More Actors with Tokio

How to not `Arc<Mutex<T>>`

and *why*

Prelude

Alan Kay on OOP

1998

> I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea. The big idea is "messaging".

Alan Kay, message to the Squeak list, 1998, as archived on c2.com

Quoted via picostitch.com/blog/2015/05/11-alan-kay-on-messaging

2003

> OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things. It can be done in Smalltalk and in LISP. There are possibly other systems in which this is possible, but I'm not aware of them.

Alan Kay, email to Stefan Ram, 23 July 2003

Source: purl.org/stefan_ram/pub/doc_kay_oop_en

Context:
Communicating Applications

What's Communication?

"When *they* see us doing *this*, *we* expect *them* to do *that*".

Lots of expectations going on:

- `curl` wttr.in will answer me in some way
- `curl` wttr.in will tell me the weather
- `curl` wttr.in uses this and that format

By nature, communication is unreliable, strange,
... and asynchronous.

We expect our systems to be...

- Open to talk (system boundary).
- Closed to modification (hiding of state).
- Resilient against internal crashes.
- Resilient against external nonsense.
- Fallible with sanity.
- Capable of finishing business and then leaving cleanly.

How do we achieve these properties?

It's clearly and evidently hard to do!

The distributed-ness of the real world infects every participant. We want things to be apart, but share state regardless.

`Arc<Mutex<Resource>>` and threads is one option.

Let's explore a different one: tasks dedicated to different purposes, connected by channels.

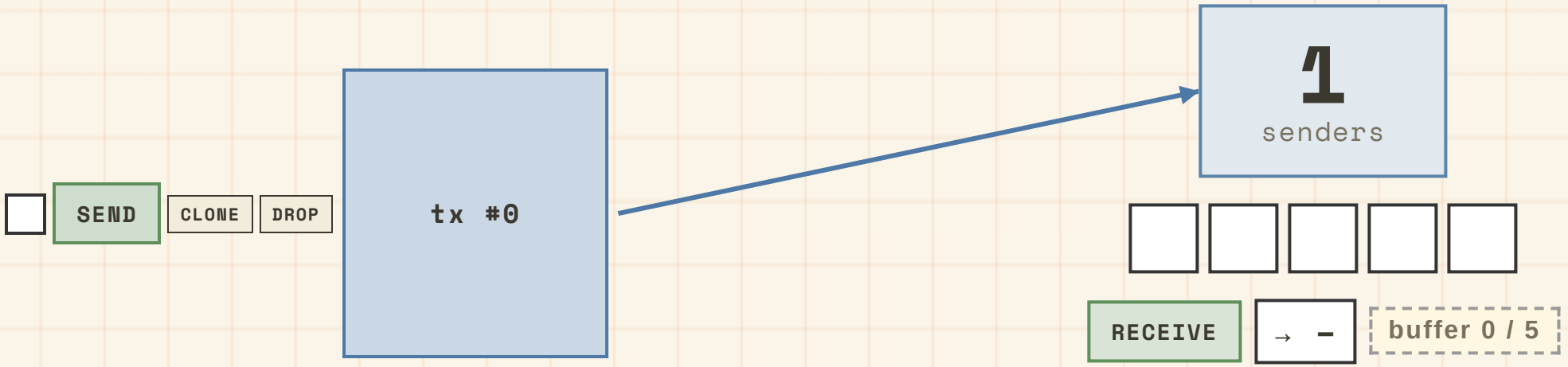
Prerequisites

`mpsc` (Multiple-Producer Single-Consumer)

- Many senders to one receiver.
- Messages are stored in a bounded buffer.
- A full buffer parks the sender until there is room. Backpressure!
- Once every sender is gone and the buffer is empty: `recv()` returns `None`.

That last line is very relevant.

mpsc



RESTART

CREATE THE CHANNEL

tx

Sender<char>

—

the channel

empty

rx

Receiver<char>

—

Ownership transfer over channel boundary

`select!`

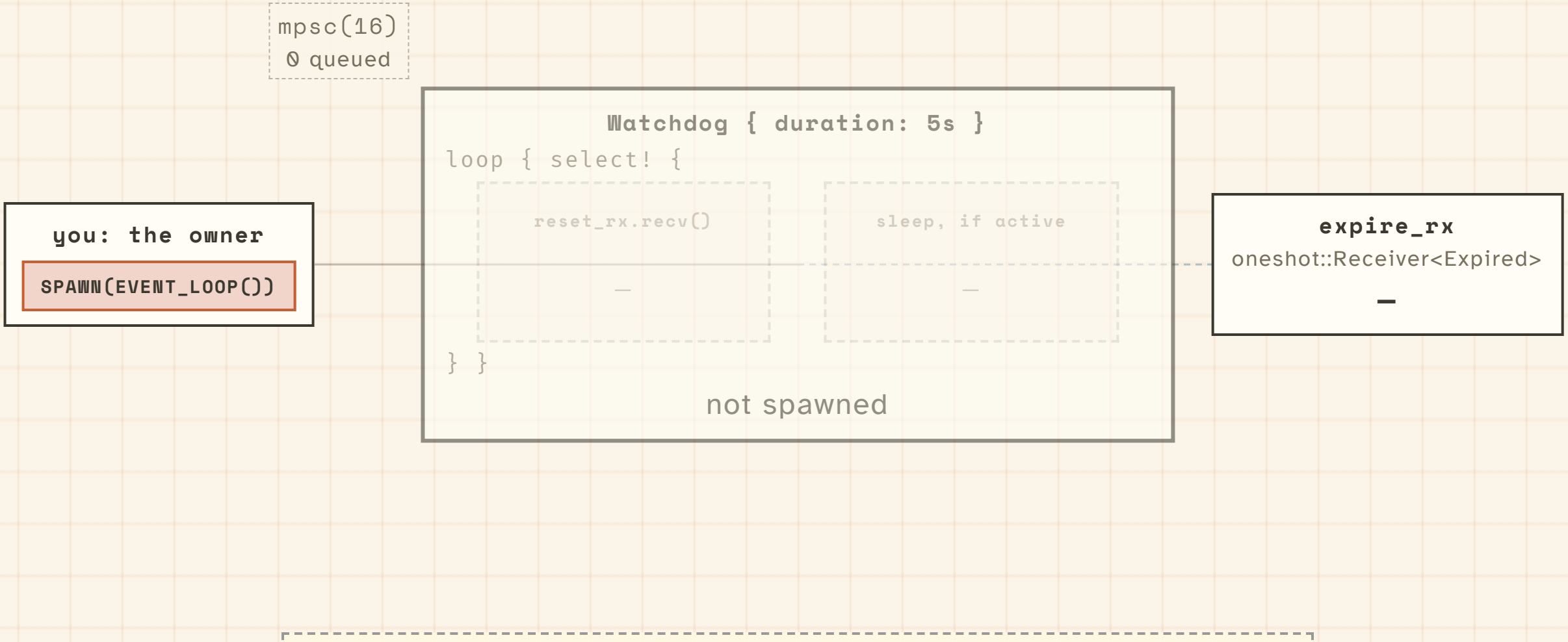
Futures combinator. Races multiple futures until one wins; "winner" gets to run some code.

The remaining futures are dropped as they are!

```
tokio::select! {
    msg = rx.recv() => { /* <deal with message> */ }
    _ = tokio::time::sleep(timeout) => { /* nobody wrote in time */ }
}
```

There's only ever one winner arm. Consequence: winner gets free access to local variables (including `self`).

Actors by Example



tokio::spawn(Watchdog::with_timeout(5s).event_loop()): you spawn it. Then keep it fed.

Watchdog

```
let sleep = sleep(self.duration);
tokio::pin!(sleep);
let mut active = true;
loop {
    tokio::select! {
        msg = reset_rx.recv() => match msg {
            Some(Signal::Reset) => {
                active = true;
                sleep.as_mut().reset(Instant::now() + self.duration);
            }
            Some(Signal::Stop) => active = false,
            None => break,
        },
        () = sleep.as_mut(), if active => {
            let _ = expire_tx.send(Expired);
            break;
        }
    }
}
```

click an actor to step it

pipeline

reader · urls.csv

- crates.io
- quickref.me
- meet.jit.si
- berline.rs
- this-week-in-rus…
- rustacean-statio…
- lib.rs

for record in csv

downloader

step select!

FuturesUnordered · 0/3

processor

step select!

FuturesUnordered · 0/2

collector

Vec::new()

while let Some(..)

mpsc · pages 0/2

mpsc · responses 0/2

mpsc · resources 0/2

warn!: failed, dropped

nothing yet

Click the reader to read a line of urls.csv, then a stage to run one turn of its select!.

channel size 2 · limits 3 and 2 (the crate: 64 and 64) · timeout simulated

RESET

Pipeline

- Each page goes through each actor stage, as resources permit.
- Concurrency within a stage can be capped: `FuturesUnordered` with "in-flight" OPs.
- A slow stage fills the channel before it. The stage before parks. Pressure travels "upstream".
- Shutdown flows downstream, with the data: drop the first sender, then every stage drains and ends.

The Pattern

An actor *is* its data

```
#[derive(Default)]
struct KvStore {
    data: HashMap<String, String>,
}
```

- Plain state. No receiver, no socket, no handle inside.
- The `data` could otherwise be an `Arc<Mutex<HashMap<...>>>`.
- Initial resources arrive as arguments to the actor construction.
- No channel handles or other runtime resources in the actor struct!

Messages own their data

```
enum Message {
    Set { key: String, value: String },
    Get { key: String, reply: oneshot::Sender<Option<String>> },
}
```

- One enum per actor, one variant per thing it can be asked.
- The message *owns* the data it carries.
- `Set` is direct. `Get` hands reply channel to actor.

The event loop is a consuming method

```
impl KvStore {
    async fn event_loop(
        mut self,
        mut rx: mpsc::Receiver<Message>,
    ) -> Self {
        while let Some(msg) = rx.recv().await {
            self.handle(msg);
        }
        self
    }
}
```

- Takes `self` by value. Gives it back when `recv()` yields `None`.

Leave the spawning to the user

```
let (tx, rx) = mpsc::channel(8);
let handle = tokio::spawn(KvStore::default().event_loop(rx));
```

- Or `.await` it directly. Or put it in a `JoinSet`. Or run it in a test.
- The actor does not know, and must not care, where it runs.

No actor handles

```
async fn get(
    tx: &mpsc::Sender<Message>,
    key: String,
) -> Option<oneshot::Receiver<Option<String>>> {
    let (reply, response) = oneshot::channel();
    tx.send(Message::Get { key, reply }).await.ok()?;
    Some(response)
}
```

- `mpsc::Sender` gives you: `closed()`, `try_send`, `reserve`, cloning.
- `Set` needs no helper: `tx.send(Message::Set { .. })`.
- You may want to use an extension trait here.

Testing with spawning and sleeping

```
#[tokio::test]
async fn should_get_what_was_set() -> Result<()> {
    let (tx, rx) = mpsc::channel(8);
    tokio::spawn(KvStore::default().event_loop(rx));
    tx.set("a", "1").await?;
    tokio::time::sleep(Duration::from_millis(100)).await;
    let a = tx.get("a").await?;
    let b = tx.get("b").await?;
    assert_eq!(a.await?, Some("1".into()));
    assert_eq!(b.await?, None);
    Ok(())
}
```

- joinhandle dropped: actor could end early or panic.
- 100ms wait: giant antipattern, may fail.
- Task swallows state (cannot be asserted after).

Testing *without* spawning or sleeping

```rust
#[tokio::test]
async fn should_get_what_was_set() -> Result<()> {
    let (tx, rx) = mpsc::channel(8);
    tx.set("a", "1").await?;
    let a = tx.get("a").await?;
    let b = tx.get("b").await?;
    drop(tx);
    let store = KvStore::default().event_loop(rx).await;
    assert_eq!(a.await?, Some("1".into()));
    assert_eq!(b.await?, None);
    assert_eq!(store.map.len(), 1);
    Ok(())
}
```

No spawn, no mock, no `sleep(100).await`. Deterministic.

Architecture

Good stuff

- **Backpressure.** The pipeline: a stage parked on a full channel.
- **Graceful shutdown.** The pipeline's `None` cascade.
- **Parallelism, free.** Every actor already runs on its own task. The topology is the parallelism plan.
- **No locking over actor data.** The actor owns its state; there is nothing to hold across an `.await`.

Costs

- **Channel hop.** Pretty quick, but different performance characteristics than a mutex.
- **Complexity.** Async, channel APIs, tasks, futures, etc.
- **Who has the receiver?** Can be confusing (low coupling).
- **Boxing.** To keep message type itself small (clippy).
- **Deadlock risk.** Two actors sending to each other over bounded `mpsc`s *seize* once both inboxes are full.

Alleviate

- **Batch.** `recv_many` drains the inbox in one go, amortising per-message overhead.
- **DAG.**
  - Prevents deadlocks.
  - Enables natural shutdown.
  - Nice to draw (easy to reason about).

Natural shutdown

- The loop ends when `recv()` says `None`.
- On the way out, the loop
  - returns its resources (connections, subscriptions, …).
  - drops its senders to the next actor.

Drop the *root* sender, and the whole graph drains.

Two actors in an `mpsc` cycle *will* keep each other alive!
Even a single actor with a self-sender does that.

Beyond `mpsc`

	reaches	buffer full / receiver asleep	in a cycle
`mpsc`	the one receiver	sender waits	can seize
`mpmc`	one of the receivers	sender waits	can seize
`broadcast`	every receiver	drops oldest, `Lagged(n)`	cannot seize
`watch`	every receiver	overwrites, latest only (there's no buffer)	cannot seize
embassy `PubSubChannel`	every subscriber	`publish().await` waits, `publish_immediate()` drops oldest	sender decides

Channel topology determines system architecture.

The only thing better than an actor

no actor

When designing, I advise to keep cutting actors.

Actors are a very nice tool for a very niche job.

Closing

There is no Rust actor framework

and it's called tokio

- Everything the pattern needed was `std` and tokio.
- Surprisingly often, the primitive building blocks (`select!`, `mpsc`) fit in with the actor pattern.
- Many actor frameworks out there are mediocre abstractions over `tokio::spawn`.
- The zoo of channels available allows extreme flexibility.

1997

> I have apologized profusely over the last twenty years for making up the term object-oriented, because as soon as it started to be misapplied, I realized that I should have used a much more process-oriented term for it.

Alan Kay, OOPSLA keynote "The Computer Revolution Hasn't Happened Yet"

In the same keynote: "did not have C++ in mind" when talking about OOP (specifically inheritance).

Source: archive.org/details/AlanKayAtOOPSLA1997TheComputerRevolutionHasntHappenedYet

2003

Kay	Actors with tokio
only messaging	channels are the only interface
local retention and protection and hiding of state-process	the task boundary: while the loop runs, nothing outside can name the state
extreme late-binding of all things	`Sender<Message>` is like a vtable.

Let's just call the objects tasks.

Thanks

- Alice Ryhl: "Actors with Tokio".
  If you read one article, read hers!
- The ideas from this talk: "More Actors with Tokio".

Appendix: more channels, `broadcast` and `watch`