Clojure Concurrency

Clojure

— A dialect of Lisp and a functional language
— Uses immutable **persistent data structures**
— Offers a **Software Transactional Memory** system and a reactive **Agent** system
— Concurrency based on Hoare's **CSP** and Dijkstra's **Guarded Commands**

Avoid mutable variables

```
(def hobbit-body-parts [{:name "head" :size 3}
                        {:name "eyes" :size 1}
                        {:name "mouth" :size 3}
                        {:name "arms" :size 3}
                        {:name "chest" :size 10}
                        {:name "feet" :size 2}])

(def size-sum (reduce + (map :size hobbit-body-parts)))

(def sum-to-10
  (loop [sum 0 x 0]
    (if (= x 11)
      sum
      (recur (+ sum x) (+ 1 x)))))

(println size-sum)
(println sum-to-10)
```

Persistent data structures

```
(def my-vector [1 2 3 4])
```

[1 2 3 4]

```
(def new-vector (conj my-vector 5))
```

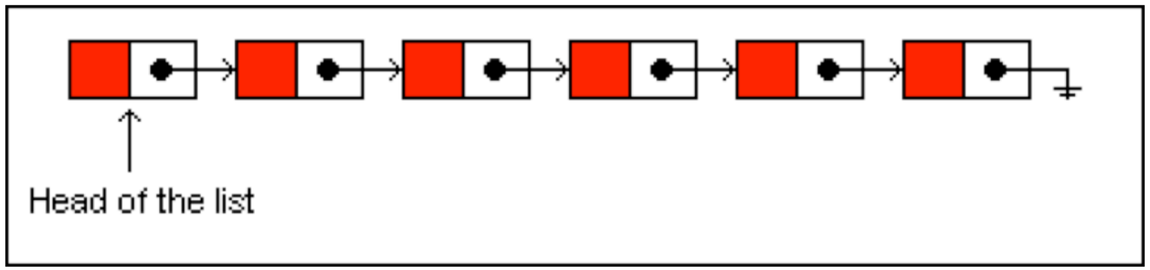
[1 2 3 4 5]

```
(println my-vector)
```

[1 2 3 4]

Persistent data structures

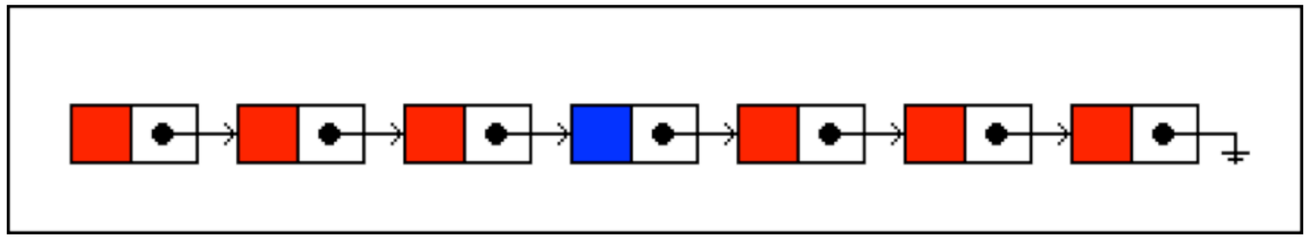
```
(def a { :name "Red" })
(...)
(def my-list '(a b c d e f))
```



Persistent data structures

```
(defn insert-at
  [index xs x]
  (let [[before after] (split-at index xs)]
    (list (concat before x after))))

(def g { :name "Blue" })
(insert-at 3 my-list g)
```



Atom, Ref and Agent

Along with Future

Atoms

— Implement Clojure's concept of state

— Allow to assign a new **value** to an **identity**

— **compare-and-set**

```
(def count (atom 0))

(swap! count + 10)

(println @count)
```

=> 10

Past state

```
(let [num (atom 1) s1 @num]
  (swap! num inc)
  (println "State 1:" s1)
  (println "Current state:" @num))
```

=> State 1: 1

=> Current state: 2

Future

```
(def task
  (future
    (Thread/sleep 1000)
    "task"))

(println @task)
```

Future and Atoms

```
(defn quote-word-count
  [number-of-quotes]
  (let [word-count (atom 0)]
    (dotimes [n number-of-quotes]
      (let [req (future (slurp "https://www.braveclojure.com/random-quote"))]
        (swap! word-count + (count @req))))
    @word-count))

(quote-word-count 5)
```

Agents

```
(ns clojure-noob.agents)

(def counter (agent 0))

(send counter (fn [c]
  (Thread/sleep 1000)
  (+ 10 c)))

(println @counter)
```

=> 0

Agents

— Like Atoms, but modified **asynchronously** by invoking a function, called **action**, in another thread

— Actions are queued up so that only one action at a time will be run per Agent

— Sending an action is not blocking

Refs

```
(def counter (ref 0))
(def counter-transaction
  (future
    (dosync
      (alter counter inc)
      (println @counter)
      (Thread/sleep 500)
      (alter counter inc)
      (println @counter))))
(Thread/sleep 250)
(println @counter)
```

=> 1

=> 0

=> 2

Software transactional memory

- **atomic**: all refs are updated or none of them are
- **consistent**: the refs always appear to have valid states
- **isolated**: changes are not visible outside the transaction until it commits

ACI D

Software transactional memory

- Operates on a consistent snapshot of the memory
- Any transaction is re-run if its memory has been modified before being committed
- **Optimistic** approach
- Implementation can guarantee that **deadlock never occurs**
- **Composable**

Software transactional memory

— Potential for large number of retries
— Overhead imposed by transaction bookkeeping
— No side-effects
— Work best in languages which distinguish mutable and immutable data (Clojure/Haskell)

Channels

clojure.core.async

Channels

— based on CSP by Hoare (1978)

— *put* and *take*

```
(def c (chan))

(go
  (dotimes [n 5]
    (>! c (str n))
    (Thread/sleep (rand-int 1000))))

(dotimes [n 5]
  (println (<!! c)))
```

Buffering

— *rendez-vous* by default

```
(def echo-chan (chan 2))
(go (println (<! echo-chan)))
(>!! echo-chan "ketchup") ; Doesn't block
```

As services

```
(defn operator
  [name]
  (let [c (chan)]
    (go
      (dotimes [n 5]
        (>! c (str name ": " n))
        (Thread/sleep (rand-int 1000))))
    c))

(def joe (operator "Joe"))
(def ann (operator "Ann"))

(dotimes [n 5]
  (println (<!! joe))
  (println (<!! ann)))
```

Multiplexing

```
(defn multiplexing
  [c1 c2]
  (let [c (chan)]
    (go (while true (>! c (<! c1))))
    (go (while true (>! c (<! c2))))
    c))

(let [c (multiplexing joe ann)]
  (dotimes [n 10]
    (println (<!! c))))
```

alts!

```
(defn multiplexing
  [c1 c2]
  (let [c (chan)]
    (go (while true
          (let [[value channel] (alts! [c1 c2])]
            (>! c value))))
    c))
```

Timeout

```
(defn multiplexing
  [c1 c2]
  (let [c (chan)]
    (go (while true
          (let [[value channel] (alts! [c1 c2 (timeout 500)])]
            (>! c (if value value "Timeout")))))
    c))
```

alt!

```
(alt!
  [c1 c2] ([val ch] (>!c val))
  timeout-ch "Timeout"
  :default 42)
```

Conclusions

— Immutable data
— Explicit variable shared state with atomic updates
— Composable transactions
— Coordination with channels

Honorable mentions

— *Delay* and *Promise*

— *core.reducers*

References I

- Source code of examples
- Clojure for the brave and true
- Persistent data structures
- Identity and State
- Clojure dosync vs Java synchronized
- Mark Volkmann's Software Transactional Memory (STM)

References III

— Concurrency via Software Transactional Memory

— Clojure Overview - Concurrent Programming

— Clojure core.async - Rich Hickey

— Go Concurrency Patterns - Rob Pike