Programación de Objetos Distribuidos Trabajo

Práctico 1: ‘Estación de tren’

Integrantes del grupo:

Bernardo Zapico - **62318**
Tomás Guillermo Sheffer - **63393**
Alejo Padula Morillo - **63571**

Decisiones de diseño e implementación

En primer lugar, decidimos representar el ID de la plataforma (autoincremental) usando una variable de clase de tipo AtomicInteger para garantizar la unicidad de ID's entre plataformas. También decidimos no guardar estados en las plataformas sobre la elección de los trenes (esto es algo complementamente "Train-side"), la plataforma solo guarda el tren cuando este está estacionado, no cuando algún tren quiere ir hacia ella.

Para el manejo de errores, decidimos utilizar la clase provista por la cátedra: **GlobalExceptionHandlerInterceptor** y cuando se lance una excepción en el server, si es un error del cliente, se lanzará una excepción custom que será atrapada por este Handler para enviarle el código correspondiente. Si es un error generado por alguna de las validaciones de nuestras clases, lanzamos una excepción de Java que será atrapada como "Exception" con código de "Internal", indicando que es nuestro.

Se decidió manejar toda la lógica de los trenes de tracción simple y doble en la misma clase. Una idea era intentar hacer una clase abstracta con una clase hija para cada tipo de tracción pero esto complicaba bastante algunas revisiones ya que se necesitaba ciertos métodos generales en la clase abstracta que quizá no tenían sentido para las clases hijas.

Se decidió del lado del Servidor separar claramente la lógica de la aplicación de la capa de comunicación. Para ello se definieron 2 paquetes principales, la del "Model" que contiene los objetos y lógica interna del servidor, y la del "Servant" que contiene las implementaciones de los servicios de gRPC. En el caso del Board se tuvo que representar el estado interno de las plataformas con la clase BoardView para mantener esta separación. Luego para el manejo de flujo de datos con el Cliente se usó el patrón Observer donde cada uno se suscribe (con el método LiveBoard) para reaccionar ante los cambios que requieren volver a enviar información.

Criterios aplicados

Para el criterio de sincronización de métodos (o partes de estos) en una clase, decidimos proteger las partes en el código que interactúan con el **estado interno accedido por varios métodos**, teniendo en mente que varios threads podrían llamar a distintos métodos a la vez. Además de la coherencia de estado interno, era importante mantener una buena coordinación entre diferentes trenes y plataformas, por ejemplo, no tendría sentido que si quiero cerrar un andén, bloquee todos andenes para evitar que alguien más realice otra operación sobre ese único andén. Primero se optó por proteger el estado de las clases Platform y Train para **su propia instancia**.

Para el caso de la clase Platform, protegimos bajo el mismo synchronized (de this) las partes que interactúan con las variables que indican su estado (un Enum representando el estado actual) y/o su tren asociado (solo en caso de tener uno estacionado), por eso los métodos como parkTrain, departTrain y toggleState están sincronizados, porque todos estos, de correr a la vez podrían llevar a inconsistencia de datos (Se podría correr dos toggleState en dos thread distintos sobre un train en IDLE y que pase a BUSY por los dos, cuando lo que debería hacer es pasar IDLE -> BUSY -> IDLE).

Para el caso de la clase Train, protegimos con el mismo synchronized (de this) las partes que interactúan con sus pasajeros, su estado (un enum representando el estado actual) o sus plataformas asociadas. Por eso protegemos métodos como associatePlatform, associateTwoPlatform, checkDisembarkAllOperation, disembarkAllPassengers y boardAndLeavePlatform porque dependen mucho de qué otro método no cambie el estado interno del tren para funcionar.

Finalmente, y con la idea de que las clases Train y Platform son Thread safe para su propia instancia, el objetivo fue coordinar las múltiples instancias de estas que se manejaron en la clase Station. Una coordinación necesaria e implementada está sobre nuestra cola de trenes esperando para estacionar (awaitingTrains), ya que hay varios métodos que dependen mucho de la consistencia de esta en el tiempo que la acceden.

En principio, se utiliza una ConcurrentLinkedQueue, que garantiza la atomicidad entre operaciones unarias (add, contains, peek, pull, ...) pero no lo garantiza con bulk operations (getAll, addAll, ...) o, más importante, con el iterator(). Decidimos sincronizar sobre la instancia de Station en las partes donde se utiliza esta cola de trenes por más que solo una operación unaria, por ejemplo en el método addTrainOrGet, utilizamos un "contains" y luego un "add" sobre la cola, ambas son operaciones unarias pero... podría pasar que en el medio de ellas el elemento no exista más en la cola (o lo opuesto), dejando inconsistente el estado y llevando a un "add" erróneo. Finalmente, también se sincronizó sobre las partes donde iteramos la cola, ya que según la documentación, el estado puede cambiar mientras se está iterando y esto no nos daría el comportamiento que esperamos.

Se intentó evitar el uso de synchronized anidados entre Platform y Train, ya que estos pueden llevar a deadlocks de usar un anidamiento y luego el opuesto. Ej de deadlock:

En un método 1:

```
synchronized (train) {
    synchronized(platform) {
        /*code here*/
    }
}
```

En un método 2:

```
synchronized (platform) {
    synchronized(train) {
        /*code here*/
    }
}
```

Si un thread 1 en el método 1 pasara el primer synchronized para una instancia de train A, y no llegara a cruzar el synchronized de una instancia de platform B, mientras que a su vez un thread 2 en el método 2 cruzara el primer synchronized para la instancia platform B pero no llegara a cruzar el synchronized para la instancia train A habría un deadlock. Esto podría pasar ya que tanto Train cómo Platform son clases que están fuertemente ligadas y al mismo nivel (ninguna es un repositorio de la otra), entonces buscamos evitar lo máximo posible estos anidamientos, y cuando los hicimos, respetamos el mismo orden de anidamiento para evitar deadlocks como el del ejemplo mencionado arriba.

Puntos de mejora y expansión

Una mejora acerca de la implementación es encontrar una misma clase que nos sirva para modelar las listas de ‘watingTrains’ y ‘abandonedTrains’. Podríamos hacer una Clase de colección customizada para obtener o reforzar ciertos aspectos funcionales y de performance, y descartar los que no nos importan. Esta colección podría hacer ciertas validaciones dentro de sí misma para encapsular la lógica, reducir la posibilidad de errores de sincronización y mejorar el rendimiento. Como ejemplo práctico tenemos el de “agregar solo si no está”, ya que en nuestra implementación estamos haciendo la validación antes de insertar el elemento y nos ha obligado a implementar lógica externa con Synchronized. Otra posible mejora es garantizar que no haya inanición y evitar bloqueos permanentes por medio de los Reentrant-Locks con colas FIFO (para que todos Threads tengan chance en algún momento) y Timeouts. Otras mejoras de implementación es hacer uso de alguna convención de nombres de métodos, ya que hasta el momento no usamos y para seguir expandiendo el proyecto sería necesario para evitar confusiones. El NotificationServant fue omitido por falta de tiempo, había problemas con su manejo en el lado del servidor y en general se empezó a consolidar demasiado sobre la hora. Además de que faltaría adaptar al cliente para el nuevo funcionamiento del servidor.

Una expansión puede ser la capacidad de gestionar múltiples estaciones de manera centralizada, esto permitiría a la aplicación modelar líneas ferroviarias y obtener de manera sencilla datos globales para la toma de mejores decisiones desde cada estación. Esto permitiría encontrar dónde se ubica un tren en particular o calcular el tiempo medio entre estaciones y así proyectar horarios de llegada para los pasajeros. Otra expansión importante sería el manejo de roles, se crearían uno por cada orden de jerarquía. Por ejemplo podríamos tener el Administrador del Sistema con acceso total; en caso de gestionar múltiples estaciones debería haber un Operador de estación (tendría el acceso a todas las capacidades actuales del sistema); debería existir el rol de

Observador para suscribirse al Live-Board y el del Conductor de tren o personal ferroviario para pedir una plataforma, confirmar su partida o en caso de ser uno de doble-tracción, conocer el estado de su otra mitad.