Integrantes: Octavio Serpe (60076), Manuel Rodríguez (60258), Gonzalo Arca (60303)

Trabajo Práctico Especial 1: Despegues
2do cuatrimestre 2021

1. *Decisiones de diseño e implementación de los servicios*

- Utilizamos clases *POJO*, como `DepartureData` y `ReassignmentLog`, definidas en `api`, que nos permiten comunicar información al cliente sin mostrar como funciona internamente el servidor.
- Lista de *callback handlers* en `Servant`, a la interfaz de los mismos se le agregó un método extra para realizar el `unexport` del *handler* tanto en el caso de despegue del vuelo como en un error que vuelva obsoleto al mismo.
- *Locks* con *fairness*, de esta manera los pedidos se atienden en el orden de mayor tiempo de espera, y no se atiende el pedido de otro thread por sobre su siguiente en la cola sobre un mismo recurso. Si bien se posee un costo de performance al despertar y dormir *threads*, permite brindar un orden cronológico a las operaciones sobre el *servant*. Un ejemplo podría ser que un *thread* T1 pide reordenar los vuelos, mientras que un *thread* T2 quiere agregar una pista, dado que trabajan sobre el mismo *lock* y en escritura (en este caso), intuitivamente uno buscaría que primero se reordenen, y luego se agregue la nueva pista (es decir, que no haya vuelos en la pista agregada), lo cual de no otorgar *fairness* produciría resultados inesperados ya que podría haber, como no, vuelos en la nueva pista.
- Estructuras de datos:
  - `HashMap<String, Runway>` para manejo de pistas de vuelo, cada Runway con una cola para vuelos actualmente en la cola, una lista para vuelos que ya salieron e información de la misma (si está abierta, categoría y nombre).
  - `HashMap<String, List<FlightTrackingCallbackHandler>>` para manejo de lista de suscriptores y llamar a sus respectivos callbacks.
- Uso de `ReadWriteLock` en lugar de un `Lock` normal, para el manejo de concurrencia sobre recursos compartidos, permitiendo ser *thread-safe*. Se utilizaron dos *locks* diferentes bajo la implementación de `ReentrantReadWriteLock`.
- Uso de `ExecutorService` tanto para el servant (`CachedThreadPool`) como los tests (`FixedThreadPool`).
- `Flight`, además de tener la información básica, contiene un dato `onDeparture` de tipo `LocalDate` que genera su valor una vez el vuelo despega. Se agregó porque nos permite ordenar en forma cronológica los valores a devolver en la *query* del servicio de consulta.

2. *Criterios aplicados para el trabajo concurrente*

Utilizamos *locks* por sobre colecciones concurrentes, pues esto otorga una mejor capacidad de decisión en cuanto a la localización de las zonas críticas de escritura y lectura. Ademas, si hubiésemos elegido la otra opción no tendríamos noción de cómo es el manejo pudiendo conllevar a una mayor ineficiencia.

Al utilizar `ReadWriteLock` se habilita al acceso tanto para lectura como escritura sobre diferentes recursos, es decir, podrían haber múltiples lectores, pero nunca más de un escritor ni lector/es y escritor/es simultáneamente[1], evitando delays

[1] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html

de exclusión mutua. Esto brinda la posibilidad de un mayor nivel de concurrencia y mejor performance.

Por otro lado, ante posibles demoras y deadlocks sobre los *threads*, cada uso de *lock* utiliza un sistema de *timeout* junto con una cantidad acotada de reintentos para probar adquirir el lock. En caso de no lograrlo, se arroja una excepción remota (`ServerError`) informando al cliente de la situación.

Con respecto a los *callbacks*, se hizo uso de la interfaz `ExecutorService` y la implementación de una `CachedThreadPool`, evitando un tope de *threads* y permitiendo la generación de los mismos a demanda y en base a su reutilización. De esta manera el *callback* no interrumpe el *thread* principal de ejecución, incrementando el tiempo de retención del *lock*. En caso que haya una falla, se libera el *handler*.

En el caso de los *tests* se utilizó una `FixedThreadPool` con el propósito de someter al *servant* a casos donde se tenga un mejor control de la máxima cantidad de *threads* generados. Podría cambiarse por una `CachedThreadPool`.

Para finalizar, se optó por realizar un `catch` para las excepciones arrojadas por los *callbacks* y reportarlas en la terminal del servidor, para así permitir la ejecución de los *callbacks* restantes.

3. *Potenciales puntos de mejora y/o expansión*

Creemos que tanto una separación en capas del *server,* como en varios *servants* (por reglas de negocio), y recursos compartidos permitiría un mejor manejo, abstracción y escalabilidad del mismo. Dado que se encuentra todo concentrado en un único *servant*, una pequeña modificación de diseño podría volverse bastante tediosa al haber métodos que utilizan las mismas estructuras, métodos y clases.

Por otro lado, la persistencia y replicación del *servant* sería algo interesante de mantener como sistema distribuido, evitando un único punto de falla.

4. *Aclaraciones*

Respecto a los *tests*, algunos son unitarios y otros de flujos completos/integración, poniendo a prueba diferentes reglas de negocio, donde las intermedias y finales precisan de las anteriores. Al tomar este enfoque se evalúan tanto excepciones y/o métodos de manera individual como flujos enteros combinando los diversos servicios, sometiendo al servant a un caso de uso “normal”.

Por otra parte, se utilizó el objeto *servant* que implementa las interfaces con las que el cliente se comunica al servidor puesto que consideramos tedioso crear diferentes objetos (4 en este caso) por cada interfaz cuando utilizan la misma implementación (el *servant*).

Finalmente, algunos métodos de las interfaces de la API especifican en su firma excepciones creadas por nosotros, donde las mismas extienden de `RuntimeException`, puesto que se buscó informar a los clientes de excepciones particulares que podrían arrojarse, de esta manera brindamos mayor información al cliente sobre qué esperar en caso de error.