# Software de Memoria Transaccional

Iván Félix Álvarez García

Alumno colaborador de la asignatura

7 de abril de 2014

## Índice

**1 Introducción** **2**

**2 La condena de la sincronización con concurrencia** **2**

**3 La deficiencia del modelo de objetos** **3**

**4 Separación de identidad y estado** **3**

**5 Software de Memoria Transaccional (STM)** **4**

**6 Transacciones en el STM** **7**

**7 Concurrencia usando STM** **8**

**8 Manejando las anomalías de escritura** **11**

**9 Usando Clojure STM en Java** **13**

**Bibliografía** **17**

# 1 Introducción

¿Recuerda la última vez que terminó un trabajo donde usted compartió variables mutables para sincronizar?. En lugar de relajarse y disfrutar de haber realizado un buen trabajo, usted probablemente tuviera ciertas dudas, preguntándose si ha realizado la sincronización en los lugares correctos. En la programación existen un gran número de esos momentos inquientantes. Si nosotros olvidamos sincronizar, nos esperan resultados impredicibles y potencialmente catastróficos. Pero errar es de humanos; olvidar está en nuestra naturaleza. En lugar de castigarnos, las herramientas de las que dependemos deben compensar nuestras deficiencias y ayudar a alcanzar los objetivos que nuestras mentes creativa buscan.

A lo largo de este documento, aprenderemos como jugar con la mutabilidad compartida de una forma segura usando el modelo de Software de Memoria Transaccional (Sotware Transactional Memory or STM) popularizado por Clojure. Donde sea posible, podemos cambiar o mezclar algo de Clojure en los proyectos. Pero nosotros no obligamos a usar Clojure, porque hay varias maneras de usar STM directamente en Java. Aprenderemos STM, un poco sobre como aparece en Clojure, y luego, cómo programar memoria transaccional en Java. Este modelo de programación es muy adecuado cuando se tienen frecuentes colisiones de lecturas y muy pocas colisiones de escritura. Es muy sencillo de usar y ofrece resultados predecibles.

# 2 La condena de la sincronización con concurrencia

La sincronización tiene algunos fallos fundamentales. Si nosotros la usamos de forma inadecuada o la olvidamos totalmente, los cambios realizados por un hilo pueden no ser visibles por otros hilos. A menudo, aprendemos por las malas donde tenemos que sincronizar a fin de garantizar la visibilidad y evitar las condiciones de carrera.

Desafortunadamente, cuando sincronizamos, forzamos a los hilos a esperar. La concurrencia se ve afectada por la granularidad de la sincronización, por lo que la colocación de esta en las manos de los programadores incrementa la oportunidad para que sea menos eficiente o definitivamente peor.

La sincronización puede conducir a una serie de problemas. Es fácil llevar a interbloqueo una aplicación si se mantienen los cerrojos mientras se esperan por otros. También es fácil encontrarse con problemas de bloqueo activo donde los hilos continuamente pierden un cerrojo en particular.

Nosotros podemos intentar mejorar la concurrencia haciendo los cerrojos de grano fino o granulares. Aunque en general esto es una gran idea, el riesgo se encuentra en sincronizar en el nivel más adecuado. Lo que es peor, nosotros no tenemos ninguna indicación de que se realice una sincronización correcta. Además, nosotros sólo hemos movido el punto donde esperan los hilos: el hilo sigue solicitando un acceso exclusivo y aguarda a que otros hilos esperen.

Esto es simplemente la vida en la gran ciudad para la mayoría de los programadores de Java que usan las comodidades de la concurrencia del JDK. Nos han conducido por el camino del estilo imprescindible de la programación con estados mutables durante tanto tiempo que es muy difícil ver las alternativas a la sincronización, pero las hay.

## 3 La deficiencia del modelo de objetos

Como programadores de Java, somos buenos expertos en la programación orientada a objetos (POO). Pero el lenguaje ha influido en gran medida la forma en que modelamos aplicaciones OO. La POO no acaba de llegar a ser lo que Alana Kay tenía en mente cuando acuñó el término. Su visión era sobre todo el paso de mensajes, y él quería deshacerse de los datos (el vio los sistemas que se construían por medio de mensajes que se pasaban entre las células biológicas - como los objetos que realizan operaciones pero no mantienen ningún estado). En algún lugar del camino, los lenguajes OO iniciaron el camino de la ocultación de los datos a través de los Tipos Abstractos de Datos (TADs), enlazando datos con procedimientos o combinando estados con comportamiento. Esto en gran medida nos ha llevado hacia los estados encapsulados y mutados. En el proceso, terminamos con la fusión de la identidad con el estado - la fusión de la instancia con sus datos.

Esta fusión de indentidad y estado persiste tercamente en muchos programadores de Java de una manera que sus consecuencias pueden no ser evidentes. Cuando se recorre un puntero o una referencia hacia una instancia, aterrizamos en el trozo de memoria que mantiene su estado. Se siente natural para manipular los datos de la ubicación que representa la instancia y lo que contiene. La combinación de la identidad y el estado resultó ser bastante simple y fácil de comprender. Sin embargo, esto tuvo varias consecuencias graves desde el punto de vista de la concurrencia.

Nos encontramos con problemas de concurrencia si, por ejemplo, nos estamos quedando con un informe para imprimir varios detalles de una cuenta bancaria - el número, saldo actual, transacciones, saldo mínimo, y así sucesivamente. La referencia es la puerta de entrada al estado que podría cambiar en cualquier momento. Por lo tanto, nos vemos obligados a bloquear otros hilos mientras vemos la cuenta. Y el resultado posee baja concurrencia. El problema no empezó con el bloqueo, sino con la fusión de la identidad de la cuenta con su estado.

Nos dijeron que la programación OO modela el mundo real. Lamentablemente, el mundo real no acaba de comportarse como lo que intenta modelar el actual paradigma OO. En el mundo real, el estado no cambia; la identida si lo hace. Vamos a hablar ahora de cómo esto es cierto.

## 4 Separación de identidad y estado

Rápidamente, ¿cuál es el precio de las acciones de Google? Podemos afirmar que el valor de las acciones cambian por minuto cuando la bolsa está abierta, pero eso simplemente es de alguna manera un juego de palabras. Por tomar un ejemplo, el precio de cierre de las acciones de Google el 10 de diciembre de 2010, fue de 592.21 $, y eso nunca va a cambiar - puesto que es inmutable y está escrito en la historia. Estamos mirando una instantánea de su valor en un tiempo dado. Seguro, el precio de las acciones de Google de hoy es diferente del de ese día. Si comprobamos de nuevo unos minutos más tarde (asumiendo que la bolsa esté abierta), observaremos un valor diferente, pero el viejo valor no cambia. Nosotros podemos cambiar la forma con la que vemos los objetos, y eso cambia la manera en la que los usamos. Identidad separada de su valor de estado inmutable. Vamos a ver cómo este cambio permite una programación libre de bloqueos, mejora la concurrencia, y reduce la contención al mínimo.

Esta separación de la identidad del estado es brillante - un paso clave que Rick Hickey tomó en la implementación del modelo de STM de Clojure. Imagina que nuestro objeto de la acción de Google tiene dos partes: la primera parte representa la identidad de las acciones, que a su vez tiene un puntero a la segunda parte, el estado inmutable del último valor de las acciones.

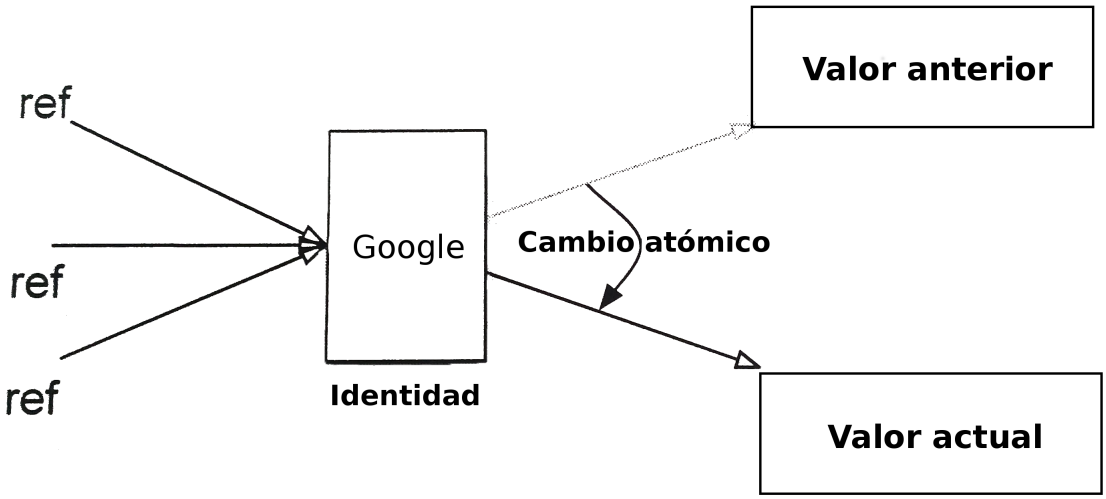


Figura 1: Ilustración del cambio del puntero de un estado inmutable a otro en la identidad Google .

Cuando recibimos un nuevo precio, lo añadimos al histórico índice de precios, sin cambiar nada de lo que sale. Ya que los viejos valores son inmutables, compartimos los datos existentes. Por lo tanto, no tenemos la duplicación y podemos disfrutar de las operaciones más rápidas, también si usamos estructuras de datos persistentes para esto. Una vez que el dato está en su lugar, hacemos un rápido cambio de la identidad para que apunte al nuevo valor.

La separación de identidad de estado es una gozada para la concurrencia. No tenemos que bloquear ninguna petición de precio de las acciones. Dado que el estado no cambia, podemos entregar fácilmente el puntero al hilo que lo solicite. Cualquier petición que llegue después de que cambiemos la identidad, verá el cambio. Lecturas sin bloqueos significa una concurrencia altamente escalable. Simplemente necesitamos garantizar que los hilos obtienen una visión coherente de su mundo. La mejor parte de todo esto es que nosotros no tenemos que hacerlo; la STM lo hace por nosotros.

## 5 Software de Memoria Transaccional (STM)

La separación de la identidad del estado ayuda a la STM a resolver los dos principales problemas con la sincronización: cruzar la barrera de la memoria y prevenir las condiciones de carrera. Primero miraremos la STM en el contexto de Clojure y luego lo usaremos en Java.

Clojure elimina los detalles amenazantes de la sincronización de la memoria, colocando los accesos a la memoria dentro de transacciones. Clojure vela y coordina las actividades de los hilos. Si no hay conflictos - por ejemplo, los hilos que trabajan con diferentes cuentas - entonces no hay cerrojos involucrados, no hay retrasos, resultando en la máxima concurrencia. Cuando dos hilos intentan acceder al mismo dato, el manejador de las transacciones interviene para resolver el conflicto, y de nuevo, no interviene ningún cierre de forma explícita en nuestro código. Vamos a investigar como funciona.

Por diseño, los valores son inmutables, y las identidades son mutables sólo dentro de las transacciones

en Clojure. Simplemente, no hay manera de cambiar el estado, y no hay facilidades de programación para eso en Clojure. Si cualquiera intenta cambiar la identidad de un objeto fuera de cualquier transacción, se producirá un fallo con un severo 'IlegalStateException'. Por otra parte, cuando se realiza con las transacciones, los cambios son instantáneos cuando no hay conflictos. Si surgen conflictos, Clojure automáticamente deshará las transacciones y lo volverá a intentar. Es nuestra responsabilidad asegurar que el código dentro de las transacciones es idempotente - en la programación funcional intentamos evitar los efectos secundarios, así que esto va bien con el modelo de programación en Clojure.

Es el momento de ver un ejemplo de Clojure STM. Usamos referencias para crear identidades mutables en Clojure. Una **ref** ofrece un cambio síncrono coordinado a la identidad del estado inmutable que representa. Vamos a crear una **ref** y vamos a intentar cambiarlo.

```
1  (def saldo (ref 0))
2
3  (println "El saldo es" @saldo)
4
5  (ref-set saldo 100)
6
7  (println "El saldo ahora es de" @saldo)
```

Considerando que el fichero que contiene el código anterior se llamase **mutate.clj**,
el comando que debemos de introducir en la terminal para su ejecución sería el siguiente:
**java -cp clojure-1.5.1.jar clojure.main mutate.clj**
Recordar que se necesita tener el fichero **clojure-1.5.1.jar** en el mismo directorio que el programa clojure que deseemos ejecutar.

Definimos una variable llamada *saldo* y la marcamos como mutable usando **ref**. Ahora *saldo* representa una identidad mutable con un valor inmutable de 0. Luego imprimimos el valor actual de esta variable. A continuación se intenta modificar la variable *saldo* usando el comando **ref-set**. Si esto sucedió, deberíamos ver el nuevo *saldo* impreso por la última sentencia. Vamos a ver cómo termina esto.

```
El saldo es 0
Exception in thread "main" java.lang.IllegalStateException: No transaction running
```

Las dos primeras sentancias trabajaron bien: hemos sido capaces de imprimir el saldo. Se puede observar que nos muestra que el valor es 0. Nuestro esfuerzo para cambiar el valor, sin embargo, falló con un 'IlegalStateException'. Hemos enfadamos a los dioses de Clojure por la mutación de una variable fuera de una transacción. Contraste este incumplimiento con Java cuando modificamos una variable mutable compartida sin sincronización. Esto supone una gran mejora, ya que preferimos que nuestro código se comporte como debe o, en caso de que fallara, que lo hiciese a gritos antes que produzca resultados impredecibles en silencio. Les invito a celebrarlo. Aunque veo que está ansioso por arreglar ese error de Clojure STM, así que vamos a seguir adelante.

Crear una transacción en Clojure es sencillo. Simplemente necesitamos envolver el bloque de código con una llamada **dosync**. Estructuralmente, esto es similar al bloque *sincronizado* en Java, pero hay algunas pequeñas diferencias:

- Obtenemos una clara advertencia si nos olvidamos de poner **dosync** alrededor del código que está mutando.
- En lugar de crear un bloqueo exclusivo, **dosync** permite al código competir en igualdad con otros hilos envolviendolo en una transacción.
- Dado que no se realiza ningún bloqueo explícito, no tenemos que preocuparnos por el orden de bloqueo y se puede disfrutar de una concurrencia libre de interbloqueos.
- En lugar de prevenir en tiempo de diseño el *que bloquea el qué y en qué orden*, STM proporcionar una sencilla composición transaccional de cerrojos en tiempo de ejecución.
- Al no tener cerrojos explícitos, significa que ningún programador colocará bloques de código mutuamente excluyentes. Por lo tanto, conseguiremos mitigar esos bloques de códigos conservadores.

Cuando una transacción está en curso, si no había ningún conflicto con otros hilos / transacciones, la transacción puede completarse, y sus cambios pueden ser escritos en memoria. Sin embargo, tan pronto como Clojure encuentre que alguna otra transacción ha progresado muy por delante de manera que ponga en peligro esta transacción, tranquilamente deshace los cambios y se repite la transacción. Vamos a arreglar el código de modo que podamos cambiar con éxito la variable *saldo*.

```
1  (def saldo (ref 0))
2
3  (println "El saldo es" @saldo)
4
5  (dosync
6    (ref-set saldo 100))
7
8  (println "El saldo ahora es" @saldo)
```

Considerando que el archivo que contiene el código anterior se llamase **mutatesuccess.clj**, el comando que debemos de introducir en la terminal para su ejecución sería el siguiente:
**java -cp clojure-1.5.1.jar clojure.main mutatesuccess.clj**

El único cambio en el código ha sido envolver la llamada a **ref-set** en una llamada a **dosync**. Por supuesto, no estamos restringidos a escribir una única línea de código dentro de *dosync*; podemos envolver todo un bloque de código o varias expresiones dentro de una transacción. Vamos a ejecutar el código para ver el cambio.

```
El saldo es 0
El saldo ahora es 100
```

El valor del estado a 0 para el saldo es inmutable, pero la identidad de la variable *saldo* es mutable. Dentro de la transacción creamos primero un nuevo valor a *100* ( tenemos que acostumbrarnos a la idea del valor inmutable ), pero el antiguo valor de *0* sigue ahí, y *saldo* está apuntando a él en este momento. Una vez que hemos creado el nuevo valor, le pedimos a Clojure que rápidamente modifique el puntero en *saldo* al nuevo valor. Si no hay otras referencias al valor antiguo, el recolector de basura se hará cargo de él adecuadamente.

Clojure proporciona tres opciones para modificar una identidad mutable, todas sólo desde dentro de una transacción:

- **ref-set** establece el valor de la identidad y devuelve el valor.
- **alter** establece el valor en transacción de la identidad con el valor resultante de la aplicación de una función especificada y devuelve el valor.
- **commute** separa los cambios en la transacción desde el punto de guardado. Se establece el valor de la identidad en la transacción con un valor resultante de la aplicación de una función específica y se devuelve el resultado. Luego, durante el punto de guardado, el valor de la identidad se establece del resultado de aplicar la función específica usando el valor guardado más reciente de la identidad.

*Commute* es útil cuando estamos contentos con el último funcionamiento y, además, ofrece una mayor concurrencia respecto *alter*. Sin embargo, en la mayoría de las situaciones, *alter* es más adecuado que *commute*.

Además de las referencias, Clojure también proporciona atómicos. Éstos proporcionan un cambio asíncrono a los datos; sin embargo, a diferencia de las *referencias*, el cambio es descoordinado y no puede ser agrupado con otro cambio en una transacción. Los atómicos no participan en la transacción ( no podemos pensar en que cada cambio atómico pertenece a una transacción por separado ). Para cambios aislados o discretos, usamos **atom**; para cambios agrupados o coordinados, usamos las **referencias**.

## 6 Transacciones en el STM

Usted ha utilizado, sin duda, las transacciones en las bases de datos y está familiarizado con su atomicidad, coherencia, aislamiento y durabilidad (ACID). La STM de Clojure proporciona las primeras tres de estas propiedades, pero no proporciona la durabilidad, porque los datos están en su totalidad en la memoria y no en un sistema de base de datos o archivo.

1. **Atomicidad**. Las transacciones de la STM son atómicas. O bien todos los cambios que nosotros realizamos en una transacción son visibles fuera de la misma o ninguno. Todos los cambios en las referencias en una transacción se mantienen o ninguno en absoluto.
2. **Consistencia**. O bien la transacción termina de ejecutarse por completo y vemos su cambio o fracasa, dejando las cosas sin afectar. Desde fuera de estas transacciones, vemos un cambio consistente después de otro. Por ejemplo, al finalizar dos depósitos a nuestro saldo de manera separada y concurrente y retirar las transacciones, el saldo se encuentra en un estado coherente con efecto acumulativo debido a ambas acciones.
3. **Aislamiento**. Las transacciones no ven los cambios parciales de otras transacciones, y los cambios se ven únicamente al finalizar.

Estas propiedades se centran en la integridad y la visibilidad de los datos. Sin embargo, el aislamiento no significa una falta de coordinación. Al contrario, la STM sigue de cerca el progreso de todas las transacciones y trata de ayudar a cada una de ellas para que se ejecuten hasta el final (salvo las excepciones en la aplicación).

Clojure STM usa el Multiversion Concurrency Control (MVCC) al igual que las bases de datos. El control de la concurrencia de la STM es similar al bloqueo optimista en las bases de datos. Cuando comenzamos una transacción, la STM realiza una marca de tiempo y copia las referencias que nuestra transacción va a utilizar. Dado que el estado es inmutable, estas copias de las referencias son rápidas y baratas. Cuando hacemos un "cambio" a cualquier estado inmutable, no estamos realmente modificando los valores. Al contrario, estamos creando nuevas copias de estos valores. Las copias se guardan localmente para la transacción, y gracias a la persistencia de las estructuras de datos este paso se realiza también rápidamente. Si en algún momento la STM determina que las referencias que hemos cambiado han sido modificadas por otra transacción, aborta y vuelve a intentar nuestra transacción. Cuando una transacción se completa con éxito, los cambios se escriben en memoria y las marcas de tiempo se actualizan.

## 7 Concurrencia usando STM

Las transacciones son bonitas, pero, ¿qué pasa si dos transacciones intenta cambiar la misma identidad?. Vamos a echar un vistazo a un par de ejemplos de esto en esta sección.

Unas palabras de advertencia que hay que considerar antes de entrar en los ejemplos. En la producción del código, tenemos que asegurarnos de que las transacciones son idempotentes y no tienen ningún efecto secundario, ya que pueden que tengan que volverse a procesar en varias ocasiones. Eso significa que no hay impresiones por la consola, no hay registro, no hay envío de emails, y no se hacen operaciones irreversibles dentro de las transacciones. Si hacemos alguna de las acciones anteriores, o somos responsables de la inversión de las operaciones o hacemos frente a las consecuencias. En general, es mejor recoger estas acciones-efectos secundarios y llevarlas a cabo después de que la transacción haya tenido éxito.

En contra del consejo que se acaba de mencionar, veremos declaraciones de impresiones dentro de las transacciones en los ejemplos. Esto es puramente para fines ilustrativos. ¡No intente hacer esto en sus proyectos!

Ya sabemos cómo cambiar el *saldo* dentro de una transacción. Vamos ahora a permitir multiples transacciones que compitan por el saldo.

```
1  (defn deposito [saldo cantidad]
2    (dosync
3      (println "Preparado para depositar..." cantidad)
4      (let [saldoactual @saldo]
5        (println "Simulando retraso en deposito...")
6        (. Thread sleep 2000)
7        (alter saldo + cantidad)
8        (println "Hecho deposito de" cantidad))))
9
10 (defn reintegro [saldo cantidad]
11   (dosync
12     (println "Preparado para reintegro..." cantidad)
13     (let [saldoactual @saldo]
14       (println "Simulando retraso en reintegro...")
15       (. Thread sleep 2000)
16       (alter saldo - cantidad)
17       (println "Hecho reintegro de" cantidad))))
18
```

```
19 (def saldo1 (ref 100))
20
21 (println "El saldo1 es" @saldo1)
22
23 (future (deposito saldo1 20))
24 (future (reintegro saldo1 10))
25
26 (. Thread sleep 10000)
27
28 (println "El saldo1 ahora es" @saldo1)
```

Considerando que el archivo que contiene el código anterior se llamase **concurrentChangeToBalance.clj**, el comando que debemos de introducir en la terminal para su ejecución sería el siguiente: **java -cp clojure-1.5.1.jar clojure.main concurrentChangeToBalance.clj**

Hemos creado dos transacciones en este ejemplo, una para el depósito y otra para el reintegro. En la función *deposito()* primero se realiza una copia local del saldo dado. Luego, simulamos un retraso para establecer las transiciones de camino a una colisión. Después del retraso, incrementamos el saldo. La función *reintegro()* es muy similar, excepto que en esta, decrementamos el saldo. Es hora de probar estos dos métodos. Primero vamos a inicializar la variable *saldo1* a 100 y permitimos que los dos métodos anteriores se ejecuten en dos hilos por separado usando la función **future()**. Sigamos adelante y ejecutemos el código y observemos la salida:

```
El saldo1 es 100
Preparado para depositar... 20
Simulando retraso en deposito...
Preparado para reintegro... 10
Simulando retraso en reintegro...
Hecho deposito de 20
Preparado para reintegro... 10
Simulando retraso en reintegro...
Hecho reintegro de 10
El saldo1 ahora es 110
```

Ambas funciones obtienen su propia copia local del saldo. Justo después de simular el retardo, la transacción *deposito()* termina, pero la transacción *reintegro()* no tuvo tanta suerte. El saldo se cambió "por la espalda"; por lo que el cambio que intenta hacer ya no es válido. La STM de Clojure aborta en silencio la transacción y vuelve a intentarla (se puede observar cuando se vuelve a ejecutar por segunda vez la sentencia que muestra *"Preparado para reintegro... 10"*). Si no tuviéramos estas sentencias para la impresión, no seríamos conscientes de estas actividades. Todo lo que nos debería de importar es que el efectivo neto del saldo es consistente y refleja tanto el depósito como el reintegro.

Intencionadamente, establecimos las dos transacciones de camino a una colisión en este ejemplo, para la obtención previa del saldo y el retraso de la ejecución. Si eliminamos la declaración 'let' de ambas transacciones, nos daremos cuenta de que ambas transacciones terminan por completo de manera consistente, sin la necesidad de que cualquiera de ellas tenga que repetirse, mostrándonos que la STM provee máxima concurrencia mientras que preserva la consitencia.

Ahora sabemos cómo cambiar una simple variable, pero y ¿Si tenemos una colección de valores?. Las listas son inmutables en Clojure. Sin embargo, nosotros podemos obtener una referencia mutable cuya identidad pueda cambiar, haciendo que parezca como si cambiamos la lista. La lista no ha cambiado; nosotros simplemente cambiamos la percepción de la visión de la lista. Vamos a investigar esto con un ejemplo. La lista de los deseos de mi familia originalmente contiene sólo un articulo, un Ipad. Me gustaría añadir un nuevo MacBook Pro (MBP), y uno de mis hijos quiere añadir una nueva bicicleta. Por lo que lo establecemos en dos hilos diferentes para añadir estos elementos a la lista. Aquí el código para eso:

```
1  (defn insertarelemento [listadeseos elemento]
2    (dosync (alter listadeseos conj elemento)))
3
4  (def listadeseosFamilia (ref '("iPad")))
5  (def listadeseosOriginal @listadeseosFamilia)
6
7  (println "La lista de los deseos original es" listadeseosOriginal)
8
9  (future (insertarelemento listadeseosFamilia "MBP"))
10 (future (insertarelemento listadeseosFamilia "Bike"))
11
12 (. Thread sleep 1000)
13
14 (println "La lista de los deseos original es" listadeseosOriginal)
15 (println "La lista de los deseos actualizada es" @listadeseosFamilia)
```

Considerando que el archivo que contiene el código anterior se llamase **concurrentListChange.clj**, el comando que debemos de introducir en la terminal para su ejecución sería el siguiente:
**java -cp clojure-1.5.1.jar clojure.main concurrentListChange.clj**

En la función *insertarelemento()*, añadimos el elemento dado a la lista. La función **alter** altera la referencia en la transacción mediante la función prevista. En este caso, la función **conj()**. La función **conj()** devuelve una nueva colección con los elementos unidos o añadidos a la colección. Nosotros luego llamamos al método *insertarelemento()* desde los dos hilos. Ejecute el código para ver el efecto de las dos transacciones.

```
La lista de los deseos original es (iPad)
La lista de los deseos original es (iPad)
La lista de los deseos actualizada es (Bike MBP iPad)
```

La lista original es inmutable, por lo que sigue siendo la misma hasta el final del código. Como añadimos elementos a esta lista, obtuvimos nuevas copias de la lista. Sin embargo, ya que la lista es una estructura de datos persistente, nos vemos beneficiados tanto en memoria como en rendimiento a la hora de compartir los elementos.

Tanto el estado de la lista como de la referencia *listadeseosOriginal* son inmutables. *listadeseosFamilia*, sin embargo, es una referencia mutable. Cada una de la solicitudes de los elementos se ejecuta en su propia transacción. La primera en tener éxito cambió la referencia mutable a la nueva lista. Sin embargo, ya que la lista en sí es inmutable, la nueva lista es capaz de compartir el elemento *iPad* de la lista original. Cuando la segunda transacción tiene éxito, comparte internamente los otros dos elementos añadidos previamente.

## 8 Manejando las anomalías de escritura

Vimos cómo la STM maneja los conflictos de escritura entre las transacciones. A veces, los conflictos no son tan directos. Imaginemos que tenemos dos cuentas, una cuenta corriente y una cuenta de ahorros en un banco que ha establecido como requisito que el total de saldo mínimo entre las dos cuentas sea de 1000$. Ahora supongamos que el saldo de estas cuentas es de 500$ y 600$, respectivamente. Podemos retirar hasta 100$ de una de estas cuentas, pero no de ambas. Si se realiza un reintegro de 100$ de cada una de ellas de manera secuencial, el primer reintegro si tendrá éxito, pero el segundo no. La denominada anomalía de escritura no impedirá que estas dos transacciones se realicen concurrentemente - ambas transacciones ven que el total del saldo mínimo es mayor que 1000$, por lo que proceden a modificar dos valores diferentes sin conflictos de escritura. El resultado neto, un saldo de 900$, es menor que el límite permitido. Vamos a crear esta anomalía en el código, a continuación, averiguaremos cómo arreglarla.

```
1  (def cuentaCorriente (ref 500))
2  (def cuentaAhorros (ref 600))
3
4  (defn retirarcuenta [saldoExtraer saldoRestringir cantidad]
5    (dosync
6      (let [saldoTotal (+ @saldoExtraer @saldoRestringir)]
7        (. Thread sleep 1000)
8        (if (>= (- saldoTotal cantidad) 1000)
9          (alter saldoExtraer - cantidad)
10         (println "Lo sentimos, no se puede realizar el reintegro, produce error")))))
11
12 (println "La cuenta corriente tiene" @cuentaCorriente)
13 (println "La cuenta de ahorros tiene" @cuentaAhorros)
14 (println "El saldo total es de" (+ @cuentaCorriente @cuentaAhorros))
15
16 (future (retirarcuenta cuentaCorriente cuentaAhorros 100))
17 (future (retirarcuenta cuentaAhorros cuentaCorriente 100))
18
19 (. Thread sleep 2000)
20
21 (println "La cuenta corriente tiene" @cuentaCorriente)
22 (println "La cuenta de ahorros tiene" @cuentaAhorros)
23 (println "El saldo total es de" (+ @cuentaCorriente @cuentaAhorros))
```

Considerando que el archivo que contiene el código anterior se llamase **writeSkew.clj**, el comando que debemos de introducir en la terminal para su ejecución sería el siguiente:
**java -cp clojure-1.5.1.jar clojure.main writeSkew.clj**

Empezamos con los saldos dados para las dos cuentas. En la función *retirarcuenta()*, leemos por primera vez el saldo mínimo de las dos cuentas y calculamos el saldo total. Luego, tras el retraso, el cual pone las transacciones en camino hacia la colisión, actualizamos el saldo de una de las cuentas representadas por *saldoExtraer* sólo si el saldo mínimo total no es menor que el mínimo requerido. En el resto del código, se ejecutan concurrentemente las dos transacciones, donde primero retira 100$ del saldo de la cuenta corriente, mientras que el segundo, retira 100$ del saldo de la cuenta de ahorros. Ya que las dos transacciones se ejecutan aisladamente y no modifican nada que tengan en común, son bastante ajenas a la violación y al desajuste en la escritura, como se puede ver en la siguiente salida:

```
La cuenta corriente tiene 500
La cuenta de ahorros tiene 600
El saldo total es de 1100
La cuenta corriente tiene 400
La cuenta de ahorros tiene 500
El saldo total es de 900
```

Con Clojure, podemos evitar estos desajustes de escritura utilizando el método **ensure()**. Podemos decirle a una transacción que eche un ojo a la variable para que sólo se lea y no se modifique. La STM entonces garantizará que las escrituras son realizadas sólo si los valores que hemos leído no cambian fuera de la transacción; en otro caso, reintenta la transacción.

Vamos a modificar el método *retirarcuenta()*:

```
1  (def cuentaCorriente (ref 500))
2  (def cuentaAhorros (ref 600))
3
4  (defn retirarcuenta [saldoExtraer saldoRestringir cantidad]
5    (dosync
6      (let [saldoTotal (+ @saldoExtraer (ensure saldoRestringir))]
7        (. Thread sleep 1000)
8        (if (>= (- saldoTotal cantidad) 1000)
9          (alter saldoExtraer - cantidad)
10         (println "Lo sentimos, no se puede realizar el reintegro, produce error")))))
11
12 (println "La cuenta corriente tiene" @cuentaCorriente)
13 (println "La cuenta de ahorros tiene" @cuentaAhorros)
14 (println "El saldo total es de" (+ @cuentaCorriente @cuentaAhorros))
15
16 (future (retirarcuenta cuentaCorriente cuentaAhorros 100))
17 (future (retirarcuenta cuentaAhorros cuentaCorriente 100))
18
19 (. Thread sleep 2000)
20
21 (println "La cuenta corriente tiene" @cuentaCorriente)
22 (println "La cuenta de ahorros tiene" @cuentaAhorros)
23 (println "El saldo total es de" (+ @cuentaCorriente @cuentaAhorros))
```

Considerando que el archivo que contiene el código anterior se llamase **noWriteSkew.clj**, el comando que debemos de introducir en la terminal para su ejecución sería el siguiente:
**java -cp clojure-1.5.1.jar clojure.main noWriteSkew.clj**

En la línea 6, llamamos a **ensure()** sobre el valor *saldoRestringir*, el cual leemos pero no modificamos en la transacción. En este punto, la STM coloca un cerrojo de lectura en ese valor, lo que impide que otras transacciones adquieran un cerrojo de escritura. Cuando la transacción termina, la STM libera todos los cerrojos de lectura antes de que realice los cambios. Esto evita cualquier posibilidad de interbloqueo al tiempo que aumenta la concurrencia.

Incluso aunque las dos transacciones se ejecuten concurrentemente como antes, con esta modificación

del método *retirarcuenta()* que llama al método **ensure()**, la restricción del saldo mínimo total se conserva, como podemos ver en la salida:

```
La cuenta corriente tiene 500
La cuenta de ahorros tiene 600
El saldo total es de 1100
La cuenta corriente tiene 500
La cuenta de ahorros tiene 500
El saldo total es de 1000
Lo sentimos, no se puede realizar el reintegro, produce error
```

El modelo de ejecución sin bloqueos explícitos de la STM es bastante potente. Cuando no hay conflictos, no hay bloqueos en absoluto. Cuando hay conflictos, un ganador avanza sin ningún tipo de problemas, mientras los contendientes se repiten. Clojure usa un número máximo de intentos asegurando que dos hilos no se repiten al mismo paso y terminan perdiendo repetidamente. La STM es una magnífico modelo cuando tenemos frecuentes colisiones de lectura pero muy pocas colisiones de escritura. Por ejemplo, este modelo se ajusta muy bien para las típicas aplicaciones web - Por lo general, tenemos varios usuarios al mismo tiempo que realizan actualizaciones de sus propios datos, pero tenemos pocas colisiones de estado compartido. Cualquiera de las infrecuentes colisiones de escritura que se puedan producir, puede ser resuelta sin esfuerzo por la STM.

Clojure STM es la aspirina del mundo de la programación concurrente - que puede eliminar mucho dolor. Por la simple colocación de **dosync** en el lugar correcto, estamos recompensados con una alta concurrencia y consistencia entre los hilos. Debido a su comportamiento conciso, altamente expresivo, y bastante predecible, todo esto hace de Clojure STM una opción digna de consideración.

## 9 Usando Clojure STM en Java

Podemos utilizar Clojure STM en Java de una forma bastante sencilla porque **Ref** y **LockingTransaction** se ofrecen como simples clases. El método **runinTransaction()** toma como parámetro una instancia que implementa la interfaz *Callable*. Así que envolver código en una transacción es tan simple como envolverlo en el método **call()** de la interfaz *Callable*.

Vamos a implementar el ejemplo de la transferencia de cuenta a través de Clojure STM en Java. En primer lugar vamos a crear una clase *Account* que contendrá una referencia a su estado inmutable representado por la variable *saldo*:

**Nota**: Todos los códigos que se van a mostrar a continuación, hasta nuevo aviso, pertenecen al mismo fichero java llamado *Account.java*.

```
public class Account {
  final private Ref saldo;

  public Account(final int saldoInicial) throws Exception {
    saldo = new Ref(saldoInicial);
  }

  public int getSaldo() { return (Integer) saldo.deref(); }
```

Inicializamos la referencia administrada con el saldo inicial. Para obtener el saldo actual, se utiliza el método **deref()**, que es el método ofrecido en la API de Java para el prefijo @ que usamos en Clojure para desreferenciar. El código será envuelto dentro de la transacción que estará situada dentro del *Callable* que se pasará al método **runinTransaction()**, como veremos en el método *deposito()*:

```
public void deposito(final int cantidad) throws Exception {
  LockingTransaction.runInTransaction(new Callable<Boolean>() {
    public Boolean call()  {
      if(cantidad > 0) {
        final int saldoActual = (Integer) saldo.deref();
        saldo.set(saldoActual + cantidad);
        System.out.println("Deposito de " + cantidad + "... se va a realizar");
        return true;
      } else throw new RuntimeException("Operacion invalida");
    }
  });
}
```

El depósito modifica el actual saldo en una transacción si la *cantidad* es mayor que 0. En otro caso, falla la transacción lanzando una excepción.

La implementación del método *reintegro()* no es muy distinta de la implementación del método anterior; Sólo se realiza una comprobación adicional.

```
public void reintegro(final int cantidad) throws Exception {
  LockingTransaction.runInTransaction(new Callable<Boolean>() {
    public Boolean call() {
      final int saldoActual = (Integer) saldo.deref();
      if(cantidad > 0 && saldoActual >= cantidad) {
        saldo.set(saldoActual - cantidad);
        return true;
      } else throw new RuntimeException("Operacion invalida");
    }
  });
}
```

Todo esto era respecto la clase *Account*. Vamos a centrarnos ahora en la transferencia; vamos a escribir una clase por separado. Para ver el efecto de un depósito rechazado a causa de un reintegro fallido, vamos a realizar el depósito primero seguido del reintegro:

**Nota**: A partir de aquí, todos los códigos pertenecen a la nueva clase Transfer localizada en el fichero *Transfer.java*.

```
public class Transfer {
  public static void transfer(
    final Account cuentaOrigen, final Account cuentaDestino, final int cantidad)
    throws Exception {
    LockingTransaction.runInTransaction(new Callable<Boolean>() {
      public Boolean call() throws Exception {
        cuentaDestino.deposito(cantidad);
```

```
        cuentaOrigen.reintegro(cantidad);
        return true;
      }
    });
```

Vamos a necesitar un método que se encarge de la transferencia fallida e informe del estado de las cuentas depués de la transferencia:

```
  public static void transferirYmostrar(
    final Account cuentaOrigen, final Account cuentaDestino, final int cantidad) {
    try {
      transfer(cuentaOrigen, cuentaDestino, cantidad);
    } catch(Exception ex) {
      System.out.println("Transferencia fallida " + ex);
    }

    System.out.println("Saldo cuenta cuentaOrigen: " + cuentaOrigen.getSaldo());
    System.out.println("Saldo cuenta cuentaDestino: " + cuentaDestino.getSaldo());
  }
```

Vamos a realizar una primera transferencia exitosa seguida de una transferencia que fracasará por falta de fondos:

```
  public static void main(final String[] args) throws Exception {
    final Account cuenta1 = new Account(2000);
    final Account cuenta2 = new Account(100);

    transferirYmostrar(cuenta1, cuenta2, 500);
    transferirYmostrar(cuenta1, cuenta2, 5000);
  }
}
```

Ejecute el código y estudie la salida:

Considerando que tenemos los siguientes ficheros java: **Account.java** y **Transfer.java**, y además, tenemos el archivo **clojure-1.5.1.jar** en el mismo directorio que ambos ficheros, el comando que debemos de introducir en la terminal para su compilación sería el siguiente:
**javac -cp clojure-1.5.1.jar Account.java Transfer.java**

Una vez obtenemos las clases como resultado de la compilación, el comando para ejecutar el programa principal, que se encuentra en la clase *Transfer*, sería:
**java -cp clojure-1.5.1.jar: Transfer**

```
Deposito de 500... se va a realizar
Saldo cuenta cuentaOrigen: 1500
Saldo cuenta cuentaDestino: 600
Deposito de 5000... se va a realizar
Transferencia fallida java.lang.RuntimeException: Operacion invalida
```

```
Saldo cuenta cuentaOrigen: 1500
Saldo cuenta cuentaDestino: 600
```

Como era de esperar, la primera transferencia tuvo éxito, y el saldo que se muestra lo refleja. La segunda transferencia debía de fallar debido a la falta de fondos. Vemos que el depósito que se llevó a cabo fue negado por el fallo en el reintegro cuando la transacción retrocedió. La transferencia fallida dejó el saldo de las dos cuentas sin afectar.

# Bibliografía

[1] Subramaniam, Venkat. *Programming Concurrency on the JVM.* The Programatic Bookshelf, 2011