

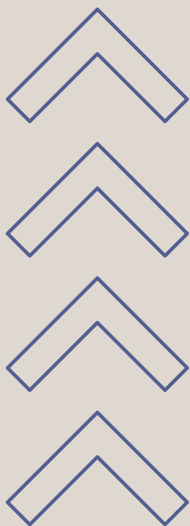


DIC 2024

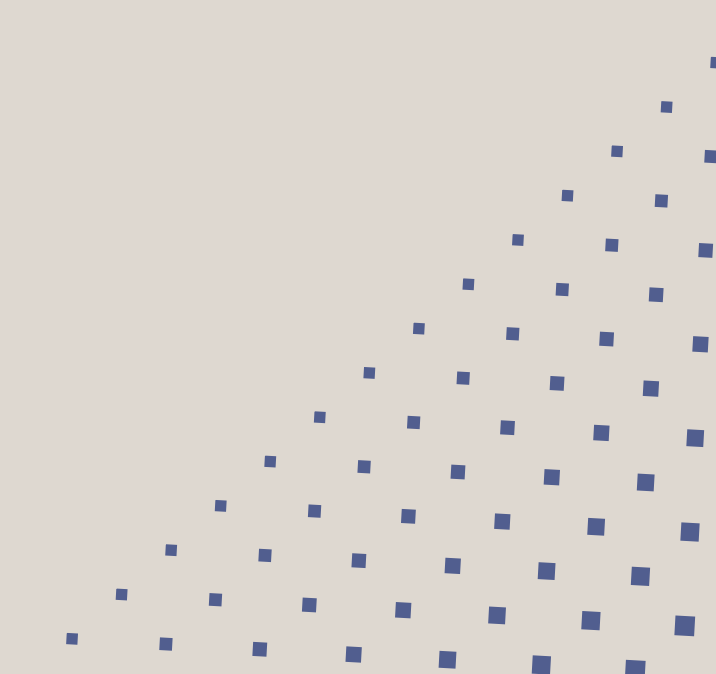
DELFI Y SUS UMPALUMPAS

# AEROLÍNEAS RÚSTICAS

CÁTEDRA DEYMONNAZ - ENTREGA FINAL




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





# TEMARIO

- Introducción e Investigación
  - Modelado y Diseño de Datos
    - Keyspaces y Tablas
    - Operaciones CRUD
    - Partition y Clustering Keys
    - Tokens y Hasheos
  - Replicación y Consistencia
  - Cluster y Seguridad
  - Simulación
  - Protocolo
  - Decisiones Tomadas
  - Diagramas
  - Interfaz Gráfica
  - Conclusión
- 



# 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:

El modelado de datos se centra en la creación y configuración de keyspaces y tablas, fundamentales para estructurar y organizar la información en el sistema.

### CREACIÓN DE KEYSPACES

Se soportan múltiples formatos para la creación de keyspaces, siguiendo la sintaxis de CQL. Ejemplos:

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

```
CREATE KEYSPACE keyspace3 WITH REPLICATION = {'class':  
'SimpleStrategy', 'replication_factor': '2'};
```

Al crear un keyspace, se realizan las siguientes acciones:

- Se crea una carpeta nombre `_keyspace` para almacenar sus tablas.
- Se registra información adicional en un archivo `info.txt` dentro de la carpeta `keyspaces_info/nombre_keyspace/`. Este archivo incluye detalles como:
  - Nombre del keyspace
  - Clase de replicación (por ahora solo se considera SimpleStrategy)
  - Factor de replicación (Replication Factor)
  - Tablas asociadas (inicialmente vacías)

Ejemplo de contenido del archivo:

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

## CREACIÓN DE TABLAS

Las tablas se crean dentro de los keyspaces con soporte para particiones y claves de clustering, optimizando el rendimiento y la distribución de datos.

Ejemplos de creación:

- ```
CREATE TABLE keyspace3.orders5 (order_id INT, user_id INT, PRIMARY KEY (order_id, user_id));
```
- ```
CREATE TABLE keyspace1.aviones_volando (flight_number INT, origin VARCHAR(100), destination VARCHAR(100), lat FLOAT, lon VARCHAR(100), altitude VARCHAR(100), speed INT, airline VARCHAR(100), direction VARCHAR(50), fuel_percentage FLOAT, status VARCHAR(100), fecha VARCHAR(100), PRIMARY KEY ((flight_number, origin, destination), origin, fecha));
```
- ```
CREATE TABLE keyspace2.aviones_en_aeropuerto ( flight_number VARCHAR(100), origin VARCHAR(100), destination VARCHAR(100), airline VARCHAR(100), departure VARCHAR(100), state VARCHAR(100), fecha VARCHAR(100), PRIMARY KEY ((flight_number, origin), origin, fecha));
```

Al crear una tabla:

- Se genera un archivo CSV en la carpeta del keyspace para almacenar los datos.
- Se crea un archivo de información adicional en `keyspaces_info/keyspace/tabla.txt`, que incluye:
  - Nombre de la tabla
  - Clave de partición
  - Clave de clustering
  - Columnas definidas

Ejemplo de información de tabla:

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



# OPERACIONES CRUD

## 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. El primer parámetro de la primary key corresponde a las partition keys

### CLUSTERING KEYS

Las clustering keys se registran en la información de la tabla, generando que luego al insertar se tenga en cuenta estas para almacenarlas en orden para mejorar su efectividad. A partir del segundo parámetro, todas las columnas seleccionadas son clustering keys.

Ejemplo:

```
CREATE TABLE keyspace2.aviones_ejemplo (flight_number VARCHAR(100), origin VARCHAR(100),  
destination VARCHAR(100), airline VARCHAR(100), departure VARCHAR(100), state  
VARCHAR(100), fecha VARCHAR(100), PRIMARY KEY ((flight_number, airline), origin, fecha));
```

Partition keys: [flight\_number, airline], Clustering keys: [origin, fecha]

## OPERACIONES BÁSICAS CON CQL

Se implementaron operaciones de creación, lectura, actualización y eliminación utilizando CQL, como:

### INSERCIÓN DE DATOS:

```
INSERT INTO keyspace3.orders5 (order_id, user_id) USING CONSISTENCY ONE VALUES (1, 100);
```

### LECTURA DE DATOS:

```
SELECT * FROM keyspace1.aviones_volando USING CONSISTENCY ONE WHERE flight_number = 123;
```

### ACTUALIZACIÓN DE DATOS:

```
UPDATE keyspace1.aviones_volando SET fuel_percentage = 80.0 USING CONSISTENCY ONE WHERE  
flight_number = 123;
```

### ELIMINACIÓN DE DATOS:

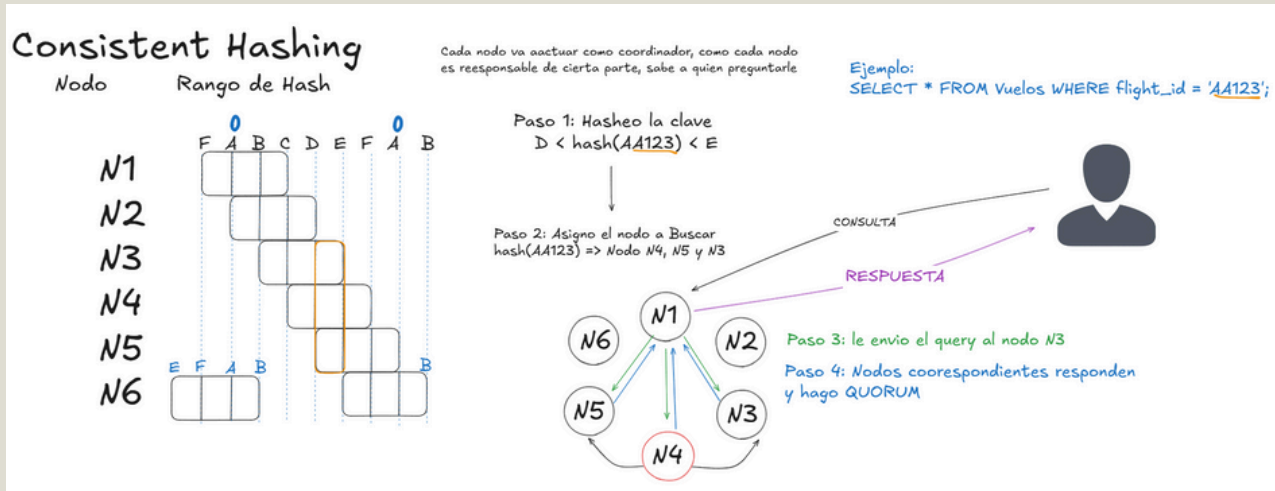
```
DELETE FROM keyspace1.aviones_volando USING CONSISTENCY ONE WHERE flight_number = 123;
```

### GESTIÓN DE CONSULTAS:

- En consultas SELECT, INSERT, UPDATE, DELETE se verifica la clave de partición en la información de la tabla, y se hashcan 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
- Cada nodo tiene un rango de tokens asignado, lo que permite determinar su responsabilidad en función del hash de la partición.

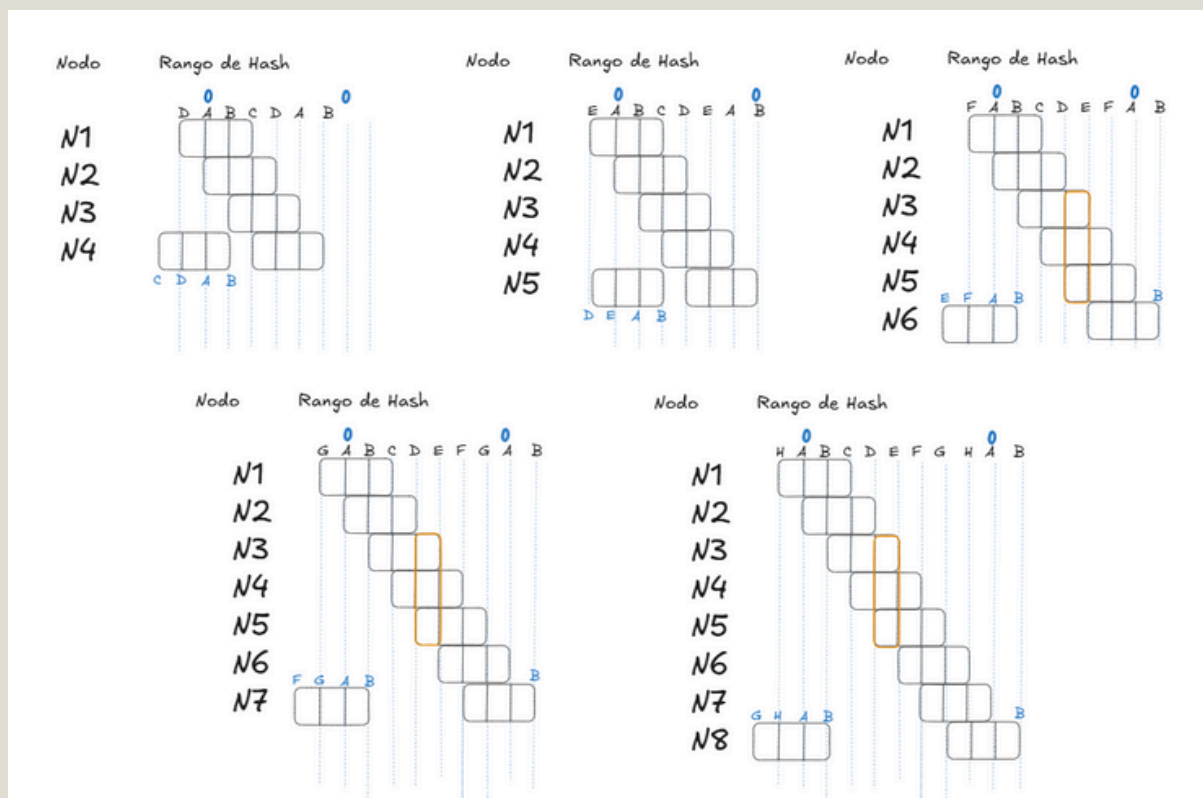


## HASHEO DE TOKENS

### PARTITION KEYS

Se pueden activar entre 4 y 8 nodos, lo que provoca cambios en el rango de alcance de cada uno de ellos.

Esto no solo expande el rango propio de cada nodo, sino que también permite adaptarse al sumar el rango de correspondencia con otros nodos a los que este replica. Así, al realizar el hash de una clave, el nodo puede determinar si dicha clave le corresponde a él o a alguno de los otros nodos



# ADAPTACION DINAMICA

Como se mencionó, al iniciar los nodos, estos reciben el tamaño del clúster como parámetro inicial. Esto les permite configurar su estructura interna, incluyendo:

- Sus peers: Los nodos con los que deben comunicarse.
- Su rango de hash (token range): Para determinar qué datos les corresponden.
- Los nodos replicados: Aquellos responsables de almacenar copias de los datos.

## CAMBIO DE TAMAÑO

- Los nodos mantienen una referencia constante al tamaño del clúster.
- Si detectan un cambio en el tamaño del clúster, actualizan automáticamente su estructura interna, lo que incluye:
  - Recalcular su rango de hash y sus nodos replicados.
  - Notificar el nuevo tamaño a los demás nodos a través del mensaje `adapt_message`.

## REACCIÓN A ADAPT\_MESSAGE

1. Cuando un nodo recibe un mensaje `adapt_message`, verifica si efectivamente el tamaño del clúster ha cambiado.
2. Si hay un cambio:
  - a. Reenvía a los nodos relevantes la instrucción para:
    - i. Crear sus keyspaces y tablas correspondientes.
    - ii. Insertar los registros que ahora les pertenecen.
  - b. Una vez completado, elimina de su base de datos los registros que ya no están en su nuevo rango de hash.

**Este mecanismo dinámico asegura que el clúster pueda adaptarse a cambios en su tamaño sin intervención manual, manteniendo la consistencia y redistribuyendo los datos de manera eficiente.**





# REPLICACIÓN Y CONSISTENCIA

## CONSISTENCIA Y READ-REPAIR

Para soportar tanto la consistencia como el read-repair, se registra un timestamp (fecha y hora) cada vez que una operación modifica o agrega un conjunto de valores.

- En caso de que un quorum detecte valores diferentes entre nodos, se toma como válido el valor con el timestamp más reciente. Esto asegura que el sistema siempre priorice la versión más actual de los datos.

Este enfoque equilibra la respuesta rápida al cliente y la integridad de los datos entre nodos a lo largo del tiempo.

### READ-REPAIR

Independientemente del nivel de consistencia elegido, el sistema aplica una estrategia de read-repair para corregir posibles inconsistencias entre nodos. Una vez que se responde al cliente, el sistema (en segundo plano, con menor prioridad que las peticiones del cliente) envía la operación a los nodos restantes.

- Cuando todos los nodos terminan de procesar, el sistema consolida las respuestas.
- Si algún nodo tiene información inconsistente (claves desactualizadas o incorrectas), se le solicita que actualice su información, pero solo para las claves que le corresponden según el hash.

### CONSISTENCIA

Al ejecutar un query, el cliente puede definir el nivel de consistencia deseado, como ONE o QUORUM.

- ONE: La operación se considera completa cuando al menos un nodo confirma la acción, sin esperar la respuesta de otros nodos.
- QUORUM: La operación requiere que una mayoría de nodos confirmen antes de responder al cliente.

Esto define cuándo se envía una respuesta al cliente, pero no garantiza que todos los nodos tengan la misma información en ese momento, ya que tras la respuesta al cliente se manda a ejecutar a los nodos restantes.

### ESTRATEGIAS DE REPLICACIÓN

- 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



# CLÚSTER Y SEGURIDAD

## FUNCIONAMIENTO DE CLÚSTER

### PROTOCOLO DE COMUNICACIÓN

Se utiliza el protocolo nativo de Cassandra para la comunicación entre el cliente y los nodos del clúster. Este protocolo garantiza la compatibilidad con herramientas estándar y mejora la eficiencia mediante la serialización y deserialización optimizada de mensajes, lo que facilita la transferencia rápida y la legibilidad del código involucrado en la comunicación.

### PROTOCOLO GOSSIP

El protocolo Gossip es clave para el funcionamiento del clúster, ya que permite a los nodos compartir información sobre su estado y el de otros nodos. Este intercambio regular de datos asegura:

- Detección de fallos: Identifica rápidamente nodos inactivos o desconectados.
- Ejecución eficiente: La implementación de claves de hash optimiza el proceso de Gossip, permitiendo que los mensajes sean dirigidos adecuadamente y que el clúster se adapte a cambios dinámicos como fallos de nodos o nuevas adiciones.

### TOLERANCIA A FALLOS

El sistema está diseñado para tolerar fallos en uno o más nodos gracias a la replicación de datos. Mientras exista al menos un nodo con los datos requeridos y se cumpla el nivel de consistencia especificado, las operaciones pueden continuar sin interrupciones. Además, se implementa el mecanismo de **Read Repair**, que asegura la actualización de nodos desactualizados al finalizar una consulta SELECT. Cuando se realiza una lectura:

1. El nodo al que está conectado el cliente, verifica los datos con los nodos implicados.
2. Si detecta inconsistencias, envía una solicitud para que los nodos desactualizados actualicen sus valores.
3. Esto es especialmente útil cuando un nodo previamente apagado vuelve a estar disponible, ya que automáticamente recibe las actualizaciones necesarias, evitando la necesidad de realizar operaciones UPDATE o INSERT explícitas.



# CLÚSTER Y SEGURIDAD

## SEGURIDAD

### AUTENTICACIÓN Y AUTORIZACIÓN

Se configuraron mecanismos básicos de autenticación para restringir el acceso al clúster. Aplicamos una autentificación unica para cada cliente. Esta autenticación se llevo a cabo en el intercambio de mensajes de *authenticate* del protocolo Cassandra (más sobre esto en la la sección de protocolo...)

### ENCRIPTACIÓN DE DATOS EN TRÁNSITO

Se implementó la encriptación de datos en tránsito utilizando TLS (Transport Layer Security) y Rustls (una implementacion moderna de TLS), asegurando la privacidad de las comunicaciones entre cliente y nodos como tambien la comunicacion entre nodo - nodo. Utilizamos certificados para establecer una conexion TLS segura entre las partes involucradas

- **openssl.cnf:**  
Configura como se crean el certificado y la clave que rustls utilizara.  
Define parametros y opciones que se deben usar
- **server.key:**  
Contiene la clave privada del servidor (secreta, solo accesible por el servidor).  
La clave privada cifra y desifra datos, prueba que el *server.crt* es legitimo, para completar el el intercambio seguro de claves
- **server.crt:**  
Contiene el certificado publico asociado (compartido con los clientes)  
Este certificado demuestra a los clientes que el servidor es legitimo  
Este certificado es validado durante el handshake TLS por el cliente



# >>>> SIMULACION

## DOS CLUSTERS DE AVIONES

Los aviones en este programa son esencial durante toda la ejecución. Precisamos saber su ubicación, de donde vienen, hacia donde van y principalmente si se encuentran volando o ya en tierra en algún aeropuerto. Para poder manejar esto de la mejor manera los clasificamos en dos categorías: Aviones Volando y Aviones En Aeropuerto.

Estos clientes, van a conectarse y enviar su ubicación en tiempo real, para que los servidores puedan actualizar esos datos y poder colocarlos en la última posición dada de la interfaz gráfica.

### AVIONES VOLANDO

Los datos de aviones volando, se actualizan con su posición (latitud, longitud), velocidad, nivel de combustible, altitud y Aeropuertos de Partida y Llegada. Se generan en tiempo real mediante la simulación en el cliente, inicialmente encontrados en un archivo de formato json para una mejor lectura.

Ademas, utilizamos una CONSISTENCY ONE para las queries de consistencia al momento de actualizar los datos.

```
SELECT * FROM keyspace_tests.orders5 USING CONSISTENCY QUORUM WHERE order_id = 1;
```

### AVIONES EN AEROPUERTO

Para los aviones en aeropuerto, estos tiene información de su hora de partida y estado actual, como por ejemplo Embarcando o Atrasado. Del mismo modo, se simula una actualización de esto para que se actualice según avanza el tiempo.

A diferencia de los Aviones Volando, utilizamos una CONSISTENCY QUORUM para las queries.

```
SELECT * FROM keyspace1.aviones_volando USING CONSISTENCY QUORUM WHERE origin = 'EZE';
```

## THREADPOOL / POOL DE HILOS

Los thread pools son una técnica eficiente y fundamental para gestionar la concurrencia y optimizar el uso de recursos en aplicaciones como las que requiere el proyecto Aerolíneas Rústicas

Un thread pool es un conjunto de hilos (threads) que son pre-creados y reutilizados para ejecutar tareas. En lugar de crear un hilo nuevo cada vez que se necesita realizar una tarea concurrente, un thread pool permite gestionar y reutilizar estos recursos para evitar el overhead de creación y destrucción de hilos constantemente.

Esto es especialmente útil para aplicaciones multithreading que necesitan manejar varias tareas simultáneamente, como en el caso de simular estas dos categorías de Aviones en tiempo real para múltiples clientes.



# >>>> 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 envíamos 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 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

### 6. 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

### 7. Encriptacion Rustls y TLS

Utilizamos un sistema para almacenar y guardar los usuarios y contraseñas, y que estas no sean expuestas a los clientes.

Encriptamos la conexión End-to-End para que los datos sean protegidos y encriptados durante toda la comunicación. Además creamos archivos como open\_ssl.cnf para generar certificados para la autenticación y que IPs utilizaremos en la implementación



## 8. Loggs

Utilizamos la biblioteca `flexi_logger` para la gestión de logs.

En la función `init_logger`, configuramos los parámetros necesarios para registrar eventos en un archivo dentro del directorio `logs`, permitiendo guardar registros importantes, como mensajes de diferentes niveles (`info`, `warn`, `error`). La configuración de logs permite la creación de archivos personalizados con timestamps UTC, asegurando que el tiempo registrado sea consistente sin importar la ubicación geográfica.

Se implementan mensajes de los siguientes niveles:

- `info`: Información general.
- `warn`: Advertencias del sistema.
- `error`: Mensajes de error críticos.

Estos mensajes se almacenan en archivos `.log`, permitiendo el registro de todos los eventos enviados y recibidos por el nodo en el archivo `server.rs`.

## 9. Docker

Docker Compose se utiliza para orquestar y configurar múltiples contenedores Docker. Simplifica el despliegue y la administración de aplicaciones multicontenedor.

El `Dockerfile` define el proceso para construir una imagen Docker. Especifica todas las instrucciones necesarias, como la configuración del entorno, dependencias y comando principal a ejecutar.

El archivo `.dockerignore` excluye archivos y directorios innecesarios del contexto de construcción de la imagen. Funciona de manera similar a un `.gitignore`, ayudando a reducir el tamaño de la imagen y acelerar su construcción.

## 10. Docker Containers

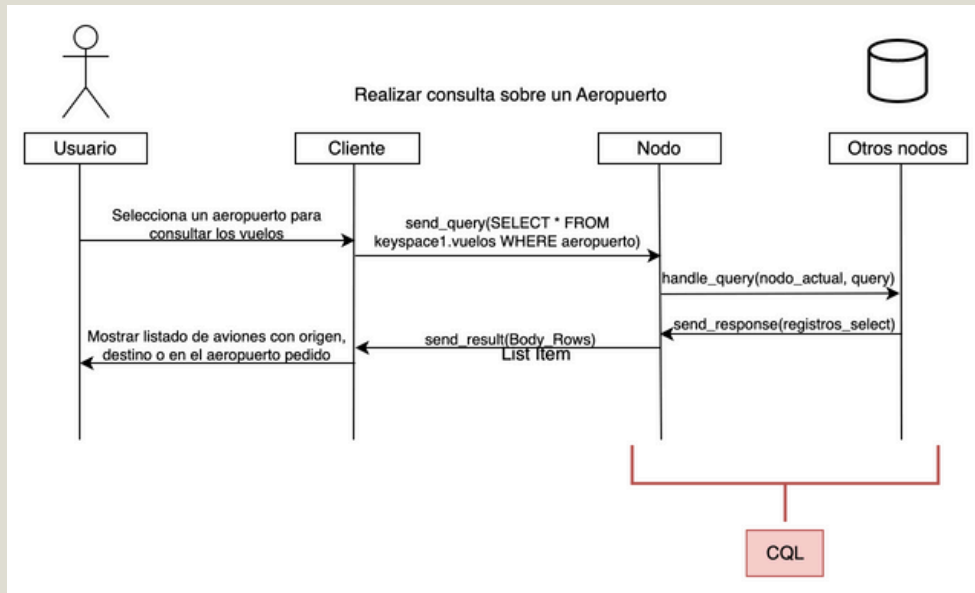
Creamos un contenedor para cada nodo encapsulan todo lo necesario para ejecutar una aplicación de manera aislada y portable.

A diferencia de las máquinas virtuales, los contenedores no incluyen un sistema operativo completo; comparten el kernel del host y son mucho más eficientes en términos de recursos.

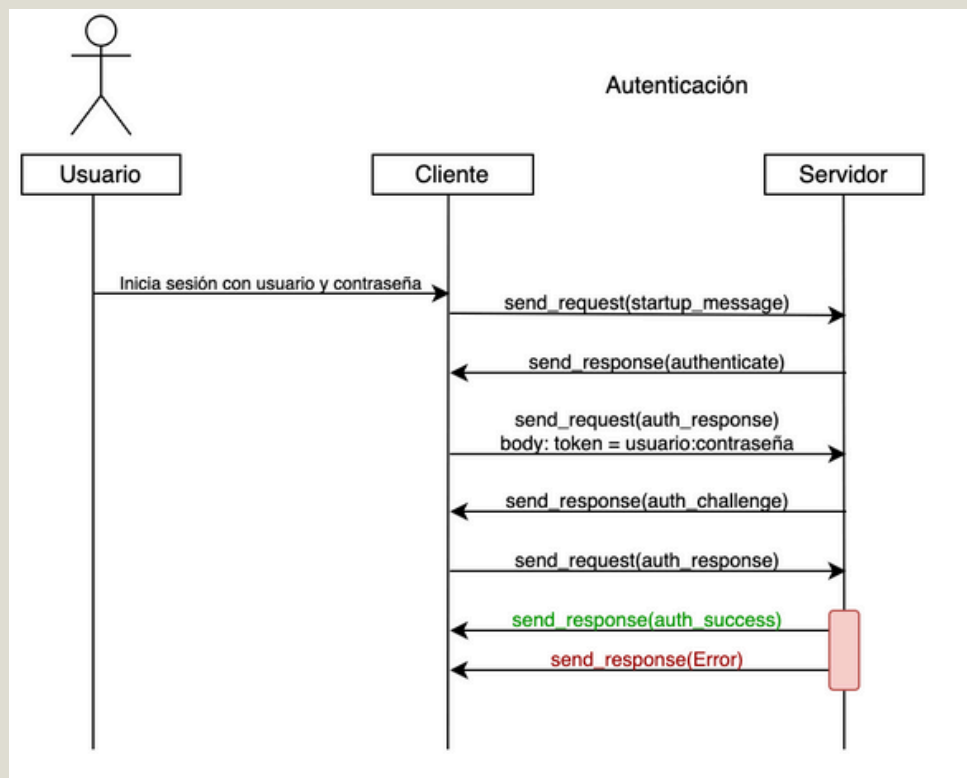
# DIAGRAMAS

## 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

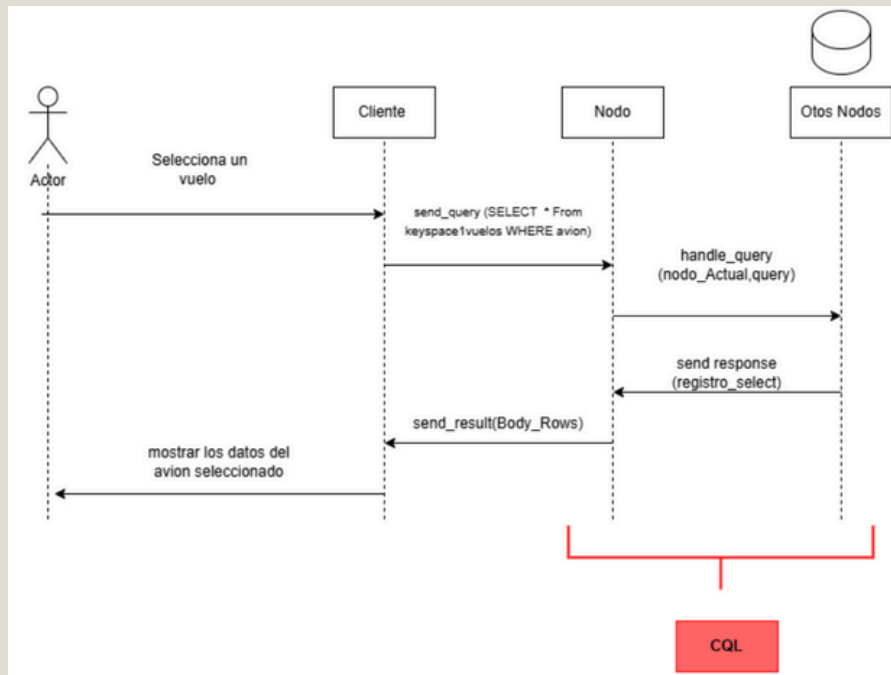


## AUTENTICACIÓN



## 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.

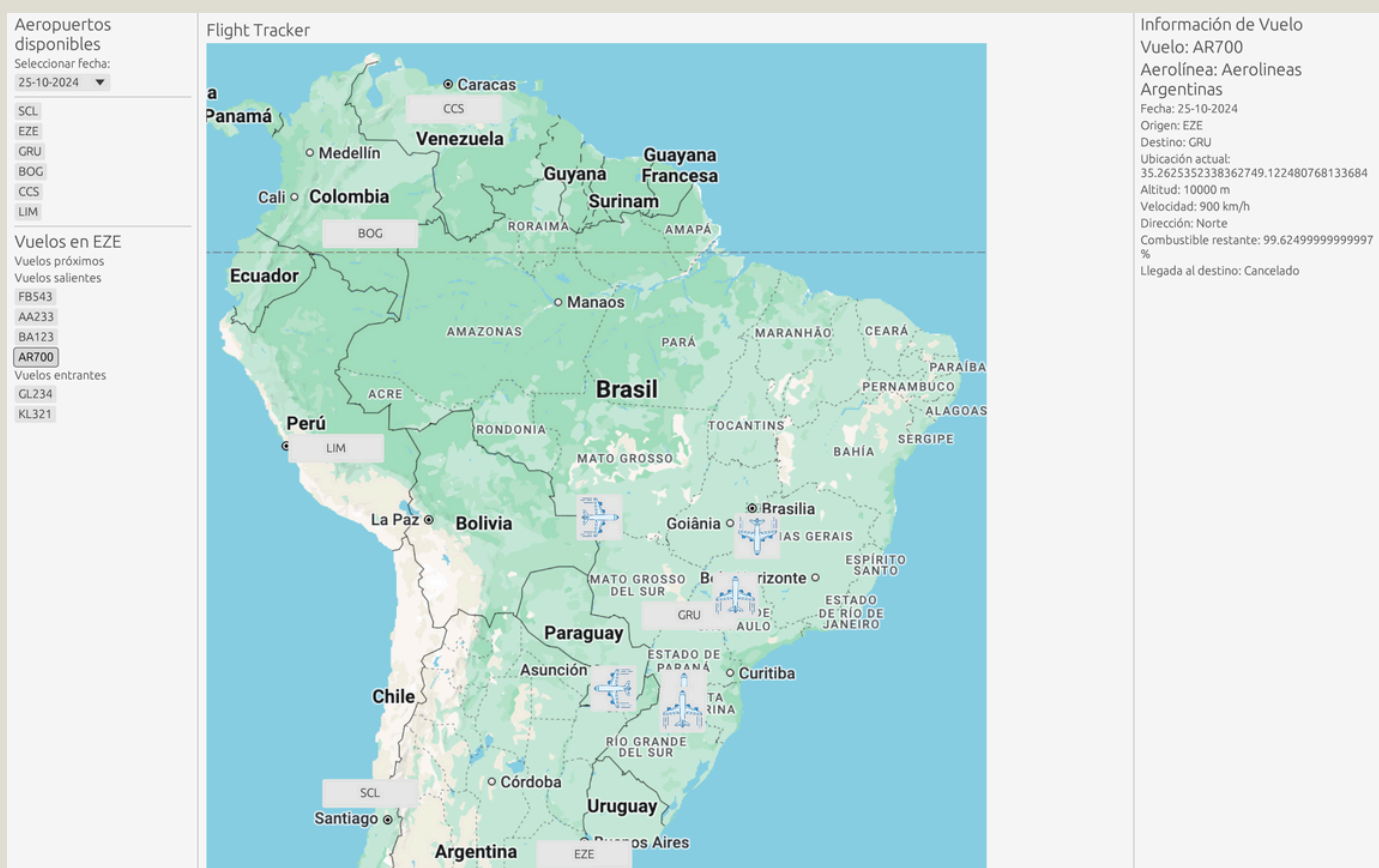


Imagen de la interfaz gráfica implementada



# CONCLUSIÓN

En esta segunda entrega y final, realizamos implementaciones importantes como el Read-Repair, Consistency Level, Clustering Keys, la encriptación de las conexiones, entre estos.

En primer lugar, las decisiones tomadas en el diseño, como la definición de claves de partición y clustering aseguran un mejor rendimiento y escalabilidad de las consultas. Por otro lado, al implementar niveles de consistencia fuertes y débiles según los requerimientos del sistema, demostramos la capacidad de balancear confiabilidad y eficiencia en una base de datos distribuida.

En el ámbito de la seguridad, aseguramos la autenticidad integridad y encriptación de los datos en tránsito. A su vez, realizamos implementación de pruebas unitarias y de integración para validar la funcionalidad del programa, minimizar los errores y asegurar el cumplimiento de los requerimientos no funcionales.

Continuando realizamos la creación de una interfaz gráfica interactiva y un simulador de vuelos multithread para simular el movimiento de los aviones y actualización de los datos.

Por último, realizamos una documentación, presentación y diagramas sobre lo abordado en el trabajo y nuestras diferencias en la implementación.