Vertiefung Funktionale Programmierung: Clojure

Programmierübung 4: core.async

Philipp Körner

Institut für Informatik
Heinrich-Heine-Universität Düsseldorf

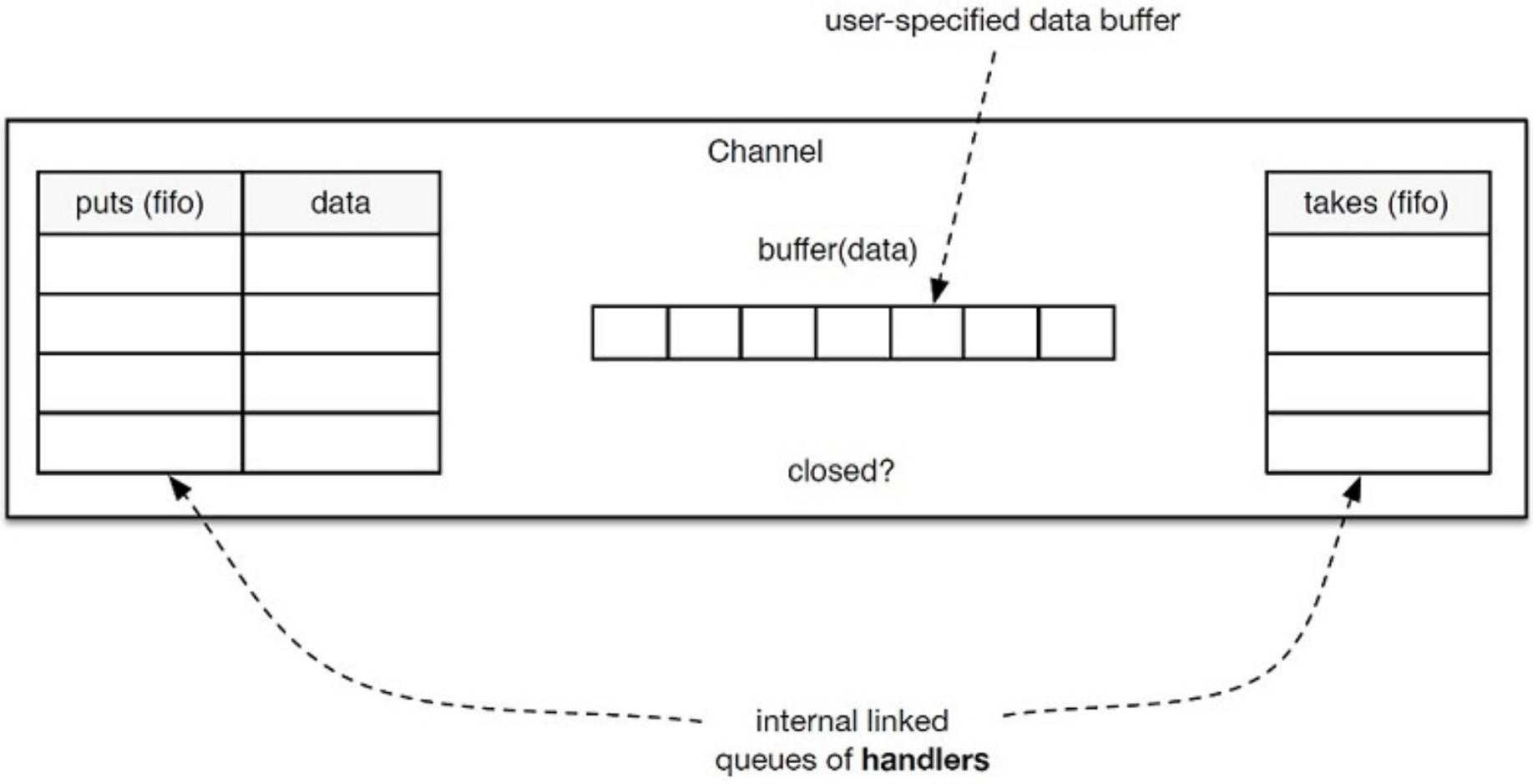
13. Mai 2024

Lernziele

Ihr solltet in der Lage sein,

- Kommunikation via core.async Channels zu erklären und zu implementieren.
- verschiedene Buffer-Strategien zu benennen und zu erklären.
- die Anatomie eines Channels zu erklären.
- zu erklären, was passiert, wenn ein Channel geschlossen wird.
- zu entscheiden, wann man core.async einsetzen sollte.

Anatomy


user-specified data buffer
Channel
puts (fifo)
data
buffer(data)
takes (fifo)
closed?
internal linked
queues of **handlers**

Basics: Werte annehmen

Schreiben Sie eine Funktion `(receive-n c n)`, die *n* Elemente von einem channel nimmt und diese als seq zurück gibt.
Beispiel:

```
user=> (let [c (async/chan)]
           (dotimes [i 10]
             (async/thread
               (Thread/sleep 10)
               (async/>!! c i)))
           (receive-n c 10))
(2 3 0 1 5 6 4 7 8 9)
```

Achtung: Die Reihenfolge kann sich hier ändern

Geteilte Ressourcen verwalten

Koordinieren Sie die geteilte Ressource (stdout), indem Sie core.async Channel verwenden.

```
(do (future (dotimes [x 100] (println "(..." x "...)")))
    (future (dotimes [x 100] (println "(..." x "...)"))))
```

Warten

Schreiben Sie ein Makro (wuäh!) `(do-or-timeout n body)`, das versucht den body zu evaluieren. Falls dies nach `n` Millisekunden noch nicht gelungen ist, soll `:timeout` zurückgegeben werden. Sie können davon ausgehen, dass der body nicht `nil` zurück gibt.

```
user=> (do-or-timeout 1000
                      (do (Thread/sleep 100)
                          (inc 1)))
2

user=> (do-or-timeout 1000
                      (do (Thread/sleep 1001)
                          (inc 1)))
:timeout
```

Multiplexer

Implementieren Sie die Funktionen `(mult c)` und `(tap a c)`. `mult` gibt ein Objekt zurück, dass als erstes Argument in `tap` weitergereicht wird. Alle Elemente auf `c` sollen dann auf allen tappenden Channels weitergereicht werden.

```
(def c (chan))  (def out1 (chan))  (def out2 (chan))

(def a (mult c))
(tap a out1)  (tap a out2)

(go (>! c 42))
(<!! out1) ;; 42
(<!! out2) ;; 42
```