Programación Concurrente Usando Actores en Scala

Laboratorio 3 - Paradigmas de la Programación 2021

¡Consigna colaborativa! Si encuentran errores o tienen dudas de interpretación, dejen un comentario, así vamos mejorando este lab entre todos.

Disclaimer: Este laboratorio requiere que **lean** (como se explicará más adelante con mayor detalle). Empiecen por leer **toda la consigna** antes de empezar a hacer nada.

Fecha de entrega: 10/06/2021 - 23:59:59

Introducción

En este laboratorio veremos una introducción a la programación concurrente mediante el uso del modelo de actores, la implementación que usaremos para esto es la librería Akka en su versión para Scala.

Los problemas con el estado mutable compartido

El enfoque predominante de la concurrencia hoy en día se basa en mantener un *estado mutable compartido*. Esto quiere decir que los sistemas se componen de un gran número de objetos con estado, el cual puede ser cambiado por múltiples partes de la aplicación, cada una corriendo en su propio hilo.

Típicamente, en el código se intercalan *bloqueos* de lectura y escritura, para asegurarse de que el estado sólo pueda cambiarse de forma controlada y evitar que múltiples hilos lo alteren simultáneamente. Al mismo tiempo, nos esforzamos por no bloquear un fragmento de código demasiado grande, ya que esto puede ralentizar drásticamente la aplicación.

La mayoría de las veces, código como este ha sido escrito originalmente sin tener en cuenta la concurrencia en absoluto —sólo para ser adecuado para un mundo de hilos múltiples una vez que surgió la necesidad. Mientras que escribir software sin la necesidad de concurrencia como este lleva a un código muy sencillo, adaptarlo a las necesidades de

un mundo concurrente lleva a un código que muchas veces se vuelve muy difícil de leer y entender.

El principal problema de implementar soluciones usando sincronización a bajo nivel es que es muy difícil abstraerse sobre el código que se está escribiendo. En consecuencia es difícil implementar el sistema correctamente, y luego ocurren errores como condiciones de carrera o *deadlocks*, o sencillamente comportamientos extraños que son difíciles de replicar.

El modelo de concurrencia por actores

El modelo de concurrencia por actores tiene como objetivo evitar todos los problemas descritos anteriormente, permitiendo escribir código concurrente de alto rendimiento sobre el que es más sencillo abstraerse. A diferencia del ampliamente utilizado enfoque de estado mutable compartido, en este modelo la concurrencia es pensada desde el diseño mismo inicial del código.

La idea es que la aplicación consiste en un montón de entidades ligeras llamadas actores. Cada uno de estos actores es responsable de una tarea muy pequeña, y por lo tanto sobre la que es fácil razonar. Sobre estas tareas pequeñas se pueden construir sistemas más complejos mediante la interacción de actores, delegando tareas en una jerarquía definida previamente a la escritura de código.

La aplicación: Un manejador de subscripciones concurrente

Si bien el modelo de actores está pensado para grandes aplicaciones que manejen concurrencia y trabajen en computación distribuida, durante este laboratorio nos centraremos ejemplificar de manera general los conceptos de programación concurrente con Actores a través del armado de un protocolo sencillo de pasaje de mensajes entre actores.

Para ello vamos a tomar inspiración del Laboratorio 2, esta vez dejando de lado la parte de NERs, y vamos a implementar una aplicación muy sencilla que será un manejador de suscripciones de RSS (y opcionalmente Reddit-JSON) que se resolverá de manera concurrente.

Definiciones

- Un **feed** es la respuesta a una consulta HTTP realizada a una URL particular. Un feed RSS es un tipo de feed que tiene un formato de archivo XML con una estructura particular.
- Un **sitio** es un servicio web con el mismo dominio, al que se le pueden pedir múltiples feeds. Por ejemplo, https://rss.nytimes.com/services/xml/rss/nyt/ es un sitio, que tiene asociadas las urls:
    - https://rss.nytimes.com/services/xml/rss/nyt/Business.xml
    - https://rss.nytimes.com/services/xml/rss/nyt/Technology.xml

El esqueleto

El esqueleto inicial del laboratorio es muy sencillo, al correrlo levanta un `ActorSystem`, espera 10 segundos y manda un mensaje para parar el sistema. Su trabajo será extenderlo para poder hacer suscripciones, que leerán mediante la implementación del método `readSubscriptions`. Este método lee un archivo que se pasa por línea de comandos y se guarda en el objeto `config` de la classe `SubscriptionApp`.

Para correr el código se utiliza el siguiente comando:

```
$ sbt "run --input ./subscriptions.json"
```

El laboratorio

Este laboratorio es similar en muchos aspectos al ejemplo práctico de la guía de iniciación de Akka. Así que es tarea suya leerla (y, en medida de lo posible, intentar hacerla), para tener una idea general de lo que se van a encontrar.

Primera parte: Diseñando la arquitectura

La primera parte de la guía habla de la arquitectura del sistema de actores; este concepto es algo que vuelve recurrentemente durante la guía pues es la base del sistema. Recuerden que en la concurrencia por actores, el sistema se piensa concurrente desde el inicio, no se puede cambiar después (o mejor dicho, eso requeriría una refactorización total). Tomando inspiración de la guía, la primera parte será diseñar la arquitectura de sus sistema teniendo en cuenta algunos de los siguientes requerimientos:

- El sistema está gobernado por un *supervisor*, que se encarga del manejo global del sistema. Este supervisor es el que se conecta con el "mundo exterior" (visto desde el

punto de vista del sistema de actores, e.g., en este caso, la aplicación `SuscriptionApp`).

- Las subscripciones son por feed/subreddit. En el sistema de base, estas son sólo a Feeds RSS, pero el sistema debe poder extenderse a otro tipo de subscripciones sin tener que refactorizar por completo (e.g. Reddit-JSON).
- Las diferentes suscripciones (feeds) de un mismo sitio deberán ser manejadas de manera conjunta. En particular, debe ser trivial poder recolectar toda la información de suscripciones de un mismo sitio.

Para completar esta parte, deberán **diseñar esta arquitectura y representarla con un diagrama** (puede ser vía DIA, o aplicaciones similares, o simplemente vía ASCII). Esta arquitectura incluye todos los actores del sistema, y los mensajes que envían o reciben de otros actores. Deben incluir el diagrama en el informe, junto con una descripción de las responsabilidades de cada actor y la función que cumple cada mensaje; y justificar sus decisiones. Para poder realizar este punto, deben **leer toda la consigna** para entender las funciones que debe realizar el sistema.

Tengan en cuenta las buenas prácticas que establece Akka a la hora de diseñar su arquitectura, y las buenas prácticas de programación orientada a objetos que vieron en el laboratorio anterior.

Segunda parte: Actores para subscribirse

En la segunda parte deberán implementar los actores que manejan la suscripción a un sitio, con sus múltiples feeds, siguiendo la arquitectura de actores que propusieron en el punto anterior. Esto requiere un diseño de clases que tenga en cuenta lo establecido en la primera parte respecto a la posible extensión en el tipo de subscripción.

El protocolo deberá tener mensajes que creen las nuevas suscripciones de un sitio en particular. Por otra parte, deberán tener mensajes que pide la lista de mensajes (título y texto principal del item) de un feed en particular o de todos en general (hint: reutilizar código en donde sea posible).

En esta parte, luego de obtener los feeds (vía algún tipo de request HTTP), estos deben ser impresos por pantalla mediante el sistema de logging mismo de akka.

En el archivo `subscriptions.json` hay una lista de sitios y feeds de dichos sitios que pueden utilizar para probar. Ese archivo (o uno similar) es leído por línea de comandos haciendo uso del objeto `config` en la aplicación principal e implementando el método `readSubscriptions` (hint: utilizar lo visto en el laboratorio 2 como base para la

implementación). A partir del resultado, deben generar todas las suscripciones correspondientes en el sistema mediante el pasaje de mensajes.

Tercera parte: Request-Response

En la tercera parte deberán *modificar* el pasaje de mensajes de la segunda parte para que soporte un modelo de Request-Response, donde el actor que está arriba en la jerarquía pide a la suscripción la lista de mensajes del feed y este último lo devuelve en lugar de simplemente imprimirlo por pantalla (i.e., el actor que hace el pedido ahora tiene acceso directo a los ítems del feed).

Existen varios patrones de interacción para realizar esta tarea. Desde la cátedra les invitamos a explorarlos y decidirse por el que les resulte más sencillo. En particular, el *ask-pattern* es el que nosotros hemos probado y cumple con la tarea en cuestión.

Justificar en el informe la decisión tomada y actualizar, de ser necesario, el protocolo de mensajes y su utilidad. ¿Por qué es necesario este comportamiento? (ejemplificar).

Cuarta parte: Integración

La última parte constará de la integración de todos los módulos y componentes de la aplicación. Para ello, su aplicación deberá realizar las siguientes tareas:

1. Carga de suscripciones de cualquier archivo pasado por línea de comandos. Estos archivos siguen el formato de `subscriptions.json` donde hay varios sitios de noticias, cada uno con sus respectivas suscripciones.
2. Pedido de las suscripciones mediante requests de HTTP (pueden utilizar lo mismo que el laboratorio pasado para hacer pedidos HTTP o utilizar cualquier otra librería que sea útil para ello).
3. Devolución de los ítems de cada suscripción al actor inmediatamente superior en la jerarquía (que hizo el pedido original).
4. Logging de dichos ítems por consola mediante el sistema de logging de akka.

Quinta parte: Investigación

La quinta parte no es de implementación sino de investigación. Se dejarán una lista de preguntas que deberán responder y justificar en el informe:

- Si quisieran extender el sistema para soportar el conteo de entidades nombradas del laboratorio 2, ¿qué parte de la arquitectura deberían modificar? Justificar.

- Si quisieran exportar los datos (ítems) de las suscripciones a archivos de texto (en lugar de imprimirlas por pantalla):
    - ¿Qué tipo de patrón de interacción creen que les serviría y por qué? (hint: es mejor acumular todo los items antes de guardar nada).
    - ¿Dónde deberían utilizar dicho patrón si quisieran acumular todos los datos totales? ¿Y si lo quisieran hacer por sitio?
- ¿Qué problema trae implementar este sistema de manera síncrona?
- ¿Qué les asegura el sistema de pasaje de mensajes y cómo se diferencia con un semáforo/mutex?

Puntos estrella[1]

Suscripción a Reddit/JSON*

Una extensión sencilla es la de permitir a la aplicación soportar suscripciones a Reddit/JSON como se vio en el laboratorio 2. Para este punto estrella deberán agregar el soporte a suscripciones a subreddits mediante la API correspondiente (la misma utilizada en el laboratorio 2). Deberán modificar todos los protocolos que consideren necesarios y convenientes y detallar dichos cambios en el informe.

Conteo de Entidades Nombradas*

Utilizando el modelo sencillo de captura de entidades nombradas del laboratorio anterior, modificar la arquitectura y los protocolos para que se haga el conteo de entidades nombradas y posterior logging de dicho conteo. Se debe minimizar la cantidad de veces que se llama al modelo, para simular la situación en la que cargar y ejecutar un modelo consume muchos recursos.

Guardar datos a disco**

Modificar el sistema para tomar por línea de comandos el path a un directorio de salida. En dicho directorio se guardarán archivos, uno por cada sitio suscripto, con los artículos de los feeds a los que se está suscrito dentro de ese sitio.

Espera de proceso de datos**

Una cosa que este laboratorio no implementa es la espera correcta del proceso de datos. Simplemente se hace un thread sleep de determinada cantidad de segundos (que se

[1] Los asteriscos indican la dificulta del punto estrella (puntos estrella con más asteriscos dan mayor puntuación en la nota final).

puede pasar por parámetro) y eventualmente se manda una señal de alto al sistema, que lo apaga. Una mejor manera sería esperar por un mensaje, una vez procesados todos los datos, para efectivamente dar de baja el sistema.

Conexión con el mundo exterior vía REST API***

Utilizando Akka-HTTP rediseñar el sistema para que, en lugar de realizar todos los pasos de acuerdo a la implementación dada, pueda servir todo lo necesario mediante una REST-API. Este punto es particularmente complejo porque requiere que diseñen la API como parte de la solución (i.e. los endpoints de la misma), y vean cómo conectar dicha API a sus sistema de actores.

Algunas recomendaciones a la hora de probar el código

Es importante poder probar las cosas a medida que se van creando, para ello se recomienda hacer uso del Scala REPL haciendo `sbt console` desde el directorio raíz de su repositorio. De esa manera podrán probar las cosas que van implementando y tendrán acceso a todas las librerías que hayan incluido en el archivo `build.sbt`. Eventualmente, si así lo desean, pueden hacer uso de notebooks de jupyter.

Comandos de SBT

- `compile`: Compila el código, necesario si quieren correr el código con los últimos cambios reflejados.
- `run`: Corre el código (en principio, también lo vuelve a compilar si detecta cambios).
- `clean`: Limpia los archivos compilados (útil para asegurarse de no estar trabajando con alguna versión vieja de algo).
- `cleanFiles`: Similar al anterior pero limpia todos los archivos generados en el proceso de compilación (muy útil para asegurarse de que no haya nada viejo dando vueltas en el código).
- `reload`: Deberán ejecutarlo cada vez que cambiar el archivo `build.sbt` (e.g. agregando una librería).

La entrega

Los entregables se detallan a continuación. Es importante detallar que **sólo con la implementación no basta para aprobar el laboratorio** aún si funciona al 100% y tienen los puntos estrella hechos. Por la naturaleza de este laboratorio, que requiere de tiempo de lectura e investigación, habrá otros puntos a tener en cuenta a la hora de entregar.

Uso del repositorio

Es importante hacer uso correcto del repositorio de BitBucket, no llenarlo de commits insignificantes (y con mensajes vagos) y hacer commits entre todos los integrantes del grupo. Recuerden que también es importante hacer commits frecuentes y no subir todo en 1 o 2 commits.

La entrega será por medio del repositorio, con fecha límite el Jueves **10 de Junio de 2021 a las 23:59**, deberán hacerlo por medio de un tag:

```
$ git tag -a lab-3 -m "Entrega Laboratorio 3"
```

Para el caso de los puntos estrella, se deberán hacer en un branch aparte y entregarse mediante tags también.

Atención: el informe deberá detallar todo lo que aparezca en los puntos estrella, por más que estos hayan sido trabajados en distintas branchs y estén subidos a distintos tags.

Informe

El informe es esencial en la entrega (cubre 1/3 de la nota final), deberá detallar claramente todos los desafíos encontrados y como los solucionaron y además deberán responder **todo lo que se les pide en la consigna** (e.g. diagrama de arquitectura, protocolo de mensajes, preguntas, etc.).

Requerimientos de diseño

- Este es un laboratorio de programación concurrente, por lo tanto deberán implementar todo de manera asíncrona mediante el uso de actores y, de ser necesario, futuros y/o promesas.
- Se espera que hagan correcto manejo de errores/fallas/excepciones (e.g. casos donde una URL no sea válida o similar):
    - Resuelvan las excepciones mediante mónadas (i.e `scala.util.Try` en lugar de `try { ... } catch { ... }`).
- Si bien Scala es un lenguaje funcional, la realidad es que el pasaje de mensajes mediante actores muchas veces no tiene respuesta (o al menos no respuesta directa), luego hay casos donde el uso del `foreach` está justificado (y es necesario). Aún así, estén atentos al uso de métodos como `map`, `filter` o `fold` cuando sea posible.
- Si la situación lo amerita, hagan uso de `flatMap` (limiten el uso de `flatten` cuando algo se puede resolver mediante `flatMap`).

- Hagan uso de *pattern matching*: El uso de `case class` y `case object` es obligatorio en los casos que lo amerite y utilizar construcciones que pueden ser reemplazadas por estos casos será penalizado.
- Si algo puede ser inmutable siempre prefieran dicha opción, limiten el uso de `var` y prefieran el uso de colecciones mutables sólo cuando sea necesario. En especial en el pasaje de mensajes entre actores.
- El uso de recursión a la cola, de ser necesario, no es obligatorio pero está recomendado.
- Hagan uso de clases y herencia cada vez que puedan.
- Agreguen a `build.sbt` cualquier librería extra que decidan utilizar y asegúrense de que todo sea compatible:
    - Vean de tener correctamente configuradas las versiones de Scala y SBT en los archivos `build.sbt` y `project/build.properties`.
    - Si utilizaran algún plugin que el archivo `project/plugins.sbt` esté agregado también.

Estilo de código

- Scala es un lenguaje que, a diferencia de Python, da lugar a hacer código ilegible, es importante que esto no suceda.
- El estilo de código es válido si el código es legible y está prolijo. Traten de no pasar de las 80 columnas, y jamás sobrepasen las 100.
- Hagan buen uso de espacios e indentaciones. Nunca utilicen tabs, siempre prefieran espacios. Scala suele indentarse con un espacio de 2 como base.
- Todos los archivos deben tener estilo consistente.
- El objetivo de clases, atributos y el output de métodos deben estar documentados en inglés. No exageren tampoco, **good code is the best documentation**.
- Por sobre todas las cosas, siempre recuerden KISS.