// Engineering Meetups

Core.async: Deep Dive

Andrey Listopadov,
Clojure Engineer at Health Samurai

About me

- Clojure engineer at Health Samurai
- Love asynchronous programming
- Interested in various programming languages
  - Helping with development of the Fennel programming language

My personal site

GitHub / GitLab: **@andreyorst**

What’ll this talk be about

- Asynchronous programming
  - When/Why?
  - History of async in Clojure
- clojure.core.async
  - Introduction to core.async
  - Implementation of go-threads
  - Implementation of channels
  - Timers
  - Event Loop/Scheduler
- Portability
- Problems

Asynchronous programming

- Async in a language either
  - Exists as a part of the language
  - Attached by some means
- Async in JVM
  - Green Threads (before JDK 1.1)
  - ...nothing?
  - Virtual Threads, since JDK21
- Async in other languages
  - BEAM - actor model
  - Go - CSP
  - .NET - async/await
  - Lua - coroutines

Parallelism VS Async – what's the difference?

- Parallelism
  - Useful mostly for compute tasks
  - Parallel execution
  - Limited by core count and their abilities
- Async
  - Useful for IO-bound tasks
  - Cooperative execution
  - Doesn't depend on core count (depends)
  - Limited by the acceptable time of reaction to events

Async in Clojure

- Agents
  - Agent is a “place” for storing modifiable data as agent state
  - Agent’s state is modified by sending tasks
  - Tasks - pure functions that accept current state and return a new state
  - Tasks are serialized in a queue, everything is thread-safe
- Third-party alternatives
  - Manifold – event-driven
  - Promesa – async/await
  - Probably some others…

Agents: Problems

- Tasks should remain short
- Blocking an agent can:
  - Exhaust agent's thread pool - no more tasks will run
  - Cause OOME by growing queue indefinitely, since there's no backpressure
- Agents have two separate thread pools:
  - Compute: unbound, for compute tasks
  - IO: bound, for IO tasks
- Error handling
  - In a complex network of messages it's often hard to find where exactly the error happened
  - Stacktraces are hard to understand
  - Managing agent's working state adds a bit of complexity to the code
- Can't be ported to all other supported runtimes!

clojure.core.async

- Universal async model for all supported runtimes
- CSP-based (Communicating Sequential Processes)
- Platform-independent asynchronous thread implementation
- Channels as synchronisation points, and process communication primitive

Channels

- A simple queue*
- Unbuffered:
  - Supports backpressure by default
- Buffered
  - Backpressure once buffer can’t accept more

- Different buffering strategies:
  - Fixed buffer: backpressure, if full
  - Sliding Window: **N** most recent messages
  - Dropping buffer: **N** oldest messages
  - Custom:
    - Drop random messages?
    - Drop **N** messages then enable backpressure?
    - Other crazy ideas

With a channel you can!

- Put data in! `>!`
- Get data out! `<!`
- Put data in, or block the thread!! `>!!`
- Get data out, or block the thread!! `<!!`

Blocking ops

- Blocks the thread currently executed in
- Better to avoid in asynchronous threads

```
(def ch (chan 10))
(>!! ch "Health Samurai") ;; puts "Health Samurai" into channel's buffer
(println (<!! ch)) ;; Prints Health Samurai to stdout
```

Parking ops

- Do not block anything
- Can be used only inside asynchronous threads
  - It’s a compile-time error to use parking ops outside of asynchronous threads

```
(def ch (chan))
(go
  (println (<! ch)))
(go
  (>! ch "Health Samurai"))
```

Asynchronous threads

- Platform independent implementation
- Lightweight
- Fast to create
- Behaves as a channel

go threads

- Go threads – are not threads at all
- **go** – is a macro, that transforms code inside to behave like a thread

```
(def ch1 (chan))
(def ch2 (chan))
(go
  (let [data (<! ch1)]
    (>! ch2 (foo data))))
```

go's output

```
(let [c (chan 1)
      captured-bindings (getThreadBindingFrame)
      f (fn state-machine
          ([] (aset-all! (AtomicReferenceArray. (int 6)) 0 state-machine 1 1))
          ([state]
           (let [old-frame (getThreadBindingFrame)
                 ret (try
                       (resetThreadBindingFrame (aget-object state 3))
                       (loop []
                         (let [result
                               (case (int (aget-object state 1))
                                 1 (take! state 2 ch1)
                                 2 (let [data (foo (aget-object state 2))]
                                     (put! state 3 ch2 data))
                                 3 (return-chan state (aget-object state 2)))]
                           (if (identical? result :recur) (recur) result)))
                       (catch Throwable ex
                         (aset-all! state 2 ex)
                         (if (seq (aget-object state 4))
                           (aset-all! state 1 (first (aget-object state 4)))
                           (throw ex))
                         :recur)
                       (finally
                         (aset-object state 3 (getThreadBindingFrame)) (resetThreadBindingFrame old-frame)))]
             (if (identical? ret :recur)
               (recur state)
               ret))))
      state (aset-all! (f) USER-START-IDX c BINDINGS-IDX captured-bindings)]
  (run-state-machine-wrapped state)
  c)
```

```
(def ch1 (chan))
(def ch2 (chan))
(go
  (let [data (<! ch1)]
    (>! ch2 (foo data))))
```

go‘s output

```
(let [c (chan 1) ;; ❶ creating a channel for the go block
      captured-bindings (getThreadBindingFrame) ;; ❷ saving the thread bindings
      f (fn state-machine ;; creating a state machine
          ([] (aset-all! (AtomicReferenceArray. (int 6)) 0 state-machine 1 1)) ;; ❸ state machine initialization code
          ([state]
           (let [old-frame (getThreadBindingFrame) ;; ❹ grabbing thread bindings before state machine runs
                 ret (try
                       (resetThreadBindingFrame (aget-object state 3)) ;; setting thread bindings that were stored in ❷
                       (loop [] ;; ❺ core of the state machine
                         (let [result
                               (case (int (aget-object state 1)) ;; ❻ obtaining the current state and jumping
                                 1 (take! state 2 ch1)
                                 2 (let [data (foo (aget-object state 2))]
                                     (put! state 3 ch2 data))
                                 3 (return-chan state (aget-object state 2)))]
                           (if (identical? result :recur) (recur) result))) ;; ❼ check if we need to park or not
                       (catch Throwable ex ;; ❽ some error handling
                         (aset-all! state 2 ex)
                         (if (seq (aget-object state 4))
                           (aset-all! state 1 (first (aget-object state 4)))
                           (throw ex))
                         :recur)
                       (finally ;; ❾ restoring thread-locals grabbed in ❹
                         (aset-object state 3 (getThreadBindingFrame)) (resetThreadBindingFrame old-frame)))]
             (if (identical? ret :recur)
               (recur state)
               ret))))
      state (aset-all! (f) USER-START-IDX c BINDINGS-IDX captured-bindings)] ;; ❿ initialization of the state machine
  (run-state-machine-wrapped state)
  c)
```

```
(def ch1 (chan))
(def ch2 (chan))
(go
  (let [data (<! ch1)]
    (>! ch2 (foo data))))
```

go macro

- Compiles the code into a state machine
    - Using the Inversion Of Control (IOC) strategy
    - Is reminiscent of a classic **generator** pattern with **yield**
- Channel operations are transformed into **callback**s
- Run by a custom scheduler

```
(defn put! [state blk c val]
  (if-let [cb (impl/put! c val
                       (fn-handler
                          ;; anonymous function, called when someone does take! on the channel c
                        (fn [ret-val]
                          ;; stores the result and the next state in the state machine itself
                          (aset-all! state 2 ret-val 1 blk)
                          ;; runs the state machine from the callback
                          (run-state-machine-wrapped state))))]
    (do (aset-all! state 2 @cb 1 blk)
        :recur)
    nil))
```

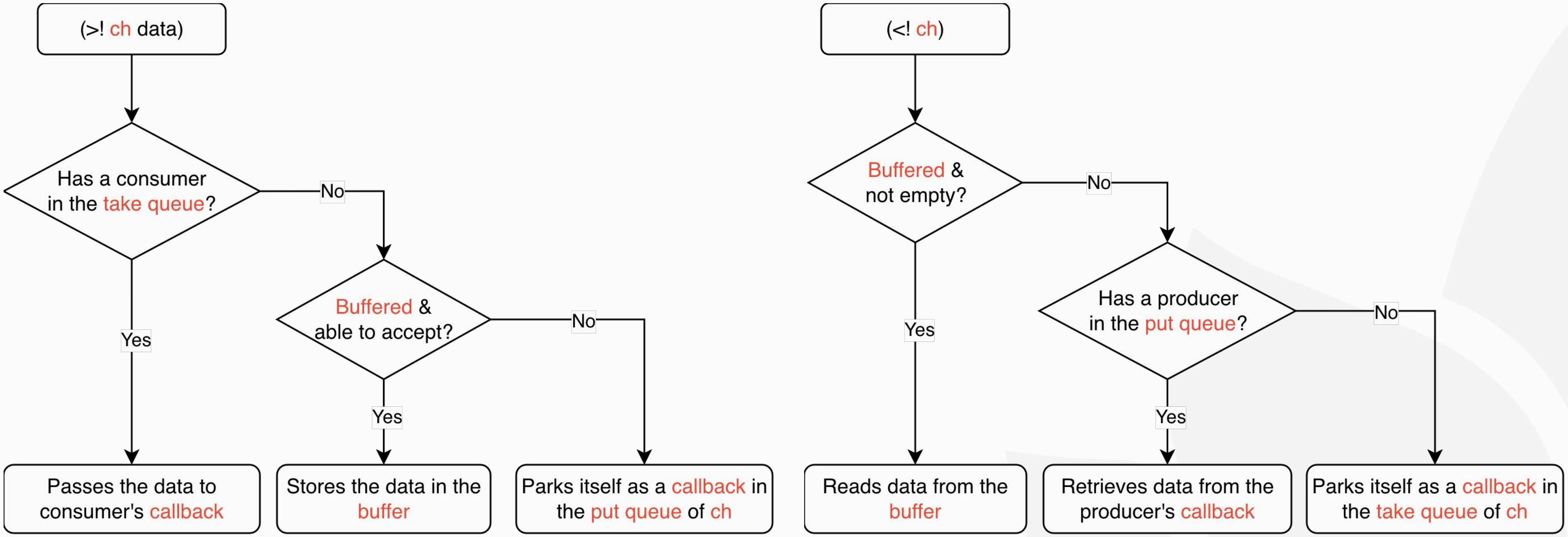
Behind the go macro: tools.analyzer

- A library for:
  - Parsing Clojure code in a host-agnostic way
  - Analysing the resulting AST
- Written in Clojure

Channels (again)

- Actually, not one, but three queues
  - Queue for **put** tasks
  - Queue for **take** tasks
  - Buffer for **data** (optional)

Channels visualized



put into a channel

take from a channel

Timers

- Timeout
    - A special type of channel
    - Automatically closed after set period
- Soft
    - timer resolution is 10ms
- shared
    - Timers created within 10ms from eachother use the same object in memory
    - Best not to close them manually, since it could be someone else’s timer too

```
(def timer
  (timeout 1000)) ;; closes after 1 second
```

Usage example: waiting with timeout

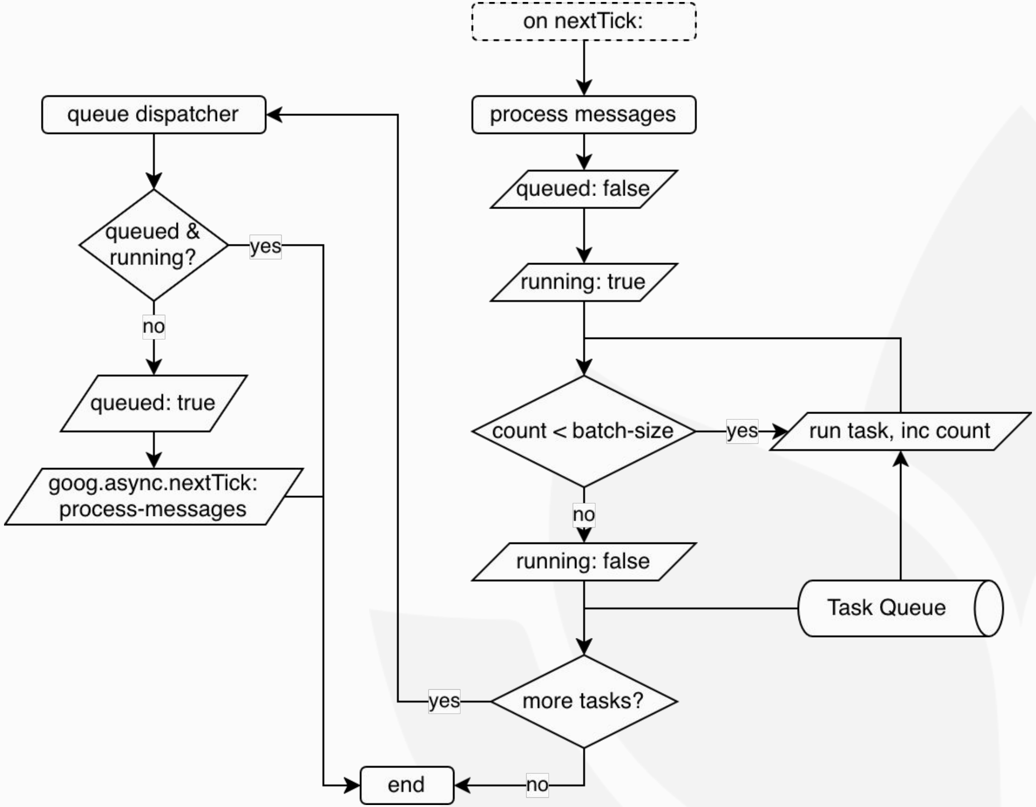
```clojure
(def ch (chan))

(go
  (let [t (timeout 1000)
        [data result-chan] (alts! [ch t])] ;; Using “select” on both channels, getting
                                           ;; the result and the channel it was returned from

    (cond (= result-chan t)   ;; if the result channel is the timeout channel,
          (println "timeout") ;; the timeout happened before we could receive data

          (= data nil) ;; if we got nil, then someone closed ch (since first check failed)
          (println ch "was closed")

          :else ;; successfully received data under one second
          (println data))))
```

Scheduler

- Platform dependent
  - ClojureScript: goog.async.nextTick
  - Clojure: thread pool

CLJS Scheduler

- Simple stop the world loop
- Only processes so many tasks (1024)
- Re-equeues itself if necessary
- Doesn’t run until requested
- Frequency determined by runtime



Do we need a scheduler?

- What scheduler does:
  - Watches over timeout channels
  - Pushes go-threads without external consumers or producers
- If thread pools are available:
  - Decrease reaction time in IO-bound tasks
  - Allows mixing parking and blocking code
- However, scheduler is optional if:
  - Fully reactive approach
  - No threads without external dependencies are allowed
    - (only in single-threaded systems)

Example: required vs optional scheduler

```clojure
(go
  (<! (timeout 1000))
  (println "done"))
```

No external consumers
(scheduler needed, otherwise thread will never complete)

```clojure
(def ch (chan))

(go
  (<! (timeout 1000))
  (>! ch "done"))

(println (<!! ch))
```

External consumer exists
(no need for scheduler, can work in a reactive manner)

Portability

- Basing go-threads on macros and state machines gives us:
  - Portability across wide range of runtimes
  - No dependencies on platform-threads
  - Ability to manually schedule tasks
- Channels give us:
  - No dependencies on platform-specific async primitives (promises, actors, etc)
  - Clear way to introduce discrete points of separation to our code to create state machines

Problems

```clojure
;; Clojure

(def ch (chan))

(defn my-async-task []
  (let [data (<! ch)] ;; compile error: can't use <! outside of go-thread
    (do-stuff data)))

(go (my-async-task))
```

```go
// Go

var ch = make(chan int)

func myAsyncTask() {
    data := <-ch // no problems (backed by runtime)
    doStuff(data)
}

func main() { go myAsyncTask() }
```

```fennel
;; Fennel (my port of core.async)

(local ch (chan))

(fn my-async-task []
  (let [data (<! ch)] ;; no problems (coroutines)
    (do-stuff data)))

(go (my-async-task))
```

Problems

```clojure
;; Clojure (state-machines)

(def ch1 (chan))
(def ch2 (chan))
(def ch3 (chan))

(go
  ;; compile error:
  ;; can't use “syntax” <! as a function
  (map <! [ch1 ch2 ch3]))

(go
  (map
   ;; compile error: can't transform lazy loop
   ;; into state machine
   (fn [c] (<! c))
   [ch1 ch2 ch3]))
```

```fennel
;; Fennel (coroutines)

(local ch1 (chan))
(local ch2 (chan))
(local ch3 (chan))

(go
  ;; OK:
  ;; <! is a function, we're good
  (map <! [ch1 ch2 ch3]))

(go
  (map
   ;; Also OK:
   ;; can yield across function calls
   (fn [c] (<! c))
   [ch1 ch2 ch3]))
```

Conclusion

- Pros:
  - Using macros as means for thread building allows great portability
  - State-machines allow for platform-independent way to start and resume code
  - Channels allow us to specify where parking happens
- Cons:
  - Nuances and unexpected behavior due to the lack of platform support
  - Wrappers may be required if we want to interface with other asynchronous solutions (virtual threads, callbacks)

Thanks!

slides

team.health-samurai.io