Instituto Tecnológico de Buenos Aires

Trabajo Práctico Especial 1 - Mostradores de Check-in

72.42 - Programación de Objetos Distribuidos

Autores:

Ballerini, Santiago[1] - 61746
Martone, Gonzalo Alfredo[2] - 62141
Bafico, Juan Cruz [3] - 62070
Bosetti, Franco [4] - 61654

Docentes:

Marcelo Emiliano Turrin

Franco Román Meola

Fecha de entrega: 28 de Abril de 2024

[1] sballerini@itba.edu.ar - @Cuinardium (github)
[2] gmartone@itba.edu.ar - @ImNotGone (github)
[3] jbafico@itba.edu.ar - @jBafico (github)
[4] fbosetti@itba.edu.ar - @francobosetti (github)

Índice

1. Decisiones de diseño e implementación de los servicios

Para la implementación de los servicios, implementamos dos tipos de clases. Una de estas son los *services*, que son las implementaciones de los servicios de *grpc* y otras son los *repositories* que manejan el estado de la aplicación. Nuestra intención es separar todo lo relacionado con el estado y el manejo de concurrencia en los *repositories* y que los *services* solo se encarguen de recibir los llamados *rpc* y realizar las acciones necesarias y devolver la información precisada.

Esto se logra inyectándoles por *constructor* todos los *repositories* que los *services* necesitan. Una de las ventajas de esto es que debido a que los *services* interactúan con interfaces, se puede cambiar la implementación de los *repositories* sin tener que tocar nada en el *service*.

El manejo de estado fue segmentado en 4 interfaces: **CheckinRepository**, **PassengerRepository**, **CounterRepository** y **EventManager**.

- **CheckinRepository:** Este repositorio maneja el historial de checkins de passajeros manteniendo el orden de llegada .
- **PassengerRepository:** Este repositorio maneja todos los pasajeros esperados por el aeropuerto.
- **CounterRepository:** Este repositorio maneja todo el estado relacionado a los sectores existentes, cuales mostradores pertenecen a que sector y a que vuelos y aerolíneas están asignados si es que lo estan. Este repositorio también maneja lo relacionado a la cola de pasajeros en cada rango de mostradores y la cola de asignaciones pendientes.
- **EventManager:** Esta clase maneja todo lo relacionado a las notificaciones del sistema. Desde cualquier *service* que lo precise, basta con llamar al método *notify* de esta clase y se mandara la notificación al cliente que este subscrito.

En un principio pensamos en manejar la lógica de la cola de pasajeros y cola de asignaciones en clases distintas al **Counter Repository** pero para poder manejar lo relacionado a la concurrencia dentro de una sola clase decidimos juntar toda la lógica dentro de una sola clase.

Para las interfaces **CheckinRepository**, **PassengerRepository** y **EventManager** solo hemos realizado una implementacion que son las siguientes: **CheckinRepositoryImpl**, **PassengerRepositoryImpl** y **EventManagerImpl**.

Para el **CounterRepository** hemos realizado dos implementaciones: **CounterRepositorySynchronized** y **CounterRepositoryImpl**. Actualmente utlizamos la implementacion **CounterRepositoryImpl**.

En cuanto al testeo, hemos realizado 159 *tests* unitarios (91 % *code coverage*) para cubrir y testear el funcionamiento de los servicios y repositorios presentes en el trabajo. Los *tests* de integración entre los clientes y el servidor los hicimos de forma manual, pese a que nos hubiera gustado implementar alguna forma de automatizarlo.

2. Criterios aplicados para el trabajo concurrente

Como fue mencionado anteriormente, diseñamos el servidor de manera tal que la concurrencia se maneja totalmente dentro de los repositorios. Las decisiones relacionadas a la concurrencia de cada repositorio son las siguientes.

- **CheckinRepositoryImpl:** Para la implementación de esta clase optamos por el uso de una **ConcurrentLinkedQueue** que es *thread-safe* y además respeta el orden de inserción. La iteración sobre esta colección es *weakly consistent*, lo que nos permite iterar de manera concurrente sin preocuparnos por **ConcurrentModificationException**.
- **PassengerRepositoryImpl:** Esta clase fue implementada sobre un **ConcurrentHashMap** que asocia un *booking* con la informacion del pasajero. De esta forma se priorizan las busquedas por *booking*. Se eligió esta colección debido a que las lecturas suelen ser no bloqueantes y también sus *iteradores* son *weakly consistent*
- **EventManagerImpl:** Para la implementación de esta clase tambien utlizamos un **ConcurrentHashMap** que asocia aerolíneas con el **StreamObserver** por donde se manda el *stream* de notificaciones. Debido a que principalmente se va a leer sobre dicho mapa, las características de casi no bloquear en el *read* son apropiadas para este uso. Debido a que los **StreamObserver** no son *thread-safe* usamos el bloque *synchronized* sobre el mismo para sincronizar llamados.
- **CounterRepositorySynchronized:** Esta fue la primer implementación que hicimos del **CounterRepository**, esta maneja concurrencia simplemente anotando todos sus métodos como *synchronized*. Sin embargo no estábamos satisfechos con la simpleza de esta implementación por lo que decidimos implementar uno nuevo.
- **CounterRepositoryImpl:** Al implementar el **CounterRepository** debíamos manejar los sectores, *counters* asignados, vuelos asignados y pasajeros en cola, para esto decidimos hacer uso de varios tipos de colecciones distintas. **Map <String, TreeSet<CountersRange> >** para mantener los *counters* asignados a los distintos sectores. **Set<String>** para mantener el historial de los vuelos asignados. **Map<String, Queue<Assignment> >** para guardar la cola de vuelos que tienen asignación de *counter* pendiente sobre un sector. **Map<Range, Queue <String> >** donde se mantienen los pasajeros esperando acceder a un *counter*. Para mantener la consistencia en todas estas estructuras, utilizamos *ReentrantReadWriteLocks*. El uso de estos *locks*, si bien permiten que 1 solo entre en escritura, en lectura pueden acceder varios procesos al mismo tiempo por lo que debería ser mas performante que un *mutex* normal.

3. Potenciales puntos de mejora y/o expansión

Al planear como se iba a implementar todo el proyecto, decidimos primero centrarnos en implementar los *services* solamente trabajando sobre las interfaces de los *repositories*, dejando para mas adelante su implementación.

Para **PassengerRepository**, **CheckinRepository** y **EventManager** esto no fue un problema. Sin embrago, para el **CounterRepository** esto nos brindo una gran complicación ya que subestimamos la complejidad del mismo por lo que nos quedo un archivo de implementación con una sola clase muy grande, con muchas responsabilidades y con código poco mantenible. Como mejora podríamos revisar las colecciones y la estructura general que decidimos utilizar para guardar la información de **CounterRepository** esta es una de las principales razones por las cuales se acomplejizo el desarrollo de la clase mencionada, ya que nos condiciona en la implementación de los *locks*.

Otro punto de mejora es, dentro del **PassengerRepositoryImpl** y **CheckinRepositoryImpl** para hacer mas fácil el manejo de la concurrencia hicimos una sola colección en cada una en vez de tener varias colecciones auxiliares. Esto genera que haya ciertos patrones de acceso que son ineficientes.

Adicionalmente, otra mejora posible sería agregar el **Global Exception Handler**. En nuestro trabajo cada *service* se encarga de manejar las excepciones, por lo que tener este *handler* haría que el código sea mas limpio.