Trabajo Práctico - Aerolíneas Rústicas

[75.42/95.09] Taller de Programación
2C 2024

Grupo: *Los Aldeanos Panas*

Integrantes:

Nombre	**Padrón**
Franco Lighterman Reismann	106714
Matias Daniel Mendiola Escalante	110379
Francisco Antonio Pereyra	105666
Victor Daniel Cipriano Sequera	106593

Introducción

En este trabajo práctico, nuestro objetivo era implementar un sistema de control de vuelos global, donde se pueden observar los aeropuertos y los vuelos en tiempo real. Para lograr eso, se debía construir un motor de base de datos distribuida que sea escalable y tolerante a fallos, compatible con Cassandra, es decir, que soporte el lenguaje de consultas CQL (Cassandra Query Language) y también su protocolo de comunicación nativo.

Desafíos de Diseño

Durante el proyecto enfrentamos distintos desafíos que quisimos destacar:

- Tuvimos que adaptar el protocolo nativo de Cassandra al lenguaje de programación Rust, lo que significó un profundo aprendizaje de dicho lenguaje y una extensa lectura de toda la documentación de dicho protocolo.

- Se llevó a cabo un exhaustivo parseo de toda la sintaxis de CQL para poder realizar su consiguiente procesamiento en nuestro proyecto.

- Tuvimos que definir y separar el proyecto en binarios diferenciados ya que habían muchas responsabilidades a cubrir, donde no alcanzaba con uno solo. Estos son: Nodo, Cliente, Interfaz de Usuario (GUI) y Simulador de Vuelos.

- Lo más central del trabajo, la implementación concreta de las conexiones cliente-nodo y nodo-nodo, donde pusimos a prueba nuestro entendimiento de los temas más importantes de la materia, como threads, sockets, concurrencia, etc.

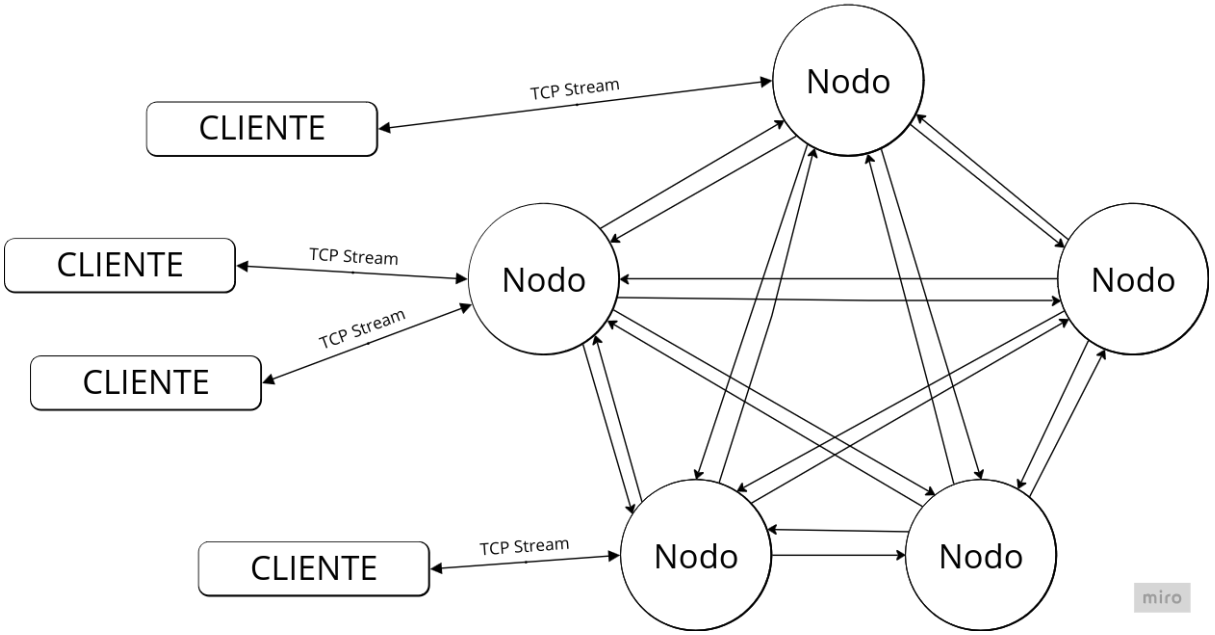
Clúster

El clúster es un “anillo” de nodos que es la pieza fundamental del motor de base de datos distribuida, es solamente un concepto porque cada nodo es independiente de los demás, es dónde se conecta el cliente. Cuando este último quiera conectarse, alguno de los nodos del clúster recibirá su conexión y actuará en consecuencia.

Conexiones con el Clúster

Todas las conexiones son bilaterales mediante TCP Streams, o sea, un cliente y un nodo se comunican entre sí por el mismo canal, de la misma manera entre nodos.
La diferencia entre las conexiones entre cliente-nodo y nodo-nodo, es que cada tipo tiene un puerto distinto para poder diferenciarlos.
Desde el lado del cliente, este va a querer conectarse al clúster, e internamente desde el clúster se elegirá algún nodo para responder de manera aleatoria.

Clúster



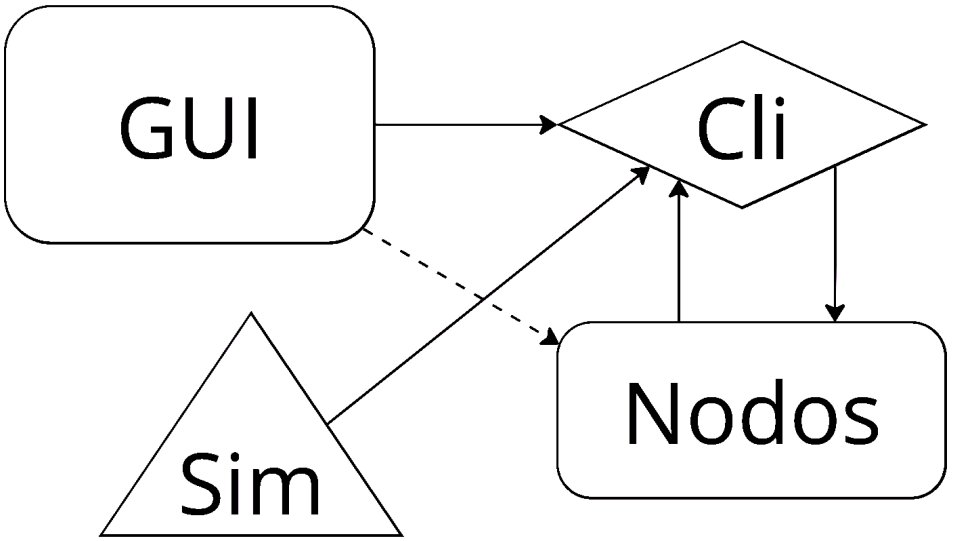
La comunicación cliente-nodo se encuentra encriptada para mayor seguridad, la comunicación nodo-nodo no es necesario ya que es interna de la propia base de datos.

Lidiar con múltiples conexiones de clientes

Ya que es ineficiente resolver de a una tarea por vez, era necesario poder soportar más de una conexión por vez y poder procesarlas en paralelo. Si bien el TCP Stream asegura que errores no hay, sí crea “una cola” de tareas, por lo que por sí sólo no puede hacer varias tareas a la vez, asi que se crea un hilo por cada cliente que ingresa, de esta manera podemos soportar múltiples clientes simultáneos por nodo.

Arquitectura del proyecto

Decidimos partir el proyecto en diferentes partes, tal que las dependencias que sólo se usan para una parte, no interfieran con las demás, y de otra manera impedir o empeorar el *performance*; además de no incluir dependencias innecesarias.



Binarios o Modularización

Nodos

Los nodos para iniciarse deben contar con un archivo previamente creado, donde se especifican ID e IP de cada uno. Actualmente la cantidad de nodos es de 5, y cada uno se inicia en una consola diferente.
El almacenamiento de las tablas está particionado entre los nodos para optimizar la carga de cada uno, se define un rango de valores para cada nodo según un valor de hashing que se obtiene de la *partition key* de la tabla, lo que asegura que la carga sea equitativa.

Cada nodo va a crear 4 hilos, donde cada uno se encarga de lo siguiente:

- Escuchar conexiones públicas desde los clientes y enviar su respuesta
- Escuchar conexiones privadas desde otros nodos y enviar su respuesta si corresponde (algunos mensajes no necesitan respuesta)
- Actualizar periódicamente el *HeartBeat* del nodo y serializar su metadata para poder ser consistente entre distintas ejecuciones.
- Ejecutar periódicamente rondas de *Gossip* entre los nodos.

Cuando un cliente se conecta y uno de los 5 nodos recibe la conexión, este nodo se denomina *nodo coordinador*, será el responsable de procesar la consulta y determinar cuál nodo le corresponde procesarla, ya que puede ocurrir que el nodo coordinador no contenga los datos necesarios que pide el cliente, o no le corresponde editar datos que no pertenecen a su rango de *hashing*.

Mensajes entre nodos

La comunicación entre nodos está hecha en base a unos mensajes previamente estructurados que contienen toda la información necesaria para procesarse, esto nos permitió tener un mayor control sobre el tráfico de mensajes y poder tener diferenciado cada proceso a realizarse.
Si un nodo recibe un mensaje y no responde, se asumirá que está apagado y se actualizará su estado a *Offline*, luego este cambio se propagará al resto de nodos automáticamente mediante *Gossip*, los demás nodos al saber esto, no le consultarán a ese nodo hasta que se sepa que fue iniciado nuevamente.

Gossip

Para Gossip asignamos arbitrariamente un peso a los nodos, eligiendo a 1 de los 5 con el triple de probabilidades de ser elegido para iniciar una ronda de Gossip, o sea el nodo semilla, este mismo nodo al ser iniciado es llenado con los estados de todos los nodos conectados previamente, para conocerlos, así que es necesario que éste nodo semilla sea iniciado último.
En cada iteración del hilo correspondiente, se elige por las probabilidades alguno de los 5 nodos para iniciar una ronda, y se comunicará con otros 3 nodos por vez.

Endpoint State

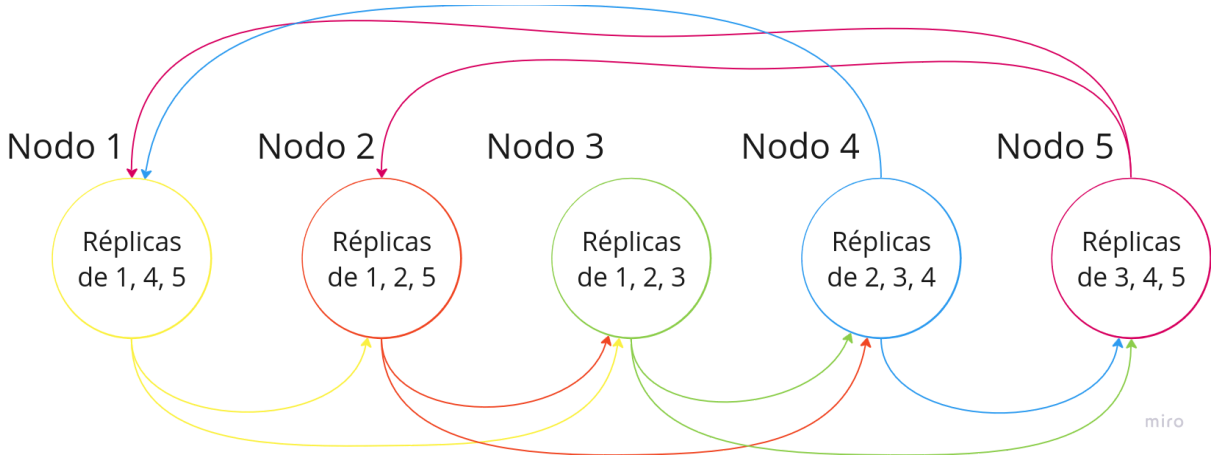
El endpoint state del nodo contiene información sobre el estado del nodo, por un lado tiene la IP y el estado actual del nodo (Normal, Bootstrap u Offline), por otro lado tiene el estado del heartbeat.
El heartbeat del nodo es lo que se comparte entre los nodos durante las rondas de gossip, para poder enterarse cuando un nodo deja de funcionar y viceversa. Contiene dos campos importantes:

- Generación: Es un timestamp del momento de creación del nodo.
- Versión: Es el tiempo en segundos desde la creación del nodo, incrementa cada segundo.

Estos dos campos ayudan a poder comparar si un estado es más antiguo o no que el que se comparte, para que todos los nodos siempre tengan la información más reciente de los demás nodos.

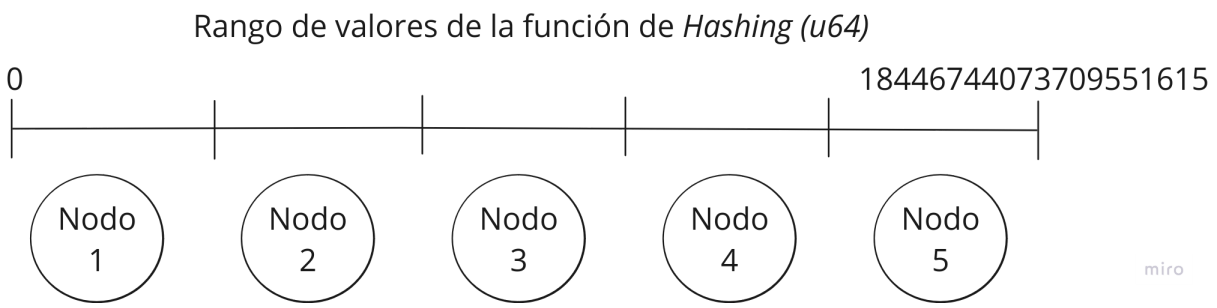
Replication Factor

Para la replicación, cada nodo va a replicar su data en n nodos dependiendo de la cantidad de replicación definida. De esta manera, dentro del almacenamiento de cada nodo, cada tabla va a tener un archivo diferenciado para poder identificar de qué nodo es la data que contiene. Para nuestro proyecto decidimos replicación de 3.
Por ejemplo, el nodo 1 tendrá tablas del estilo: *mytable_1.csv*, *mytable_4.csv*, *mytable_5*.csv. O sea, su data más 2 réplicas de otros nodos.



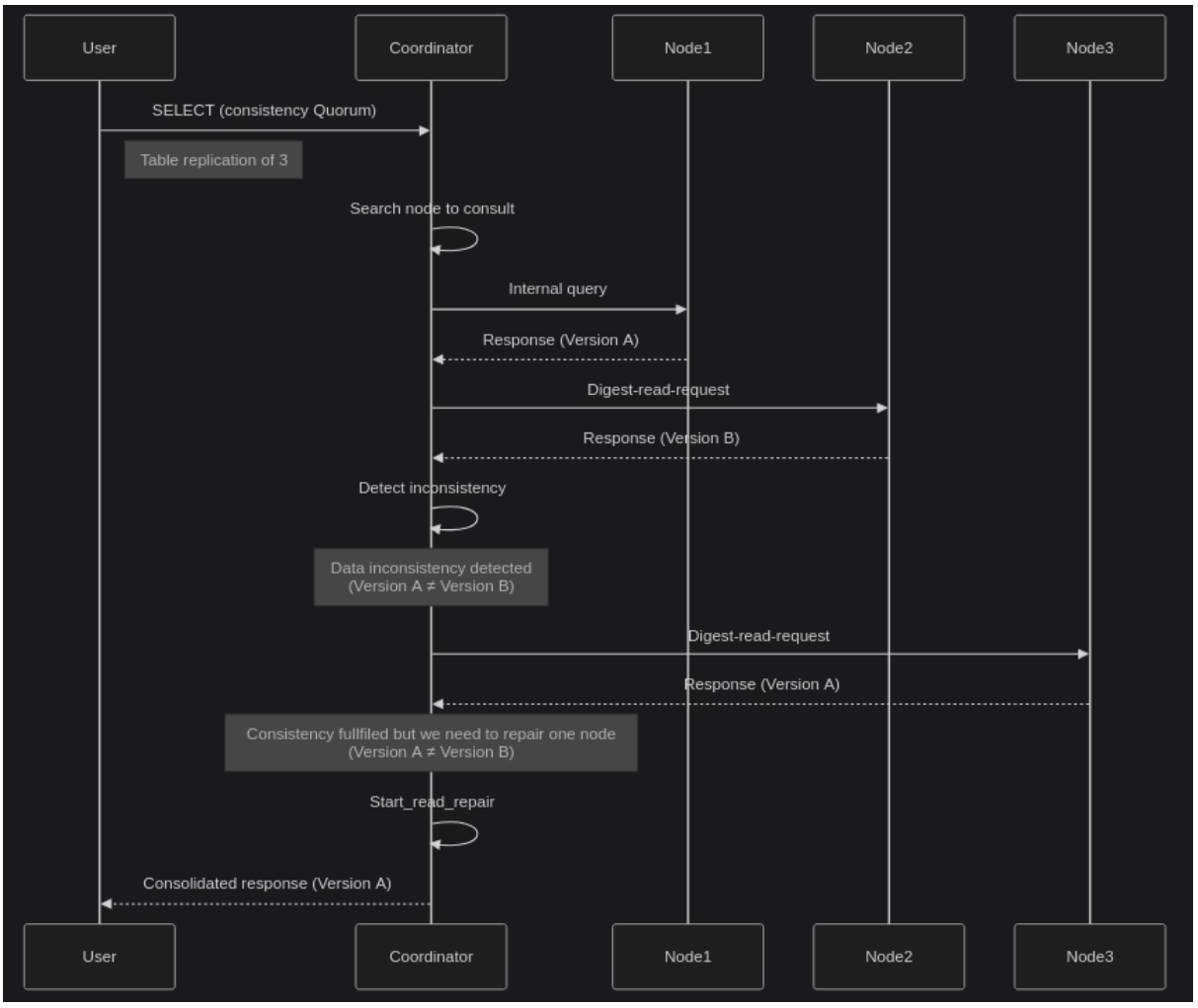
Hashing y Particiones

Para particionar las tablas, decidimos dividir en n subdivisiones, según la cantidad de nodos, el rango de la función de hashing que utilizamos, y a cada nodo asignarle una de esas subdivisiones. Esto hará que la distribución de carga sea lo más equitativa posible.
A cada fila nueva se le va a hacer un hash al valor de la columna que corresponda a la columna de la *partition key* de la tabla, según el resultado, se asignará esa fila al nodo correspondiente.



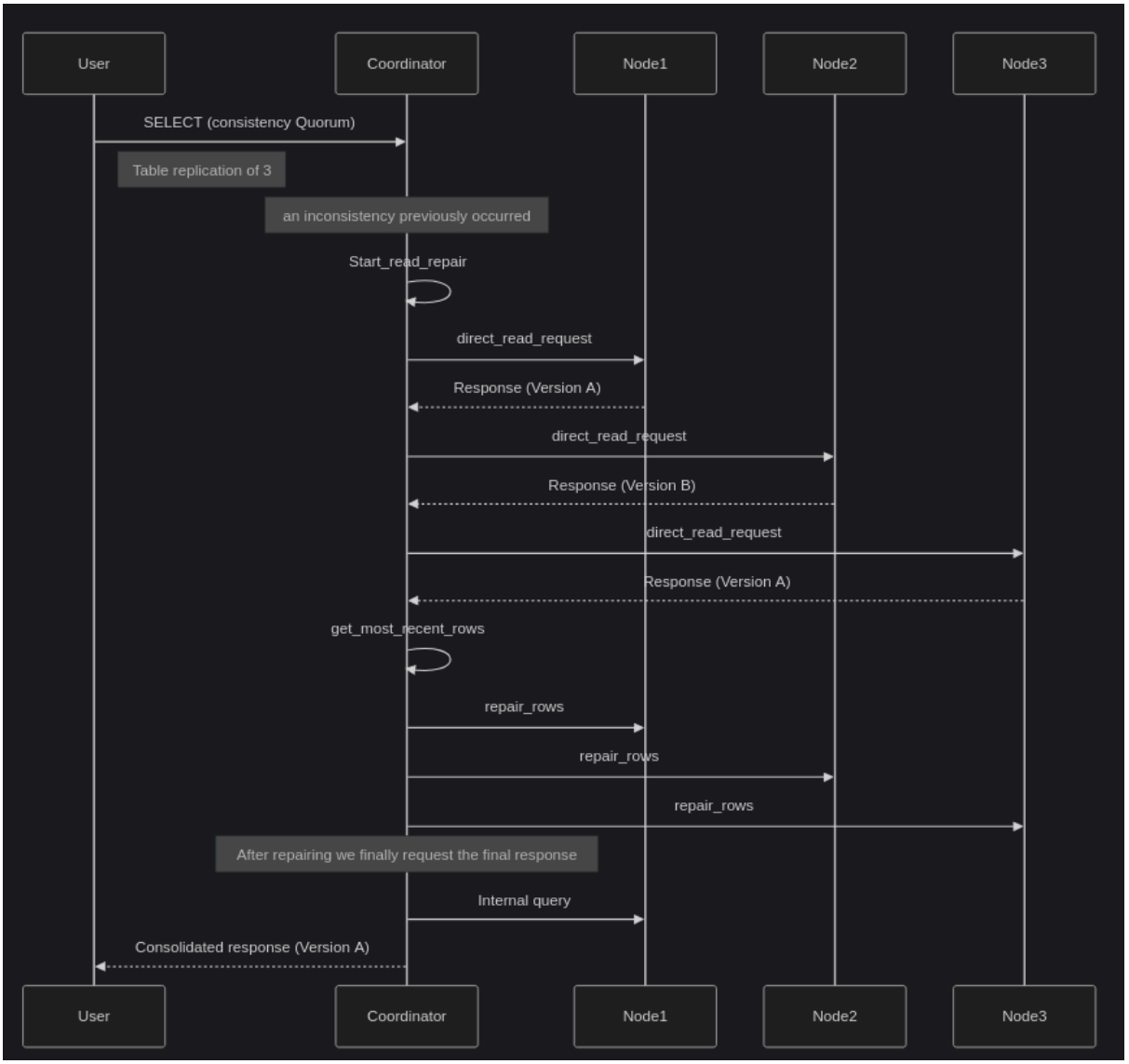
Consistency

Para la consistency se replicó la misma estrategia de Cassandra, primero hacemos la consulta, después obtenemos los digest-read-request necesarios para cumplir la consistency, y si alguno genera alguna inconsistencia llamamos a read-repair. Notar que en este ejemplo hicimos 2 read-request porque el nodo 2 tenía una inconsistencia y nosotros queríamos completar la consistencia, por eso se consulto al nodo 3, en caso de que el nodo 2 hubiera estado actualizado entonces no se le hubiera consultado al nodo 3.



Read-repair

Para el read-repair luego de detectar una inconsistencia, pedimos las réplicas de las tablas a todos los nodos con réplica, ya que no sabemos cual replica es la desactualizada. Luego con todas las réplicas se ve cuales son las filas más actualizadas y estas se devuelven a los nodos de las réplicas. Finalmente volvemos a hacer la consulta inicial para obtener las filas que queríamos ya reparadas para devolverlas al cliente.



Cliente

El cliente tiene pocas funcionalidades, se encarga de conectarse a algún nodo del clúster obteniendo aleatoriamente alguno de los IPs guardados, creando una conexión encriptada mediante una biblioteca de rust llamada `rustls` para conectarse, y esperará la respuesta, luego hará todo el procesamiento necesario para mostrarlo al usuario.
Además tiene la opción de definir el tipo de consistencia que se desea para cada query que se envíe, por defecto es *Quorum*, la consistencia que se elija siempre va a ser la misma entre queries hasta que se especifique lo contrario.

Interfaz de Usuario

La interfaz de usuario se pensó para ser lo más simple posible: un mapa geográfico con aeropuertos resaltados, coloreados según su tipo e importancia. Utiliza para ello la librería `egui` (concretamente el *framework* `eframe`) y tiene las siguientes características:

Al hacer *click primario* en alguno de esos aeropuertos:

- Se pueden ver los detalles del mismo.
- Se pueden mostrar las listas de vuelos entrantes y salientes respecto a ese aeropuerto.
- Se pueden editar o borrar las entradas de esos vuelos.

Al hacer *click secundario* sobre cualquier otro aeropuerto que no sea el primariamente seleccionado:

- Se pueden ver los detalles del mismo.
- Se puede ver un botón que permite insertar un vuelo. A este punto, la interfaz insertará un vuelo con el aeropuerto primario como origen y el secundario como destino, a la hora y fecha actual seleccionadas.

Widgets

Representan una parte integral de la GUI y se pueden ver realizando tareas de conveniencia:

- Un calendario para seleccionar la fecha actual.
- Un menú *dropdown* para seleccionar la hora actual.
- Una ventana de *login* simple.
- Ventanas auxiliares para editar un vuelo en concreto.
- En raras ocasiones, una barra de carga para mostrar el progreso actual de todos los aeropuertos en memoria.

Plugins

La librería `walkers` incluye como funcionalidad lo que éste llama *“Plugins”.* Son similares a los *widgets* de `egui`, excepto que el mapa geográfico es en sí un *widget*, y por lo tanto hacen falta otras estructuras auxiliares para modularizar la lógica de nuestro proyecto.

Los mismos se nos dividen en:

- Un cargador de aeropuertos.
- Otro cargador para todos los vuelos (incluso los vuelos en vivo).
- Un dibujante para los aeropuertos (dibuja los aviones de colores).
    - Los *plugins* vienen acomodados con un `Painter` capaz de dibujar figuras e imágenes sobre el canvas que es el mapa.
- Otro dibujante para los vuelos en vivo (dibuja la trayectoria).
- Uno que vigila los clicks del usuario y realiza una lógica diversa dependiendo de cómo y dónde se hace *click*.
    - Es así que por ejemplo seleccionamos los aeropuertos.

Simulador de Vuelos

El simulador de vuelos posee un cliente para así comunicarse con el servidor. Durante su ejecución pide al usuario detalles del vuelo a generar (ID,origen y destino) para que con ayuda del Threadpool que posee cree y maneje cada vuelo; estos se insertan en una lista de vuelos presente en el hilo principal de la aplicación, y a su vez se ejecuta su simulación en hilos independientes.

Cada vuelo en base a los datos iniciales hace cálculos de desplazamiento, y actualizaciones thread-safe tanto de características preestablecidas como el estado del vuelo, y otras variaciones aleatorias durante un período fijo de tiempo, como velocidad, altura, y combustible restante.

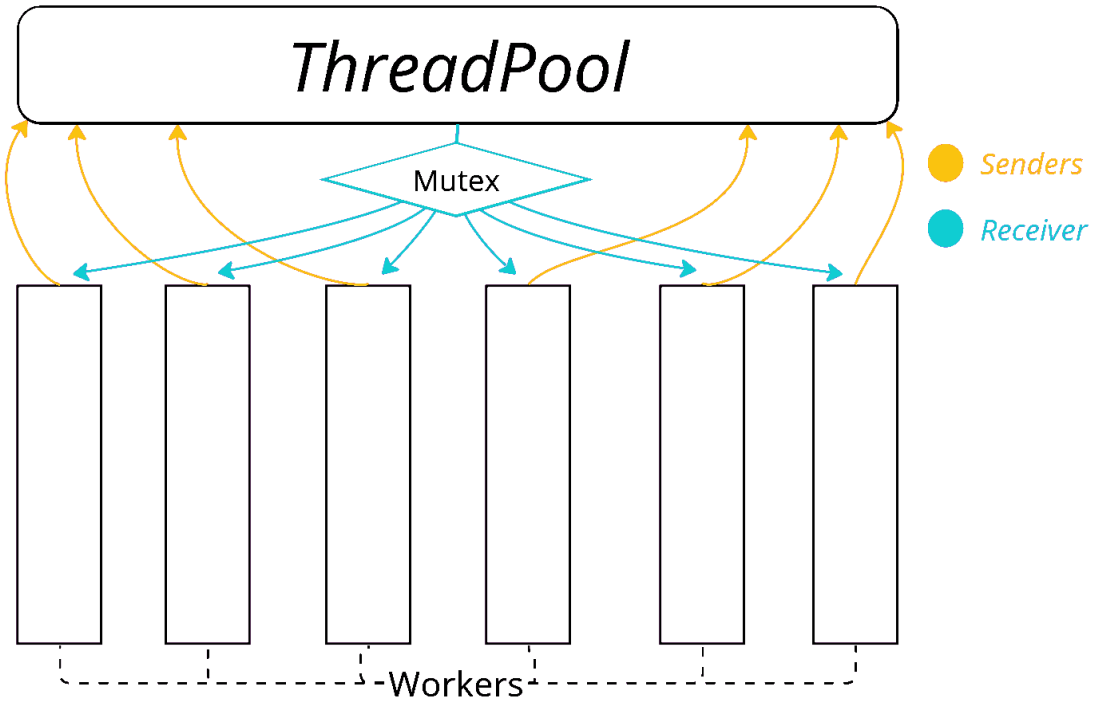
Desde la misma terminal también se puede acceder a la lista de vuelos totales para ver datos básicos de los mismos, buscar en ella según el ID los datos específicos de un vuelo y por último chequear una lista de aeropuertos (cargada en memoria) disponibles para usar de origen y destino en memoria.

ThreadPool

Esta estructura hace uso de sus finitos hilos (o “*threads”*) en vez de potencialmente crear uno por vuelo y lidiar con una cantidad indefinida.
Cada vez que hay una tarea por hacer (un *“Job”*), el *ThreadPool* se lo delega a uno de sus hilos (*“Workers”*) para que alguno lo ejecute. Utiliza los canales de Rust para comunicarse con los mismos, con una particularidad:

El extremo de envío *“Sender”* puede clonarse fácilmente y asignar las copias a cada hilo, pero a nosotros nos interesa que los hilos **reciban**, y el extremo receptor *“Receiver”* no puede clonarse. Más allá de ser un error conceptual, el lenguaje está diseñado para no permitirlo.

Entonces, usamos un `Arc` para compartir la referencia y `Mutex` para asegurarnos de que sólo un hilo acceda al extremo receptor a la vez. Con esto, también solucionamos el potencial problema de que, al mandar una nueva tarea, más de un hilo lo reciba y ejecute doblemente o haya *deadlocks*. También asegura que los hilos que no lo recibieron se queden dormidos, de manera que no hacen *“Busy Waiting”* comprobando por nuevos mensajes a cada iteración.



Características del Motor de Base de Datos

---

1. **Modelado de Datos**
    - Diseño de la creación de keyspaces y tablas, replicando su creación entre los nodos.

- Optimización del almacenamiento particionando los datos de las tablas entre los nodos, definiendo un rango de *hashing* para cada nodo al escribir datos, de esta forma la carga se distribuye de la manera más equitativa posible.

2. **Operaciones CRUD**
   - Implementación de las operaciones CRUD (creación, lectura, actualización y eliminación) utilizando la sintaxis de CQL.
   - Implementación de *Read-Repair* en cada lectura para asegurarnos de que los datos leídos de los nodos son los más actuales, y si uno contiene datos más antiguos o faltantes, se repara antes de responder al cliente.
3. **Replicación y Consistencia**
   - Configuración de replicación para que cada nodo tenga copias de sus datos en otros nodos para poder asegurarse de que los datos sigan siendo accesibles si un nodo falla. La replicación se definió que sea de 3 nodos.
   - Soporte a distintos niveles de consistencia, definido por cada cliente antes de enviar las consultas. La consistencia se asegura de que N nodos respondan que procesaron correctamente la consulta recibida, sino se devuelve un error. La consistencia por defecto es *Quorum*.
4. **Funcionamiento del Clúster**
   - Soporte al protocolo de comunicación nativo de Cassandra, para el envío de mensajes durante las conexiones.
   - Implementación del protocolo *Gossip* para que todos los nodos tengan el estado más actualizado de los demás, para poder enterarse de cualquier cambio.
   - Tolerancia a fallo de uno o más nodos, cuando un nodo no responde, el nodo que le envió un mensaje asume que está apagado, se guarda ese dato y luego lo compartirá en las rondas de *Gossip* para que los demás nodos se enteren.
5. **Seguridad**
   - Autenticación y Autorización, cada cliente debe iniciar sesión con un usuario y una contraseña antes de poder enviar consultas al clúster, sino serán rechazadas, recibiendo un error.

- Encriptación de datos en tránsito, los datos enviados entre los clientes y el clúster son encriptados para mayor seguridad ya que esa conexión es considerada “pública”, la conexión entre nodos se considera “privada”, por lo que no cuenta con dicha encriptación.

Containerization

Cada nodo estará en entornos aislados llamados contenedores, haciéndolos más ligeros y eficientes en términos de recursos. Esto se logrará mediante *Docker*, además se le suman dos mejoras: reconfiguración dinámica del cluster y un logging del intercambio de mensajes entre los nodos. A continuación detallaremos estos tres conceptos.

Docker

Los nodos “iniciales” *(los que están en el archivo de IPs al principio)* fueron configurados para poder correrse con Docker.
Esto se logra creando una imagen ***nodos*** o ***nodos-slim***, y luego levantando contenedores en base a dicha “plantilla”.

Creando una imagen

Cuando sabemos que vamos a usar una plantilla para copiar en cada contenedor, obviamente nos interesa dejar la imagen con el proyecto ya compilado. O al menos, compilar las partes justas tal que el binario **nd** funcione. Así, cada contenedor sólo tendría que referirse a ese binario mediante *cargo run*.
Sin embargo, esto resultaba en un problema: la imagen resultante tenía un tamaño considerable. Esto es porque las herramientas de compilado *(como **cargo** o **rustup**)* todavía quedan en la imagen cuando ya no nos interesa.

Optimización *Slim*

Con sólo compilar y dejar el directorio listo, la imagen pesa **1.19 GB**, debido a herramientas de compilación, paquetes del proyecto no relevantes a la imagen, o simplemente programas del SO que traía de la imagen base.
Si quitamos todo eso y nos quedamos puramente con el binario y los datos estáticos necesarios para el funcionamiento del programa, logramos bajar el peso de la imagen a **16.59 MB**, más de 70 veces más ligera.

Los contenidos todavía pueden ser inspeccionados con ***logs*** o ***inspect***, en caso de ser necesario.

Docker Compose

Toda instancia del clúster, al ser iniciada, cuenta con 5 nodos. Levantar este estado inicial puede ser automatizado con ayuda de la herramienta **docker compose**, y provee una manera útil de levantar, detener o destruir estos nodos iniciales. Cada nodo sale en la consola distinguido por un color distinto. En cualquier caso, los clientes como la interfaz o el simulador, se pueden conectar a este clúster incluso afuera de Docker.

Reconfiguración dinámica del clúster

El clúster puede soportar nodos agregados y borrados dinámicamente, esto es, las réplicas y los rangos de partición, así como todos los datos, se adaptarán según la entrada o la salida de un nodo.

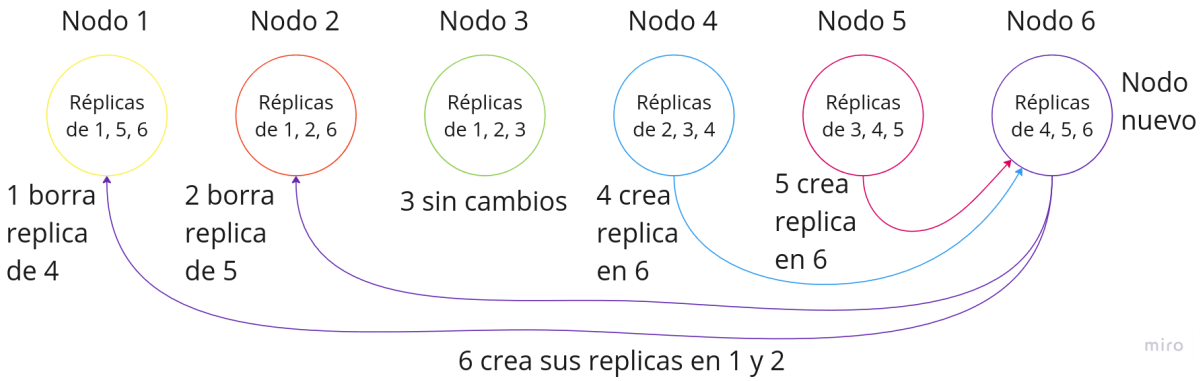
Agregado de un nodo

Para agregar un nodo al clúster, al ejecutar el comando, se notifica a los nodos del clúster, entonces uno de ellos va a enviarle la metadata necesaria al nodo nuevo para que pueda acoplarse al clúster, y mediante *gossip* el nuevo nodo se presentará con ellos. Una vez que el nuevo nodo recibe la metadata, envía un mensaje a los nodos para que comiencen a acomodar sus réplicas, mediante este proceso los nodos que correspondan van a hacer cambios en sus almacenamientos.

Actualización de réplicas para agregado

Específicamente, en base al nuevo nodo, los que tendrán cambios serán los que están a distancia (a ambos lados) igual a la cantidad de réplicas que tiene el keyspace de las tablas a replicar.
Los cambios que ocurren son:

- Los nodos a la izquierda del nuevo nodo (pensando a los nodos como un vector), crearán sus réplicas en el nuevo nodo.
- Los nodos a la derecha, borrarán las réplicas de los nodos a la izquierda mencionados previamente.
- El nuevo nodo creará sus réplicas en los nodos a la derecha.
- Los nodos fuera del alcance de la cantidad de réplicas del keyspace de las tablas, no tendrán cambios.



Cada nodo esperará a saber que todos los nodos del clúster terminaron de actualizar sus réplicas, una vez eso ocurre, comienza un proceso llamado *relocalización*.

Relocalización

Al entrar un nuevo nodo, el rango de los partition key que le pertenece a cada nodo cambia, entonces puede que algunas filas de los nodos ya no le correspondan. Además, previo a entrar en este proceso, se cambia el estado de los nodos para que no se puedan recibir *queries,* ya que el contenido de cada nodo podría no ser correcto.
Entonces, cada nodo va a recorrer cada fila de cada tabla y va a reevaluar su valor de *hashing* y compararla en el nuevo rango de su *partition key*.

- Para las filas que siguen perteneciendo al mismo nodo, va a reinsertarlas truncando la tabla para quedarse solamente con las que son correctas.
- Para las filas que ya no pertenecen, las va a enviar al nodo correcto. Las filas enviadas a otros nodos, se insertan sin truncar, para prevenir perder filas que sí corresponden, porque luego ese nodo va a hacer el filtro de sus filas.

Al finalizar esto, cada nodo va a ordenar sus tablas y eliminar filas duplicadas ya que puede suceder en pocos casos, con esto se asegura que el resultado final de las tablas es correcto.
Una vez que todos los nodos terminaron de relocalizar todas sus filas, van a esperar que todos los nodos del clúster hayan terminado también, y finalmente vuelven a estar en estado normal para recibir *queries*.

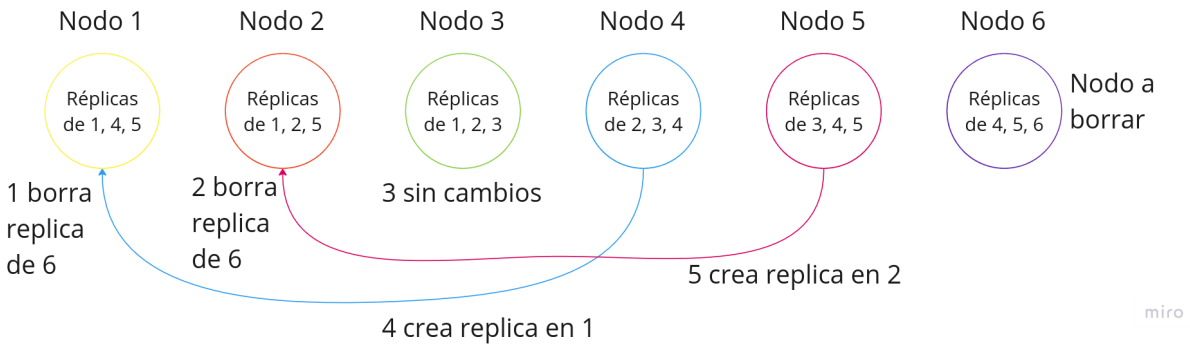
Borrado de un nodo

Para borrar un nodo del clúster, es similar al agregado de un nodo, pero con algunas diferencias. Al ejecutar el comando, se notifica a los nodos del clúster que un nodo debe ser borrado, entonces los nodos deben acomodar sus réplicas.

Actualización de réplicas para borrado

Esto es similar al agregado de un nodo, los nodos que estén a distancia igual a la cantidad de réplicas, en base al nodo a borrar (a ambos lados), del keyspace de las tablas serán afectados.

- Los nodos a la izquierda del nodo borrado (pensando a los nodos como un vector), crearán sus réplicas en la posición más lejana de la cantidad de réplicas en los nodos a la derecha, o sea, igual a la cantidad.
- Los nodos a la derecha, borrarán las réplicas del nodo a borrar.
- El nodo a borrar no tendrá cambios ya que dejará de ser tomado en cuenta en el clúster.
- Los nodos fuera del alcance de la cantidad de réplicas del keyspace de las tablas, no tendrán cambios.



Cada nodo esperará a saber que todos los nodos del clúster terminaron de actualizar sus réplicas, entonces comenzará el proceso de *relocalización* de la misma manera que se hizo en el agregado de un nodo, ya que depende de la cantidad de nodos en el clúster, además se incluye al nodo a borrar ya que debe reasignar sus filas antes de ser borrado, sino dejarán de ser accesibles.
Una vez que el nodo a borrar termina de relocalizar sus filas, frena sus hilos para finalizar el proceso de borrado.
Finalmente, los nodos del clúster volverán a su estado normal una vez terminen el proceso de *relocalización*.

Logger

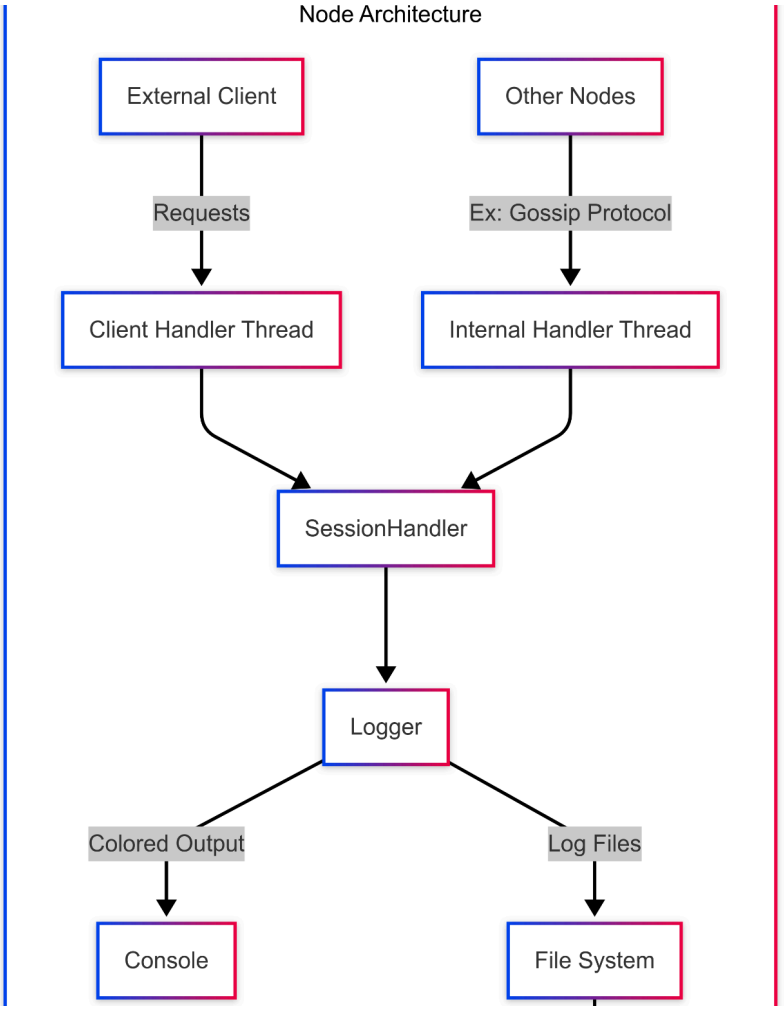
Durante la inicialización de cada nodo creamos dos hilos que están en constante escucha de posibles mensajes; uno para recibir y enviar mensajes a clientes externos (Peticiones) y otro para manejo de comunicación interna (Protocolo Gossip, Read Repair, etc). Para el procesamiento, envío y recepción de dichos mensajes hacemos uso de una estructura SessionHandler única para cada nodo.

Al ser esta una estructura única en cada nodo y encargarse del procesamiento de mensajes, era el foco perfecto para manejar un registro de los mismos. Aquí fue donde se incorporó el uso del Logger, una clase externa con la cual a partir de un tipo de mensaje y el mensaje en sí permite el registro y categorización de los mismos.

Características:

- Formato único: En función de la categoría de cada mensaje se usa un color distinto en la salida en consola.

- Extensible sin modificar código previo: Si se desean crear nuevas categorías basta definir una función con la misma y el color que se desea para la salida estándar.
- Salida por archivos y soporte para Docker: A la par que por consola, en uso local se tiene acceso a un archivo .log para cada nodo con accesibilidad a todos los mensajes procesados por el mismo. Además, se cuenta con soporte levantando los nodos con docker usando docker logs “nodo_id”.
- Garantía de funcionamiento concurrente: De la misma forma que cada nodo escribe sobre sus propios archivos para almacenar metadatos y registros de consultas CQL, cada nodo tiene su propio logger que escribe en archivos distintos. Ahora, como hay 2 hilos que registran “acciones” de nodos (comunicación interna y externa), se decidió hacer un bloqueo del recurso compartido (el mismo Logger) para evitar conflictos de concurrencia a la hora de operar sobre un mismo archivo.



Conclusiones

En este trabajo, aprendimos a realizar un proyecto de mayor escala que en otras materias, primero en un lenguaje de programación que no conocíamos, aprendiendo sus conceptos únicos como ownership y otros, después aplicando temas nuevos como sockets, clientes, servidores y programación asincrónica, luego utilizando elementos que se utilizan en los proyectos reales de hoy en día, como una herramienta de control de versiones, pruebas unitarias y de integración, pair programming, etc.
Por último también fueron necesarios una gran investigación por internet y autoaprendizaje ya que muchos conceptos eran desconocidos por nosotros y además tuvimos muchas trabas en el camino, las cuales pudimos superarlas.
Este trabajo práctico nos pareció semejante a lo que podría ser un proyecto de la vida real pero en menor escala y con menos gente, y del mismo sacamos muchas enseñanzas, mejoras y más.