ITBA - "72.42 - Programación de Objetos Distribuidos"

Trabajo Práctico Especial 1: Reservas de Atracciones de Parques Temáticos

Alumnos:

De Simone, Franco	61100
Dizenhaus, Manuel	61101
Anselmo, Sol	61278

Índice

1. Decisiones de diseño e implementación de los servicios

Para este trabajo, se requirió implementar un sistema cliente - servidor con el uso de gRPC como mecanismo de comunicación entre ambas. La primer decisión de diseño giró en torno al guardado de la información del sistema. Al no ser necesario tener que tratar con persistencia en disco, se realizó pleno uso de la memoria para le almacenamiento volátil de la información. A nivel código, se crearon colecciones para mantener la información ordenada en 3 listas: `attractions`, `reservations` y `passes`. Autoexplicativas, estas 3 mantenian la información de las 3 entidades principales que manejaba el trabajo. A su vez, se crearon modelos dentro del servidor para poder tratar con esta persistencia.
Para mantener la estructura del proyecto de forma ordenada, se creó una clase `parkRepository`. Esta clase manejaba las 3 colecciones mencionadas previamente. Además, esta abstracción nos permitió que los servicios se preocupen unicamente del manejo de los pedidos, y no de lo que fuera concurrencia ni impacto en la "base de datos".
Creímos importante mantener la abstracción del uso de *gRPC* limitado a la declaración de los *requests/responses*. Todos los modelos están basados en las estructuras declaradas de *request/response* dentro de la consigna.
Otra decisión de diseño fue respecto a la reubicación de reservas. Al agregar capacidad a una atracción, si la cantidad de reservas en estado PENDING de un slot excedía la capacidad, entonces la cantidad sobrante debía reubicarse a slots subsiguientes, y quedar en estado PENDING a la espera de confirmación por parte del usuario. Sin embargo, nos encontramos ante la posibilidad que, en el mientras tanto, ocurrieran más reservas en un slot al cual había sido reubicada una reserva, llenándolo y dejando el aviso de reubicación obsoleto, y haciendo surgir la necesidad de volver a reubicar la reserva. Para evitar esto, decidimos hacer que, al reubicar una reserva, "guardarle el lugar.ª la misma, decrementando en uno el cupo disponible para ese slot; si el usuario luego confirma la reserva, se ratifica ese cupo ocupado, y si la cancela, se lo libera, incrementando en uno y devolviendo esa cantidad a su estado original.

2. Criterios aplicados para el trabajo concurrente

Este trabajo presentaba un desafío particular en cuanto a concurrencia. Intencionadamente, existían pedidos que trabajan sobre la misma entidad que manejaba, por ejemplo, las atracciones. Por ejemplo, si dos *threads* deseaban agregar una atracción en simultaneo (o cambiar algún elemento del estado), es necesario que solo una agregue a la vez dado que no proteger esta zona de memoria puede generar problemas.
Por ello, se apelo al uso de la implementación de *ReentrantReadWriteLocks* de Java. Los mismos manejan, como estudiamos en la teórica, la concurrencia de threads para impactar en una zona de memoria. Esta implementación particularmente permite dos operaciones: Lectura y escritura. N threads pueden leer el elemento en memoria, pero si hay un escritor con el lock, unicamente ese thread estará habilitado para manejar lo que se encuentre protegido por el mismo.
De esta forma, al tratar de operar sobre una misma colección para leer, todos los threads que lo requerían pueden hacerlo dado que no se modificarán los contenidos del mismo. También es importante notar que la única forma de modificar las instancias de los elementos de la lista es mediante el acceso por la colección, por lo que la posibilidad de modificar un elemento de la colección por referencia sin tener el lock no es posible.
Una cuestión interesante para analizar de la implementación de *ReentrantReadWriteLock* es que pasa si siguen llegando *reading threads*, y un *writing thread* queda esperando eternamente. Esto se puede producir, y es popularmente conocido como *starvation*. De no modificar la implementación base, e instanciando los locks simplemente con:

```
ReadWriteLock lock = new ReentrantReadWriteLock();
```

es altamente probable que se genere una situación donde haya *starvation*. Pero por ello existe un parámetro de *fairness* para indicar a los locks que manejen un pseudoorden a la hora de permitir el

uso del lock. Justamente el parámetro *fairness* se indica de manera explícita a la hora de instanciar el lock:

```
boolean fairness = true;
ReadWriteLock lock = new ReentrantReadWriteLock(fairness);
```

Realizar esto nos permite asegurar que, si bien no es exacto, se maneja un pseudoórden de llegada. Citando a la documentación de Java:
"When constructed as fair, threads contend for entry using an approximately arrival-order policy. When the currently held lock is released, either the longest-waiting single writer thread will be assigned the write lock, or if there is a group of reader threads waiting longer than all waiting writer threads, that group will be assigned the read lock." [1]

3. Potenciales puntos de mejora y/o expansión

Algunos puntos donde creemos que el trabajo podría escalar y mejorar:

- **Persistencia**: Está claro que presenta una capa de dificultad superior, y que excede el *scope* académico que busca este trabajo, pero sería interesante llevar este sistema a un ejemplo con persistencia en alguna base de datos (ya sea hosteada o local). Por un lado, sería positivo para hacer el trabajo mas real", pero también ayudaría a explorar como manejan las bases de datos sus sistemas de concurrencia, dado que necesitan una sincronización perfecta para poder operar correctamente.
- **Exploración en profundidad de gRPC**: gRPC se presentó como una novedad absoluta para el grueso de los alumnos, y, con una sintaxis extremadamente simple, permitió generar toda una arquitectura cliente servidor. Sería muy interesante explorar el potencial que tiene este sistema, y como puede permitir que el trabajo escale a otro nivel.
- **Manejo por fecha**: Si bien está claro que el foco del trabajo está en otro punto, hubiera sido interesante manejar fechas reales como tipos de dato, y no solo un *int* representativo.
- **Agregar elementos únicos**: La consigna pedía únicamente agregar elementos a partir de un *.csv* donde cada línea era representativa de una atracción/pase. Sería útil también, pensandolo para un funcionamiento real, tener la posibilidad de agregar de a un elemento por vez, y así

Referencias

[1] Oracle. *'Class ReentrantReadWriteLock'*. Revisado: 13/09/2023.