Taller de Programación I

Cátedra Deymonnaz

Aerolíneas Rústicas - 2C 2024

Ayudantes

Rafael Berenguel
Mauro Di Pietro

Integrantes

Mateo Requejo - 110109
Blas Sebastian Chuc - 110253
Helen Elizabeth Chen - 110195
Franco Alexis Guardia - 109374

Índice

1. Introducción

En este informe se abordará nuestro diseño e implementación de un sistema de control de vuelos basado en una base de datos distribuida, escalable y tolerante a fallos, compatible con el modelo de Cassandra. A través de este proyecto, integramos numerosos conceptos, que detallaremos a continuación, atendiendo a las demandas de Aerolíneas Rústicas y su operación global.

2. Implementación

2.1. Interfaz

Para nuestra implementación decidimos utilizar el crate walkers de Rust ya que nos proveía de muchas funcionalidades que se adaptaban a lo que necesitábamos hacer, por ejemplo el hecho de tener ya implementada la lógica de un mapa interactivo que se muestra y sobre el que es posible seguir desarrollando funcionalidades. Además este se complementa muy bien con otro crate que usamos que es eframe, que usamos para el manejo de nuestra aplicación (la lógica interna y todas las acciones que pueden realizarse por el usuario en la ventana.

Nuestra interfaz se basa principalmente en subventanas por las que el usuario se va moviendo con las distintas acciones que realiza. Esto lo hicimos así para que sea más interactivo y sencillo ya que el pasaje entre estas subventanas es solo con botones y no es necesario que se introduzca ninguna clase de texto en la terminal, dejando esta reservada para otros aspectos del TP en el que tengamos que utilizarla y también lo abstrae al usuario de tener que conocer más de la implementación para poder usar la aplicación.

Entonces en una situación de uso de la interfaz se le despliega el mapa interactivo al usuario mostrándole los aeropuertos que tenemos guardados en la base de datos y que dibujamos en las posiciones que corresponden, y se identifican estos por un símbolo. Puede o no también haber vuelos pero en caso de que sea la primera vez que se usa la interfaz no aparecen. El usuario además de poder desplazarse por el mapa puede seleccionar un aeropuerto y luego de eso una fecha cualquiera, y se le mostrarán los vuelos que tengan como origen y como destino a ese aeropuerto seleccionado y que tengan que volar en esa fecha. En caso de que el usuario quiera ingresar un vuelo tomando como origen a ese aeropuerto seleccionado y en esa fecha en otra subventana tiene que elegir el destino y ya está creado.

Pero en caso de que se quiera verlo es necesario simularlo por consola, cuando esto se hace se lo puede observar en el mapa desplazándose y cuando llega al destino deja de mostrarse. Incluso en el trayecto de este se lo puede seleccionar para modificar su estado y dependiendo de cual sea este el cambio se ve reflejado también en ese vuelo

La operación mencionada se encuentra representada en el siguiente diagrama:

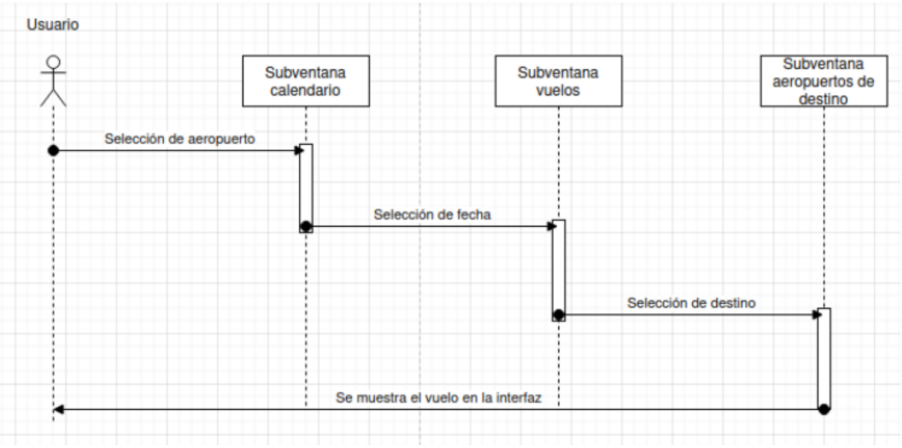

Figura 1: Diagrama de secuencia de la interfaz

2.2. Simulador de vuelos

Nuestro simulador de vuelo consta de un workspace propio en el que definimos el procesamiento de los datos ingresados por el usuario, la generación de un vuelo simulado con dicha información y la actualización de los valores con el fin de simular el movimiento de un vuelo de un aeropuerto a otro en línea recta, todo esto siendo posibles gracias a qué somos un cliente para la base de datos y tenemos la posibilidad de generar consultas CQL que impacten en la base de datos, mientras nos abstraemos de cómo se ejecuta por detrás dicha conexión y comunicación.

Una vez creado nuestro primer vuelo, tenemos la posibilidad de seguir creando vuelos que se ejecutarán en paralelo entre sí, gracias a qué el simulador incorpora el uso de un Threadpool, que consiste en un conjunto de "trabajadores"que se encargarán de realizar las funciones que le enviemos (siempre y cuando estén disponibles) en un thread distinto al thread que está corriendo la consola (el principal); y cuando un vuelo simulado llegué a su destino, entonces en ese momento se liberará el "trabajador"para recibir y procesar nuevas simulaciones de vuelos.

2.3. Protocolos de comunicación

- **Cliente Servidor**

  Para nuestro trabajo decidimos implementar la versión 4 de las 3 versiones disponibles de protocolos de Cassandra (Versión 3, Versión 4 y Versión 5), ya que en una primera instancia consideramos que las actualizaciones con respecto a la versión anterior serían necesarias y que la mejoras de la versión 5 eran demasiadas. Acotamos la cantidad de casos a utilizar del protocolo, debido a que en nuestro trabajo práctico no eran necesarios muchos de los que si contempla el protocolo, por lo tanto este fué ajustado y simplificado para atender únicamente las necesidades específicas del sistema que desarrollamos. Esta decisión también facilitó la comprensión y la depuración de los casos posibles de comunicación dentro del sistema, dado que se eliminó complejidad innecesaria de los casos no utilizados.

- **Internodos**

  El protocolo internodos fue creado enteramente por y para las necesidades que iban surgiendo durante el progreso del trabajo práctico, dónde cada dato enviado mediante este protocolo tiene una utilidad específica dentro del sistema distribuido. Además se priorizó la simplicidad y la escalabilidad, con el fin de que el protocolo pueda adaptarse a diferentes tamaños de clúster y manejar incrementos en la carga de trabajo sin comprometer la totalidad de los datos.

- **Gossip**

  En nuestro cluster de Cassandra, los nodos pueden comunicarle a otros la existencia de un nodo caído con el que no se puede establecer una comunicación. Esto lo logramos mediante nuestra implementación del protocolo Gossip.

  Para esto, decidimos mantener esta comunicación en un nuevo puerto para cada nodo: 9044. Desde el momento de nacimiento de un nodo, este comienza a realizar rondas de gossip cada 5 segundos (este tiempo es configurable). Si por alguna razón el proceso se detuviera, establecimos un timeout de 10 segundos para reiniciar las rondas.

  Para que el protocolo pudiera cumplirse, a cada nodo se le creó un campo llamado metadata_nodos. Este HashMap guarda como clave la IP de un nodo, y su EndpointData como valor. El EndpointData es una estructura compuesta por el heartbeat_state (generation, version) y application_state (status) de cada nodo. La generación es el timestamp en milisegundos de cuándo nació el mismo. La versión, un valor numérico que se aumenta con cada ronda de gossip. Y el estatus, un Enum que puede tomar los valores: Normal, Bootstrap y Down.

  En cada ronda de gossip se mantiene el siguiente proceso:

  1. El nodo iniciador toma una IP aleatoria del clúster para iniciar gossip. Comprueba que esa IP no esté en su campo "gossip_recientes" ni en el vector "conectados". El primero es un vector propio del nodo en el que se almacenan los nodos que iniciaron gossip con

él en esta ronda. El segundo, un vector en el que se almacena las IPs con las que este nodo inició gossip.

2. Si la IP no está en ninguno de los vectores, se realiza gossip con ese nodo.
3. Se vuelve a elegir una IP aleatoriamente. Se valida si la IP del punto 2 era correspondiente a un nodo semilla. Si lo fue, se ejecuta gossip con esta nueva IP. Si no lo fue, se descarta esta IP y se toma aleatoriamente una IP semilla.

En pocas palabras, en una ronda de gossip un nodo se conecta con otros dos, garantizando que al menos uno sea un nodo semilla.

El proceso que sigue un nodo al conectarse con otro es el siguiente:

1. El nodo iniciador arma un mensaje SYN donde pone la metadata que tiene almacenada para cada nodo, con el formato: "ip:generacion:versión".
2. El nodo que recibe, compara la versión y generación de la metadata que recibe con la que tiene almacenada, y arma un mensaje ACK donde pondrá la siguiente información: Por un lado, la "ip:generación:versión" de los nodos que tiene desactualizados (es decir, con una versión o generación más vieja). Por otro lado, agrega la "ip:generación:versión:status" de los nodos que el nodo iniciador tiene desactualizados.
3. El nodo iniciador lee el ACK y actualiza sus nodos desactualizados guardando la generación, versión y estatus que el otro nodo le envió. Finalmente, arma un ACK2 donde pondrá la "ip:generación:versión:status" de los nodos que el otro nodo tiene desactualizados.

   Así, se completa una ronda de gossip. El nodo iniciador guardará en el vector "conectados" la IP del nodo receptor, y esté último guardara la IP del nodo iniciador en su campo gossip_recientes. Así, en una misma ronda de gossip, dos nodos no inician la comunicación mutuamente. Estos vectores se limpian en cada ronda.

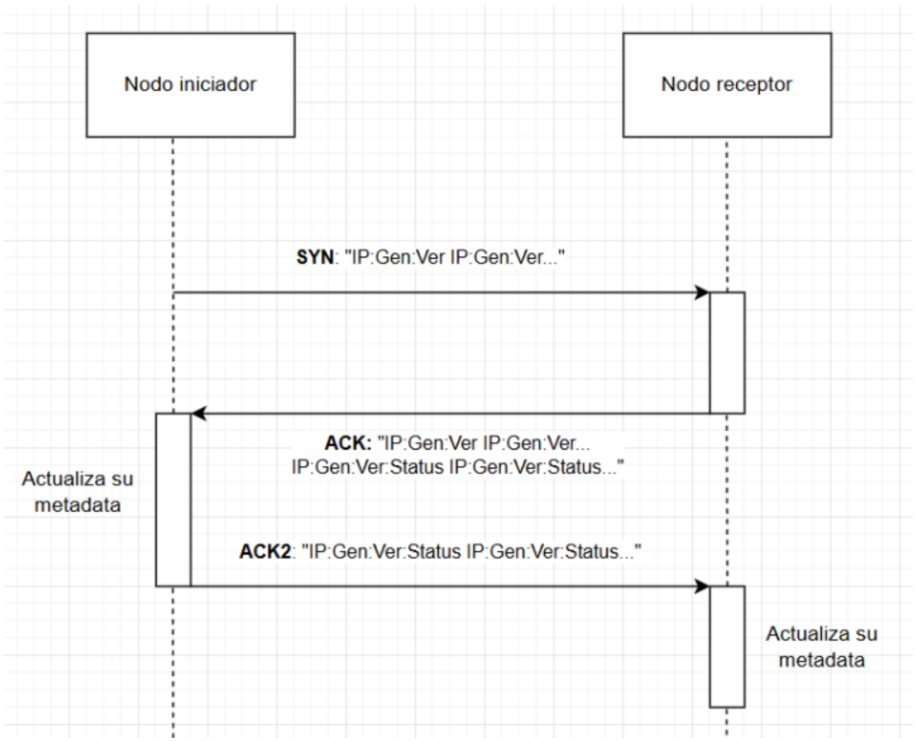

Figura 2: Gossip

2.4. Base de datos

2.4.1. Operaciones CRUD con CQL

De forma de poder ejecutar las operaciones que necesitamos hacer en nuestra aplicación implementamos las operaciones de creación, lectura, actualización y eliminación de datos y como la base de datos que implementamos tiene que ser compatible con Cassandra entonces estas consultas tienen que estar escritas respetando la sintaxis y estructura de CQL.

Decidimos implementar estas operaciones ya que por ejemplo tenemos que poder crear vuelos ya sea desde la consola como desde la interfaz y esta es la forma que tenemos de hacerlo. Luego también importante para el seguimiento de vuelos en curso poder actualizar la posición de estos (desde el lado de la consola) y poder hacer una lectura de esos vuelos en esas nuevas posiciones y así poder actualizarlo en la interfaz y que sea apreciable por el cliente. Por último cuando estos vuelos llegan ya a destino no hay razón de que sigan guardados en la base de datos ya que no volverán a mostrarse ni se volverá a hacer ningún tipo de operación con ellos, entonces los eliminamos.

Para que estas operaciones funcionen correctamente es necesario que se ejecuten con una consistencia que garantice tanto que se hagan correctamente como que el impacto que tengan sea el adecuado. Y también para evitar inconsistencias en los datos implementamos una lógica interna de la base de datos que permite que se actualicen los nodos que quedaron con información desactualizada.

2.4.2. Consistent Hashing y Replication Factor

En nuestra implementación de Consistent Hashing, utilizamos el crate MurmurHash para asignar dentro de un anillo lógico la ubicación de los nodos según los tokens generados a partir de sus ips. Al crear un nodo, primero se hashean todas las direcciones de los nodos para obtener el token que le corresponde a cada uno. Ese token simboliza el rango de responsabilidad que el nodo tiene, es decir, si a un nodo le toca un token del valor de 20, significa que va a almacenar todos los datos que se hasheen y caigan entre el token del nodo anterior y el 20, como se muestra en las siguientes imágenes:

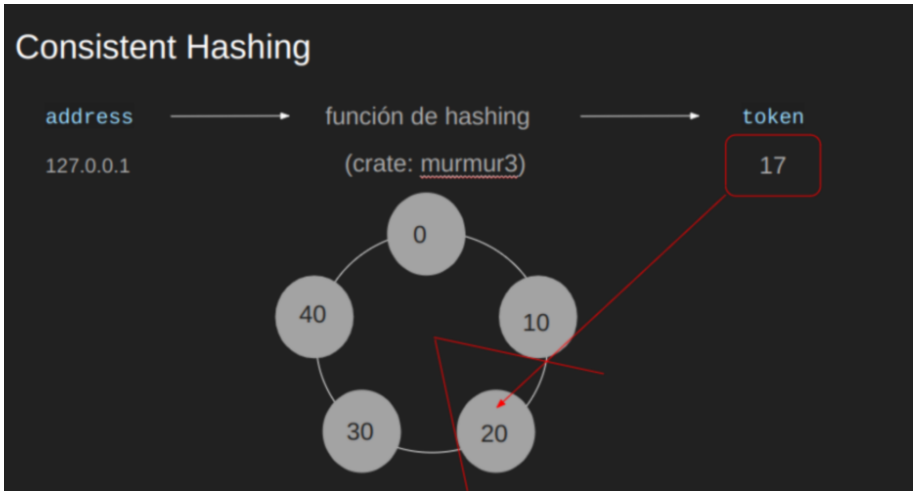

Figura 3: Consistent hashing con IPs

Esta información se guarda dentro su estructura que contiene un campo “nodos” del tipo BTreeMap donde sus claves son los tokens y sus valores la dirección ip. La elección del BTreeMap se debió en gran parte a que al iterar sobre este, los tokens se encuentran ordenados de menor al mayor, aportando no solo a la lógica del cluster, si no que también a la implementación de replicaciones según el valor que tenga el Replication Factor. El valor del Replication Factor se puede modificar según la cantidad de nodos en los que se quiera almacenar la misma información. En nuestro caso, optamos por un Replication Factor de 3 porque nos pareció lo más acorde al tener únicamente 5 nodos en el cluster. Para esto, tenemos una función get_replicas() que devuelve los

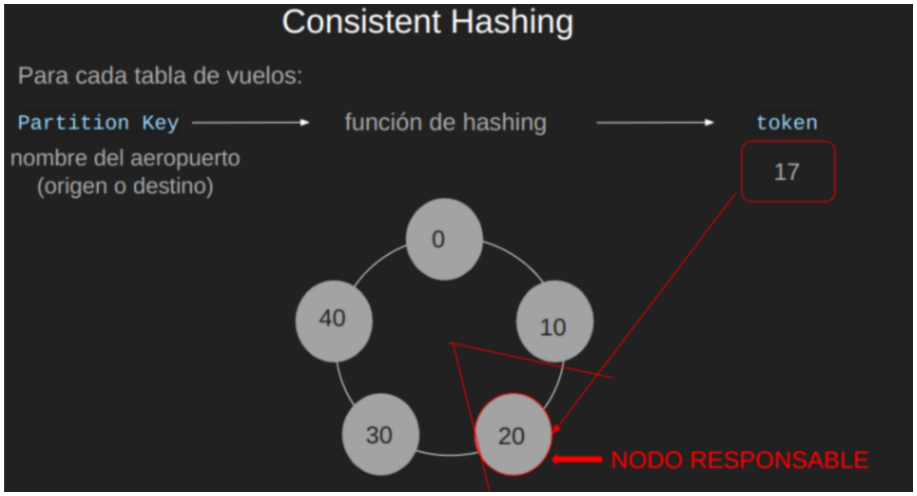

Figura 4: Consistent Hashing con datos

nodos siguientes del cluster (dependiendo del valor del replication factor). Por lo tanto, si se quiere realizar un cambio sobre un dato (por ejemplo, una inserción de un dato), este se verá reflejado también en los siguientes.

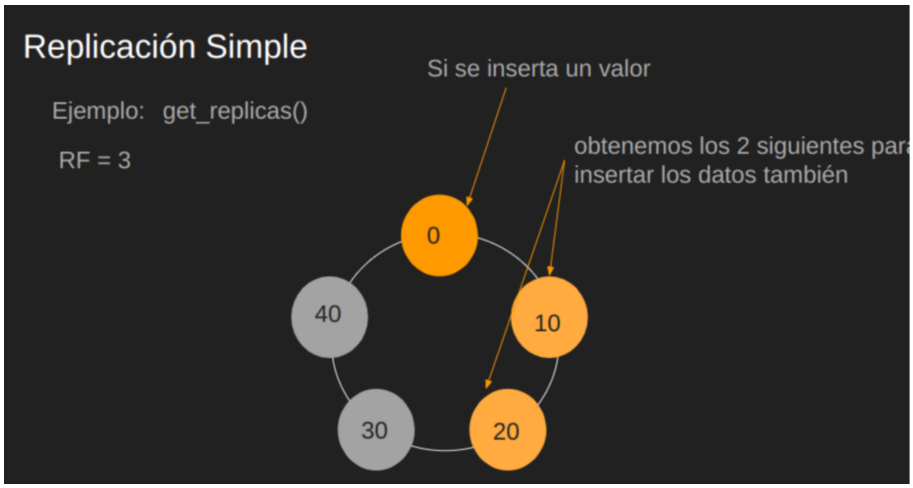

Figura 5: Replication Factor

2.4.3. Consistency Level

Para nuestra implementación decidimos utilizar solo dos formas de consistencia de las provistas en el protocolo nativo de Cassandra, WEAK (para consistencia ONE) y STRONG (para consistencia QUORUM). De esta forma podemos cumplir con el requerimiento del trabajo de poder tener datos que se guarden con consistencia débil y otros con consistencia fuerte. La consistencia WEAK al ser nuestra forma de implementar la consistencia ONE eso significa que el funcionamiento es el mismo, en el sentido de que para las lecturas solo es necesario que una de las réplicas del cluster envíe los datos para responder la consulta recibida y en las escrituras solo una de las réplicas tiene que ejecutar la operación. La misma similitud se da también para STRONG y QUORUM.

Consideramos también en esto la tolerancia a fallos de nodos, conectándonos con las réplicas en caso de ser necesario pero siempre verificando que se cumpla la consistencia en cada consulta y ejecutando por ejemplo la operación de read repair en caso de ser necesario para así todas las réplicas de un dato tienen la información más actualizada.

2.4.4. Read Repair

Con el objetivo de mantener la información actualizada en el cluster implementamos la operación de read repair en donde frente a una consulta de lectura que reciba la base de datos, si hay réplicas que tienen información desactualizada, entonces se las actualiza. Esto por supuesto está fuertemente ligado a la consistencia que tenga esa consulta ya que por ejemplo la consulta recibida tiene consistencia WEAK entonces solo se hace la lectura en una de las réplicas pero es posible que no se actualicen todas ya que tampoco se ve a todas. Sin embargo consideramos que la combinación de consultas de tipo WEAK y STRONG son un balance que permite cubrir casos como estos y garantizar siempre que la información que se envía como respuesta sea la más actualizada.

Para plasmar esta lógica al código, dentro del nodo creamos un campo TIMESTAMP que se va actualizando cada vez que se realiza alguna modificación dentro de los datos del nodo (es decir, cuando hay una inserción, actualización o eliminación). A su vez, para realizar el Read Repair, creamos un HashMap cuyas claves son los TIMESTAMP de los nodos, y sus valores son una tupla con el vector de los datos y la ip del responsable de los datos. Por lo tanto, primero añadimos la información del responsable dentro del HashMap, y luego este se contacta con sus réplicas para que le devuelvan su información al responsable para añadirla al diccionario. Tener los TIMESTAMP como claves, nos permite obtener el nodo con el dato más actualizado (el que tiene el valor más alto en su timestamp) de manera eficiente.

Dado que la operación de read repair implica actualizar datos, se les envía a los nodos desactualizados, sean el responsable o las réplicas, un UPDATE con la información más reciente.

2.4.5. Tolerancia a fallos

Dado que tenemos que cubrir el caso de fallos de nodos y que el cluster siga funcionando ya que esa es una de las cosas que caracterizan a Cassandra y que la hacen tan útil como base de datos distribuida, entonces en nuestra implementación decidimos que si la conexión a un nodo falla cuando se le quiere enviar una consulta entonces es porque ese nodo está caído y en lugar de devolverle un error al usuario mostrando el mensaje por terminal intentamos conectarnos a sus réplicas ya que estas también tienen el mismo dato que se necesita responder (si la consulta es de lectura) o también deben reflejar la operación que se realizan sobre datos que tengan (si la consulta es de escritura), esto por el replication factor y para evitar inconsistencia de datos

Esto por supuesto también está atado a la consistencia que tenga la consulta así como pasaba en read repair y quizás sea necesario conectarnos a tantas réplicas como la consistencia de la consulta necesite o incluso no conectarnos a ninguna ya que el mismo nodo coordinador puede ser una réplica y al ya recibir la consulta puede directamente ejecutarla

De esta forma en momentos críticos como al estar haciendo el seguimiento de un vuelo que se necesitan enviar y recibir información de la base de datos y que todo siga funcionando sin ninguna interrupción en el medio, si en algún momento una conexión con un nodo que venía funcionando falla directamente pasa la conexión a una de sus réplicas y todo continúa como estaba antes, algo sumamente importante de cara a la experiencia de usuario y también evitar posibles errores que puedan hacer que la ejecución se pare por completo.

2.5. Seguridad

2.5.1. Autenticación

Antes de que el usuario pueda hacer consultas con la base de datos es necesario que realice una autenticación de forma de hacerle saber a esta la versión de CQL que va a usar en las consultas y de esa forma poder luego enviar las querys para hacer las operaciones. Si bien no es necesario que haya una autorización por parte de la base de datos en el que solo permite a un cliente que haga consultas si este le envía su usuario y contraseña, de todas formas ambas partes tienen que estar conscientes de la versión que se va a usar para que luego la comunicación sea más fluida.

Entonces el modo de autenticación que decidimos implementar es AllowAllAuthenticator en el cual solo hay autenticación pero no autorización, ya que además es el que se usa por defecto en Cassandra.

2.5.2. Encriptación

Para garantizar la confidencialidad e integridad de los datos en tránsito en nuestra implementación de Cassandra, se decidió integrar una encriptación en todas las comunicaciones entre los componentes del sistema mediante TLS, incluyendo cliente-servidor, protocolo de gossip y comunicaciones intra-nodo. Esto se logró utilizando la biblioteca Rustls.

Para establecer conexiones TLS, es esencial contar con certificados y claves privadas que autentiquen a los nodos y permitan el cifrado de las comunicaciones. Se utilizaron certificados autofirmados generados con OpenSSL, al igual que para las claves privadas:

```
1         openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout key.pem -out
2         cert.pem -config openssl.cnf
3
```

Decidimos crear el módulo "seguridad" donde se alojan las funciones relacionadas a la configuración de cliente y servidor TLS, y las funciones encargadas de obtener y leer los certificados y claves privadas.

La integración de TLS requirió ajustar las secciones del código donde se establecían conexiones TCP, envolviendo los TcpStream existentes con las estructuras proporcionadas por Rustls.

Para el servidor, se creó una función create_server_config que carga los certificados y claves privadas, configurando el ServerConfig de Rustls. De manera similar para el cliente, create_server_config prepara el ClientConfig, cargando los certificados de autoridad y estableciendo parámetros de seguridad. En las funciones donde se establecían comunicaciones mediante TCP, se modificó la aceptación de conexiones para envolver los TcpStream entrantes en ServerConnection y posteriormente en StreamOwned, permitiendo manejar conexiones TLS.

De esta manera, logramos cifrar los datos en tránsito, así se evita que información sensible pueda ser interceptada y leída por actores no autorizados, garantizando la protección de los datos.

2.6. Agregado final

2.6.1. Containerization

Para poder incorporar la containerización de Docker al proyecto, fue necesario definir a cada nodo posible que se pueda incorporar al cluster como servicio en el archivo docker-compose. De esta forma, en cada uno se pudo especificar los elementos imprescindibles para el funcionamiento de la base de datos (como por ejemplo el mapeo de puertos que se usarán ahora que los nodos funcionan como contenedores; o los volúmenes compartidos desde donde se modificarán los archivos para seguir persistiendo todos los datos que se hayan agregado en una ejecución).

Por otro lado, un aspecto fundamental es la comunicación de los nodos, que ahora debe adaptarse por el agregado de Docker. Para eso, hicimos uso de la red bridge predeterminada de Docker a la que todos los contenedores se conectan cuando se inician. Desarrollando lo dicho, los nodos se conocían entre ellos y podían comunicarse utilizando el protocolo Gossip de Cassandra implementado, lo que hizo posible el funcionamiento correcto de la base de datos (dado que ahora al estar todos los nodos activos comunicándose, es entonces cuando se crea el "cluster" donde se almacenarán los datos de la aerolínea).

A su vez, uno de los elementos importantes relacionado a la comunicación son los puertos que los nodos utilizaban para tal fin (tanto entre nodos como entre cliente y servidor para realizar peticiones). Al pasar los nodos a ser contenedores, un conflicto que surgió es la necesidad de modificar las direcciones que se usaban para las conexiones, ya que las IP de localhost que formaban parte de ellas no nos servían para conectarnos. Por esa razón, también es que cada nodo se definió como un servicio, para que de esa forma el nombre que se le asignó sirva como reemplazo de la IP en la dirección. Luego de esa modificación, las conexiones se hicieron exitosamente permitiendo también que la base de datos funcione.

Finalmente, en el caso de la interfaz y el simulador de vuelos, dos pilares fundamentales del programa, decidimos no incorporarlos a Docker. Si bien esto es realizable y puede parecer contradictorio con el objetivo de la containerización, que es que todo el proyecto esté cubierto para que de esa forma poder ejecutarse en otras nuevas máquinas sin tener que instalar todas las dependencias necesarias para que pueda funcionar, notamos que el tiempo que demoraba la construcción de la

imagen base (con todo lo necesario definido en el archivo dockerfile) se incrementaba considerablemente al tener todo el proyecto en Docker llegando a los 10 minutos. Esto evidentemente volvía inviable para todos los cambios y pruebas que hacíamos en el desarrollo, por lo que optamos por combinar nodos que funcionen como containers, y una interfaz y consola que estén por fuera de eso pero que igualmente puedan seguir comunicándose para que la base de datos funcione y puedan simularse vuelos. Con los cambios mencionados anteriormente, notamos que algo como esto es posible ya que las conexiones a los nodos se realizaban utilizando sus nombres de servicio definidos y luego por esos mismos sockets los nodos devolvían la respuesta (si es que había alguna), sin obligar a que todo el conjunto esté en el mismo ambiente para que funcione.

2.6.2. Reconfiguración dinámica del cluster

Para poder implementar la reconfiguración del cluster dinámicamente, tuvimos que considerar el agregado de nuevos nodos además de los 5 iniciales que usamos hasta ahora, esto implicó también considerar las IPs que estos nodos tengan. Para poder implementarlo nos valimos principalmente del protocolo Gossip, ya que el nuevo nodo que se agrega recibirá en algún momento una conexión para iniciar una comunicación Gossip y es ahí cuando se comienza a saber entre todos los nodos que uno nuevo se agregó. Esto también es efectivo por el uso de la creación de tablas y keyspaces por comando de terminal, el cual siempre se ejecuta luego de levantar los 5 nodos iniciales. Lógicamente el nuevo nodo no existía para cuando ese comando se ejecutó por lo que no tiene tablas y no archivos creados para guardar los datos que le correspondan, entonces esa es una forma de diferenciar a un nuevo nodo que se agrega.

Por otro lado, el nuevo nodo ya se conecta a la red a la que están conectados los demás al iniciarse su contenedor, y el "lugar" en el que pueda caer en el cluster no es algo que se sabe con anterioridad. Lo que sí se sabe es que cuando se cree le corresponderá un cierto rango de datos de acuerdo con la partition key y por la forma en que nuestro replication factor funciona eso significará que el nodo inmediatamente siguiente a él será una réplica suya, por lo que entonces este nodo se convierte en un nodo redistribuidor que le tiene que enviar al nuevo nodo los datos que tiene. Pero de forma de seguir cumpliendo el replication factor eso significará que la réplica más alejada del nodo redistribuidor debe dejar de tener los datos (por ejemplo, si el RF es de 3 habrían 4 nodos con el mismo dato si esto no se hace), entonces ese nodo redistribuidor también tiene que encargarse de eliminar los datos de esa réplica

Luego de esto, el nuevo nodo funciona normalmente como los demás, calculó sus réplicas (y es réplica de otros) al unirse y le corresponde un rango de datos del cual es responsable, y realiza comunicaciones Gossip con los otros periódicamente

2.6.3. Logger

Para la implementación del logger, el cual se hace necesario dada la gran cantidad de prints que tenemos en el programa (siendo todos importantes por la información que pueden mostrar respecto a una funcionalidad), lo que hicimos fue una redirección del stdout a un archivo especifico cuyo nombre definimos y que expresa los logs de un nodo en particular. Esta redirección la hicimos con el operador ">" pero escrita como "2>&1", ya que de esa forma los prints se siguen mostrando en la terminal y luego pueden verse utilizando el comando "docker logs", además de durante la ejecución ya que se siguen mostrando cuando se hacen. La incorporación de los archivos donde quedan registrados los logs nos permitió poder hacer un mejor seguimiento del funcionamiento cuando agregamos nuevas cosas o cambiamos las que ya están, ya que por la cantidad de prints que hay se nos dificultaba el seguimiento en la propia terminal y nos demoraba mucho tiempo.

3. Dificultades encontradas

- Error conceptual respecto al cluster

  Uno de los principales desafíos iniciales fue comprender correctamente la arquitectura del clúster. Partimos de la idea errónea de que el clúster era una estructura general monolítica que contenía nodos interconectados internamente. Sin embargo, aprendimos que, en realidad,

el clúster está compuesto por múltiples nodos independientes que se conectan entre sí directamente, intercambiando información a través de protocolos específicos. Este error conceptual nos llevó a replantear varias partes de nuestra implementación y ajustar las interacciones entre nodos, lo que supuso un retraso en las primeras etapas del desarrollo.

- Todo el trabaja en una "super rama"

  Al inicio del proyecto, adoptamos un enfoque centralizado en el que todo el código se integraba en una "super rama". Esto nos obligaba a realizar sesiones de LiveShare constantemente, lo que dificultaba la productividad y la distribución eficiente del trabajo. Con el tiempo, logramos establecer un flujo de trabajo paralelo utilizando ramas individuales para cada tarea y unificando los cambios mediante revisiones colaborativas en el control de versiones. Este cambio no solo mejoró nuestra productividad, sino que también permitió una integración más ordenada y estructurada del proyecto.

- Necesidad de un tercer puerto para el gossip

  Originalmente, habíamos planeado utilizar los mismos puertos que empleábamos para las operaciones de datos y replicación. Sin embargo, esto generó conflictos y limitaciones en la comunicación, ya que las operaciones regulares interferían con los mensajes del protocolo Gossip. Esto nos llevó a incorporar un tercer puerto dedicado exclusivamente a las actividades de Gossip, permitiendo que los nodos intercambiaran información de estado y metadatos sin interrupciones.

- Cambios en las IPs debido a la containerización

  Con la implementación de Docker, los nodos no se podían comunicar entre sí usando las IPs que usaban anteriormente: 127.0.0.1, 127.0.0.2, ... Con Docker, debían empezar a conectarse a los nombres del servicio de los otros nodos. Por ejemplo: node1:puerto, node2:puerto... Esto nos supuso una dificultad debido a que se tuvo que mapear en todo el proyecto las conexiones de este tipo para realizar los cambios correspondientes. Además, dejaba de funcionar nuestro archivo de seeds que teníamos, donde estaban alojadas las IPs de los nodos semilla del clúster. Estas cuestiones hicieron que en algunos lugares del proyecto sea difícil reconocer dónde se debía usar el nombre del servicio y dónde usar la IP original que habíamos designado.

4. Conclusión

Este proyecto no solo permitió profundizar en conceptos claves (como la comunicación entre procesos mediante TCP, la creación de una base de datos distribuida y la generación de un entorno seguro), sino que también ofreció una experiencia práctica en el desarrollo de soluciones distribuidas adaptadas a escenarios reales. En un entorno donde la disponibilidad a datos debe mantenerse constante y las fluctuaciones entre los distintos estados de los nodos debe mantenerse monitoreada, este sistema presentó un desafío para realizar ejecuciones sin interferencias ni pérdidas de información.

El desarrollo de este proyecto ha sido una experiencia enriquecedora que nos ha permitido ampliar nuestros conocimientos. Queremos expresar nuestro agradecimiento a nuestros docentes, quienes nos han acompañado en cada etapa del proceso.