Intro to Asynchronous Programming with `async`/`.await` in Rust

State Machines all the way down!

Rust in 3 Minutes

- Modern (primarily) systems language
- Strict type system enforces correct usage of shared resources
- Strict compiler is getting friendlier all the time (but still strict)
- Friendly and extensible tooling (test, bench, metrics, dependencies, linting, …)
- Dependency management: lots of small, interoperating libraries (some big ones around too, but no Qt)

Rust in 3 Minutes – Goals and Tradeoffs

- Binaries and Performance like C
- Type System inspired by Haskell (Hindley–Milner)
- Ergonomics inspired by Python
- Tooling like Javascript/node.js
- “Unique” learning curve
- Compile–times like C++ (sometimes worse)

www.rust-lang.org

Rust in 3 Minutes – Guarantees

- No SEGFAULTS
  - Panic: Structured Deconstruction
- No Undefined Behaviour
- No Data Races
- Zero-Cost Abstractions

www.rust-lang.org

Asynchronous Programming in 3 Minutes

I/O can take arbitrarily long time:

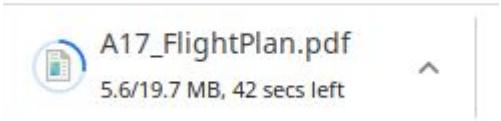
- Download
- UI action (click)
- Bytes on sockets
- Signals, Events
- TCP Server, waiting on clients

I/O bound: Programs which spend most of their time waiting on the real world

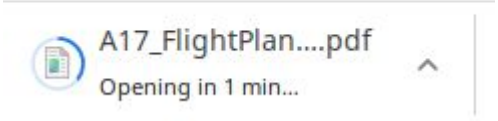
Asynchronous Programming in 3 Minutes

Oft: I/O bound Prozess beschreibt Sequenzen von Schritten, zwischen denen Zeit vergehen kann.

Einfache Warten-Operation



Sequenz: Warten + Aktion



Beispiel im Rust Playground

Control Flow is a Resource

- A blocking, waiting function is wasting control flow (busy waiting)

Goal of Asynchronous Programming

- Multithreading/Multiprocessing
- Cooperative Multitasking: Tasks yield control voluntarily
- Blocking operations are natural yield points

Foundations:
Futures,
State Machines,
Runtime Environments

What's a "Future"?

A Future ist just a value which will exist.

- Declaratively describes an action, but does no work (**lazy**)
- **Polling** will lead to the value, eventually
  - A future is either **`Pending`** or **`Ready(T)`**
- **No Busy-Polling** required:

  Future states event upon which it wants to be polled again

What's a "Future"?

```
enum Poll<T> {
    Pending,
    Ready(T),
}

trait SimplifiedFuture {
    type Output;
    fn poll(&mut self, waker: &mut Waker) -> Poll<Self::Output>;
}
```

Futures as State Machines

State: Waiting on event

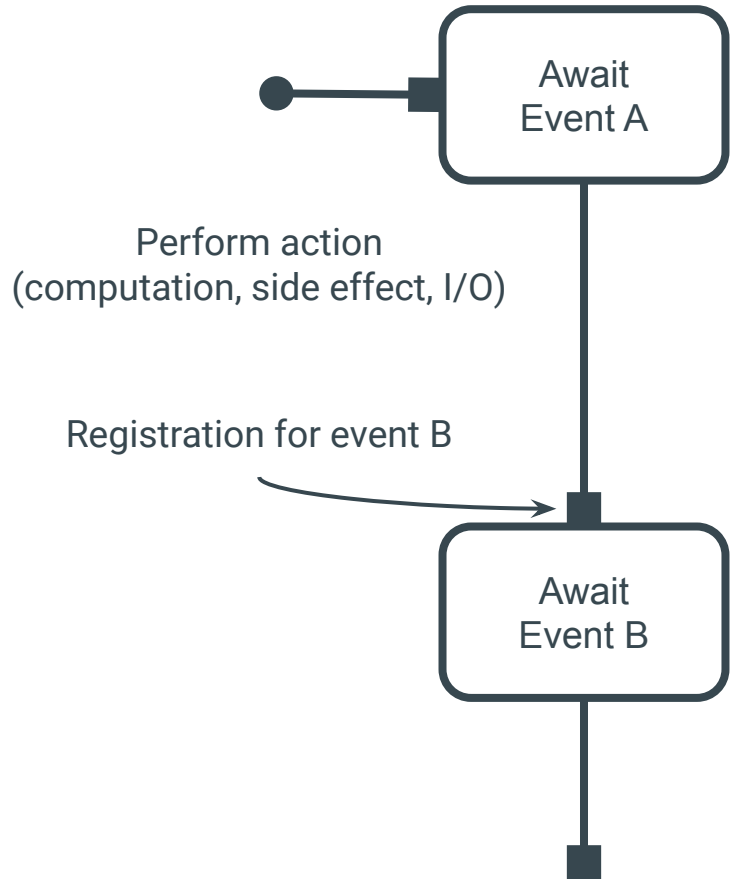
Transition:

Side effect, action, calculation

On state entry:

Registration for wake-event

(at event loop of a runtime environment)

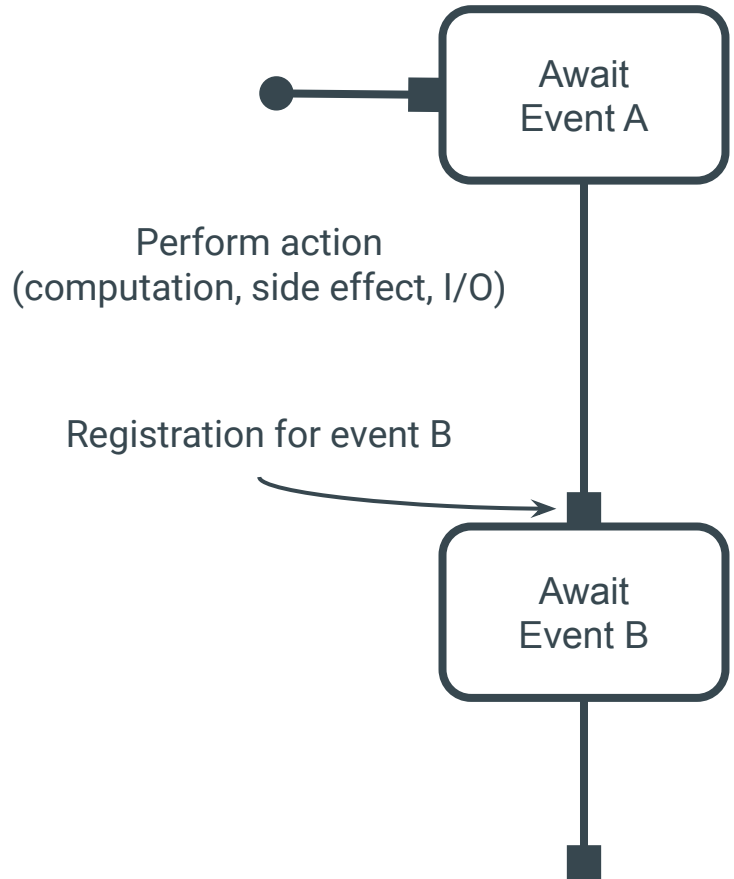


Futures as State Machines

```
async fn example(a: A) {
    let b = a.await;
    info!("Future 'A' completed!");
    b.await;
}
```

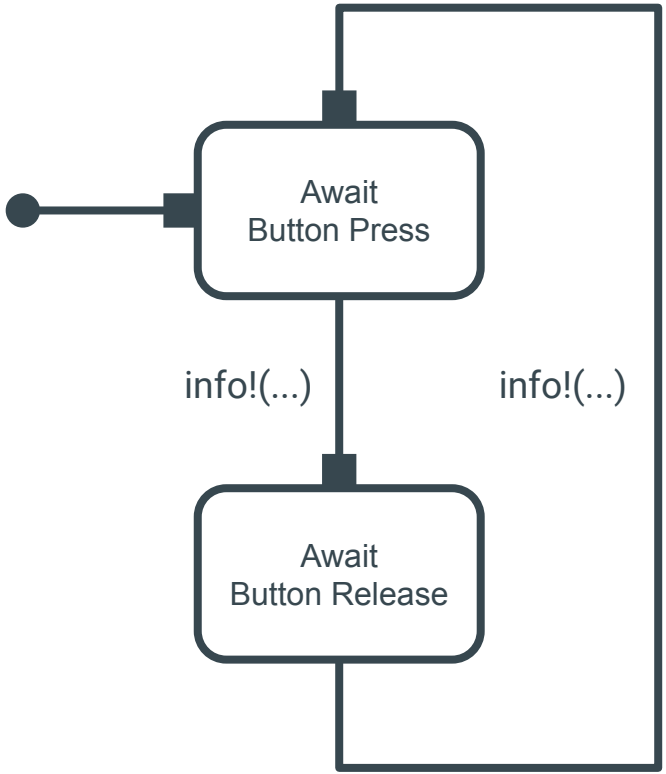
async creates a Future

.await consumes a Future



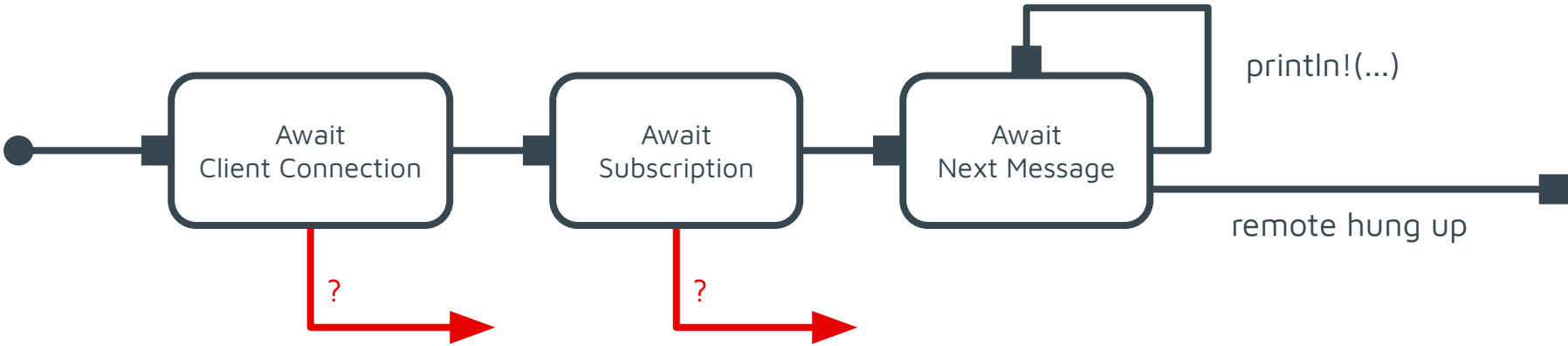
Looping on a Button

```
async fn print_button(button: Input) {
    loop {
        button.until_press().await;
        info!("Button Pressed!");
        button.until_release().await;
        info!("Button Released!");
    }
}
```



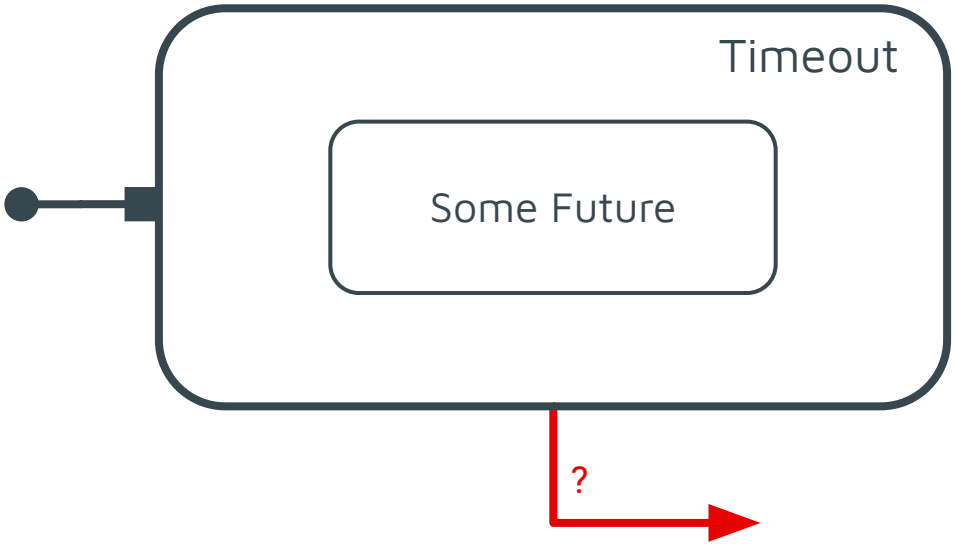
Looping on a Subscription Channel

```
// Open a connection to the remote address.
let client = client::connect("127.0.0.1:6379").await?;

// Subscribe to topic 'peanuts'.
let mut subscriber = client.subscribe("peanuts").await?;

// Await messages on channel `subscriber`.
while let Some(Message { channel, content }) = subscriber.next_message().await {
    println!("got message = {content:?}");
}
```

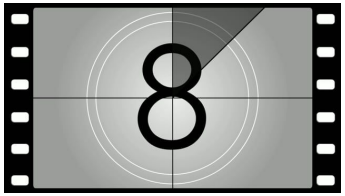


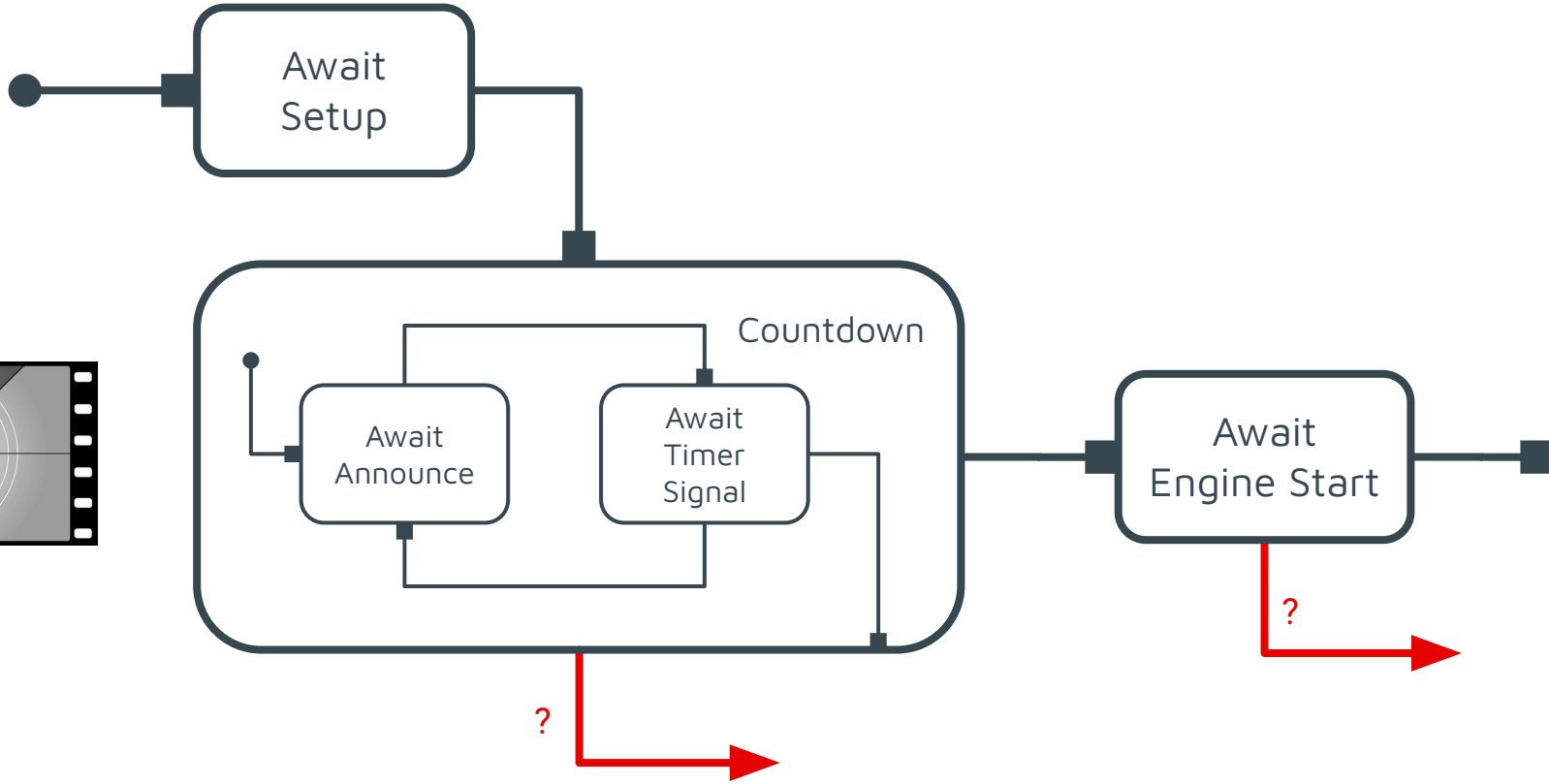
Nested Futures

```
if let Err(_) = timeout(Duration::from_millis(10), fut).await {
    log!("Did not resolve within 10 ms");
}
```

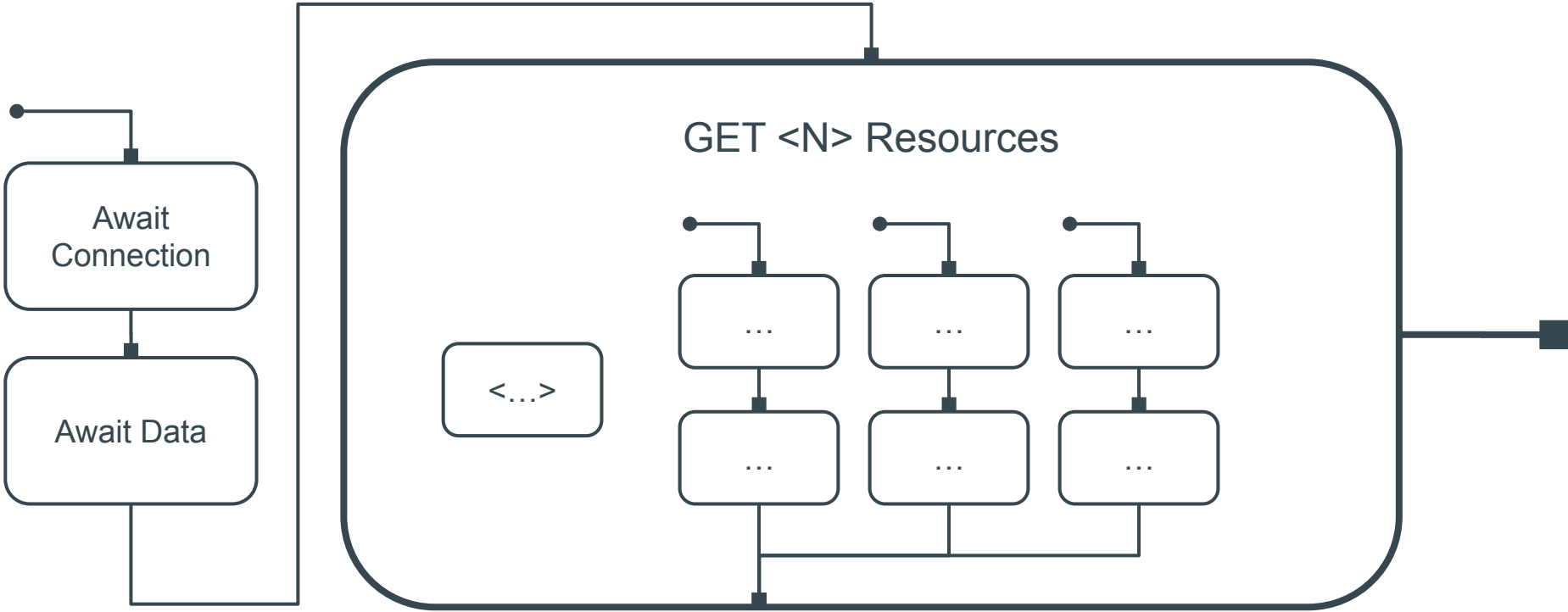


Hierarchical State Machines

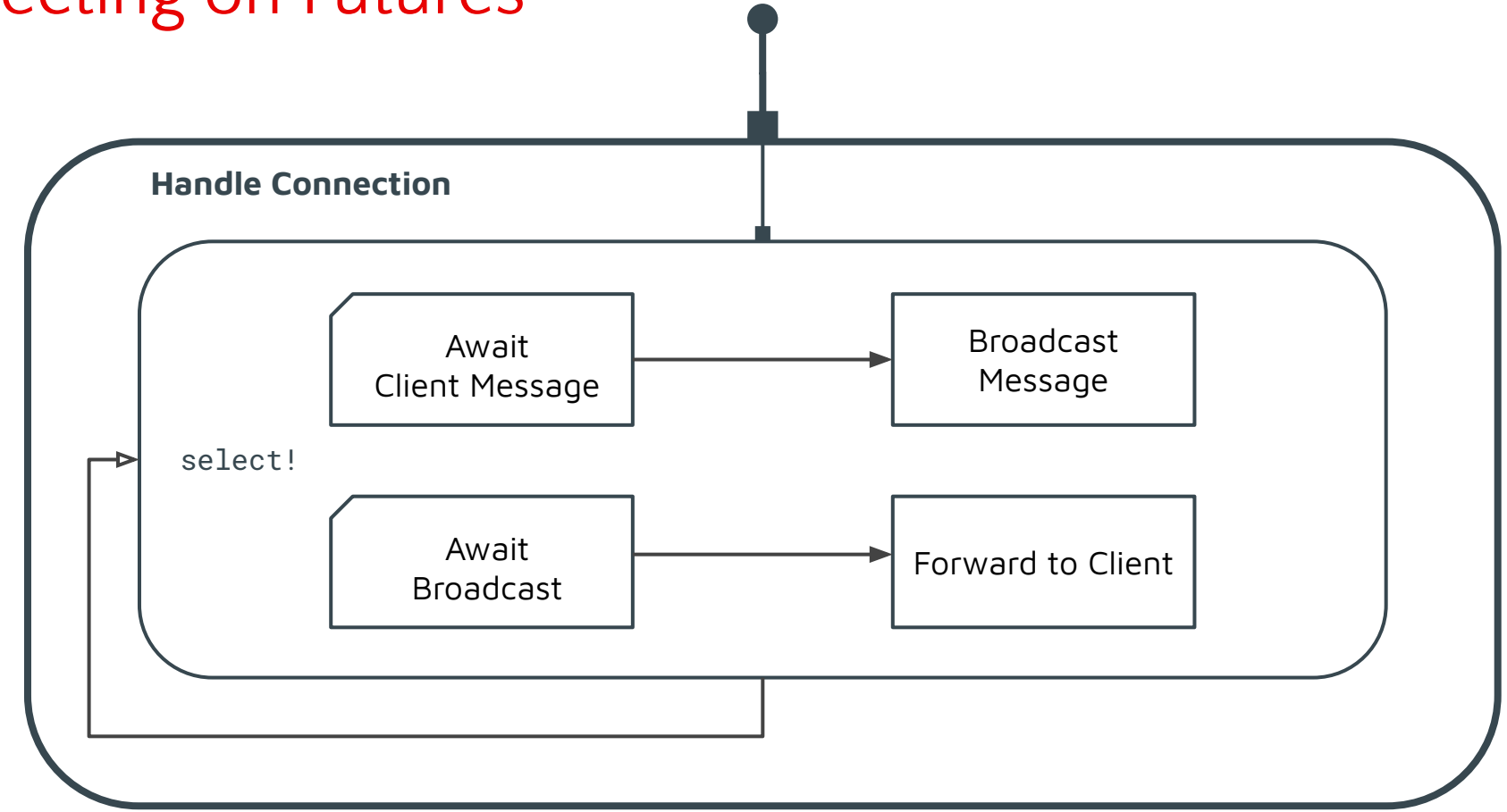




Joining on Futures

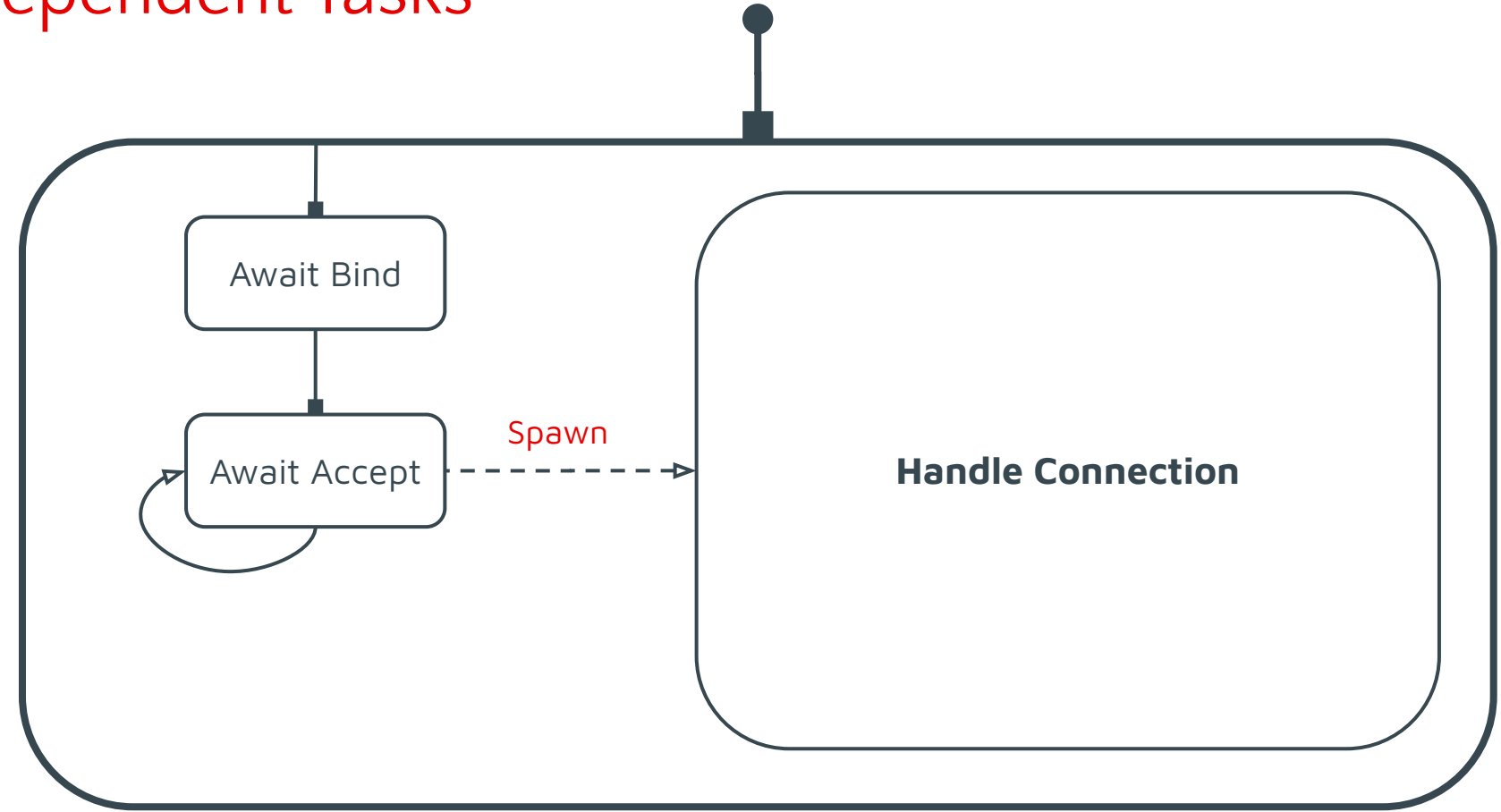


Selecting on Futures



This code on GitHub

Independent Tasks



This code on GitHub

Where's the "Top-Level .await"?

In the runtime environment!

- Interacts with the platforms' event loop
- Wakes Futures when their wake events occur (informed polling)
- APIs to create, compose, and manage futures
- APIs to interact with sockets, files, … (often imitating standard lib)
- Channels, Mutexes, Arcs, etc.

Where's the "Top-Level .await"?

In the runtime environment!

- Distributes Futures among worker threads (optional)
- Allows tracing und profiling (optional)

tokio.rs async-std embassy

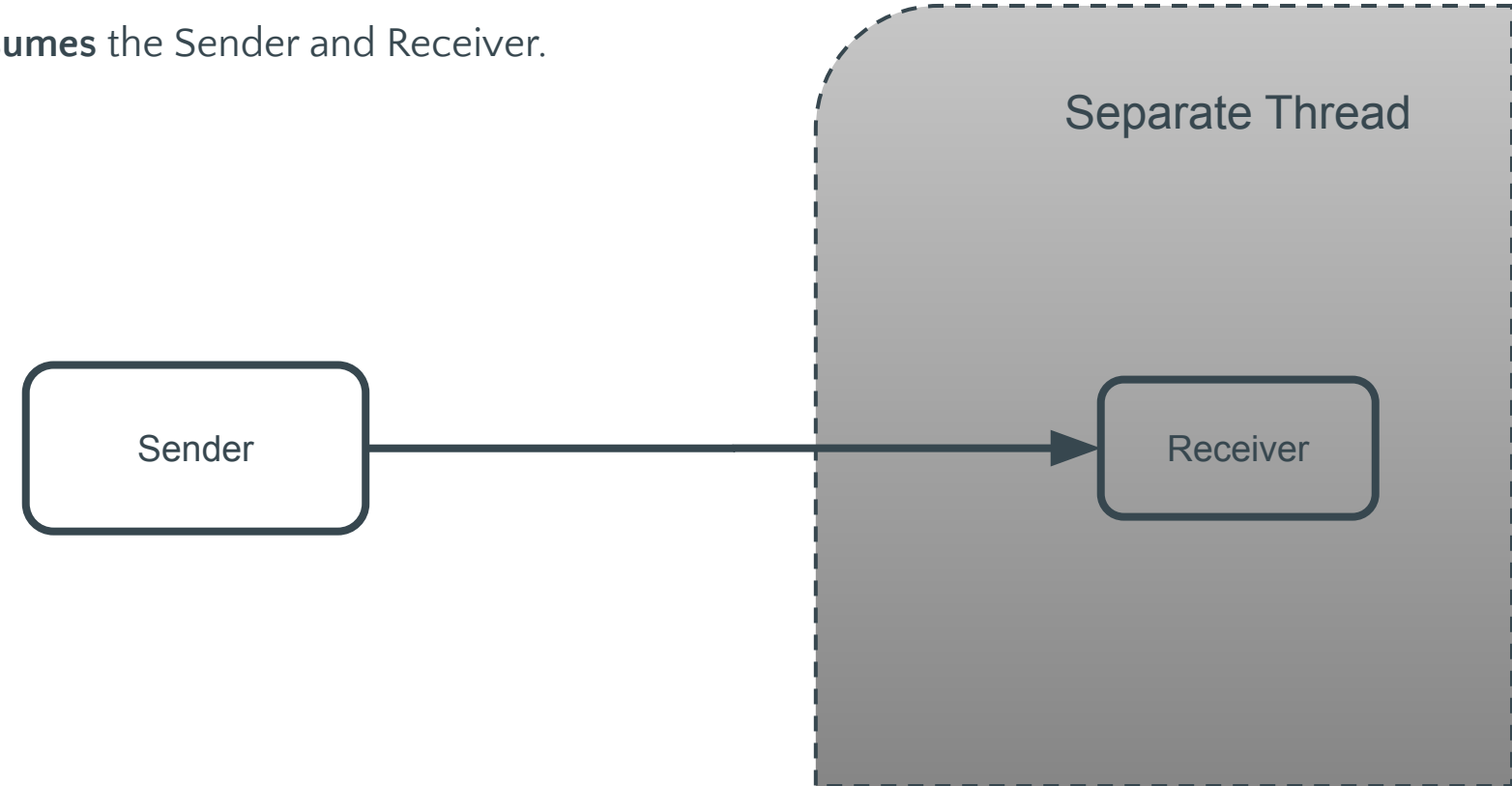
cassette smol-rs/async-executor simple-async-local-executor

Channels

Don't communicate by sharing memory;
share memory by communicating.
– Rob Pike

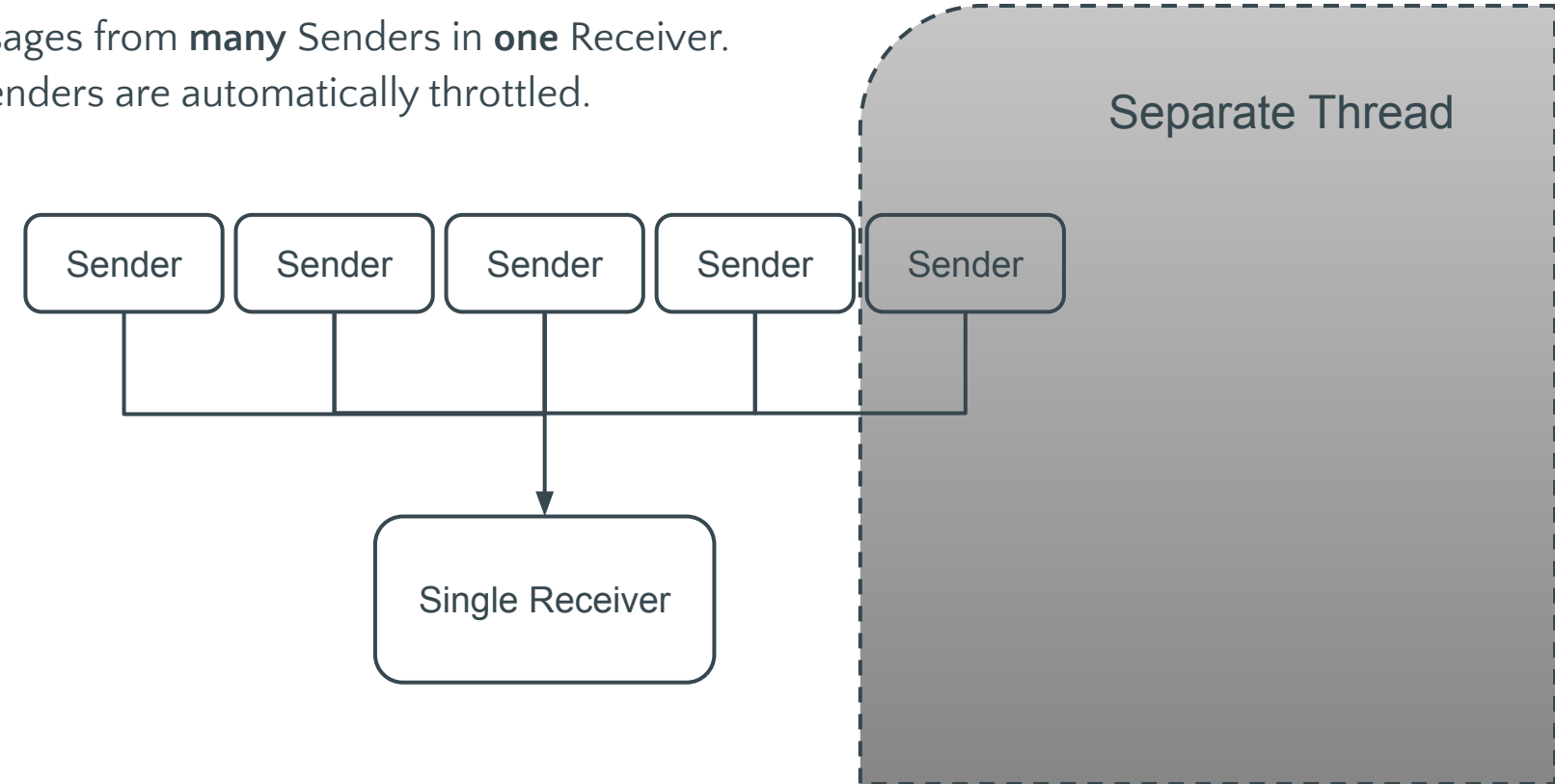
Oneshot Channel

Sends **one** value from **one** Sender to **one** Receiver.
The process **consumes** the Sender and Receiver.

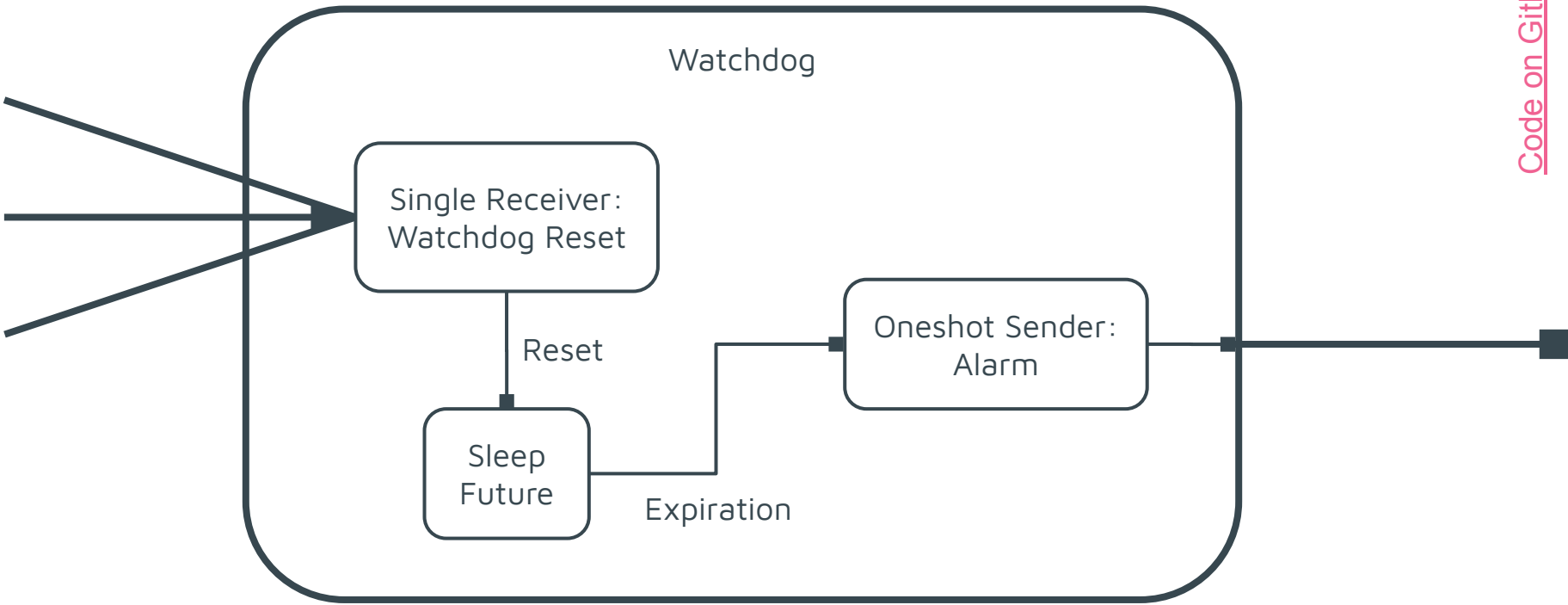


Many Producers, Single Consumer (MPSC Channel)

Collects messages from **many** Senders in **one** Receiver.
Overly fast Senders are automatically throttled.



Watchdog with MPSC + Oneshot



Code on GitHub

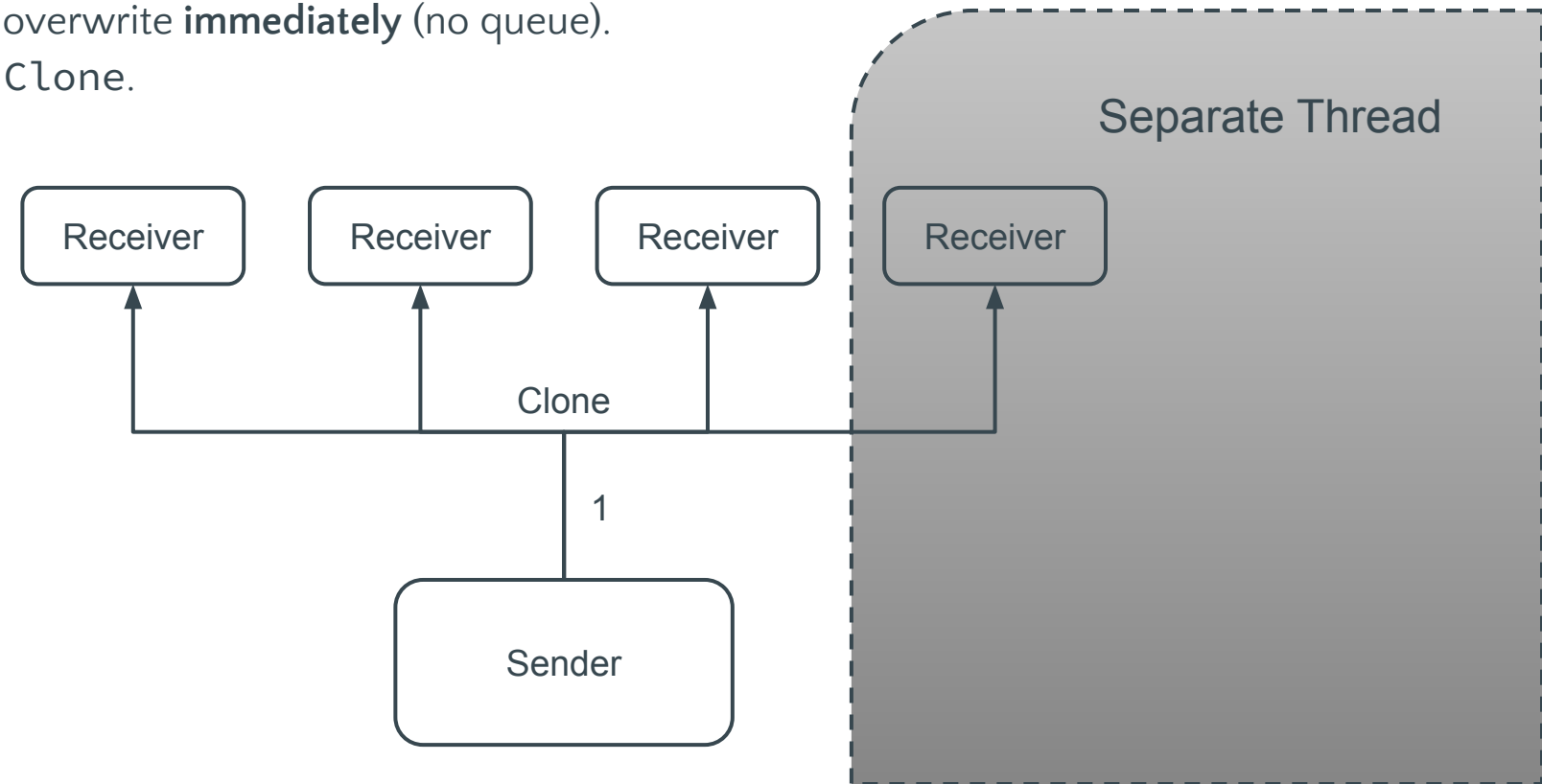
Watchdog with MPSC + Oneshot

```
loop {
    select! {
        msg = reset.recv() => {
            match msg {
                Some(_) => sleep.as_mut().reset(...),
                None => break,
            }
        }
        _ = sleep.as_mut() => {
            let _ = elapsed.send(Elapsed);
            break;
        },
    }
}
```

Code on GitHub

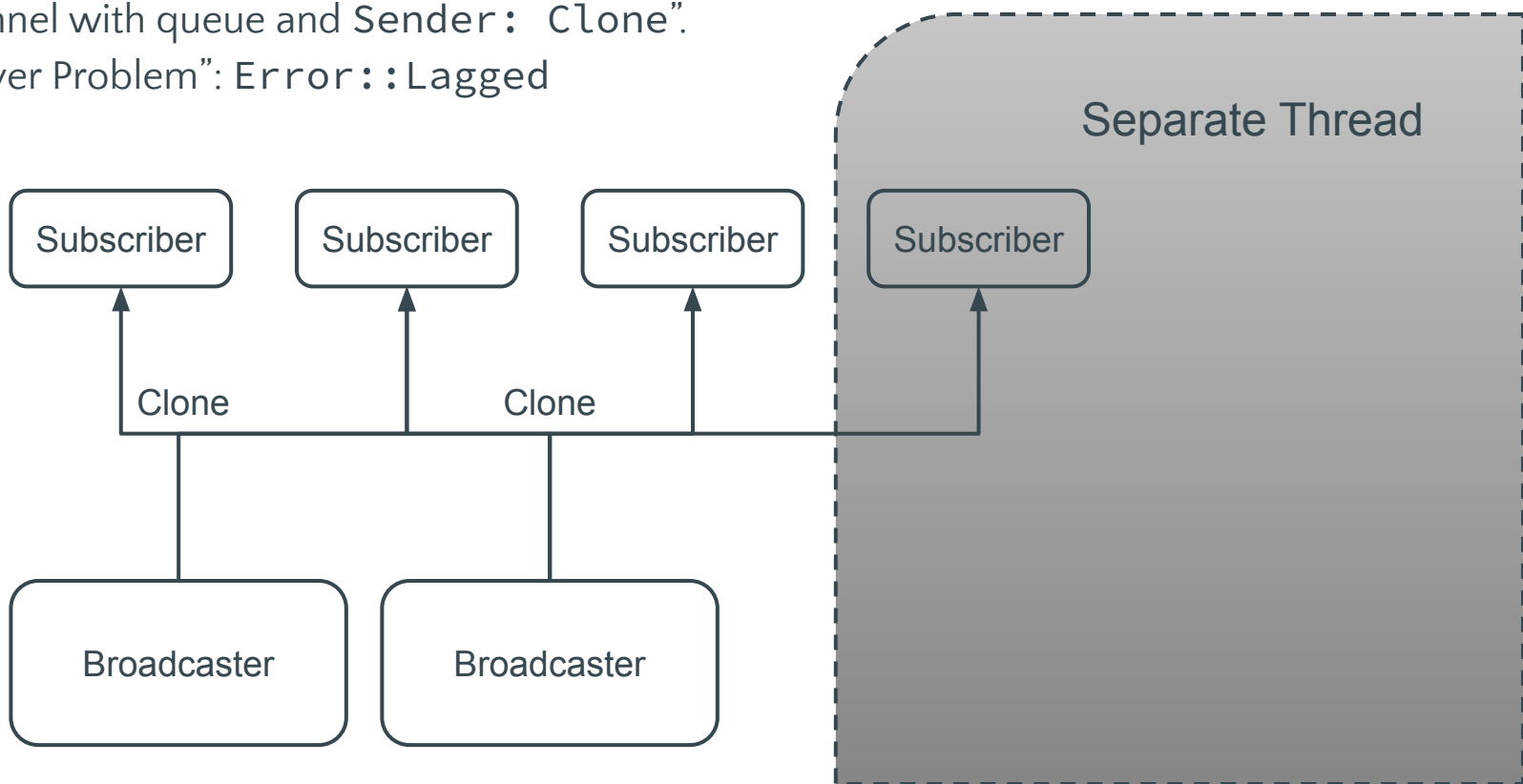
Watch Channel

The **last** value is available for **many** Receivers.
New values overwrite **immediately** (no queue).
Sender isn't `Clone`.



Broadcast Channel

Sends **many** values to **many** Receivers.
“Watch channel with queue and `Sender: Clone`”.
“Slow Receiver Problem”: `Error::Lagged`



Demo: AChat

„I hear and I forget. I see and I remember. I do and I understand.“ – Confucius

AChat: Async IO Example Programs

- Simple TCP Server application: Futures, `async/.await`, Structured Concurrency, Channels, ...
- `tokio` runtime (+ some other crates)

github.com/barafael/achat

Documentation

AChat: Async IO Example Programs

Example binaries, for example:

- Simple Chat (broadcast)
- Chat with announce (broadcast, watch)
- Collector (broadcast, mpsc, oneshot)
- Echo (tokio::io::copy)

Unit testing with Mocks

Tracing with `tokio-console`

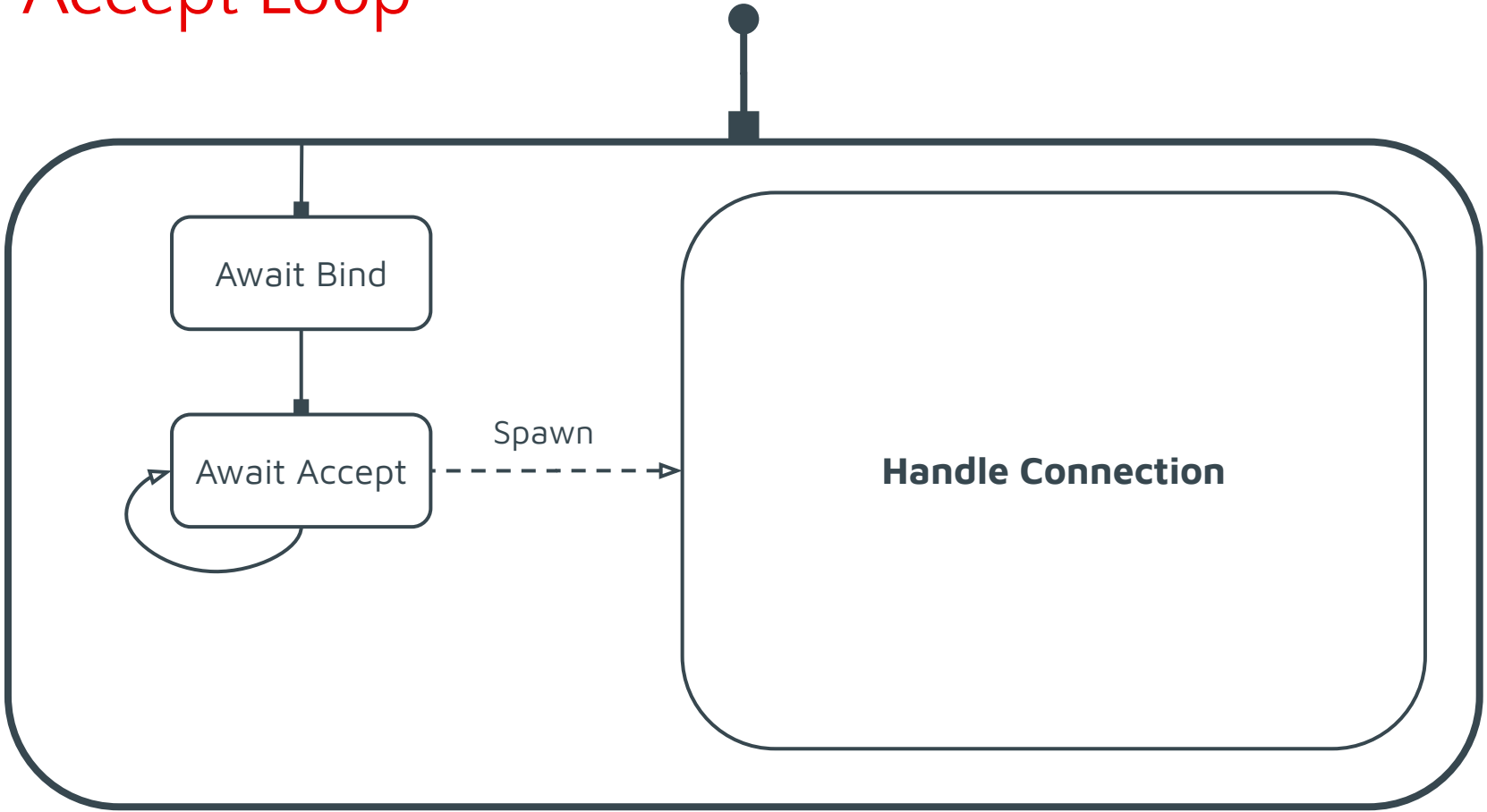
Actor Design Pattern

Actor Design Pattern

- Free-standing asynchronous task
- Local mutable state
- Structured Concurrency (often `select!`) within the actor
- Channels for communication with other actors

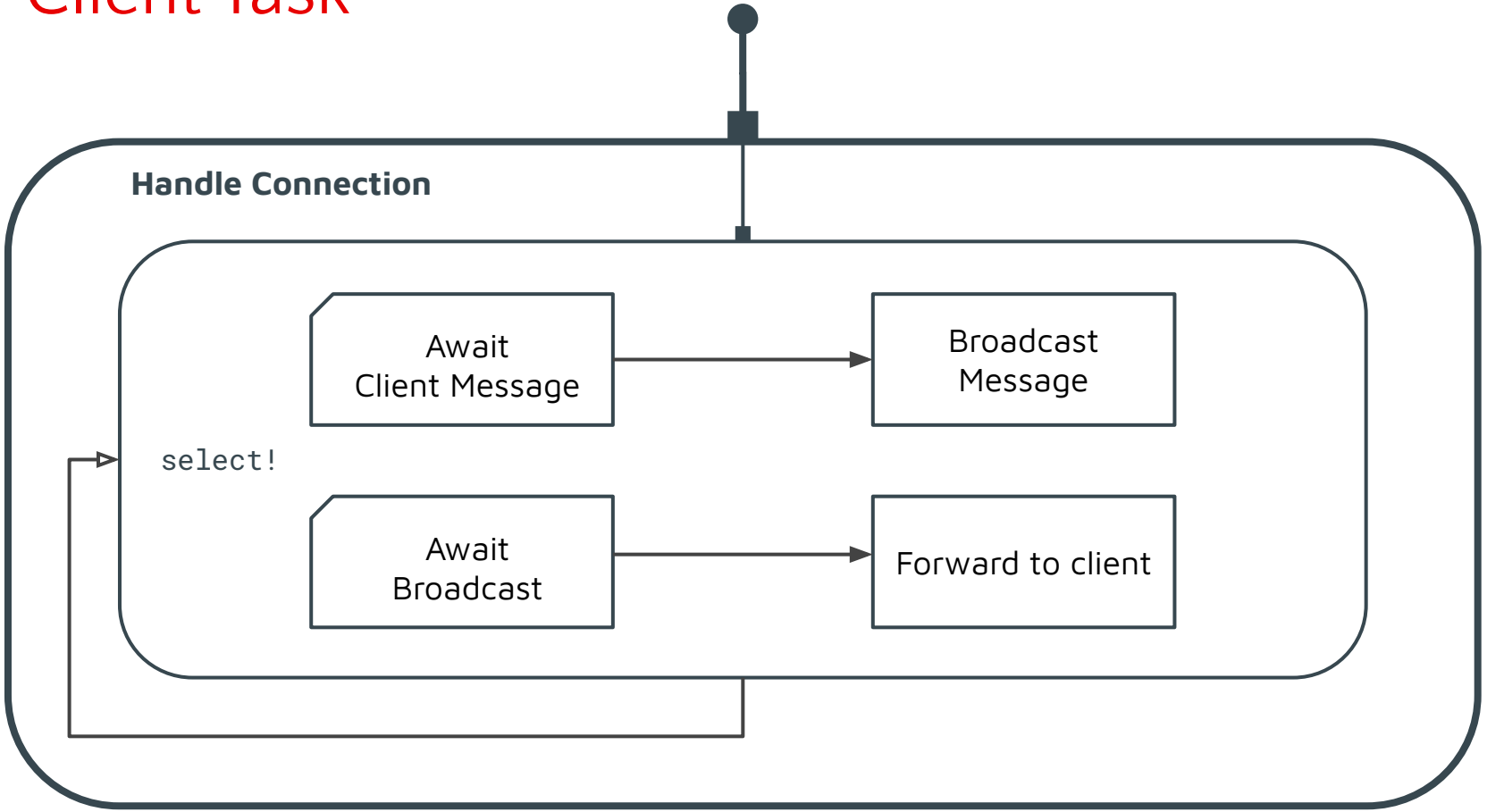
ryhl.io/blog/actors-with-tokio/

TCP Accept Loop



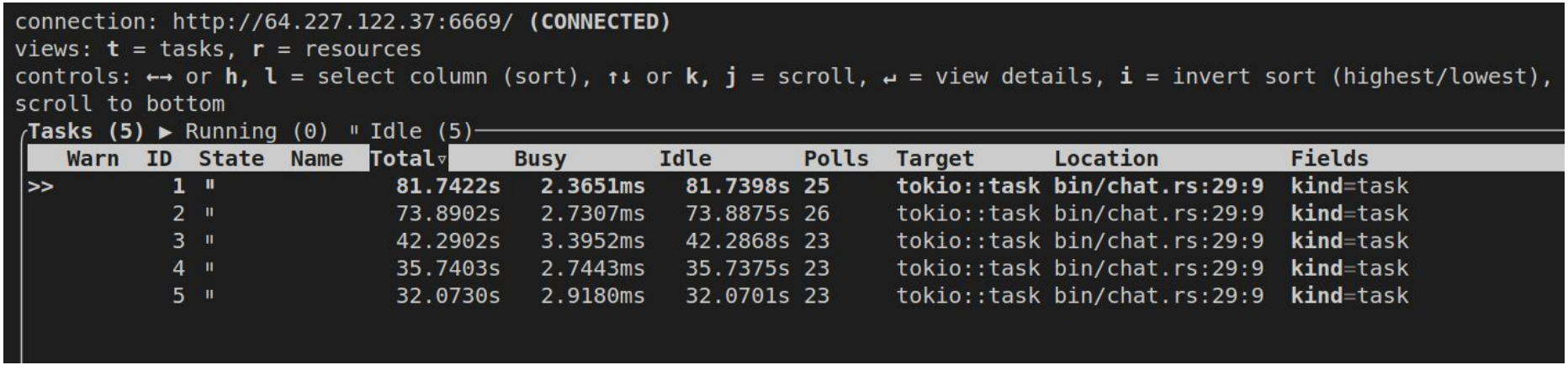
This code on GitHub

TCP Client Task

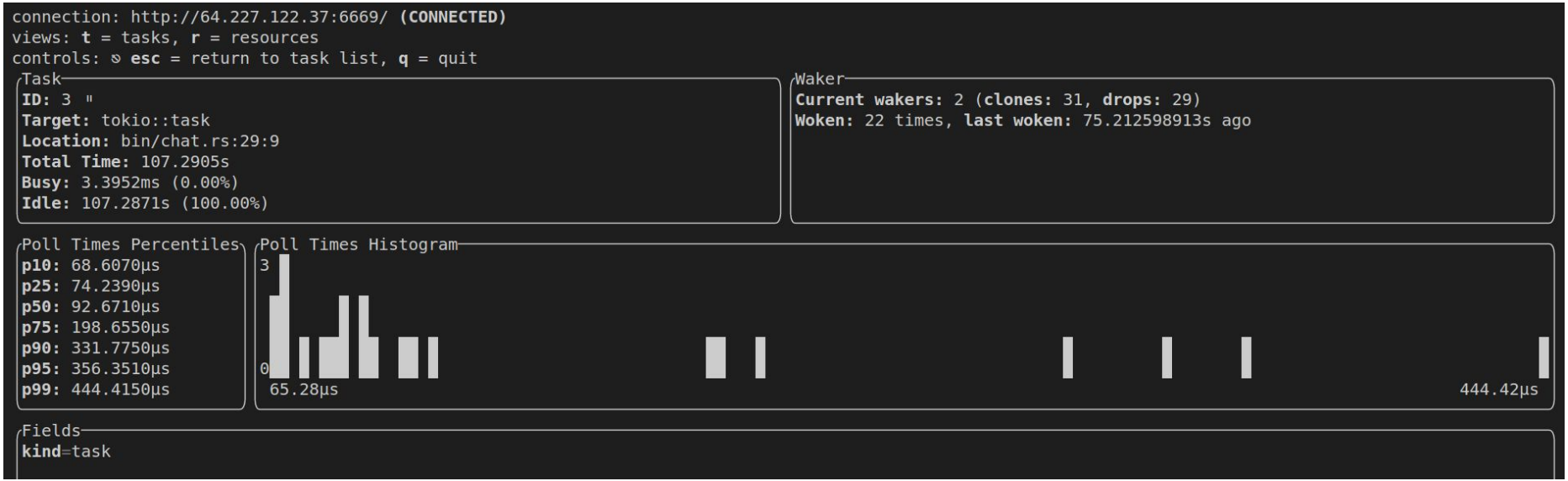


This code on GitHub

Tracing with Tokio-Console



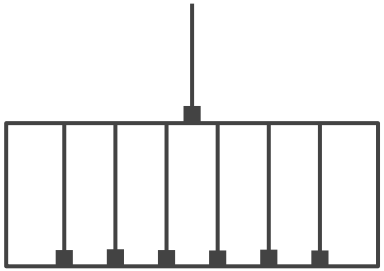
Tracing with Tokio-Console



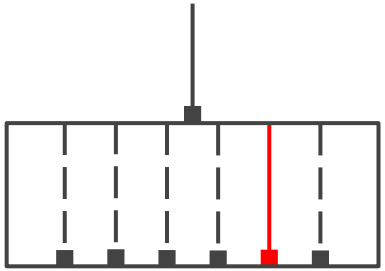
Structured Concurrency

Logical Operators

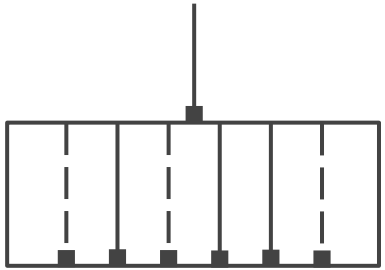
AND



XOR

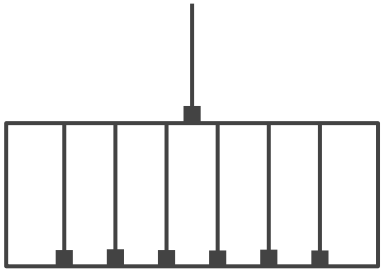


IOR

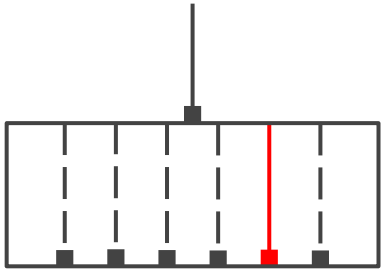


Data Structure Primitives

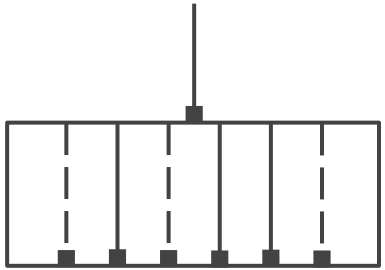
STRUCT



ENUM

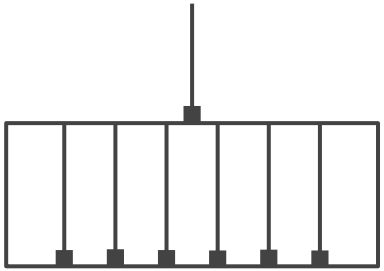


IOR/UNION

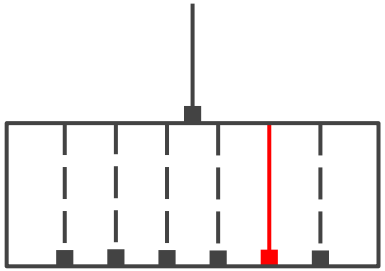


Control Flow Primitives

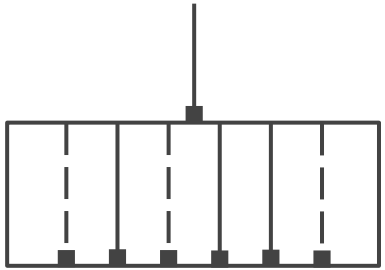
LOOP



IF

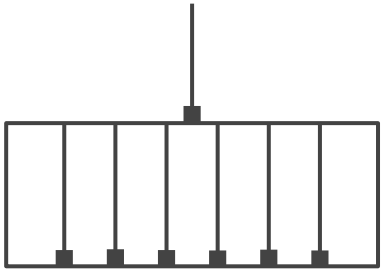


GOTO

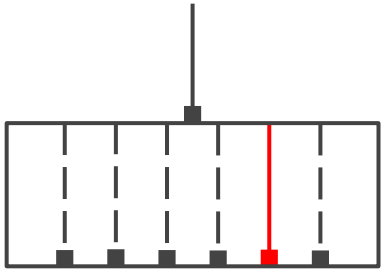


Future Combinators

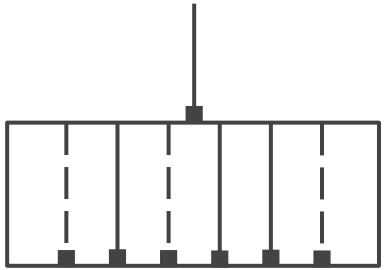
JOIN



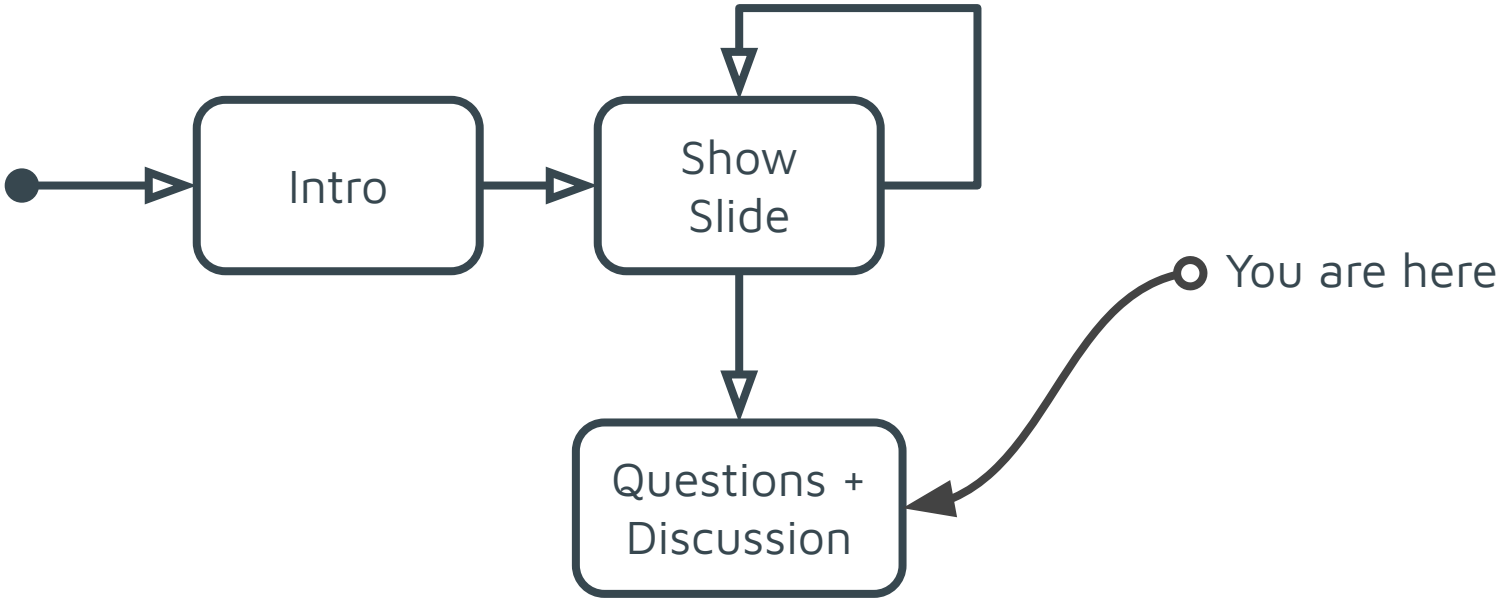
SELECT



GO/DETACH/SPAWN



go statement considered harmful?


Intro
Show
Slide
Questions +
Discussion
You are here