Eduardo Blanco Bielsa – UO285176
Aarón Orozco Fernández – UO281997

Aarón:

Michael Perry un matemático del software. Para él el software es matemática, cada clase es un teorema, el compilador son las pruebas … Este es su lema. Michael realiza cursos de Pluralsight sobre CQRS, patrones XAML, criptografía… Pero nos vamos a centrar en el concepto de la arquitectura inmutable que es uno de sus principales estudios y del cual incluso a escrito un libro.

Él comenzó a pensar en esta idea en 2001 mientras trabajaba en un Sistema distribuido. Era un Sistema de tarjetas que tenía la capacidad de canjear el valor sin realizar ninguna conexión a la red. Al continuar con su trayectoria en los siguientes proyectos se fue dando cuenta que cuanto menos cambien las cosas más fáciles serán de mantener (aquí comienza a desarrollar o pensar sobre el conjunto de reglas que rigen los datos inmutables). El primer Sistema en el que comenzó a aplicar estos principios fue Peer-to-Peer Game Engine (construido en Java) sobre juegos de mesa, vio que no solo servía para juegos y lo renombró como Peer-to-Peer Correspondence Engine, eligiendo .Net como su plataforma preferida y traduciéndolo a C#.

¿Por qué utilizamos por defecto sistemas mutables?

Hay dos razones.

Así es como funciona el mundo, constantemente vemos cosas que están cambiando y en software muchas veces lo que hacemos es resolver un problema modelando el mundo real, es la forma natural de hacerlo. La otra es por la propia naturaleza de los ordenadores, están hechos para ser mutables, tienen memoria en la que puede cambiar el valor, tienen discos duros en los que podemos sobrescribir un archivo con el contenido de otro …

Pero ¿Qué es la arquitectura inmutable?

Primero comparándolo con la programación funcional, usamos la inmutabilidad para razonar en cómo se comporta un programa. Cosas como, si tuviera que hacer la misma pregunta a dos nodos diferentes dentro de un clúster, ¿obtendré la misma respuesta? Esa es una garantía que llamamos consistencia y es realmente importante. Es muy importante saber cómo un Sistema ha llegado a un determinado estado y para ello nos ayudamos de la propia inmutabilidad. La historia como tal ya es inmutable por naturaleza, cualquier hecho histórico está en el pasado, no va a cambiar. Por lo que utilizar de herramienta a la inmutabilidad para resolver problemas distribuidos nos puede ser muy útil.

Pero el concepto de arquitectura inmutable, no se refiere a una arquitectura que no pueda cambiar, porque eso obviamente sería una mal idea. Sino a una arquitectura basada en la inmutabilidad, no que ella misma lo sea.

Una arquitectura inmutable es otro paradigma en el que los servidores nunca son modificados después de ser desplegados. Si algo necesita ser actualizado o arreglado, se construyen nuevos servidores a partir de la imagen de los ya existen (cambiando lo necesario). Una vez los tenemos validados, desecharemos los anteriores.

Edu:

Aplicaciones reales de la arquitectura inmutable.

Michael menciona 3 ejemplos que todos conocemos de arquitectura inmutable: Docker, Git y el Blockchain. Todos ellos tienen algo en común que los hace tener este tipo de arquitectura, y ese algo es la identidad.

En Docker, cuando una persona se descarga una imagen, lo hace en base a un hash, siendo ese hash es el conjunto de pasos o instrucciones que se tuvieron que llevar a cabo para conseguir crear esa imagen. De esta forma, si otra persona cualquiera siguiese exactamente (y recalco el exactamente) los mismos pasos, podría conseguir exactamente la misma imagen. Por lo tanto, podría decirse que ese historial de pasos o instrucciones es de hecho la propia imagen.

En Git sucede algo similar, pues cada commit tiene un identificador único (hash), lo que le da una identidad. Lo que es realmente interesante es que ya estés en cualquier otra máquina, ya sea tuya o de un amigo o de cualquier otra persona, si tienes acceso a ese commit de ese repositorio, podrás interpretar ese commit de la misma manera en cualquier dispositivo, pues en cada dispositivo aparecerá con exactamente la misma identidad (hash). De esta forma, cuando intentáis subir a Github un código que no ha sido modificado, el propio Git te dice que no ha habido ningún cambio, pues ha comparado el hash del código existente con el del que pretendías subir y ha comprobado que eran idénticos. Podemos decir entonces que el commit es inmutable.

Esta organización se corresponde con el concepto de **location independent identy.** Este concepto dice que si una persona está hablando de algo y otra persona totalmente distinta habla de ese algo, entonces las dos hablan de lo mismo, de forma que cuando dos programadores trabajan sobre el mismo código de la misma rama, se está garantizando que trabajan sobre el mismo código.

Por último, un ejemplo que muchos conocemos pero que realmente pocos entienden en profundidad es el blockchain. Esto es un Sistema donde tu intercambias transacciones, siendo cada una de ellas inmutable. Se puede decir que son inmutables basándonos de nuevo en su identidad, pues cada hash de cada transacción es único. Si cualquier otra persona intentase replicar la transacción no podría, porque sería incapaz de conseguir el mismo hash.

CRDT (Conflict-Free Replicated Data Types)

Los CRDT son un tipo de estructuras de datos que han ganado popularidad en el contexto de sistemas distribuidos y replicación de datos, se diseñaron para permitir que los datos se repliquen en múltiples sitios sin la necesidad de coordinación centralizada o de resolución de conflictos.

Esto hace que los CRDTs puedan utilizarse para mantener el estado consistente en múltiples nodos y que su diseño conmutativo les permita manejar la concurrencia de manera efectiva.

Cabe destacar que, aunque son una herramienta poderosa, su uso no es una solución universal para todos los problemas en sistemas distribuidos y hay casos en los que pueden ser menos adecuados, por ejemplo, cuando los datos son demasiado grandes o cuando se requiere un alto grado de precisión en los resultados.

Aarón:

Theorem CAP (Consistency, Availability, Partition Tolerance)

El Teorema CAP establece que es imposible para un Sistema distribuido garantizar simultáneamente la consistencia, la disponibilidad y la tolerancia a particiones en todas las situaciones de fallo de red.

En cualquier Sistema distribuido, sólo se pueden garantizar dos de estas características al mismo tiempo. Esto significa que un Sistema distribuido debe sacrificar al menos una de estas características. Por ejemplo, si se prioriza la consistencia y la tolerancia a particiones, se puede sacrificar la disponibilidad.

Una forma de para solucionar esto podrían ser los CRDTs, una herramienta que puede ayudar a hacer frente a los desafíos del Teorema CAP, permitiendo la replicación de datos sin la necesidad de coordinación centralizada o de resolución de conflictos.

FACTS (Flexible, Available, Consistent, Tolerant and Scalable)

Los FACTS son un conjunto de principios que Michael Perry considera importantes en el diseño de sistemas distribuidos.

- La flexibilidad se refiere a la capacidad de un Sistema para adaptarse a diferentes contextos de aplicación.
- La disponibilidad se refiere a la capacidad de un Sistema para responder a las solicitudes de los clientes en todo momento.
- La consistencia se refiere a la capacidad de un Sistema para garantizar que todos los nodos del Sistema vean la misma información al mismo tiempo.
- La tolerancia se refiere a la capacidad de un Sistema para seguir funcionando incluso si hay fallos en la red que provoquen que los nodos pierdan la comunicación entre sí.
- La escalabilidad se refiere a la capacidad de un Sistema para manejar un aumento en el volumen de datos o de usuarios.

También son un complemento al teorema CAP, ya que ofrecen un conjunto de principios más específicos que pueden ayudar a los desarrolladores a tomar mejores decisiones sobre el diseño de sus sistemas distribuidos