Programación de Objetos Distribuidos - 72.42

Airport Service

Trabajo práctico 1

Profesores:

Marcelo Emiliano, Turrin.
Franco Román, Meola.

Autores:

Francois, Gaston Ariel (62500)
Mentasti, José Rodolfo (62248)
Preiti Tasat, Axel Facundo (62618)
Hernando, Lautaro (62329)

28 de Abril de 2024

Decisiones de diseño

Se decidió crear diversos repositorios, en la que cada uno de ellos contiene un tabla de hash para lograr accesos en órdenes de tiempo O(1) a los distintos objetos por su identificador, y así agilizar la ejecución de las solicitudes de los clientes en el servidor. En ellas, se comprenden:

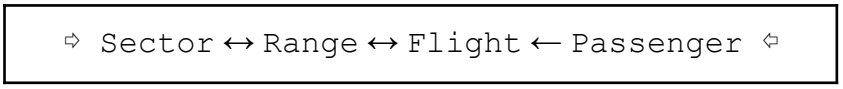
- `SectorRepository`: Mapea los sectores (`Sector`) por su nombre
- `PassengerRepository`: Mapea los pasajeros (`Passenger`) por su código de reserva
- `FlightRepository`: Mapea los vuelos (`Flight`) por su código
- `AirlineRepository`: Mapea las aerolíneas (`Airline`) por su nombre

También, se podrá notar la presencia del repositorio `RangeRepository`, pero esto es simplemente para generar los rangos de mostradores que irán solicitando los clientes mediante la acción `addCounters` y mantener el conteo incremental para todos los sectores.

Otra estrategia de optimización fue la técnica de referencias cruzadas, de manera que las relaciones entre instancias se pudieran navegar en ambos sentidos, considerando evitar en todo momento una potencial ocurrencia de deadlocks. Es así como:

- Un `Sector` almacena una lista de sus `Range` de mostradores, y cada `Range` sabe su `Sector` (`Sector` ↔ `Range`)
- Un `Flight` conoce su `Range` asignado y un `Range` sabe que lista de `Flight` están realizando check-in en él en ese instante (`Flight` ↔ `Range`)

Esto se debió principalmente a que los puntos de acceso son siempre desde el nombre del `Sector` o bien desde el código de reserva de un `Passenger`, quien conoce su `Flight`. Como resultado, se obtiene el siguiente esquema de navegación entre clases:



Evolución del diseño

En una primera instancia, se planteó hacer un análisis profundo del sistema y una revisión total a todas las funcionalidades. En este sentido, fueron muy utilizadas las visualizaciones a continuación mediante las cuales se tomaron decisiones de diseño, de concurrencia y de performance.

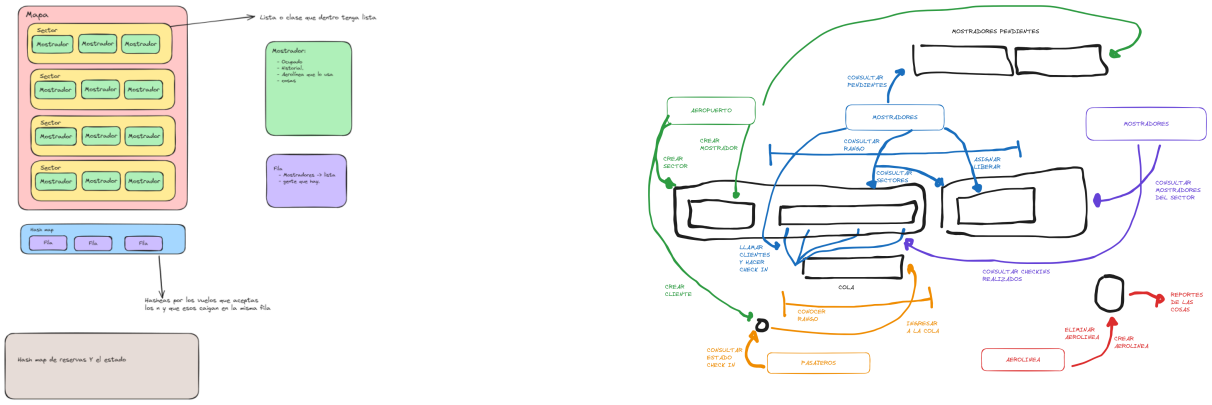


Fig 1 y 2: Esquemas e ideas iniciales de la implementación

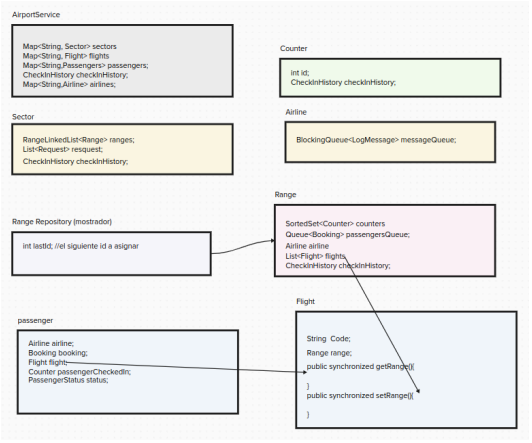


Fig 3: Esquema simplificado de diseño final

Implementación de los servicios

Para la separación de responsabilidades, se decidió desarrollar un *servant* para cada uno de los servicios a ofrecer, los cuales simplemente se encargaban de mapear los objetos autogenerados por gRPC a instancias de nuestra implementación, y viceversa en la respuesta.

Cada uno de ellos contaba con una referencia al servicio `AirportService` que aunaba las diferentes acciones que el sistema debía resolver. Luego, el servicio solicitaba la información requerida a los repositorios mencionados anteriormente para poder llevar a cabo cada una de las acciones y ordenaba la ejecución de los métodos a las instancias.

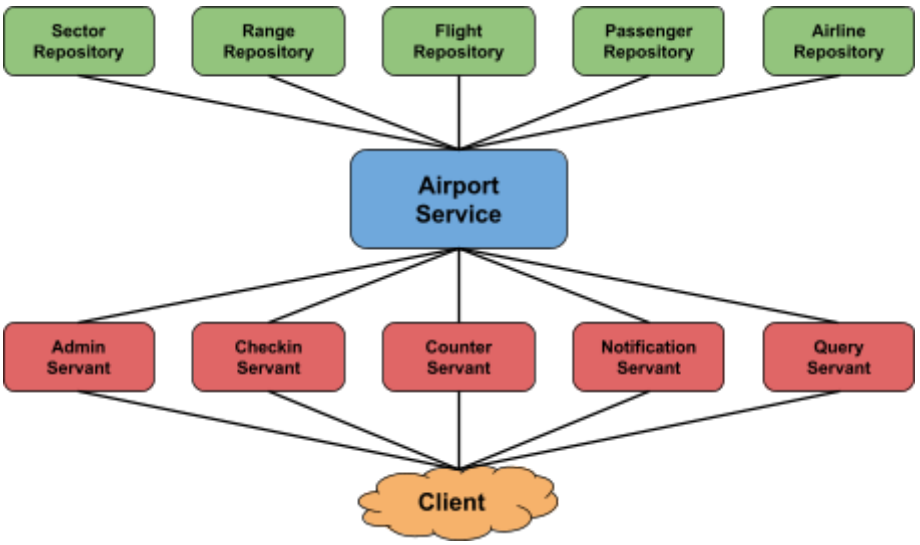


Fig 4: Esquema lógico de implementación

Criterios aplicados para el trabajo concurrente

A la hora de hacer un análisis concurrente del servicio, contamos con diversas estrategias:

- **Métodos *synchronized***: Para los casos donde se debía obtener una copia de los datos en un repositorio (como cuando se buscan todos los sectores), se decidió proteger el acceso con métodos *synchronized* sobre el uso de colecciones concurrentes (como ConcurrentHashMap) para garantizar la consistencia de la lectura de todos los datos.
- **Utilización de clase `Flight` como recurso de concurrencia**: Durante el análisis del sistema notamos que contar con estructuras de datos que guarden el par (vuelo,mostrador) contenía información redundante, ya que el mostrador a su vez cuenta con la información de un vuelo y que daba lugar a situaciones de concurrencia que obligaban a bloquear el sistema para cualquier consulta a un rango. En este sentido, pensamos en utilizar una clase `Flight`, la cual almacene el `Range` donde el mismo fue/está siendo asignado, para tener una única fuente de verdad y sincronizar esa fuente.
- **Utilización de `CountDownLatch`**: En acciones server-side streaming, se utiliza una de las implementaciones de un semáforo (`Latch`) para bloquear el main thread hasta que termine la respuesta del servidor.

Consideraciones adicionales

Con respecto al comportamiento de los rangos en un sector, al agregar un nuevo rango de mostradores (**1.2 - *addCounters***) o bien al liberarlo (**2.4 - *freeCounters***), si se cumple que dicho rango es consecutivo o inmediatamente anterior a otro rango libre, ambos rangos se "mergean", se convierten en un único rango. En otras palabras, **dado el caso en que cierto sector A tiene los rangos (S_1-E_1) y (S_2-E_2), si ambos están libres y**

$$S_2 = E_1 + 1$$

entonces se convertirá en un único rango (S_1-E_2) para el sector A.

Con respecto a la acción **2.2 - *listCounters***, se decidió que un rango de mostradores aparezca en el listado si y sólo si está completamente contenido dentro del rango solicitado. Es decir, **un rango de mostradores (S-E) será listado en la respuesta siempre y cuando**:

$$counterFrom \leq S \wedge counterTo \geq E$$

Potenciales puntos de mejora y/o expansión

Algunas potenciales mejoras encontradas durante el desarrollo del proyecto se enumeran a continuación:

- **Reutilización de mensajes en archivos .PROTO**: Durante el desarrollo de los archivos mencionados encontramos que muchos mensajes contenían campos similares. Si bien se logró parametrizar alguno de estos casos, como el rango de mostradores, es esperable que exista un mejor análisis de esta reutilización.
- **Clase `AirportServiceImpl`**: Entendemos que si bien resulta funcional contar con una única clase con todas las funciones a utilizar por los *servant*s, puede existir una implementación superadora donde no todos ellos consuman un mismo y único servicio.
- **Salida de notificaciones**: Si bien como parte de la consigna resulta esperable que algunas salidas sean directo a consola, entendemos que podría ser superador que estas salidas salgan directo a un archivo para su persistencia y su posible comparación.
- **Diagramas explicativos**: A la hora de entender cómo resulta el funcionamiento del sistema en su conjunto puede resultar un tanto agobiante contar con la explicación de cada uno de los servicios en modo de texto, en este sentido sentimos que contar con un diagrama de los servicios puede ayudar al cliente.
- **Automatización de test**: Si bien se cuenta con test de casos de usos de los clientes, se podría extender estos tests para correr todos en un solo ejecutable o incluso extender para que las salidas de cada test cuenten con un archivo de comparación, como es el caso de *SimplyRegisterAirline.sh*.