Informe TPE1 - Grupo 7
Programación de Objetos Distribuidos - 72.42

Comisión S

Tobias Perry (62064), ... tperry@itba.edu.ar
Manuel Esteban Dithurbide (62057), ... mdithurbide@itba.edu.ar
Christian Ijjas (63555), ... cijjas@itba.edu.ar
Luca Seggiaro (63855), ... lseggiaro@itba.edu.ar

28 de Abril de 2024

Contents

1 Decisiones de diseño e implementación de los servicios **2**
- 1.1 Testing . . . 2
- 1.2 Events . . . 2
- 1.3 Casos extraoficiales . . . 2

2 Criterios aplicados para el trabajo concurrente **3**
- 2.1 Nombramiento declarativo y poco overhead sobre las funciones . 3
- 2.2 Usar estructuras concurrentes . . . 3
- 2.3 Buscar granularidad, pero nunca a costa de la eficiencia . . . 3
- 2.4 Disociar las funciones para entender sobre qué estructuras se está trabajando . . . 4
- 2.5 Asumir que todas las funciones internas a una son concurrentes, pero nunca que la padre es concurrente . . . 4
- 2.6 Utilizar una estructura escalable, pero no adelantarnos a casos futuros . . . 4
- 2.7 Ejemplo . . . 4

3 Potenciales puntos de mejora y/o expansión **6**
- 3.1 Concurrencia . . . 6
- 3.2 Parche eventos . . . 6

4 Bibliografia **7**

1 Decisiones de diseño e implementación de los servicios

Decidir la estructura de los servicios resultó ser desafiante. Con el objetivo de generar una estructura que evite repetición e iteraciones innecesarias, optamos por agrupar los contadores en grupos, estén ocupados o no. Adicionalmente, hicimos uso de `ConcurrentSkipListMap` (un reemplazo concurrente de `SortedMap`), ya que nos permiten recorrer la información de la manera solicitada, particularmente a la hora desplegar los sectores, y recorrer los grupos de contadores.

1.1 Testing

A medida que desarrollamos el proyecto, optamos por implementar un sistema de testing para las diferentes áreas del mismo. Esto ayudó con el orden general así también como en la detección de fallas ante un nuevo cambio o funcionalidad agregada.

Para los test más sencillos, que no requieren concurrencia se realizaron tests a nivel servidor, dado que es más sencilla la invocación. Se buscó abarcar los casos solicitados por la cátedra con rigurosidad, realizando por lo menos un test para cada caso. Los testeos de concurrencia, realizados a nivel cliente, no abarcan el 100% de la aplicación, dada su complejidad. Estos últimos se hicieron creando archivos de scripting de bash que hacían invocaciones en background y/o secuenciales. Este formato de este testeo es rústico y como ya se dijo, no engloba todos los casos.

1.2 Events

Para la implementación del servicio de eventos (Punto 5), se consideró inicialmente realizar polling de nuevas notificaciones en un while. Esta opción se descartó rápidamente. Finalmente se decidió implementar el patrón pub-sub [5]. El servant guarda en un mapa el Airline name y el ResponseObserver. Al realizar una notificación, el servant envía por el stream correspondiente la notificación. Finalmente al des-suscribirse, se elimina el mapeo y se cierra el stream anterior.

1.3 Casos extraoficiales

Buscando mantenernos dentro de los parámetros del enunciado, encontramos algunos casos de error que no fueron aclarados explícitamente, pero que derivan de los posibles estados del servidor. El caso pertinente es cuando una aerolínea libera un grupo de contadores, y un usuario con uno de los vuelos busca los counters check-in. Consideramos que "`Flight ... from ... has no counters assigned yet`" no era lo suficientemente descriptivo, por lo que incluimos el caso "`Flight ... from ... has already expired`", ya que nunca podrá volver a asignarse un grupo de contadores para ese vuelo.

2 Criterios aplicados para el trabajo concurrente

Podemos afirmar que aplicar concurrencia al trabajo resultó ser el aspecto más desafiante de éste. Decidimos hacer que todos los incisos funcionen antes de implementar semáforos, ya que, con el objetivo de disociar las responsabilidades, la concurrencia pueda aplicarse sobre el trabajo existente con la menor cantidad de "mezcla" posible. Esta decisión resultó ser un arma de doble filo, ya que, al comenzar a implementar este requisito, nos encontramos con un mar de dudas, sin saber por dónde empezar.

A partir de ello, establecimos un par de lineamientos para ser consistentes y claros:

2.1 Nombramiento declarativo y poco overhead sobre las funciones

Para mantener la legibilidad del código, deseamos que los semáforos no ocupen más de una línea a la vez. el `SemaphoreAdministor` nos permite extraer la lógica de los semáforos de nuestros repositorios / servicios.

2.2 Usar estructuras concurrentes

Si bien no resuelve todos nuestros problemas, usar estructuras concurrentes (de la librería `java.util.concurrent`) [1] facilita el proceso de pensar en posibles condiciones de carrera, ya que uno puede considerar atómicas la mayoría de las funciones de agregación y búsqueda. Por ello decidimos usarlas cuando fuese posible.

2.3 Buscar granularidad, pero nunca a costa de la eficiencia

Para lograr concurrencia, una manera sencilla de reducir la posibilidad de una condición de carrera es poner synchronized en todas las funciones. No es una solución muy elegante, ya que, si dos sectores desean agregarse a una lista (que implementa concurrencia) no hay razón para evitar que ocurran en simultáneo. Teniendo esto en cuenta, el otro extremo tampoco es deseable, si una función recibe una lista de `flightCodes`, anidar n locks es poco eficiente, ya que se deben buscar dichos locks en el `SemaphoreAdministrator`. Ni hablar del riesgo a la hora de considerar un deadlock. Por ello haciendo uso de la librería `java.util.concurrent.locks` [2], apuntamos a un punto medio, donde buscamos usar la menor cantidad de locks a la vez, sin limitar las situaciones que pueden ocurrir en simultáneo.

2.4 Disociar las funciones para entender sobre qué estructuras se está trabajando

Acompañando el punto anterior, decidimos separar las diferentes estructuras almacenadas por el aeropuerto en distintas clases: pasajeros - aerolíneas y vuelos - sectores y sus counters. Esto nos permitió enfocarnos en cada una por separado, y poder asumir que, dada una función, sólo las funciones dentro de una su misma clase puede generar condiciones de carrera. Dicho lineamiento nos ayudó a plantear el próximo punto.

2.5 Asumir que todas las funciones internas a una son concurrentes, pero nunca que la padre es concurrente

A la hora de implementar concurrencia, nos encontramos muy pendientes de las funciones que se llaman dentro de la función, y que locks fueron aplicados a la hora de llamar a la actual. Las combinaciones y posibilidades resultaban agotadoras, por lo que decidimos asumir que todas las funciones internas a una son concurrentes, pero nunca que la padre es concurrente. Esto nos permitió enfocarnos en la lógica de cada función individualmente y ser más consistentes. Eventualmente, al analizar todas las funciones, sabríamos que una función que llama a muchas es concurrente, sin tener que preocuparnos de considerar todas las ramificaciones.

2.6 Utilizar una estructura escalable, pero no adelantarnos a casos futuros

Naturalmente, a la hora de definir la estructura del proyecto, nos surgieron muchas preguntas relacionadas a la implementación, pero que superan lo solicitado. Por ejemplo: ¿Qué pasaría si se pudiesen borrar sectores?

Si uno tuviese tiempo y recursos extensos, se podrían tomar en cuenta estas preguntas para hacer que sea más sencilla la implementación a futuro. Sin embargo, acorde a la filosofía YAGNI, esto complejizaría la versión actual, y nada nos garantiza que se pedirá en el futuro.

2.7 Ejemplo

A continuación se presenta un ejemplo donde se ponen en práctica los lineamientos mencionados:

```
1 public Pair<Boolean, Integer> assignCounters(String
    sectorName, String airlineName, List<String>
    flightCodes, int counterCount) {
2         if (!sectors.containsKey(sectorName))
3             throw new IllegalArgumentException("Sector
                not found");
4
```

```
5           semaphoreAdmin.readLockFlightCodes();
6
7           if (!airlineRepository.allFlightCodesRegistered(
                flightCodes)) {
8               semaphoreAdmin.readUnlockFlightCodes();
9               throw new IllegalArgumentException("At least
                    one of the flight codes have not been
                    registered");
10          }
11
12          if (airlineRepository.
                flightCodeAlreadyExistsForOtherAirlines(
                airlineName, flightCodes)) {
13              semaphoreAdmin.readUnlockFlightCodes();
14              throw new IllegalArgumentException("A
                    requested flight code is assigned to
                    another airline");
15          }
16
17          if (!airlineRepository.allFlightCodesAreNew(
                airlineName, flightCodes)) {
18              semaphoreAdmin.readUnlockFlightCodes();
19              throw new IllegalStateException("At least one
                     flight code has been assigned, is pending
                     or has ended");
20          }
21
22          CheckinAssignment checkinAssignment = new
                CheckinAssignment(airlineName, flightCodes,
                counterCount);
23
24          semaphoreAdmin.unlockReadLockWriteFlightCodes();
25
26          Pair<Boolean, Integer> toRet = sectors.get(
                sectorName).assignCounterGroup(
                checkinAssignment);
27          airlineRepository.markFlightsAsAssigned(
                airlineName, flightCodes);
28
29          semaphoreAdmin.writeUnlockFlightCodes();
30          return toRet;
31      }
```

`assignCounters()` es una de nuestras funciones más extensas, y se puede descomponer en dos partes: las verificaciones necesarias y la ejecución de la función. Sin duda, esta función necesita implementar semáforos ya que, una vez

pasadas las verificaciones, se puede ejecutar un segundo thread y que pase estas mismas, con códigos de vuelo repetidos, resultando en dos grupos de contadores con códigos repetidos. Por ello se bloquea sobre los `flightCodes`, ya que es el parámetro que puede provocar comportamiento no deseado. A diferencia de los sectores y aerolíneas, los `flightCodes` comparten un solo semáforo, dado que resultaba mucho más difícil soportarlos, considerando que en muchos casos se trabaja con una lista de estos (2.3). La verificación del sector está por fuera del lock, ya que no corremos riesgos de que se borre el sector (2.6).

3 Potenciales puntos de mejora y/o expansión

3.1 Concurrencia

Observando el código en más detalle, se podrá notar que existen muchas más instancias de write locks que de read locks. Como las estructuras ya implementan concurrencia, nos resultó difícil encontrar casos en donde fuese necesario el lock para la lectura. Desearíamos dedicar más tiempo a entender las posibles condiciones de carrera y cómo evitarlas.

Por otro lado, nos gustaría repensar la estructura de semáforos implementada, los repositorios acceden directamente al `semaphoreAdministrator`, cuando claramente se podría dividir en administradores por repositorio. Un ejemplo sería tener un `airlineSemaphoreAdministrator`, y que el repositorio de airline acceda a este.

Adicionalmente, según nuestro análisis, `passengerRepository` no requiere implementar dicha estructura, ya que creemos que los mapas concurrentes resuelven los posibles problemas.

3.2 Parche eventos

Como mencionado previamente, a la hora de implementar eventos, decidimos hacer una estructura minimal, con un único método `.notify()`. En una primera instancia, todos los eventos ocurrían a partir de llamados al servidor, por lo que el `eventServant` sería una estructura transversal a los demás servants. Sin embargo, hubo un caso que no cumple con esta suposición, las notificaciones que ocurren cuando aumenta la cantidad de espacio, y vuelos pendientes se asignan. Para estos casos, el `eventServant` se pasa como parámetro, mezclando su lógica con el servicio de contadores. En caso de continuar expandiendo este aspecto, agruparíamos las notificaciones en una clase que se encarga de enviar los mensajes, agrupando la lógica en un solo lugar.

4 Bibliografia

References

[1] Librería concurrent de Java, https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html

[2] Librería de locks de Java, https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/package-summary.html

[3] Parser del lado de clientes https://www.youtube.com/watch?v=w0Bckb9Znfg&ab_channel=DanielPersson

[4] Documentación protobuf https://protobuf.dev/

[5] Pub-sub en grpc https://stackoverflow.com/questions/54942240/how-to-design-publish-subscribe-pattern-properly-in-grpc