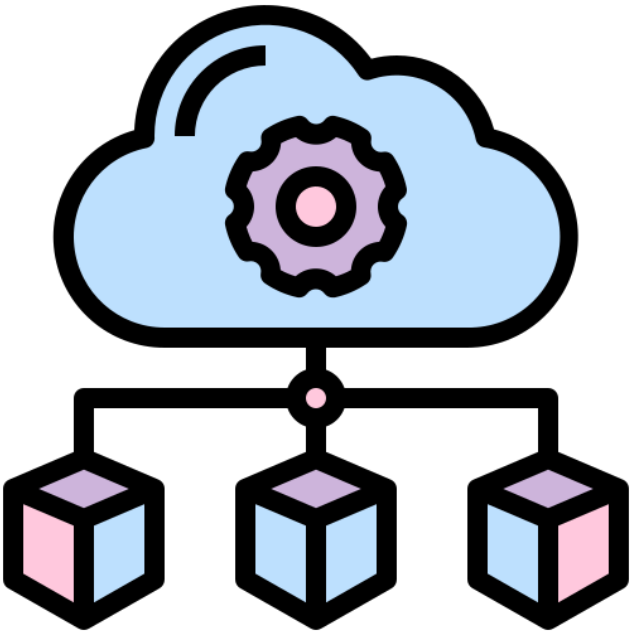
Trabajo Práctico 1 - Grupo 4

Mizrahi, Thomas (60154)
tmizrahi@itba.edu.ar

Chayer, Iván (61360)
ichayer@itba.edu.ar

Di Toro, Camila (62576)
cditoro@itba.edu.ar

gRPC



Profesores:
Marcelo Emiliano Turrin
Franco Roman Meola

72.42 Programación de Objetos Distribuidos
Instituto Tecnológico de Buenos Aires
Primer cuatrimestre 2023

Introducción

En este breve informe, vamos a comentar algunas de las decisiones de diseño que tomamos a lo largo de este trabajo práctico.

Manejo de errores

En la fase inicial de desarrollo de nuestro proyecto, se adoptó la estrategia de definir varios tipos de `enum` dentro de los archivos `.proto` para manejar los errores que podían surgir durante la comunicación entre clientes y servidores a través de gRPC. Sin embargo, a medida que avanzamos en el proyecto, nos enfrentamos a un incremento en la cantidad de `enums` y a la aparición de errores repetitivos. Esta complejidad creciente dificultó la tarea de registrar y gestionar adecuadamente los errores, en muchos casos resultaba más práctico lanzar excepciones en lugar de guardar y configurar el estado de error de forma manual.

Por consiguiente, tomamos la decisión de implementar un componente denominado `ExceptionHandler`. Este componente desempeña el rol de *middleware* y nos permite interceptar excepciones en el servidor, tal de catchearlas y responder a los clientes de manera coherente ante situaciones de error.

Un aspecto fundamental a considerar es que gRPC define códigos de error estandarizados en una clase denominada `Status`, los cuales deben ser enviados al cliente en caso de que se produzca un error. Estos códigos de error pueden ser acompañados por un mensaje descriptivo. Sin embargo, optamos por evitar que el servidor genere un mensaje en caso de error para que el cliente pueda identificar fácilmente nuestros propios códigos de error y tomar decisiones apropiadas sobre la presentación de mensajes al usuario.

En nuestro diseño, todas las excepciones generadas en el lado del servidor extienden la clase `ServerException`. Estas clases especializadas tienen la capacidad de realizar un mapeo preciso entre el `Status` y el código de error correspondiente definidos por nosotros. De esta manera, se logra una estandarización en el manejo de errores y se facilita la tarea de comunicar de manera efectiva a los clientes sobre el tipo de error ocurrido, sin necesidad de generar mensajes adicionales en el servidor.

Estructuras de datos

El servidor precisa poder almacenar un conjunto complejo de datos, dada a la alta relación entre los datos y la necesidad de concurrencia. Por esto, decidimos estructurar los datos de las reservas en forma de árbol, donde cada nivel de arriba hacia abajo acota más las reservas a encontrar en esas ramas del árbol, siendo las hojas las reservas en sí. Los niveles del árbol son manejados por distintas clase, y de arriba hacia abajo estas son:

- `AttractionHandler`: La raíz del árbol, se encarga de manejar atracciones, y divide las reservas por atracción. En base a un nombre de atracción, puede retornar una atracción, el siguiente nivel del árbol.

- `Attraction`: Representa una atracción del parque, y divide los datos por día del año. En base a un día del año, puede encontrar el `ReservationHandler` que maneja las reservas de dicha atracción para dicho día.
- `ReservationHandler`: Maneja las reservas de una atracción para un determinado día. Divide las reservas por slot horario, y entre estas las puede rápidamente encontrar por ID del visitante.

Cada clase también maneja la concurrencia a sus accesos. Decidimos no dividir los datos en más clases ya que esto habría resultado en una lógica de sincronización mucho más compleja, en especial para el algoritmo de realojación de reservas.

En cuanto a las colecciones Java usadas, se hizo uso extenso de mapas, dado que permiten rápido acceso por claves. En aquellos lugares donde estas "claves" eran valores secuenciales con alta probabilidad de que, en casos de uso reales del sistema, hayan pocos baches (en particular, días del año y/o slots horarios), se usaron arreglos en vez de mapas, indexados por dicho dato. Esto lo podemos ver en las tres clases anteriormente mencionadas.

Una colección particular utilizada fue dentro de `ReservationHandler`, donde inicialmente se usó una `Queue` para reservas pendientes (ya que así se pueden remover en orden de llegada). Sin embargo, al momento de cancelar o confirmar una reserva pendiente esta se debe acceder por ID del visitante, lo que resembla las necesidades de un `HashMap`. Para satisfacer ambas necesidades de forma eficiente en una sola colección, usamos el poco conocido `LinkedHashMap`, que es un mapa que preserva el órden de inserción.

Manejo de Concurrencia

Se usaron varias estrategias para evitar problemas de concurrencia. Entre estas:

- Clases Readonly: Clases que son de solo lectura, como por ejemplo `Reservation` y `ConfirmedReservation`. Notar que por más que estas sean *read-only*, no todos sus campos (en particular `Ticket` y `Attraction`) son inmutables.

Inicialmente se usaba una clase mutable `Reservation` que incluía, entre otras cosas, el estado (confirmada o pendiente) y el slot. El problema con esto es que si, por ejemplo, el servicio de notificaciones recibe un evento, entre que se envió el evento y que se procesó podrían haber cambiado los valores de la reserva, produciendo una condición de carrera. Si se creaba una reserva pendiente y se la confirmara instantes después, podría ocurrir que cuando se procese el evento de reserva creada se la vea como ya confirmada, resultando en una notificación de "reserva creada como confirmada" seguida de otra notificación de "tu reserva pendiente ha sido confirmada".

- Arreglos *read-only*: En el proyecto tenemos varios arreglos de colecciones. Inicialmente, estos arreglos empezaban con todos sus elementos en `null` y estos se iban creando a medida que era necesario. Sin embargo, esto significaba que era necesaria sincronización para todo acceso, tal que dos threads no vean ambos una

posición del arreglo en `null` e intenten crear y guardarlo al mismo tiempo. Por esto decidimos que estos arreglos empiecen con todos sus elementos ya instanciados, resultando en mejor concurrencia. Ejemplos en: `Attraction`, `ReservationHandler`.

- Colecciones concurrentes: En particular de `ConcurrentHashMap`, que nos alivia tener que preocuparnos de sincronizar tanto lecturas como escrituras al mapa, cosa que es particularmente importante cuando es necesario recorrer el mapa. De todos modos, estos solo garantizan que sus métodos sean operaciones atómicas. Si quiero hacer un "crear si no existe", no se puede hacer `get(key)` y si es `null` hacer `put(key, value)`, eso son dos operaciones atómicas separadas. Para evitar esto, debemos usar `putIfAbsent(key, value)`. Ejemplo en `AttractionHandler`.

- Uso de `synchronized`: Nos permite adquirir un `lock` sobre un objeto. Se usa, por ejemplo, en `ReservationHandler`, para asegurarse que no se choquen dos pedidos de reservas, o que esperen en el caso que esté corriendo el algoritmo de realojación de reservas.

También se usa en `NotificationServiceImpl`, en la clase interna que se encarga de traducir los avisos de notificaciones a llamados `onNext()` del `StreamObserver` de gRPC, dado que esos llamados deben ser sincronizados porque `StreamObserver` no garantiza ser *thread-safe*.

Un patrón interesante es el que surgió en la clase `Ticket`. Originalmente, el método `AttractionHandler.makeReservation()` aseguraba atomicidad con un `synchronized (ticket)`, ya que los métodos para validar y/o acceder a la variable `bookings` (contador de cantidad de reservas realizadas con dicho ticket) no eran *thread-safe*. Esto significaba que en todos lados donde se precisaba acceder a esta variable habría que sincronizar con el ticket. Nos pareció que esto era muy propenso a errores, entonces hicimos que `Ticket` garantice atomicidad en estos accesos por sí mismo, tal que uno no deba preocuparse por "adquirir este *lock* antes de llamar este método", cosa fácil de olvidar.

Este simple patrón consiste de indicarle al `Ticket` la función a ejecutar con un `Supplier<T>`, siendo `T` un tipo genérico cualquiera. La llamada es envuelta en un *lock*, con el ticket chequeando previamente que el visitante pueda realizar dicha reserva, y si todo sale bien incrementando el contador al final. Por último, el resultado de la llamada al `Supplier` es retornado por el ticket:

```
public synchronized <T> T bookTransactional(LocalTime slotTime,
                                            Supplier<T> transaction) {
    if (!this.ticketType.canBook(this.bookings, slotTime))
        throw new MissingPassException();
    T result = transaction.get();
    if (result != null)
        this.bookings++;
    return result;
}
```

Preferimos siempre utilizar `synchronized` por sobre objetos `lock`, dado que la sintaxis misma nos asegura que no es posible olvidarse de liberar el `lock`.

Posibles mejoras a futuro

Una mejora que se podría hacer a futuro es reemplazar el mecanismo de sincronización dentro de `ReservationHandler` para permitir que ocurran varias operaciones al mismo tiempo, siempre y cuando estas vayan a distintos *slots*. La razón por la que no se implementó esto es porque esta operación es muy rápida, consistiendo de un simple acceso a un mapa, además de que de todos modos se precisa poder *lockear* todos los *slots* para correr el algoritmo de realojación de reservas, entonces esto habría agregado complejidad innecesaria al proyecto.

Otra mejora posible es permitir que los clientes que leen archivos `csv` puedan cargar las atracciones y los tickets de forma concurrente, en diferentes *threads*. Al hacer esto, se podrían mandar varios requests al mismo tiempo y no sería necesario esperar la respuesta del servidor para enviar el *request* que se encuentra en la siguiente línea del `csv`. Una vez que se carga todo el archivo, sería necesario realizar el conteo final de todas las atracciones y los tickets cargados.