Programación de Objetos Distribuidos

Trabajo práctico Nº1

Grupo 6

Integrantes

- Dey, Patrick 59290
- Kim, Azul 60264
- Lombardi, Matías 60527
- Mihura, Uriel 59039
- Rosati, Santos 60052

Decisiones de diseño e implementación de los servicios

Se creó el proyecto a partir del arquetipo provisto por la cátedra: *pod-archetype*. Este se divide en 3 módulos: *api*, *client* y *server*.

Dentro del módulo *api* se encuentran los modelos, las excepciones y las interfaces remotas que comparten tanto el cliente como el servidor. Además, cabe destacar la interfaz *NotificationHandler*, el *callback* para avisar que hay una nueva notificación, que cuenta con métodos necesarios para manejar las notificaciones para el usuario. También es importante resaltar que se crearon modelos específicos para cierto tipo de retornos. Entre ellos se encuentran *ResponseRow*, que por términos de seguridad cuenta únicamente con la categoría de la fila y las iniciales de los pasajeros que la ocupan; *ResponseAlternativeFlight*, que contiene toda la información relevante a la hora de listar los vuelos alternativos; y por último, *ResponseCancelledList*. Este modelo fue implementado ya que la respuesta que devuelve *reticketing* contiene 2 valores: la cantidad de tickets reasignados y una lista de los tickets que no pudieron reasignarse, es por esto que no se devuelve directamente una lista de tickets.

Dentro del módulo *client* se encuentran los 4 clientes solicitados en el enunciado, donde cada uno se encargará de parsear su respectivo input y con el, realizar el request necesario al servicio. Todas las llamadas al servicio cuentan con el mensaje de error correcto según la excepción obtenida.

Por último, el módulo *server* es quien cuenta con la implementación de los servicios ofrecidos que serán llamados por el cliente. Para este trabajo se decidió implementar un *Servant* por cada interfaz remota, para que sea más clara la separación de funcionalidades. Todos comparten una instancia de *ServerStore* que tiene almacenada toda la información de los modelos, vuelos y notificaciones en tiempo de ejecución. Además, cuenta con algunos métodos compartidos entre servicios y el *threadpool* que se encarga de las notificaciones.

Dentro del *Store*, los vuelos están almacenados en 4 mapas diferentes. El primero se corresponde con la relación código-estado, mientras que los otros 3 guardan el código y el vuelo relacionado según su estado (es decir, se cuenta con un mapa diferente para cada estado posible definido en el enunciado), para facilitar la obtención de los datos. Si bien esto aumenta la complejidad espacial al repetir información, se decidió priorizar la complejidad temporal. A su vez, se cuenta con 2 mapas más, uno para los modelos de avión y otro para las notificaciones. El primero relaciona nombres de modelos con el modelo, mientras que el segundo guarda para cada código de vuelo y cada pasajero dentro de ese vuelo (siempre y cuando cuente con un *ticket* y se haya registrado para recibir notificaciones), una lista de los *callbacks* recibidos (ya que como se menciona en el enunciado, un mismo cliente se puede registrar varias veces para recibir notificaciones).

Como comentario adicional, se desea aclarar que el método *changeFlightState* dentro de *FlightManagerServiceImpl* no fue modularizado debido a que las funciones que realiza son muy específicas al método en cuestión. Es por ello que se decidió dejarlo como está para no agregar dificultades extra.

Finalmente, se realizaron tests unitarios para las implementaciones de los servicios, los mismos se encuentran ubicados en el directorio *test* del módulo *server*.

Criterios aplicados para el trabajo concurrente

Primero se realizó un análisis de las zonas que podrían llegar a ser afectadas por la funcionalidad concurrente. Se debe garantizar la sincronización, disponibilidad y coordinación en el proyecto. Para ello, identificamos los métodos que deberían ser protegidos, para evitar, por ejemplo, que se obtenga información desactualizada o que dos llamados diferentes intenten modificar la misma variable al mismo tiempo.

Como resultado de este análisis, se decidió proteger tanto el estado como los asientos dentro los vuelos, así como también, los modelos de aviones y las notificaciones. Para la protección de los vuelos (como se mencionó, estado y asientos) se optó por *ReentrantLock*, ya que es la solución de menor complejidad para el problema en cuestión. Se cuenta con 2 *locks* diferentes, uno para el estado y otro para los asientos. Para el caso de modelos, también se cuenta con un *ReentrantLock*, en donde se *lockea* cada vez que se quiera agregar/retirar un modelo.

En el caso de notificaciones se utiliza un bloque *synchronized* sobre la instancia del mapa que los almacena para obtener el valor asociado (se podría haber usado un lock adicional, pero en este caso, no agregaba nada que el bloque no solucionara). Una vez obtenido el mapa de notificaciones del vuelo correspondiente (aquel que asocia el vuelo, con sus pasajeros y finalmente, con sus respectivos callbacks) se sincroniza sobre él y se procede a obtener la notificaciones a enviar.

Al momento de hacer cualquier acción sobre un vuelo, la forma de obtenerlo es muy parecida a lo descrito en el párrafo anterior para las notificaciones. Cabe destacar que se usa synchronized sobre todos los mapas necesarios (ya que como se mencionó, no agrega nada utilizar un *lock*). Primero se obtiene el estado del avión y luego, con ese estado, es fácil saber a qué mapa hay que recurrir para obtener el vuelo, ya que se encuentran separados por estado en 3 instancias diferentes.

Potenciales puntos de mejora y/o expansión

Como potenciales puntos de mejora y/o expansión se pensó en las siguientes.

La primera sería implementar de manera más granular el lockeo por *Row* en vez de ser a nivel *Flight*, pero, como agrega mucha complejidad a la resolución, se decidió dejar la implementación actual que cumple con lo pedido.

En segundo lugar, se podría examinar la posibilidad de implementar dos veces el lock a la hora de agregar un modelo de avión o vuelo, ya que en la implementación actual se realiza la creación del modelo/avión mientras se encuentra *lockeado*, sin embargo, no es una creación costosa por lo que se decidió dejar con un único lock.

Otra posibilidad en cuanto a la complejidad espacial, podría ser dividir ciertas tareas paralelizables en threads, por ejemplo al momento de fijarse que vuelos alternativos existen

Por último, se piensa que una mejora a la implementación podría ser que los modelos fueran *thread-safe* en vez de manejarlo desde afuera, sin embargo agregaría mucha complejidad a la solución.