Universidad de los Andes

Facultad de Ingeniería
Maestría en Ingeniería de Sistemas
Concurrencia, Paralelismo y Distribución
Periodo 2024-10

Elixir
Reporte de trabajo

Juan Diego Yepes Parra - 202022391
Esteban Gonzalez Ruales - 202021225
Felipe Núñez - 202021673

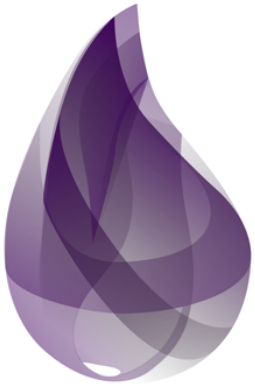
11 de abril de 2024
Bogotá D.C.

Índice

1. Introducción

Como parte del trabajo de este semestre, hemos realizado varias tareas que efectivamente hacen el mismo programa, un *Chatroom*. El comportamiento de la sala de chat es bastante sencillo. En primer lugar, los usuarios pueden unirse en cualquier momento y obtener acceso inmediato a los mensajes publicados a partir de ese momento. En segundo lugar, cualquier usuario puede enviar mensajes en la sala de chat. Por último, en cualquier momento, los usuarios pueden salir de la sala de chat y dejar de recibir mensajes. Sin embargo, hemos desarrollado individualmente esta sala de chat utilizando 4 paradigmas de programación concurrente diferentes, los cuales son Hilos (más conocidos en inglés como *Threads*), Memoria transaccional (por sus siglas STM o *Software Transactional Memory*), Comunicación de procesos secuenciales (por sus siglas en inglés CSP) y el modelo de Actores.

El propósito de este documento es reflexionar sobre los diferentes modelos de programación concurrente que utilizamos, con el fin de entender un poco mejor aplicaciones que pueden tener, casos de uso y demás.



elixir

2. Hilos / Mutex

La primera versión del Chatroom fue desarrollada utilizando Hilos. Esto puede parecer contra intuitivo a primera vista, ya que como es bien sabido, en Elixir no existe el concepto de hilos como en Java u otros lenguajes de programación. El modelo de concurrencia de Elixir está basado en actores, los cuales son completamente aislados entre sí, se ejecutan concurrentemente y se comunican mediante el paso de mensajes. Estos procesos son extremadamente ligeros en términos de memoria y CPU, incluso en comparación con los threads en otros lenguajes de programación. Por ejemplo, es común tener decenas o incluso cientos de miles de procesos ejecutándose simultáneamente en Elixir.

De igual forma, en Elixir los datos no son mutables. Esto quiere decir que cuando realizamos operaciones sobre los mismos, nunca estamos modificando el dato como tal, sino que hacemos una copia del mismo con la operación. Una ventaja de la inmutabilidad es que hace que el código sea más fácil de entender. Puedes enviar datos sin preocuparte de que alguien los cambie en la memoria, solo los transformará. En conclusión, en Elixir tampoco existe el concepto de "Memoria Compartida"por lo cual los semáforos o Mutex no son de mucha utilidad.

Teniendo todas estas consideraciones en cuenta, de igual forma podemos "simular"que estamos utilizando semáforos con el módulo Mutex que fue proporcionado por el enunciado. Esta tarea representó el reto más grande para todos los integrantes del equipo, debido a que, estábamos utilizando el lenguaje de una forma que no está optimizado.

3. Memoria Transaccional (STM)

La segunda versión del chatroom fue desarrollada utilizando el paradigma de Memoria Transaccional (STM). Este enfoque se basa en la idea de transacciones, donde un conjunto de operaciones se ejecutan como una unidad atómica e indivisible. Si alguna de las operaciones falla, se revierten todas las operaciones realizadas dentro de la transacción, manteniendo así la consistencia de los datos.

La librería que usamos para esta implementación es Mnesia, que viene del lenguaje padre de Elixir, Erlang. Sin embargo, para implementar un sistema de STM en nuestro chatroom, debimos diseñar y desarrollar nuestras propias funciones para garantizar la integridad de los datos.

Aunque el enfoque STM proporcionó un mecanismo robusto para garantizar la consistencia de los datos, también introdujo una complejidad adicional en el código, ya que aunque Mnesia es transaccional no es relacional, luego hay que tener cuidado con el manejo de llaves y la repetición de los datos.

El uso de Memoria Transaccional en nuestro chatroom nos permitió mantener la integridad de los datos compartidos entre múltiples procesos concurrentes, aunque con un costo adicional en complejidad y rendimiento.

4. Comunicación de procesos secuenciales (CSP)

En la tercera iteración del chatroom, utilizamos el modelo de Comunicación de Procesos Secuenciales (CSP) utilizando la biblioteca CSPEx (CSP). Esta biblioteca proporciona una implementación de CSP en Elixir, permitiendo la introducción de canales para la comunicación entre procesos.

Al adoptar el modelo CSP con CSPEx, nos fue posible hacer una separación de funcionalidades en canales dedicados. Lo que hicimos fue un canal dedicado para la entrada de mensajes, al cual todos los usuarios escribían, y luego otro proceso hacía una tranmisión estilo broadcast a todos los canales dedicados de cada usuario para que todos tuvieran todos los mensajes, en el orden que deberían estar.

Esta división de responsabilidades en procesos independientes facilitó significativamente la implementación, ya que fomentó una arquitectura modular y desacoplada.

5. Actores

La última versión del chatroom se desarrolló en grupo (ver repositorio aquí: `https://github.com/juanyepesp/isis4218-actors-task`) utilizando el modelo de Actores. En este paradigma, cada componente del sistema se representa como un actor independiente que encapsula su estado y comportamiento, y se comunica con otros actores mediante el intercambio de mensajes.

Al adoptar el modelo de Actores, ni siquiera tuvimos que importar ninguna librería. Cada componente del chatroom, como la gestión de usuarios, la entrada de mensajes o la distribución de mensajes, se implementó como un actor (Task) independiente.

Aunque el modelo de Actores proporcionó la implementación más sencilla y optimizada para el lenguaje, también introdujo cierta complejidad en la gestión de la concurrencia y la comunicación entre actores. La coordinación de múltiples actores para lograr un comportamiento coherente fue un desafío significativos que enfrentamos durante el desarrollo.

6. Apreciaciones generales

En este documento, hemos reflexionado sobre nuestras experiencias al desarrollar un chatroom en Elixir utilizando diferentes paradigmas de programación concurrente: Hilos/Mutex, Memoria Transaccional (STM), Comunicación de Procesos Secuenciales (CSP) y el modelo de Actores.

A lo largo del desarrollo de las diferentes versiones del chatroom, pudimos explorar y experimentar con diferentes paradigmas de programación concurrente en Elixir. Cada enfoque tenía sus propias ventajas

y desafíos, y nos brindó una perspectiva única sobre cómo abordar problemas de concurrencia en sistemas distribuidos.

El uso de Hilos/Mutex nos permitió familiarizarnos con conceptos tradicionales de concurrencia, como la sincronización y la exclusión mutua, aunque con limitaciones inherentes en el contexto de Elixir y su modelo de concurrencia basado en actores.

La adopción de Memoria Transaccional nos brindó una forma elegante de garantizar la consistencia de los datos en un entorno altamente concurrente, aunque con un costo adicional en complejidad y rendimiento.

El modelo de Comunicación de Procesos Secuenciales (CSP) nos proporcionó una forma mucho más eficiente de manejar la concurrencia en el lenguaje, facilitando así la implementación, sin embargo presentó un desafio en la coordinación de los procesos.

Finalmente, el modelo de Actores nos permitió aprovechar la escalabilidad y la concurrencia de Elixir para crear un sistema altamente concurrente y robusto, aunque con desafíos en la coordinación y gestión de múltiples actores.

En general, el proceso de explorar y experimentar con diferentes paradigmas de programación concurrente nos brindó una comprensión más profunda de las fortalezas y limitaciones de cada enfoque.

En última instancia, hemos llegado a la conclusión de que no existe un enfoque único o "mejor"para abordar la programación concurrente en Elixir, excluyendo el Mutex. En su lugar, depende del contexto específico del problema y de los requisitos del sistema para determinar qué enfoque es más adecuado. Sin embargo, cada paradigma proporciona herramientas y técnicas útiles que pueden ser aprovechadas de manera efectiva para crear sistemas concurrentes robustos y escalables en Elixir.

7. Referencias

- The Erlang/OTP Team. (s.f.). Processes. Recuperado de `https://hexdocs.pm/elixir/processes.html`
- Elixir School. (s.f.). Almacenamiento con Mnesia. Recuperado de `https://elixirschool.com/es/lessons/storage/mnesia`
- Hexdocs.pm. (s.f.). CSPex - CSP: Constraint Satisfaction Problem library for Elixir. Recuperado de `https://hexdocs.pm/cspex/CSP.html`
- Hexdocs.pm. (s.f.). Listas y Tuplas. Recuperado de `https://hexdocs.pm/elixir/lists-and-tuples.html`

- Elixir School. (s.f.). ¿Por qué Elixir? Recuperado de https://elixirschool.com/es/why