STM in Clojure

Tom Van Cutsem
Multicore Programming

https://github.com/tvcutsem/stm-in-clojure
http://soft.vub.ac.be/~tvcutsem/multicore

Goal

- We have already seen Clojure's built-in support for STM via refs
- Recall:

```
(defn make-account [sum]
  (ref sum))

(defn transfer [amount from to]
  (dosync
    (alter from - amount)
    (alter to + amount)))

(def accountA (make-account 1500))
(def accountB (make-account 200))

(transfer 100 accountA accountB)
(println @accountA) ; 1400
(println @accountB) ; 300
```

Goal

- Now we will build our own STM system in Clojure to better understand its implementation

```
(defn make-account [sum]
  (mc-ref sum))

(defn transfer [amount from to]
  (mc-dosync
    (mc-alter from - amount)
    (mc-alter to + amount)))

(def accountA (make-account 1500))
(def accountB (make-account 200))

(transfer 100 accountA accountB)
(println (mc-deref accountA)) ; 1400
(println (mc-deref accountB)) ; 300
```

Almost-meta-circular implementation

- We will represent refs via atoms
- We will call such refs “mc-refs” (meta-circular refs)
- Recall: atoms support synchronous but *uncoordinated* state updates
- We will have to add the coordination through transactions ourselves
- Why “almost”? A truly meta-circular implementation would represent mc-refs using refs

Atoms: recap

- Atoms encapsulate a value that can be atomically read and set
- Safe to read/write an atom concurrently from multiple threads
- Unlike refs, two or more atoms cannot be updated in a coordinated way

```
(def x (atom 0))

@x
=> 0
(swap! x inc)
=> 1
@x
=> 1
```

```
(def y (atom {:a 0 :b 1}))

@y
=> {:a 0, :b 1}
(swap! y assoc :a 2)
=> {:a 2, :b 1}
```

MC-STM: API

- A copy of the Clojure ref API:
  - `(mc-ref val)`
  - `(mc-deref mc-ref)`
  - `(mc-ref-set mc-ref val)`
  - `(mc-alter mc-ref fun & args)`
  - `(mc-commute mc-ref fun & args)`
  - `(mc-ensure mc-ref)`
  - `(mc-dosync & exprs)`

MC-STM: overview

- Redo-log approach: transactions do not modify the “public” value of an mc-ref until they commit
- Each mc-ref has a *revision number*
- Each transaction stores its own copy of the values for read/written mc-refs. These are called the *in-transaction-values*
- Transactions also remember what refs they have written, and the revision number of each mc-ref they read or write for the first time

MC-STM: overview

- For example:

```
(def x (mc-ref 42))
(mc-dosync
  (let [y (mc-deref x)]
    (mc-ref-set x (inc y))))
```

global state

Ref	val	rev

```
>
     (def x (mc-ref 42))
 T1: (mc-dosync
 T1:  (let [y (mc-deref x)]
 T1:   (mc-ref-set x (inc y))
 T1: commit
```

MC-STM: overview

- For example:

```
(def x (mc-ref 42))
(mc-dosync
  (let [y (mc-deref x)]
    (mc-ref-set x (inc y))))
```

global state

Ref	val	rev
x	**42**	**0**

```
>     (def x (mc-ref 42))
 T1: (mc-dosync
 T1:  (let [y (mc-deref x)]
 T1:   (mc-ref-set x (inc y))
 T1: commit
```

MC-STM: overview

- For example:

```
(def x (mc-ref 42))
(mc-dosync
  (let [y (mc-deref x)]
    (mc-ref-set x (inc y))))
```

global state

Ref	val	rev
x	42	0

T1

Ref	val	rev

```
      (def x (mc-ref 42))
>T1: (mc-dosync
 T1:   (let [y (mc-deref x)]
 T1:     (mc-ref-set x (inc y))
 T1: commit
```

MC-STM: overview

- For example:

```
(def x (mc-ref 42))
(mc-dosync
  (let [y (mc-deref x)]
    (mc-ref-set x (inc y))))
```

global state

Ref	val	rev
x	42	0

T1

Ref	val	rev
x	**42**	**0**

```
     (def x (mc-ref 42))
 T1: (mc-dosync
>T1:  (let [y (mc-deref x)]
 T1:   (mc-ref-set x (inc y))
 T1: commit
```

MC-STM: overview

- For example:

```
(def x (mc-ref 42))
(mc-dosync
  (let [y (mc-deref x)]
    (mc-ref-set x (inc y))))
```

global state

Ref	val	rev
x	42	0

T1

Ref	val	rev
x	**43**	0

```
     (def x (mc-ref 42))
 T1: (mc-dosync
 T1:  (let [y (mc-deref x)]
>T1:   (mc-ref-set x (inc y))
 T1: commit
```

MC-STM: overview

- For example:

```
(def x (mc-ref 42))
(mc-dosync
  (let [y (mc-deref x)]
    (mc-ref-set x (inc y))))
```

global state

Ref	val	rev
x	**43**	**1**

T1

Ref	val	rev
x	43	0

```
     (def x (mc-ref 42))
 T1: (mc-dosync
 T1:  (let [y (mc-deref x)]
 T1:   (mc-ref-set x (inc y))
>T1: commit
```

MC-STM: overview

- For example:

```
(def x (mc-ref 42))
(mc-dosync
  (let [y (mc-deref x)]
    (mc-ref-set x (inc y))))
```

global state

Ref	val	rev
x	43	1

```
     (def x (mc-ref 42))
 T1: (mc-dosync
 T1:  (let [y (mc-deref x)]
 T1:   (mc-ref-set x (inc y))
 T1: commit
>
```

Read/write conflicts

```
(def x (mc-ref :a))
(def y (mc-ref :b))
T1: (mc-dosync
      (list (mc-deref x)
            (mc-deref y)))
T2: (mc-dosync
      (mc-ref-set x :c))
```

global state

Ref	val	rev
x	:a	0
y	:b	0

T1

Ref	val	rev

T2

Ref	val	rev

```
>
 T1: (mc-deref x)
 T2: (mc-ref-set x :c)
 T1: (mc-deref y)
 T2: commit
 T1: commit
```

Read/write conflicts

```
(def x (mc-ref :a))
(def y (mc-ref :b))
T1: (mc-dosync
      (list (mc-deref x)
            (mc-deref y)))
T2: (mc-dosync
      (mc-ref-set x :c))
```

global state

Ref	val	rev
x	:a	0
y	:b	0

T1

Ref	val	rev
x	**:a**	**0**

T2

Ref	val	rev

```
>T1: (mc-deref x)
 T2: (mc-ref-set x :c)
 T1: (mc-deref y)
 T2: commit
 T1: commit
```

Read/write conflicts

```
(def x (mc-ref :a))
(def y (mc-ref :b))
T1: (mc-dosync
      (list (mc-deref x)
            (mc-deref y)))
T2: (mc-dosync
      (mc-ref-set x :c))
```

global state

Ref	val	rev
x	:a	0
y	:b	0

T1

Ref	val	rev
x	:a	0

T2

Ref	val	rev
x	**:c**	**0**

```
 T1: (mc-deref x)
>T2: (mc-ref-set x :c)
 T1: (mc-deref y)
 T2: commit
 T1: commit
```

Read/write conflicts

```
(def x (mc-ref :a))
(def y (mc-ref :b))
T1: (mc-dosync
      (list (mc-deref x)
            (mc-deref y)))
T2: (mc-dosync
      (mc-ref-set x :c))
```

global state

Ref	val	rev
x	:a	0
y	:b	0

T1

Ref	val	rev
x	:a	0
y	**:b**	**0**

T2

Ref	val	rev
x	:c	0

```
 T1: (mc-deref x)
 T2: (mc-ref-set x :c)
>T1: (mc-deref y)
 T2: commit
 T1: commit
```

Read/write conflicts

```
(def x (mc-ref :a))
(def y (mc-ref :b))
T1: (mc-dosync
      (list (mc-deref x)
            (mc-deref y)))
T2: (mc-dosync
      (mc-ref-set x :c))
```

T1

Ref	val	rev
x	:a	0
y	:b	0

T2

Ref	val	rev
x	:c	0

global state

Ref	val	rev
x	**:c**	**2**
y	:b	0

```
 T1: (mc-deref x)
 T2: (mc-ref-set x :c)
 T1: (mc-deref y)
>T2: commit
 T1: commit
```

Read/write conflicts

```
(def x (mc-ref :a))
(def y (mc-ref :b))
T1: (mc-dosync
      (list (mc-deref x)
            (mc-deref y)))
T2: (mc-dosync
      (mc-ref-set x :c))
```

T1

Ref	val	rev
x	:a	**0**
y	:b	0

global state

Ref	val	rev
x	:c	**2**
y	:b	0

```
 T1: (mc-deref x)
 T2: (mc-ref-set x :c)
 T1: (mc-deref y)
 T2: commit
>T1: commit
```

Read/write conflicts

```
(def x (mc-ref :a))
(def y (mc-ref :b))
T1: (mc-dosync
      (list (mc-deref x)
            (mc-deref y)))
T2: (mc-dosync
      (mc-ref-set x :c))
```

global state

Ref	val	rev
x	:c	**2**
y	:b	0

T1

Ref	val	rev
x	:a	**0**
y	:b	0

```
 T1: (mc-deref x)
 T2: (mc-ref-set x :c)
 T1: (mc-deref y)
 T2: commit
>T1: commit
```

T1 will notice during validation that x has changed. It discards all its in-transaction-values and tries again.

Write/write conflicts

```
(def x (mc-ref :a))
T1: (mc-dosync
      (mc-ref-set x :b))
T2: (mc-dosync
      (mc-ref-set x :c))
```

global state

Ref	val	rev
x	:a	0

T1

Ref	val	rev

T2

Ref	val	rev

```
>
 T1: (mc-ref-set x :b)
 T2: (mc-ref-set x :c)
 T2: commit
 T1: commit
```

Write/write conflicts

```
(def x (mc-ref :a))
T1: (mc-dosync
      (mc-ref-set x :b))
T2: (mc-dosync
      (mc-ref-set x :c))
```

global state

Ref	val	rev
x	:a	0

T1

Ref	val	rev
x	**:b**	**0**

T2

Ref	val	rev

```
>T1: (mc-ref-set x :b)
 T2: (mc-ref-set x :c)
 T2: commit
 T1: commit
```

Write/write conflicts

```
(def x (mc-ref :a))
T1: (mc-dosync
      (mc-ref-set x :b))
T2: (mc-dosync
      (mc-ref-set x :c))
```

global state

Ref	val	rev
x	:a	0

T1

Ref	val	rev
x	:b	0

T2

Ref	val	rev
x	**:c**	**0**

```
 T1: (mc-ref-set x :b)
>T2: (mc-ref-set x :c)
 T2: commit
 T1: commit
```

Write/write conflicts

```
(def x (mc-ref :a))
T1: (mc-dosync
      (mc-ref-set x :b))
T2: (mc-dosync
      (mc-ref-set x :c))
```

global state

Ref	val	rev
x	**:c**	**2**

T1

Ref	val	rev
x	:b	0

T2

Ref	val	rev
x	:c	0

```
 T1: (mc-ref-set x :b)
 T2: (mc-ref-set x :c)
>T2: commit
 T1: commit
```

Write/write conflicts

```
(def x (mc-ref :a))
T1: (mc-dosync
      (mc-ref-set x :b))
T2: (mc-dosync
      (mc-ref-set x :c))
```

global state

Ref	val	rev
x	:c	**2**

T1

Ref	val	rev
x	:b	**0**

```
 T1: (mc-ref-set x :b)
 T2: (mc-ref-set x :c)
 T2: commit
>T1: commit
```

Write/write conflicts

```
(def x (mc-ref :a))
T1: (mc-dosync
      (mc-ref-set x :b))
T2: (mc-dosync
      (mc-ref-set x :c))
```

global state

Ref	val	rev
x	:c	**2**

T1

Ref	val	rev
x	:b	**0**

```
 T1: (mc-ref-set x :b)
 T2: (mc-ref-set x :c)
 T2: commit
>T1: commit
```

T1 will notice during validation that x has changed. It discards all its in-transaction-values and tries again.

Multiple readers

```
(def x (mc-ref :a))
(def y (mc-ref :b))
T1: (mc-dosync
      (mc-deref x))
T2: (mc-dosync
      (mc-deref x)
      (mc-ref-set y :c))
```

global state

Ref	val	rev
x	:a	0
y	:b	0

T1

Ref	val	rev

T2

Ref	val	rev

```
>
 T1: (mc-deref x)
 T2: (mc-deref x)
 T2: (mc-ref-set y :c)
 T2: commit
 T1: commit
```

Multiple readers

```
(def x (mc-ref :a))
(def y (mc-ref :b))
T1: (mc-dosync
      (mc-deref x))
T2: (mc-dosync
      (mc-deref x)
      (mc-ref-set y :c))
```

global state

Ref	val	rev
x	:a	0
y	:b	0

T1

Ref	val	rev
x	**:a**	**0**

T2

Ref	val	rev

```
>T1: (mc-deref x)
 T2: (mc-deref x)
 T2: (mc-ref-set y :c)
 T2: commit
 T1: commit
```

Multiple readers

```
(def x (mc-ref :a))
(def y (mc-ref :b))
T1: (mc-dosync
      (mc-deref x))
T2: (mc-dosync
      (mc-deref x)
      (mc-ref-set y :c))
```

global state

Ref	val	rev
x	:a	0
y	:b	0

T1

Ref	val	rev
x	:a	0

T2

Ref	val	rev
x	**:a**	**0**

```
 T1: (mc-deref x)
>T2: (mc-deref x)
 T2: (mc-ref-set y :c)
 T2: commit
 T1: commit
```

Multiple readers

```
(def x (mc-ref :a))
(def y (mc-ref :b))
T1: (mc-dosync
      (mc-deref x))
T2: (mc-dosync
      (mc-deref x)
      (mc-ref-set y :c)
```

global state

Ref	val	rev
x	:a	0
y	:b	0

T1

Ref	val	rev
x	:a	0

T2

Ref	val	rev
x	:a	0
y	**:c**	**0**

```
 T1: (mc-deref x)
 T2: (mc-deref x)
>T2: (mc-ref-set y :c)
 T2: commit
 T1: commit
```

Multiple readers

```
(def x (mc-ref :a))
(def y (mc-ref :b))
T1: (mc-dosync
      (mc-deref x))
T2: (mc-dosync
      (mc-deref x)
      (mc-ref-set y :c))
```

global state

Ref	val	rev
x	:a	0
y	**:c**	**2**

T1

Ref	val	rev
x	:a	0

T2

Ref	val	rev
x	:a	0
y	:c	0

```
 T1: (mc-deref x)
 T2: (mc-deref x)
 T2: (mc-ref-set y :c)
>T2: commit
 T1: commit
```

Multiple readers

```
(def x (mc-ref :a))
(def y (mc-ref :b))
T1: (mc-dosync
      (mc-deref x))
T2: (mc-dosync
      (mc-deref x)
      (mc-ref-set y :c))
```

global state

Ref	val	rev
x	:a	**0**
y	:c	2

T1

Ref	val	rev
x	:a	**0**

```
 T1: (mc-deref x)
 T2: (mc-deref x)
 T2: (mc-ref-set y :c)
 T2: commit
>T1: commit
```

Multiple readers

```
(def x (mc-ref :a))
(def y (mc-ref :b))
T1: (mc-dosync
      (mc-deref x))
T2: (mc-dosync
      (mc-deref x)
      (mc-ref-set y :c))
```

global state

Ref	val	rev
x	:a	**0**
y	:c	2

T1

Ref	val	rev
x	:a	**0**

```
 T1: (mc-deref x)
 T2: (mc-deref x)
 T2: (mc-ref-set y :c)
 T2: commit
>T1: commit
```

Revision numbers for T1's references still match, so T1 is allowed to commit. Since T1 only read x, it does not change the global state

MC-STM version 1: mc-refs

- mc-refs are represented as atoms encapsulating a map
- The map contains the ref's publicly visible value and its revision number

```
(defn mc-ref [val]
  (atom {:value val
         :revision 0}))
```

- Each time a transaction commits a new value, the revision number will be updated

MC-STM version 1: the current transaction

- Thread-local Var holds the current transaction executed by this thread
- If the thread does not execute a transaction, set to nil

```
(def *current-transaction* nil)
```

MC-STM version 1: public API

- refs can be read but not written to outside of a transaction

```
(defn mc-deref [ref]
  (if (nil? *current-transaction*)
      ; reading a ref outside of a transaction
      (:value @ref)
      ; reading a ref inside a transaction
      (tx-read *current-transaction* ref)))

(defn mc-ref-set [ref newval]
  (if (nil? *current-transaction*)
      ; writing a ref outside of a transaction
      (throw (IllegalStateException. "can't set mc-ref outside transaction"))
      ; writing a ref inside a transaction
      (tx-write *current-transaction* ref newval)))

(defn mc-alter [ref fun & args]
  (mc-ref-set ref (apply fun (mc-deref ref) args)))
```

MC-STM version 1: public API

- Naive but correct implementations of commute and ensure, for now
  - both implemented in terms of altering an mc-ref
  - commutes and ensures will cause needless conflicts

```
(defn mc-commute [ref fun & args]
  (apply mc-alter ref fun args))

(defn mc-ensure [ref]
  (mc-alter ref identity))
```

MC-STM version 1: transactions

- Each transaction has a unique ID
- Also stores the “in-transaction-values” of all refs it reads/writes
- Technically, in-tx-values, written-refs and last-seen-rev don’t need to be atoms (Vars are sufficient), as they are thread-local

```
(def NEXT_TRANSACTION_ID (atom 0))

(defn make-transaction
  "create and return a new transaction data structure"
  []
  { :id (swap! NEXT_TRANSACTION_ID inc),
    :in-tx-values (atom {}), ; map: ref -> any value
    :written-refs (atom #{}), ; set of refs
    :last-seen-rev (atom {}) }) ; map: ref -> revision id
```

MC-STM version 1: reading a ref

- If the ref was read or written before, returns its in-transaction-value
- If the ref is read for the first time, cache its value and remember the first revision read

```
(defn tx-read
  "read the value of ref inside transaction tx"
  [tx ref]
  (let [in-tx-values (:in-tx-values tx)]
    (if (contains? @in-tx-values ref)
      (@in-tx-values ref) ; return the in-tx-value
      ; important: read both ref's value and revision atomically
      (let [{in-tx-value :value
             read-revision :revision} @ref]
        (swap! in-tx-values assoc ref in-tx-value)
        (swap! (:last-seen-rev tx) assoc ref read-revision)
        in-tx-value))))
```

MC-STM version 1: writing a ref

- Update the in-transaction-value of the ref and remember it was “written”
- If the ref was not read or written to before, remember its current revision

```
(defn tx-write
  "write val to ref inside transaction tx"
  [tx ref val]
  (swap! (:in-tx-values tx) assoc ref val)
  (swap! (:written-refs tx) conj ref)
  (if (not (contains? @(:last-seen-rev tx) ref))
    (swap! (:last-seen-rev tx) assoc ref (:revision @ref)))
  val)
```

MC-STM version 1: committing a transaction

- Committing a transaction consists of two parts:
  - Validation: check revision numbers to see if any read or written refs have since been modified by another committed transaction
  - If not, make the in-transaction-value of all written-to refs public *and* update their revision number
- These two steps need to happen atomically: requires locks, since multiple atoms cannot be updated atomically
- In this version: a single lock guards *all* mc-refs. Only one transaction can commit at a time.

```
(def COMMIT_LOCK (new java.lang.Object))
```

MC-STM version 1: committing a transaction

- If validation fails, it is up to the caller of tx-commit to retry the transaction

```
(defn tx-commit
  "returns a boolean indicating whether tx committed successfully"
  [tx]
  (let [validate
        (fn [refs]
          (every? (fn [ref]
                    (= (:revision @ref)
                       (@(:last-seen-rev tx) ref))) refs))]

  (locking COMMIT_LOCK
    (let [in-tx-values @(:in-tx-values tx)
          success (validate (keys in-tx-values))]
      (if success
        ; if validation OK, make in-tx-value of all written refs public
        (doseq [ref @(:written-refs tx)]
          (swap! ref assoc
            :value (in-tx-values ref)
            :revision (:id tx) )))
      success))))
```

MC-STM version 1: running a transaction

- The transaction body is run with *current-transaction* thread-locally bound to the transaction
- If the transaction commits successfully, return its result
- If not, the current transaction (including its in-transaction-values) is discarded and the entire process is *retried* with a fresh transaction

```
(defn tx-run
  "runs zero-argument fun as the body of transaction tx"
  [tx fun]
  (let [result (binding [*current-transaction* tx] (fun))]
    (if (tx-commit tx)
        result
        (recur (make-transaction) fun))))
```

MC-STM version 1: running a transaction

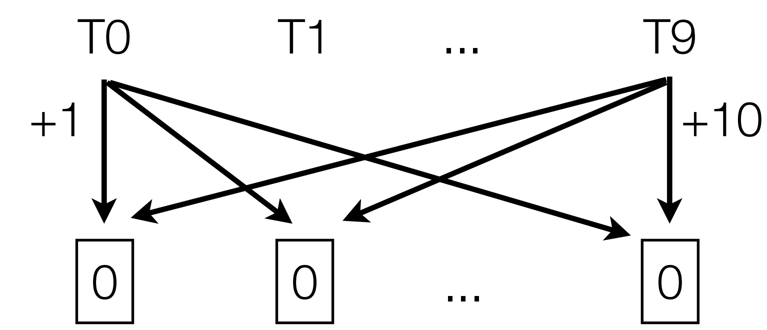
- mc-dosync is a *macro* that simply wraps its arguments in a function
- If a transaction is already running, this indicates a nested mc-dosync block. Nested blocks implicitly become part of their "parent" transaction.

```
(defmacro mc-dosync [& exps]
  `(mc-sync (fn [] ~@exps)))

(defn mc-sync [fun]
  (if (nil? *current-transaction*)
      (tx-run (make-transaction) fun)
      (fun)))
```

MC-STM version 1: test

- Test from clojure.org/concurrent_programming:

```
(defn test-stm [nitems nthreads niters]
  (let [refs  (map mc-ref (replicate nitems 0))
        pool  (Executors/newFixedThreadPool nthreads)
        tasks (map (fn [t]
                     (fn []
                       (dotimes [n niters]
                         (mc-dosync
                           (doseq [r refs]
                             (mc-alter r + 1 t))))))
                   (range nthreads))]
    (doseq [future (.invokeAll pool tasks)]
      (.get future))
    (.shutdown pool)
    (map mc-deref refs)))

; threads increment each ref by 550000 in total
; 550000 = (* (+ 1 2 3 4 5 6 7 8 9 10) 10000)
(def res (time (test-stm 10 10 10000)))
"Elapsed time: 8105.424 msecs" ; built-in stm: "Elapsed time: 2731.11 msecs"
=> (550000 550000 550000 550000 550000 550000 550000 550000 550000 550000)
```



MC-STM version 1: limitations

- Internal consistency is not guaranteed: a transaction may read a value for a ref before another transaction T committed, and read a value for another ref after T committed, leading to potentially mutually inconsistent ref values

- Naive implementations of commute and ensure

- A single global commit-lock for all transactions (= severe bottleneck, but makes it easy to validate and commit)

MC-STM version 2: internal consistency

- In previous version, internal consistency is not guaranteed: transactions may read reference states *before* another transaction committed, then read other reference states *after* a transaction committed.

- Ref values may become mutually inconsistent

- This may violate invariants in code, leading to bugs, exceptions or infinite loops

Recall: internal consistency & zombies

- This code sometimes crashes with a Divide by zero exception:

```
; invariant: x = 2y
(def x (mc-ref 4))
(def y (mc-ref 2))

(def T1 (Thread. (fn []
                   (mc-dosync
                     (mc-alter x (fn [_] 8))
                     (mc-alter y (fn [_] 4))))))
(def T2 (Thread. (fn []
                   (mc-dosync
                     (/ 1 (- (mc-deref x) (mc-deref y)))))))
(.start T1) (.start T2)
(.join T1) (.join T2)
```

Recall: internal consistency & zombies

- Why?

global state

Ref	val	rev
x	4	0
y	2	0

T1

Ref	val	rev

T2

Ref	val	rev

```
>
 T1: (mc-alter x (fn [_] 8))
 T2: x' = (mc-deref x)
 T1: (mc-alter y (fn [_] 4))
 T1: commit
 T2: y' = (mc-deref y)
 T2: (/ 1 (- x' y'))
```

Recall: internal consistency & zombies

- Why?

global state

Ref	val	rev
x	4	0
y	2	0

T1

Ref	val	rev
x	**8**	**0**

T2

Ref	val	rev

```
>T1: (mc-alter x (fn [_] 8))
 T2: x’ = (mc-deref x)
 T1: (mc-alter y (fn [_] 4))
 T1: commit
 T2: y’ = (mc-deref y)
 T2: (/ 1 (- x’ y’))
```

Recall: internal consistency & zombies

- Why?

global state

Ref	val	rev
x	4	0
y	2	0

T1

Ref	val	rev
x	8	0

T2

Ref	val	rev
x	**4**	**0**

```
 T1: (mc-alter x (fn [_] 8))
>T2: x’ = (mc-deref x)
 T1: (mc-alter y (fn [_] 4))
 T1: commit
 T2: y’ = (mc-deref y)
 T2: (/ 1 (- x’ y’))
```

Recall: internal consistency & zombies

- Why?

T1

Ref	val	rev
x	8	0
y	**4**	**0**

T2

Ref	val	rev
x	4	0

global state

Ref	val	rev
x	4	0
y	2	0

```
 T1: (mc-alter x (fn [_] 8))
 T2: x’ = (mc-deref x)
>T1: (mc-alter y (fn [_] 4))
 T1: commit
 T2: y’ = (mc-deref y)
 T2: (/ 1 (- x’ y’))
```

Recall: internal consistency & zombies

- Why?

global state

Ref	val	rev
x	**8**	**1**
y	**4**	**1**

T1

Ref	val	rev
x	8	0
y	4	0

T2

Ref	val	rev
x	4	0

```
 T1: (mc-alter x (fn [_] 8))
 T2: x’ = (mc-deref x)
 T1: (mc-alter y (fn [_] 4))
>T1: commit
 T2: y’ = (mc-deref y)
 T2: (/ 1 (- x’ y’))
```

Recall: internal consistency & zombies

- Why?

global state

Ref	val	rev
x	8	1
y	4	1

T1

Ref	val	rev

T2

Ref	val	rev
x	4	0
y	**4**	**1**

```
 T1: (mc-alter x (fn [_] 8))
 T2: x’ = (mc-deref x)
 T1: (mc-alter y (fn [_] 4))
 T1: commit
>T2: y’ = (mc-deref y)
 T2: (/ 1 (- x’ y’))
```

Recall: internal consistency & zombies

- Why?

global state

Ref	val	rev
x	8	1
y	4	1

T1

Ref	val	rev

T2

Ref	val	rev
x	4	0
y	**4**	**1**

```
 T1: (mc-alter x (fn [_] 8))
 T2: x' = (mc-deref x)
 T1: (mc-alter y (fn [_] 4))
 T1: commit
>T2: y' = (mc-deref y)
 T2: (/ 1 (- x' y'))
```

T2 is now a zombie: it will never pass the validation step

Recall: internal consistency & zombies

- Why?

global state

Ref	val	rev
x	8	1
y	4	1

T1

Ref	val	rev

T2

Ref	val	rev
x	**4**	0
y	**4**	1

```
 T1: (mc-alter x (fn [_] 8))
 T2: x' = (mc-deref x)
 T1: (mc-alter y (fn [_] 4))
 T1: commit
 T2: y' = (mc-deref y)
>T2: (/ 1 (- x' y'))
```

Division by zero

MC-STM version 2: internal consistency

- We will solve this by using multiversion concurrency control (MVCC), like Clojure itself
- All reads of Refs will see a consistent snapshot of the global “Ref world” as of the starting point of the transaction (its **read point**).
- All changes made to Refs during a transaction will appear to occur at a single point in the global “Ref world” timeline (its **write point**).
- When the transaction commits, no changes will have been made by any other transactions to any Refs that have been ref-set/altered/ensured by this transaction (otherwise, it is retried)

MC-STM: version 2, internal consistency

T1 `Read-point: 0`

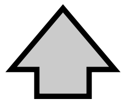
Ref	val

T2 `Read-point: 0`

Ref	val

global state

Ref	v0	
x	4	
y	2	



`Write-point: 0`

```
>
 T1: (mc-alter x (fn [_] 8))
 T2: x’ = (mc-deref x)
 T1: (mc-alter y (fn [_] 4))
 T1: commit
 T2: y’ = (mc-deref y)
 T2: (/ 1 (- x’ y’))
```

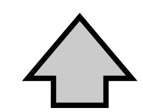
MC-STM: version 2, internal consistency

T1 `Read-point: 0`

Ref	val
x	**8**

global state

Ref	v0	
x	4	
y	2	



`Write-point: 0`

T2 `Read-point: 0`

Ref	val

```
>T1: (mc-alter x (fn [_] 8))
 T2: x' = (mc-deref x)
 T1: (mc-alter y (fn [_] 4))
 T1: commit
 T2: y' = (mc-deref y)
 T2: (/ 1 (- x' y'))
```

MC-STM: version 2, internal consistency

global state

Ref	v0	
x	4	
y	2	

`Write-point: 0`

T1 `Read-point: 0`

Ref	val
x	8

T2 `Read-point: 0`

Ref	val
x	**4**

```
 T1: (mc-alter x (fn [_] 8))
>T2: x' = (mc-deref x)
 T1: (mc-alter y (fn [_] 4))
 T1: commit
 T2: y' = (mc-deref y)
 T2: (/ 1 (- x' y'))
```

MC-STM: version 2, internal consistency

T1 `Read-point: 0`

Ref	val
x	8
y	**4**

global state

Ref	v0	
x	4	
y	2	

`Write-point: 0`

T2 `Read-point: 0`

Ref	val
x	4

```
 T1: (mc-alter x (fn [_] 8))
 T2: x' = (mc-deref x)
>T1: (mc-alter y (fn [_] 4))
 T1: commit
 T2: y' = (mc-deref y)
 T2: (/ 1 (- x' y'))
```

MC-STM: version 2, internal consistency

global state

Ref	v0	**v1**
x	4	**8**
y	2	**4**

`Write-point: 1`

T1 `Read-point: 0`

Ref	val
x	8
y	4

T2 `Read-point: 0`

Ref	val
x	4

```
 T1: (mc-alter x (fn [_] 8))
 T2: x’ = (mc-deref x)
 T1: (mc-alter y (fn [_] 4))
>T1: commit
 T2: y’ = (mc-deref y)
 T2: (/ 1 (- x’ y’))
```

MC-STM: version 2, internal consistency

global state

Ref	v0	v1
x	4	8
y	2	4

`Write-point: 1`

T1 `Read-point: 0`

Ref	val
x	8
y	4

T2 `Read-point: 0`

Ref	val
x	4
y	**2**

```
 T1: (mc-alter x (fn [_] 8))
 T2: x' = (mc-deref x)
 T1: (mc-alter y (fn [_] 4))
 T1: commit
>T2: y' = (mc-deref y)
 T2: (/ 1 (- x' y'))
```

MC-STM: version 2, internal consistency

global state

Ref	v0	v1
x	4	8
y	2	4

`Write-point: 1`

T1 `Read-point: 0`

Ref	val
x	8
y	4

T2 `Read-point: 0`

Ref	val
x	4
y	**2**

Since T2's read-point is 0, it reads v0 of the global state

```
T1: (mc-alter x (fn [_] 8))
T2: x' = (mc-deref x)
T1: (mc-alter y (fn [_] 4))
T1: commit
>T2: y' = (mc-deref y)
T2: (/ 1 (- x' y'))
```

MC-STM: version 2, internal consistency

global state

Ref	v0	v1
x	4	8
y	2	4

`Write-point: 1`

T1 `Read-point: 0`

Ref	val
x	8
y	4

T2 `Read-point: 0`

Ref	val
x	4
y	2

```
T1: (mc-alter x (fn [_] 8))
T2: x’ = (mc-deref x)
T1: (mc-alter y (fn [_] 4))
T1: commit
T2: y’ = (mc-deref y)
>T2: (/ 1 (- x’ y’))
```

Now calculates 1/2 as expected

MC-STM version 2: mc-refs

- mc-refs are now represented as a list of {:value, :write-point} pairs, potentially followed by trailing nil values. These pairs represent successive values assigned to the mc-ref, also called the *history chain* of the mc-ref.

- Pairs are ordered latest :write-point first, oldest :write-point last

- Only the last MAX_HISTORY assigned values are stored in the history chain

```
(def MAX_HISTORY 10)
(def DEFAULT_HISTORY_TAIL (repeat (dec MAX_HISTORY) nil))

(defn mc-ref [val]
  (atom (cons {:value val :write-point @GLOBAL_WRITE_POINT}
              DEFAULT_HISTORY_TAIL)))

(def most-recent first)
```

MC-STM version 2: the current transaction

- Unchanged from v1
- Thread-local Var holds the current transaction executed by this thread
- If the thread does not execute a transaction, set to nil

```
(def *current-transaction* nil)
```

MC-STM version 2: public API

- Unchanged from v1, except how to access the most recent mc-ref value:

```
(defn mc-deref [ref]
  (if (nil? *current-transaction*)
      ; reading a ref outside of a transaction
      (:value (most-recent @ref))
      ; reading a ref inside a transaction
      (tx-read *current-transaction* ref)))

(defn mc-ref-set [ref newval]
  (if (nil? *current-transaction*)
      ; writing a ref outside of a transaction
      (throw (IllegalStateException. "can't set mc-ref outside transaction"))
      ; writing a ref inside a transaction
      (tx-write *current-transaction* ref newval)))

(defn mc-alter [ref fun & args]
  (mc-ref-set ref (apply fun (mc-deref ref) args)))
```

MC-STM version 2: public API

- Unchanged from v1
- Naive but correct implementations of commute and ensure, for now
  - both implemented in terms of altering an mc-ref
  - commutes and ensures will cause needless conflicts

```
(defn mc-commute [ref fun & args]
  (apply mc-alter ref fun args))

(defn mc-ensure [ref]
  (mc-alter ref identity))
```

MC-STM version 2: transactions

- Transactions no longer have a unique ID but record their *read point* as the value of the global *write point* when they start
- Still stores the “in-transaction-values” of all refs it reads/writes
- No need for :last-seen-rev map anymore

```
(def GLOBAL_WRITE_POINT (atom 0))

(defn make-transaction
  "create and return a new transaction data structure"
  []
  { :read-point @GLOBAL_WRITE_POINT,
    :in-tx-values (atom {}), ; map: ref -> any value
    :written-refs (atom #{}) }) ; set of refs
```

MC-STM version 2: reading a ref

- If the ref was read or written before, returns its in-transaction-value
- If the ref is read for the first time, only read a value whose write-point <= the transaction’s read-point. If such a value was not found, abort and retry.

```
(defn tx-read
  "read the value of ref inside transaction tx"
  [tx mc-ref]
  (let [in-tx-values (:in-tx-values tx)]
    (if (contains? @in-tx-values mc-ref)
      (@in-tx-values mc-ref) ; return the in-tx-value
      ; search the history chain for entry with write-point <= tx's read-point
      (let [ref-entry (find-entry-before-or-on @mc-ref (:read-point tx))]
        (if (not ref-entry)
          ; if such an entry was not found, retry
          (tx-retry))
        (let [in-tx-value (:value ref-entry)]
          (swap! in-tx-values assoc mc-ref in-tx-value) ; cache the value
          in-tx-value)))))) ; save and return the ref's value
```

MC-STM version 2: reading a ref

- Auxiliary function to scan the history list of an mc-ref

```
(defn find-entry-before-or-on
  "returns an entry in history-chain whose write-pt <= read-pt,
   or nil if no such entry exists"
  [history-chain read-pt]
  (some (fn [pair]
          (if (and pair (<= (:write-point pair) read-pt))
            pair)) history-chain))
```

MC-STM version 2: writing a ref

- Update the in-transaction-value of the ref and remember it was "written" to
- No need to remember the revision of the ref anymore

```
(defn tx-write
  "write val to ref inside transaction tx"
  [tx mc-ref val]
  (swap! (:in-tx-values tx) assoc mc-ref val)
  (swap! (:written-refs tx) conj mc-ref)
  val)
```

MC-STM version 2: committing a transaction

- Committing a transaction still consists of two parts:
  - Validation: for each written ref, check if the ref has since been modified by another committed transaction
  - If not, store the in-transaction-value of all written-to refs in the history chain of the refs under a new write-point. *Then* update the global write-point such that new transactions can see the new values.
- These two steps need to happen atomically: requires locks, since multiple atoms cannot be updated atomically
- In this version: still a single lock that guards *all* mc-refs. Only one transaction can commit at a time.

```
(def COMMIT_LOCK (new java.lang.Object))
```

MC-STM version 2: committing a transaction

- Note: transactions that only read refs will always commit, and don't need to acquire the lock

```
(defn tx-commit
  "returns normally if tx committed successfully, throws RetryEx otherwise"
  [tx]
  (let [written-refs @(:written-refs tx)]
    (when (not (empty? written-refs))
      (locking COMMIT_LOCK
        (doseq [written-ref written-refs]
          (if (> (:write-point (most-recent @written-ref))
                 (:read-point tx))
            (tx-retry)))

        (let [in-tx-values @(:in-tx-values tx)
              new-write-point (inc @GLOBAL_WRITE_POINT)]
          (doseq [ref written-refs]
            (swap! ref (fn [history-chain]
                         (cons {:value (in-tx-values ref)
                                :write-point new-write-point} (butlast history-chain)))))
          (swap! GLOBAL_WRITE_POINT inc)))))) ; make the new write-point public
```

MC-STM version 2: retrying a transaction

- Retrying causes a special exception to be thrown
- The exception is a java.lang.Error, not a java.lang.Exception, so applications will not normally catch this

```
(defn tx-retry []
  (throw (new stm.RetryEx)))

; in a separate file stm/RetryEx.clj
(ns stm.RetryEx
  (:gen-class :extends java.lang.Error))
```

MC-STM version 2: running a transaction

- To catch RetryEx, must run the function in a try-block
- Cannot perform tail-recursion with recur from within a catch-clause, so need to exit try-block and test the value before calling recur:

```
(defn tx-run
  "runs zero-argument fun as the body of transaction tx."
  [tx fun]
  (let [res (binding [*current-transaction* tx]
              (try
                (let [result (fun)]
                  (tx-commit tx)
                  ; commit succeeded, return result
                  {:result result}) ; wrap result, as it may be nil
                (catch stm.RetryEx e
                  nil)))]
    (if res
      (:result res)
      (recur (make-transaction) fun)))) ; read or commit failed, retry with fresh tx
```

MC-STM version 2: running a transaction

- mc-dosync and mc-sync unchanged from v1

```
(defmacro mc-dosync [& exps]
  `(mc-sync (fn [] ~@exps)))

(defn mc-sync [fun]
  (if (nil? *current-transaction*)
      (tx-run (make-transaction) fun)
      (fun))) ; nested blocks implicitly run in parent transaction
```

MC-STM: version 2 limitations

- Naive implementations of commute and ensure
- A single global commit-lock for all transactions (= severe bottleneck, but makes it easy to validate and commit)

MC-STM version 3: support for commute/ensure

- Up to now, commute and ensure resulted in needless conflicts, as both were implemented in terms of mc-alter:

```
(defn mc-commute [ref fun & args]
  (apply mc-alter ref fun args))

(defn mc-ensure [ref]
  (mc-alter ref identity))
```

- Ensure needed to prevent write skew

Recall: write skew

```
(def cats (mc-ref 1))
(def dogs (mc-ref 1))
(def john (Thread. (fn []
  (mc-dosync
    (if (< (+ (mc-deref cats) (mc-deref dogs)) 3)
        (mc-alter cats inc))))))
(def mary (Thread. (fn []
  (mc-dosync
    (if (< (+ (mc-deref cats) (mc-deref dogs)) 3)
        (mc-alter dogs inc))))))
(doseq [p [john mary]] (.start p))
(doseq [p [john mary]] (.join p))
(if (> (+ (mc-deref cats) (mc-deref dogs)) 3)
  (println "write skew detected"))  ; can occur!
```

Recall: write skew

```
(def cats (mc-ref 1))
(def dogs (mc-ref 1))
(def john (Thread. (fn []
  (mc-dosync
    (mc-ensure dogs)
    (if (< (+ (mc-deref cats) (mc-deref dogs)) 3)
        (mc-alter cats inc))))))
(def mary (Thread. (fn []
  (mc-dosync
    (mc-ensure cats)
    (if (< (+ (mc-deref cats) (mc-deref dogs)) 3)
        (mc-alter dogs inc))))))
(doseq [p [john mary]] (.start p))
(doseq [p [john mary]] (.join p))
(if (> (+ (mc-deref cats) (mc-deref dogs)) 3)
  (println "write skew detected"))  ; cannot occur!
```

MC-STM version 3: public API

- Like alter, commute and ensure can only be called inside a transaction:

```
(defn mc-commute [ref fun & args]
    (if (nil? *current-transaction*)
      (throw (IllegalStateException. "can't commute mc-ref outside transaction"))
      (tx-commute *current-transaction* ref fun args)))

(defn mc-ensure [ref]
    (if (nil? *current-transaction*)
      (throw (IllegalStateException. "can't ensure mc-ref outside transaction"))
      (tx-ensure *current-transaction* ref)))
```

MC-STM version 3: transactions

- Transactions now additionally store:
  - A map containing all commutative updates
  - A set of ensure'd refs

```
(defn make-transaction
  "create and return a new transaction data structure"
  []
  { :read-point @GLOBAL_WRITE_POINT,
    :in-tx-values (atom {}), ; map: ref -> any value
    :written-refs (atom #{}), ; set of written-to refs
    :commutes (atom {}), ; map: ref -> seq of commute-fns
    :ensures (atom #{}) }) ; set of ensure-d refs
```

MC-STM version 3: ensure

- To ensure a ref, simply mark it as “ensured” by adding it to the ensures set
- When the transaction commits, it will check to see if these refs were not changed

```
(defn tx-ensure
  "ensure ref inside transaction tx"
  [tx ref]
  ; mark this ref as being ensure-d
  (swap! (:ensures tx) conj ref))
```

MC-STM version 3: commute

- When a ref is commuted, its function is applied to either the in-transaction-value or the most recent ref value

- Add function and arguments to the list of commutative updates for the ref

```
(defn tx-commute
  "commute ref inside transaction tx"
  [tx ref fun args]
  (let [in-tx-values @(:in-tx-values tx)
        res (apply fun (if (contains? in-tx-values ref)
                           (in-tx-values ref)
                           (:value (most-recent @ref))) args)]
    ; retain the result as an in-transaction-value
    (swap! (:in-tx-values tx) assoc ref res)
    ; mark the ref as being commuted,
    ; storing fun and args because it will be re-executed at commit time
    (swap! (:commutes tx) (fn [commutes]
                             (assoc commutes ref
                                (cons (fn [val] (apply fun val args))
                                      (commutes ref)))))
    res))
```

MC-STM version 3: writing a ref

- Commuted refs cannot later be altered by the same transaction

```
(defn tx-write
  "write val to ref inside transaction tx"
  [tx ref val]
  ; can't set a ref after it has already been commuted
  (if (contains? @(:commutes tx) ref)
    (throw (IllegalStateException. "can't set after commute on " ref)))
  (swap! (:in-tx-values tx) assoc ref val)
  (swap! (:written-refs tx) conj ref)
  val)
```

MC-STM version 3: committing a transaction

- Committing a transaction consists of three parts:
  - 1: For each written ref and ensured ref, check if the ref was not modified by other transactions in the mean time
  - 2: For each commuted ref, re-apply all commutes based on the most recent value
  - 3: Make the changes made to each written and commuted ref public

MC-STM version 3: committing a transaction

- 1: For each written ref and ensured ref, check if the ref was not modified by other transactions in the mean time

```
(defn tx-commit
  "returns normally if tx committed successfully, throws RetryEx otherwise"
  [tx]
  (let [written-refs @(:written-refs tx)
        ensured-refs @(:ensures tx)
        commuted-refs @(:commutes tx)]
    (when (not-every? empty? [written-refs ensured-refs commuted-refs])
      (locking COMMIT_LOCK
        ; validate both written-refs and ensured-refs
        ; Note: no need to validate commuted-refs
        (doseq [ref (union written-refs ensured-refs)]
          (if (> (:write-point (most-recent @ref))
                 (:read-point tx))
            (tx-retry)))
        ; part 2 ...
```

MC-STM version 3: committing a transaction

- 2: For each commuted ref, re-apply all commutes based on the most recent value

```
(defn tx-commit
  "returns normally if tx committed successfully, throws RetryEx otherwise"
  [tx]
  (let [written-refs @(:written-refs tx)
        ensured-refs @(:ensures tx)
        commuted-refs @(:commutes tx)]
    (when (not-every? empty? [written-refs ensured-refs commuted-refs])
      (locking COMMIT_LOCK
        ; ... part 1

        ; if validation OK, re-apply all commutes based on its most recent value
        (doseq [[commuted-ref commute-fns] commuted-refs]
            (swap! (:in-tx-values tx) assoc commuted-ref
              ; apply each commute-fn to the result of the previous commute-fn,
              ; starting with the most recent value
              ((reduce comp commute-fns) (:value (most-recent @commuted-ref)))))
              ; ... part 3
```

MC-STM version 3: committing a transaction

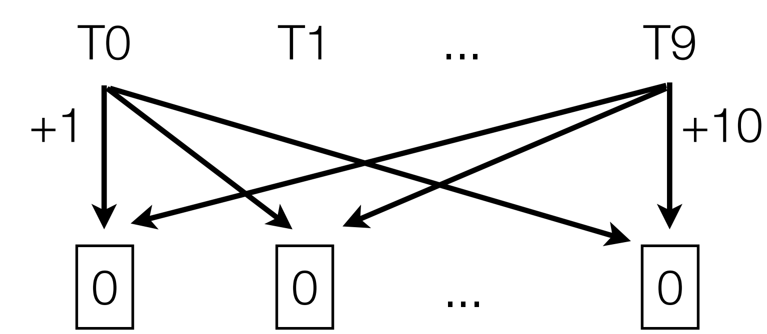
- 3: Make the changes made to each written and commuted ref public (almost identical to v2)

```
(defn tx-commit
  "returns normally if tx committed successfully, throws RetryEx otherwise"
  [tx]
  (let [written-refs @(:written-refs tx)
        ensured-refs @(:ensures tx)
        commuted-refs @(:commutes tx)]
    (when (not-every? empty? [written-refs ensured-refs commuted-refs])
      (locking COMMIT_LOCK
        ; ... part 1 and 2

        (let [in-tx-values @(:in-tx-values tx)
              new-write-point (inc @GLOBAL_WRITE_POINT)]
          (doseq [ref (union written-refs (keys commuted-refs))]
            (swap! ref (fn [history-chain]
                         (cons {:value (in-tx-values ref)
                                :write-point new-write-point} (butlast history-chain)))))
          (swap! GLOBAL_WRITE_POINT inc)))))) ; make the new write-point public
```

MC-STM version 3: test

- Test from clojure.org/concurrent_programming, now using commute:

```
(defn test-stm [nitems nthreads niters]
  (let [refs  (map mc-ref (replicate nitems 0))
        pool  (Executors/newFixedThreadPool nthreads)
        tasks (map (fn [t]
                     (fn []
                       (dotimes [n niters]
                         (mc-dosync
                           (doseq [r refs]
                             (mc-commute r + 1 t))))))
                   (range nthreads))]
    (doseq [future (.invokeAll pool tasks)]
      (.get future))
    (.shutdown pool)
    (map mc-deref refs)))

; threads increment each ref by 550000 in total
; 550000 = (* (+ 1 2 3 4 5 6 7 8 9 10) 10000)
(def res (test-stm 10 10 10000))
=> (550000 550000 550000 550000 550000 550000 550000 550000 550000 550000)
; using mc-alter: 112677 retries, using mc-commute: 0 retries
```



MC-STM: version 3 limitations

- A single global commit-lock for all transactions (= severe bottleneck, but makes it easy to validate and commit)

    - Transactions that modify disjoint sets of references can’t commit in parallel

MC-STM version 4: fine-grained locking

- Instead of a single global commit lock, use fine-grained locking
- One lock per mc-ref (we will reuse internal Java object locks)
- Transactions that alter/commute/ensure disjoint sets of mc-refs can commit in parallel
- To prevent deadlock, transactions must all acquire mc-ref locks in the same order
  - Add a unique ID to each mc-ref
  - mc-refs are sorted according to unique ID before being locked

MC-STM version 4: fine-grained locking

- Each mc-ref is guarded by a lock. Lock is only held for very short periods of time, *never* for the entire duration of a transaction.
  - Lock held for “writing” by a committing transaction when it publishes a new value
  - Lock held for “reading” by a transaction the first time it reads the value of an mc-ref
    - To ensure that a new transaction, started after the write-point was increased, waits for a committing transaction that is still writing to that write-point
- Note: could use a multiple reader/single writer lock (didn’t do this because the overhead of using such locks from Clojure was prohibitive)

MC-STM version 4: fine-grained locking

- As before, when a transaction is created it saves the current global write point as its read point

```
(defn make-transaction
  "create and return a new transaction data structure"
  []
  { :read-point @GLOBAL_WRITE_POINT,
    :in-tx-values (atom {}), ; map: ref -> any value
    :written-refs (atom #{}), ; set of written-to refs
    :commutes (atom {}), ; map: ref -> seq of commute-fns
    :ensures (atom #{}) }) ; set of ensure-d refs
```

MC-STM version 4: mc-refs

- mc-ref is now a map storing both the history list, a unique ID and a lock
- We will use built-in Java locks, so the lock is just a fresh Java object

```
(def REF_ID (atom 0))

(defn mc-ref [val]
  {:id (swap! REF_ID inc)
   :lock (new Object)
   :history-list (atom (cons {:value val
                              :write-point @GLOBAL_WRITE_POINT}
                            DEFAULT_HISTORY_TAIL))})
```

MC-STM version 4: transaction commit

- On commit, a transaction first acquires the lock for all mc-refs it altered, commuted or ensured, in sorted order:

```
(defn tx-commit
  "returns normally if tx committed successfully, throws RetryEx otherwise"
  [tx]
  (let [written-refs @(:written-refs tx)
        ensured-refs @(:ensures tx)
        commuted-refs @(:commutes tx)]
    (when (not-every? empty? [written-refs ensured-refs commuted-refs])
      (with-ref-locks-do (sort-by :id <
                           (union written-refs ensured-refs (keys commuted-refs)))
        (fn []
         ; ...
```

MC-STM version 4: transaction commit

- The transaction can make the new write-point public even before it writes the new mc-ref values, as it still holds the lock. Other transactions will not be able to access these values yet (note: reads outside of a transaction will!)

```
; ... (while holding locks)
(let [in-tx-values @(:in-tx-values tx)
      new-write-point (swap! GLOBAL_WRITE_POINT inc)]
  ; make in-tx-value of all written-to or commuted refs public
  (doseq [ref (union written-refs (keys commuted-refs))]
    (swap! (:history-list ref)
      (fn [prev-history-list]
        ; add a new entry to the front of the history list and remove the eldest
        (cons {:value (in-tx-values ref)
               :write-point new-write-point} (butlast prev-history-list)))))))))
```

MC-STM version 4: transaction commit

- Auxiliary function to acquire all mc-refs' locks

```
(defn with-ref-locks-do
  "acquires the lock on all refs, then executes fun"
  [refs fun]
  (if (empty? refs)
    (fun)
    (locking (:lock (first refs))
      (with-ref-locks-do (next refs) fun))))
```

MC-STM version 4: transaction read

- When a transaction first reads an mc-ref's value, it acquires the lock to ensure it is not reading from a write-point still being committed

```
(defn tx-read
  "read the value of ref inside transaction tx"
  [tx mc-ref]
  (let [in-tx-values (:in-tx-values tx)]
    (if (contains? @in-tx-values mc-ref)
      (@in-tx-values mc-ref) ; return the in-tx-value
      ; search the history chain for entry with write-point <= tx's read-point
      (let [ref-entry
            ; acquire read-lock to ensure ref is not modified by a committing tx
            (locking (:lock mc-ref)
              (find-entry-before-or-on
                @(:history-list mc-ref) (:read-point tx)))]
        (if (not ref-entry)
          ; if such an entry was not found, retry
          (tx-retry))
        (let [in-tx-value (:value ref-entry)]
          (swap! in-tx-values assoc mc-ref in-tx-value) ; cache the value
          in-tx-value)))))) ; save and return the ref's value
```

MC-STM version 4: lock on read really necessary?

- Is it really necessary to acquire a lock when reading? Can't we just increment the write-point after having updated all mc-refs as in version 3?

- Unfortunately, no: because of fine-grained locking, transactions T1 and T2 that modify disjoint sets of mc-refs can commit in parallel. Assume T1 and T2 are committing, T1 has write-point w and T2 has write-point w+1

  - Say T2 finishes committing first. It needs to increment the write-point to make its changes public, but it can't because incrementing the write-point would also make T1's changes public, and T1 is still committing.

  - By requiring acquisition of a lock when reading a ref, we allow transactions to increment the public write-point even before all other transactions that are still writing to it (or even to an earlier write-point) have committed.

MC-STM version 4: fine-grained locking

- Example of why locking on read is required:

T1 `Read-point: 0`

Ref	val

global state

Ref	v0
x	1
y	1
z	1

`Global write-point: 0`

T2 `Read-point: 0`

Ref	val

```
>
 T1: (mc-alter x inc)
 T2: (mc-alter z inc)
 T2: starts to commit
 T1: (mc-alter y inc)
 T1: starts to commit
 T2: finished committing
```

MC-STM version 4: fine-grained locking

- Example of why locking on read is required:

T1 `Read-point: 0`

Ref	val
x	**2**

global state

Ref	v0
x	1
y	1
z	1

`Global write-point: 0`

T2 `Read-point: 0`

Ref	val

```
>T1: (mc-alter x inc)
 T2: (mc-alter z inc)
 T2: starts to commit
 T1: (mc-alter y inc)
 T1: starts to commit
 T2: finished committing
```

MC-STM version 4: fine-grained locking

- Example of why locking on read is required:

T1 `Read-point: 0`

Ref	val
x	2

global state

Ref	v0
x	1
y	1
z	1

`Global write-point: 0`

T2 `Read-point: 0`

Ref	val
z	**2**

```
 T1: (mc-alter x inc)
>T2: (mc-alter z inc)
 T2: starts to commit
 T1: (mc-alter y inc)
 T1: starts to commit
 T2: finished committing
```

MC-STM version 4: fine-grained locking

- Example of why locking on read is required:

T1 `Read-point: 0`

Ref	val
x	2

global state

Ref	v0	v1
x	1	
y	1	
z	**1**	**2**

`Global write-point: 1`

T2 `Read-point: 0`

Ref	val
z	2

`Write-point: 1`

```
 T1: (mc-alter x inc)
 T2: (mc-alter z inc)
>T2: starts to commit
 T1: (mc-alter y inc)
 T1: starts to commit
 T2: finished committing
```

MC-STM version 4: fine-grained locking

- Example of why locking on read is required:

T1 `Read-point: 0`

Ref	val
x	2
y	**2**

global state

Ref	v0	v1
x	1	
y	1	
z	1	2

`Global write-point: 1`

T2 `Read-point: 0`

Ref	val
z	2

`Write-point: 1`

```
 T1: (mc-alter x inc)
 T2: (mc-alter z inc)
 T2: starts to commit
>T1: (mc-alter y inc)
 T1: starts to commit
 T2: finished committing
```

MC-STM version 4: fine-grained locking

- Example of why locking on read is required:

T1 `Read-point: 0`

Ref	val
x	2
y	2

`Write-point: 2`

global state

Ref	v0	v1	v2
x	**1**		**2**
y	1		
z	1	2	

`Global write-point: 2`

T2 `Read-point: 0`

Ref	val
z	2

`Write-point: 1`

```
 T1: (mc-alter x inc)
 T2: (mc-alter z inc)
 T2: starts to commit
 T1: (mc-alter y inc)
>T1: starts to commit
 T2: finished committing
```

MC-STM version 4: fine-grained locking

- Example of why locking on read is required:

T1 `Read-point: 0`

Ref	val
x	2
y	2

Note that T1 first acquires locks on *all* refs it wrote to before changing any of them

global state

Ref	v0	v1	v2
x	**1**		**2**
y	1		
z	1	2	

`Global write-point: 2`

T2 `Read-point: 0`

Ref	val
z	2

`Write-point: 1`

```
 T1: (mc-alter x inc)
 T2: (mc-alter z inc)
 T2: starts to commit
 T1: (mc-alter y inc)
>T1: starts to commit
 T2: finished committing
```

MC-STM version 4: fine-grained locking

- Example of why locking on read is required:

T1 `Read-point: 0`

Ref	val
x	2
y	2

`Write-point: 2`

T2 `Read-point: 0`

Ref	val
z	2

`Write-point: 1`

global state

Ref	v0	v1	v2
x	1		2
y	1		
z	1	2	

`Global write-point: 2`

```
 T1: (mc-alter x inc)
 T2: (mc-alter z inc)
 T2: starts to commit
 T1: (mc-alter y inc)
 T1: starts to commit
>T2: finished committing
```

MC-STM version 4: fine-grained locking

- Example of why locking on read is required:

T1 `Read-point: 0`

Ref	val
x	2
y	2

A transaction T3 that starts with read-point 2 will not see an inconsistent state where x = 2 and y = 1 because T1 still holds the locks, and T3 will acquire these on first read

T2 `Read-point: 0`

Ref	val
z	2

`Write-point: 1`

global state

Ref	v0	v1	v2
x	1		2
y	1		
z	1	2	

`Global write-point: 2`

```
 T1: (mc-alter x inc)
 T2: (mc-alter z inc)
 T2: starts to commit
 T1: (mc-alter y inc)
 T1: starts to commit
>T2: finished committing
```

MC-STM: version 4 limitations

- MC-STM v1-v4 does *lazy* conflict detection: transactions with write-conflicts abort only when they fail validation at commit-time

- Can lead to lots of irrelevant computation before retrying

Contention Management

- Clojure STM uses “barging”: transactions detect write conflicts during the transaction and proactively try to “barge” other transactions.
  - Transactions publicly “mark” refs written inside transaction. This enables early conflict detection before commit *(eager acquire)*
  - Transaction A can only barge transaction B if A is older than B (according to starting time), and B is still running. Otherwise, A itself retries.
  - When a transaction is barged, it retries

MC-STM version 5: barging

- Transactions extended with a start timestamp and a status field (status is one of :RUNNING, :RETRY, :KILLED, :COMMITTING, :COMMITTED)
- Each mc-ref extended with :acquired-by field pointing to the last transaction that successfully acquired it
- On tx-write, a transaction actively checks for write conflicts and either barges the other transaction or retries itself.
- On tx-commit, no longer necessary to validate written-refs
- Whenever a transaction reads/writes/ensures/commutes a ref or commits, it checks whether it was barged and if so, retries.
- Won't cover all the details, see https://github.com/tvcutsem/stm-in-clojure

MC-STM version 5: barging

- Example of eager acquisition: T1 and T2 both try to increment x by 1

T1 `Read-point: 0`

Ref	val

id: 1
status: RUNNING

T2 `Read-point: 0`

Ref	val

id: 2
status: RUNNING

global state

Ref	Acq	v0
x		1

`Global write-point: 0`

```
>
 T1: (mc-alter x inc)
 T2: (mc-alter x inc)
 T2: retry
 T1: commits
 T2: restarts
 T2: (mc-alter x inc)
 T2: commits
```

MC-STM version 5: barging

T1 `Read-point: 0`

Ref	val
x	**2**

```
id: 1
status: RUNNING
```

T2 `Read-point: 0`

Ref	val

```
id: 2
status: RUNNING
```

global state

Ref	Acq	v0
x	**T1**	1

```
Global write-point: 0
```

T1 notices that x was not yet acquired by any other transaction, so acquires x by marking it as acquired by T1

```
T1: (mc-alter x inc)
T2: (mc-alter x inc)
T2: retry
T1: commits
T2: restarts
T2: (mc-alter x inc)
T2: commits
```

MC-STM version 5: barging

T1

`Read-point: 0`

Ref	val
x	2

```
id: 1
status: RUNNING
```

T2

`Read-point: 0`

Ref	val

```
id: 2
status: RUNNING
```

global state

Ref	Acq	v0
x	T1	1

`Global write-point: 0`

T2 notices that x was acquired by T1. Since T1 is still RUNNING, T2 tries to barge T1 but fails since T1's id < T2's id

```
 T1: (mc-alter x inc)
>T2: (mc-alter x inc)
 T2: retry
 T1: commits
 T2: restarts
 T2: (mc-alter x inc)
 T2: commits
```

MC-STM version 5: barging

T1 `Read-point: 0`

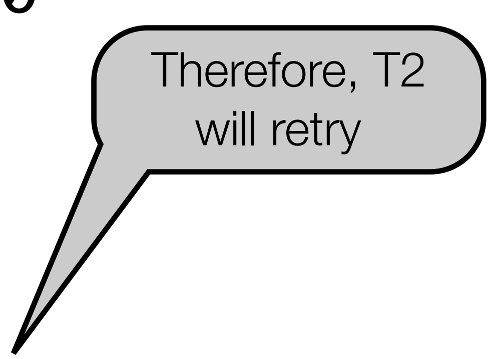
Ref	val
x	2

```
id: 1
status: RUNNING
```

T2 `Read-point: 0`

Ref	val

```
id: 2
status: RETRY
```



global state

Ref	Acq	v0
x	T1	1

`Global write-point: 0`

```
 T1: (mc-alter x inc)
 T2: (mc-alter x inc)
>T2: retry
 T1: commits
 T2: restarts
 T2: (mc-alter x inc)
 T2: commits
```

MC-STM version 5: barging

T1 `Read-point: 0`

Ref	val
x	2

```
id: 1
status: COMMITTED
```

T2 `Read-point: 0`

Ref	val

```
id: 2
status: RETRY
```

global state

Ref	Acq	v0	v1
x	T1	1	2

```
Global write-point: 1
```

```
 T1: (mc-alter x inc)
 T2: (mc-alter x inc)
 T2: retry
>T1: commits
 T2: restarts
 T2: (mc-alter x inc)
 T2: commits
```

MC-STM version 5: barging

T1 `Read-point: 0`

Ref	val
x	2

```
id: 1
status: COMMITTED
```

T2 `Read-point: 1`

Ref	val

```
id: 2
status: RUNNING
```

global state

Ref	Acq	v0	v1
x	T1	1	2

```
Global write-point: 1
```

```
 T1: (mc-alter x inc)
 T2: (mc-alter x inc)
 T2: retry
 T1: commits
>T2: restarts
 T2: (mc-alter x inc)
 T2: commits
```

MC-STM version 5: barging

T1 `Read-point: 0`

Ref	val
x	2

```
id: 1
status: COMMITTED
```

global state

Ref	Acq	v0	v1
x	**T2**	1	2

`Global write-point: 1`

T2 notices that x was acquired by T1. Since T1 is COMMITTED, so no longer active, T2 can safely acquire x

T2 `Read-point: 1`

Ref	val
x	**3**

```
id: 2
status: RUNNING
```

```
 T1: (mc-alter x inc)
 T2: (mc-alter x inc)
 T2: retry
 T1: commits
 T2: restarts
>T2: (mc-alter x inc)
 T2: commits
```

MC-STM version 5: barging

T1 `Read-point: 0`

Ref	val
x	2

```
id: 1
status: COMMITTED
```

T2 `Read-point: 1`

Ref	val
x	3

```
id: 2
status: COMMITTED
```

global state

Ref	Acq	v0	v1	v2
x	T2	1	2	3

```
Global write-point: 2
```

```
 T1: (mc-alter x inc)
 T2: (mc-alter x inc)
 T2: retry
 T1: commits
 T2: restarts
 T2: (mc-alter x inc)
>T2: commits
```

MC-STM: summary

- Like Clojure, based on MVCC to guarantee internal consistency
- Supports conflict-free commutative updates
- Supports ensure to prevent write skew
- From single global commit-lock to fine-grained locking (one lock / mc-ref)