Programación de Objetos Distribuidos
TPE 1
2°C 2021

Integrantes:

- Rolandelli, Alejandro (56644)
- La Mattina, Luca (57093)
- Bossi, Agustin (57068)
- Sampedro Delgado, Ignacio (58367)

Decisiones de diseño e implementación de los servicios .

Para el diseño de nuestro proyecto nos basamos en la estructura recibida de la cátedra la cual divide al proyecto en 3 módulos *API*, *Cliente* y *Servidor*.

En el módulo *API*, nos enfocamos en colocar las Excepciones que puede arrojar el servidor, en conjunto con todas las interfaces de los servicios, nos fijamos que en la *API* no hubiese nada que no necesitemos en la comunicación entre el *cliente* y el *servidor*.

En cuanto al *Servidor*, en este se encuentran el *Servant* y los objetos utilizados para manejar el aeropuerto, *Flight* y *Lane*:

- Un *Flight* está conformado por su id, categoría, aerolínea, aeropuerto destino y tiene un contador (*takeOffsOrdersQuantity*) de despegues hasta el despegue del mismo.
- El objeto *Lane* está conformado por un nombre, un estado, una categoría y una *Queue* de vuelos a despegar en esa pista. Esto se hizo así, porque estos despegan según el orden de llegada a la pista.
- El *Servant*, se maneja con 3 mapas.
    - El primero (*laneMap*) es donde guardamos las pistas según su categoría, siendo su clave el nivel de autorización de cada categoría (A=1, …, F=6) .
    - Una vez que un vuelo despega, este es almacenado en el segundo mapa(*flightHistory*), que sirve para mantener un historial de los vuelos que despegaron por cada pista.
    - En el tercero (registeredAirlines) se guardan las aerolíneas registradas por el método *registerAirline*, del *FlightTracingService*, con su lista de notificaciones respectivas por aerolínea, para el servicio de notificación de eventos.

Optamos por elegir este tipo de implementación, debido a la eficiencia de los Hashmaps, para poder buscar por clave, que nos ayuda dado que la comunicación entre servidor y cliente es por nombres o ids. Esta opción fue debatida contra el caso de que cada objeto tenga su propia lista, como por ejemplo *Lane* con su historial de vuelos, pero optamos la opción de los mapas debido a que nos podemos ahorrar la búsqueda de la *Lane* por nombre en el mapa principal.

Por último, en el *Cliente,* tenemos 4 clases principales que representan a los distintos clientes, donde cada una de estas se ocupa de una sección del aeropuerto (*management*, *airline*, *query*, *runway*). Los clientes comparten una estructura similar, que se basa en recibir/parsear los argumentos y proceder al llamado del servicio correspondiente.

Para la comunicación entre el cliente y el servidor, tratamos de que sea lo más transparente posible, y por esto que creamos un archivo de *logs*. En este archivo se muestra la información provista por el servidor, en conjunto con los errores de conexión, como *RemoteException*.

Criterios aplicados para el trabajo concurrente.

Se decidió no aplicar programación concurrente ni en el *Servant* ni en los *clientes* dado que no se consideró necesario. Sin embargo, el servidor realiza trabajo concurrente, ya que todos los clientes se conectan a un mismo servlet y trabajan sobre los mismos recursos.
Es por esto, que es importante la coordinación al momento de acceder a las funciones del mismo y, para ello, se utilizaron 2 estrategias vistas en clase.
Para proteger el acceso a la información del servidor, se utilizaron *ReadWriteLocks,* los cuales permiten acceder a un lock de escritura y uno de lectura, proveyendo en el primer caso escrituras secuenciales y, en el segundo, múltiples lecturas simultáneas.

Potenciales puntos de mejora y/o expansión.

La mejora que nos surge es la de la persistencia. Actualmente hay distintas estructuras de datos en memoria, la mayoría es o involucran un mapa para poder relacionar datos. Por ejemplo pistas con sus respectivos despegues. Esto podría ser reemplazado por una base de datos relacional, que permitiría este mismo manejo de datos, con más poder en caso de ser necesario fundamentalmente que no se pierdan los datos cada vez que se corre el programa.
Una alternativa que sería una mejora en el caso de que este sea un trabajo que va a expandir sus funcionalidades es el uso de colecciones concurrentes para facilitar el manejo de un código thread-safe. Esta opción fue discutida, pero no fue implementada debido a que la implementación de la concurrencia fue hecha luego de pensar la estructura del código y preferimos no tocar dicha estructura por miedo a generar problemas en el código.