

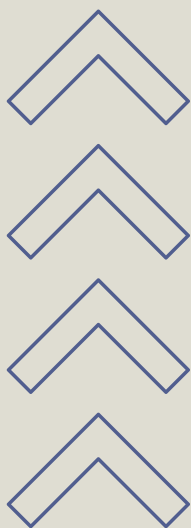


OCT 2024

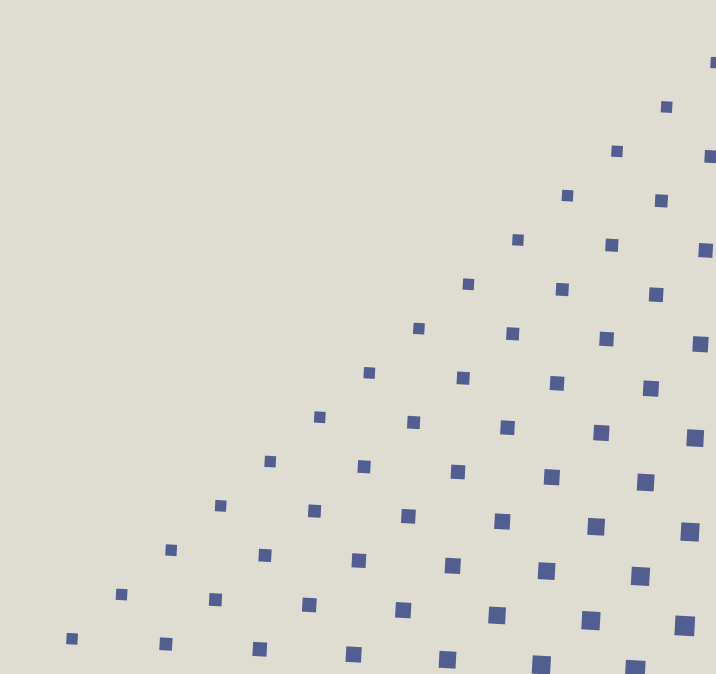
DELFI Y SUS UMPALUMPAS

AEROLÍNEAS RÚSTICAS

CÁTEDRA DEYMONNAZ




Borthaburu, Isidro Héctor
Cano Ros Langrehr, María Delfina
Dinucci, Tomás Franco
Wainwright, Martín





ÍNDICE

• Introducción e Investigación	3
• Modelado y Diseño de Datos	
◦ Keyspaces y Tablas	4
◦ Partition y Clustering Keys	5
◦ Operaciones Básicas	5
◦ Replicación Básica	6
• Protocolo	7
• Decisiones Tomadas	10
• Diagramas	12
• Interfaz Gráfica	14
• Conclusión	16





INTRODUCCIÓN E INVESTIGACIÓN

A. INTRODUCCIÓN

En el presente informe se detalla el desarrollo de un proyecto orientado a sistemas de aeropuertos donde a estos les corresponden distintos aviones. Para poder verificar la base de datos de cada aeropuerto y la información de cada avión, se decidió utilizar el protocolo Cassandra Query Language o más conocido como CQL. Por lo tanto, en primer lugar implementamos gran parte del protocolo CQL que nos sirviera para poder realizar las distintas queries y accesos a la base de datos para poder mantener el sistema de aviones que queríamos desarrollar. Luego realizamos la parte de la interfaz gráfica que le realiza consultas a la base de datos implementada anteriormente.

B. INVESTIGACIÓN PRELIMINAR

Antes de comenzar con el desarrollo, fue crucial realizar una investigación detallada sobre la base de datos Cassandra. Inicialmente desconocida para nuestro equipo, Cassandra se reveló como fundamental debido a sus características de alta escalabilidad y disponibilidad en entornos distribuidos. Esta base de datos se destacó frente a otras soluciones NoSQL como MongoDB o HBase, principalmente por su capacidad de manejar grandes volúmenes de datos distribuidos en múltiples nodos, lo cual es esencial para garantizar la tolerancia a fallos en nuestro proyecto. Además, su enfoque en la replicación y la consistencia eventual la convierte en una opción ideal para manejar los datos generados por los dispositivos en tiempo real, en nuestro caso los aviones y aeropuertos. Su arquitectura descentralizada asegura que no exista un punto único de fallo, mejorando la robustez de todo el sistema y la oportunidad de fallo ya que al caerse un nodo por ejemplo, esto no rompería la ejecución.



MODELADO Y DISEÑO DE DATOS

A. KEYSPACES Y TABLAS:

CREACIÓN DE KEYSPACES

Se soportan múltiples formatos para la creación de keyspaces:

Ejemplos:

```
CREATE KEYSPACE keyspace1 WITH REPLICATION =  
    {'class': 'SimpleStrategy',  
     'replication_factor': '3'};  
CREATE KEYSPACE keyspace3 WITH REPLICATION =  
    {'class': 'SimpleStrategy',  
     'replication_factor': '1'};
```

Al crear un keyspace:

- Se crea una carpeta nombre_keyspace para almacenar sus tablas.
- Se registra información adicional en keyspaces_info/nombre_keyspace/info.txt, incluyendo:

```
Keyspace: keyspace1  
Replication Class: SimpleStrategy  
Replication Factor: 3  
Tables: []
```

Nota: Aunque la estrategia de replicación se guarda, por ahora solo se considera el Replication Factor.

CREACIÓN DE TABLAS

Formatos soportados para creación:

- `CREATE TABLE keyspace1.orders (order_id INT, user_id INT, PRIMARY KEY (order_id), CLUSTERING KEY (user_id));`
- `CREATE TABLE keyspace2.products (product_id INT PRIMARY KEY, category_id INT, supplier_id INT, CLUSTERING KEY (category_id, supplier_id));`

Al crear una tabla:

- Se genera un archivo CSV en la carpeta del keyspace y un archivo de información en keyspaces_info/keyspace, como::

```
Table: aviones_volando  
Partition Key: [flight_number, origin]  
Clustering Key: []  
Columns: flight_number INT, origin  
VARCHAR(100), destination VARCHAR(100), ...)
```

Nota: Aunque la estrategia de replicación se guarda, por ahora solo se considera el Replication Factor.

B. PARTITION Y CLUSTERING KEYS

PARTITION KEYS

Se permite definir múltiples claves de partición. Estas claves se utilizan para identificar de forma

única cada partición y determinar el nodo correspondiente en el sistema.

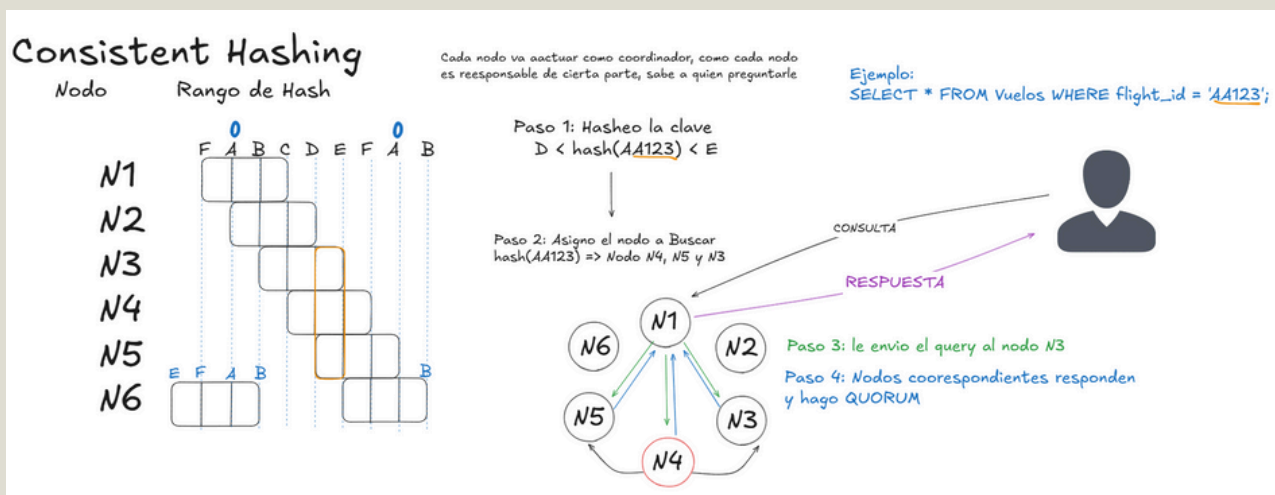
CLUSTERING KEYS

Las clustering keys se registran en la información de la tabla, pero aún no se aplican en las consultas ni en la organización interna de los datos.

C. OPERACIONES BÁSICAS

INSERCIÓN, CONSULTA, ACTUALIZACIÓN Y ELIMINACIÓN

- **Gestión de Consultas:**
 - En consultas SELECT o INSERT, se verifica la clave de partición en la información de la tabla, y se hashean para determinar el nodo correspondiente.
 - Si corresponde al nodo actual, se reenvía la consulta a los nodos replicadores definidos por el Replication Factor.
 - En caso contrario, el sistema de tokens permite identificar el nodo al que corresponde el rango de hash, asegurando que cada consulta sea procesada en el nodo correcto.
- **Sistema de Tokens:**
 - Distribución de tokens por nodos



- **Descripción:** Cada nodo tiene un rango de tokens asignado, lo que permite determinar su responsabilidad en función del hash de la partición.

- **Consultas Disponibles:**

- Ejemplos de consultas permitidas:

- `SELECT * FROM keyspace1.users WHERE id = 2000;`
 - `INSERT INTO keyspace1.users (id, name) VALUES (2000, 'John Doe');`
 - `UPDATE keyspace1.users SET name = 'Jane Doe' WHERE id = 1;`
 - `DELETE FROM keyspace1.users WHERE id = 1;`

Nota: Aunque se implementaron las consultas UPDATE y DELETE, no han sido testeadas y no se utilizan actualmente.

D. REPLICACIÓN BÁSICA

- Se implementa una replicación simple basada en el Replication Factor especificado en cada keyspace.
- Distribución: La consulta principal se reenvía a un número de nodos replicadores igual al Replication Factor para mantener la consistencia de datos.
- Rango de Tokens y Nodo Responsable: La estrategia utiliza la información del sistema de tokens para localizar el nodo correcto y propagar los datos a sus réplicas



>>>> PROTOCOLO

MENSAJES

El servidor CQL puede soportar múltiples mensajes del tipo Request que envía el cliente o Response que envía el servidor. En el siguiente listado explicaremos estos y sus implementaciones,

1. Startup

El cliente realiza un mensaje del tipo request startup donde se envía para poder realizar handshake entre el cliente y servidor, para demostrarle al cliente que la conexión está activa y es segura.

2. Authenticate

El servidor le solicita al cliente que se autentique mediante este mensaje. Nosotros decidimos que el cliente al recibir la solicitud de autenticación, ingrese su usuario y contraseña para poder verificar que las credenciales son correctas y el usuario si tiene acceso a realizar consultar.

3. AuthChallenge

Se envía el mensaje del tipo response luego de recibir el primer mensaje de auth_response donde se solicita que el cliente realice una segunda verificación siguiente el challenge propuesto por el crate “rust-sasl”.

Luego de recibir la segunda auth_response que corresponde a este mensaje, se utiliza el mecanismo Plain de SASL, donde el cliente envía las credenciales (usuario y contraseña) en texto plano, lo cual se realiza sólo en conexiones seguras (luego de realizar el handshake). Tenemos las siguientes condiciones:

1. **Validador:** el servidor define un validador para utilizar en el challenge donde se compraran las credenciales proporcionadas con las almacenadas, si coincide, la autenticación es exitosa, sino falla.
2. **Finish:** La función finish coordina el intercambio de mensajes entre el cliente y el servidor. Primero, el cliente envía sus credenciales al servidor y se tienen dos tipos de respuestas:
 - a. **Succeed:** una confirmación si ya está lista la autenticación
 - b. **Proceed** si se debe continuar con el proceso
3. **Revisión del Auth Challenge:** se configura y ejecuta este flujo de autenticación hasta que se reciba el mensaje de éxito, validando así el proceso completo de auth_response y verificando que el cliente y servidor coincidan en las credenciales.

Al finalizar el challenge y la comprobación, se devuelve AuthSuccess o Error.

4. AuthResponse

Esta mensaje del tipo request se envía en dos ocasiones:

1. Luego del mensaje `authenticate`, cuando el servidor le solicita al cliente que se autentique, este envía las credenciales cliente actual en el body del mensaje serializado en formato token: `{usuario:contraseña}`. El servidor deserializa el mensaje y corrobora que las credenciales correspondan a un usuario activo que se encuentra en el archivo `usuarios.csv`. Si es afirmativa la respuesta, el servidor le va a solicitar al cliente una segunda verificación utilizando el mensaje `auth_challenge`.
2. Luego de recibir el mensaje `auth_challenge`, el servidor envía este mensaje de `auth_response` utilizando el mecanismo de autenticación solicitado por el crate "rust-sasl", y enviamos en el body del mensaje el `response_mechanism` que leerá el el `auth_challenge` al verificarlo.

5. AuthSuccess

El mensaje de respuesta se envía luego de que el servidor haya realizado todas las verificaciones y, efectivamente, el cliente tiene acceso al servidor y puede realizar consultas.

Todo el proceso de autenticación se puede visualizar en el diagrama

6. Error

En la parte de error, se siguieron los distintos tipos de errores que ya estan definidos para el protocolo Cassandra y seguimos la misma lógica de este y mantuvimos el mismo flujo de serialización y deserialización del flujo de mensajes. Por ejemplo

- a. Bad Credentials: las credenciales ingresadas son incorrectas en el proceso de autenticación
- b. Syntax Error: la query enviada no es válida y tiene un error de tipeo o lógica, por ejemplo enviar `SLECT` en vez de `SELECT`
- c. Unauthorized: el cliente no tiene permitido realizar consultas todavía porque no empezó o finalizo el proceso de autenticación.

7. Query

La query principal se define en función de la acción que se quiera realizar en la base de datos Cassandra. Cada consulta se organiza mediante una cadena de texto que sigue la sintaxis CQL (Cassandra Query Language) y se adapta para cada tipo de consulta: vuelos entrantes, salientes, y estacionados en aeropuertos.

8. Prepare

El objetivo de la instrucción PREPARE es crear una consulta precompilada que el servidor pueda ejecutar varias veces sin recompilarla. Esto es útil en el caso de consultas con parámetros que cambian en tiempo real, como el estado o la posición de vuelos, donde la estructura de la consulta permanece constante mientras que los valores específicos pueden variar.

9. Execute

La instrucción EXECUTE en Cassandra permite ejecutar una consulta previamente preparada con el comando PREPARE. Este enfoque es particularmente eficiente para consultas repetitivas, ya que permite al cliente enviar solo el identificador de la consulta precompilada junto con los parámetros específicos que pueden cambiar con cada ejecución, en lugar de enviar toda la consulta nuevamente.

10. Schema Change

El tipo de mensaje SCHEMA_CHANGE en el protocolo de Cassandra permite informar sobre modificaciones en la estructura de la base de datos, tales como la creación, actualización o eliminación de keyspaces, tablas o índices. Este mensaje se utiliza para notificar tanto al cliente como a otros nodos en el clúster de cambios estructurales que impactan la organización de los datos. En nuestra implementación no lo utilizamos ya que CREATE no es una query pedida

11. Set Keyspace

El mensaje Set_keyspace es una respuesta del servidor Cassandra cuando un cliente envía una consulta de tipo USE, que se utiliza para seleccionar un keyspace específico en el que se ejecutarán las consultas posteriores. En nuestra implementación no lo utilizamos ya que USE no está dentro de las queries pedidas



DECISIONES TOMADAS

1. Arquitectura Modular

Motivación: Una arquitectura modular permite una mejor organización del código, facilitando la escalabilidad y el mantenimiento. Además, cada módulo puede ser desarrollado y probado de manera independiente.

Implementación: El proyecto se dividió en cinco módulos principales: Aviones, Aeropuertos, archivos, red y cliente usuario. Cada módulo tiene su propia lógica y responsabilidades claramente definidas.

2. Implementación de Estructuras y Cuerpo de Mensajes (Body)

Para la correcta comunicación entre cliente y servidor, decidimos implementar un enfoque estructurado mediante el uso de una **enumeración Body**. Esta estructura centralizada permite organizar todos los posibles mensajes que viajan en ambas direcciones, de manera que cada tipo de mensaje (por ejemplo, Query, Startup, AuthResponse) esté claramente definido y encapsulado. Esto no solo nos facilita el envío y recepción de datos, sino que también asegura que cada mensaje sea interpretado adecuadamente según su propósito.

Cada variante de Body está acompañada por **métodos de serialización y deserialización**, permitiendo que los datos se transformen sin problemas entre el formato binario, necesario para la transmisión en red, y el formato estructurado en Rust. Este esquema, compatible con el protocolo de Cassandra, asegura que cada campo del mensaje se gestione en formato big-endian, cumpliendo así con los estándares del protocolo. Esto nos proporciona gran ventaja a nivel de control y seguridad de datos, ya que cada mensaje se convierte directamente en bytes antes de enviarse y se reconstruye correctamente en el destino sin pérdida de información ni errores de interpretación.

Esta implementación no solo optimiza la comunicación y el procesamiento de datos, sino que también nos permite **gestionar las interacciones internas** en los nodos. Además, tener la serialización y deserialización centralizadas en Body garantiza que, si en el futuro se desea añadir un nuevo cliente, este pueda integrarse sin problemas al sistema y cumplir con el protocolo de comunicación, manteniendo una arquitectura clara, extensible y compatible con los principios de Cassandra.



3. Interfaz de Usuario Egui

Motivación: Se eligió Egui para la interfaz gráfica de usuario debido a su simplicidad, eficiencia y una integración fluida con el ecosistema Rust. Egui proporciona las herramientas necesarias para diseñar una interfaz intuitiva y eficiente.

Implementación: La Egui desarrollada permite la visualización en tiempo real de la ubicación y estado de los aviones y Aeropuertos, facilitando la gestión y monitoreo de pasajes.

4. Autenticación

Únicamente en el caso de la interfaz gráfica, se le solicita al cliente el usuario y contraseña una única vez, y así poder realizar el handshake y authenticate para todos los nodos. De esta forma, la interfaz de usuario es más amigable, ya que el usuario se autentica una única vez y el código se encarga de autenticarlo en cada nodo.

5. Mensajes Protocolo

Dentro del protocolo decidimos no implementar los siguientes mensajes

- **Set Keyspace:** Tomamos la decisión como equipo de no tomar en cuenta este tipo de mensaje Result ya que es un caso muy específico cuando el cliente manda una consulta con "USE". Cómo select abarca esta función y más decidimos que Set Keyspace podría no ser implementado
- **Options:** Tomamos la decisión de no implementar el opcode Options ya que los pasos a seguir son siempre los mismos y agregarle el Options agregaría más complejidad para el usuario

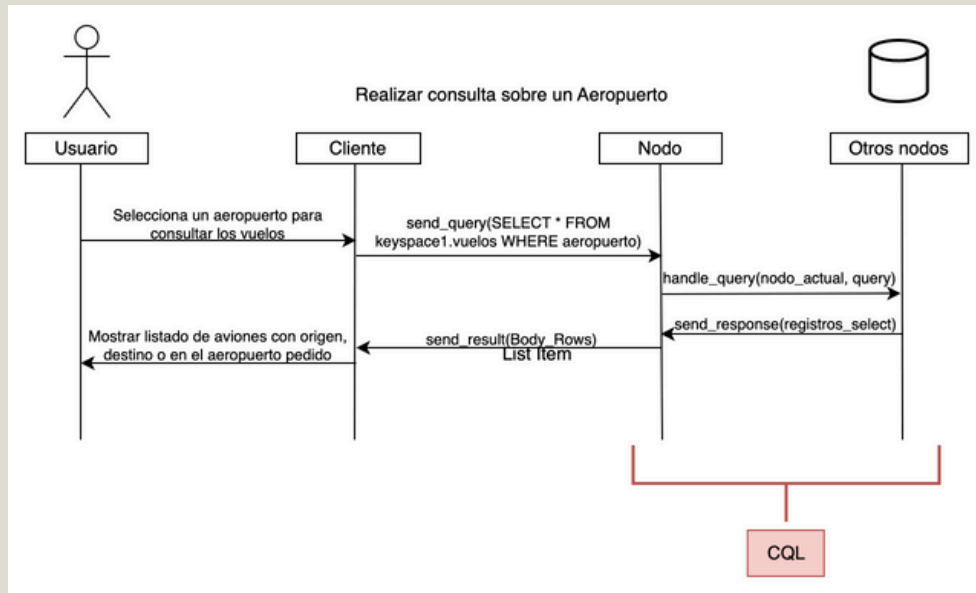
5. Cassandra Query Language

Drop y Update: Como equipo decidimos que la query update y Drop no sean usada ya que el Insert puede cumplir todas estas funciones, de todas maneras creamos su implementación aunque no se utilice

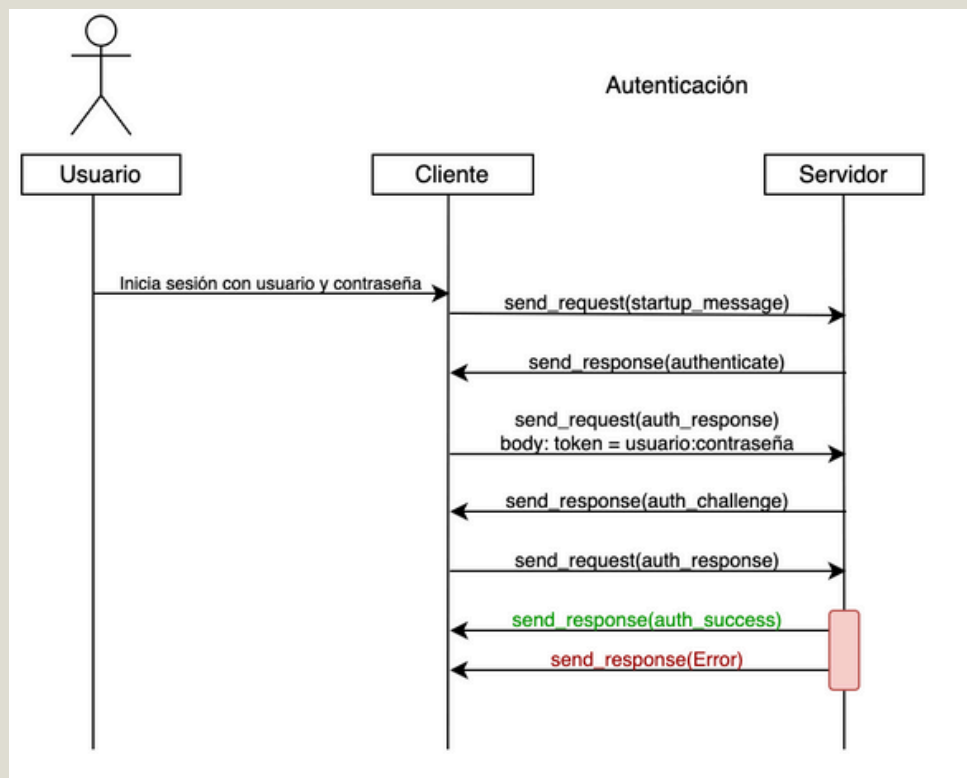
DIAGRAMAS

CONSULTAS A AEROPUERTOS

El cliente le envía una consulta al servidor que la lee y las deserializa para enviársela a CQL que le entrega las filas que cumple el lo pedido

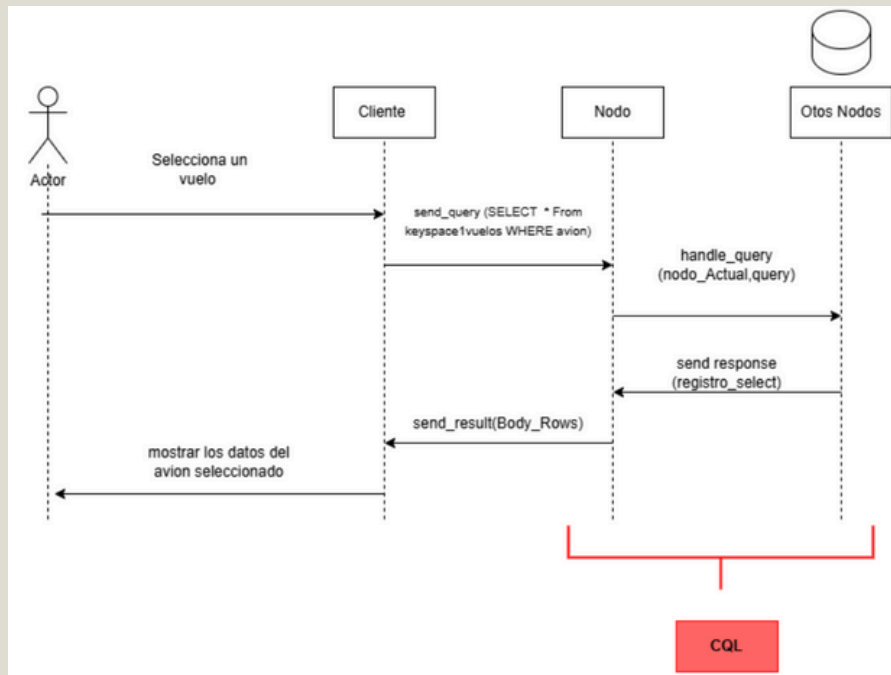


AUTENTICACIÓN



CONSULTAS AVIÓN

El cliente le envía una consulta al nodo, quien que la lee y las deseariliza para enviársela a CQL que le entrega las filas que cumple el lo pedido





>>>> INTERFAZ GRÁFICA

ACTIVACIÓN DE AEROPUERTOS/NODOS

- Al ejecutar un mensaje de autenticación en la aplicación, se activa automáticamente el nodo del aeropuerto correspondiente y se establecen las conexiones necesarias. Esto permite que el usuario pueda visualizar los vuelos asignados a cada aeropuerto de manera específica y eficiente.
- Una vez autenticado, al seleccionar un aeropuerto, se despliega toda la información relevante de los vuelos en ese aeropuerto, incluyendo vuelos próximos, entrantes y salientes. Cada nodo de aeropuerto utiliza conexiones **TcpStream** para la comunicación en tiempo real con el servidor, permitiendo que los datos de vuelos se actualicen automáticamente.

DESPLIEGUE DE AVIONES

- La interfaz presenta una vista del mapa, donde los aeropuertos disponibles se muestran en su ubicación geográfica. Al seleccionar un aeropuerto, la lista de vuelos asignados a ese nodo (aeropuerto) se despliega en el panel lateral.
- Cada vuelo puede visualizarse en el mapa con un ícono de avión, el cual incluye detalles de posición y dirección (norte, sur, este, oeste). Los aviones se actualizan automáticamente en el mapa mientras están en curso hacia su destino, mostrando datos en tiempo real como velocidad, altitud, dirección y porcentaje de combustible, lo que ofrece una experiencia dinámica y visualmente interactiva para el usuario.

FIN DE CONSULTA

- La consulta de cada vuelo termina automáticamente cuando el avión llega a su destino, momento en el cual se detiene la actualización de sus datos. Esto permite una gestión automática y eficiente del tráfico aéreo en el sistema, manteniendo la interfaz limpia y enfocada en los vuelos activos.

En la siguiente imagen se puede observar como está la interfaz gráfica actualmente, donde se puede seleccionar una fecha y el aeropuerto del cuál se quiere consultar distintos vuelos ya sea próximos a despegar, vuelos que despegaron del aeropuerto seleccionado y los que están llegando también.

Continuando, se puede seleccionar alguno de los vuelos de ese aeropuerto para mostrar toda su información y esta se va actualizando cada corto plazo. De esta forma, los valores del avión y su posición van variando a medida que el tiempo pasa. Los vuelos se pueden seleccionar desde el mapa seleccionando el avión o desde el listado de los disponibles.

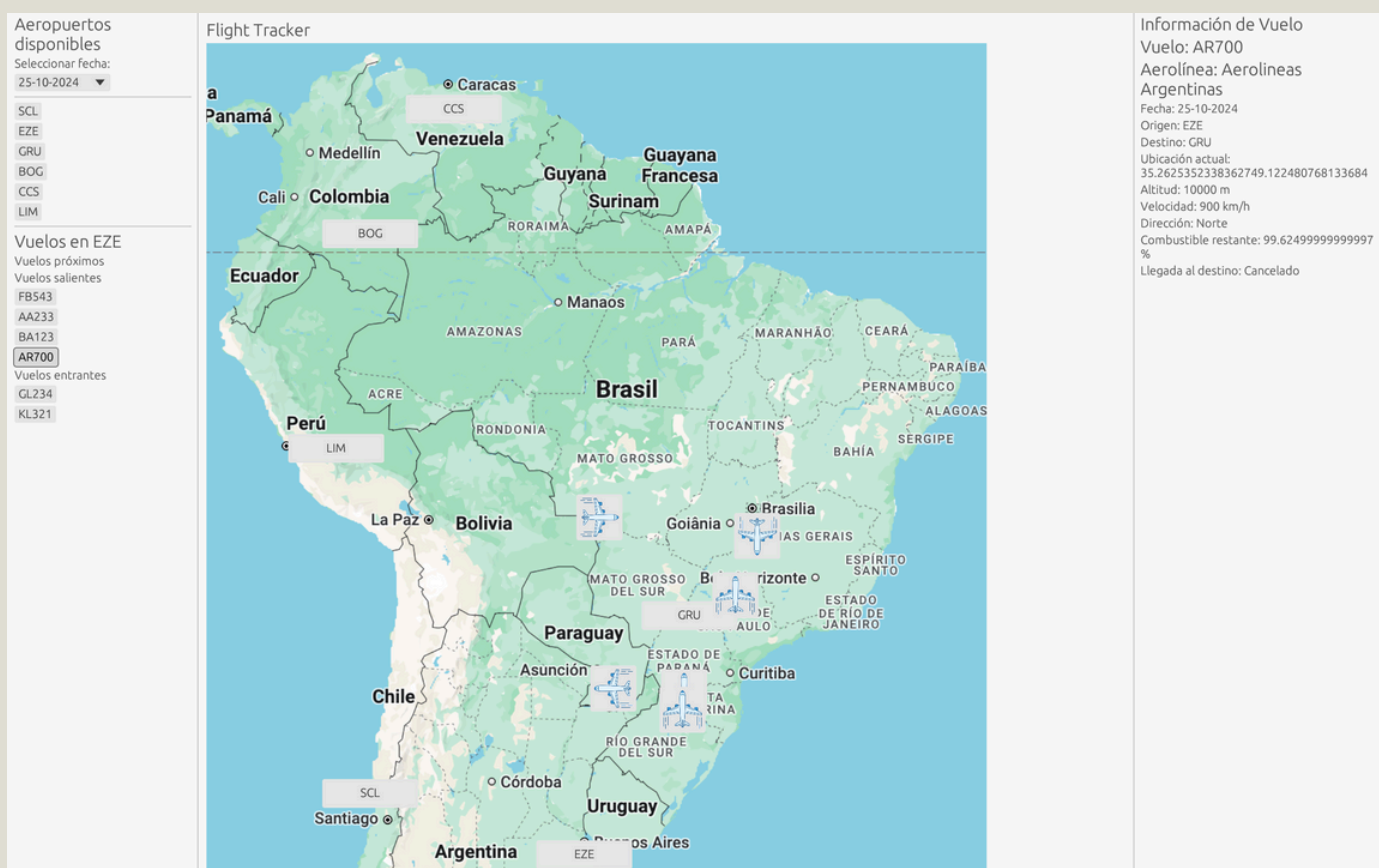


Imagen de la interfaz gráfica implementada



>>>>> CONCLUSIÓN

El desarrollo de este sistema de gestión de aeropuertos y vuelos basado en el protocolo Casssadra Query Language (CQL) ha permitido que creemos y mantengamos una estructura estable que permita que consultemos sobre diferentes registros de grandes volúmenes de datos en tiempo real.

Gracias al modelo de Cassandra pudimos implementar un modelo de datos que nos permitió crear y manipular las keyspaces, tablas y claves de partición, de esta forma podemos optimizar el almacenamiento de como se guardan los datos y como los obtenemos utilizando la partition key y el consistent hashing.

Además, al permitir la replicación básica de datos, nos permite que en caso de que alguno o varios nodos fallen, el sistema no y se pueda continuar utilizandolo hasta que los nodos se reconecten, aumentando la tolerancia de fallos.

Finalmente, optamos por el crate Egui para desarrollar la interfaz gráfica, lo que permitió que la experiencia del usuario sea *user-friendly* e intuitiva.