Informe taller de programación (75.42)

Brzoza Valeria (107523) y Cichero Tomás (107973)

Protocolos de comunicación

Introducción

Los protocolos de publicación y suscripción (pub/sub) son un modelo de comunicación en sistemas distribuidos que permite a los publicadores enviar mensajes a un canal o tópico, sin conocer a los suscriptores que recibirán dichos mensajes. Los suscriptores se registran en estos canales para recibir la información relevante.

El modelo pub/sub es esencial en aplicaciones donde la escalabilidad y la respuesta en tiempo real son cruciales, como en mensajería, IoT, redes sociales y servicios financieros. Ejemplos de sistemas que utilizan este modelo incluye MQTT, STOMP, Apache Kafka y Google Cloud Pub/Sub.

Características generales

Además de la posibilidad de publicar y suscribirse a tópicos o canales. Existen varias características extra que pueden tener los diferentes protocolos de comunicación. Algunas de las cuales son requisitos necesarios para la realización de este proyecto.

Jerarquía de tópicos:

Nos permite tener una organización del flujo de las publicaciones generadas más eficiente. Los que esto significa es que por cada tópico es posible definir un sub tópico o filtro que recibe una parte solamente de los mensajes enviados al tópico de mayor jerarquía

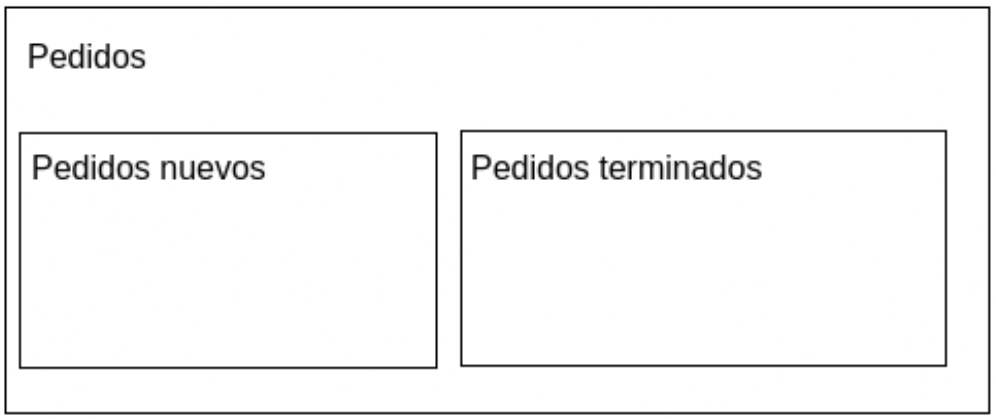


Gráfico 1: *Representación de un tópico "Pedidos" que engloba a los subtópicos "Pedidos nuevos" y "Pedidos terminados"*

"Quality of Service" (QoS)

Esta característica representa con que nivel se asegura el sistema que los mensajes están siendo recibidos y consumidos efectivamente y no se está perdiendo información en el proceso. En general se recomienda que al menos se utilice un nivel de QoS 1 lo que significa que los mensajes se entregan al menos una vez sin posibilidad de que se salten mensajes.

Otra forma en la que se refiere al QoS es el "acknowledgment" o "ACK". Esto representa la confirmación de envío o recepción de un mensaje, implicando al menos un QoS de nivel 1.

Retención de Mensajes

Esta característica permite que los servidores o "brokers" guarden mensajes en memoria o almacenamiento del mismo hasta que un cliente se conecte o desconecte. Esto implica que no es necesario que el cliente esté conectado o depende el caso suscripto a un tópico para poder igualmente recibir información del mismo en el futuro.

Esto es indispensable para casos donde es necesario que un cliente o consumidor reciba mensajes incluso cuando tenga períodos de desconexión. Esta característica está asociada directamente con el QoS (depende del protocolo).

Request/Response

El paradigma request response es muy utilizado en diferentes situaciones. El caso más común es en la web, la cual utiliza el protocolo HTTP el cual es un protocolo de request response. Esta metodología de comunicación es muy útil cuando se espera una respuesta a un mensaje o por ejemplo un comando o una acción.

No es una característica que tenga que ser parte necesariamente de un protocolo pub/sub pero es un buen agregado para tener.

Elección de Protocolo de Comunicación

Para la realización del proyecto es necesario elegir un protocolo de comunicación. Este debe ser posible de implementar en tiempo y costo razonable. Debe cumplir al menos con las características de retención de mensajes y entrega garantizada de mensajes (QoS 1). A continuación explicaremos las características de diferentes opciones propuestas.

MQTT

MQTT es uno de los protocolos más conocidos y utilizados por la industria. Es un protocolo que existe desde el año 1999 creado inicialmente para la industria del petróleo y gas. Es un protocolo binario, es muy sencillo y requiere mínimos recursos para ser utilizado. Es probablemente el protocolo más utilizado para IoT.

Tiene la ventaja de ser uno de los protocolos más usados y entendidos del sector. Existen montones de opciones comerciales y open source del mismo y de servidores o "brokers" para este. Dependiendo de la implementación del broker, puede soportar todas las características mencionadas anteriormente.

De todas formas, tiene algunas desventajas. No hay una documentación centralizada sencilla. MQTT tiene muchas versiones, variantes e implementaciones. Haciendo que existan muchas variantes, documentación incompleta o difícil de entender para poder empezar de cero a implementar el protocolo. Adicionalmente no hay un estándar sobre cómo tienen que funcionar ciertas características en un broker de MQTT. Algunas implementaciones pueden incluso ser incompatibles entre sí (aunque no tiene por qué ser el caso).

Dicho esto, podemos decir que más allá de sus problemas, es una opción excelente que ya está probada en el mundo real que funciona y cumple con las necesidades del proyecto.

STOMP

El protocolo STOMP soporta el uso de comandos para enviar y suscribirse a tópicos. Soporta acknowledgment para recibir mensajes y confirmación de envío de comandos. Está basado en texto y es extremadamente sencillo. No soporta jerarquías de tópicos ni request response. Tampoco tiene nada especificado sobre retención de mensajes.

En general es una opción que podría ser suficiente para el alcance de este proyecto pero no es ideal.

Apache Kafka

Kafka es una opción altamente utilizada en la industria del software para el manejo de eventos y colas en tiempo real. No es un protocolo que esté diseñado principalmente para IoT y no es recomendado para este tipo de uso. Kafka es más complejo que otros protocolos además de no ser especialmente eficiente. Entre otras cosas Kafka no soporta jerarquías de tópicos ni request/response.

En general podemos decir que se desvía de lo que estamos buscando para este proyecto.

Google Pub/Sub

Google Pub/Sub no es open source. Esto descarta automáticamente la elección. Sería técnicamente posible implementar una copia compatible del mismo pero preferimos utilizar un protocolo open source. Además de esto, Google Pub/Sub no es un protocolo especialmente diseñado para IoT.

AMQP

Este protocolo soporta la mayoría de las características mencionadas a excepción de la jerarquía de tópicos. La complejidad de este protocolo es mayor que otros más sencillos como MQTT o STOMP. Tiene algunas características como "bindings", "exchanges" complejas que se van del alcance de este proyecto.

En base a esta información, podemos decir que posiblemente no sea la mejor opción para implementar y utilizar en este proyecto.

NATS

Es un protocolo relativamente nuevo y moderno. Está basado en texto (aunque soporta el envió de datos binarios). La implementación de NATS Core es muy sencilla. NATS permite la jerarquía de tópicos y request/response. Sumado a esto, el protocolo está excelentemente documentado y explicado en su sitio oficial. Además está detallado la implementación de un broker o servidor del mismo.

El mayor inconveniente del mismo es que la versión core del mismo no soporta QoS ni retención de mensajes. Para sumar estas características es necesario incluir una función extra de NATS que funciona sobre su versión core llamada JetStream que incluye una serie de características que cumplen todos los requerimientos.

JetStream se implementa sobre el propio servidor de NATS pero utiliza las primitivas de comunicación propias de la versión core. Una gran ventaja del mismo es que está altamente documentado su funcionamiento e implementación.

De todas formas hay que destacar que implementar el 100% de las características de NATS + JetStream no es posible dentro del alcance del proyecto. Dicho esto NATS es probablemente una gran opción si se utiliza solo con las funcionalidades mínimas de la capa de JetStream para cumplir los requisitos. Debido a su relativa simplicidad e información necesaria disponible para simplificar su desarrollo.

Otros

Existen otros protocolos de pub sub. Entre otros tenemos Sockets IO, DDS y Apache Pulsar. Para nuestro análisis se consideraron solo las opciones explicadas anteriormente y se dejaron afuera muchas otras que no consideramos adecuadas para esta búsqueda.

Decisión de Protocolo

Luego del análisis de las diferentes alternativas nos decidimos por implementar el protocolo NATS Core + Parcialmente la capa de JetStream. El motivo de esta decisión se debe principalmente a la simplicidad de NATS core y el gran nivel de documentación del mismo.

La decisión de implementar una parte de JetStream fue la más complicada, porque si bien también está claramente documentado su funcionamiento, su nivel de complejidad si es mayor que el acknowledgment y la capa de retención de otros protocolos. De todas formas, teniendo una primera implementación del core del protocolo, simplifica bastante la implementación de JetStream, haciéndola posible en tiempo y costo razonable. Sumado a esto, debido al gran alcance de JetStream, es posible agregar características en el futuro conforme a nuevos requerimientos del proyecto.

Un detalle más, también elegimos NATS porque nos pareció una opción más interesante que MQTT (siendo este la segunda opción), ya que NATS no está tan generalizado y utilizado públicamente.

Implementación del servidor de NATS

Nats Core

Implementamos una versión funcional del protocolo NATS. Su realización fue en el lenguaje Rust utilizando la menor cantidad de librerías externas. Tampoco se utilizó ninguna función de rust async. Esta versión es compatible con otras implementaciones del protocolo. Fue probada su compatibilidad utilizando el CLI de nats.

Actualmente soporta la mayoría de las características de NATS core a excepción de algunas opciones de configuración. Tampoco se soporta el parámetro de desuscripción *max_msgs*.

Arquitectura multi-thread

El servidor utiliza una arquitectura conocida como “thread pool”. Esto significa que cuando se inicia el servidor se crean una cantidad fija de hilos de ejecución (threads) y el trabajo del servicio se distribuye en estos. La cantidad de hilos por defecto es equivalente a la cantidad de núcleos del procesador pero es posible configurarlo a mano.

La manera en la que el servidor distribuye el trabajo entre los diferentes hilos es creando grupos de conexiones y cada grupo es gestionado por uno de los hilos. Cuando se crea una conexión nueva, esta es enviada a uno de los hilos.

Adicionalmente hay un hilo el cual se encarga de recibir las conexiones entrantes y enviarlas a través de canales al hilo correspondiente.

Comunicación Entre Hilos del “Pool”

Cada hilo puede comunicarse con el resto de los hilos del pool. Esto lo hace utilizando canales de la librería estándar de rust. La información que se envían entre los hilos son las suscripciones que se genera, las desuscripciones y las publicaciones.

Suscripciones

Cuando un cliente dentro de un hilo se suscribe a un tópico, el hilo le informa a los demás sobre la creación de esa nueva suscripción. Esto es necesario para que cuando se publique un mensaje, el hilo correspondiente sepa a cuales otros distribuirlo. Todos los hilos conocen en todo momento todas las suscripciones. Estos mantienen una copia de un objeto “Suscripciones” el cual permite acceder rápidamente a los hilos y clientes interesados en un tópico.

Uso de Comunicación “Non Blocking”

Para la implementación del servidor, decidimos utilizar el TcpListener y los TcpStream en modo no bloqueante o “nonblocking”. Esto significa que el programa no se detiene a esperar información nueva desde la red. Esto nos permite que un solo hilo pueda

leer y escribir a múltiples streams. La manera de hacer esto es comprobando constantemente si hay nuevos datos recibidos en cada stream.

Utilizar esta metodología presenta varios desafíos, entre ellos parsear un mensaje tiene que poder hacerse de manera parcial y resumible. Es decir, que el parseador de mensajes tiene que poder mantener hasta qué punto del mensaje entrante analizó y poder continuar desde ese punto más adelante cuando se reciben nuevos bytes de manera eficiente.

Recepción y Distribución de Mensajes

Cuando una conexión recibe un mensaje publicado por un cliente, esta le informa al hilo, el cual conociendo las suscripciones existentes, decide reenviar ese mensaje a los otros hilos que corresponda.

Además, en el caso de enviar a un “queue group”, utiliza una suscripción aleatoria del grupo para publicar el mensaje.

Implementación de JetStream

Nuestra implementación soporta algunas características de JetStream. Como mencionamos anteriormente, JetStream es una capa extra implementada utilizando el propio protocolo de NATS la cual agrega características como la persistencia y el QoS.

Las funciones principales de JetStream son, los *streams* y los *consumers*. Un *stream* recibe mensajes de uno o varios tópicos. Estos pueden ser configurados con diferentes reglas de persistencia y límites. Los *consumers* leen publicaciones de los streams. Un *consumer* consume publicaciones de un *stream*. Un *stream* puede tener varios consumers pero un consumer solo puede consumir de un *stream*. Nuestra implementación es bastante limitada. Por el momento solo soporta un mínimo de características de JetStream, aún así compatibles con las herramientas oficiales.

Actualmente soportamos estas operaciones:

- Crear stream
- Ver info de stream
- Listar streams
- Eliminar stream
- Crear consumer
- Ver info de consumer
- Listar consumers
- Eliminar consumers
- Consumir mensaje siguiente de consumer y hacer *acknowledgment*

Decisiones de Implementación de JetStream

Decidimos implementar solamente una mínima parte de las características de JetStream dado que abarcar su totalidad quedaba completamente fuera del alcance del

proyecto. Actualmente no soportamos guardar en el almacenamiento de la computadora los mensajes del stream. Tampoco soportamos consumir varios mensajes a la vez, actualizar streams y consumers, extraer mensajes del stream directo sin usar consumer, reglas de retención y límites, etc…

Arquitectura de JetStream

Debido a que JetStream utiliza el protocolo de NATS para su funcionamiento en vez de extenderlo, es posible tratar a los diferentes elementos de JetStream como clientes o concretamente conexiones del servidor. Actualmente tratamos a cada cliente del servidor como una conexión dentro del servidor, para la implementación de JetStream hicimos exactamente lo mismo. Tanto la api de administración de JetStream que permite crear y listar streams, como cada stream y cada consumer, implementan la misma interfaz que la conexión de un cliente. Esto nos permite aprovechar la arquitectura multithread del servidor y simplificar la implementación del mismo. Cada stream y consumer se suscribe a los tópicos relevantes para el mismo y el propio servidor base se encarga de la distribución de los mensajes. Adicionalmente los streams utilizan canales para comunicarse con sus consumers.

Autenticación y Cifrado

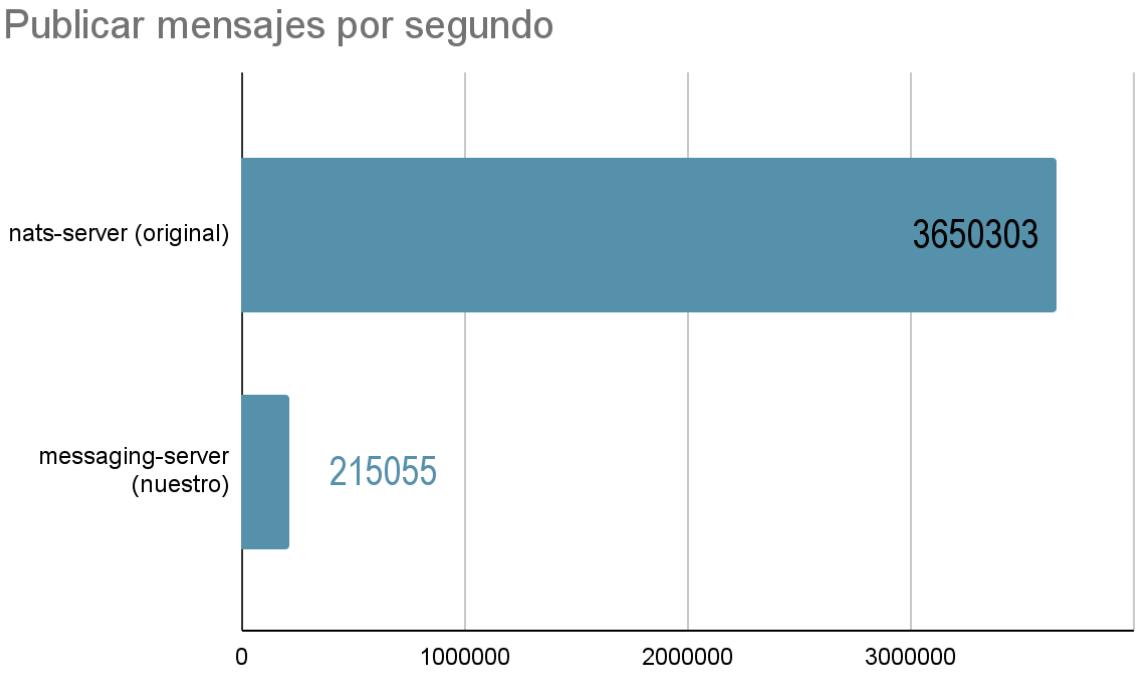
El servidor tiene soporte para usuarios y contraseñas. Se le puede pasar el parámetro *cuentas* el cual debe contener la ruta a un archivo *csv* con los usuario y contraseñas hasheadas con SHA256. El formato es *id,nombre usuario,contraseña hasheada*.

Además, el servidor soporta no solo escuchar en TCP plano, sino que al mismo tiempo puede escuchar en un puerto alternativo con el protocolo TLS por arriba de TCP para tener conexiones cifradas y seguras.

Oportunidad de Mejora

Uno de los problemas que tiene el servidor es la lectura de los streams. Actualmente revisa constantemente en todos las conexiones abiertas que tiene en búsqueda de nueva información. Hacer esto de manera constante consume mucho CPU y es en general ineficiente. Se podrían utilizar patrones de polling como por ejemplo la librería *epoll*. Para recibir eventos de manera más eficiente.

Comparando el rendimiento del servidor de mensajería implementado por nosotros y la implementación oficial de *nats-server*, encontramos que la capacidad de la nuestra es de entre un 5% y un 10% de la capacidad del original. Es decir, existe un montón de espacio para la mejora.



Esta prueba se realizó utilizando el CLI de NATS. Se utilizó el comando *nats bench foo --pub 1 --size 16*. Se realizaron varias pruebas y los resultados mostrados son un promedio de las mismas.

Implementación del cliente de NATS

Core

El cliente de NATS permite la comunicación con el servidor. La implementación consta de dos partes principales, la instancia del cliente y el hilo de ejecución. Cuando se instancia un nuevo cliente, este crea un hilo o *thread* el cual se encarga exclusivamente de la comunicación con el servidor. La instancia es controlada por el programador, y expone los métodos necesarios para la utilización del mismo.

La instancia se comunica con el hilo de comunicación a través de un canal de instrucciones. Estas instrucciones pueden ser publicar, suscribirse, desuscribirse y desconectar. Cuando se suscribe a un tópico, se le envía una instrucción al hilo de comunicación la cual incluye en su contenido un emisor de un channel de mensajes exclusivo para los mensajes correspondientes a esa suscripción. El programador se queda con un objeto independiente del cliente el cual le permite leer los mensajes de la suscripción.

Las desuscripciones y desconexiones por lado del cliente se manejan implementando el trait *drop*, es decir, cuando se elimina una suscripción o el cliente, se envía automáticamente la instrucción correspondiente al hilo de comunicación. Evitando que queden suscripciones o conexiones abiertas que ya no son alcanzables. En el caso de una desconexión del servidor inesperada, el hilo de comunicación muere y el cliente y las

suscripciones detectan esto cuando se intentan utilizar, dado que el resultado de su intento es un error de *io* de *desconexión*.

El cliente tiene una característica especial, esta es que puede ser clonado las veces que sea necesario dentro del mismo *thread* y cada clon del mismo es equivalente al mismo cliente y mismo hilo de comunicación. Esto permite mayor flexibilidad en el uso del mismo. Además las suscripciones son independientes del cliente, siendo posible tratar su *ownership* de manera independiente al cliente dando aún mayor flexibilidad.

JetStream

El cliente de JetStream se crea en base a un cliente normal de NATS. Este trae métodos útiles para crear streams y consumers. Permite consumir mensajes de un consumer pull de a uno y también consumir de manera continua mensajes de un consumer pull de la misma manera que se trabaja con una suscripción tradicional. En el último caso mencionado, por defecto cada vez que intenta consumir el mensaje siguiente de un consumer se hace *acknowledgement* del anterior aunque aunque también es posible controlar cuando se hace el *acknowledgement* manualmente con los métodos *no_ack* y *ack*.

Dominio del Problema

Para este trabajo nos pidieron 3 aplicaciones:

Un sistema central de cámaras, donde se podría conectar, desconectar, modificar y listar las cámaras de seguridad.

El software de control del dron, es el software que corre dentro de cada dron. Recibe la información de los incidentes para atenderlos y manda de forma recurrente su ubicación y su estado.

Por último nos pidieron una aplicación de monitoreo. En ella se debía visualizar un mapa con todos los incidentes, cámaras y drones, aparte de que debía permitirte marcar nuevos incidentes. Posteriormente, nos pidieron que esta misma aplicación permitiera conectar, desconectar y editar cámaras remotamente y ver la información de los diferentes drones.

Sistema de cámaras

Para la realización del sistema de cámaras empezamos realizando la cámara. La misma simplemente guarda su propia información y tiene funciones para brindarnos su posición y si está activa o no. Cada cámara se activa si tiene algún incidente en su rango o si alguna cámara lindante se activa.

El sistema de cámaras tenía que comunicarse con la aplicación de monitoreo para saber cuándo un incidente se generaba o terminaba. Debía saber que cámaras encender o

apagar y necesitaba un medio de comunicación por el cual nosotros pudiéramos agregar, sacar o modificar cámaras. Estas 3 funcionalidades fueron divididas de la siguiente manera.

Interfaz

La interfaz vía terminal fue el medio que usamos para poder agregar, modificar y eliminar cámaras dentro del sistema de cámaras, podríamos decir que es la forma local de hacer las cosas, ya que ocurre todo dentro de la misma aplicación. La interfaz se encarga de recibir los comandos que enviemos por terminal, interpretarlos y de ser válidos, responder a los mismos. Esto nos permite agregar y desconectar cámaras, cambiar sus rangos o ubicaciones, y pedir información de una o todas las cámaras.

Estado

El estado del sistema es el lugar donde guardamos toda la información que el sistema va a necesitar. Consta de todas las cámaras, todos los incidentes activos y todas las cámaras lindantes para cada una. Ante cualquier cambio en alguna cámara o incidente el estado se actualiza y guarda la información modificada.

Sistema

Finalmente el sistema de cámaras, es donde ocurre el loop del sistema y donde realizaremos las publicaciones en los tópicos correspondientes. El sistema inicia conectándose a NATS y suscribiéndose a los diferentes tópicos. Informamos por NATS de todas las cámaras y pedimos la información de los incidentes actuales. A partir de ese momento arrancará el loop infinito.

En él vamos a leer la información de los incidentes, los comandos enviados por la interfaz y los comandos enviados por red de la aplicación de monitoreo. Durante este loop se actualizará el estado de ser necesario y también volveremos a publicar todas las cámaras con sus respectivos estados.

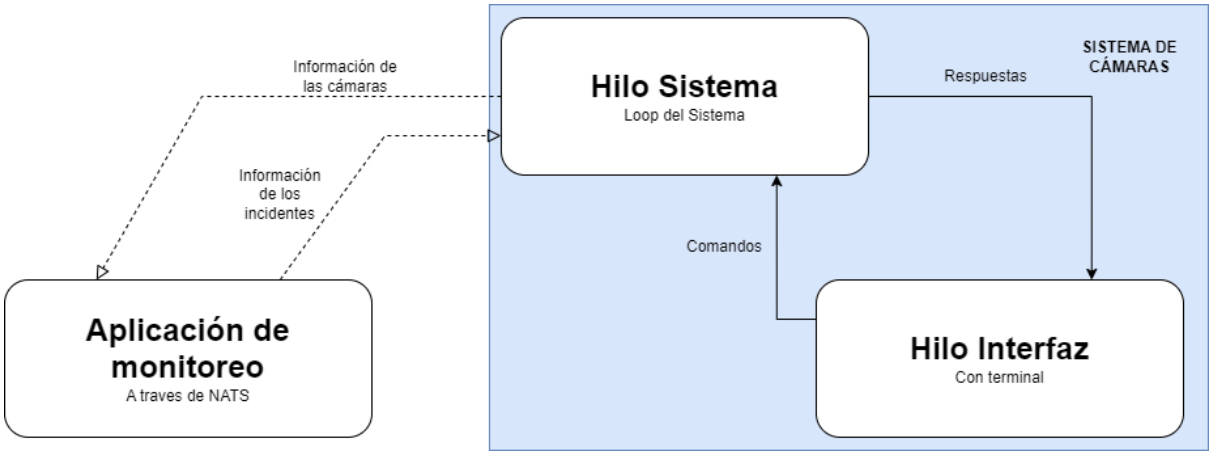


Gráfico 2: Sistema de cámaras, Comunicación con la interfaz y la aplicación de monitoreo.

Drones

Se nos solicitó hacer drones autónomos para la atención de incidentes. Para conseguir esa autonomía la comunicación con NATS era necesaria. Cada dron debía poder comunicarse con la aplicación de monitoreo para estar al tanto de los incidentes y se nos pidió que el dron enviara de forma regular su estado. Por otro lado, necesitamos la información de la batería para poder llevar el dron a su centro de carga y tiene que poder moverse para llegar a cada incidente.

Dron

Empezamos modelando el dron, el cual conoce su central de carga, su punto de espera, su velocidad, su ubicación actual, su batería, su dirección y el incidente que está atendiendo de ser así. Esto nos permite saber su acción (Incidente, Cargar, Espera) ya que la misma depende de la batería y de si el dron está atendiendo un incidente. Con esta acción podemos saber hacía dónde se dirige el dron. Sabiendo este destino podemos predecir la ubicación del dron entre cada vez que envíe su estado.

Comunicación

Antes de armar la comunicación decidimos que lo ideal sería tener un contexto, donde tener nuestro medio de comunicación, y nuestras suscripciones a incidentes finalizados y a comandos. Una vez realizado esto, generamos nuestra comunicación la cual conoce toda la información del servidor de comunicación (dirección, puerto y datos de autenticación) y la cual tiene un contexto. Esta comunicación tiene un ciclo en el cual de ser necesario envía su estado a través de NATS.

Es importante recalcar que el dron no elige presentarse o no a un incidente. Ante un nuevo incidente la aplicación de monitoreo será la encargada de revisar qué drones están disponibles y con el incidente en su rango. De esos elegirá los 2 más cercanos de haberlos, y les enviará comandos pidiéndoles que atiendan el incidente. Con esto aclarado se ve más claramente de que durante el ciclo de la comunicación revisemos si recibimos algún comando de la aplicación de monitoreo. Por último en el ciclo revisamos si hay alguna publicación de incidente finalizado, esta es nuestra forma en la que los drones salen de la acción incidente.

Sistema

Por último está el sistema interno del dron. Este conoce el dron, su comunicación,cuándo fue su última iteración y cuánto tiempo pasó desde entonces. El sistema funciona en bucle, donde descargamos la batería, movemos el dron, y vemos si hubo algún cambio importante. Si el dron llegó a destino tiene que frenar, si ese destino es su central de carga tiene que recargarse. En caso contrario calculamos la velocidad y dirección del dron. En ambos casos enviaremos el estado del dron por NATS y actualizaremos la última iteración.

Gráfico 3: Sistema del dron, recibimiento de comandos de la aplicación de monitoreo.

Aplicación de monitoreo

Por último nos pidieron una aplicación de monitoreo. En ella se debería poder registrar incidentes, ver su información y modificarlos. También debía poder ver las cámaras y los drones junto con su información. Tiempo después nos pidieron poder editar y modificar las cámaras desde la misma. El mayor distintivo de esta aplicación, es ser la única para la cual tuvimos que hacer una interfaz gráfica.

Incidente

El primer paso era tener incidentes que crear y atender. Creamos un pequeño struct Incidente el cual tuviera el ID, detalle, ubicación, cuando fue creado y cuánto tiempo lleva atendido. El mismo te puede devolver su posición.

Reglas generales de la aplicación

Para esta implementación necesitábamos una interfaz gráfica que nos permitiera ver lo que ocurría, y una parte lógica para realizar todas las modificaciones que se nos pidieron gestionar desde acá. En resumidas palabras, los cambios que uno ingresará por la UI, se debían realizar correctamente. Por ende, el lado lógico y el lado visual de la aplicación debían poder comunicarse entre sí. Como se ve en el gráfico a continuación, ante diversas acciones en la interfaz gráfica (crear incidente, modificar cámaras, etc) enviamos comandos al lado lógico, para que se encargue de realizar los cambios, comunicarlos y devolvernos el estado actualizado de la aplicación para poder mostrarlo.

Gráfico 4: comunicación entre la parte lógica y visual de la aplicación de monitoreo

Lógica

Para el lado lógico de la aplicación contamos con:

- Un enum con todos los comandos que podemos mandar.
- Un estado en que guardamos toda la información sobre cámaras, drones e incidentes.
- Un sistema, compuesto de un estado, el próximo ID para un incidente, su último ciclo y un booleano sobre actualizar estado.

El estado es el lugar donde guardamos toda la información. Esta información es la que podremos visualizar y en algunos casos editar. Cada elemento que se agregue, modifique o elimine, se debe actualizar en el estado. En el caso puntual de los drones, donde la información no puede ser modificada por la aplicación, el estado mantiene los drones que hayan mandado su estado hace menos de 10 segundos. En el mismo podremos consultar por información con diferentes filtros (drones disponibles, incidentes sin drones, pedir algún elemento según su ID, etc).

El sistema arranca viendo si hay incidentes que levantar de algún archivo previo. De ser así, veremos sus IDs para poder revalorizar la variable del próximo ID. Posteriormente actualizaremos el estado de la interfaz gráfica, nos conectaremos con NATS, publicaremos el estado de los incidentes, realizaremos todas las suscripciones necesarias y pediremos la información de las cámaras. Una vez finalizado esto arrancará el loop en el que:

- Leemos las cámaras
- Leemos el comando desde la Ui y actuamos en consecuencia.
- Si alguna aplicación nos lo pide, publicamos el estado de los incidentes
- Leemos los drones

y una vez por segundo además realiza las siguientes tareas.

- Finalizamos los incidentes que hayan pasado su tiempo (y lo publicamos por NATS)
- Desasignamos drones si algún incidente tiene drones de más.
- Eliminamos los drones que no hayan mandado su estado hace más de 10 segundos
- Asignamos drones a los incidentes a los que les falte (lo publicamos por NATS)

Interfaz gráfica

Gráfico 5: Vista general de la aplicación de monitoreo

Como se ve en el gráfico 5, una vez autenticado la aplicación tiene 4 puntos importantes. En la esquina inferior izquierda podemos ver 3 botones, dos de ellos manejan el zoom (más o menos zoom) y el tercero te lleva al punto de inicio.

En la esquina inferior derecha podemos elegir qué queremos listar en la esquina superior derecha. También tiene el botón que te permite salir que te devuelve a la ventana de autenticación.

En la esquina superior derecha contamos con el listado seleccionado previamente, este listado es scrollable y cada elemento listado es clickeable. En dicho caso, el mapa se moverá para centrarte el elemento seleccionado.

Por último tenemos la esquina superior izquierda, la cual es la ventana que más variaciones tiene. Para esto tomaré un momento para explicar la línea de pensamiento para la interfaz gráfica. Dependiendo que se este listando, vos podrías seleccionar un elemento distinto, y lo que puedas hacer con ese elemento depende de qué elemento es. Si volvemos al gráfico 5, estamos listando incidentes, por lo que podemos seleccionar incidentes en la esquina superior derecha, y al ser incidentes, podemos modificarles la ubicación, el detalle o directamente finalizar el incidente, además de ver toda la información del mismo. Si estuviésemos listando incidentes pero no hubiéramos seleccionado ninguno, al hacer click en cualquier posición tendríamos una ventana de texto por si quisiéramos generar un incidente en el punto marcado. El caso de las cámaras es bastante similar, si no hay ninguna seleccionada, te aparecerá una ventana para escribir el rango y conectar una cámara en ese punto, y si hubiera alguna cámara seleccionada, podríamos ver su información, desconectarla, cambiar su ubicación o su rango. El más limitado en este aspecto son los drones. Al ser independientes no puedes generarlos ni modificarlos. Si seleccionamos un dron, sólo podrás

ver su información, y de no haber ningún dron seleccionado, simplemente te dejará seguir creando incidentes.

Organización del Código

El código en el repositorio se organiza en una estructura de monorepo. Es decir, cada parte del proyecto es un paquete de cargo independiente. Además, los paquetes pueden depender entre sí.

Paquetes del repositorio:

- lib
- messaging-client *(utiliza lib)*
- messaging-server *(utiliza lib)*
- drone *(utiliza de lib y messaging-client)*
- monitoring *(utiliza de lib y messaging-client)*
- cameras *(utiliza de lib y messaging-client)*
- drones-launcher *(utiliza drone)*

Esta organización permite tener un punto de compilación por cada paquete, además de tener una separación lógica de cada elemento del proyecto.

Serialización y Deserialización

Para la implementación de las diferentes partes del proyecto se utilizaron varios "structs" que se comparten en diferentes partes del proyecto. Además estos structs son enviados muchas veces a través de la red y/o guardados en archivos. Para hacer esto de manera estandarizada y evitar repetir tanto código, creamos y utilizamos algunas herramientas para esta tarea.

Creamos un "trait" o interfaz genérica que implementan estos objetos llamada "Serializable". El cual indica que que el struct puede serializarse a un string y puede ser serializado desde un string. Utilizamos el formato *csv* para las serializaciones. Esto debido a su simplicidad y su capacidad de ser leído y modificado directamente por una persona.

Para simplificar el proceso creamos dos estructuras "Serializador" y "Deserializador" que nos dan las primitivas básicas para serializar y deserializar elementos de la manera más simple posible. La serialización funciona teniendo dos métodos, *agregar_elemento* y *agregar_elemento_serializable* ambos métodos permiten agregar valores a la fila del csv. La principal diferencia es que *agregar_elemento* agrega elementos primitivos, del propio lenguaje o que implementen *to_string* y a *agregar_elemento_serializable* agrega elementos que implementan el trait "Serializable".

De manera equivalente pero al revés, "Deserializador" tiene los métodos *sacar_elemento* y *saccar_elemento_serializable* los cuales van extrayendo en orden los elementos de la línea csv. Un detalle importante es que se utilizan tipos de datos genéricos

para que automáticamente se serialice o deserialice desde o hacia el tipo de dato correspondiente según el contexto del código o explicitado.

Configuración

Tanto los drones, como la app de monitoreo y el servidor de mensajería tienen varias opciones configurables al inicio de los mismos. Por ejemplo, el servidor permite configurar el puerto y la dirección en las cuales escucha. Para simplificar este proceso, creamos un struct que nos permite fácilmente leer parámetros de configuración en el formato *clave=valor* o *clave="valor"* ya sea desde un archivo y/o desde los propios argumentos de ejecución del programa *(argv)*. Por ejemplo, se puede llamar al servidor como *messaging-server puerto=4444* esto indica que debe escuchar en el puerto 4444.

Esta estructura se encarga de leer y parsear automáticamente los parámetros de *argv* y de el archivo de configuración que se especifique en el código o en el argumento *config=*. Además permite extraer los diferentes valores transformándolos automáticamente al tipo correspondiente. Para esto, los tipos de datos que se extraen deben implementar *FromStr*. Y para setear valores deben implementar *to_string*.

Detección de incidentes con AI

Introducción

Como parte del proyecto se nos pidió que las cámaras pudieran detectar incidentes utilizando inteligencia artificial. Para realizar esto, el sistema debe simular la captura de imágenes por parte de las cámaras y analizar las mismas de alguna forma en busca de posibles incidentes.

Simulación de captura de imágenes

En el alcance del proyecto, no es posible que las cámaras hagan capturas reales. Por este motivo, para poder hacer que las cámaras graben imágenes se utiliza el sistema de archivos. Cada cámara tiene un directorio asignado según su id dentro de un directorio padre donde están también el resto de las cámaras, en el cual constantemente busca archivos de imágenes nuevos. Cuando detecta una imagen nueva, la lee, la analiza y luego elimina el archivo. Si hay varios archivos utiliza uno random.

Proveedores de AI

Características generales

Para poder analizar y detectar un incidente en una imagen, necesitamos un sistema o modelo que nos permita analizar la imagen dentro de un grupo definido de opciones. Estas opciones son los diferentes tipos de incidentes (o no incidentes) que pueden ocurrir.

Generalmente a este tipo de tecnología se la conoce como clasificación de imágenes o “labeling”. Existen varias alternativas comerciales y open source para esto.

Necesitamos que la tecnología que utilicemos soporte la posibilidad de definir etiquetas de clasificación personalizadas, definir un dataset de entrenamiento personalizado y que tenga la capacidad de escalar eventualmente a una gran cantidad de cámaras. Para la elección de la tecnología probamos cuatro diferentes alternativas para cumplir con esta tarea.

Google Vertex AI - AutoML

Google tiene un servicio anteriormente conocido como Cloud Visión, con varias funciones de análisis de imágenes entre ellas AutoML. Esta nos permite crear un dataset de imágenes y etiquetas personalizados, entrenarlo automáticamente y posteriormente utilizarlo con una API.

Inicialmente hicimos unas pruebas con un dataset de 800 imágenes y categorías de incendios, accidentes y normal (siendo normal en caso de que no haya accidentes). En esta prueba descubrimos que las interfaces gráficas de Google Cloud y en general el uso de la consola es lento e ineficiente. Es posible que se pueda automatizar todo desde un script pero utilizando la consola las cosas eran lentas e ineficientes. Además no tenía una opción rápida de etiquetar las imágenes según el nombre de la carpeta. Probando el modelo entrenado en Google Cloud encontramos que los resultados eran aceptables pero aún así fallaba en muchos casos.

Tensor Flow y Python

Una alternativa era utilizar una modelo entrenado localmente. Siguiendo el tutorial oficial del canal de Tensor Flow en Youtube generamos un modelo de clasificación de imágenes que utiliza automáticamente una carpeta dividida en subcarpetas de imágenes según sus etiquetas. Este modelo tenía varias ventajas, entre ellas el costo cero, la posibilidad de personalizarlo y una mejor experiencia de desarrollo ya que no dependemos de subir archivos a cloud y seguir procesos lentos de cada proveedor. Desafortunadamente, dado a no ser una opción 100% administrada y automática, nos encontramos con que la precisión del modelo era baja y por algún motivo creía que muchas cosas eran incendios cuando no las eran. Una desventaja importante de esta solución es que requiere configurar varios parámetros de entrenamiento de manera manual los cuales no tenemos suficiente conocimiento para utilizarlos eficazmente.

Microsoft Azure

Intentando crear un proyecto en Azure nos encontramos varias barreras entre ellas la ridículamente desastroso sistema para registrarse y la falta de documentación e información clara en como utilizar los diferentes servicios. Debido a esto decidimos no continuar con este proveedor.

AWS Rekognition

Dentro de las opciones consideradas esta fue la que mejor experiencia nos dió de todas. Las opciones para crear datasets y inferir las etiquetas de las carpetas sumado a mejores herramientas para cargar los archivos facilitaron mucho el trabajo. Además los tiempos de espera para las diferentes operaciones, especialmente entrenar el modelo fueron los más bajos de todos. Encontramos que con un dataset similar al utilizado en Google Cloud pero revisado y con una menor cantidad de imágenes logramos una precisión suficientemente buena para nuestras necesidades. Además existe una librería que nos permite utilizar la API desde rust directamente con menos complicaciones.

Elección de servicio

Dado la experiencia con las tecnologías mencionadas anteriormente, decidimos quedarnos con AWS Rekognition por ser esta la más simple de utilizar y la que mejores resultados nos dió.

Creación de Dataset

Para el entrenamiento de la AI debimos generar un dataset de imágenes de diferentes tipos de accidentes. Está separado en tres categorías, normal, accidentes e incendios. Las imágenes que entran dentro de la categoría normal son una mezcla de imágenes de diferentes datasets y Google Imágenes que incluyen imágenes de diferentes partes de una ciudad, imágenes de tráfico en diferentes situaciones que no son accidentes y otras imágenes variadas que no corresponden a incidentes pero que sirven para evitar el sobreajuste del modelo.

La categoría de incendios contiene imágenes de diferentes situaciones con fuego y humo, en ciudades e incluso imágenes de fuego en incendio en situaciones no tan específicas para agregar variedad al modelo. Por último, las imágenes de accidentes fueron sacadas de Google Imágenes y la mayoría de dos datasets distintos de accidentes captados por cámaras de seguridad en la calle.

En total tenemos 144 imágenes de incendios, 114 imágenes de accidentes y 443 imágenes variadas de ciudades y otros elementos que corresponden a casos de no accidentes.

Dataset de prueba

Para la demostración de funcionamiento del sistema, utilizamos un dataset de imágenes con situaciones variadas de emergencias en la ciudad las cuales no tienen en general ningún parecido cercano con las imágenes de entrenamiento. Es decir, para probar el correcto funcionamiento del modelo utilizamos imágenes de diferente fuente que las de entrenamiento. De esta forma nos aseguramos de que el modelo no está sobreentrenado con las imágenes del dataset original y reducimos los sesgos del mismo.

Cabe destacar que de todas formas el dataset de entrenamiento sigue siendo reducido y la eficacia del modelo actual no sería suficiente para condiciones reales. En caso de utilizar este sistema en la vida real sería necesario entrenar el modelo con muchísimas más imágenes de mayor calidad y probablemente agregar etiquetas para otros tipos de accidentes.

Análisis de costos

La detección de etiquetas personalizadas de AWS Recognition tiene unos costos de $4.00 USD por hora de ejecución de una unidad de inferencia y $1.00 USD por hora de entrenamiento. Para este análisis de costos, vamos a ignorar el costo de entrenamiento ya que este es mínimo en comparación del costo de utilización de unidades de inferencia a gran escala y de manera permanente.

Valores considerados para el cálculo

Vamos a definir algunos valores generales para realizar el cálculo de costo de utilizar Rekognition para detectar etiquetas en imágenes.

- Una instancia de inferencia puede analizar 5 imágenes por segundo (puede variar según el modelo, cinco por segundo es razonable para el modelo utilizado en nuestro caso). Equivale a 18.000 imágenes por hora.
- Vamos a tomar en cuenta que en un principio tenemos 5000 cámaras de seguridad con esta tecnología habilitada.
- Cada cámara captura una imagen por segundo pero solo cuando detecta movimiento.
- Las cámaras en promedio detectan movimiento el 40% del tiempo

Cálculo de costo

Considerando estos valores establecidos, tenemos que por hora se generan 7.200.000 imágenes que deberán ser analizadas. Esto requiere 400 unidades de inferencia llegando a un costo de $1600 USD por hora o un equivalente a $1.168.000 USD mensuales (con un mes de 730 horas). El costo por mes por cámara sería de 233 dólares. Se podría optimizar el costo cambiando la frecuencia de análisis de imágenes y optimizando el funcionamiento del modelo para soportar más transacciones por segundo por unidad de inferencia.

Implementación de la solución

Para la implementación de la solución se creó un paquete en el repositorio del proyecto exclusivamente para la llamada a la API de AWS Rekognition. Utiliza la librería para rust *aws-sdk-rekognition* la cual nos permite utilizar la API fácilmente. Dentro de la estructura existente del proyecto, el sistema de cámaras es el que se encarga del proceso de lectura de imágenes y llamadas a la función de reconocer imagen.

El análisis de los directorios en búsqueda de nuevas imágenes para analizar está implementado utilizando un hilo de ejecución por cada cámara existente en el sistema de cámaras. Cada hilo de detección conoce a qué cámara pertenece y dónde debe buscar archivos nuevos. Este se ejecuta de manera síncrona en un loop el cual busca una imagen nueva, si la encuentra la analiza y envía el resultado al hilo principal del sistema de cámaras usando un channel y vuelve a realizar el proceso indefinidamente. El hilo de detección de cámara se detiene cuando se detiene el sistema de cámaras o cuando se desconecta una cámara.

El hilo principal del sistema de cámaras recibe a través de un channel las detecciones de etiquetas y en base a los resultados decide si crear o no un incidente. En caso de que efectivamente haya un incidente válido, se crea el mismo, en su detalle se explicita el tipo de incidente según la detección y se envía por un stream de NATS al la app de monitoreo la detección para que esta genere el incidente.

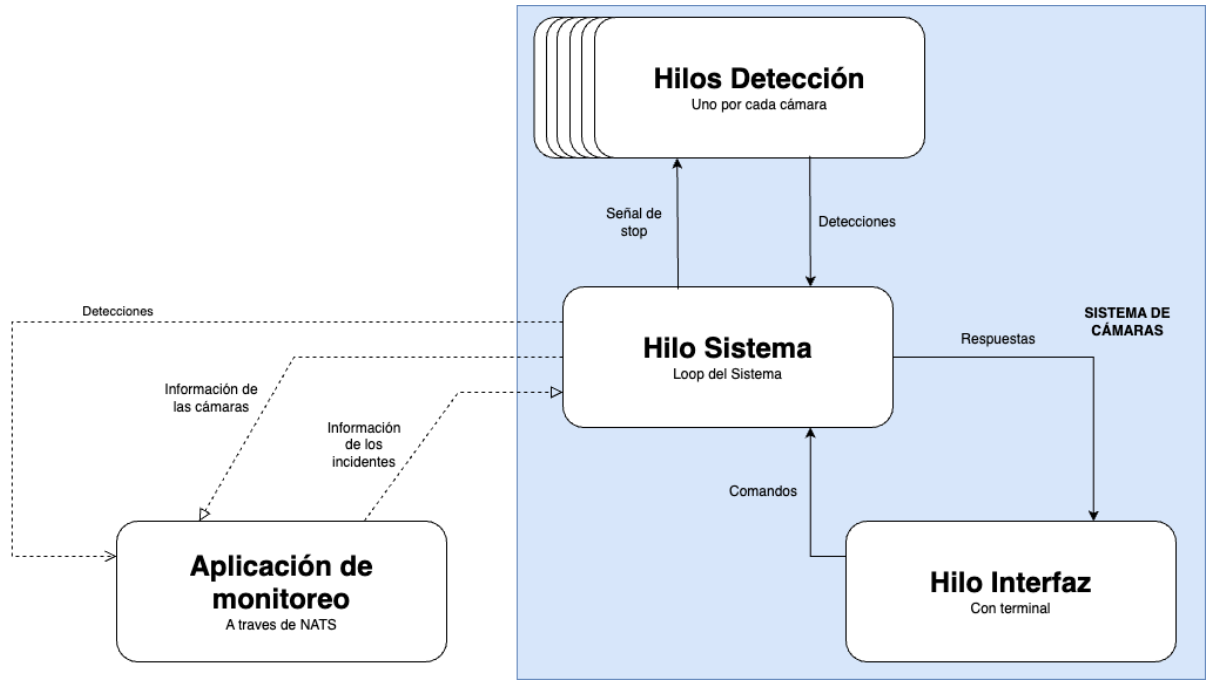


Gráfico 6: Implementación del sistema de cámaras con reconocimiento de imágenes.

Fuentes

- https://aws.amazon.com/es/blogs/machine-learning/calculate-inference-units-for-an-amazon-rekognition-custom-labels-model/
- https://aws.amazon.com/es/rekognition/pricing/
- https://www.youtube.com/watch?v=u2TjZzNuly8