Clojure

A (not-so-pure) functional approach to concurrency

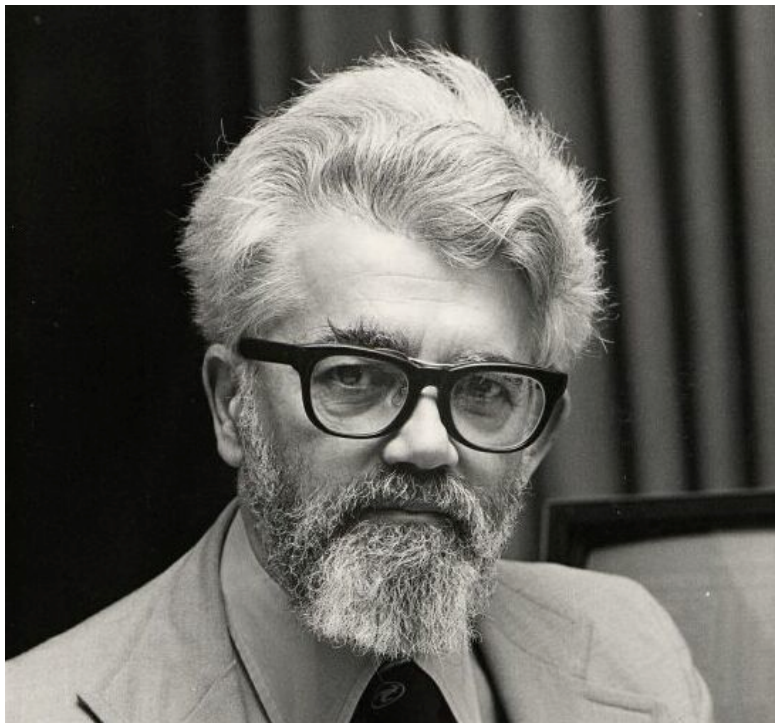
Paolo Baldan

Linguaggi per il Global Computing
AA 2018/2019

In the words of the inventor

- Functional programming language (rooted in Lisp, from 60s … old but beautifully compact language)
- symbiotic with an established platform (JVM)
- designed for concurrency
- with focus on immutability

Where all started



"uncle"
John McCarthy

Turing award
1971

"Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I" MIT 1960

Pragmatic

- Focus on pure functions (immutable data types), but allows for mutability
- Functional, but allows for control (e.g., iteration)
- Direct compilation into JVM (strings are java strings, …, access to Java's libraries)
- Used in industry (CapitalOne, Amazon, Facebook, Oracle, Boeing)

Outline

- **Basics**: expressions and evaluation
- **Functional parallelism** (on data, for free)
- **Programmable concurrency**: futures, promises and all that

Basics

Program as data

- **Code** as a **data structure** (**everything** is a **list**)
- Tools offered for manipulating data as code (eval, quote, apply, etc.)
- Make the language easily **extensible** (see macros)

"A programmable programming language."

Starting …

- Numbers

```
user> 9
9
```

- Strings

```
user> "Here I am"
"Here I am"
```

- Keywords (atoms)

```
user> :name
:name
```

Evaluate to themselves

Lists

- … it's essentially all about that

```
(mylist 1 2)
```

```
nil
```

- In the evaluation the first element is intepreted as a function

```
user> (+ 1 2)
3
```

```
user> (* 2 (+ 1 2 3))
12
```

Defining things: vars

- **Vars**: names for pieces of Clojure data

```
user> (def word "cap")
#'user/word
```

```
user> (def len (count word))
#'user/len
```

- Evaluate to the corresponding value

```
user> len
3
```

Vectors

- Int indexed vectors

```
user> (def myvec [12 "some string" :name])
#'user/myvec

user=> (myvec 2)
:name

user=> (myvec 1)
"some string”

user=>(myvec 3)
IndexOutOfBoundsException … (PersistentVector.java:158)
```

Maps

- Key/val maps

```
user=> (def mymap {"one" 1, "two" 2, "three" 3})
#'user/mymapuser=> (mymap "one")
1

user=> (mymap "on")
Nil
```

- Comma ok style

```
user=> (mymap "on" :error)
:error
```

Functions

- **Unnamed** functions

```
(fn [x y]
    "Take the x percentage of y"
    (* x (/ y 100.0)))      ; evaluate in sequence
                            ; value is last
```

- **Naming** functions with **def**

```
(def percentage
    "Take the x percentage of y"
    (fn [x y]
        (* x (/ y 100.0)))
)
```

Functions

- Defining functions with **defn**

```
(defn percentage
    "Take the x percentage of y"
    [x y]
    (* x (/ y 100.0))
)
```

- Different **arities**

```
(defn percentage
    "Take the x percentage of y (50% by default)"
    ([x y] (* x  (/  y 100.0)))
    ([y]   (* 50 (/ y 100.0)))
)
```

Control & Recursion

- **If**

```
(defn sum-down-from [x]
  (if (pos? x)
    (+ x (sum-down-from (dec x)))
    0 )
)
```

- **Case-like**

```
(defn fib [n]
  (cond
    (= n 0) 0
    (= n 1) 1
    :else (+ (fib (- n 1))
             (fib (- n 2)))))
```

Careful with recursion

- Very natural, but can have horrible performances (try with 9 or 100)
- Tail recursive

```
(defn tail-fib [n]
   (letfn [(fib [current next k]
             ; idea: fib [fib(n-k) fib(n+1-k) k]
             (if (zero? k)
                current
                (fib next (+ current  next) (dec k))))]
    (fib 0N 1N n)))
```

- Better, but no tail call elimination (try with 10000)

Tail elimination, do it yourself with recur

- Tail recursion, explicitly with loop-recur

```
(defn fib-recur [n]
  (loop [current 0N, next 1N, k n]
    (if (zero? k)
      current
      (recur next (+ current  next) (dec k)))))
```

- Works better!

```
=> (fib-recur 10000)
336447648764317832666216120051075433103021484606800639065647699746800814421666623681555955136337340255820653326808361593737347904838652682630408924630564318873
545443695598274916066020998841839338646527313000888302692356736131351175792974378544137521305205043477016022647583189065278908551543661595829872796829875106312
005754287834532155151038708182989697916131278562650331954871402142875326981879620469360978799003509623022910263681314931952756302278376284415403605844025721143
349611800230912082870460889239623288354615057765832712525460935911282039252853934346209042452489294039017062338889910858410651831733604374707379085526317643257
339937128719375877468974799263058370657428301616374089691784263786242128352581128205163702980893320999057079200643674262023897831114700540749984592503606335609
338838319233867830561364353518921332797329081337326426526339897639227234078829281779535805709936910491754708089318410561463223382174656373212482263830921032977
016480547262438423748624114530938122065649140327510866433945175121615265453613331113140424368548051067658434935238369595634280717687753283482343455573667197313
927462736291082106792807847180353291311767789246590899386354593278945237776744061922403376386740040213303432974969020283281459334188268176838930720036347956231
171031012919531697946076327375892535307725523759437884345040677155557790564504430166401194625809722167297586150269684431469520346149322911059706762432685159928
347098912847067408620085871350162603120719031720860940812983215810772820763531866246112782455372085323653057759564300725177443150515396009051686032203491632226
408852488524331580515348496224348482993809050704834824493274537326245677558790891871908036620580095947431500524025327097469953187707243768259074199396322659841
474981936092852239450397071654431564213281576889080587831834049174345562705202235648464951961124602683139709750693826487066132645076650746115126775227486215986
425307112984411826226610571635150692600298617049454250474913781151541399415506712562711971332527636319396069028956502882686083622410820505624307017949761711212
330660733100599473668755N
```

Quoting and unquoting

- **Quote**: prevent evaluation

```
=> (+ 1 2)
3
=> '(+ 1 2)                ; also (quote (+ 1 2))
(+ 1 2)
=> (first '(+ 1 2))
+
```

- **Eval**: force evaluation

```
=> (def mysum (quote (+ 1 1)) )
=> (first  mysum)
+
=> (eval  mysum)
2
```

Interfacing with Java

- Accessing and using Java classes

```
user> (new java.util.HashMap {"answer" 42,
                              "question" "who knows"})
```

```
user> (java.util.HashMap. {"answer" 42,
                           "question" "who knows"})
```

```
user> (def myhash
           (java.util.HashMap. {"answer" 42,
                                "question" "who knows"})
#'user/myhash
user> (.get myhash "answer")
42
```

Functional Parallelism (and laziness)

Functional parallelism

- Program are pure functions, copying not modifying
  - No mutable state:
  - No side effects
- Parallelization for map, reduce and all that
- Some form of laziness
  - Evaluate (realize) it when (if) you need it
  - Clojure is not lazy, in general, but sequences are

Summing numbers

- Sum of a sequence of numbers, recursively

```
(defn recursive-sum [numbers]
  (if (empty? numbers)
    0
    (+ (first numbers) (recursive-sum (rest numbers)))))
```

- Get rid of the tail

```
(defn recur-sum [numbers]
  (loop [ acc 0, list numbers ]
    (if (empty? list)
      acc
      (recur (+ acc (first list)) (rest list)))))
```

Using reduce

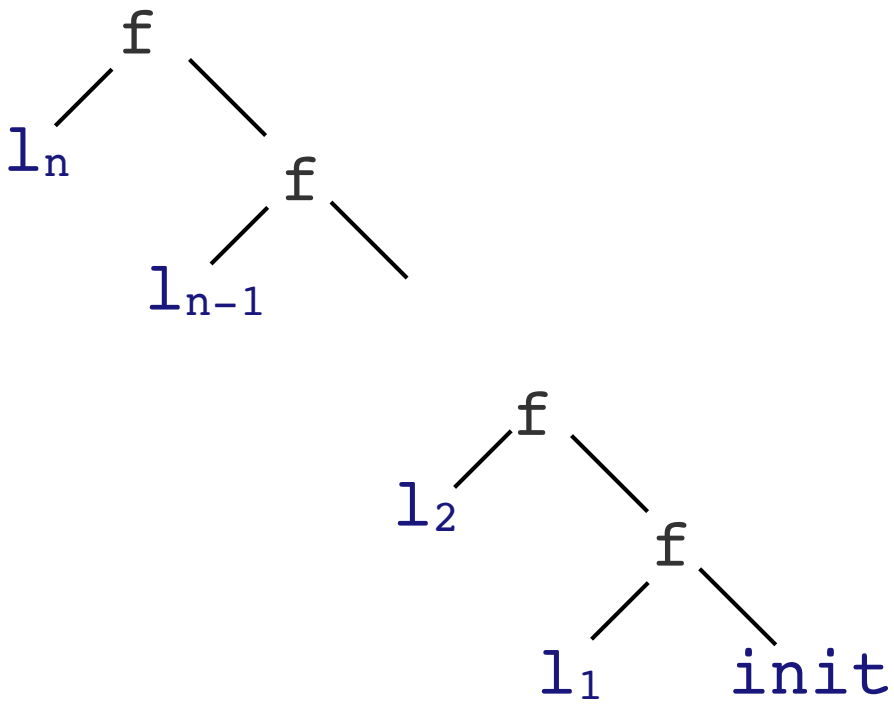
- Much simpler

```
(defn reduce-sum [numbers]
  (reduce (fn [acc x] (+ acc x)) 0 numbers))
```

- **Reduce**: applies the function once for each item of the collection
  - Initial result 0
  - Then apply the function on (result, 1st element)
  - then on (result, 2nd),
  - then on (result, 3rd) etc.

Reduce

```
(reduce f init [l1 ... ln])
```



Reduce, reprise

- Even simpler: since + is function summing two (or more) elements

```
(defn sum [numbers]
  (reduce + 0 numbers))
```

- reduce takes as standard initializer the zero of the type ...

```
(defn sum [numbers]
  (reduce + numbers))
```

- Looks fine, but still sequential. Can't we parallelize?

Computing frequencies

- Compute the number of occurrences of each word in a (large) text
- **Idea**: use a map

```
"the cat is on the table" -->
{"the" 2, "cat" 1, "is" 1, "on" 1, "table" 1}
```

Maps recap

```
=>(def counts {"the" 2, "cat" 1})
#user/counts
=>(get counts "the" 0)
2
=>(get counts "tho" 0)
0
=>(assoc counts "is" 1)
{"the" 2, "cat" 1, "is" 1 } ; returns new map
```

Word frequencies

- Word frequencies, sequentially, with reduce

```
(defn word-frequencies [words]
  (reduce
   (fn [counts word]
       (assoc counts
              word (inc (get counts word 0))))
   {} words))
```

```
=> (word-frequencies ["the" "cat" "is" "on" "the" “table"])
{"the" 2, "cat" 1, "is" 1, "on" 1, "table" 1}
```

- Compute frequencies, from the string

```
=> (word-frequencies (get-words "the cat is on the table"))
{"the" 2, "cat" 1, "is" 1, "on" 1, "table" 1}
```

Counting words, from several sources

- Imagine we want to count words from several strings
- **Idea**: List of lists of words of all the strings using `map`

```
=> (map get-words ["the cat is on the table"
                   "the dog is not on the table"])

(("the" "cat" "is" "on" "the" "table")
 ("the" "dog" "is" "not" "on" "the" "table"))
```

- and same for the frequencies

```
=> (map word-count
        (map get-words ["the cat is on the table"
                        "the dog is not on the table"])
```

Merging

- Then we need to merge the resulting maps
  - union of the keys
  - sum counts (for keys existing in both)

```
(merge-with f & maps)
```

- Proceeds left-to-right, using **f** for combining the values with common keys

```
(def merge-counts (partial merge-with +))

=>(merge-counts {:x 1 :y 2} {:y 1 :z 1})
{:z 1, :y 3, :x 1}
```

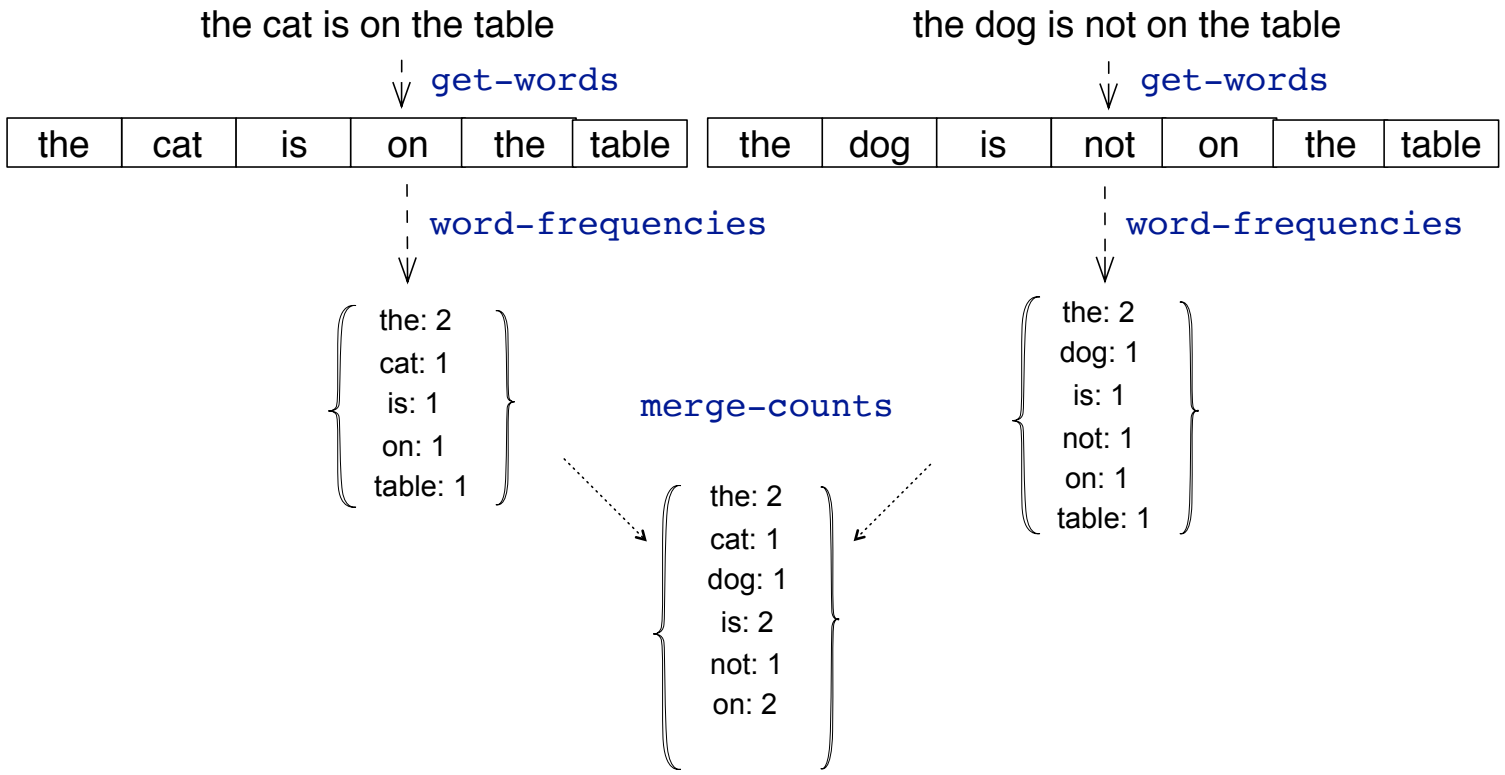
Counting words, from several sources

- Putting things together

```
(defn count-words-sequential [strings]
  (merge-counts
     (map word-frequencies
        (map get-words strings))))
```

Parallelizing!

- Idea:



- Process different strings and merge in parallel

Parallel Map

- Apply a given function to all elements of a sequence, in parallel

```
=> (pmap inc [1 2 3])
[2 3 4]
```

- Example: Slow inc

```
(defn slow-inc [x] (Thread/sleep 1000) (inc x))
```

```
(map slow-inc (range 10))

(pmap slow-inc (range 10))
```

- What's happening?

Parallelising

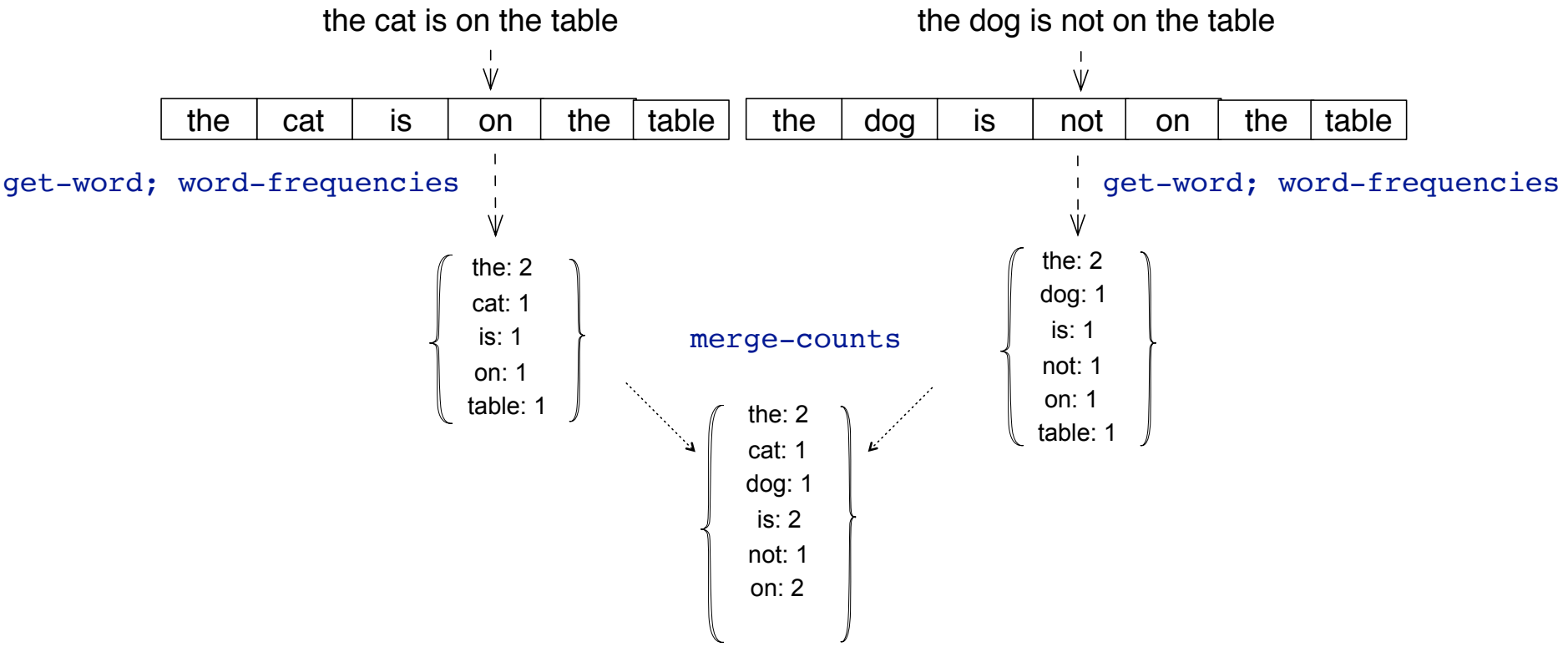
- Use pmap to perform map in parallel
- Avoid going through the sequences twice

```
(defn count-words-parallel [strings]
  (reduce merge-counts
    (pmap #(word-frequencies (get-words %)) strings)))
```

- Macro: Creates a function taking `%1, …,%n` as parameters (`%` stands for `%1`, if none constant fun)

Parallelizing!

- Idea:



- Process different strings and merge in parallel

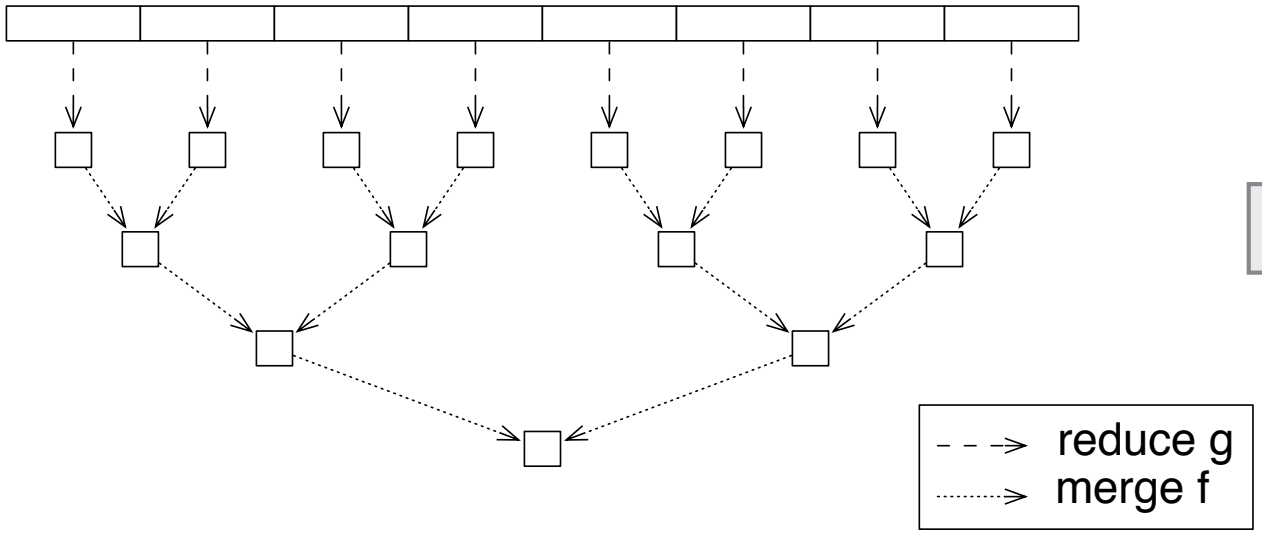
Batching

- The parallel version creates lot of processes for possibly too small jobs
- **Idea**: Create larger batches (size n)

```
(defn count-words-parallel-batch [strings n]
  (reduce merge-counts
          (pmap count-words-sequential
                (partition-all n strings))))
```

Using fold

- Fold works similarly:
  - Split in subproblems (divide & conquer)
  - Different functions for base and merge




```
(fold f g coll)
```

Fold

- The two functions can coincide …

```
(defn parallel-sum [numbers]
  (fold + numbers))
```

- Subproblems parallelized, more efficient!

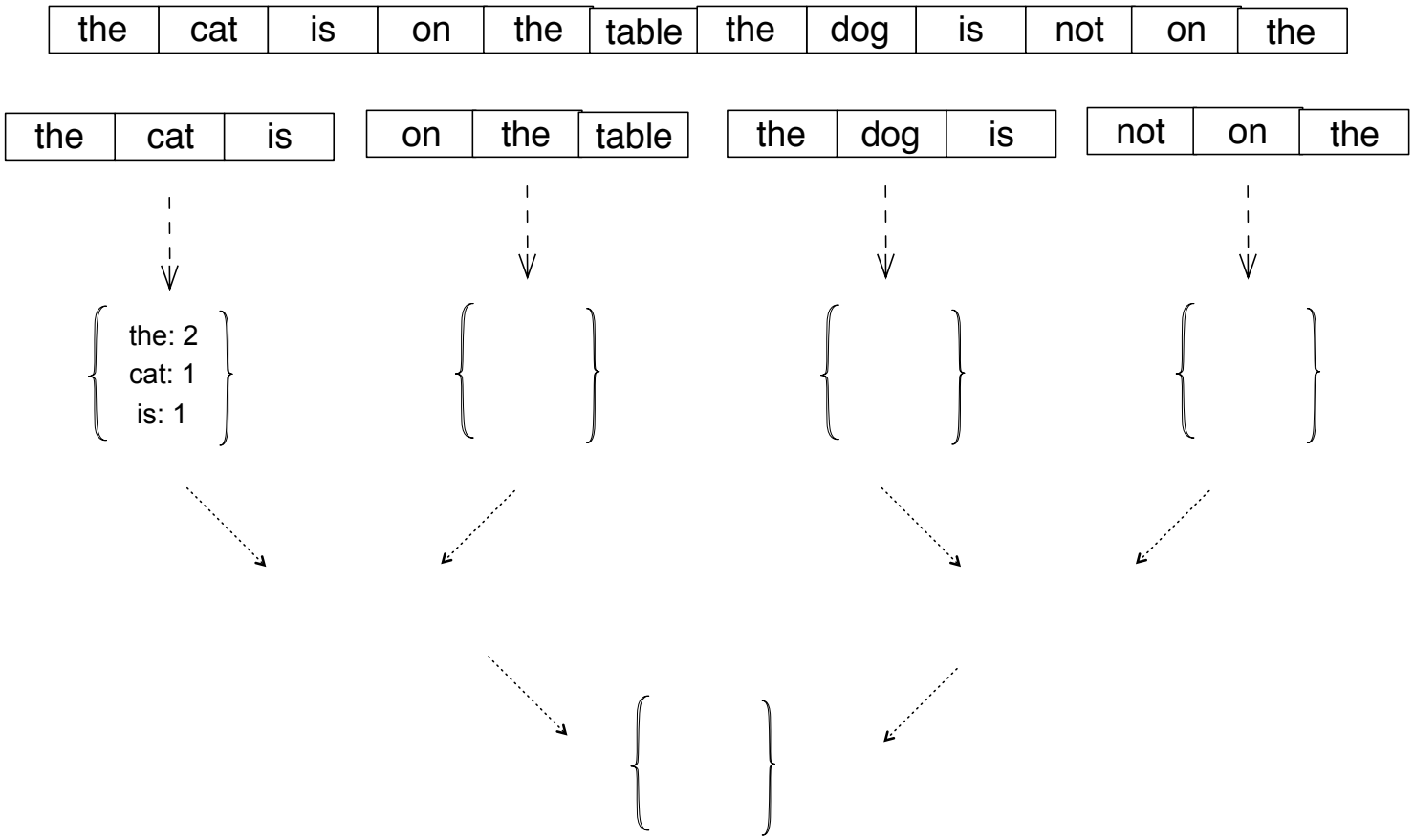
```
user> (def numbers (range 0 10000000))

user> (time (parallel-sum numbers))
"Elapsed time: 226.982582 msecs"

user> (time (recur-sum numbers))
"Elapsed time: 617.350992 msecs"
```

Counting words with fold

- Idea: From the sequence of strings construct a unique sequence of words and fold it with different functions:



Counting words with fold

- Idea: From the sequence of strings construct a unique sequence of words and fold with different functions

```
(defn counting [counts word]
  (assoc counts
         word (inc (get counts word 0))))

(defn count-words-fold [strings]
  (fold merge-counts counting
          (mapcat get-words strings)))
```

Summary

- A functional Lisp-based language compiled to the JVM
- Functional paradigm goes fine with parallel processing
- Map, Reduce, Fold naturally admit concurrent realisations

Laziness

Lazy sequences

- Sequences are lazy in Clojure, elements generated “on demand”
- Very long sequence of integers

```
=> (range 0 1000000)
```

- Generated on demand ...

```
=> (take 2 (range 0 1000000))
```

- Only two (actually a bit more) elements generated
- **`Doall`** for reifying a sequence

(Lazy) Streams

- Lazy sequences can be infinite
- Iterate: lazy sequence by iterating a function to an initial value

```
(def naturals (iterate inc 0))
```

```
(take 10 naturals)
(0 1 2 3 4 5 6 7 8 9)
```

- Repeatedly apply a constant function

```
(def rand-seq (repeatedly #(rand-int 10)))
```

Delay as much as you can

- When transforming a sequence the actual transformation is only "recorded"

```
(def numbers (range 1000000))
```

```
(def shift (map inc numbers))
```

```
(def doubleshift (map inc shift))
```

- Some real computation only if sequence accessed

```
(take 2 doubleshift)
```

What happens

- Conceptually each sequence **`seq`** is associated with a transforming function **`f`**.

```
<f,seq>
```

- Applying a function to the elements of the sequence just means composing with **`f.`**

```
(map g <f,seq>) -> < g∘f, seq>
```

- See also the concept of reducibles

Different orders

- Functions are referential transparent: an expression can be replaced by its value without changing the overall behaviour
- Different evaluation orders produce the same results

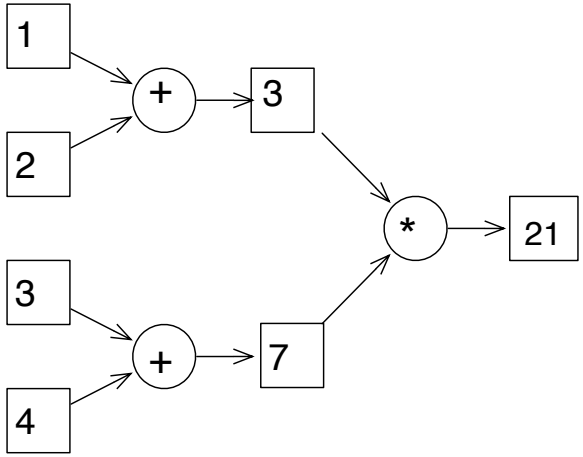
```
(reduce + (doall (map inc (range 1000))))
```

```
(reduce + (map inc (range 1000)))
```

```
(fold + (map inc (doall (range 1000))))
```

Self-made concurrency

- Independent functions – in principle - can be evaluated in parallel




```
(* (+ 1 2) (+ 3 4))
```

- Can we do this?

Future and Promises

Futures

- Intuitively: expression evaluated in a different thread

```
(future Expr)
```

- Value does not immediately exist, (might be) available at some later point
- Realised by an asynchronous concurrent thread

Futures

- Example

```
user=> (def sum (future (+ 1 2 3 4 5)))
#'user/sum
user=> sum
#object[clojure.core$future_call …]
```

- Deref (or @ for short): get the value

```
user=> (deref sum)
15
```

```
user=> @sum
15
```

Wait until the value is realised (available)

Timing Out and checking

- Possibility of timing out when waiting for a value

```
(deref ref tout-ms tout-val)
```

- Example

```
user=> (def sum (future (+ 1 2 3 4 5)))
user=> (deref sum 100 :timed_out)
```

- Checking if a future is realised

```
user=> (realized? sum)
```

Promises

- Placeholder for a value realised asynchronously

```
(promise)
```

- (Might be) later written (delivered) only once

- Again deref (@ for short) for getting the value, and realised? for checking availability

Promises

- Example

```
user=> (def answer (promise))
```

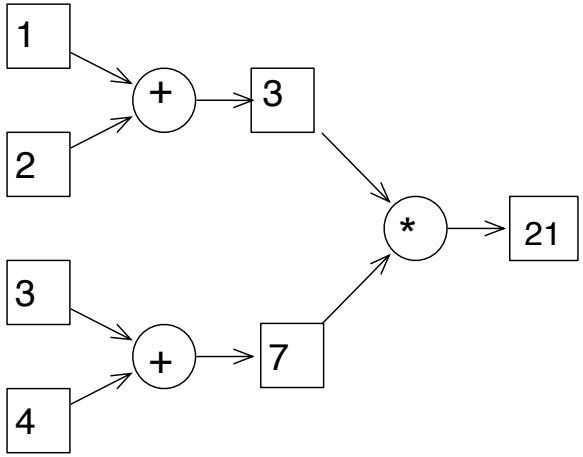
```
user=> (deref answer)
```

```
user=> (future
          (println "Waiting for answer ... ")
          (println "Here it is" @answer))

user=> (deliver answer 42)
```

Self-made concurrency

- Getting back




```
(* (+ 1 2) (+ 3 4))
```

```
(let [ a (future (+ 1 2))
       b (future (+ 3 4)) ]
     (+ @a @b))
```

- More generally they can be used to structure concurrent applications

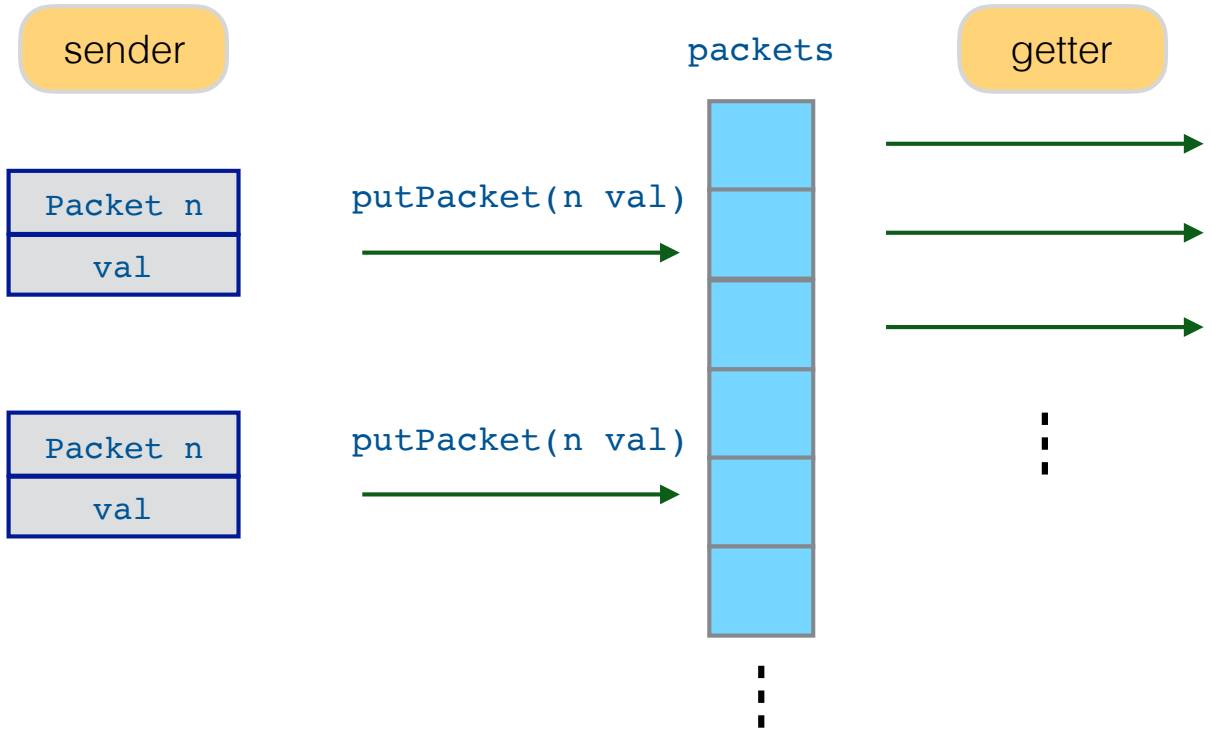
Example

- Call services and wait for the first result

```
(def search-engines
    {:bing   "https://www.bing.com/"
     :google "https://www.google.com/"})

(defn search [search-term & engines]
  (let [result (promise)]
    (doseq [engine engines]
      (future (deliver result
                       (slurp (str (engine search-engines)
                                   "search?q%3D"
                                   search-term)))))
    @result))
```

Example


sender
packets
getter
Packet n
val
putPacket(n val)
Packet n
val
putPacket(n val)

Receiver

```
; packets are in a lazy sequence of promises
(def packets (repeatedly promise))

; when a packet arrives, the promise is realised
(defn put-packet [n content]
  (deliver (nth packets n) content))

; process taking each packet as long as it is available
; and all its predecessesors have been realised
(defn getter []
  (future
    (doseq [packet (map deref packets)]
      (println (str "*** GETTER: " packet)))))
```

Sender

```
; simulate the sender: split str in words and randomly
; send the words until finished (possibly with repetitions)
(defn send-str [str]
  (let [words (get-words str)
        len   (count words)]
    (send-words words len)))
```

Send words

```
; process that randomly sends the words
; until all have been successfully sent
(defn send-words [words len]
  (future
  (loop [values-to-send len]
    (if (> values-to-send 0)
      (let [n (rand-int len)
            word (nth words n)]

        (if (nil? (put-packet n word))
          (do (println (str "* SENDER:" word " already sent"))
              (recur values-to-send))
          (do (println (str "* SENDER:" word " successfully sent"))
              (recur (dec values-to-send)))
          ))))))
```

Mutable state

Processes

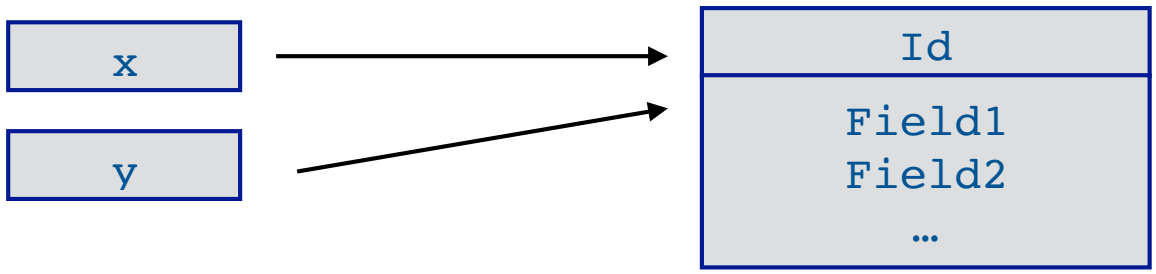
- Dealing with concurrency, the notion of process comes in
- Processes
  - Wait for external events and produce effects on the world
  - Answers change over time
  - Processes have **mutable state**

Identity and state

- **Identity**
  logical entity associated to a series of values
- **State**
  the value of an identity at some time

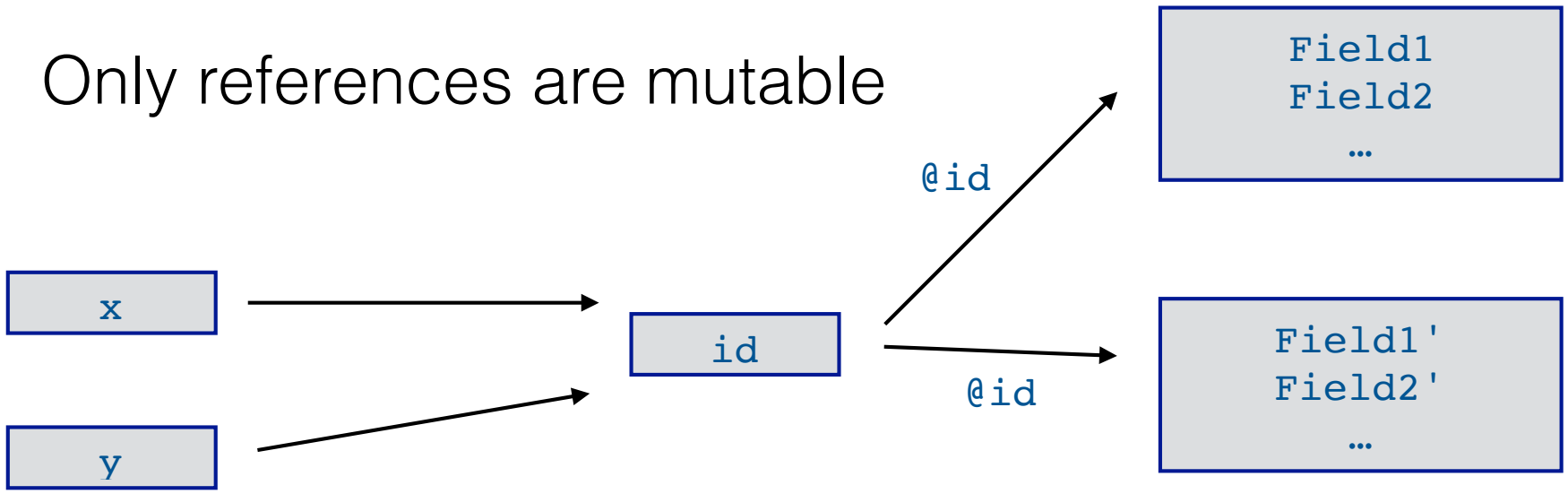
Imperative world

- Identity and state are mixed up
- The state of an identity is changed by **locally modifying** the associated value
- Changing state requires **locking**
- Risk of inconsistency



The clojure way

- **Identity** and **state** are kept **distinct**
- Symbols refer to **identities** that refers to **immutable values** (never inconsistent)
- Only references are mutable



Mutable references

- Only references change, in a controlled way
- Four types of mutable references:
  - **Atoms** - shared/synchronous/autonomous
  - **Agents** - shared/asynchronous/autonomous
  - **Refs** - shared/synchronous/coordinated
  - (**Vars** - Isolated changes within threads)

Atoms

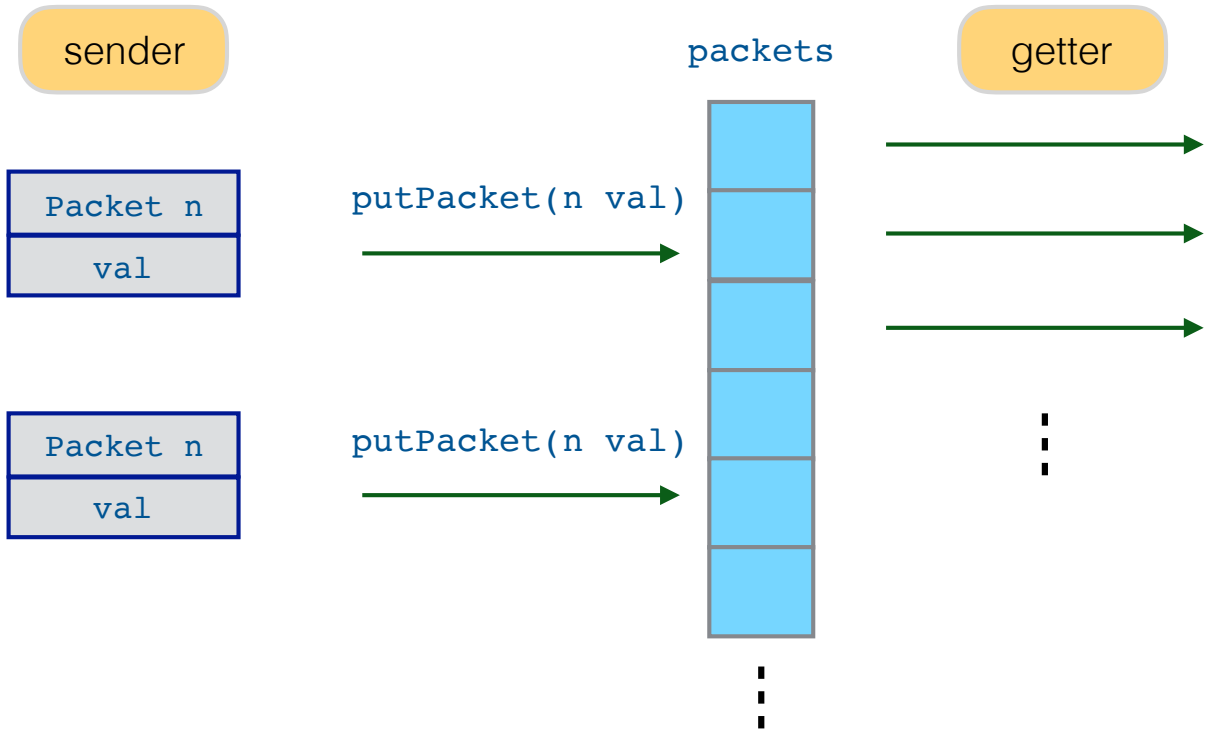
- Atomic update of a **single** shared ref
- Changes occur **synchronously** on caller
  - Change is requested
  - Caller “blocks” until change is completed

Updating Atoms

- `swap! atom f new`
  - a function computes the new value from the old one
  - called repeatedly until the value at the beginning matches the one right before change
- `reset! atom new`
  changes without considering old value
- `compare-and-set! atom old new`
  changes only if old value is identical to a specified value

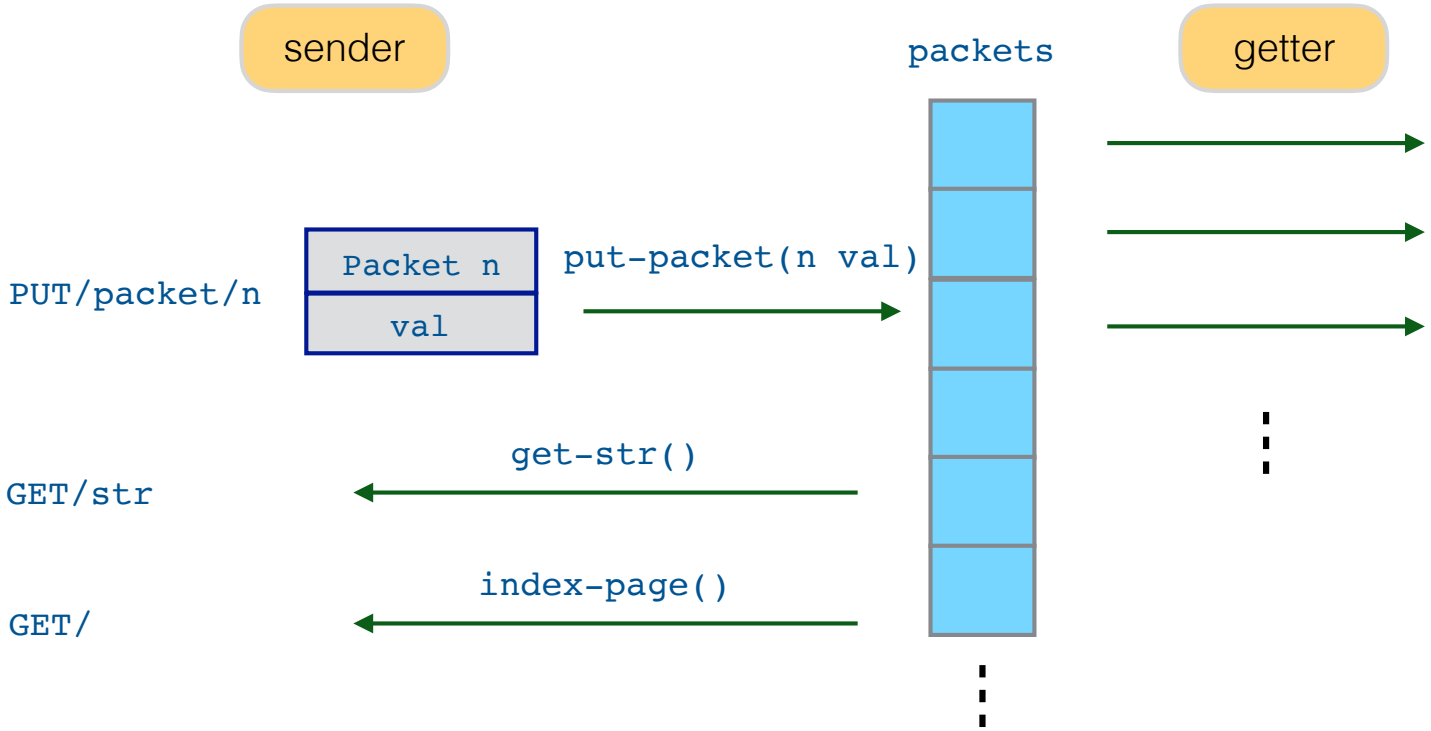
Example

- Consider the packet example



Example

- and turn it into a web service



Routes

```
(defroutes main-routes

  ; receive a packet
  (PUT "/packet/:n" [n :as {:keys [body]}]
       (put-packet (edn/read-string n)
                                (slurp body)))

  ; get the current string
  (GET "/str" [] (get-str))

  ; getting the current string as an html page
  (GET "/" [] (index-page))
```

Views

```
; state

; current collected string
(def msg (atom ""))

; packets in a lazy sequence of promises
(def packets (repeatedly promise))

; take each packet as long as it is available and all
; its predecessesors have been realised and join it
; the msg
(future
  (doseq [packet (map deref packets)]
    (swap! msg #(str % packet))))
```

Handlers

```
; handler for PUT/packet/n
; when a packet arrives, the promise is realised
(defn put-packet [n content]
  (if (nil? (deliver (nth packets n) content))
    "FAILED\n" "OK\n"))

; handler for GET/str
; client asking for the current string
(defn get-str [] (str @msg \newline))

; handler for GET/
; html page showing the current string
(defn index-page []
  (html5
   [:head [:title "Packets”" ... ]
   [:body [:h1
     (str "String:" @msg)]]))
```

Agents

- Atomic update of a **single** shared ref
- Changes occur **asynchronously**
  - Change is requested (via **send**) which immediately returns
  - Executed asynchronously (queued and sequentialised on target)
- Useful for performing updated that do not require coordination

Operating on Agents

- `send`
  - a function computes the new value from the old one
  - asynchronously
- `@, deref`
  access the current value (some updated possibly queued)
- `await`
  wait for completion

Example: Back to the server

- In-memory logging could be done via agent

```
; state

; log
(def log-entries (agent []))

....

; adding a log-entry
(def log [entry]
     (send log-entries conj [(now) entry]))
```

Handlers

```
; handler for PUT/packet/n
; when a packet arrives, the promise is realised
(defn put-packet-log [n content]
  (if (nil? (deliver (nth packets n) content))
    (do (log (str "Packet " n " duplicated"))
        "FAILED\n")
   "OK\n")
  )
```

Software Transactional Memory

- Software transactions ~ DB concept
  - **Atomic**
    From outside: either it succeeds and produce all side effects or it fails with no effect
  - **Consistent**
    From a consistent state to a consistent state
  - **Isolated**
    Effect of concurrent transactions is the same as a sequentialised version

Software Transactions in Clojure

- Based on **refs**

```
(def account (ref 0))
```

- Refs can only be changed within a transaction
- Changes to refs are visible outside only after the end of the transaction

Software Transactions in Clojure

- A transaction can be **rolled back** if
  - it fails
  - it is involved in a deadlock

- A transaction can be retried hence it should avoid side effects (on non refs)

Transactions on refs

- Transactions enclosed in **dosync**

```
; transfers some amount of money between two accounts
(defn transfer [from to amount]
  (dosync
   (alter from - amount)
   (alter to + amount)))
```

- Side effects (ops on atoms) shouldn't be there
- Operation on agents executed only upon successful try (agents work well with transactions)

Concluding …

- Functional paradigm pairs nicely with concurrency
- Clojure takes a pragmatic view, it is functional but with support to (controlled) mutable state
- Why don’t we work with imperative languages altogether then?
  - mutable state as an “exception”
  - actually, mutable refs to immutable values

Concluding

- Futures and promises
- Software Transactional Memory
- Sometimes we miss channel based concurrency … also Rich Hickey did (implemented from 1.5)
- No direct support for distribution and fault tolerance (but integration with Java … you have Akka there)