Taller de Programación I - Cátedra Deymonnaz FIUBA

# AEROLÍNEAS RÚSTICAS


**Facundo Andrés Calderan [110873]**

**Nicolás Khalil Chaia [110768]**

**Gonzalo Nicolas Crudo [110816]**

**Mariano Gabriel Merlinsky Camins [110446]**

# Índice

# Introducción

Se planteó la necesidad de implementar un sistema global de control de vuelos que permita gestionar eficientemente tanto vuelos nacionales como internacionales, con operaciones distribuidas en diversas ubicaciones alrededor del mundo. Para cumplir este propósito, se busca desarrollar una solución basada en una base de datos distribuida que ofrezca escalabilidad y tolerancia a fallos, garantizando la continuidad del servicio incluso en caso de fallas en uno o más nodos de la red.

El desarrollo se llevó a cabo utilizando el lenguaje de programación Rust, conocido por su enfoque en la seguridad y por ofrecer múltiples herramientas para facilitar la programación concurrente, características esenciales en la implementación de sistemas distribuidos de alto rendimiento.

# Objetivo

El objetivo del proyecto es diseñar e implementar un sistema de gestión de información para vuelos aeroportuarios, utilizando una base de datos distribuida compatible con Cassandra. Este sistema debe permitir el almacenamiento y la consulta eficiente de datos distribuidos entre múltiples nodos, asegurando tolerancia a fallos y un desempeño adecuado en un entorno distribuido.

# Módulos y Código

Para alcanzar los objetivos planteados, el sistema se estructuró en diversos módulos que permiten dividir las responsabilidades y garantizar un diseño escalable, mantenible y eficiente. Cada módulo está diseñado para cumplir una función específica dentro del sistema, desde la gestión de los datos distribuidos hasta la interacción con los nodos y la base de datos.

En esta sección se describen los módulos principales que componen la solución, junto con una explicación detallada de su funcionalidad y cómo interactúan entre sí.

## Errors

Este módulo consta de un único archivo en el que se define un tipo de dato enum llamado Error Types. Este tipo de dato se utiliza para encapsular y manejar los errores que puedan ocurrir durante la ejecución del programa.

Cuenta con dos atributos principales:

- **Código**: Identifica el tipo de error de manera única.
- **Mensaje**: Proporciona una descripción personalizada del error, lo que facilita el diagnóstico y resolución del problema.

Gracias a esta estructura, es posible personalizar los errores y rastrear de forma más sencilla el lugar exacto donde ocurrió un fallo en el sistema, mejorando la depuración.

## Nodes

Este módulo consta de un único archivo llamado node, el cual define la estructura y funcionalidad de los nodos que interactúan en el programa. Su principal objetivo es gestionar la

configuración e inicialización de los nodos desde la consola, donde se reciben dos parámetros fundamentales:

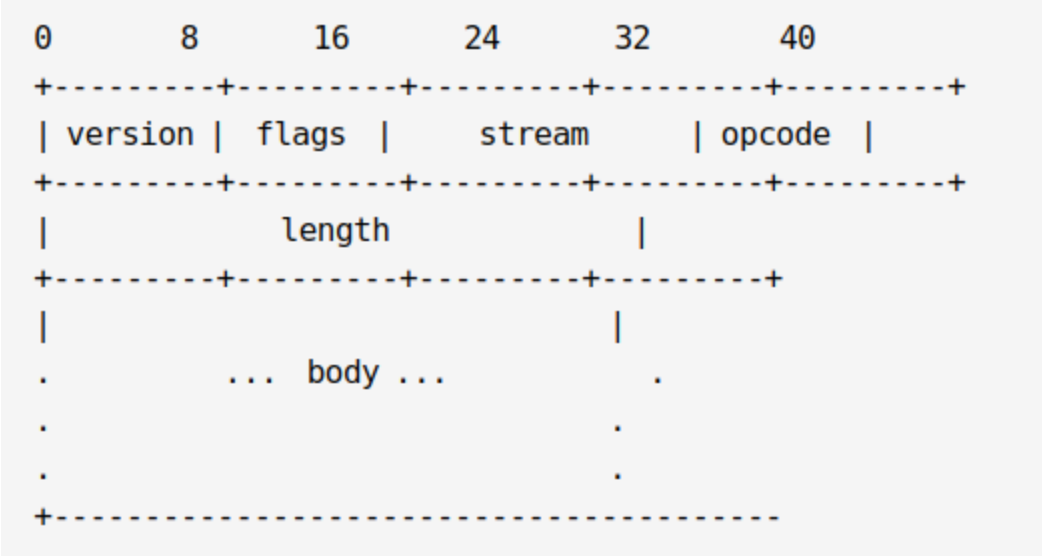
- Un puerto para la comunicación entre nodos.
- Un puerto para la comunicación con el cliente.

El proceso inicia con la validación de los parámetros proporcionados. Una vez que los valores ingresados son verificados, el nodo comienza a ejecutarse, asumiendo su rol dentro del sistema distribuido.

## Protocol

Este módulo encapsula todo lo que tiene que ver con el protocolo nativo de Cassandra, está dividido en 4 subcarpetas y un archivo, cada cual con diferentes tareas. Antes de abordar cada una, es importante repasar la estructura de los mensajes del protocolo:

Cada frame está definido como:



Cada frame tiene un header de tamaño fijo (9 bytes) y un body de longitud variable. El header es una estructura que contiene flags, un opcode y una versión. El contenido del body depende del valor del opcode header.

Entonces, estas son las responsabilidades de cada carpeta:

**frames_header:** Contiene varios archivos. Uno para cada apartado del header (opcode, versión y flags), en estos archivos se definen enums que contienen lo que cada atributo requiere, y uno que define al header como estructura. En este último, se crea un header con los atributos ya mencionados y se definen una serie de primitivas que van desde retornar valores de los atributos del mismo hasta convertir el contenido en binario.

**protocol_body:** También contiene un archivo para cada apartado del body:

- compression: Es un enum que representa las diferentes maneras de comprimir los datos, los algoritmos soportados son “LZ4” y “Snappy” junto con la implementación para poder efectuar esta compresión.
- event_kind: Representa el body para los mensajes de diferentes tipos de eventos que pueden ocurrir durante la ejecución.
- query_flags: Es un enum que representa los diferentes flags que pueden tener las queries.
- result_kind: Es un enum que representa los resultados de haber solicitado una query.
- schema_change: Es un enum que representa los diferentes tipos de cambios en el esquema que pueden tener lugar durante la ejecución.
- status_node: Es un enum que representa el estado de un nodo, ya sea caído (Down) o funcionando (Up).
- topology_change: Es un enum que representa cambios en la topología del cluster, cosas como por ejemplo, la eliminación/agregación de un nodo.

**protocol_notations:** Cuenta con los siguientes archivos:

- bytes_map: Es un type que define a un Map que tiene como clave a un String y como valores a una tupla de i32, Value. Donde Value es un enum que será descrito más adelante.
- consistency: Es un enum que define los diferentes tipos de niveles de consistencia soportados.
- flags_row: Es un enum que es utilizado para el momento de devolver filas de la base de datos.
- protocol_body_writer: Es una estructura que implementa al body. Es una parte esencial del programa puesto que es la que construye este apartado tan importante de los mensajes. Cuenta con primitivas para escribir cosas en él y también para devolver parte de su información, como por ejemplo la longitud. En este archivo es donde se fueron utilizando todas las demás cosas definidas anteriormente.
- result: Es un enum que representa los diferentes resultados que pueden ser recibidos luego de enviar una query.
- value: Es un enum que representa los diferentes tipos de valores que pueden ser devueltos al momento de leer bytes de una protocol notation.

**query_parser:** Esta subcarpeta se encarga de todo lo que es el parseo de las queries. Cuenta con diferentes estructuras como por ejemplo Clause y Relation las cuales son utilizadas para resolver las cláusulas Where de una consulta. Contiene un archivo para cada operación soportada (Creación de tablas, Actualización de filas, Seleccionar filas, Insertar filas, Eliminar filas).

**protocol_writer:** Este archivo combina todo lo dicho anteriormente, genera un mensaje construyendo paso a paso. Primero crea un header y luego el body. Retornando el mensaje completo.

## Receiver

El propósito principal del módulo es una vez recibido el mensaje (se encuentra en binario), traducido de modo tal que pueda ser entendido por el programa.

Para esto se tiene un enum llamado Message, que dentro tiene otro enum que puede ser ResponseMessage para representar los mensajes de respuesta y RequestMessage para representar las peticiones.

Toda la lógica de la "traducción" se encuentra en el archivo receiver_impl el cual va leyendo los bytes y construye el mensaje final para que luego sea procesado por los nodos.

## Server

Para hablar del servidor hay que empezar definiendo su unidad fundamental, el nodo. Los nodos son unidades que se encargan del procesamiento y almacenamiento de datos trabajando en conjunto y comunicándose continuamente. En la implementación se utilizan 8 (aunque podrían crearse más sin problema).

Cuando un nodo se crea, es necesario asignarle 2 puertos, uno para conexiones del cliente y otro para la comunicación interna.  Cuando un nodo quiere conectarse con otro, se genera un thread (único para cada una) que se encargará de la comunicación entre ellos hasta que alguno se caiga. De este modo, cada nodo tiene 7 threads (además de uno principal) para manejar este intercambio. De ser necesario se utilizan "channels" para poder conectar un thread con el principal, estos son unidireccionales. Al momento de conectarse con el cliente, se generará un thread y por cada mensaje del mismo, se crea otro para poder ejecutar varias cosas a la vez.

Al momento de inicializarlo, si no hay un esquema ya definido se define y, en caso contrario, directamente se lo levanta y utiliza el esquema actual.

Para la comunicación interna se utiliza el protocolo gossip, el mismo cuenta de tres mensajer:

- Syn
- Ack
- Ack2

Además se definió una estructura llamada *Gossiper,* posee el estado de todos los nodos, las conexiones a los distintos threads de cada uno, los nodos "conocidos" y el HashRing. Este último es un árbol binario ordenado, que contiene Vnodes.  De este modo, en lugar de hashear la IP e insertarla en el Hash Ring, se hashea (con sutiles diferencias) e inserta una cantidad de veces preestablecida (32), esto trae como beneficio una distribución más equitativa del cluster.

El almacenamiento de datos se organiza en dos partes principales: una memtable y una SSTable. La memtable reside en memoria y tiene un límite definido de filas. Una vez alcanzado este límite, se inicia un proceso conocido como *flush*, que transfiere todas las filas de la memtable al disco, almacenándose en la SSTable. Durante este proceso, si hay datos duplicados, se conserva aquel que fue insertado más recientemente.

## UI

La interfaz gráfica está diseñada para visualizar toda la información relacionada con los aeropuertos y sus vuelos. La ubicación inicial será la facultad, y desde ese punto podremos dirigirnos al aeropuerto que deseemos. Al hacer clic en un aeropuerto, se desplegará un apartado donde podremos seleccionar una fecha para buscar vuelos. En caso de que haya vuelos disponibles, se marcarán con una línea roja si son de arribo, y verde en caso contrario.

Si se decide simular un vuelo en ese momento, se agregará una nueva línea, junto con un punto negro que representará el avión, el cual avanzará con el paso del tiempo. Al hacer clic sobre el avión, podremos visualizar toda la información relevante, como su posición exacta, nivel de combustible, velocidad, altitud, entre otros datos.

Además, en la parte inferior derecha de la interfaz habrá una pestaña que permitirá modificar el estado del vuelo (retrasado o a tiempo), ejecutando internamente una actualización en el sistema.

## Simulador

Este apartado corresponde a la aplicación que sirve para simular vuelos. Su funcionamiento es simple, le pide los parámetros del vuelo al usuario (id, destino, origen, fecha de salida y llegada, combustible), valida que los datos sean correctos y de serlo, utilizando un pool de threads, simula el vuelo realizando actualizaciones cada 5 segundos.

## Encriptación

Se utilizó la librería nativeTLS donde se podrá envolver una conexión TCP en una TLS para la comunicación. El servidor creará un certificado el cual verificará si los clientes que deseen conectarse cumplen con este, de ser así se establece la conexión.

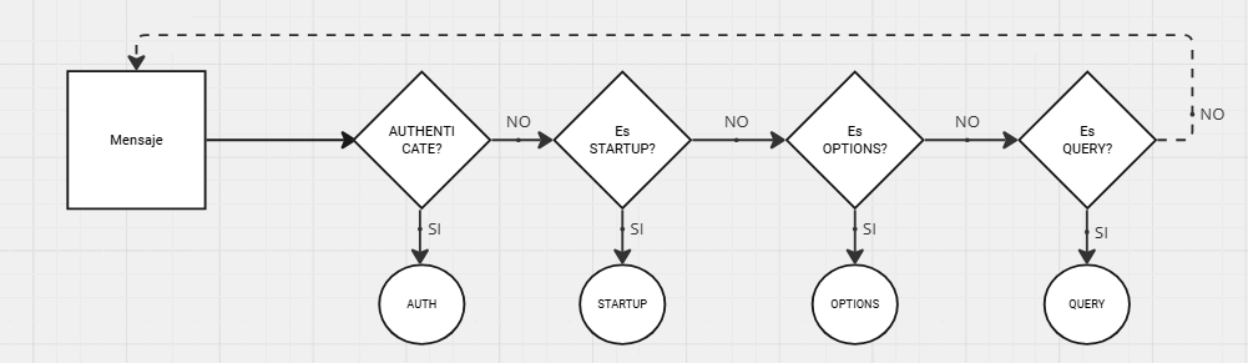
## Más información

Para obtener más información detallada sobre los módulos y cómo se relacionan entre sí, se recomienda consultar la documentación del proyecto, recordar que desde la consola ejecutando el siguiente comando se puede ver: “cargo doc --open”.

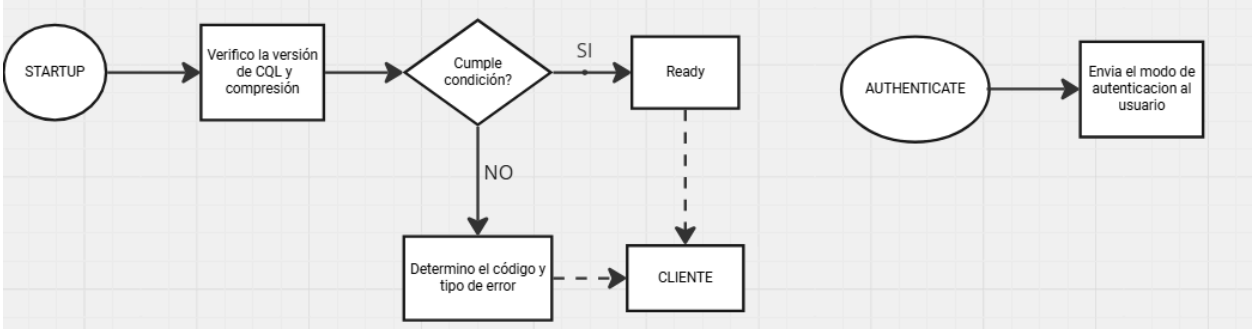
En ella se proporciona una descripción exhaustiva de todos los enum, estructuras y funciones implementadas.

# Diagramas

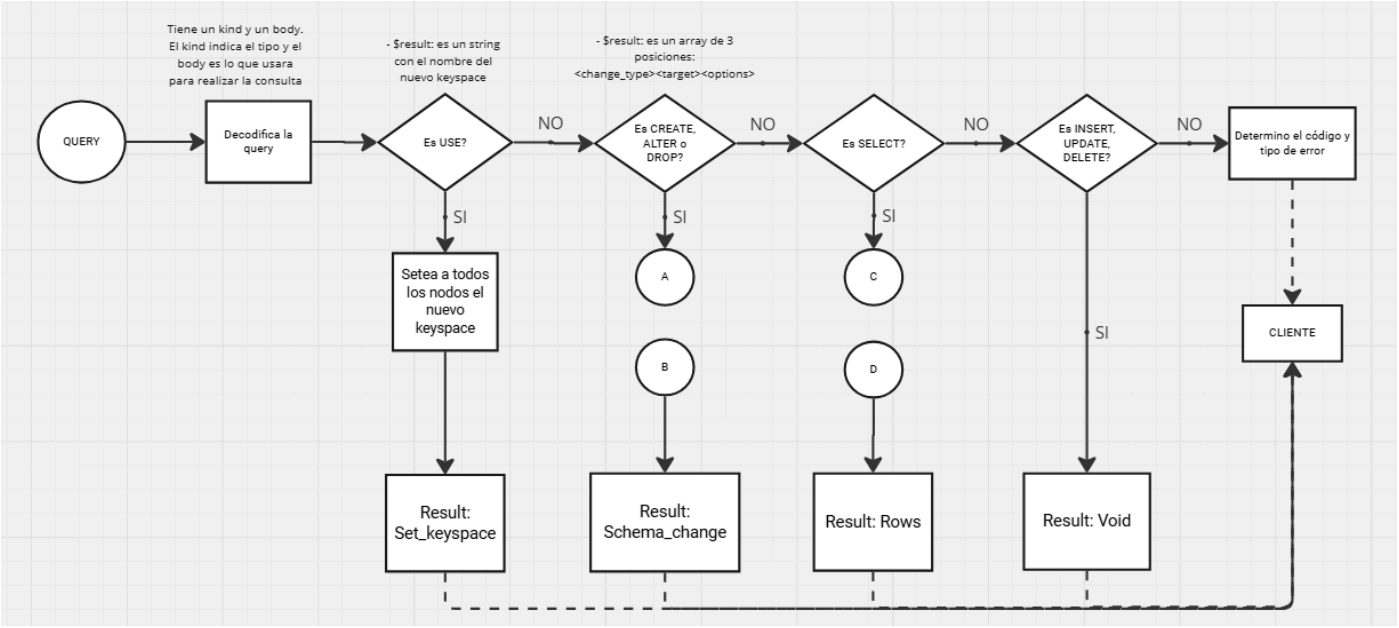
Durante la cursada y en la etapa de desarrollo de la consigna final, utilizamos a Miro como la herramienta para poder hacer los diagramas, este fue el board. Estos son los principales:



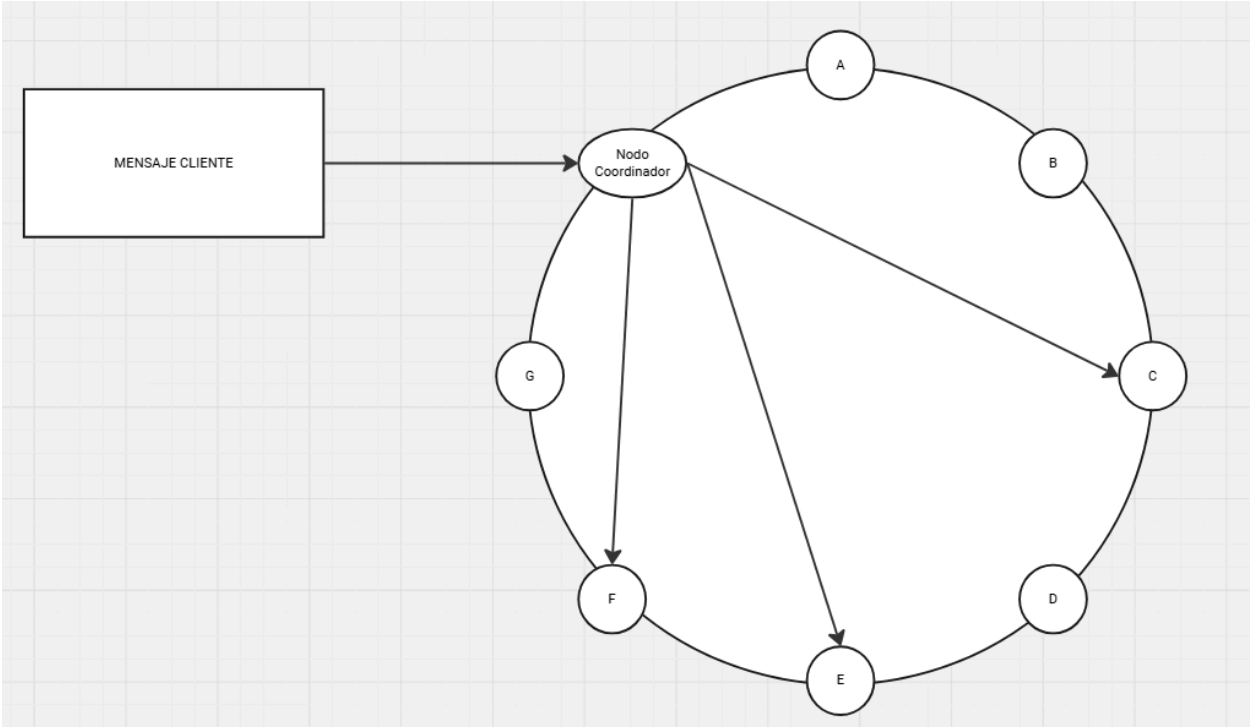

Este diagrama representa como es el proceso de recepción de mensajes, verificando primero si es de algún tipo soportado y luego operando en base al mismo.



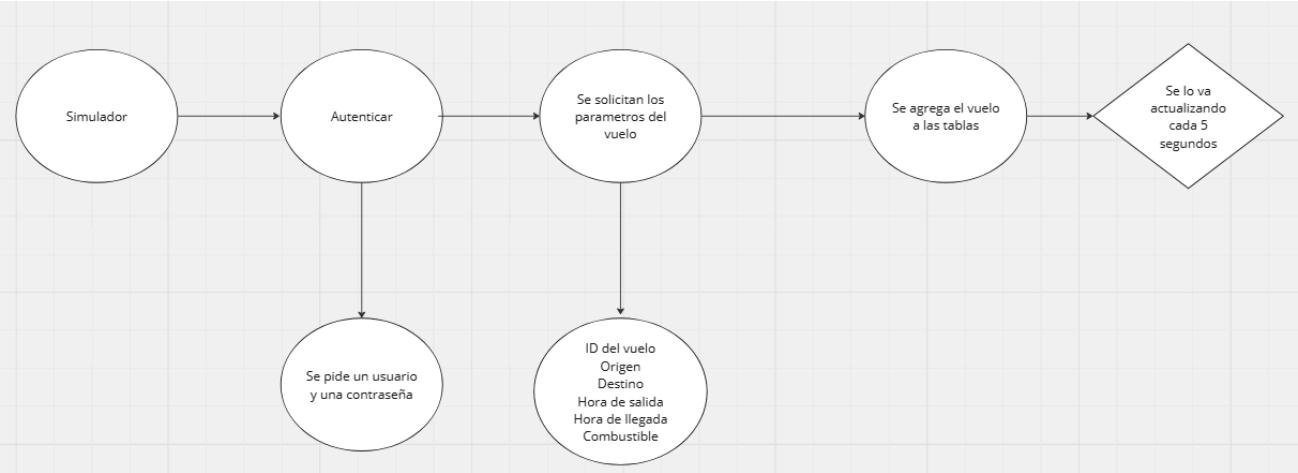

Este diagrama representa el proceso de cómo se inicializa el programa, enviando el mensaje de STARTUP y también el proceso de autenticación.



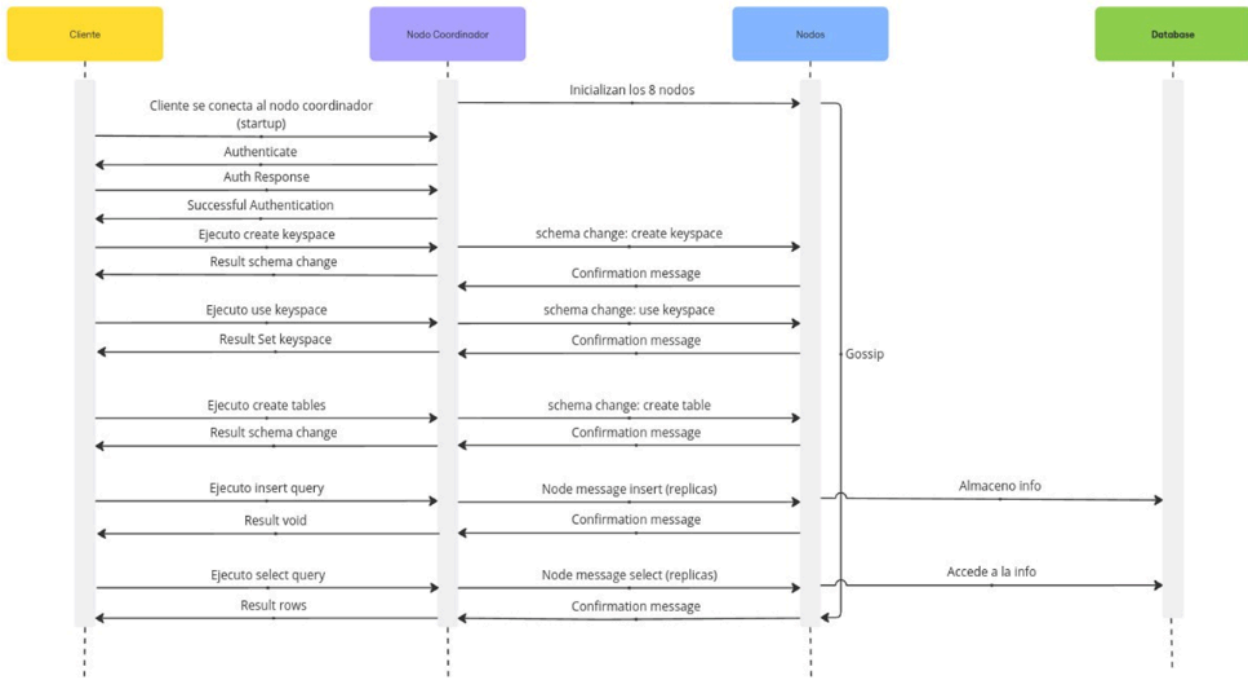

Este diagrama representa cómo se procesan y ejecutan las consultas de CQL.



Este diagrama representa como una vez recibido el mensaje, el nodo coordinador se lo envía al correspondiente.




Este diagrama representa el funcionamiento del simulador.




Este es un diagrama de secuencia que describe el flujo de comunicación y las acciones entre distintos componentes del sistema

# Entrega final

El objetivo es implementar la aplicación en un entorno basado en contenedores de Docker, la inclusión de un cluster dinámico que permita poder añadir y quitar nodos y por último, un "*Logger*" que permita visualizar el comportamiento del nodo durante la ejecución.

Para la implementación del servidor distribuido se utilizó Docker, una plataforma de software que permite crear, implementar y ejecutar aplicaciones en contenedores, empaquetando el código junto con sus dependencias, bibliotecas y configuraciones necesarias para su funcionamiento.

El enfoque adoptado se basó en la creación de un Dockerfile dividido en dos etapas principales:

1. **Buildeo de la imagen:** En esta etapa, se compila el binario de Rust utilizando *cargo build*. Antes de realizar esta acción, se copian las dependencias definidas en los archivos *Cargo.toml* y *Cargo.lock* al contenedor.
2. **Optimización y release de la imagen:** La segunda etapa genera una imagen optimizada para la ejecución del binario, eliminando archivos temporales, dependencias innecesarias y cachés. Esto reduce significativamente el tamaño de la imagen y asegura que sólo se incluyan los archivos estrictamente necesarios, como certificados o configuraciones específicas en formato JSON.

Para hacer funcionar el cluster dinámico, era necesario tener una terminal para así poder añadir o quitar nodos del cluster, por lo que en este archivo se instaló Xterms con el fin de interactuar con el nodo.  Además, se definieron variables de entorno para configurar los puertos de comunicación externa entre contenedores y clientes. Esto replica el comportamiento previo del servidor, que utilizaba una dirección interna para la comunicación entre nodos y una externa para los clientes. Las variables de entorno se especifican en cada servicio del *docker-compose*.

Para habilitar la interacción entre los contenedores, se configuró una red (Network) compartida, permitiendo que los nodos puedan comunicarse. Esta red es fundamental para el correcto funcionamiento del clúster distribuido.

Previo a ejecutar el archivo *docker-compose,* se buildea la imagen por consola; en este archivo, se tiene la configuración básica de cada servicio. En cada contenedor hay una referencia a la imagen recién creada, comandos a ejecutar, los puertos a usar, la red a la que se conectará y su nombre.

Finalmente, se desarrollaron scripts en Bash (.sh) para inicializar el cluster y un nodo externo, automatizando el proceso y simplificando su gestión.

Cuando se añade un nuevo nodo, es necesario que la información existente sea repartida. Para hacer esa búsqueda se debe tener en cuenta el replication factor atado a los datos. Por eso cuando un nodo es añadido, cada nodo se da cuenta y comienza a calcular las particiones que debería pasarle al nuevo. Revisa todos sus keyspaces y tablas, y le transmite toda la información. Solamente le transmiten la información los nodos que son últimas réplicas del dato, para también en el proceso eliminarlo y quedarnos con un estado de los datos coherente.

El logger fue diseñado para registrar toda la información de los nodos. Para ello, cada mensaje que el nodo desee imprimir será almacenado en un archivo de tipo log. Esto permite ejecutar el visualizar todos los eventos registrados por ese container.

# Conclusiones

Con el trabajo ya finalizado, hemos podido sacar conclusiones en diferentes ámbitos. En primer lugar, queremos destacar que esta fue la primera vez que nos acercamos de forma práctica a lo que realmente implicaría ejercer nuestra profesión. El desafío de levantar un proyecto grande desde cero, teniendo que tomar decisiones constantemente, fue una experiencia significativa que nos permitió comenzar a experimentar lo que es trabajar como ingenieros informáticos.

Además, este proyecto nos brindó la oportunidad de adquirir conocimientos valiosos, no solo a través del desarrollo del sistema, sino también gracias al enfoque de la materia. Temas como servidores, sockets y protocolos, que desde un principio habían despertado nuestro interés pero nunca habíamos explorado en profundidad, finalmente los comprendimos a fondo. Esta experiencia no solo nos permitió aplicarlos en un contexto práctico, sino que también representó un gran logro en nuestro aprendizaje.

Por último, cabe mencionar que adquirimos un conocimiento sustancial sobre bases de datos distribuidas, en particular sobre Cassandra. Logramos entender cómo funcionan y cómo integrarlas en un sistema para garantizar escalabilidad y rendimiento, lo cual fue un aspecto crucial del proyecto. Esto nos permitió ampliar nuestra visión sobre tecnologías modernas y cómo se aplican en el desarrollo de sistemas distribuidos.

# Bibliografía


https://cassandra.apache.org/doc/latest/cassandra/developing/cql/index.html

https://cassandra.apache.org/_/native_protocol.html

https://cassandra.apache.org/doc/latest/cassandra/architecture/dynamo.html

https://cassandra.apache.org/doc/stable/cassandra/operating/read_repair.html

https://www.youtube.com/watch?v=69pvhO6mK_o&list=PL2g2h-wyI4Spf5rzSmesewHpXYVnyQ2TS

https://www.youtube.com/watch?v=wOyQlbFM1Uk&list=PL2g2h-wyI4SqzmsHQZaYcK9uDmPuihPz2

https://doc.rust-lang.org/book/ch20-02-multithreaded.html#improving-throughput-with-a-thread-pool