensaje, además, la consola que está suscrita al mismo tópico recibe la publicación del cliente creado y de la aplicación. De esta manera se verifica que tanto el publicador como el suscriptor funcionan correctamente.

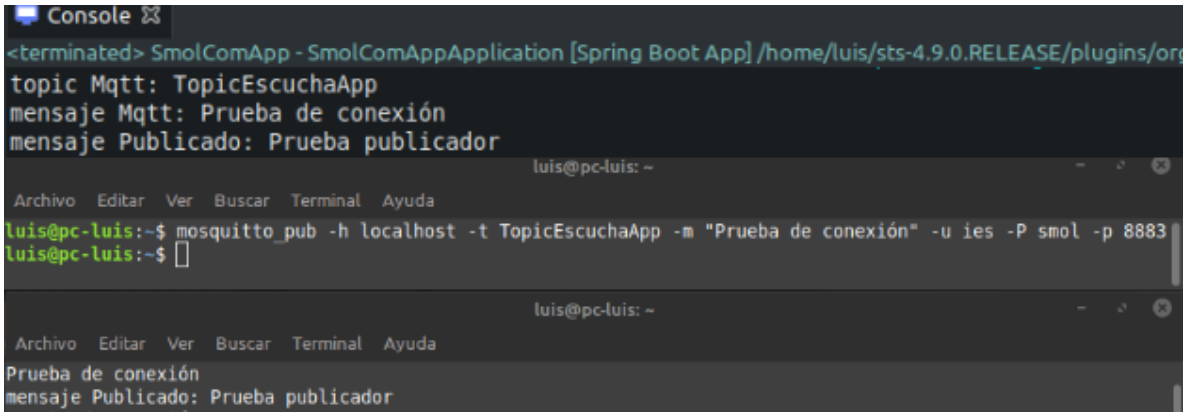

Figura 8: Consola de la aplicación, publicador y suscriptor de mosquitto

De la misma manera se verifican los resultados parciales de la conexión del lado del sistema general (SMOL), en este caso también se crean dos consolas para simular un cliente que publica y se suscribe en **redis**, en la figura 9 se observan

estos dos clientes, donde el publicador usa el comando **publish**, el tópico y mensaje a publicar, mientras que el suscriptor usa el comando **subscribe** y el tópico donde desea escuchar, en la misma figura vemos que el mensaje que se publicó en el tópico "*topicPruebaConexion*", llega al cliente y a la consola de la aplicación, además, el mensaje publicado desde la aplicación (*Mensaje enviado desde App*), llega a este cliente, permitiendo verificar que la conexión del lado del smol se ha establecido correctamente.

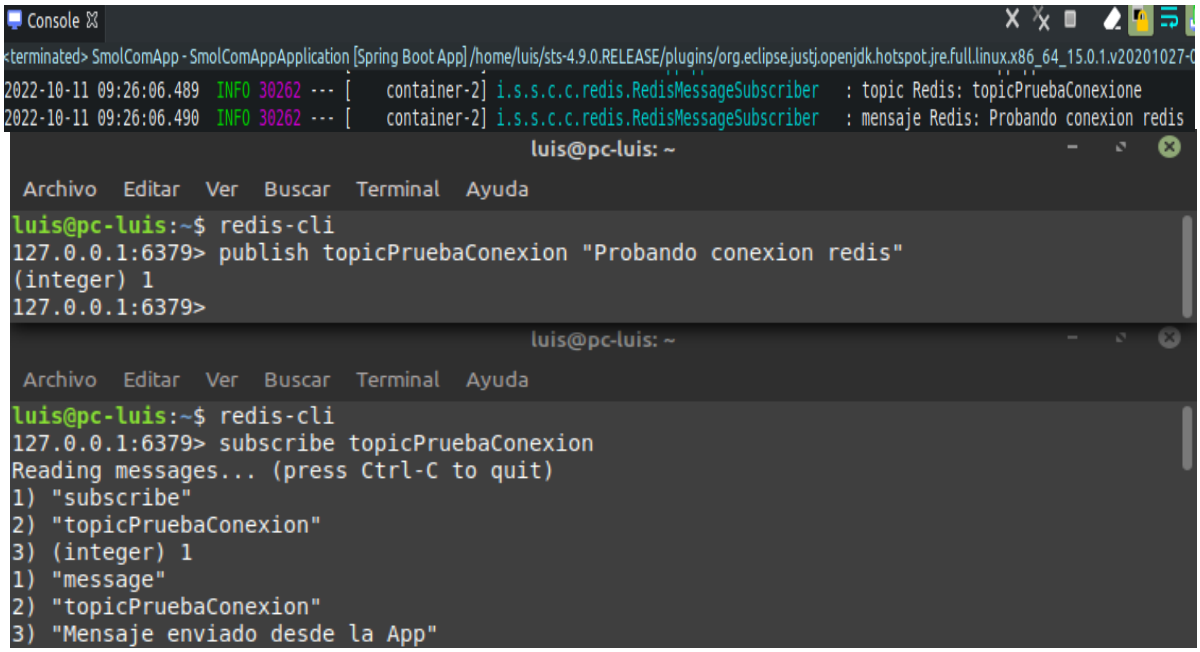

Figura 9: Consola de la aplicación y consolas que simulan el publicador y suscriptor de redis.

Con lo anterior funcional se procede a verificar el correcto funcionamiento de toda la aplicación, para ello se hace uso de los clientes ya creados, con la diferencia de que ahora se hace con comandos reales, para esta demostración se usa el comando **ConfInit,** usado por todas las máquinas para pedir configuración.

En la figura 10 y 11 se observan las consolas que harán las veces de publicador y suscriptor para simular la tarjeta, y en las figuras 12 y 13 para el cliente que simula el sistema general.

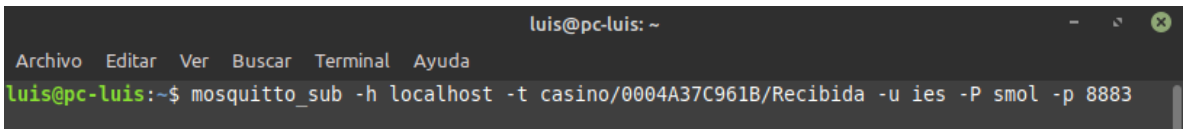

Figura 10: creación del cliente que se suscribe para simular la tarjeta

Figura 11 creación del cliente que simula la tarjeta cuando publica

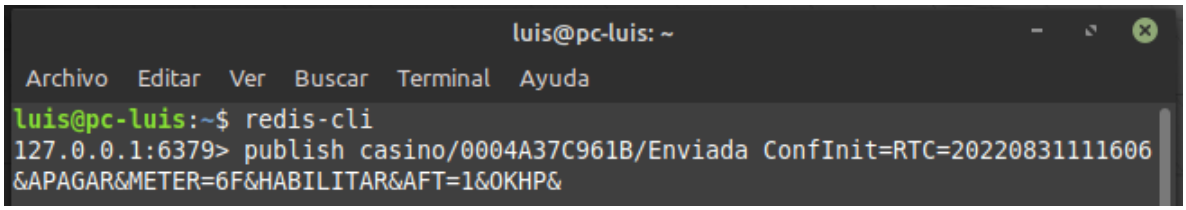

Figura 12: Cliente para simular el smol como publicador.

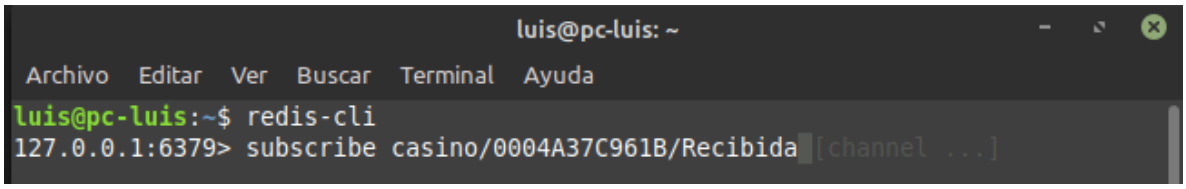

Figura 13: Cliente para simular el smol como suscriptor

Ahora al ejecutar estos comandos se obtienen los resultados que se muestran en la figura 14.

Para un mejor entendimiento el flujo que ocurre se describe a continuación; con la consola superior izquierda, se publica en mosquito el mensaje "ConfInit" que corresponde a pedir configuración para la máquina, vemos que en la consola inferior derecha justo llega el mensaje de solicitud de configuración, en seguida

desde la consola superior derecha se responde con la configuración, e inmediatamente llega a la consola inferior izquierda, que corresponde a la tarjeta que pidió la configuración, terminando así el ciclo para este mensaje.

Un comando "IsOk" también llegó a consola inferior izquierda, este corresponde al mensaje que se lleva para el control de la comunicación, pero como nunca se respondió a este mensaje, vemos que en la consola inferior derecha se reporta la desconexión de esta máquina.

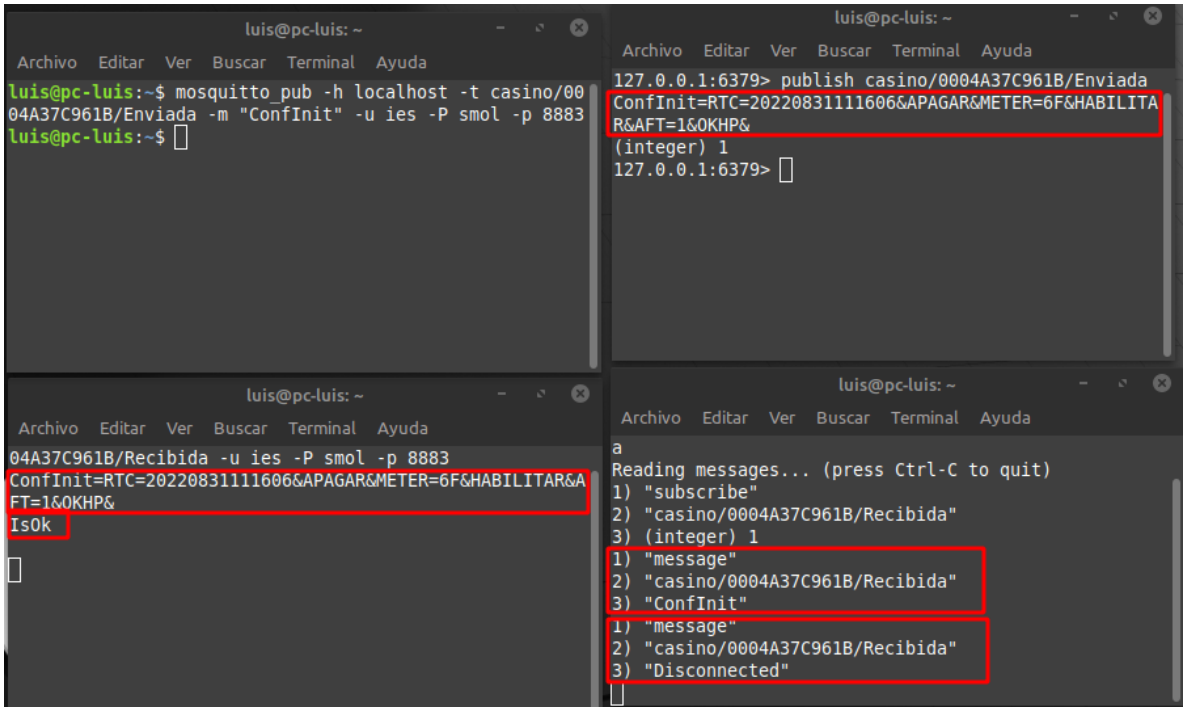

Figura 14: Resultados de simulación para el comando que pide configuración.

Finalmente, en la figura 15 se observa la consola de la aplicación, donde se muestra el flujo de mensajes que pasaron por ella mientras ocurría el evento descrito anteriormente.

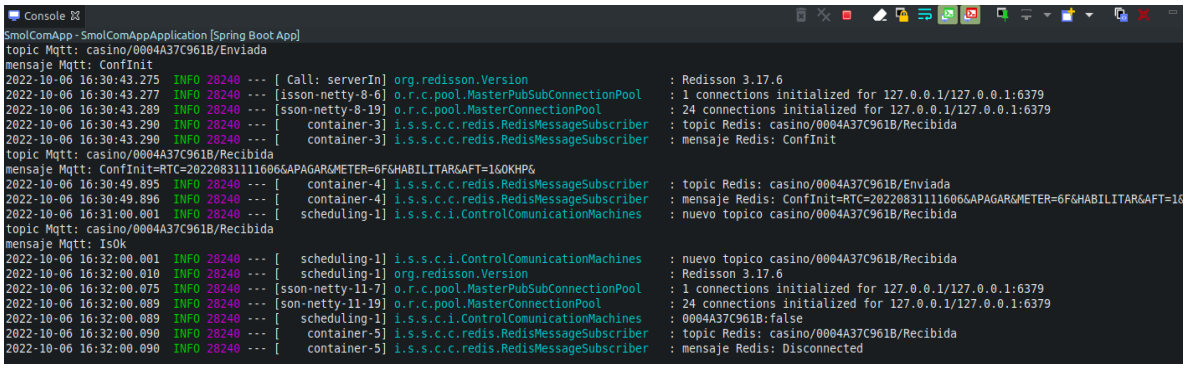

Figura 15: Flujo de mensajes en consola que llegaron desde el smol y la tarjeta

Por otro parte, para probar la aplicación a mayor escala se implementó un software que simula muchas tarjetas, buscando evidenciar errores de conexión, de estas pruebas se obtuvieron los resultados mostrados en la tabla 1.

Número de tarjetas simuladas	Errores de comunicación reportados
10	0
100	0
500	0
1000	0

Tabla 1: resultados de errores en pérdida de comunicación

De esta manera se verifica la implementación integral de toda la aplicación, donde se cumple con los requerimientos establecidos y se logran los objetivos planteados.

6. Conclusiones

Este trabajo de grado ha desarrollado un módulo de comunicación independiente, logrando comunicar las tarjetas con el sistema de general (smol).

Esto se logró implementando nuevas tecnologías como mosquitto y redis que sirvieron de intermediarios para facilitar la comunicación, además del buen diseño y desarrollo de cada clase involucrada.

El modelo de comunicación publicador suscriptor al ser asíncrono nos permite tener una mejor modularidad, además, aumenta el rendimiento y la escalabilidad, con esto el acoplamiento se reduce a un nivel donde hacer cambios a futuro resulta ser sencillo y rápido.

El protocolo Mqtt al ser simple, liviano y fácil de implementar, permite reducir los recursos de cómputo notablemente, es utilizado en el internet de las cosas (IoT) y cada vez toma más fuerza en la industria, por lo que en este caso resultó ser útil y confiable para la comunicación de las máquinas con el sistema de gestión.

El uso de redis permite el acceso a la información de forma rápida, puesto que al ser una base de datos en memoria su acceso es inmediato, y hace que el sistema tenga un mejor rendimiento cuando se presentar grandes flujos de información, resultando conveniente para nuestro caso ya que nos evita ir al sistema general a buscar la información requerida.

7. Proyectos a futuro

Como el módulo es exclusivamente para facilitar la comunicación entre las máquinas y el sistema de gestión, resulta ser un poco limitado, por lo que a futuro se podría implementar de manera más genérica, haciendo que no solo acepte máquinas, si no cualquier otro dispositivo que se quiera vincular al sistema.

Teniendo un módulo más genérico, se podría implementar como una librería, para facilitar múltiples conexiones que se quieran vincular al sistema.

8. Referencias Bibliográficas

[1] *Com.co*. [Online]. Available: http://iesonline.com.co.

[2]"Herramientas administrativas: Cómo pueden ayudar a tu empresa en 2022," *Com.mx*. [Online]. Available: https://blog.bind.com.mx/herramientas-administrativas.

[3] Netsoft, "8 Beneficios de contar con un Software de gestión empresarial a tu medida para impulsar tu negocio," *Netsoft*, 02-Mar-2021. [Online]. Available: https://netsoft.com/2021/03/02/beneficios-de-contar-con-un-software-de-gestion-empresarial-a-tu-medida-para-impulsar-tu-negocio/.

[4] "Integración de aplicaciones," *Jtech.ua.es*. [Online]. Available: http://www.jtech.ua.es/j2ee/2011-2012/restringido/arq/sesion03-apuntes.html.

[5] "¿Qué significa standalone en el mundo del desarrollo de software?," *Desarrolloweb.com*. [Online]. Available: https://desarrolloweb.com/faq/que-significa-standalone-en-el-mundo-del-desarrollo-de-software.

[6] *Amazon.com*. [Online]. Available: https://aws.amazon.com/es/pub-sub-messaging/.

[7] "IBM Documentation," *Ibm.com*. [Online]. Available: https://www.ibm.com/docs/es/iis/1.

[8] "MQTT," *Paessler.com*. [Online]. Available: https://www.paessler.com/es/it-explained/mqtt.

[9] "MQTT Version 3.1.1," *Oasis-open.org*. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.

[10] Luis, "Qué son y cómo usar los Topics en MQTT correctamente," *Luis Llamas*, 19-Jun-2019. [Online]. Available: https://www.luisllamas.es/que-son-y-como-usar-los-topics-en-mqtt-correctamente/.

[11] "Eclipse Mosquitto," *Eclipse Mosquitto*, 08-Jan-2018. [Online]. Available: https://mosquitto.org/.

[12] *Amazon.com*. [Online]. Available: https://aws.amazon.com/es/elasticache/what-is-redis/.

[13] "Redis Java client with code example," *Redisson*. [Online]. Available: https://redisson.org/redis-java-client-with-code-example.html

[14] IBM Cloud Education, "What is Java Spring Boot?," *Ibm.com*, 25-Mar-2020. [Online]. Available: https://www.ibm.com/cloud/learn/java-spring-boot.

[15] I. Craggs, "Eclipse paho," *Eclipse.org*. [Online]. Available: https://www.eclipse.org/paho/index.php?page=clients/java/index.php.

[16] *Amazon.com*. [Online]. Available: https://aws.amazon.com/es/docker/.

[17] "¿Qué es Docker Compose? | KeepCoding Tech School". KeepCoding Tech School. [Online]. Available: https://keepcoding.io/blog/que-es-docker-compose/

[18] G. Eichemberger, "¿Cómo puedo crear un volumen en docker?," *Somos PNT - Desarrollamos Software*, 17-Mar-2022. [Online]. Available: https://somospnt.com/blog/252-como-puedo-crear-un-volumen-en-docker.

[19] M. M. Canelo, "Qué es el testing de software," *Profile Software Services*, 07-Sep-2021. [Online]. Available: https://profile.es/blog/que-es-el-testing-de-software/.

[20] "Mockito Tutorial," *Tutorialspoint.com*. [Online]. Available: https://www.tutorialspoint.com/mockito/index.htm.