Inside core.async Channels

Rich Hickey

Warning!

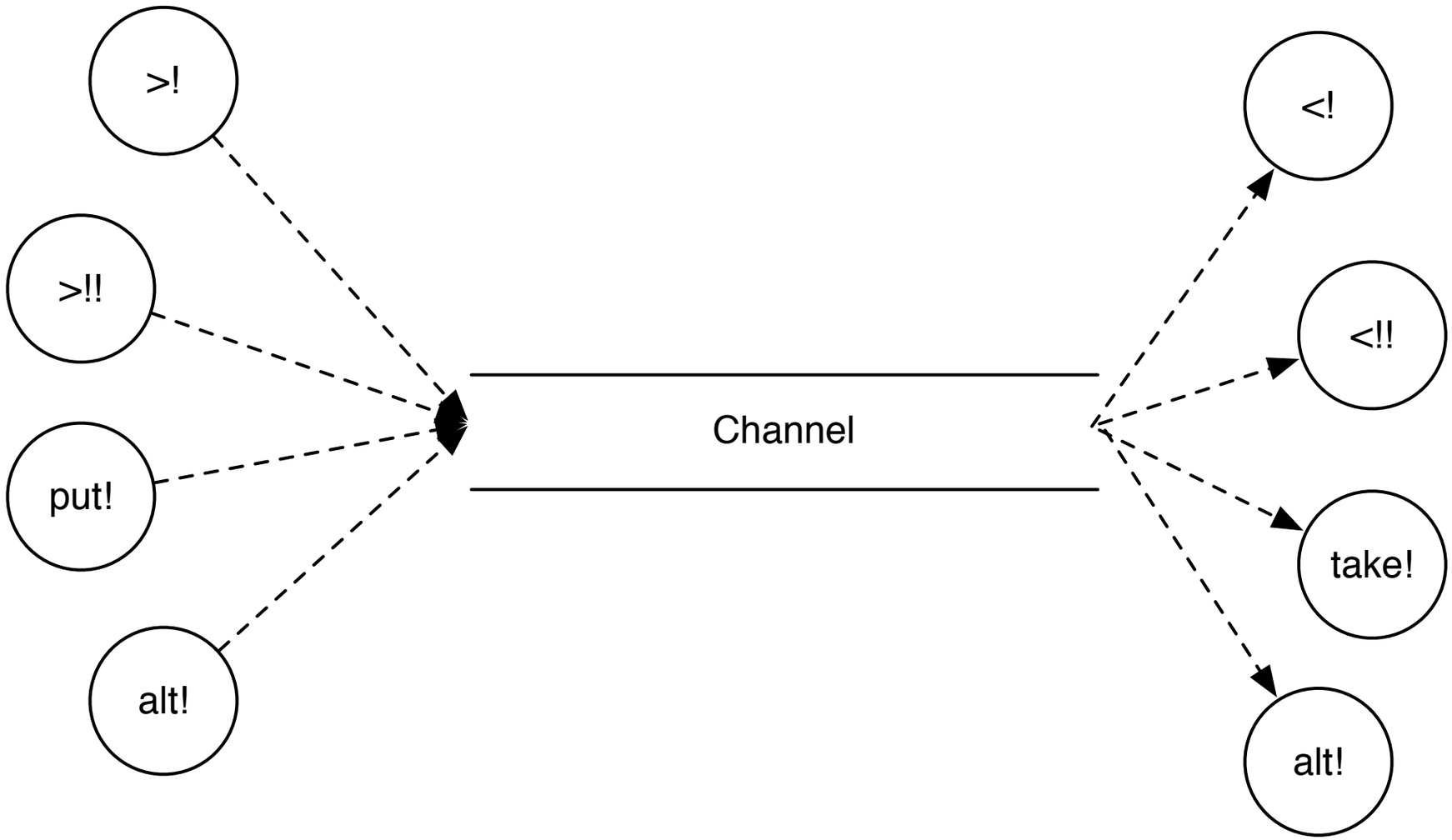
- Implementation details
- Subject to change

The Problems

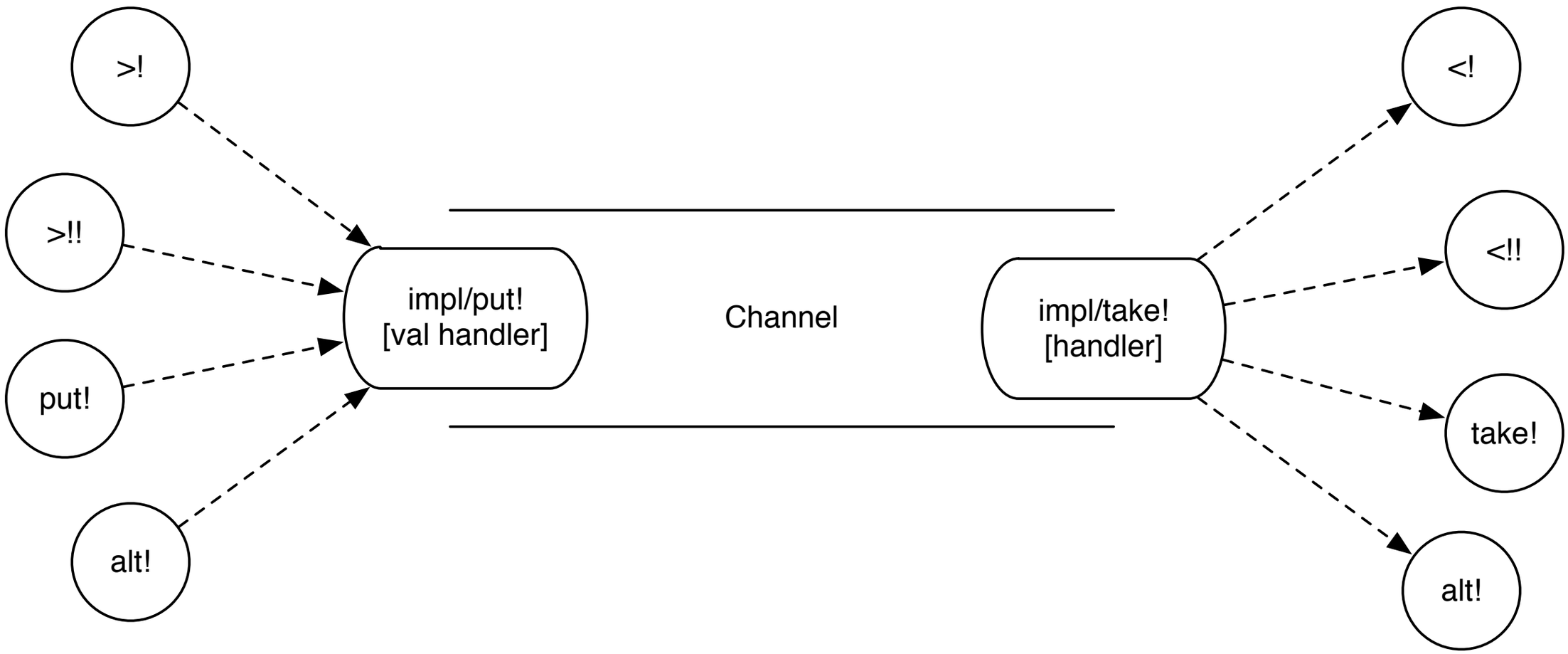
- Single channel implementation
- For use from both dedicated threads and go threads

  simultaneously, on same channel
- alt and atomicity
- multi-read/write
- concurrency

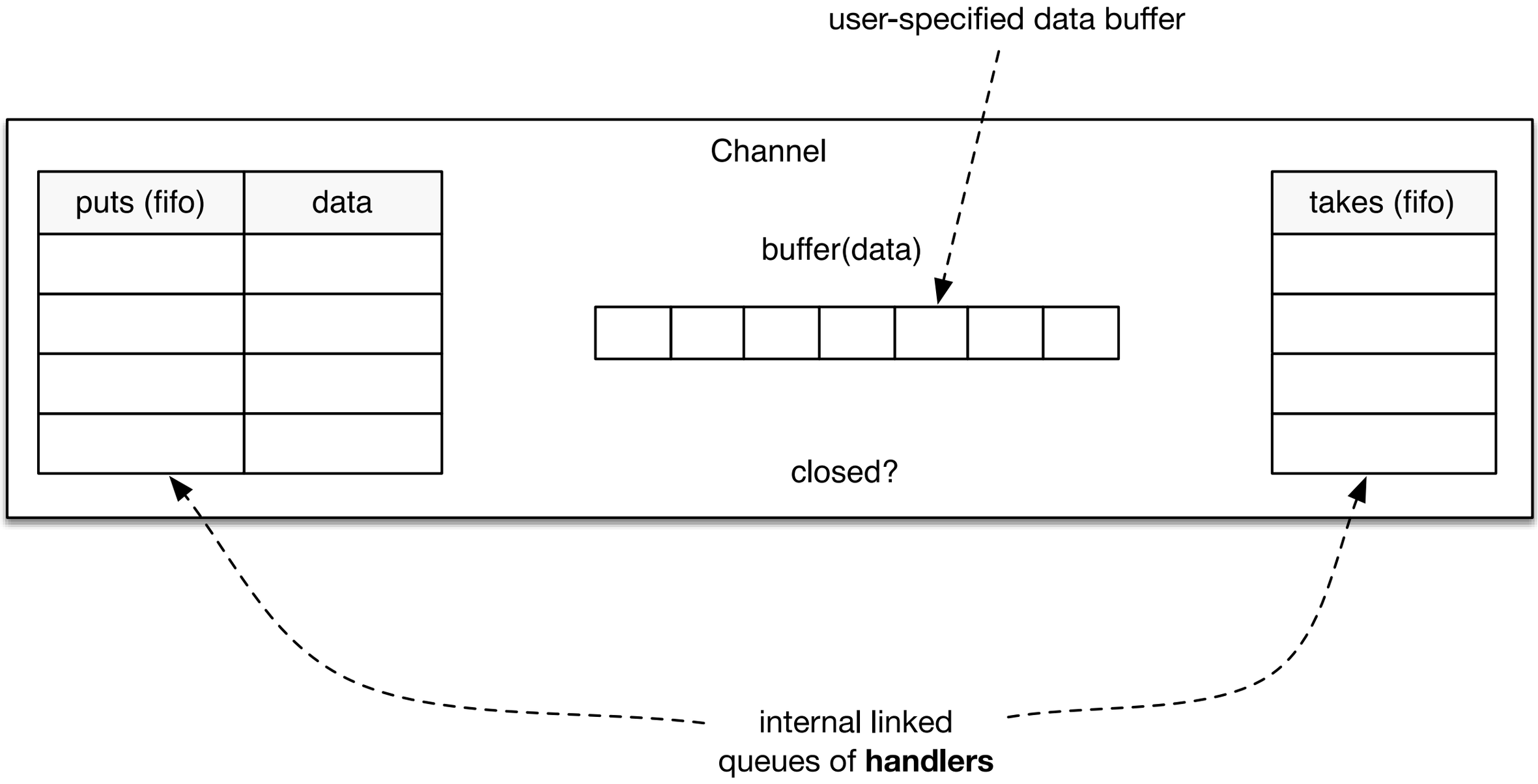
API



SPI


>!
>!!
put!
alt!
impl/put!
[val handler]
Channel
impl/take!
[handler]
<!
<!!
take!
alt!

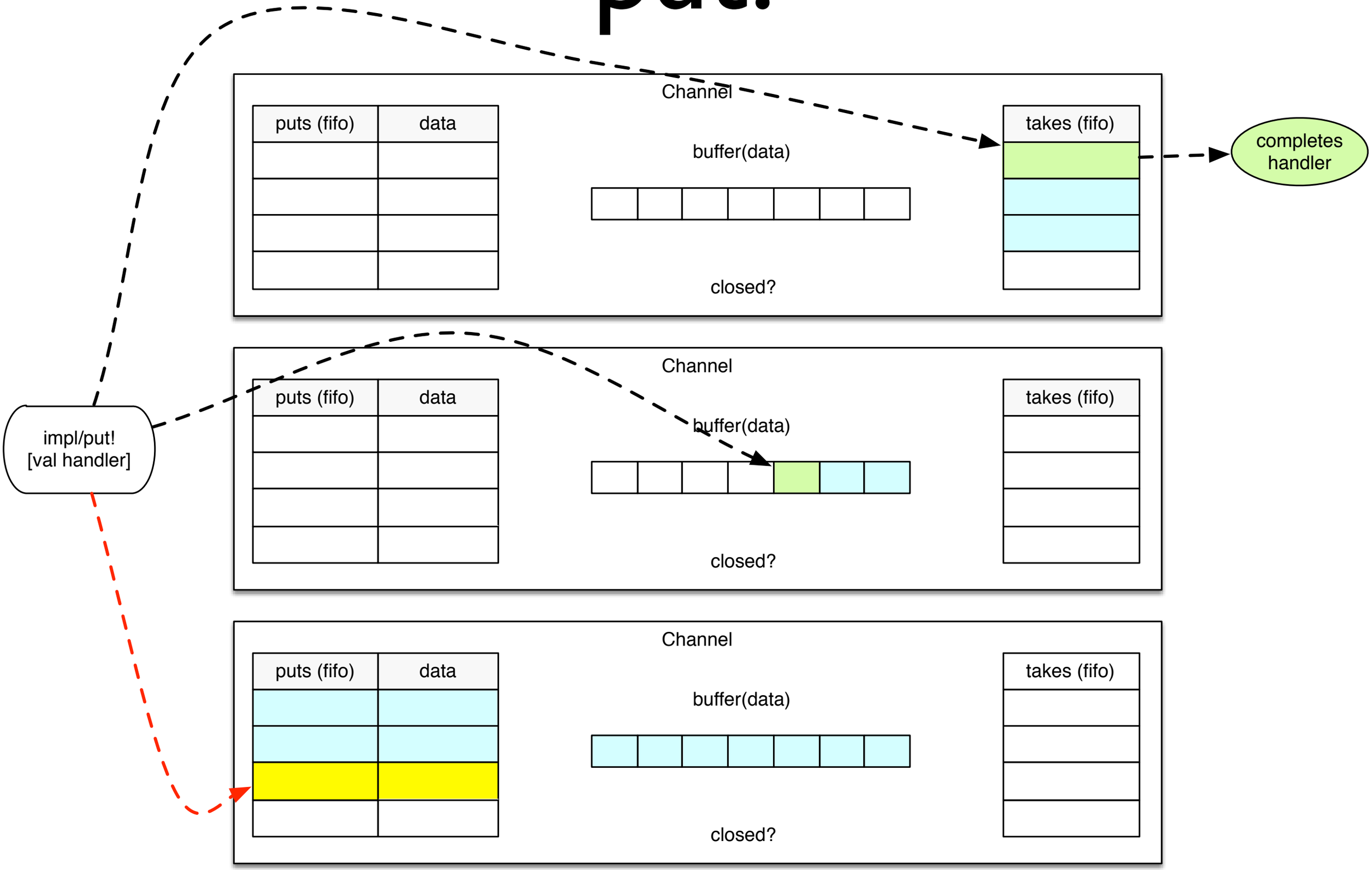
Anatomy


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

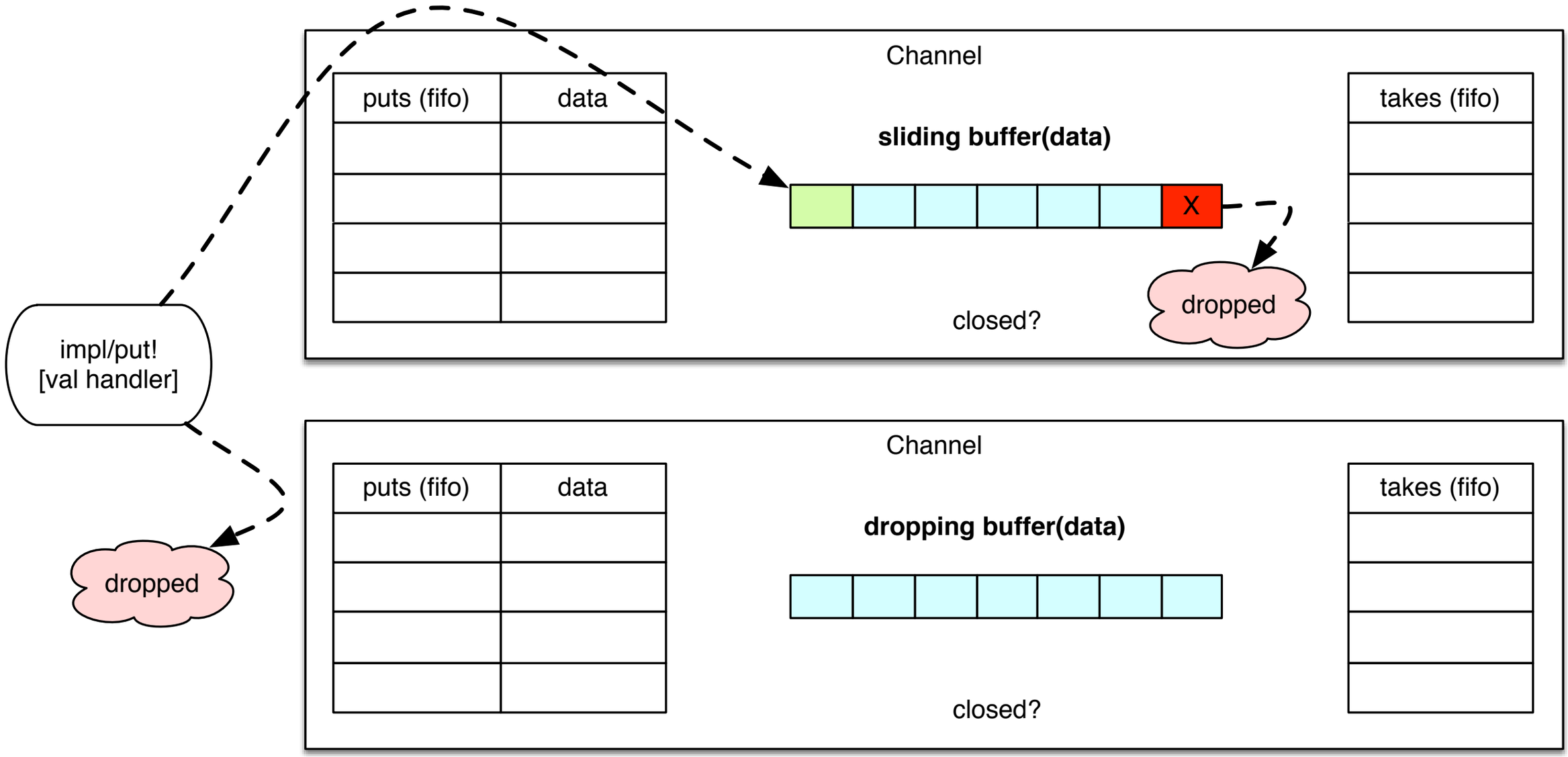
Invariants

- Never pending puts and takes
- Never takes and anything in buffer
- Never puts and room in buffer
- take! and put! use channel mutex
- no global mutex

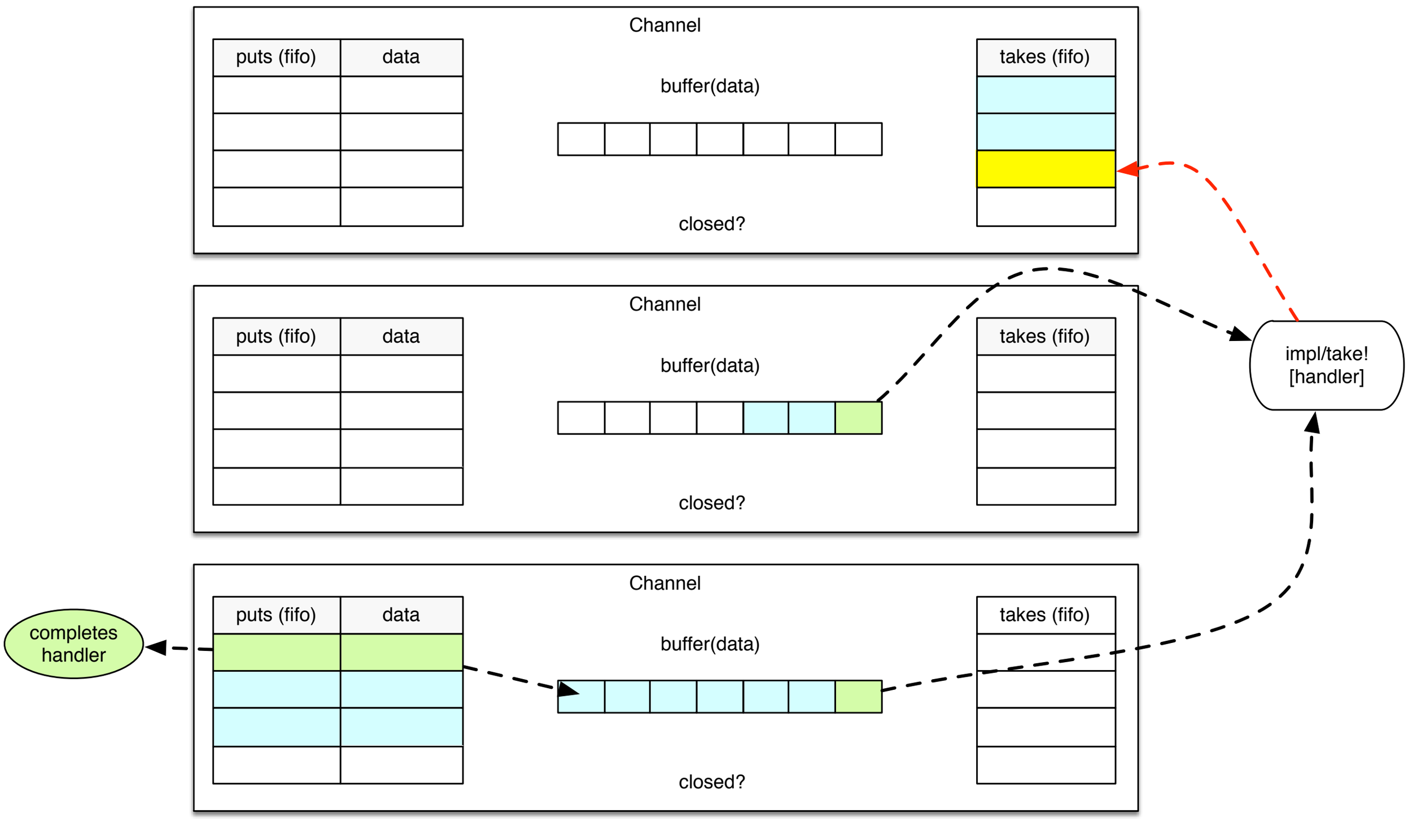
put!


Channel
puts (fifo)
data
buffer(data)
takes (fifo)
closed?
completes handler
impl/put! [val handler]
Channel
puts (fifo)
data
buffer(data)
takes (fifo)
closed?
Channel
puts (fifo)
data
buffer(data)
takes (fifo)
closed?

put! - windowed buffers



take!


Channel
puts (fifo)
data
buffer(data)
closed?
takes (fifo)
impl/take!
[handler]
completes
handler

close!

- all pending takes complete with nil (closed)
- subsequent puts complete with nil (already closed)
- subsequent takes consume ordinarily until empty

  any pending puts complete with true

  takes then complete with nil

Queue Limits

- puts and takes queues are not unbounded
- 1024 pending ops limit
- will throw if exceeded
- not for buffering, use buffers/windowing

alt(s!!)

- attempts more than one op
- on more than one channel
- without global mutex

  nor multi-channel locks
- exactly one op can succeed

alt implications

- registration of handlers is **not** atomic
- completion might occur before registrations are finished

  or any time thereafter
- completion of one alternative must ‘disable’ the others

  atomically
- cleanup

Handlers

- Wrapper around a callback
- SPI
  - active?
  - commit -> callback-fn
  - lock-id -> unique-id
  - java.util.concurrent.locks.Lock: lock, unlock

take/put handlers

- simple wrapper on callback
- lock is no-op
- lock-id is 0
- active? always true
- commit -> the callback

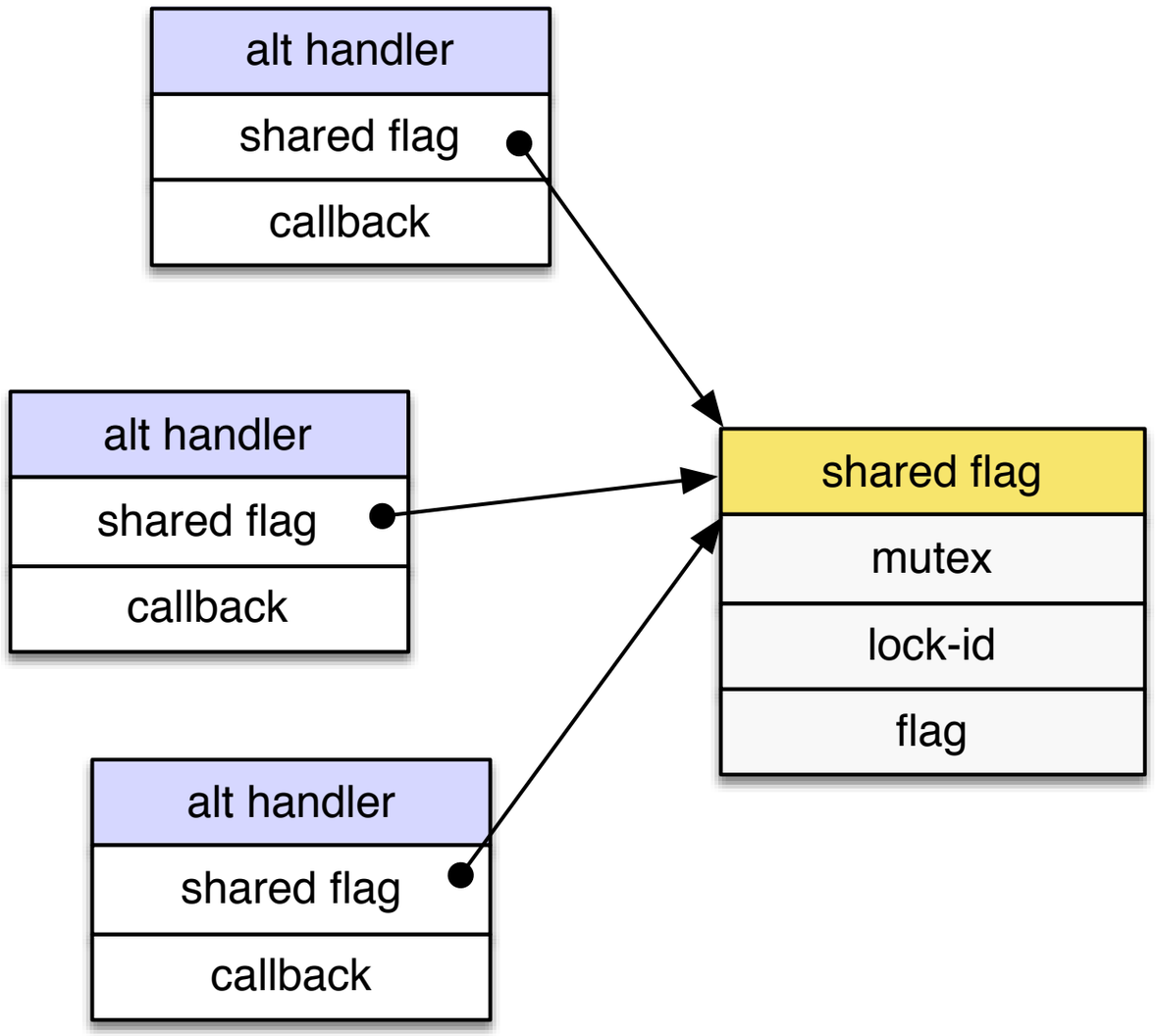
alt handlers

- each op handler wraps its own callback, but delegates rest to shared ‘flag’ handler
- flag handler has lock

  a boolean active? flag that starts true and makes one-time atomic transition
- commit transitions shared flag and returns callback

  must be called under lock

alt handlers



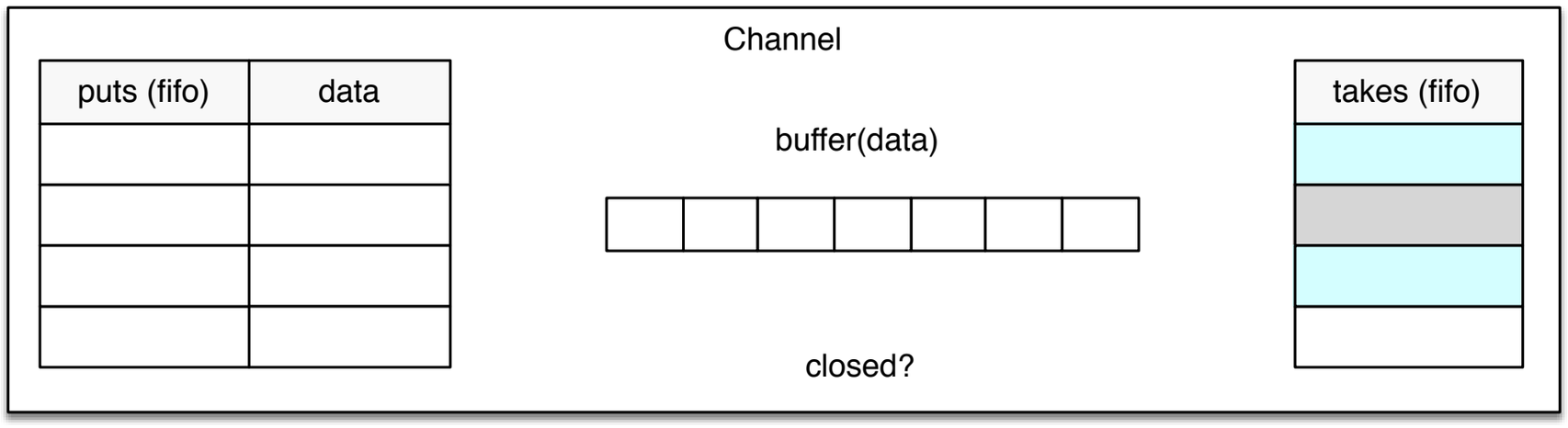
alt concurrency

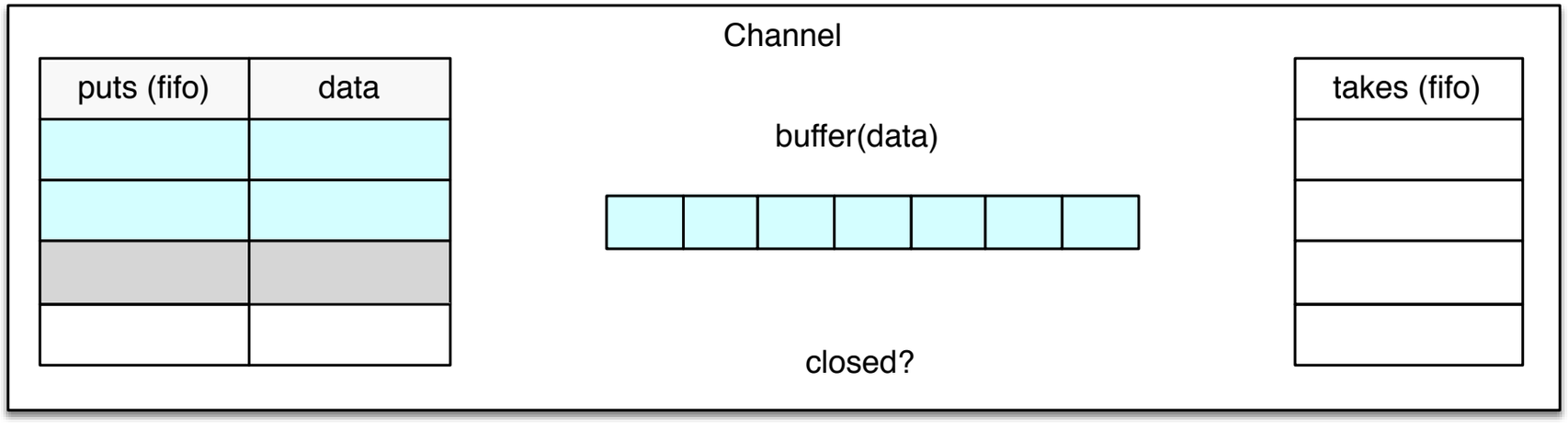
- no global or multi-channel locking
- but channel does multi-handler locking

  some ops commit both a put and take
- lock-ids used to ensure consistent lock acquisition order

alt cleanup

- ‘disabled’ handlers will still be in queues
- channel ops purge

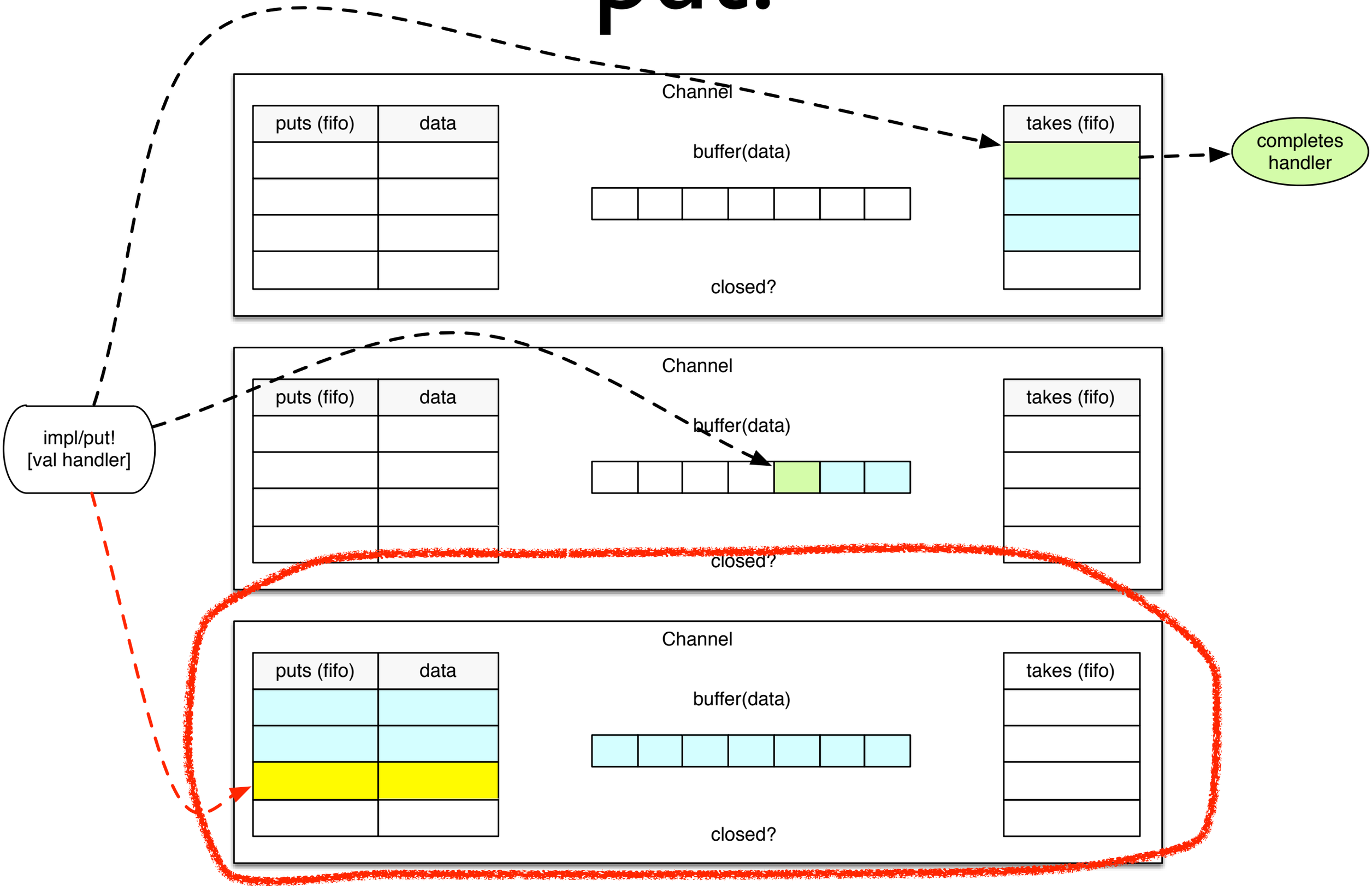




SPI revisited

- handler callback only invoked on async completion
  - only 2 scenarios
- when not ‘parked’, op happens immediately
  - *callback is not used*
  - *non-nil return value is op return*

put!


Channel
puts (fifo)
data
buffer(data)
takes (fifo)
closed?
completes handler
impl/put! [val handler]
Channel
puts (fifo)
data
buffer(data)
takes (fifo)
closed?
Channel
puts (fifo)
data
buffer(data)
takes (fifo)
closed?

take!


Channel
puts (fifo)
data
buffer(data)
closed?
takes (fifo)
impl/take!
[handler]
completes
handler

Wiring !/!!

- blocking ops (!!)

  create promise

  callback delivers

  only deref promise on nil return from op

- parking go ops (!)

  IOC state machine code is callback

Summary

- You don't need to know any of this
- But understanding the 'machine' can help you make good decisions