Proyecto: Agentes Autónomos de Prevención

Facultad de Ingeniería, Universidad de Buenos Aires

Grupo “La Krupoviesa”

Rafael Ortegano - 108313
Martin Gonzalez Prieto - 105738
Gian Luca Spagnolo - 108072
Alexis Martin Ramos - 98891

Taller de Programación
1º Cuatrimestre 2024

Índice

Conceptos del Servicio de Envío de Mensajes

Servicio de mensajería asincrónica

Este servicio corresponde a un método de comunicación en el que el sistema coloca un mensaje en una cola y no requiere una respuesta inmediata para continuar el procesamiento. Los ejemplos incluyen una solicitud de información, explicación o datos necesarios, pero no de inmediato, permitiendo así establecer una comunicación sin necesidad de que ambos clientes (o por lo menos el receptor) estén activos. El emisor no se queda esperando una respuesta, por el contrario, puede seguir iniciando o manteniendo otras conversaciones de la misma naturaleza. Esta forma de comunicación permite que los clientes puedan conectarse y recibir mensajes cuando quieran, y que estos mensajes queden documentados en un registro de *logging*. Asimismo, se permiten varias interacciones a la vez, y garantiza que los mensajes se entreguen al menos una vez en el orden correcto.

Message brokers

Conforman un módulo de software que está orientado a la mensajería que define un estándar o protocolo de comunicación entre los componentes de una aplicación. Funciona como una abstracción de comunicación, encargándose del protocolo a utilizar y del flujo de mensajes entre los distintos componentes, permitiendo así la implementación de la mensajería asincrónica. Existen varios modelos, que cubren distintas necesidades y dan lugar a distintas implementaciones de message brokers, cada uno puede usar una estructura de datos o protocolos distintos.

Modelo de Message Brokers

- **Mensajería punto a punto**

Patrón que consiste en el uso de colas de mensajes, en donde cada mensaje tiene un sólo destinatario y se consumen una sola vez, y un emisor tiene la capacidad de enviarle a uno o varios receptores definidos, el mismo o distintos mensajes.

- **Mensajería de publicación / suscripción**

También conocido como “pub/sub” (editor/receptor). A diferencia del patrón anterior, los clientes eligen qué tipos de mensajes quieren recibir, esto mediante suscripciones a uno o más tópicos de interés, los cuales el emisor publicará los respectivos mensajes que sean enviados a dichos tópicos, permitiendo así que los clientes puedan recepcionarlos.
El emisor no necesita conocer a los receptores, pero los receptores necesitan conocer los tópicos para poder suscribirse. Esto permite establecer órdenes jerárquicos entre los tópicos de mensajes.

Protocolos de Message Brokers

MQTT (Message Queuing Telemetry Transport)

Es un protocolo perteneciente al modelo pub/sub que utiliza encabezados para identificar los diferentes paquetes que recibe, y responde a cada mensaje con un acknowledge correspondiente. Este protocolo provee diferentes componentes:

- **Cliente MQTT**: componente de la aplicación que implementa la librería MQTT.
- **Agente MQTT (Broker)**: componente de la librería MQTT que se encarga de gestionar los mensajes, y reenviarlos a aquellos clientes suscritos al tópico el cual se ha enviado.
- **Conexión MQTT**: canal de transmisión entre cliente y agente. No permite la conexión de clientes entre sí, sino que se vale del uso del protocolo TCP/IP para establecer una conexión mediante el agente.

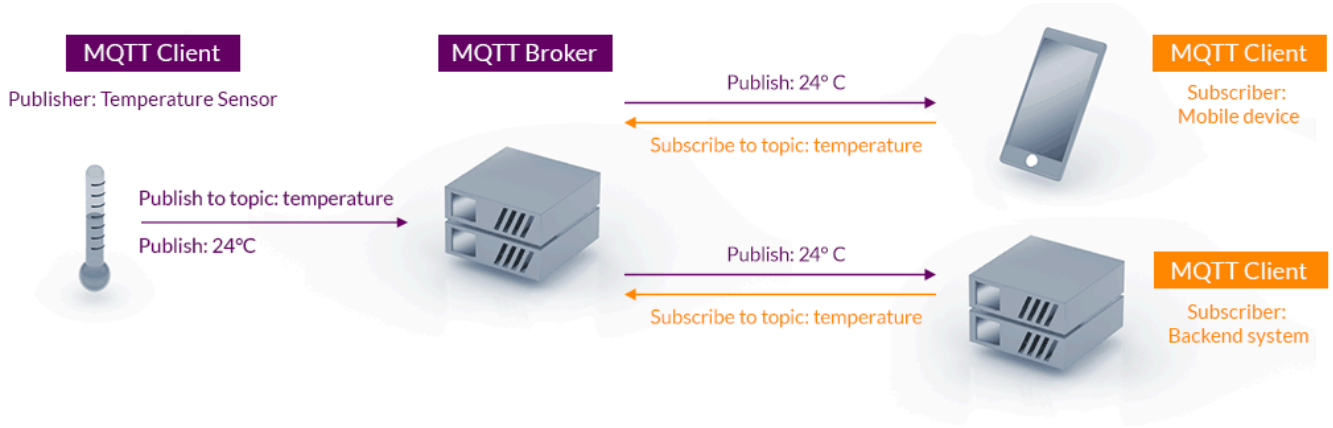

Fig 1. Visualización de protocolo MQTT entre diferentes dispositivos

El protocolo, como se puede visualizar en la Fig 1, dispone de la posibilidad de generar tópicos los cuales los clientes se suscriben y permiten el ingreso de mensajes enviados por clientes a aquel tópico, con el objetivo de filtrar los mensajes e incluso organizarlos por jerarquía.

Para suscribirse, los clientes envían un packet Subscribe sobre el tópico correspondiente, y enviar los mensajes con un packet Publish, referidos a un tópico y enviando unos datos definidos por el cliente, en formato de bytes.

Cuenta con 3 configuraciones posibles de calidad de servicio que define el publicador:

- QoS 0: Se entrega un mensaje como máximo una vez, sin garantía de recepción.
- QoS 1: Se envía un mensaje y se espera por el reconocimiento de que el mensaje ha sido entregado correctamente.
- QoS 2: Se confirma un mensaje mediante un handshake, teniendo al menor 4 mensajes involucrados en total, garantizando no solo confirmación del mensaje sino de la confirmación enviada, lo que disminuye el rendimiento.

Quality of Service: At Least One

En el QoS 1 de MQTT, se enfoca en garantizar la entrega del mensaje **al menos una vez** al receptor. Cuando se publica un mensaje con esta calidad de servicio, el **cliente** guarda una copia

del mensaje hasta que recibe un paquete Acknowledge del **receptor** (servidor), confirmando la recepción exitosa. Si el remitente no recibe el paquete de reconocimiento dentro de un plazo razonable, retransmite el mensaje para asegurar su entrega.

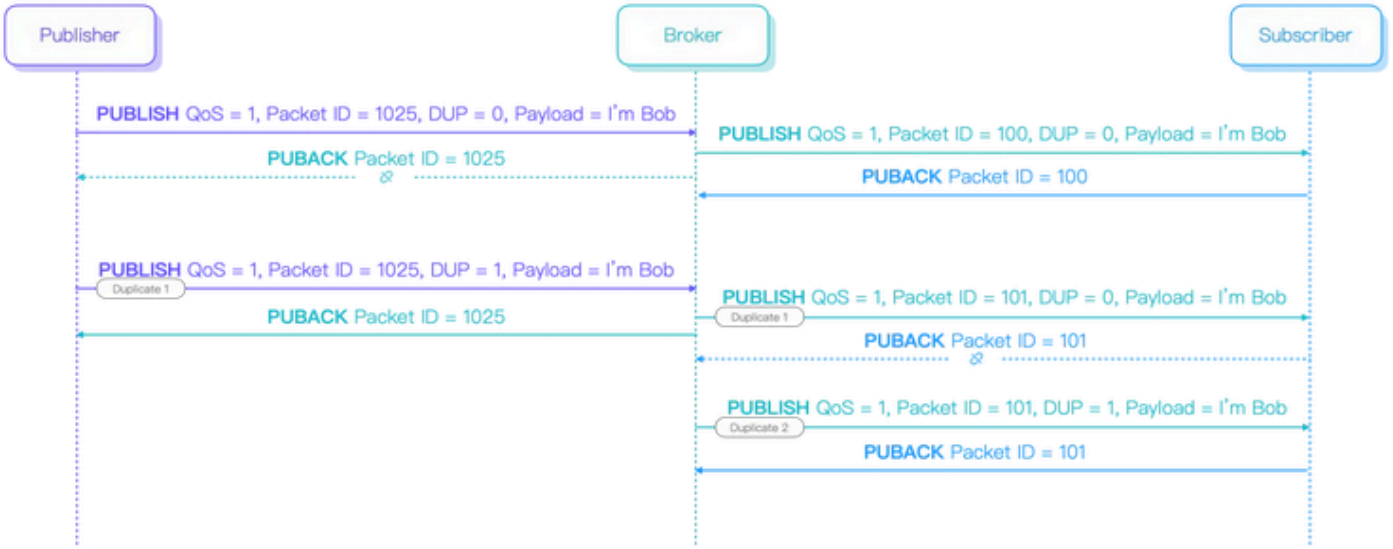

Fig 2. Representación de la Calidad de Servicio nivel 1: entrega de "al menos una vez"

Al recibir el mensaje, el receptor puede procesar el mensaje inmediatamente. Por ejemplo, si el packet enviado fue un Publish, y el receptor es un *broker MQTT*, distribuye el mensaje a todos los clientes suscriptores y responde con un paquete Puback para confirmar la recepción del packet.

Este enfoque de QoS 1 logra un equilibrio entre confiabilidad y eficiencia, garantizando que el mensaje llegue al receptor al menos una vez y permitiendo que los posibles duplicados se manejen adecuadamente.

Características del protocolo MQTT

- Su fácil implementación y fiabilidad permiten una correcta escalabilidad.
- Garantiza que los mensajes se reciban al menos una vez en el orden estipulado, brindando también numerosas posibilidades de QoS.
- Desacopla el remitente del emisor, y permite que el flujo de comunicación funcione aunque ambos no estén conectados al mismo tiempo, permitiendo un envio de mensajes sin interrupción.
- Se usa para dispositivos de IoT por su bajo coste de recursos, y utiliza bajo ancho de banda, lo que lo hace perfecto para redes inalámbricas.
- No se basa en la comunicación solicitud-respuesta.

AMQP (Advance Messagge Quening Protocol)

Es un patrón versátil que puede modificarse para aplicar tanto para el modelo punto a punto, como el de pub/sub. Dispone de los siguientes componentes, cada uno con su respectivo ID:

- Corredor de mensajes (broker): Es el emisor central de mensajes.
- Usuario (cliente): Es la entidad que se conecta a los corredores para comunicarse.
- Conexión: Entre usuario y corredor, como por ejemplo TCP/IP
- Canal: es el medio lógico por el cual se conectan usuario y servidor. Por ejemplo, un corredor puede tener un canal de envío y otro de recepción para cada usuario.
- Intercambiador: Entidad filtro que se encarga de enviar los mensajes a la cola de mensajes correspondiente, determinado por una vinculación. Por ejemplo, intercambiador directo, intercambiador abanico, intercambiador de tema definido, intercambiador de encabezamientos, etc.

Este protocolo dispone de **Frames AMQP** que representan la unidad básica del patrón: un conjunto ordenado de frames que componen una conexión y respeta el orden al enviarlos. Los frames se componen de 3 partes: Header, Header extendido y el Cuerpo, el cual puede adoptar diferentes formas para modificar coherentemente el estado de los componentes.

También se dispone de **Mensajes** y de una **Cola de Mensajes**, sobre los cuales ante un caso de mensaje no entregado (o fallido), se puede establecer que el consumidor pueda hacer un *acuse* de recibo, o reintentar enviar el mensaje nuevamente. También permite la posibilidad de que un consumidor rechace un mensaje, por el motivo que sea, haciendo que se deba borrar o colocarlo en otra cola.

Características del protocolo AMQP

- Es un patrón versátil y escalable y con alta seguridad, permitiendo operar grandes volúmenes de datos y ofreciendo posibilidades para varios modelos de brokers.
- Tiene alta fiabilidad y alta interoperabilidad, permitiendo compatibilidad plena entre múltiples dispositivos.
- Tiene una necesidad de que los clientes estén continuamente activos, lo que puede no cumplir con reliabilidad en caso de que el cliente no esté activo y no cuente con acuse de recibo.
- Por su complejidad, presenta una dificultad mayor para implementar.

Comparación entre protocolos

Si bien AMQP posee una curva de aprendizaje demasiado pronunciada (más aún frente a la curva de aprendizaje de MQTT), este demanda un mayor uso de recursos y ofrece una versatilidad mayor, así como mayor soporte en términos de seguridad.
Sin embargo, MQTT demanda un menor costo de implementación, y ofrece un marco de uso excelente para cumplir el modelo pub/sub, lo que lo convierte en el protocolo predominante en el marco del IoT.

Implementación del Proyecto

Requerimientos del proyecto

Se implementa un protocolo a elección, recomendando seguir el patrón de comunicación *publisher-subscriber* y la arquitectura *cliente-servidor*, para lo cual se implementa por un lado el servidor de mensajería y por otro lado una library que permite la comunicación por parte de los clientes, garantizando seguridad (encriptación, autenticación, autorización), reliabilidad, configuraciones, logging y, como mínimo, un Quality of Service (QoS) 1.
Siguiendo este desarrollo, se crean 3 aplicaciones diferentes que utilizan el protocolo implementado:

- Aplicación de Monitoreo
- Sistema de Cámaras
- Software de Control del Agente (drone)

Cada aplicación tiene sus responsabilidades correspondientes e interactúan entre sí mediante un **broker** (servidor).
Asimismo, el proyecto es realizado en Rust, utilizando únicamente la librería estándar, así como aquellos crates permitidos por los docentes. Se implementan pruebas unitarias y de integración correspondientes, y se garantiza un funcionamiento en ambiente Unix / Linux. Por último, el código es formateado y la compilación del proyecto no genera advertencias de ningún tipo.

Elección del protocolo

Frente a los requisitos del proyecto, decidimos implementar el protocolo **MQTT** ya que presenta una mayor cantidad de elementos a favor a comparación del resto de los protocolos. Su compatibilidad con las aplicaciones pertenecientes al IoT, sumado al bajo costo de recursos para procesarlo y su fácil implementación llaman la atención ante el enunciado de las aplicaciones. Sin embargo, gracias a su QoS 1 y su amplia escalabilidad, respetando el modelo pub/sub que imponen los requisitos, determinan una superioridad a comparación de AMQP, por ejemplo. Debido a esto, MQTT es un protocolo que cumple con todas las características necesarias y óptimas para desarrollar un trabajo de esta magnitud.

Para la configuración de calidad de servicio se utilizara un QoS 1. La versión del protocolo a implementar será la 5.0 debido a su adaptación al dominio actual y sus características perfectas para cumplir con todo lo propuesto. Sin embargo, la versión 3.1.1 de MQTT también podría haber sido utilizada aunque con menos características novedosas.

Implementación del protocolo

Para cumplir con los requisitos previamente establecidos, se ha implementado el protocolo MQTT disponiendo de la totalidad del funcionamiento de su versión 5.0, así como respetando el QoS 1 para garantizar una conexión entre múltiples clientes mediante un broker. Para cumplir con la implementación de las aplicaciones, se ha dispuesto una interfaz de uso completo para un **Cliente MQTT**, el cual debería ser utilizado por aquellas aplicaciones que quieran conectarse al

servidor; y un **Server MQTT** el cual es utilizado por el broker configurado para establecer las conexiones y enviar los mensajes.

Para manejar la comunicación, se implementó la capacidad de establecer una conexión entre numerosos clientes con un mismo server, donde se envían *packets* entre sí, mediante el servidor MQTT. Estos fueron implementados en sus respectivas clases, y son usados por el cliente y el servidor siempre que les correspondan. Los *packets* comparten una misma estructura correspondiente a un *fixed header* el cual establece una distinción entre sí, y también notifica el largo del resto del mensaje de lo cual están acompañados.

La conexión se produce mediante un packet *Connect* por parte de un cliente al servidor, y luego este último envió un packet *Connack* representando el correspondiente acknowledge (reconocimiento) de que el envío de aquel packet ha sido exitoso. Luego, se envía un packet Auth el cual provee con la información necesaria para autenticar al cliente con el Servidor, disponiendo de una contraseña la cual el servidor validará junto a su ID para verificar si tiene permitido establecer una conexión, determinado por la posibilidad de crear una lista de clientes autorizados mediante el ID.

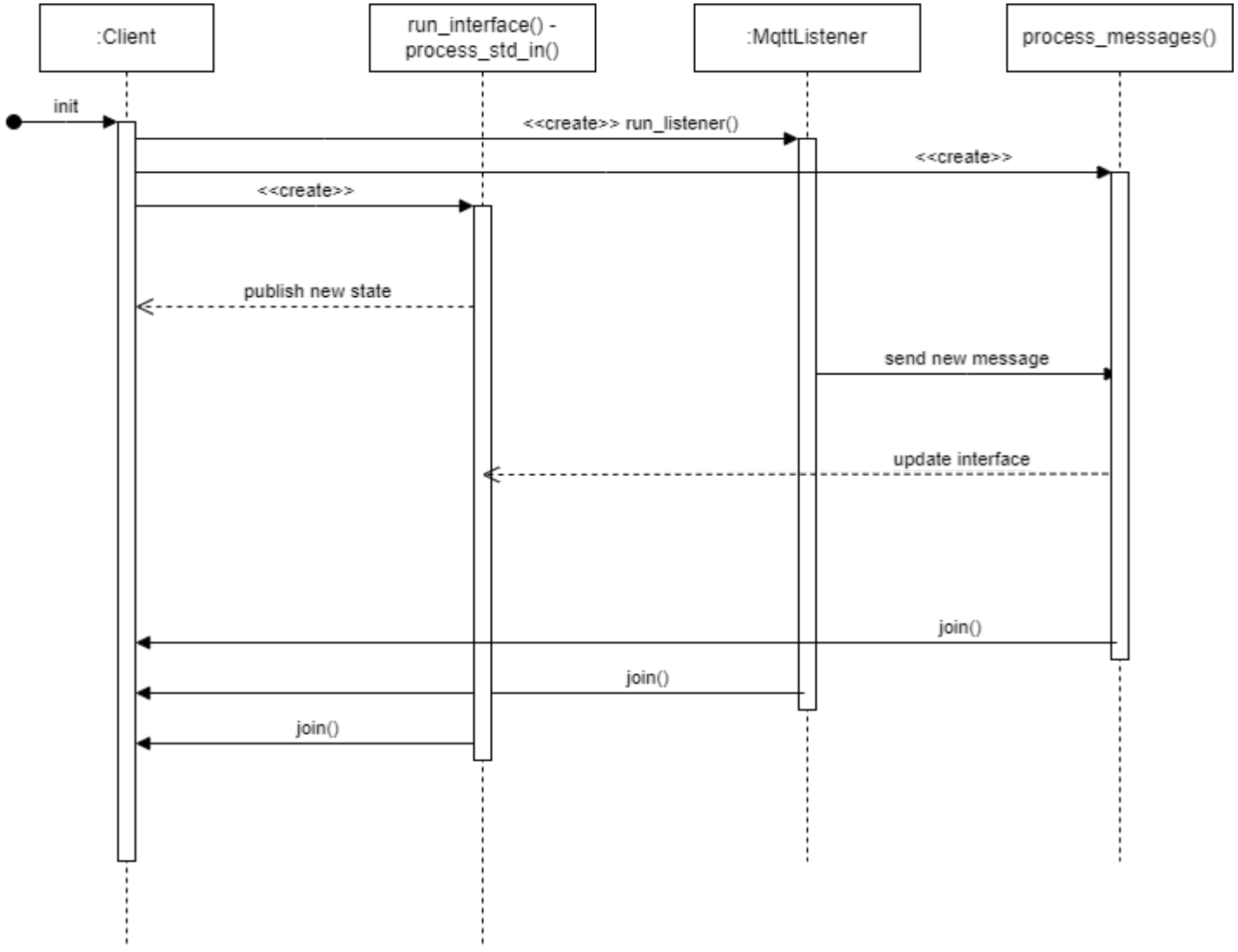

Fig 3. Diagrama de secuencia del cliente MQTT

Para el establecimiento de la conexión se utiliza *native_tls* permitiendo encriptar la conexión y establecer certificados validados por el sistema operativo, garantizando una capa de autorización más por parte del servidor, y estableciendo así la conexión mediante TLS.
El protocolo soporta la suscripción de los clientes a determinados tópicos, donde comenzarán a recibir los mensajes provenientes de aquellas aplicaciones que publiquen en aquel tópico. Se disponen de todos aquellos packets correspondientes para poder suscribirse y publicar mensajes, junto a sus respectivos acknowledge para validar el envío efectivo de aquellos mensajes.

Por último, se permite la desconexión de los clientes y garantizando la posibilidad de enviar un Will Message una vez se ha desconectado, para notificar a aquellas aplicaciones que estén suscritas a un determinado tópico que aquella aplicación se ha desconectado.

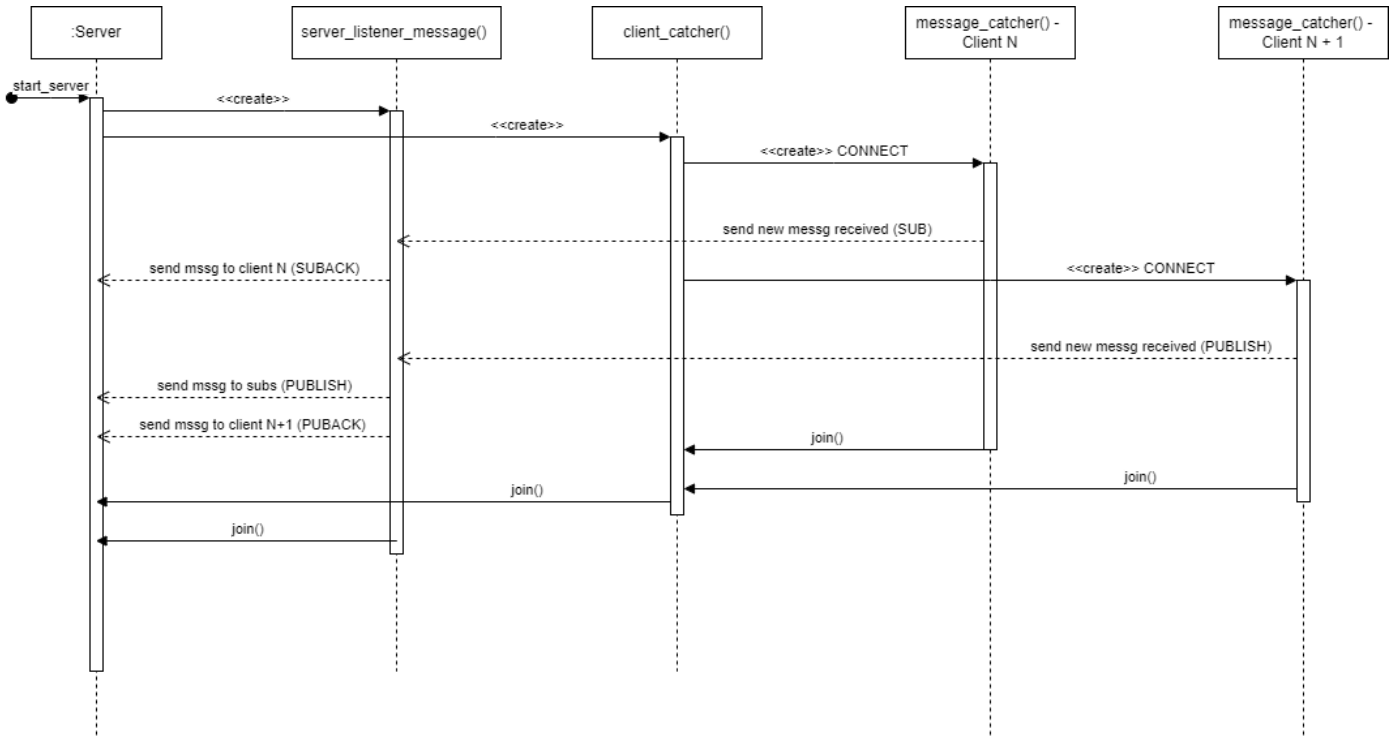

Fig 4. Diagrama de secuencia del server MQTT

Diagramas de actividad de las sesiones

En lo que respecta a la implementación de las sesiones, a continuación se presenta la lógica en diagramas de actividad. Esto es con el objetivo de dar a entender la interacción y las decisiones que debe tomar el servidor a la hora de establecer la conexión de un cliente, y a la hora de reenviar mensajes a los destinatarios correspondientes.

A continuación se demuestran, en las Fig 5. y Fig 6. un esquema representando la actividad entre los clientes y el servidor, figurando la conexión producida entre ambos como también el envío de mensajes mediante el broker.

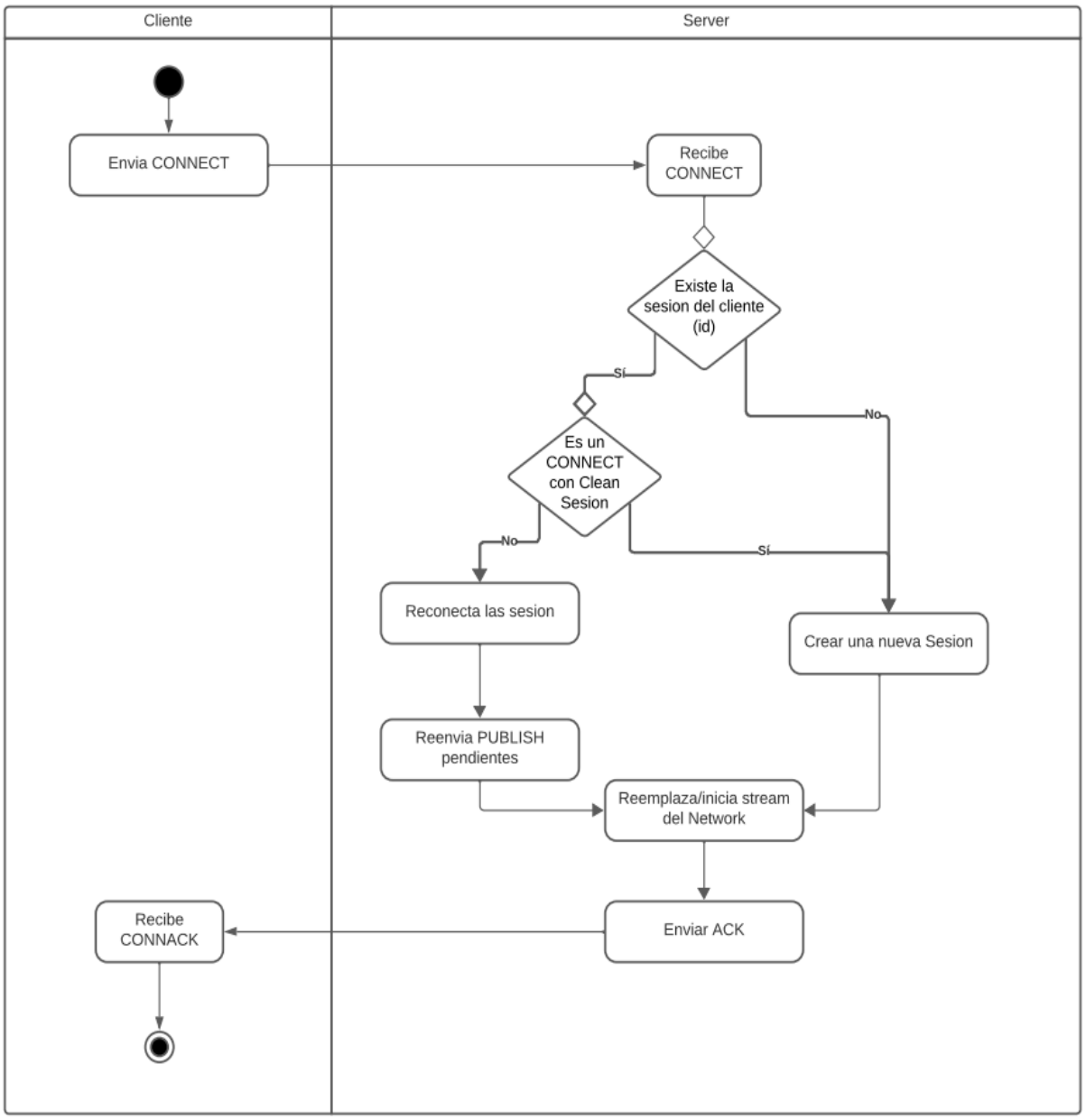

Fig 5. Conexión entre un cliente y un servidor MQTT

El servidor maneja las conexiones mediante el uso de sesiones, las cuales se crean la primera vez que la conexión es establecida, y persisten inclusive si el cliente se desconecta. Este correcto uso de la sesiones permite que aquel cliente que se ha desconectado pueda volver a hacerlo exitosamente con el servidor, así como establecer una cola de mensajes de aquellos tópicos a los cuales se ha suscrito previamente, de modo que una vez la conexión sea restablecida, reciba aquellos mensajes que se han enviado mientras no ha estado conectado, permitiendo así una comunicación eficiente inclusive si algun cliente se ha desconectado.

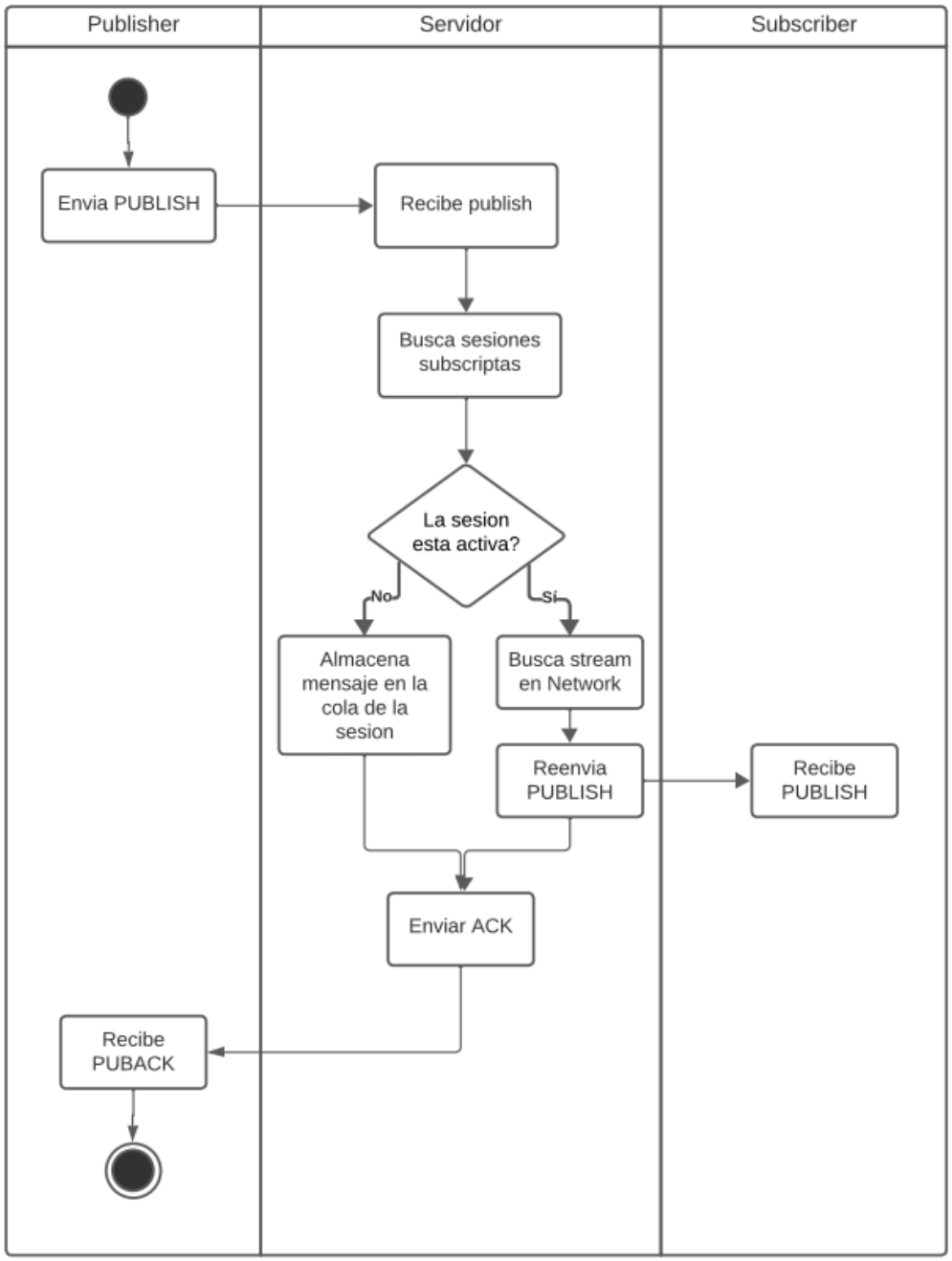

Fig 6. Demostración de la comunicación entre clientes, donde el servidor maneja la conexión mediante la sesión establecida

Manejo de los threads en el protocolo

Con respecto al manejo de los threads por parte del cliente, representado en la Fig 3. se utiliza un hilo para el desarrollo de la aplicación que utiliza el protocolo, enviando los mensajes correspondientes, mientras se dispone de otro thread *listener* que se encarga de la recepción de mensajes publicados que el servidor ha reenviado, para poder procesarlos y permitir que la aplicación decida el uso del mensaje. Ademas, cada aplicación puede tener otro thread que se encargue de establecer la interfaz gráfica, o de ejecutar la interfaz presente en la terminal.

Por el lado del servidor, representado en la Fig 4. se dispone de un hilo *listener* por cada cliente conectado el cual escucha aquellos mensajes enviados y se comunica mediante un thread con el servidor el cual procesa el mensaje y lo reenvía, en caso de ser necesario, a aquellos clientes que se encuentran suscritos al tópico del mensaje. También, existe un hilo que se encarga de la recepción de nuevas conexiones por parte de nuevos clientes que se conectan al broker.

Este manejo simple pero efectivo de la ejecución del cliente y server MQTT permite que el consumo de los recursos sea mínimo, y garantiza una correcta comunicación entre numerosos clientes mediante un broker.

Implementación de aplicaciones

Hemos creado un módulo para cada aplicación correspondiente (sistema de cámaras, drone, aplicación de monitoreo) así como para el broker (servidor). También, se dispone de un módulo correspondiente para el protocolo, el cual es utilizado por las aplicaciones.

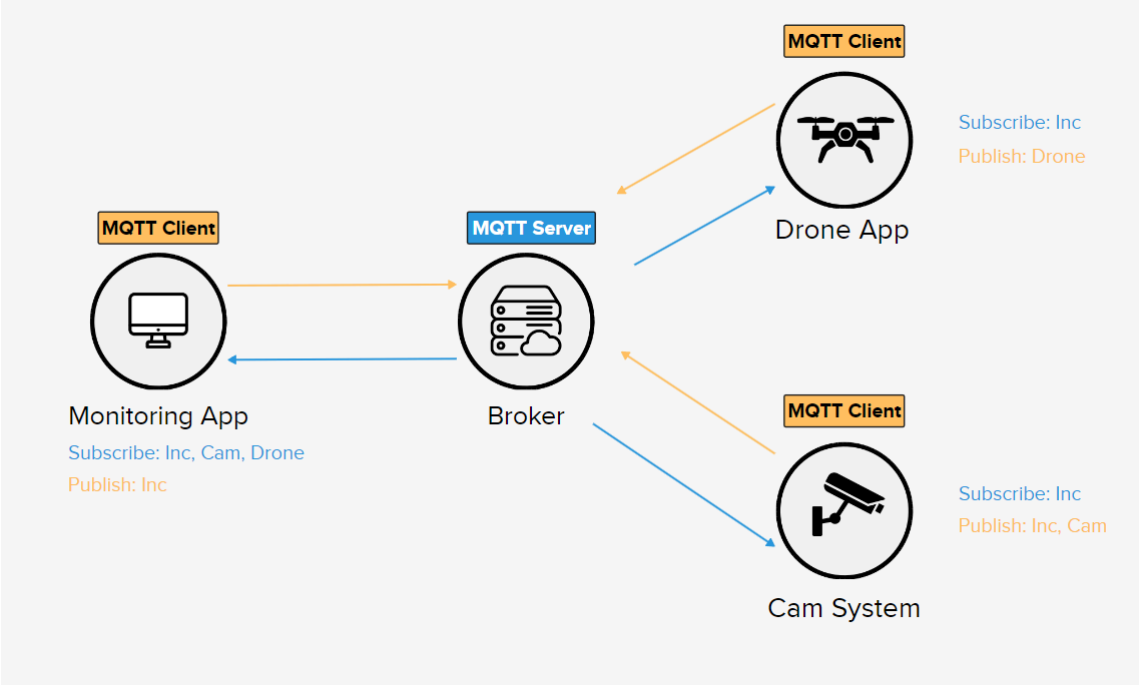

Fig 7. Instancias de las aplicaciones del proyecto. Aclaración: El Cam System publica en el tópico Inc debido a las modificaciones de la consigna final. Lo mismo con la suscripción de la Monitoring App al mismo tópico.

También, se destaca como cada aplicación posee su propio archivo de configuración, incluyendo características propias de su funcionamiento, como también un archivo de configuración propio del protocolo MQTT con sus características más importantes. Y además, cada una de las aplicaciones cuentan con sus respectivos sistemas de persistencia los cuales garantizan que, en caso de cerrarse, al volver a iniciar la aplicación ésta sea restaurada en el estado exacto en el cual se encontraba anteriormente. También se guarda un historial de acciones correspondiente a cada aplicación, el cual permite identificar aquellos paquetes enviados y

recibidos a lo largo de la ejecución, junto a demás hechos que pueden suceder a lo largo de la ejecución de cada aplicación y puede afectar en medida el desarrollo de la misma.

Message broker

Se dispone de un Message Broker (servidor) el cual se encarga de administrar las sesiones de las aplicaciones que intentan conectarse. Dispone de un servicio de autenticación el cual valida las aplicaciones teniendo en consideración sus ID, generando las sesiones correspondientes usando el servidor MQTT implementado. El broker, también en su parte del servidor MQTT, cuenta con una persistencia correspondiente, permitiendo almacenar aquellas sesiones que se han iniciado previamente asi como la cola de mensajes de cada sesión, lo cual permite que al iniciar de nuevo el servidor se restaure el mismo estado sobre el cual ha estado anteriormente.

Así como se ha mencionado anteriormente, esta aplicación tiene la funcionalidad de enviar los mensajes correspondientes hacia aquellos clientes suscritos al tópico correspondiente, los cuales pueden haberse desconectado en medio de la ejecución, garantizando así que reciban los mensajes correspondientes.

Más allá de esto, el broker cuenta con el archivo de configuración del protocolo, necesario para poder iniciar el servidor MQTT, con la información necesaria para establecer las conexiones con los clientes y manejar los mensajes correctamente. Además, cuenta con un archivo de autenticación el cual tiene aquellos IDs de aplicaciones que tienen permitido conectarse como clientes al broker.

Sistema de Cámaras

La aplicación del Sistema de Cámaras ofrece al usuario el manejo general de todas aquellas cámaras que se pueden activar a lo largo de la Ciudad de Buenos Aires. Esta aplicación cuenta con una interfaz dentro de la terminal, la cual ofrece numerosos comandos para poder realizar determinadas acciones, como agregar, eliminar o borrar cámaras, las cuales se verán reflejadas en la interfaz de la Aplicación de Monitoreo.

El funcionamiento de las cámaras ofrece diferentes modos en los que puede estar activo: *ahorro de batería, en alerta, desconectado*. Una vez se conecta el sistema al broker, las cámaras pasarán a estar en modo ahorro de batería (o puede que se pongan en modo alerta en caso de que se haya cerrado el sistema de cámaras con un incidente cerca a una de ellas).

El modo de las cámaras es determinado a la presencia de incidentes, donde cada cámara tiene un radio de identificación y de notificación a otras cámaras cercanas a su posición, de modo que al aparecer un incidente dentro del radio de visión de una cámara, esta se pondrá en modo alerta y notificara a aquellas cámaras cercanas del incidente que se ha producido. Finalmente, una vez el incidente se haya solucionado, tanto la cámara como aquellas activadas por su cercanía, pasarán a estar en modo ahorro de batería.

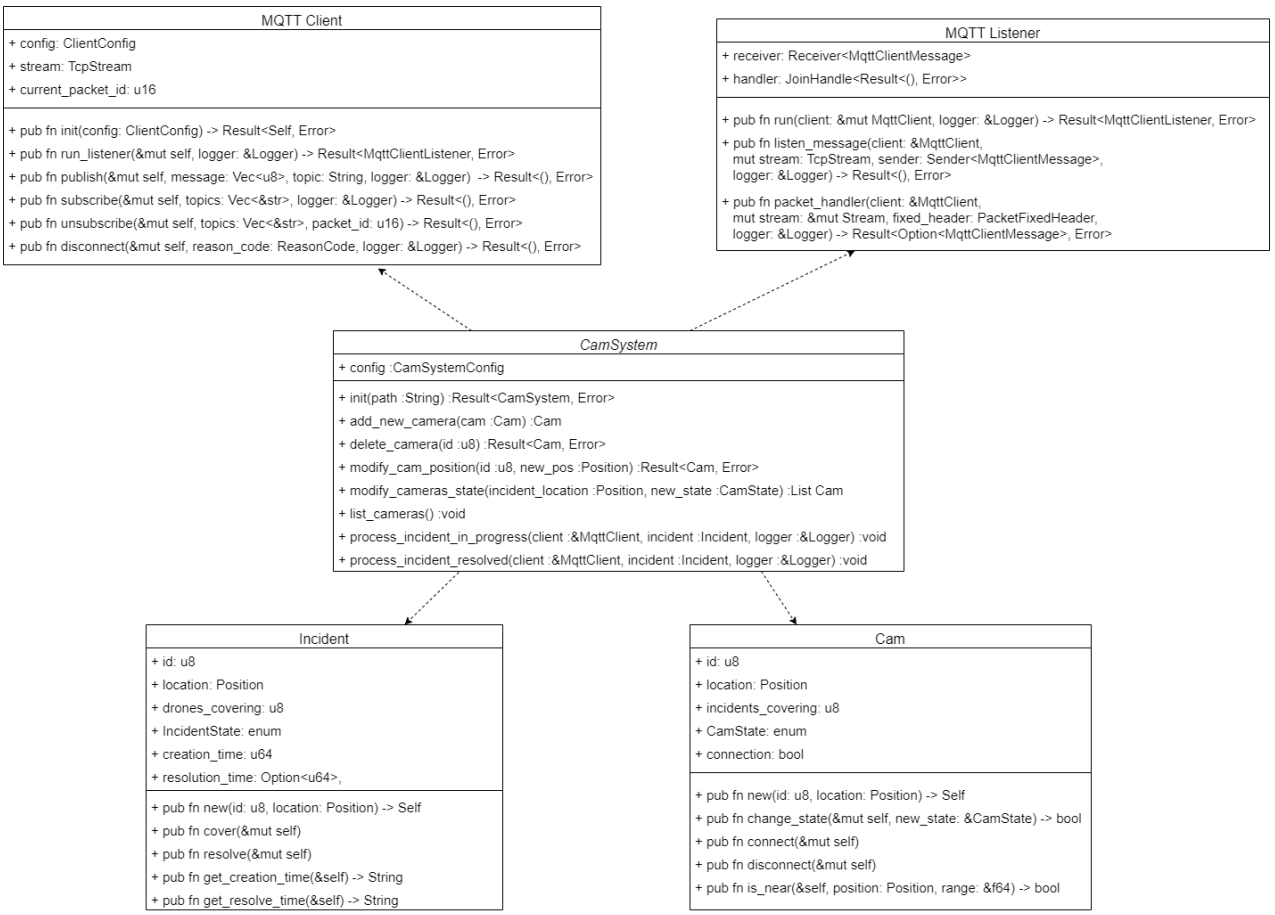

Fig 8. Diagrama simplificado del manejo de las clases dentro del sistema de cámaras

El sistema de cámaras utiliza las clases pertenecientes al directorio *shared,* el cual provee clases que se comparten entre las aplicaciones, como la clase *Incident,* la cual representa un incidente creado por la aplicación de monitoreo y se resuelve a través de los drones; o (justamente) la clase *Cam* representa una cámara, la cual reacciona ante un incidente y manejan su propia lógica de operación, y también le informan al sistema de cámaras sus cambios de estado.

De esta forma, y así como se representa en la Fig. 9, los incidentes publicados por otra aplicación (en este proyecto, la aplicación de monitoreo) son enviados desde el broker debido a la suscripción, y estos son capaces de modificar el estado de aquellas cámaras que componen el sistema.

Se detalla más información sobre el funcionamiento de las cámaras con los incidentes en la sección de la implementación del agregado final.

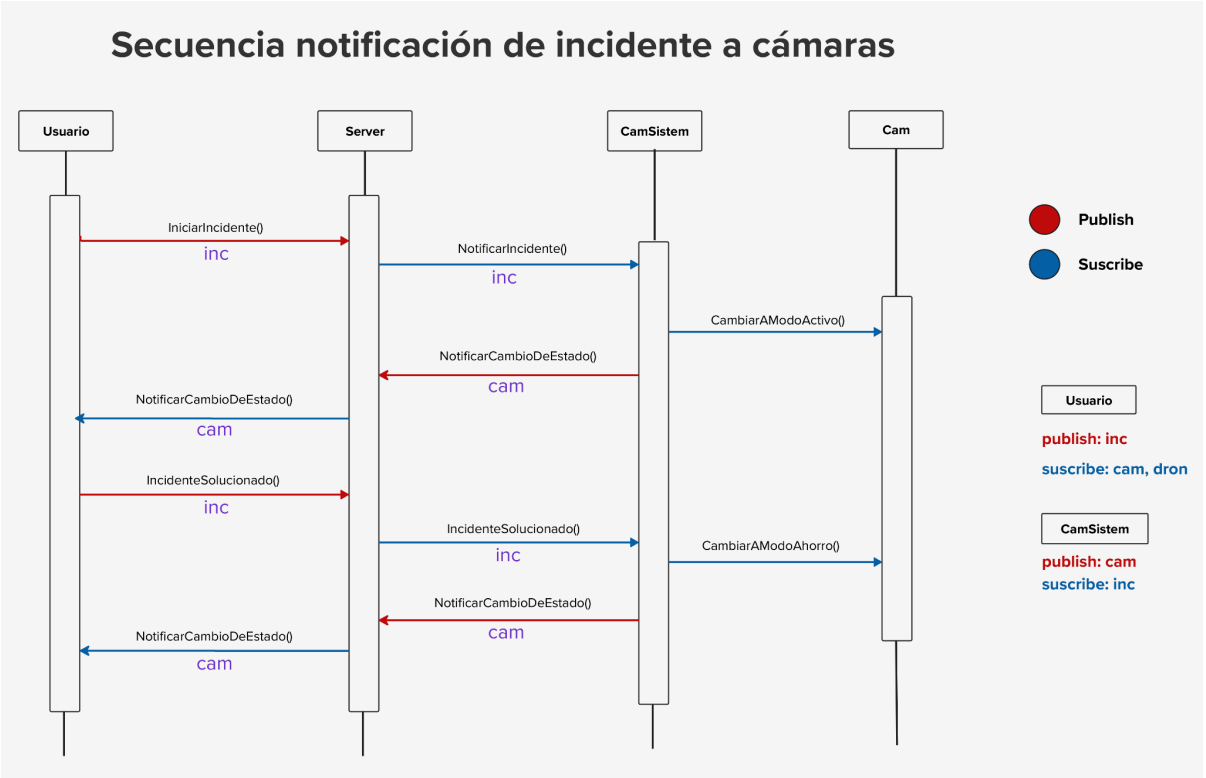

Fig 9. Diagrama de secuencia de notificación de un incidente a las cámaras

Aplicación de Monitoreo

La Aplicación de Monitoreo dispone una interfaz con un mapa que visualiza la Ciudad Autónoma de Buenos Aires, sobre la cual el usuario podrá informar un incidente y reconocer en una lista aquellos incidentes activos o ya resueltos. También podrá conocer el estado de los *drones* y de las *cámaras* que se encuentren activas o desconectadas a través de esta misma. Para implementar tanto la interfaz gráfica de esta aplicación como su propio mapa, se ha utilizado la interfaz gráfica *egui* junto al crate *walkers*, ideales para esta tarea por su interfaz apta para establecer la conexión con la aplicación, y las clases que involucra, con la capacidad de establecer controladores correspondientes.

Esta aplicación actúa en el medio del envío de mensajes de las demás aplicaciones, con el objetivo de poder levantar la interfaz con toda la información dentro y fuera del mapa, que corresponde a las demás aplicaciones. También, esta aplicación cuenta con un registro de los incidentes cargados y la información sobre estos, como también de un "generador de incidentes" dentro de la interfaz para facilitar el uso de las demás aplicaciones en presencia de un incidente, generando uno en una posición geográfica determinada.

Para poder recopilar la información de manera eficiente, está suscrita a todos los tópicos disponibles, de modo de poder recibir la totalidad de mensajes entre las demás aplicaciones para poder visualizarlo. Para cumplir esa tarea, mediante el controlador y la vista correspondiente de la interfaz, utiliza aquellas clases compartidas dentro del directorio *shared*.

Por último, esta aplicación también cuenta con su propia persistencia, guardando el estado de toda la interfaz en el estado previo a su cierre para poder reiniciarlo correctamente, teniendo exactamente la misma información que mostraba antes de cerrarse.

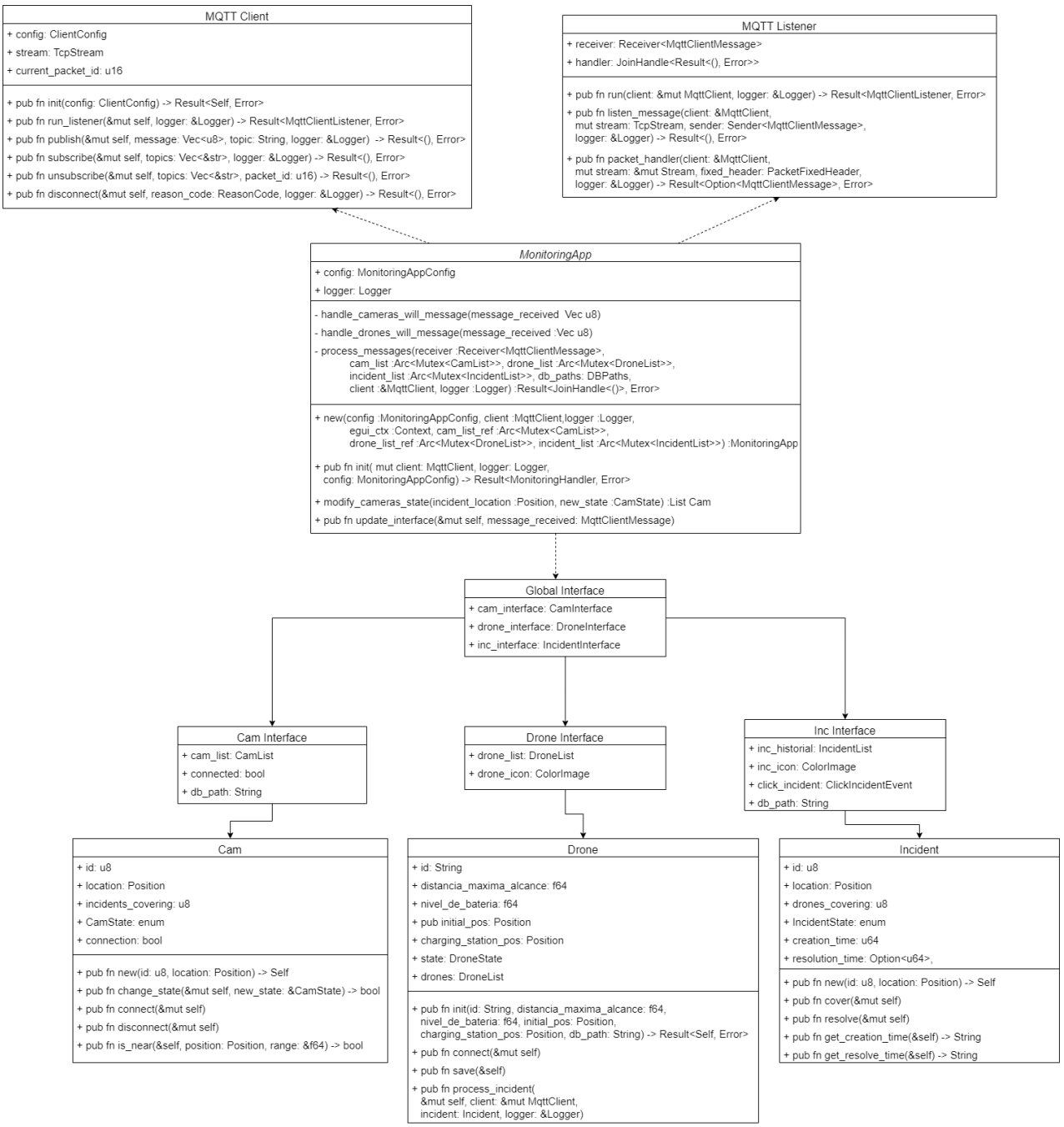

Fig 10. Representación de la interacción entre la Aplicación de Monitoreo y las demás aplicaciones

Aplicaciones de Drones

El Software de Control del Agente que se provee es una aplicación drone la cual, en base a sus configuraciones presentes dentro de su propio archivo de configuración, como dentro del archivo de configuración del protocolo (con su ID correspondiente) permite iniciar una instancia de la aplicación de drone con determinadas características. Estas características que poseen incluyen la posición inicial de los drones, la posición de su central para carga, radio del alcance para cubrir incidentes, y la capacidad de batería.

El funcionamiento del drone es determinado por el estado en el que se encuentra: activo, en camino a un incidente, resolviendo incidente y en camino a la posición original. Estos estados son determinados por la presencia de incidentes activos dentro de su radio de visualización. Si el drone está en estado activo ante la aparición de un nuevo incidente, este acudirá a su posición y, una vez allí, esperará por otro drone para poder cubrir aquel incidente, ya que es requisito que dos drones estén cubriéndolo para poder resolverlo. Una vez resuelto, regresará a su posición original y seguirá activo.

Para determinar si un drone se debe dirigir a cubrir un incidente o no, cada drone posee en su ejecución una lista con los estados y las posiciones de cada uno de los agentes disponibles, de modo que al aparecer un incidente depende del cálculo que hace cada drone para identificar si se debe dirigir hacia este nuevo incidente, o existen al menos dos agentes más cercanos que pueden cubrirlo eficazmente. Debido a esta implementación, el cálculo para determinar el estado de un drone depende únicamente del mismo drone. Y, en caso de haber ido a cargar batería o cubrir otro incidente previamente, este drone será capaz de acudir para cubrir el nuevo incidente una vez regrese a su posición.

Además de aquellos estados previamente mencionados, los drones también poseen de una batería que es manejada en otro thread aparte del principal que conlleva la ejecución principal del programa. Esta batería disminuye cada un determinado periodo de tiempo, y una vez llega a un mínimo previamente configurado, este agente cambia su estado en dirección a la central de carga para que, una vez allí, comienza la recarga de su batería. Una vez ésta llegue al 100%, el drone regresará a su ubicación definida.

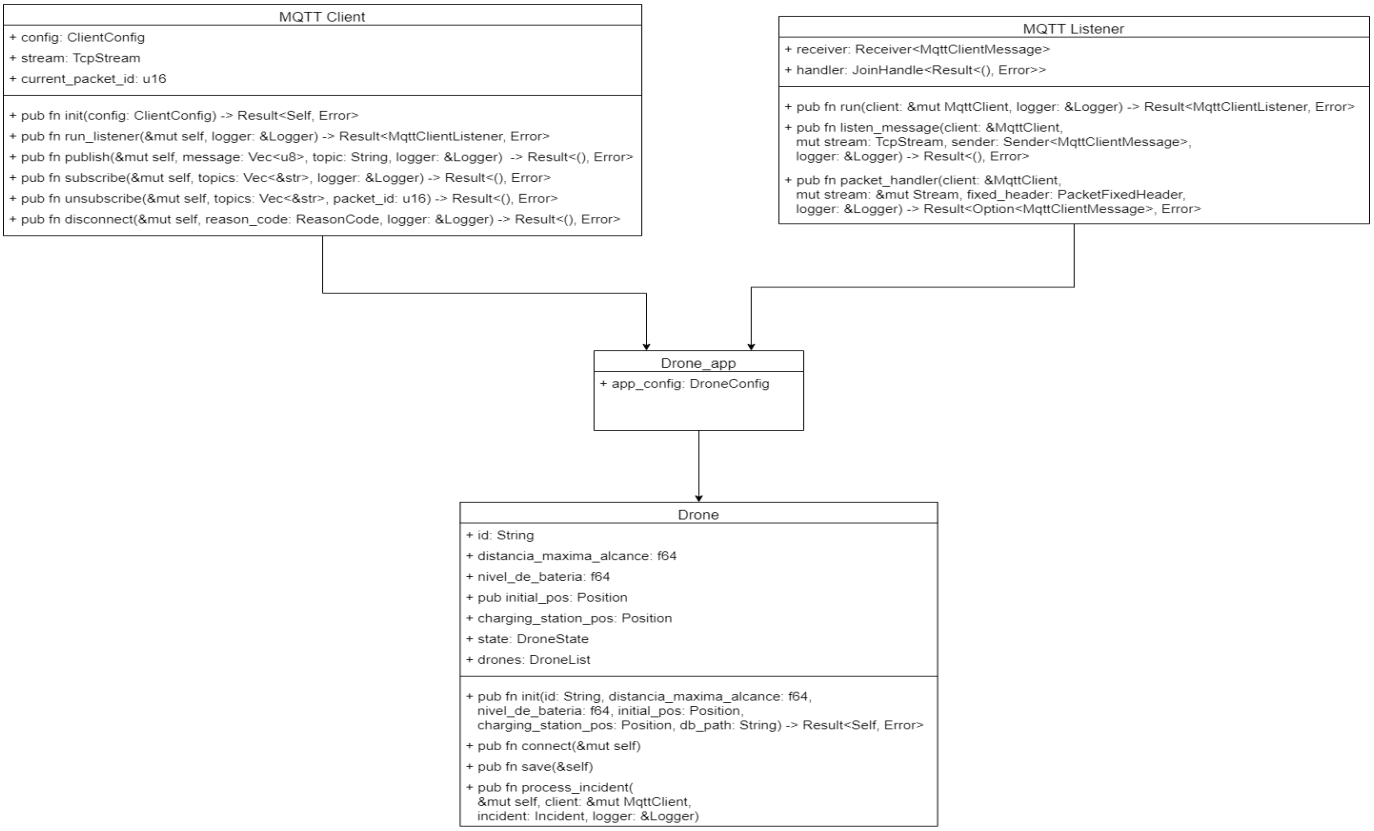

Fig 11.Diagrama de clases de la aplicación de un drone

Por último, también existe la posibilidad de ingresar un comando por la consola, para poder desconectar un drone y notificarlo inmediatamente al broker y a las demás aplicaciones (incluidos otros drones) de su detención en su ejecución. Asimismo, en la terminal de cada instancia de drone correspondiente, se imprimen los estados sobre los cuales el drone es notificado, aunque estos cambios de estado son siempre visibles desde la interfaz de la aplicación de monitoreo.

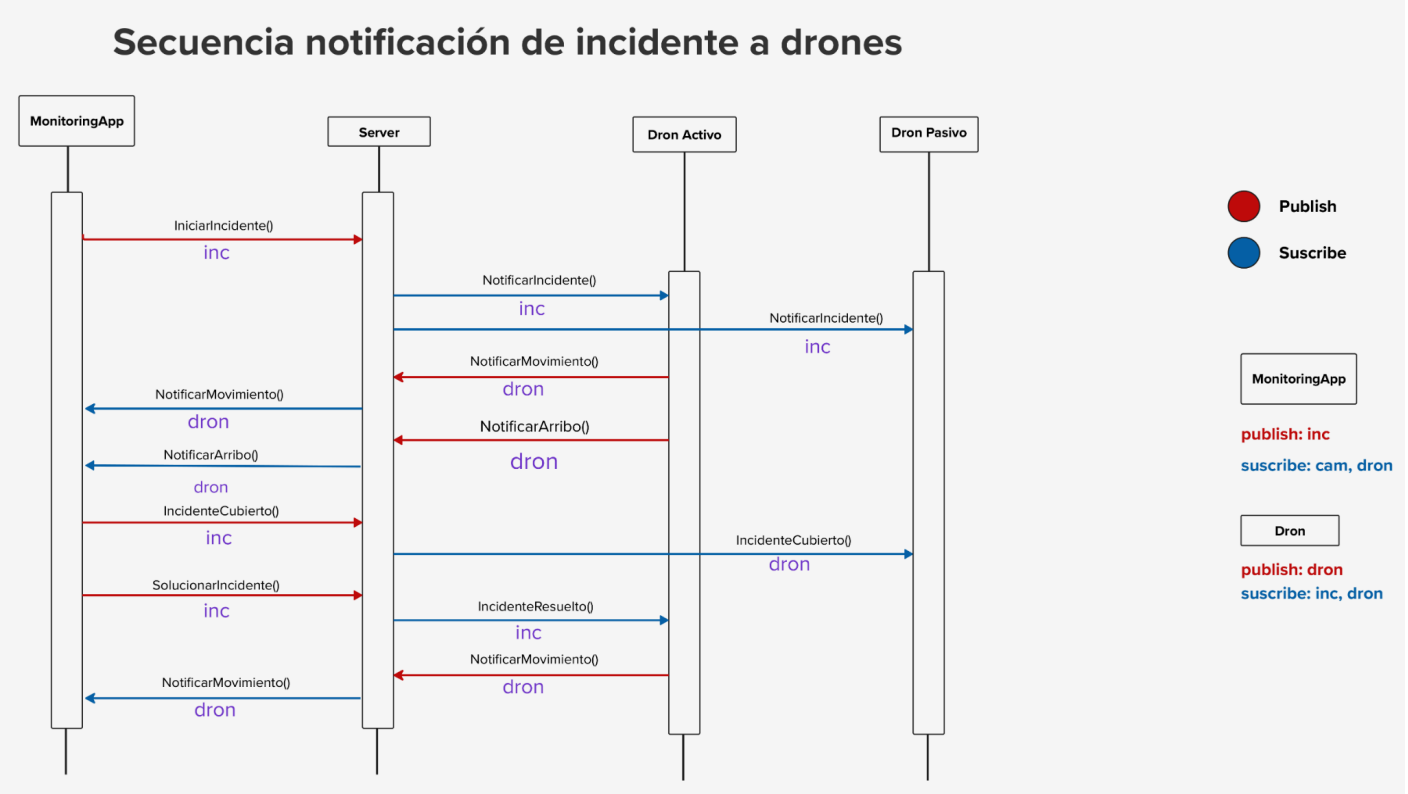

Fig 12. Diagrama de secuencia de notificacion de un incidente a instancias de drones

El diagrama de la Fig. 12 se representa la idea de operatoria de la notificación de un incidente a alto nivel, donde el usuario es quien crea un incidente a través de la interfaz de la aplicación de monitoreo, y esta le informa al broker para que replique el mensaje en los tópicos, informando en este caso a los drones, representados por los dos estados principales:

- En estado activo, y siendo capaz de cubrir el incidente, un dron le informa al servidor que acudirá al incidente.
- En estado pasivo, (o en estado activo, no pudiendo acudir el incidente), el dron solo recibe las notificaciones y no cambia su estado.

Siguiendo el funcionamiento del proyecto y todas las aplicaciones, en primer lugar un usuario crea un incidente mediante la aplicación de monitoreo, la cual publica el incidente en el tópico correspondiente para que el broker lo replique y así los drones, que están suscritos al tópico de incidente, reciben la notificación. Luego, 2 drones cercanos capaces de asistir el incidente publican en el tópico de drone su estado en movimiento para asistir el incidente, el cual el broker replica el mensaje y envía el mensaje a la aplicación de monitoreo. Una vez llegado este punto, los mismos drones publican nuevamente en el mismo tópico el arribo al lugar del incidente, la aplicación de monitoreo recibe las 2 notificaciones y publica en el tópico de incidentes la resolución de este mismo, para lo cual el broker entrega el mensaje en el tópico de incidentes para

que los drones reciben la actualización y publiquen en sus tópicos sus nuevos estados en movimiento para volver a su posición inicial.

Implementación del Agregado Final

Requerimientos del final

En este agregado final se le añade la nueva funcionalidad al sistema de cámaras, siendo capaz de procesar imágenes generadas por cámaras en modo ahorro de energía, utilizando una tecnología de reconocimiento de imágenes para detectar posibles incidentes. En caso de detectar uno, el sistema de cámaras utiliza el servicio de mensajería implementado para notificar al resto de las aplicaciones del nuevo incidente presente y así lograr movilizar todo el circuito de resolución de incidentes implementado.

Para simular esta generación de incidentes, cada cámara tiene un directorio específico sobre el cual se van copiando imágenes las cuales son procesadas por la aplicación inmediatamente, monitoreando constantemente sobre la presencia de nuevas imágenes dentro del incidente. El directorio de cada cámara se encuentra dentro de un directorio perteneciente al sistema de cámaras, el cual identifica a qué cámara le corresponde la nueva imagen cargada en base a su directorio.

Elección del proveedor externo de AI

En el proceso de selección del proveedor externo de inteligencia artificial para la incorporación de tecnología de reconocimiento de imágenes al sistema central de cámaras se han evaluado varias opciones de infraestructura en la nube, considerando a:

- Azure AI Vision
- Google Vision AI
- Amazon Rekognition Image

Si bien existen numerosos más proveedores, se tiene en cuenta como característica primordial que ofrezca una capa de uso gratuito (free tier) suficiente para probar el trabajo realizado con una carga de procesamiento mediana, como mínimo de 10 requests por minuto.

Tras un análisis del servicio que ofrece cada proveedor, se ha decidido utilizar **Azure AI Vision**. Esta decisión se fundamenta en varios factores clave como el modelo "Extract common tags from images" que permite identificar fácilmente el reconocimiento que realiza la aplicación, y se ajusta a las necesidades específicas de este agregado. Esto sumado a su facilidad y sencillez para generar requests como la alta fiabilidad para poder diferenciar un incidente de una situación normal, convirtió a este modelo como el más óptimo para cumplir con los requisitos establecidos.

Análisis del costo de funcionamiento

Basándonos en un mínimo de 10 cargas de incidentes por minuto, asegurando el correcto funcionamiento del programa, se estima aproximadamente 432.000 transacciones por mes, considerando un funcionamiento continuo de 24/7 con el sistema de cámaras abierto y revisando posibles incidentes en todo momento.

En base a la información de costos del proveedor **Microsoft Azure AI Vision**, para cumplir con la anterior demanda de recursos, se debería adquirir el *Standard (S1) - Web/Container Grupo 1,* el cual ofrece hasta un millón de transacciones por mes, cobrando un dólar por cada mil transacciones.

Teniendo en cuenta los precios de este servicio, y localizando en una ubicación lo más cercana posible a Argentina: Sur de Brasil, estima que, para poder cumplir con nuestra cantidad de solicitudes se debe pagar 432 US$ por mes, para asegurar un correcto minimo funcionamiento del sistema de cámaras cargando hasta 10 incidentes por minuto.

Uso de Microsoft Azure AI Vision

Como se mencionó en el apartado anterior, decidimos utilizar el servicio provisto por Microsoft, específicamente el modelo llamado "*Extract common tags from images*". El mismo, extrae etiquetas basadas en miles de objetos reconocibles, seres vivos, paisajes y acciones. Por lo que provee la posibilidad de identificar y etiquetar elementos comunes en imágenes.

El modelo funciona con base en los siguientes elementos:

1. **API de Azure:** Microsoft Azure ofrece una API llamada *Analyze Image* que se utiliza para interactuar con el modelo.. Esta API es el punto de entrada para enviar solicitudes y recibir respuestas del modelo.
2. **Envío de Solicitudes:**
    - URL de la API: La solicitud se envía a una URL específica proporcionada por Azure.
    - Método HTTP: La solicitud se realiza mediante el método POST.
    - Autenticación: Para autenticar la solicitud, se requiere una clave de suscripción de Azure.
3. **Cuerpo de la Solicitud:**
    - Imagen: La imagen puede ser enviada como una URL (si está disponible en línea) o como un archivo binario en el cuerpo de la solicitud.
    - Parámetros de Análisis: Puedes especificar los detalles de lo que deseas analizar. Para el modelo de extracción de etiquetas comunes, debes incluir el parámetro *visualFeatures* con el valor *Tags*.

4. **Respuesta de la API:** La respuesta de la API es un objeto JSON que contiene la siguiente información:
    - Tags: El análisis de imágenes puede devolver etiquetas de contenido para miles de objetos reconocibles. La etiquetación no se limita solo al sujeto principal, como una persona en primer plano, sino que también incluye el entorno (interior o exterior), muebles, herramientas, plantas, animales, accesorios, gadgets, entre otros. Cuando las etiquetas son ambiguas o no son de conocimiento común, la respuesta de la API proporciona pistas para aclarar el significado de la etiqueta en el contexto de un entorno conocido.

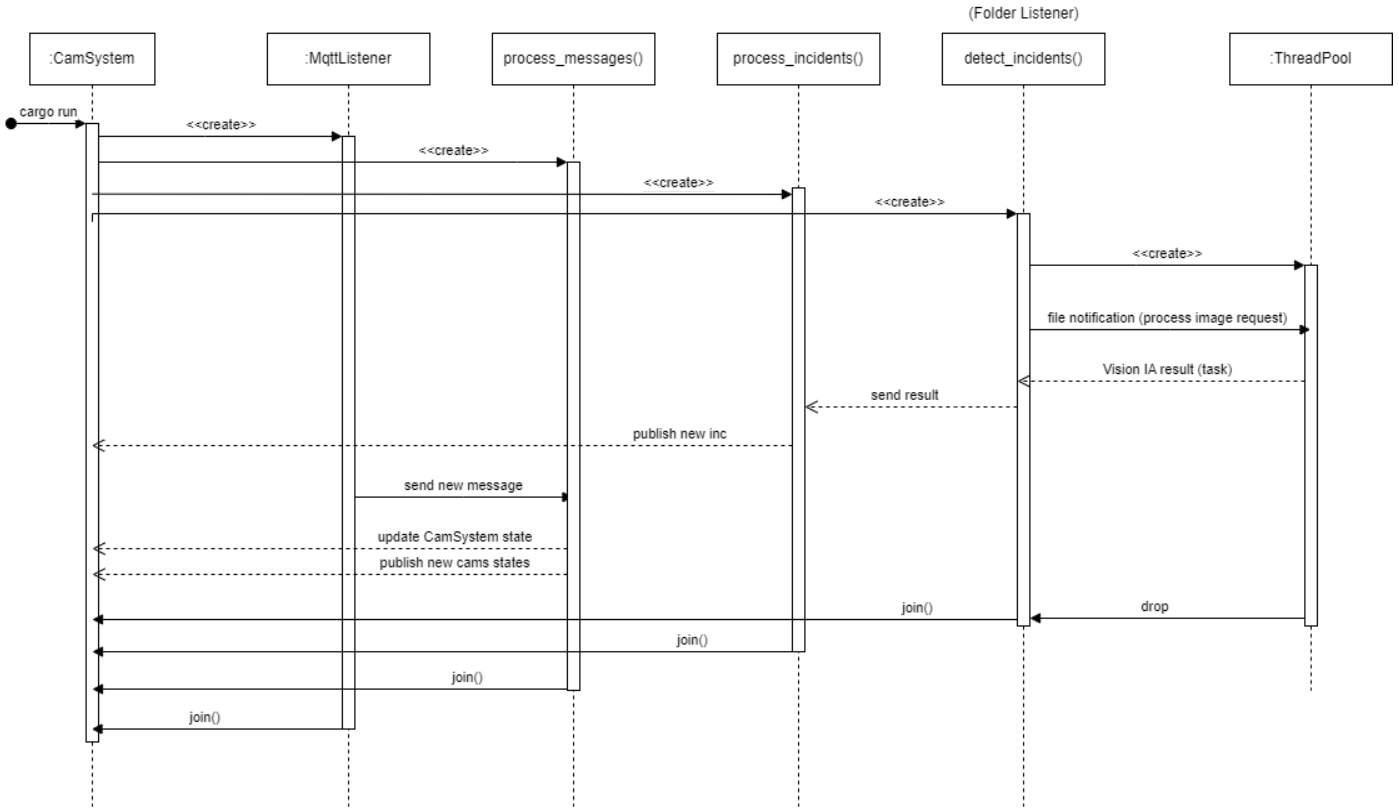

Fig 13. Diagrama de secuencia correspondiente al manejo de threads del Sistema de Cámaras, incluyendo aquellos pertenecientes al agregado final, junto al procesamiento de imágenes con un Threadpool

En este caso, el threadpool funciona como una herramienta para poder ejecutar en simultáneo varias request a la API de Microsoft Azure AI Vision. Primeramente se corre en un hilo el listener de nuevos cambios en las carpetas de las cámaras (*detect_incidents*), una vez detecta un nuevo archivo para procesar, ejecuta en el threadpool una request con dicho archivo. Recordemos que dentro del threadpool hay varios workers threads que se encargan de ejecutar las tareas que se le asignan al threadpool. Una vez ejecutada esa tarea, el worker thread se libera para volver a tomar otra request. Luego de esto, mediante un sender, *detect_incidents* envía el resultado de la request al hilo *process_incidents* que se encarga de mandar el nuevo incidente si hace falta.

De esta forma, se puede garantizar un correcto funcionamiento de las requests, utilizando recursos de manera óptima sin sobrecargar el uso del CPU.

Nueva funcionalidad del Sistema de Cámaras

Habiendo implementado correctamente el modelo de reconocimiento de imágenes, se le ha entregado una nueva responsabilidad al sistema de cámaras: la capacidad de poder publicar incidentes, que anteriormente era exclusiva de la aplicación de monitoreo.
De esta forma, una vez que una cámara dentro del sistema detecta una imagen en su directorio asignado, y confirmando que se trata de un nuevo incidente, está publicará en el tópico correspondiente para notificarles tanto a la aplicación de monitoreo como a las instancias de agentes autónomos de este nuevo incidente que ha aparecido, pudiendo asi movilizar al sistema completo para resolver aquella aparición.

En la generación de este nuevo incidente, la posición del mismo es calculada en base a un número aleatorio sobre la posición de la cámara, estableciendo el nuevo incidente dentro de su radio de detección y notificando así a aquellas cámaras que se encuentren en cercanía. Por otro lado, en caso de que no se haya considerado aquella imagen como un incidente, simplemente ignora el nuevo potencial incidente y mantiene el estado anterior de la cámara.

Conclusiones

Una vez hecho funcional este proyecto, ha servido como experiencia para identificar el uso del protocolo MQTT y su aplicación en redes IoT, permitiendo así una posible implementación real con hardware correspondiente.

El lenguaje Rust, al ser más nuevo a comparación de otros más populares, exige su tiempo para no solo aprender la sintaxis sino también las buenas prácticas y el correcto manejo de la ejecución de un programa, utilizando las diversas opciones que ofrece para poder compilar un proyecto de esta magnitud sin ningún problema.

Con respecto a la implementación del proyecto en sí, haber definido una interfaz efectiva para la implementación de la base del protocolo permite un rápido diseño en los componentes de la interfaz del protocolo. El polimorfismo aplicado con los diferentes packets y las numerosas clases que componen todo MQTT permite un código ordenado y bien establecido, y fácil de usar tanto en el cliente como en el servidor del protocolo.

De esta forma, pudiendo asegurar la correcta implementación de aquellos módulos correspondientes, el testing está presente en su medida dentro del proyecto de forma general, alcanzando un total de 50% de **coverage** entre todas las aplicaciones y nuestra implementación del protocolo, garantizando la presencia de tests en las secciones más críticas: la serialización de todos los mensajes, la comunicación eficiente entre un cliente y un servidor, y un correcto funcionamiento del modelo de IA reconociendo correctamente las imágenes.

Habiendo considerado todo, este proyecto ciertamente nos enseño numerosas cantidades de funcionalidades las cuales no habíamos puesto en práctica previamente en la carrera, como también un correcto trabajo en equipo para dividir las tareas y poder trabajar tanto en conjunto como en paralelo, maximizando la eficiencia de nuestro trabajo.

Bibliografía

MQTT:

- https://github.com/bytebeamio/rumqtt/tree/main/rumqttc
- https://github.com/bytebeamio/rumqtt/tree/main/rumqttd
- https://docs.rs/rumqttc/latest/rumqttc/
- https://github.com/taller-1-fiuba-rust/taller-1-fiuba-rust.github.io/blob/proyectos-realizados/src/proyecto/anteriores/21C2/proyecto.md
- https://www.hivemq.com/mqtt/
- **https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf**

QoS 1:

- https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/
- https://www.emqx.com/en/blog/introduction-to-mqtt-qos

AMQP:

- https://www.amqp.org/product/overview
- https://www.youtube.com/watch?v=pKnbYGrDAKY
- https://www.youtube.com/watch?v=ODpeIdUdClc&list=PLmE4bZU0qx-wAP02i0I7PJWvDWoCytEjD
- https://www.youtube.com/watch?v=oNnZAsN1BCU

Arquitectura del Modelo de IA:

- https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber
- Vision Studio (azure.com)
- Pricing - Computer Vision API | Microsoft Azure