Adventures in Clojure
Navigating the STM sea and exploring Worlds

Tom Van Cutsem
Part 1: Clojure in a
Clojure in a nutshell

- A modern Lisp dialect (2007), designed by Rich Hickey
- JVM as runtime platform
- Promotes a Functional Programming style
- Designed for Concurrency
Functional Style

• Clojure is not a pure functional language (like Haskell), but...

• Emphasis on immutable data structures

• Lisp’s lists generalized to abstract sequences: list, vector, set, map, ...

  • Used pervasively: all Clojure collections, all Java collections, Java arrays and Strings, regular expression matches, directory structures, I/O streams, XML trees, ...

  • Sequences are lazy and immutable
Clojure and Java

• Clojure compiles to JVM bytecode

• Easy for Clojure to reuse Java libraries

(new java.util.Random); Java: “new java.util.Random()”
=> java.util.Random@18a4f2

(. aRandom nextInt); Java: “aRandom.nextInt()”
=> 23494372
Part 2: Concurrency in Clojure
Persistent Data Structures

• The problem with immutable data structures: updates are costly (copy)

• Persistent data structures preserve old copies of themselves by efficiently *sharing structure* between older and newer versions.

• Simplest example: consing an element onto a linked list

  ```lisp
  (def a '(1 2))
  (def b (cons 0 a))
  ```

• b reuses all of a’s structure instead of having its own private copy
Persistent Data Structures

• Not only for linked lists, also for vectors, sets, maps, ...

• Example: binary tree insert

(def map1 {"a" 1, "b" 2, "d" 4, "e" 5})
(def map2 (assoc map1 "c" 3))
Persistent Data Structures

- Not only for linked lists, also for vectors, sets, maps, ...

- Example: binary tree insert

```scheme
(def map1 {"a" 1, "b" 2, "d" 4, "e" 5})
(def map2 (assoc map1 "c" 3))
```
• Clojure reuses JVM threads as the unit of concurrency

    (.start (Thread.
              (fn [] (println "Hello from new thread"))))

• Not as bad as it looks: Clojure does not combine threads with unbridled
  access to pervasive shared mutable state
## Clojure Philosophy

- Immutable state is the default

- Where mutable state is required, programmer must explicitly select one of the following APIs:

<table>
<thead>
<tr>
<th>State change is</th>
<th>Asynchronous</th>
<th>Synchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordinated</td>
<td>-</td>
<td>Refs</td>
</tr>
<tr>
<td>Independent</td>
<td>Agents</td>
<td>Atoms</td>
</tr>
</tbody>
</table>
### Clojure’s concurrency primitives

<table>
<thead>
<tr>
<th>state change is</th>
<th>Asynchronous</th>
<th>Synchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordinated</td>
<td>-</td>
<td>Refs</td>
</tr>
<tr>
<td>Independent</td>
<td>Agents</td>
<td>Atoms</td>
</tr>
</tbody>
</table>
Refs and Software Transactional Memory (STM)

- Ref: mutable reference to an immutable object

```
(def today (ref "Monday"))
```

- The ref wraps and protects its internal state. To read its contents, must explicitly dereference it:

```
(deref today)
=> "Monday"
```

```
@today
=> "Monday"
```
Refs and Software Transactional Memory (STM)

- To update a reference:

  `(ref-set today "Tuesday")`

- Updates can only occur in the context of a transaction:

  `(ref-set today "Tuesday")
  => java.lang.IllegalArgumentException: No transaction running`
Refs and Software Transactional Memory (STM)

- To start a transaction:

  \[(\text{dosync \ body})\]

- Example:

  \[(\text{dosync (ref-set today \text{"Tuesday"})})\]

  => "Tuesday"
Coordinated updates

- Changes to multiple refs within a transaction are **atomic** and **isolated**

```
(dosync
  (ref-set yesterday "Monday")
  (ref-set today "Tuesday"))
```

- No other thread will be able to observe a state in which `yesterday` is already updated to “Monday”, while `today` is still set to “Monday”. 
alter

- Often, the new state of a reference is dependent on the old state

```
(def weekdays ["mon","tue","wed","thu","fri","sat","sun"])

(def today-idx (ref 0))

(dosync
 (ref-set today-idx (mod (inc @today-idx) 7))))

; alternatively (preferred)
(defn next-day-idx [i] (mod (inc i) 7))
(dosync
 (alter today-idx next-day-idx))
```
Example: money transfer

- Transferring money atomically from one bank account to another

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

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

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

(transfer 100 accountA accountB)
(printn @accountA); 1400
(printn @accountB); 300
```
How STM Works: MVCC

• Multiversion concurrency control (MVCC): each transaction starts with a "snapshot" of the database (i.e. the state of all refs).

• Instead of updating data directly, each transaction modifies its own private copy of the data.

  • Persistent data structures: private copy shares most of its structure with the original value

  • Changes made to private copies will not be seen by other transactions until the transaction commits.
MVCC: Example

(def today (ref "mon"))
(def yesterday (ref "sun"))

T1: (dosync
   (list (deref today)
         (deref yesterday)))

T2: (dosync
   (ref-set today "tue")
   (ref-set yesterday "mon"))
MVCC: Example

```clojure
(def today (ref "mon"))
(def yesterday (ref "sun"))

T1: (dosync
  (list (deref today)
    (deref yesterday)))

T2: (dosync
  (ref-set today "tue")
  (ref-set yesterday "mon"))
```

Global "ref" state:

<table>
<thead>
<tr>
<th>Ref</th>
<th>rev 0</th>
<th>rev 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;mon&quot;</td>
<td></td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;sun&quot;</td>
<td></td>
</tr>
</tbody>
</table>

Both T1 and T2 start with read-point 0

> T2: (ref-set today "tue")
  T1: (deref today)
  T2: (ref-set yesterday "mon")
  T1: (deref yesterday)
  T2: commit
  T1: commit
MVCC: Example

(def today (ref "mon"))
(def yesterday (ref "sun"))
T1: (dosync
   (list (deref today)
         (deref yesterday)))
T2: (dosync
   (ref-set today "tue")
   (ref-set yesterday "mon"))

in-transaction-values of T1

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

in-transaction-values of T2

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;tue&quot;</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

global "ref" state

<table>
<thead>
<tr>
<th>Ref</th>
<th>rev 0</th>
<th>rev 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;mon&quot;</td>
<td></td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;sun&quot;</td>
<td></td>
</tr>
</tbody>
</table>
MVCC: Example

(def today (ref "mon"))
(def yesterday (ref "sun"))

T1: (dosync
  (list (deref today)
    (deref yesterday)))

T2: (dosync
  (ref-set today "tue")
  (ref-set yesterday "mon"))

global "ref" state

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;mon&quot;</td>
<td>0</td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;sun&quot;</td>
<td></td>
</tr>
</tbody>
</table>

in-transaction-values of T1

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;mon&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

in-transaction-values of T2

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;tue&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>
MVCC: Example

```clojure
(def today (ref "mon"))
(def yesterday (ref "sun"))
T1: (dosync
    (list (deref today)
          (deref yesterday)))
T2: (dosync
    (ref-set today "tue")
    (ref-set yesterday "mon"))
```

**global “ref” state**

<table>
<thead>
<tr>
<th>Ref</th>
<th>rev 0</th>
<th>rev 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;mon&quot;</td>
<td></td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;sun&quot;</td>
<td></td>
</tr>
</tbody>
</table>

**in-transaction-values of T1**

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;mon&quot;</td>
<td>0</td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;mon&quot;</td>
<td></td>
</tr>
</tbody>
</table>

**in-transaction-values of T2**

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;tue&quot;</td>
<td>0</td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;mon&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>
MVCC: Example

(def today (ref "mon"))
(def yesterday (ref "sun"))

T1: (dosync
    (list (deref today)
          (deref yesterday)))

T2: (dosync
    (ref-set today "tue")
    (ref-set yesterday "mon"))

in-transaction-values of T1

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;mon&quot;</td>
<td>0</td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;sun&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

global "ref" state

<table>
<thead>
<tr>
<th>Ref</th>
<th>rev 0</th>
<th>rev 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;mon&quot;</td>
<td></td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;sun&quot;</td>
<td></td>
</tr>
</tbody>
</table>

in-transaction-values of T2

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;tue&quot;</td>
<td>0</td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;mon&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

T2: (ref-set today "tue")
T1: (deref today)
T2: (ref-set yesterday "mon")
>T1: (deref yesterday)
T2: commit
T1: commit
**MVCC: Example**

```clojure
(def today (ref "mon"))
(def yesterday (ref "sun"))
T1: (dosync
  (list (deref today)
        (deref yesterday)))
T2: (dosync
  (ref-set today "tue")
  (ref-set yesterday "mon"))
```

### Global "ref" State

<table>
<thead>
<tr>
<th>Ref</th>
<th>rev 0</th>
<th>rev 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;mon&quot;</td>
<td>&quot;tue&quot;</td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;sun&quot;</td>
<td>&quot;mon&quot;</td>
</tr>
</tbody>
</table>

T2 has write-point 1, updates the global ref state atomically.

---

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;mon&quot;</td>
<td>0</td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;sun&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;tue&quot;</td>
<td>0</td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;mon&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

T2: (ref-set today "tue")
T1: (deref today)
T2: (ref-set yesterday "mon")
T1: (deref yesterday)

> T2: commit
T1: commit
MVCC: Example

```clojure
(def today (ref "mon"))
(def yesterday (ref "sun"))
T1: (dosync
    (list (deref today)
          (deref yesterday)))
T2: (dosync
    (ref-set today "tue")
    (ref-set yesterday "mon"))
```

Global "ref" state

<table>
<thead>
<tr>
<th>Ref</th>
<th>rev 0</th>
<th>rev 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;mon&quot;</td>
<td>&quot;tue&quot;</td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;sun&quot;</td>
<td>&quot;mon&quot;</td>
</tr>
</tbody>
</table>

In-transaction-values of T1

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;mon&quot;</td>
<td>0</td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;sun&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

In-transaction-values of T2

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;tue&quot;</td>
<td>0</td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;mon&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

T1 has read consistent versions of both refs, no conflict

T2: (ref-set today "tue")
T1: (deref today)
T2: (ref-set yesterday "mon")
T1: (deref yesterday)
T2: commit
>T1: commit
MVCC: Example of a conflict

(def today (ref "mon"))
(def yesterday (ref "sun"))
T1: (dosync
  (ref-set today "sun")
  (ref-set yesterday "sat"))
T2: (dosync
  (ref-set today "tue")
  (ref-set yesterday "mon"))
MVCC: Example of a conflict

```clojure
(def today (ref "mon"))
(def yesterday (ref "sun"))
T1: (dosync
    (ref-set today "sun")
    (ref-set yesterday "sat"))
T2: (dosync
    (ref-set today "tue")
    (ref-set yesterday "mon"))
```

Both T1 and T2 start with read-point 0

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;mon&quot;</td>
<td></td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;sun&quot;</td>
<td></td>
</tr>
</tbody>
</table>

> T1: (ref-set today "sun")
T2: (ref-set today "tue")
T1: (ref-set yesterday "sat")
T2: (ref-set yesterday "mon")
T1: commit
T2: commit
MVCC: Example of a conflict

(def today (ref "mon"))
(def yesterday (ref "sun"))

T1: (dosync
    (ref-set today "sun")
    (ref-set yesterday "sat"))

T2: (dosync
    (ref-set today "tue")
    (ref-set yesterday "mon"))

global “ref” state

<table>
<thead>
<tr>
<th>Ref</th>
<th>rev 0</th>
<th>rev 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>“mon”</td>
<td></td>
</tr>
<tr>
<td>yesterday</td>
<td>“sun”</td>
<td></td>
</tr>
</tbody>
</table>

in-transaction-values of T1

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>“sun”</td>
<td>0</td>
</tr>
</tbody>
</table>

in-transaction-values of T2

>T1: (ref-set today “sun”)
T2: (ref-set today “tue”)
T1: (ref-set yesterday “sat”)
T2: (ref-set yesterday “mon”)
T1: commit
T2: commit
MVCC: Example of a conflict

```
(def today (ref "mon"))
(def yesterday (ref "sun"))

T1: (dosync
  (ref-set today "sun")
  (ref-set yesterday "sat"))

T2: (dosync
  (ref-set today "tue")
  (ref-set yesterday "mon"))
```

**global “ref” state**

<table>
<thead>
<tr>
<th>Ref</th>
<th>rev 0</th>
<th>rev 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;mon&quot;</td>
<td></td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;sun&quot;</td>
<td></td>
</tr>
</tbody>
</table>

**in-transaction-values of T1**

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;sun&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

**in-transaction-values of T2**

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;tue&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

T1: (ref-set today "sun")

> T2: (ref-set today "tue")

T1: (ref-set yesterday "sat")

T2: (ref-set yesterday "mon")

T1: commit

T2: commit
MVCC: Example of a conflict

(def today (ref "mon"))
(def yesterday (ref "sun"))
T1:  (dosync
    (ref-set today "sun")
    (ref-set yesterday "sat"))
T2:  (dosync
    (ref-set today "tue")
    (ref-set yesterday "mon"))

in-transaction-values of T1

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;sun&quot;</td>
<td>0</td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;sat&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

in-transaction-values of T2

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;tue&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

T1:  (ref-set today "sun")
T2:  (ref-set today "tue")
>T1:  (ref-set yesterday "sat")
T2:  (ref-set yesterday "mon")
T1: commit
T2: commit

global “ref” state

<table>
<thead>
<tr>
<th></th>
<th>rev 0</th>
<th>rev 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;mon&quot;</td>
<td></td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;sun&quot;</td>
<td></td>
</tr>
</tbody>
</table>
MVCC: Example of a conflict

(def today (ref "mon"))
(def yesterday (ref "sun"))

T1: (dosync
  (ref-set today "sun")
  (ref-set yesterday "sat"))

T2: (dosync
  (ref-set today "tue")
  (ref-set yesterday "mon"))

in-transaction-values of T1

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;sun&quot;</td>
<td>0</td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;sat&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

in-transaction-values of T2

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;tue&quot;</td>
<td>0</td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;mon&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

global “ref” state

<table>
<thead>
<tr>
<th></th>
<th>rev 0</th>
<th>rev 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;mon&quot;</td>
<td></td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;sun&quot;</td>
<td></td>
</tr>
</tbody>
</table>

T1: (ref-set today "sun")
T2: (ref-set today "tue")
T1: (ref-set yesterday "sat")
>T2: (ref-set yesterday "mon")
T1: commit
T2: commit
MVCC: Example of a conflict

```
(def today (ref "mon"))
(def yesterday (ref "sun"))

T1: (dosync
  (ref-set today "sun")
  (ref-set yesterday "sat"))

T2: (dosync
  (ref-set today "tue")
  (ref-set yesterday "mon"))
```

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;sun&quot;</td>
<td>0</td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;sat&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>&quot;tue&quot;</td>
<td>0</td>
</tr>
<tr>
<td>yesterday</td>
<td>&quot;mon&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

T1 has write-point 1, updates global ref state atomically

```
> T1: commit
  T2: commit
```
MVCC: Example of a conflict

(def today (ref “mon”))
(def yesterday (ref “sun”))

T1: (dosync
    (ref-set today “sun”)
    (ref-set yesterday “sat”))

T2: (dosync
    (ref-set today “tue”)
    (ref-set yesterday “mon”))

goal “ref” state

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>“sun”</td>
<td>0</td>
</tr>
<tr>
<td>yesterday</td>
<td>“sat”</td>
<td>0</td>
</tr>
</tbody>
</table>

in-transaction-values of T1

<table>
<thead>
<tr>
<th>Ref</th>
<th>val</th>
<th>rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>“mon”</td>
<td>0</td>
</tr>
<tr>
<td>yesterday</td>
<td>“sun”</td>
<td>0</td>
</tr>
</tbody>
</table>

T2 notices that the refs it modified have already been modified, since the latest version of the refs (1) is no longer equal to its read-point (0)

T2 will abort and retry, this time with read-point 1

T1: (commit)
T2: (commit)
Transactions, side effects, retries

• Transactions may be aborted and retried.

• The transaction body may be executed multiple times.
  • Should avoid side-effects other than assigning to refs
  • Especially: avoid any form of I/O (launchMissiles())
### Clojure’s concurrency primitives

<table>
<thead>
<tr>
<th>state change is</th>
<th>Asynchronous</th>
<th>Synchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordinated</td>
<td>-</td>
<td>Refs</td>
</tr>
<tr>
<td>Independent</td>
<td>Agents</td>
<td>Atoms</td>
</tr>
</tbody>
</table>
Atoms

- For uncoordinated (independent), synchronous updates
- More lightweight than refs: atoms are updated independently, no need for transactions
- Two or more atoms cannot be updated in a coordinated way

```clojure
(def today-idx (atom 0))

@today-idx
=> 0
```
Updating Atoms

• To update an atom, use swap!

  \(\text{(swap! today-idx inc)}\)

• swap! calculates new value and performs an atomic test-and-set: if the atom’s value was changed concurrently (by another thread), it will retry

  • The update function may be called multiple times => should be side-effect free

  • Concurrently calling swap! on the same atom is thread-safe
## Clojure’s concurrency primitives

<table>
<thead>
<tr>
<th>state change is</th>
<th>Asynchronous</th>
<th>Synchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordinated</td>
<td>-</td>
<td>Refs</td>
</tr>
<tr>
<td>Independent</td>
<td>Agents</td>
<td>Atoms</td>
</tr>
</tbody>
</table>
Agents

- Both refs and atoms can be updated synchronously

- If you can tolerate updates happening asynchronously, use agents

  \[(\text{agent } \text{initial-state})\]

- Can send a function (“action”) to an agent to update its state at a later point in time:

  \[(\text{send } \text{agent } \text{update-fn})\]

- send queues an update-fn to run later, on a thread in a thread pool
Agents: example

(defn make-account [init]
  (agent init))

(defn deposit [account amnt]
  (send account (fn [bal] (+ bal amnt))))

(defn withdraw [account amnt]
  (send account (fn [bal] (- bal amnt))))

(def a (make-account 0))
(deposit a 100) ; asynchronous
(withdraw a 50) ; asynchronous
(await a)
@a
=> 50
Unified Update Model

- Refs, Atoms and Agents all enable mutation of state by applying a function on an “old state” returning a “new state”:
  - Refs: \((\text{alter a-ref update-fn})\)
  - Atoms: \((\text{swap! an-atom update-fn})\)
  - Agents: \((\text{send an-agent update-fn})\)

- To read, call \texttt{deref/@}

<table>
<thead>
<tr>
<th>state change is</th>
<th>Asynchronous</th>
<th>Synchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordinated</td>
<td>-</td>
<td>Refs</td>
</tr>
<tr>
<td>Independent</td>
<td>Agents</td>
<td>Atoms</td>
</tr>
</tbody>
</table>
Part 3: A meta-circular STM in Clojure
Goal

- We have seen Clojure’s built-in support for STM via refs

- Recall:

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

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

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

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

• Build our own STM system in Clojure to better understand its implementation

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

(defn transfer [amount from to]
  (mc-dosync
   (mc-alter from (fn [bal] (- bal amount)))
   (mc-alter to (fn [bal] (+ bal 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 represent refs via atoms
- We call such refs “mc-refs” (meta-circular refs)
- Recall: atoms support synchronous but *uncoordinated* state updates
- We have to add the coordination through transactions ourselves
Iterative approach

• Developed 4 versions:

  • v1: does not use MVCC, simple but transactions may have an inconsistent view on the world (~120 loc)

  • v2: uses MVCC (like real Clojure), simple version with 1 global lock (~155 loc)

  • v3: adds support for advanced features (commute and ensure) (~197 loc)

  • v4: uses fine-grained locking (1 lock / mc-ref) (~222 loc)

• v5 upcoming: introduce contention management to ensure liveness (current versions prone to livelock)
Demo

- https://github.com/tvcutsem/stm-in-clojure
Part 4: Worlds
Worlds

- ECOOP 2011 paper by Alex Warth (Viewpoints Research Institute)
- Goal: scoped side-effects

```
p = new Point(1, 2);
```
• Javascript implementation of Worlds:

A = thisWorld;
p = new Point(1, 2);
B = A.sprout();
in B { p.y = 3; }
C = A.sprout();
in C { p.y = 7; }
C.commit();
• A Clojure Library for Worlds

• As in the STM experiment, we implemented our own new type of “ref”

• A “world-aware” ref or w-ref

```clojure
A = thisWorld;
p = new Point(1, 2);
B = A.sprout();
in B { p.y = 3; }
C = A.sprout();
in C { p.y = 7; }
C.commit();
```

```
(let [A (this-world)
p (new Point 1 2)
B (sprout A)]
  (in-world B
    (w-ref-set (:y p) 3))
(let [C (sprout A)]
  (in-world C
    (w-ref-set (:y p) 7))
  (commit C))
```
Example

(let [w (sprout (this-world))
       r (w-ref 0)]
  (w-deref r) ; 0
(in-world w
  (w-deref r) ; also 0
  (w-ref-set r 1))
  (w-deref r) ; still 0!
  (commit w)
  (w-deref r)) ; 1
Example: safe exception handling

```clojure
(try
doseq [elt seq]
  (alter elt update-fn)
catch e
  ; undo successful updates)
```
Example: safe exception handling

(try
  (in-world (sprout (this-world)))
  (doseq [elt seq]
    (w-alter elt update-fn))
  (commit (this-world))
  (catch e
    ; no cleanup required!
    ))
More examples

- "undo" functionality for objects / applications

- Scoped **monkey-patching**. E.g. extending java.lang.Object, but only for your application

- Safe **backtracking** in a logic language with side-effects (think Prolog assert)

  - Or in any kind of backtracking search in general...

```plaintext
1. choice
2. try 1st alternative (causes side-effects)
3. stuck
4. undo side-effects
5. try 2nd alternative
```
Future steps

• Experiment with concurrent Worlds

• How to merge concurrent updates to parallel worlds?
Conclusion

• Clojure: Lisp on the JVM

• Functional, but not pure

• Unified update model: refs, atoms, agents

• Experiments with extending the unified update model:
  
  • MC-STM: implementing meta-circular refs
  
  • clj-worlds: adding “world-refs” for scoped side-effects