Memoria Transaccional Software con Clojure/Java

Seminario Número 2

Antonio J. Tomeu[1]

[1]Departamento de Ingeniería Informática
Universidad de Cádiz

PCTR, 2019

Contenido

1. Concepto de Memoria Transaccional Sofware
2. Transacciones: Definición y Propiedades
3. Funcionamiento de una Transacción
4. Algoritmo de Procesamiento de una Transacción
5. Memoria Transaccional Software con Clojure
6. Memoria Transaccional Software con Java sobre Clojure

Contexto Actual de la Concurrencia

- Estamos inmersos en la que se ha llamado revolución `multicore`
- Es necesario explotar el nuevo hardware reestructurando la forma de escribir aplicaciones
- La concurrencia y el paralelismo resuelven las nuevas necesidades...
- pero añaden problemas nuevos: exclusión mutua y sincronización
- Las técnicas tradiciones de bloqueo para resolverlas presentan serios incovenientes:
  - reducen el paralelismo posible
  - introducen efectos indeseables como los interbloqueos

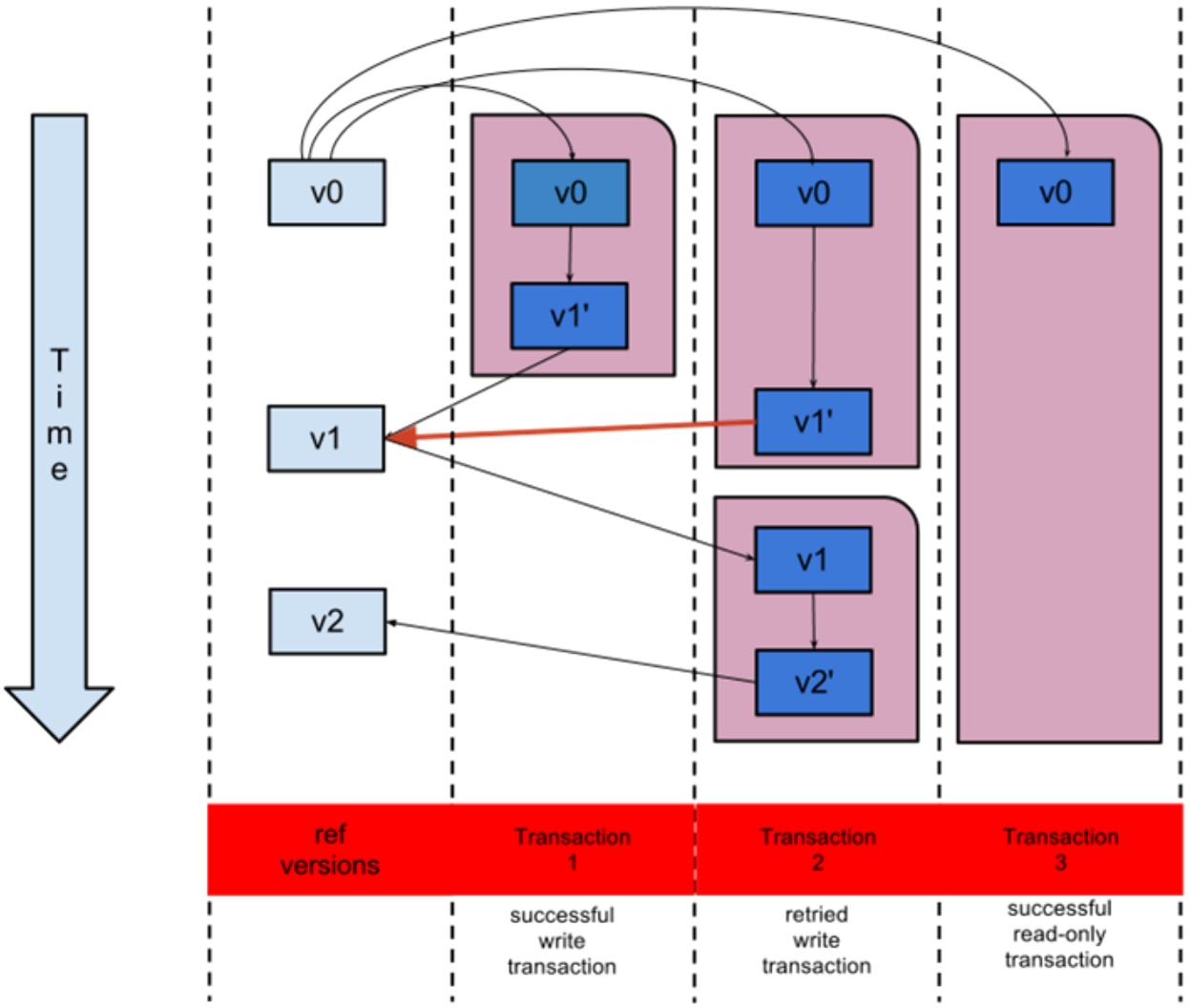
¿Qué es la Memoria Transaccional Software (STM)?

- Una técnica de control de la concurrencia alternativa al modelo de bloqueos, similar al sistema de transacciones de una base de datos
- Las operaciones sobre datos compartidos se ejecutan dentro de una transacción, lo cuál permite:
  - disponer de operaciones libres de bloqueos (y de interbloqueos)
  - aumentar el paralelismo en el acceso a los datos compartidos
  - evitar las condiciones de concurso continuas sobre un mismo cerrojo
- La STM simplifica la programación concurrente sustituyendo el acceso a datos compartidos bajo control de cerrojo, por el acceso a los mismos dentro de una transacción

Concepto de Transacción

- Una transacción es una secuencia de operaciones desarrolladas por una tarea concurrente (proceso o hebra) sobre un conjunto de datos compartidos que puede terminar de dos maneras:
  - Validando (*commit*) las transacción: todas las actualizaciones efectuadas por la tarea sobre los datos compartidos tienen efecto atómicamente
  - Abortando la transacción; los cambios no tienen efecto y los datos no cambian. Típicamente la transacción se intenta de nuevo (*roll-up*).
- Las transacciones habitualmente cumplen las conocidas como propiedades ACID:
  - Atomicity (atomicidad)
  - Consistency (consistencia)
  - Isolation (aislamiento)
  - Durability (durabilidad)

¿Cómo Funciona una Transacción?



¿Y desde un punto de vista técnico?

- El marco que da soporte a la STM no es técnicamente sencillo ya que
  - Debe garantizar la ejecución atómica de las transacciones
  - Asegura que se cumple ACID, y
  - todo ello sin utilizar cerrojos
- En la literatura se proponen enfoques basados en hardware, en software e híbridos para soportar técnicamente el concepto de memoria trasaccional software
- El enfoque software el más utilizado con diferencia (por ejemplo, para Java, hay decenas de implementaciones posibles)
- Todo lenguaje o API que soporte STM necesita un **manejador de transacciones**, que es el algoritmo que se encarga de procesar las transacciones que el programador escribe garantizando las propiedades ACID.

Algoritmo de Procesamiento de una Transacción I

Código 1: code/algorithm.cpp

```
1  Algoritmo de procesamiento de una Transaccion
2  1. Inicio
3  2. Hacer copia privada de los datos compartidos
4  3. Hacer actualizaciones en la copia privada
5   4. Ahora:
6    4.1 Si datos compartidos no modificados->actualizar datos
7      compartidos con copia privada y goto(Fin)
8    4.2 Si hay conflictos, descartar copia privada y goto(Inicio)
9  5. Fin
```

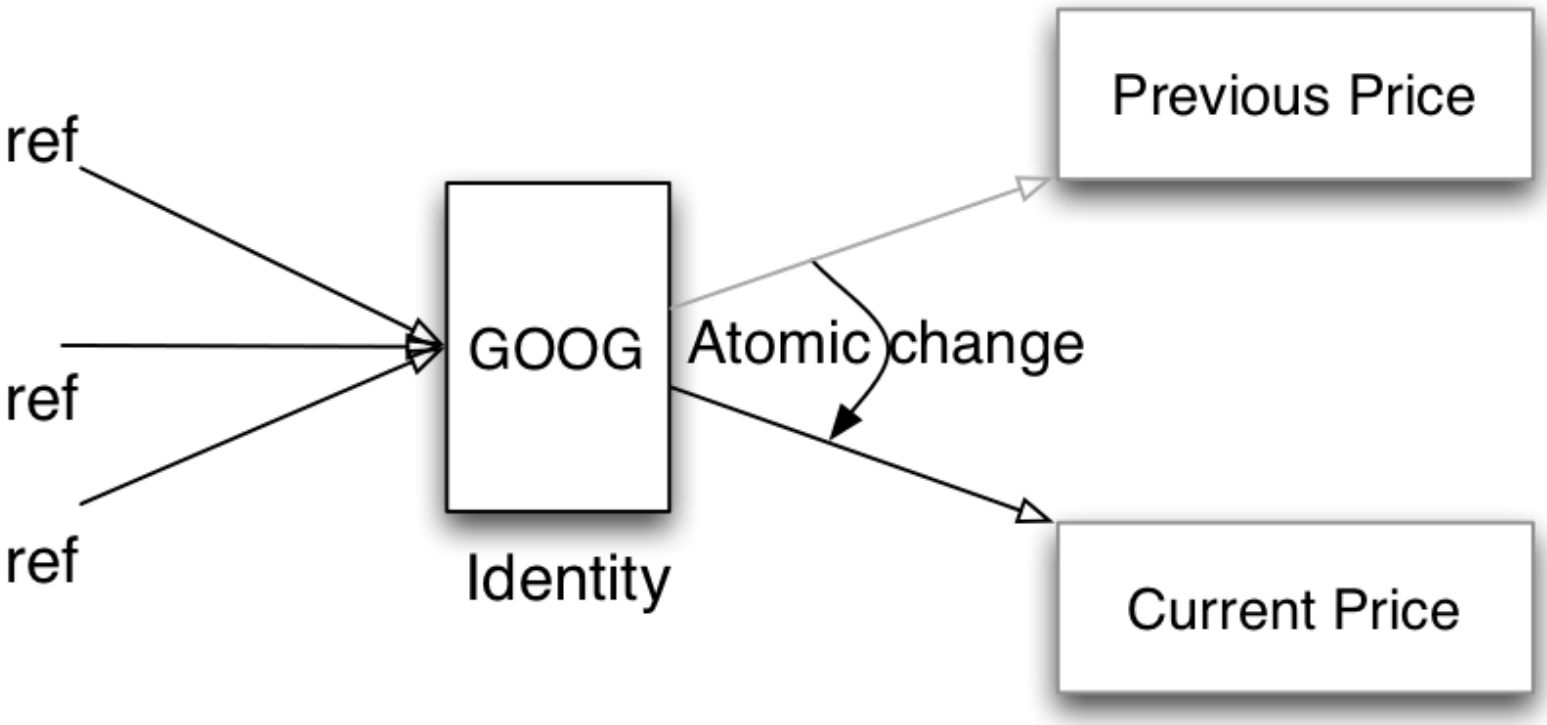
STM con el Lenguaje Clojure I

- Clojure es un lenguaje funcional interpretado diseñado para utilizar como intéprete la JVM
- No usa punto y coma, llaves, etc. Su notación es -casi- puramente funcional
- Utiliza notación prefija
- En Clojure, todo es una función; y todo programa, una lista de funciones

Separación Identidad y Estado con Clojure I

- La orientación a objetos estándar fusiona los conceptos de identidad y estado para un objeto dado
- Esto no modela adecuadamente las situaciones del mundo real; en él, la identidad (el objeto) está separada del estado que posee
- Ejemplo: el precio en bolsa de la acción de Google (identidad, objeto) el día 10/12/2010 era de 592.21 dólares. Este valor (el estado) de la acción es inmutable, con independencia del valor actual (nuevo estado) de la acción
- Con Clojure, los valores (estados) son inmutables, y las identidades únicamente pueden cambiar dentro de una transacción

Separación Identidad y Estado con Clojure



Guía Elemental de Clojure

- Suma o división de dos números: `(+ 4 6)` y `(/ 5 3)`
- Suma de una lista: `(+ 4 6 1 2 2 3 5 6 4)`
- Funciones definidas por el usuario:
  - Se utiliza **defn** seguido del nombre de la función, los parámetros y el cuerpo de la función
  - Definimos la función `misuma`: `(defn misuma [x y] (+ x y))`
  - Y la utilizamos: `misuma 4 6)`
- En Cloure, las funciones son un tipo más. Pueden ser definidas anónimamente con `fn` y evaluadas de inmediato:
- `((fn [x y] (- x y)) 5 2)`
- En realidad, `defn` es un sinónimo para `def` y `fn`. Con `fn` declaramos una función, y con `def` creamos para ella una referencia en forma de nombre
- La función `misuma` también se puede definir así:
- `(def suma (fn [x y] (+ x y)))`

¿Para qué sirve todo esto?

- Recordemos: en Cloure todos los valores (estados) son inmutables y las identidades sólo pueden cambiar dentro de una transacción.
- `ref` permite crear identidades mutables.
- `def` no es únicamente válido para funciones, sino que es aplicable a cualquier otro tipo. Nosotros la utilizaremos junto con `ref`

¿Cómo Tener Transacciones con Clojure?

- Una transacción se crea envolviendo el código de interés (que normalmente serán los datos compartidos) en un bloque **dosync** de forma parecida a lo que hacíamos con `synchronized` en Java. Ojo: ¡no es lo mismo!
- Una vez dentro del bloque transaccional, podemos cambiar el estado de una identidad de dos formas:
  - utilizando `ref-set` que establece el nuevo estado de la identidad y lo devuelve
  - utilizando `alter` que establece el nuevo estado de la identidad como el resultado de aplicar una función y lo devuelve

Ejemplo: Intento Ilegal de Cambio de Estado I

Código 2: code/mutate.clj

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

- La identidad balance tiene como estado 0
- Se intenta cambiar el estado de la identidad mediante (ref-set balance 100)...
- ... lo cuál lanza una excepción, ya que Clojure sólo permite el cambio de una identidad dentro de una transacción

Ejemplo: Intento Legal de Cambio de Estado (con transacción) I

Código 3: code/mutatesuccess.clj

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

- La identidad balance tiene como estado 0
- Se intenta cambiar el estado de la identidad mediante (ref-set balance 100)...
- ... pero dentro de una transacción (dosync)
- La modificación es realizada

Ejemplo: Cuenta Corriente I

- Para controlar el acceso concurrente a los datos de una cuenta corriente, escribiremos funciones para `deposito` y `reintegro`
- El código que altera la identidad `saldo` cambiando el estado irá dentro de una transacción
- Lanzaremos dos tareas concurrentes que harán respectivamente un depósito y un reintegro
- El saldo final resultante será coherente, gracias al control transaccional de las modificaciones realizadas a la identidad de `saldo`

Ejemplo: Cuenta Corriente I

Código 4: code/concurrentChangeToBalance.clj

```
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))
```

Ejemplo: Cuenta Corriente II

```
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)
```

¿Cómo Aprovechamos Todo esto en Java? I

- Recordemos: Clojure es intepretado por la JVM
- Existe cierto grado de compatibilidad entre las API de ambos lenguajes
- En particular, si utilizamos el API de Clojure, en Java podemos tener disponibles dos nuevas clases:
  - La clase `Ref` que sirve para que una identidad se asocia a un estado (valor) en el ambiente de Clojure
  - La clase `LockingTransaction` que proporciona (entre otros) el método `runInTransaction`, cuyo paramétro es un objeto que implementa la interaz Callable (es decir, una tarea concurrente), y que la ejecutra dentro de una transacción, gracias al manejador de transacciones de Cloure

Ejemplo: Cuenta Corriente con Java I

Código 5: code/Account.java

```
1  import clojure.lang.Ref;
2  import clojure.lang.LockingTransaction;
3  import java.util.concurrent.Callable;
4
5  public class Account {
6    final private Ref saldo;
7
8    public Account(final int saldoInicial) throws Exception {
9      saldo = new Ref(saldoInicial);
10   }
11
12   public int getSaldo() { return (Integer) saldo.deref(); }
13   public void deposito(final int cantidad) throws Exception {
14     LockingTransaction.runInTransaction(new Callable<Boolean>()
           {
15       public Boolean call()  {
16         if(cantidad > 0) {
17           final int saldoActual = (Integer) saldo.deref();
18           saldo.set(saldoActual + cantidad);
```

Ejemplo: Cuenta Corriente con Java II

```
19              System.out.println("Deposito de " + cantidad + "...
                    se va a realizar");
20              return true;
21            } else throw new RuntimeException("Operacion invalida");
22          }
23        });
24      }
25      public void reintegro(final int cantidad) throws Exception {
26        LockingTransaction.runInTransaction(new Callable<Boolean>()
              {
27          public Boolean call() {
28            final int saldoActual = (Integer) saldo.deref();
29            if(cantidad > 0 && saldoActual >= cantidad) {
30              saldo.set(saldoActual - cantidad);
31              return true;
32            } else throw new RuntimeException("Operacion invalida");
33          }
34        });
35      }
36  }
```

Utilizando la cuenta corriente de forma segura I

Código 6: code/Transfer.java

```
1  import clojure.lang.LockingTransaction;
2  import java.util.concurrent.Callable;
3
4  public class Transfer {
5    public static void transfer(
6      final Account cuentaOrigen, final Account cuentaDestino,
           final int cantidad)
7      throws Exception {
8      LockingTransaction.runInTransaction(new Callable<Boolean>()
           {
9        public Boolean call() throws Exception {
10         cuentaDestino.deposito(cantidad);
11         cuentaOrigen.reintegro(cantidad);
12         return true;
13       }
14     });
15   }
16
17   public static void transferirYmostrar(
```

Utilizando la cuenta corriente de forma segura II

```
18        final Account cuentaOrigen, final Account cuentaDestino,
              final int cantidad) {
19        try {
20          transfer(cuentaOrigen, cuentaDestino, cantidad);
21        } catch(Exception ex) {
22          System.out.println("Transferencia fallida " + ex);
23        }
24        System.out.println("Saldo cuenta cuentaOrigen: " +
              cuentaOrigen.getSaldo());
25        System.out.println("Saldo cuenta cuentaDestino: " +
              cuentaDestino.getSaldo());
26      }
27      public static void main(final String[] args) throws Exception
            {
28        final Account cuenta1 = new Account(2000);
29        final Account cuenta2 = new Account(100);
30        transferirYmostrar(cuenta1, cuenta2, 500);
31        transferirYmostrar(cuenta1, cuenta2, 5000);
32      }
33    }
```

Bibliografía

Clojure and Concurrency

https://www.ibm.com/developerworks/library/wa-clojure/wa-clojure-pdf.pdf

2010

Clojure Reference

https://clojure.org/reference/reader

2019

Subramanian, V.

Programming Concurrency on the JVM

2011