Eine Einführung in Asynchrone Programmierung mit async/.await in Rust

Rafael Bachmann
Software Developer

ESRLabs

Embedded Automotive (embedded Linux)

Netzwerkapplikationen (u.a. mit Rust)

We are hiring!

esr_

Part of **Accenture**

Rust in 3 Minuten

- Moderne Systemprogrammiersprache
- Typsystem erzwingt korrekten Umgang mit gemeinsamen Ressourcen
- Compiler + Tooling: freundlich, ergonomisch, einfach erweiterbar
- Vielfältige Bibliotheken durch

  Abhängigkeitsmanagement und Modulsystem

Rust in 3 Minuten – Ziele und Tradeoffs

- Binaries, Performance wie C
- Typsystem angelehnt an Haskell
- Ergonomie wie Python
- Tooling wie… Javascript/Node.js
- “Einzigartige” Lernkurve
- Compile-times wie C++ (oder schlimmer)

www.rust-lang.org

Rust in 3 Minuten – Garantien

- Keine SEGFAULTS
  - Panic: Strukturierte Dekonstruktion
- Kein Undefiniertes Verhalten
- Keine Data Races
- Zero-Cost Abstractions

www.rust-lang.org

Asynchrone Programmierung in 3 Minuten

I/O kann beliebig lange dauern:

- Download
- Mausklick auf Button
- Bytes über Sockets
- Signale, Events
- TCP Server wartet auf Clients

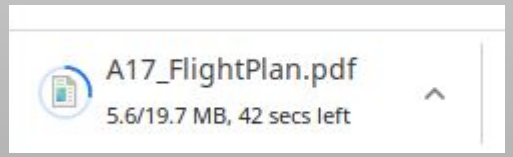
I/O bound: Aktionen innerhalb eines Programmes können beliebig lange dauern, und das Programm kann in der Zeit nichts sinnvolles tun.

Asynchrone Programmierung in 3 Minuten

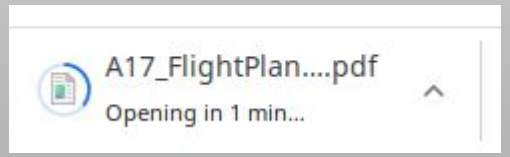
Oft: I/O bound Prozess beschreibt Sequenzen von Schritten,

zwischen denen Zeit vergehen kann.

Einfache Warten-Operation



Sequenz: Warten + Aktion



Beispiel im Rust Playground

“Kontrollfluss” ist auch eine Ressource

- Eine blockende, wartende Funktion verschwendet Kontrollfluss (busy waiting)

Asynchrone Programmierung nutzt Kontrollfluss effizient

- Multithreading/Multiprocessing
- Kooperatives Multitasking: Tasks geben Kontrolle freiwillig ab
- Blockende Operationen sind gute solche Yield-Points

Grundlagen:
Futures,
Zustandsautomaten,
Laufzeitumgebungen

Was ist eine “Future”?

Eine Future ist einfach ein Wert, den es erst in der Zukunft geben wird.

- Beschreibt eine Aktion, arbeitet aber nicht (**lazy**)
- Um den Wert zu erhalten, kann man **pollen**
- Die Future ist dann entweder **Ready** oder noch **Pending**
- **Kein Busy-Polling** notwendig:

  Future gibt Ereignis an, bei welchem sie gepollt werden will

Was ist eine “Future”?

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

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

Futures als Zustandsautomaten

Zustand: Warten auf Ereignis

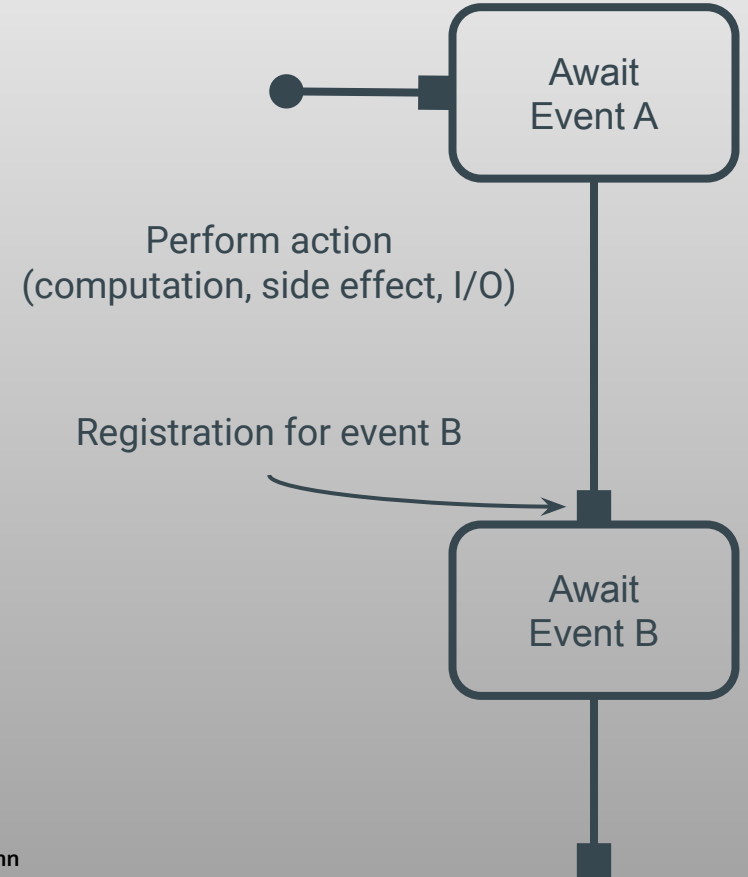
Transition:

Seiteneffekt, Aktion, Berechnung

Eintritt in Zustand:

Registrierung für Wake-Event

(im Event-Loop einer Laufzeitumgebung)

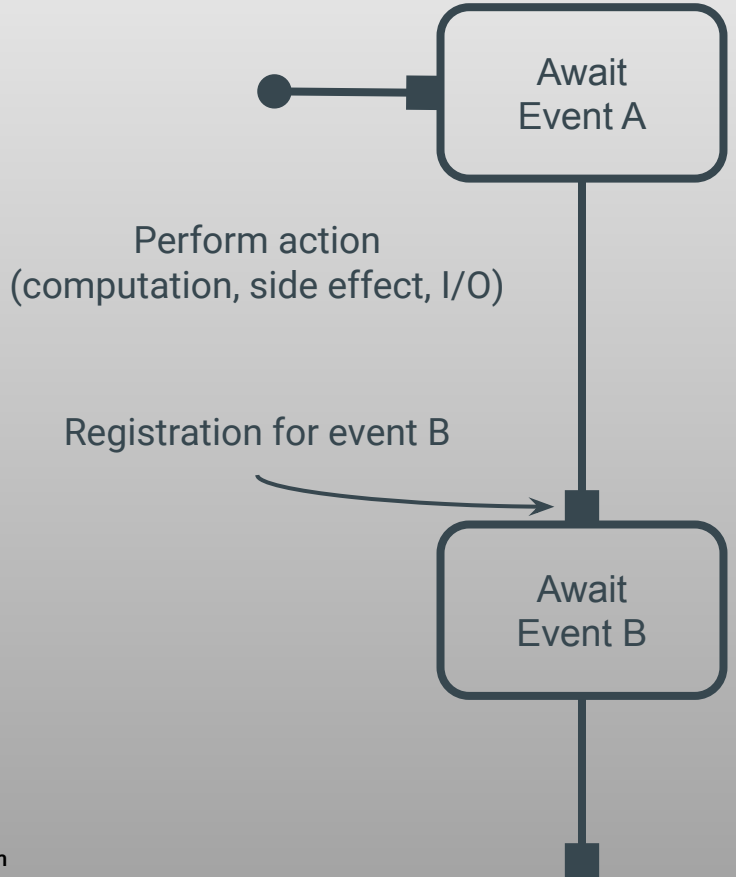


Futures als Zustandsautomaten

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

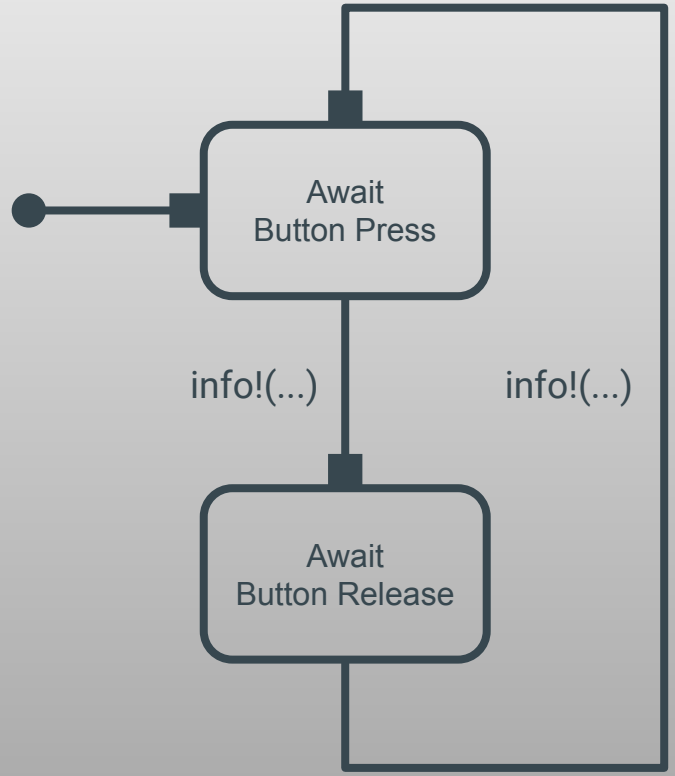
async erstellt eine Future

.await konsumiert eine Future



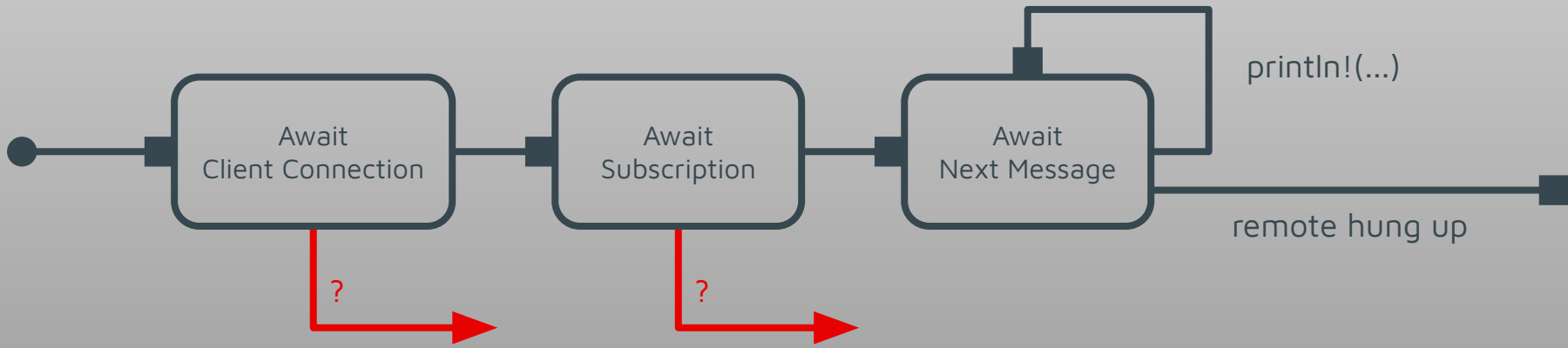
Praktischeres Beispiel 1

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



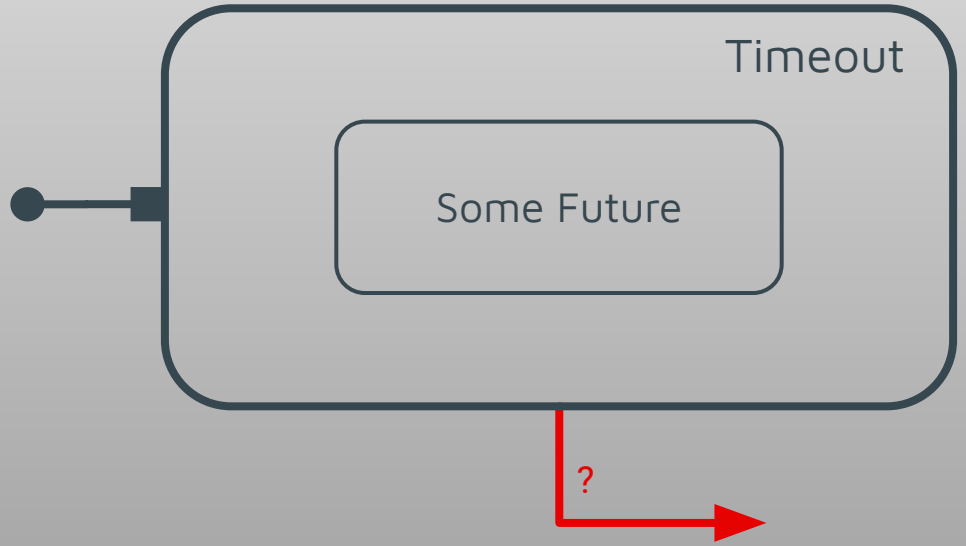
Praktischeres Beispiel 2

```rust
// 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:?}");
}
```

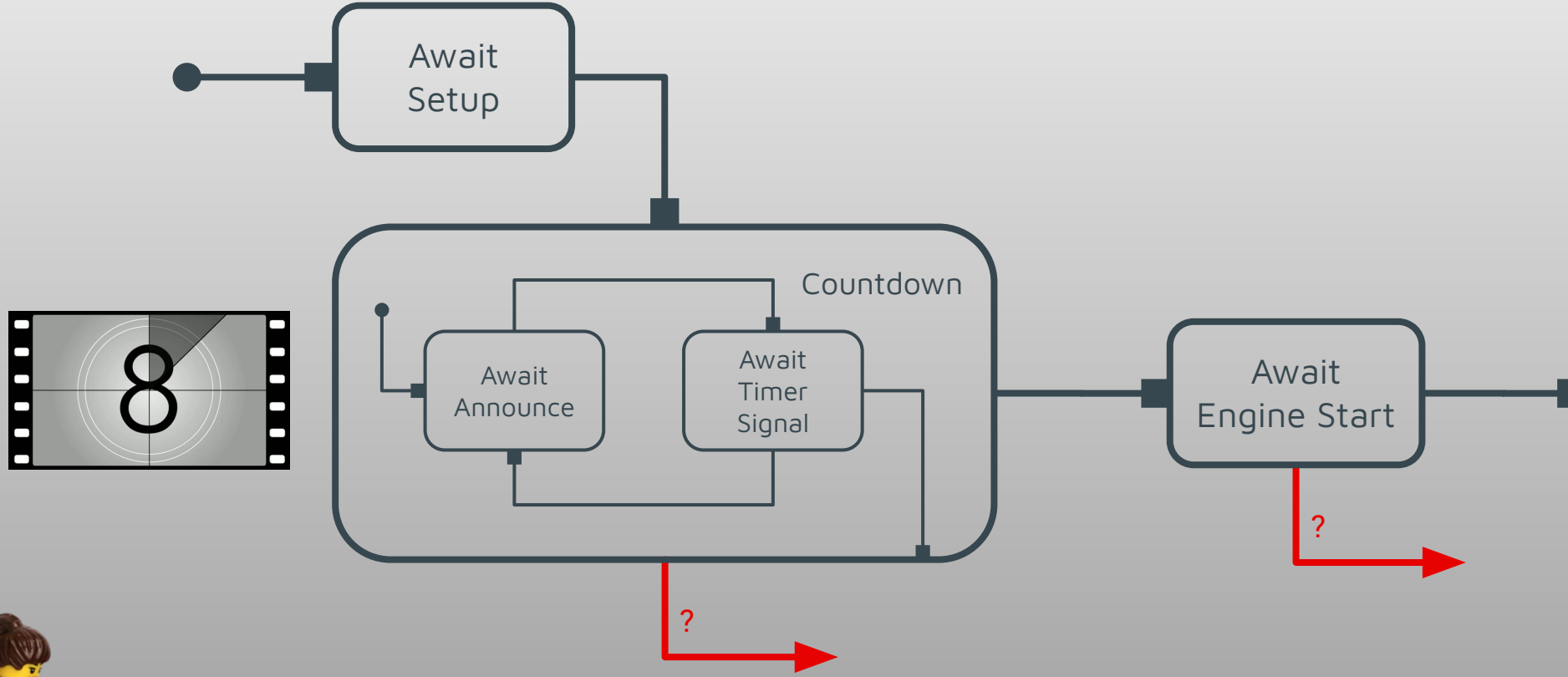


Geschachtelte Futures

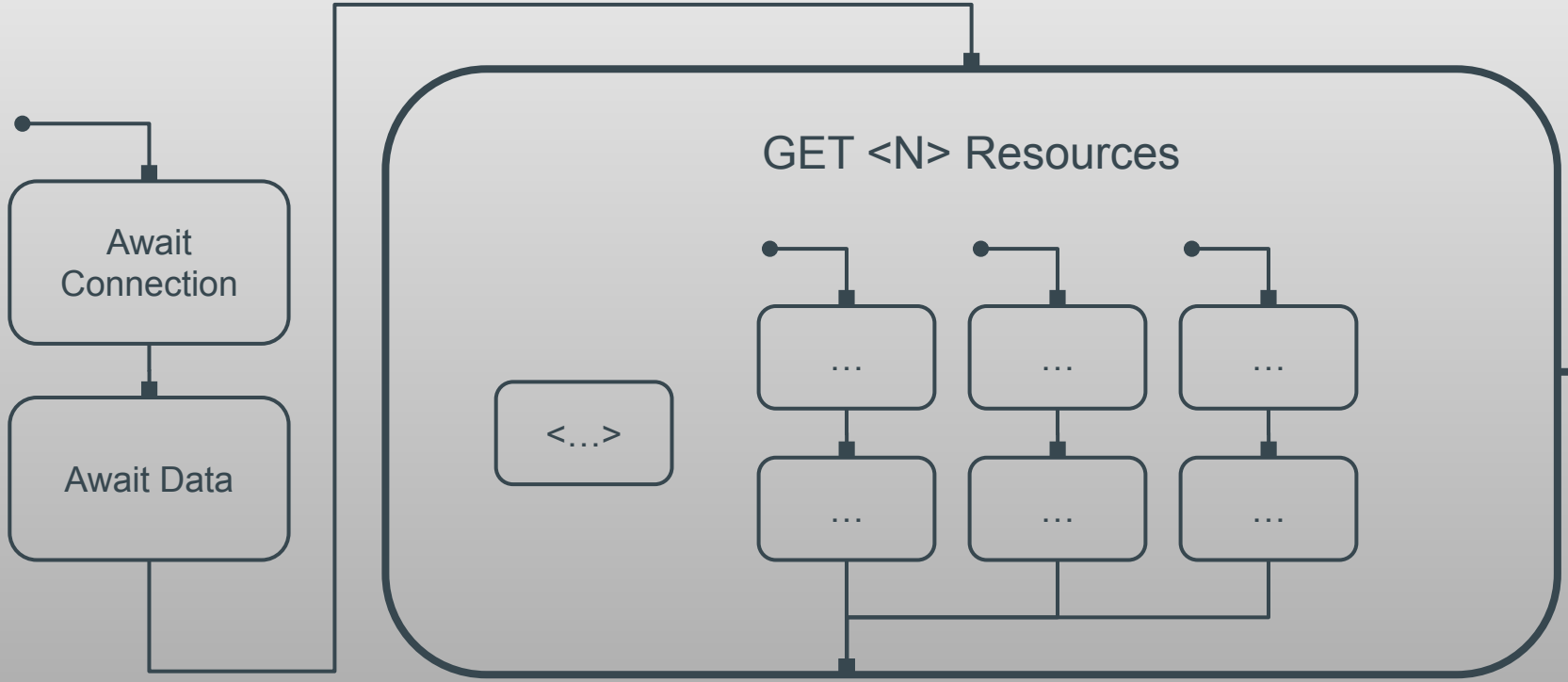
```rust
if let Err(_) = timeout(Duration::from_millis(10), fut).await {
    log!("Did not resolve within 10 ms");
}
```



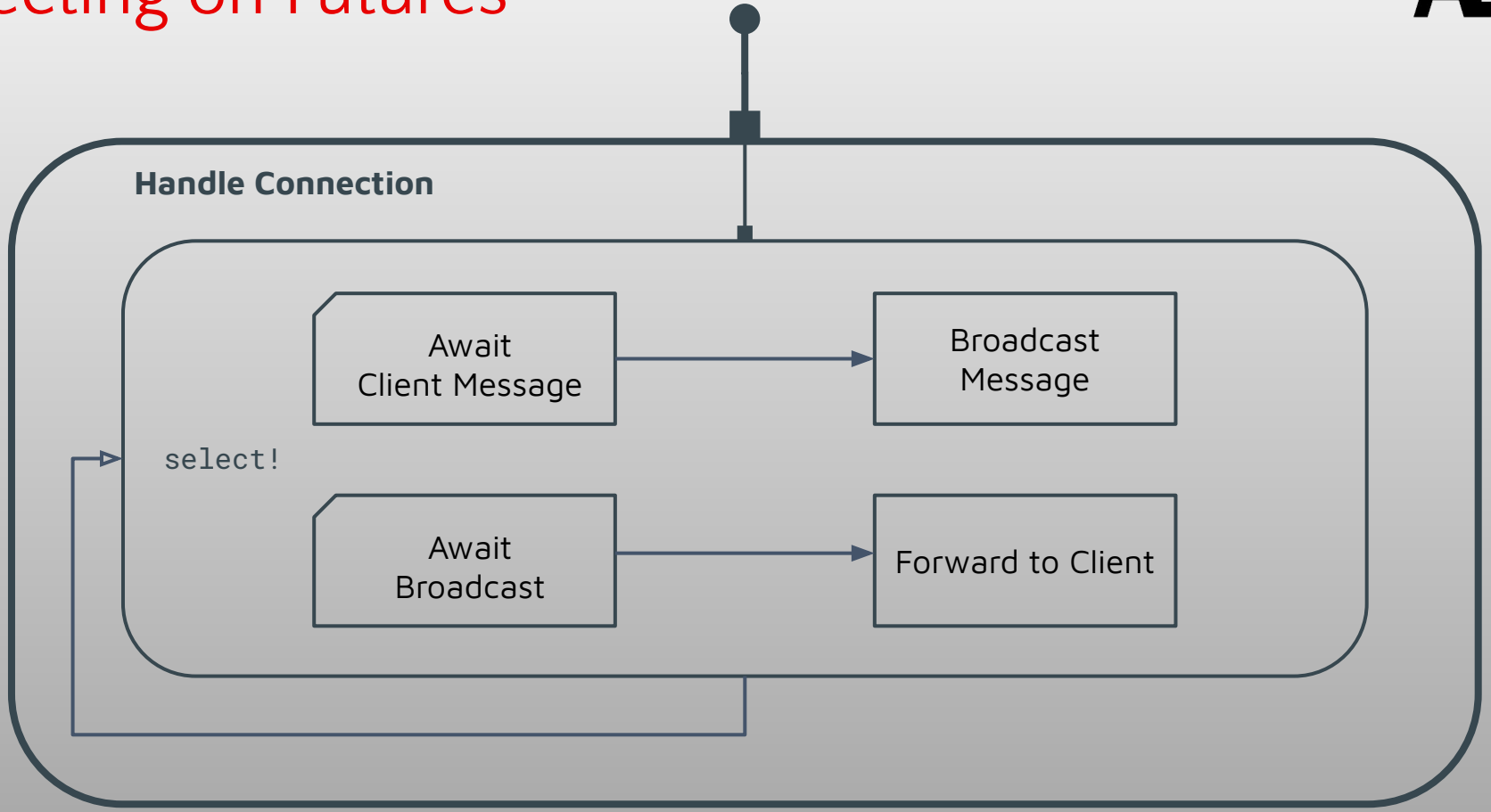
Hierarchische Automaten



Joining on Futures

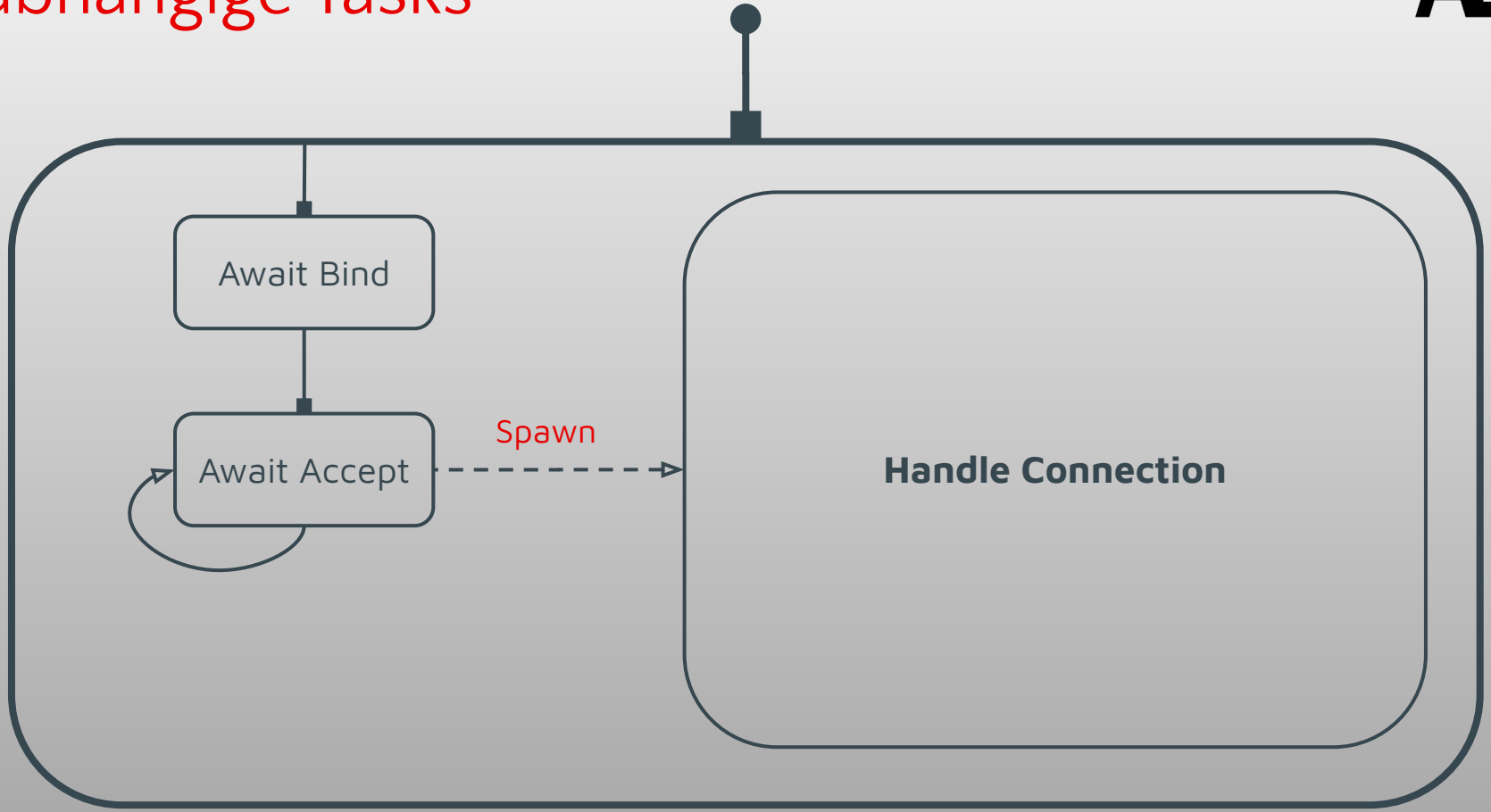


Selecting on Futures



This code on GitHub

Unabhängige Tasks



This code on GitHub

Wo ist das “Top-Level .await”?

In der Laufzeitumgebung!

- Interagiert mit Event-Loop der Plattform
- Weckt Futures wenn ihre Wake-Ereignisse eintreten (informed polling)
- Verteilt Futures auf Worker Threads (optional)
- Erlaubt Tracing und Profiling (optional)

Wo ist das “Top-Level .await”?

In der Laufzeitumgebung!

- APIs zum Erstellen, Schachteln, und Verwalten von Futures
- APIs zur Interaktion mit Files, Sockets, Zeit, etc.

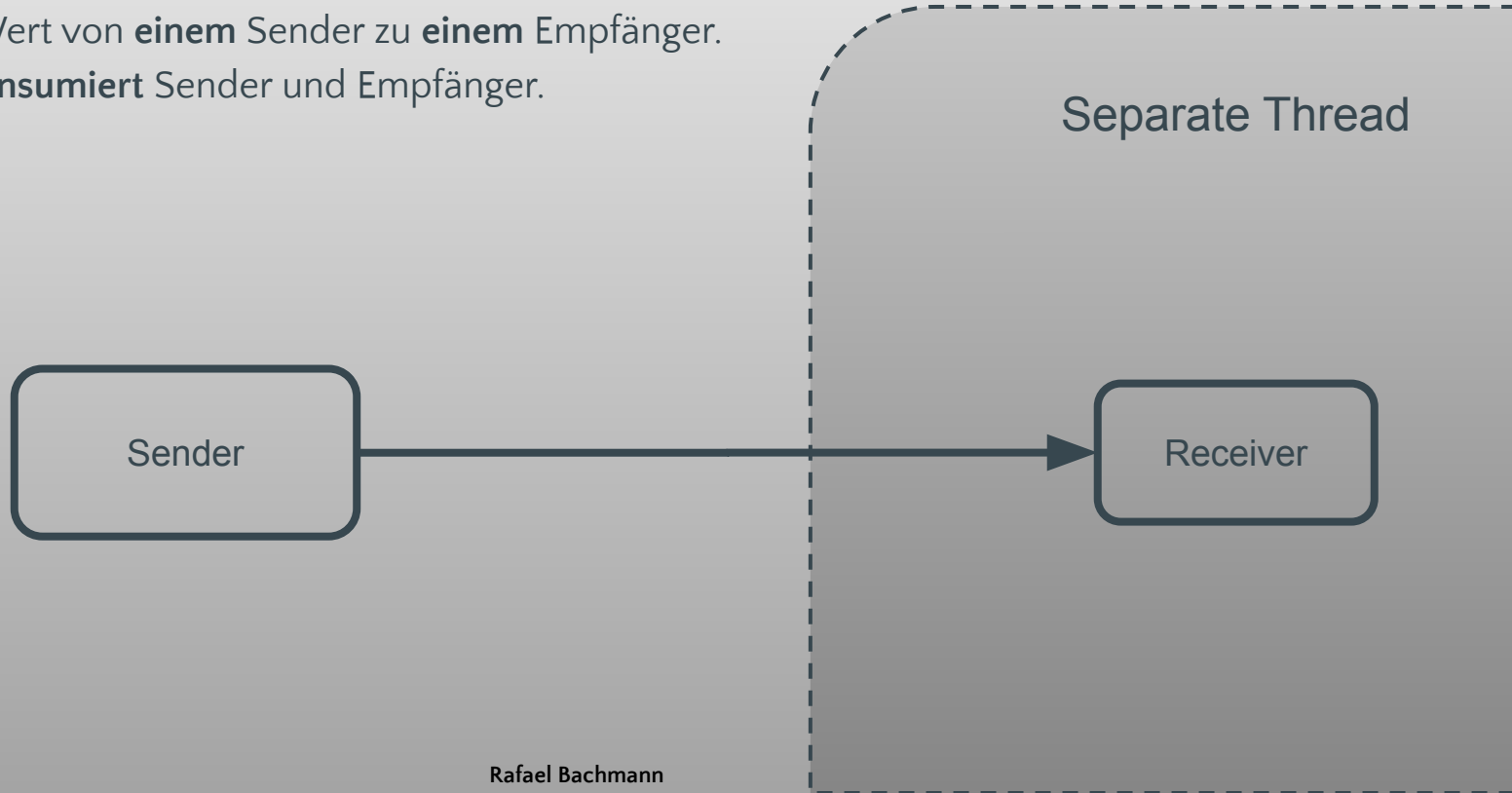
  (Oft asynchrone Varianten der Funktionen im Standard Library)
- Channels, Mutexes, Arcs, etc.

Channels

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

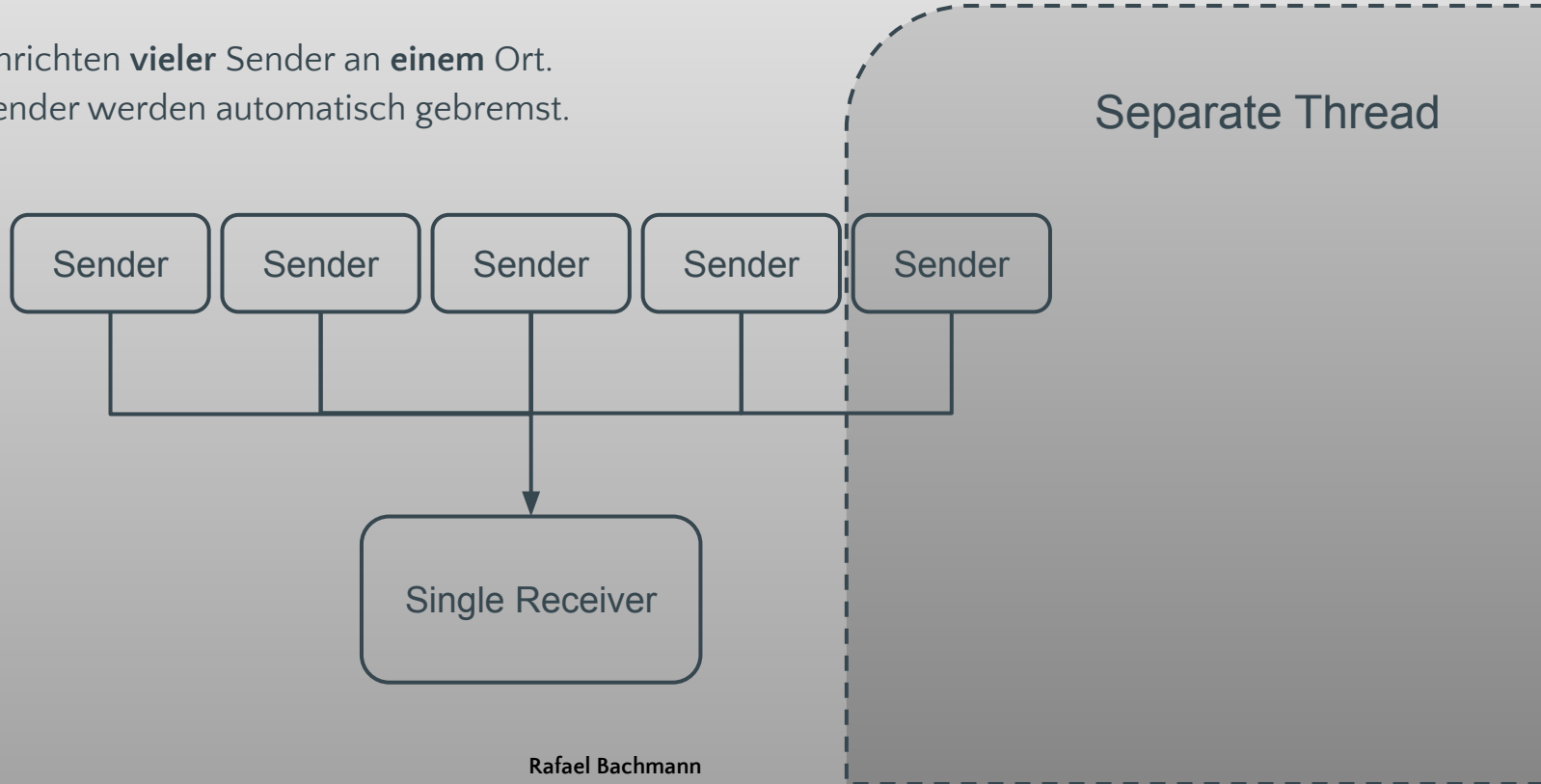
Oneshot Channel

Sendet **einen** Wert von **einem** Sender zu **einem** Empfänger.
Der Vorgang **konsumiert** Sender und Empfänger.



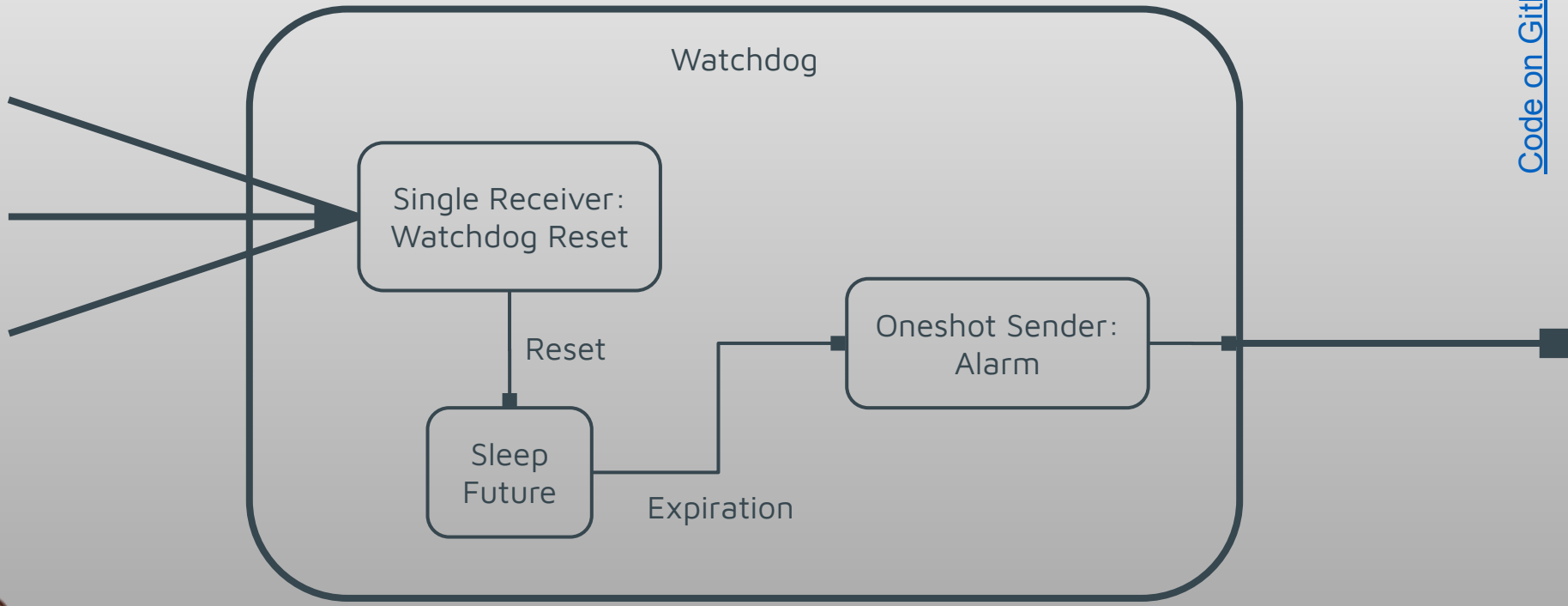
Many Producers, Single Consumer (MPSC Channel)

Sammelt Nachrichten **vieler** Sender an **einem** Ort.
Zu schnelle Sender werden automatisch gebremst.



Watchdog mit MPSC + Oneshot

Code on GitHub



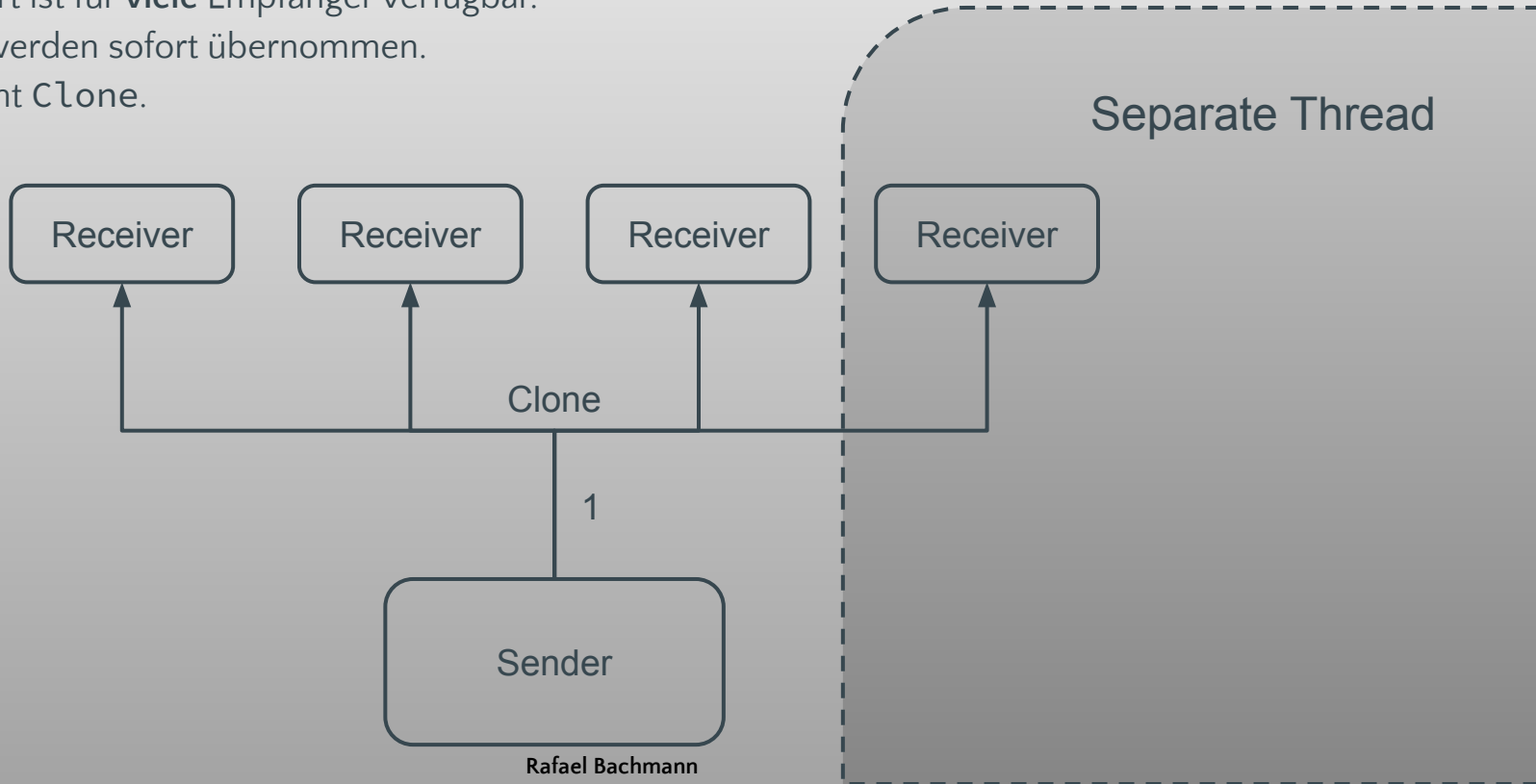
Watchdog mit MPSC + Oneshot

```
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

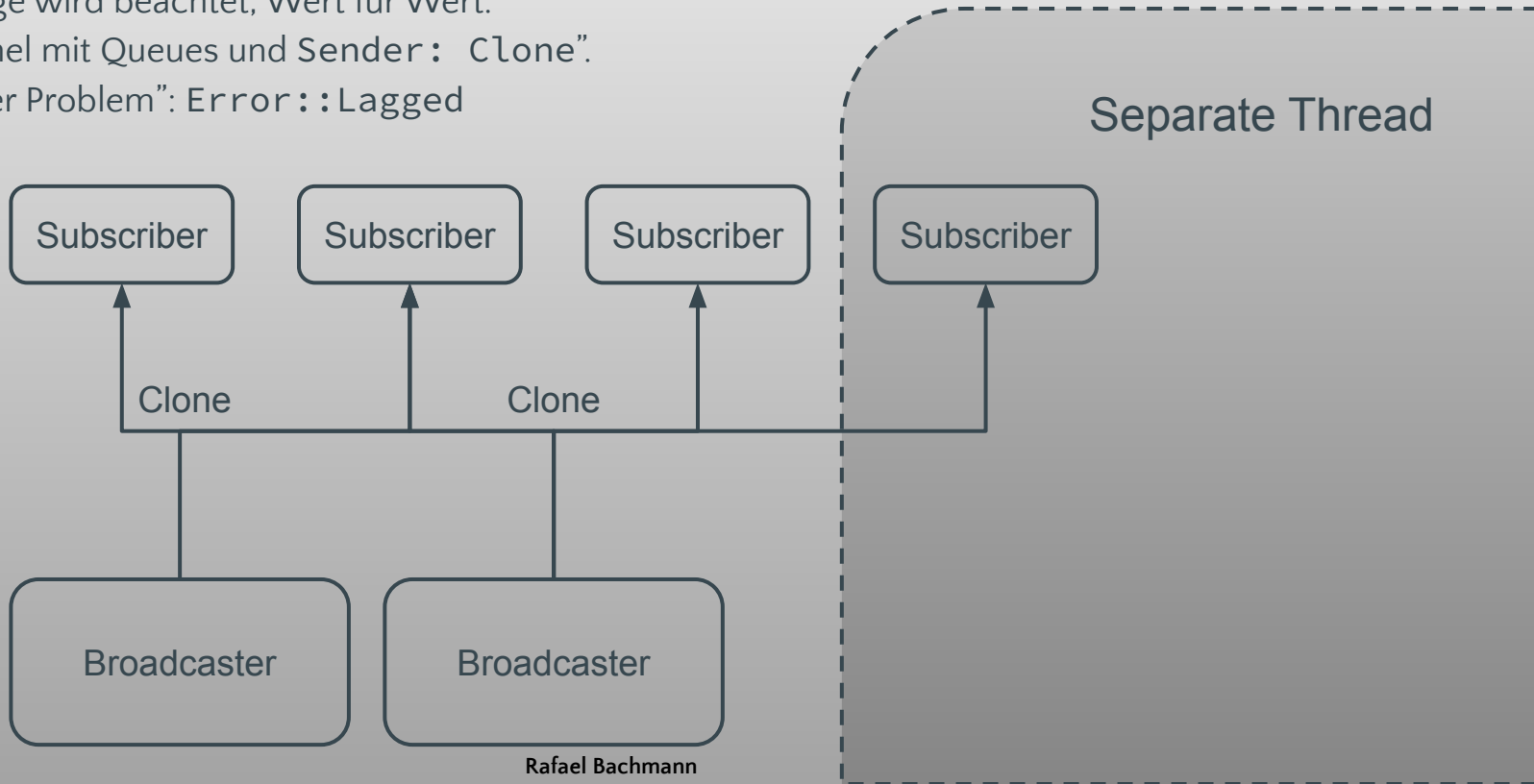
Watch Channel

Der **letzte** Wert ist für **viele** Empfänger verfügbar.
Neue Werte werden sofort übernommen.
Sender ist nicht `Clone`.



Broadcast Channel

Sendet **viele** `Clone`-able Werte an **viele** Empfänger.
Die Reihenfolge wird beachtet, Wert für Wert.
“Watch channel mit Queues und `Sender: Clone`”.
“Slow Receiver Problem”: `Error::Lagged`



Demo: AChat

„Ich höre und ich vergesse. Ich sehe und ich erinnere mich. Ich tue und ich verstehe.“ – Konfuzius

AChat: Async IO Beispielprogramme

- Einfache TCP Serveranwendung für Futures, `async`/`.await`, Strukturierte Nebenläufigkeit, Channels
- `tokio` als Laufzeitumgebung (+ ein paar andere Crates)

github.com/barafael/achat

Documentation

Beispielprogramme

Beispielprogramme, z.b.:

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

Unit testing mit Mocks

Tracing mit `tokio-console`

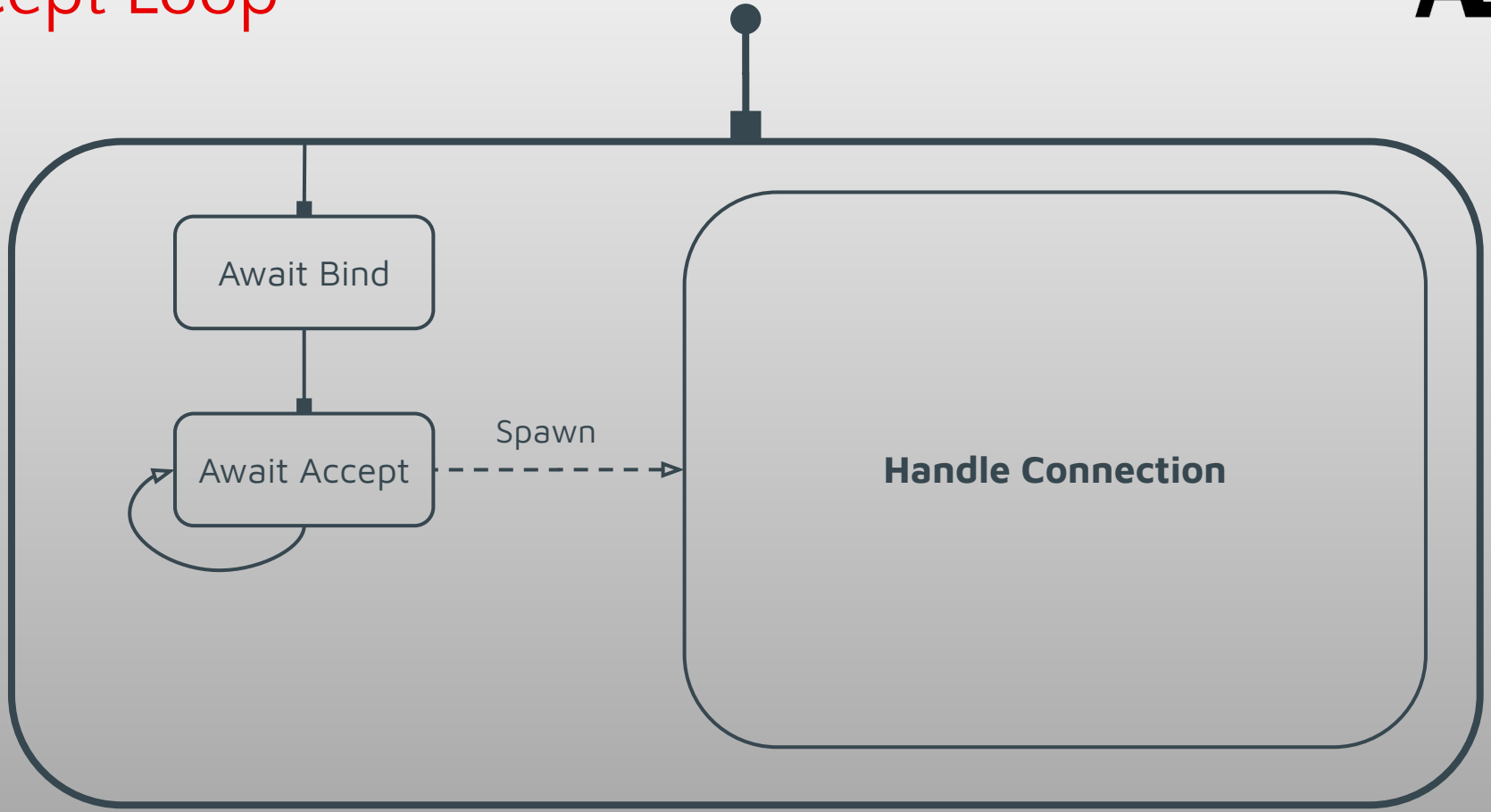
Actor Design Pattern

Actor Design Pattern

- Freistehender asynchroner Task
- Lokaler mutierbarer Zustand
- Strukturierte Nebenläufigkeit (oft select!) innerhalb eines Aktoren
- Channels zur Kommunikation mit anderen Aktoren

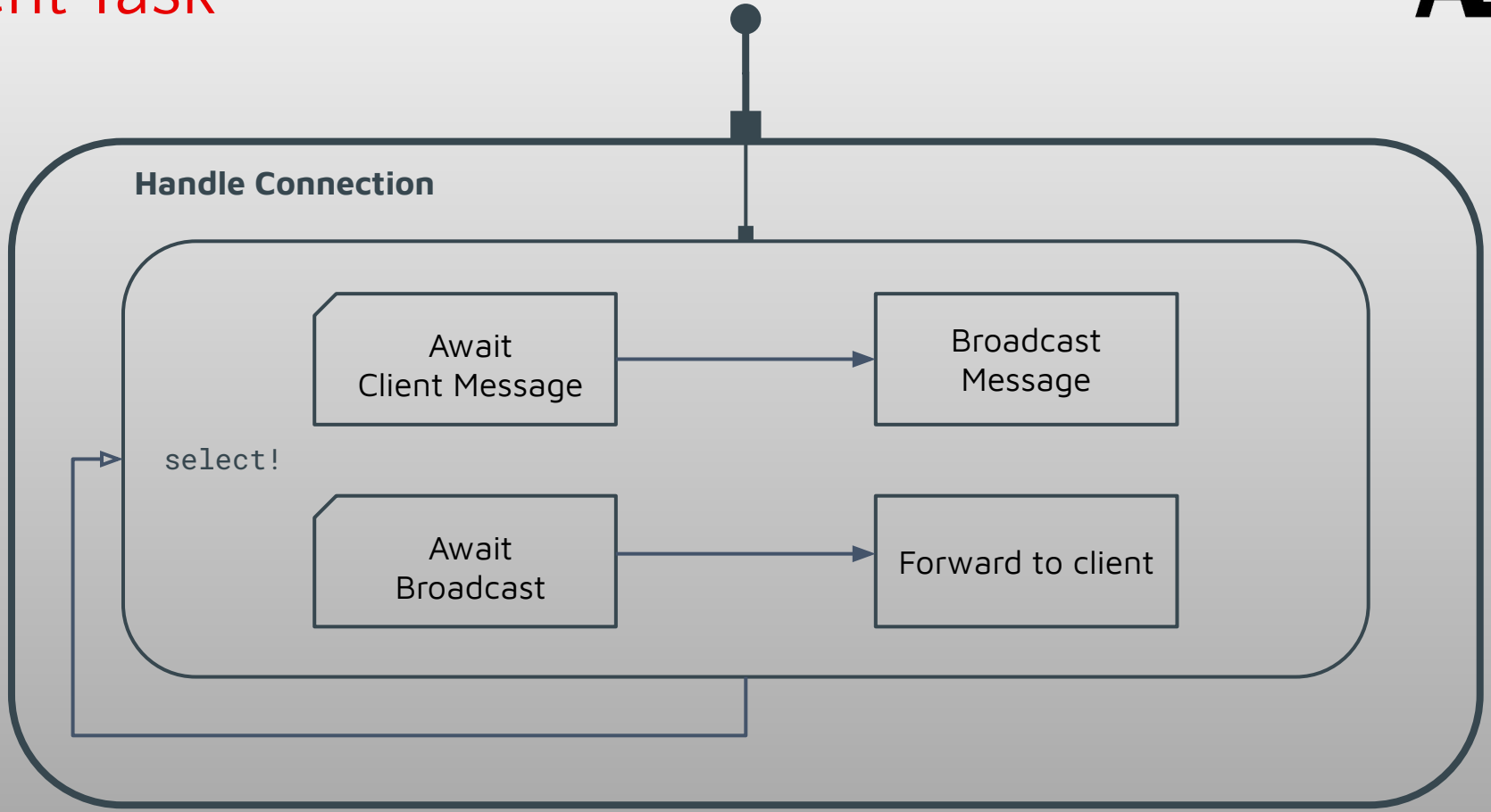
ryhl.io/blog/actors-with-tokio/

Accept Loop



This code on GitHub

Client Task

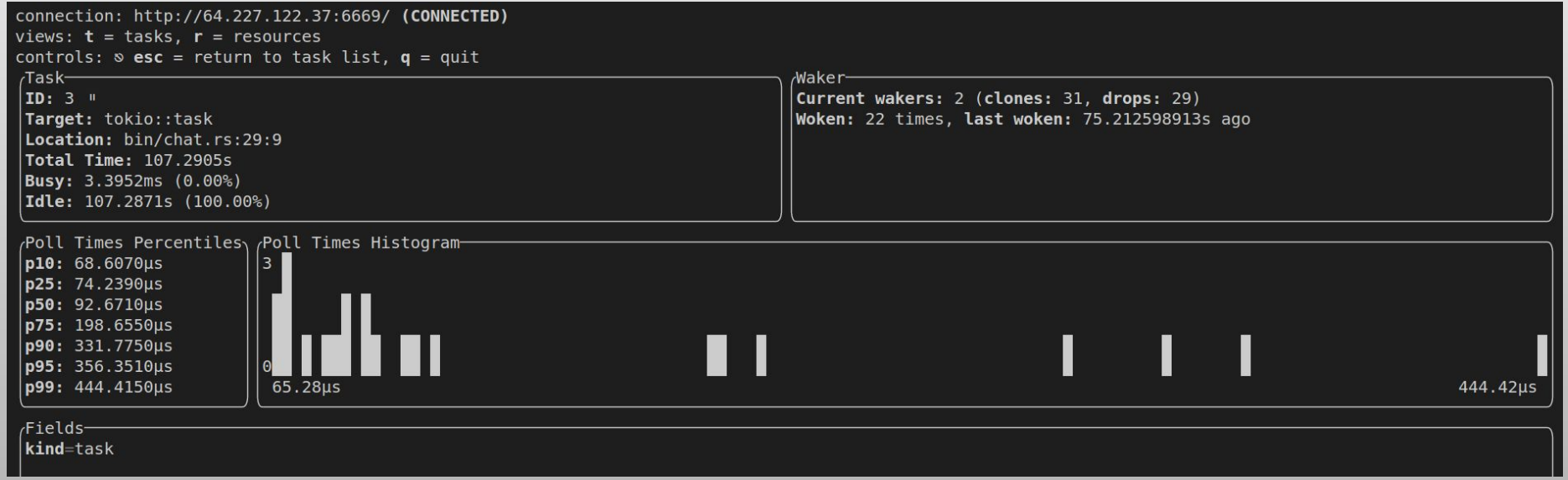


This code on GitHub

Tracing mit Tokio-Console

```
connection: http://64.227.122.37:6669/ (CONNECTED)
views: t = tasks, r = resources
controls: ←→ or h, l = select column (sort), ↑↓ or k, j = scroll, ↵ = view details, i = invert sort (highest/lowest),
scroll to bottom
Tasks (5) ▶ Running (0) " Idle (5)
   Warn  ID  State  Name  Total▿      Busy        Idle       Polls  Target      Location          Fields
>>         1  "             81.7422s   2.3651ms   81.7398s 25     tokio::task bin/chat.rs:29:9  kind=task
           2  "             73.8902s   2.7307ms   73.8875s 26     tokio::task bin/chat.rs:29:9  kind=task
           3  "             42.2902s   3.3952ms   42.2868s 23     tokio::task bin/chat.rs:29:9  kind=task
           4  "             35.7403s   2.7443ms   35.7375s 23     tokio::task bin/chat.rs:29:9  kind=task
           5  "             32.0730s   2.9180ms   32.0701s 23     tokio::task bin/chat.rs:29:9  kind=task
```

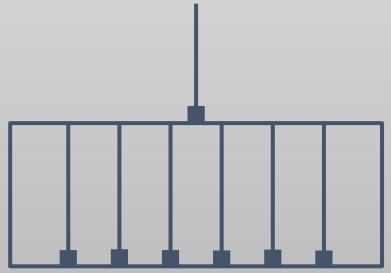
Tracing mit Tokio-Console



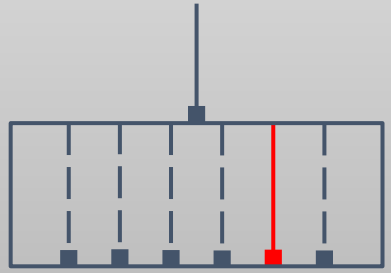
Strukturierte Nebenläufigkeit

Logische Operatoren

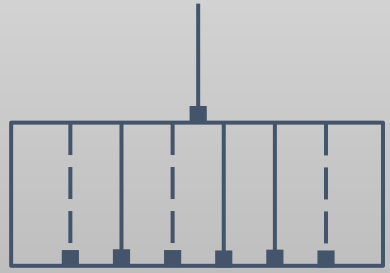
AND



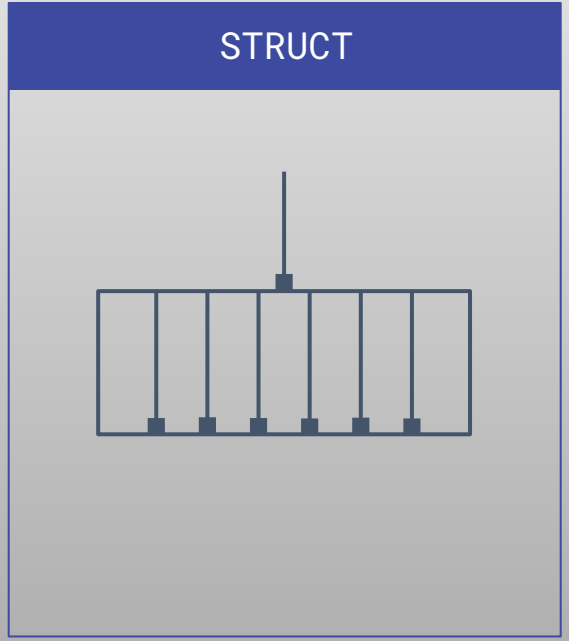
XOR

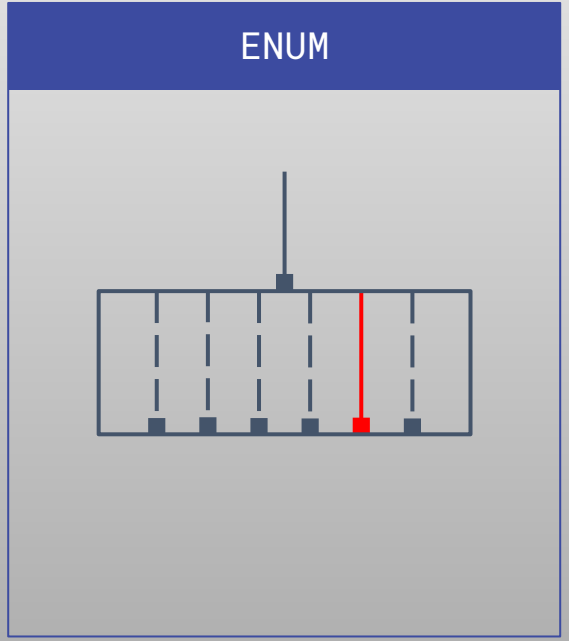


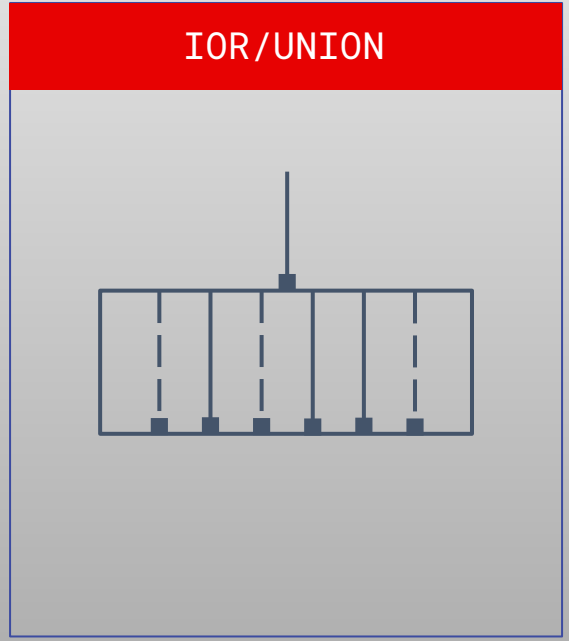
IOR



Datenstrukturen-Primitive

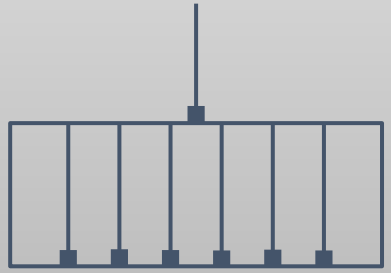




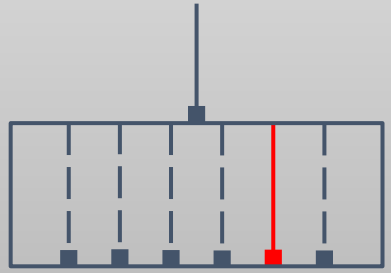


Kontrollfluss-Primitive

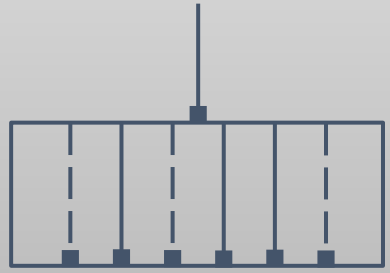
LOOP



IF

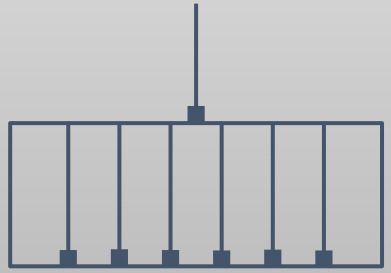


GOTO

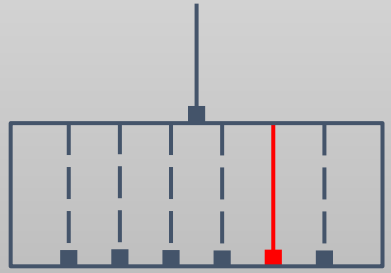


Future-Kombinatoren

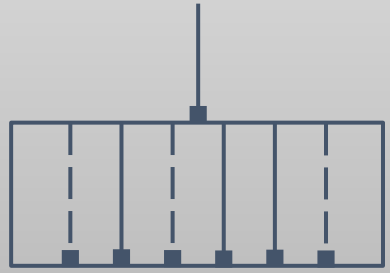
JOIN



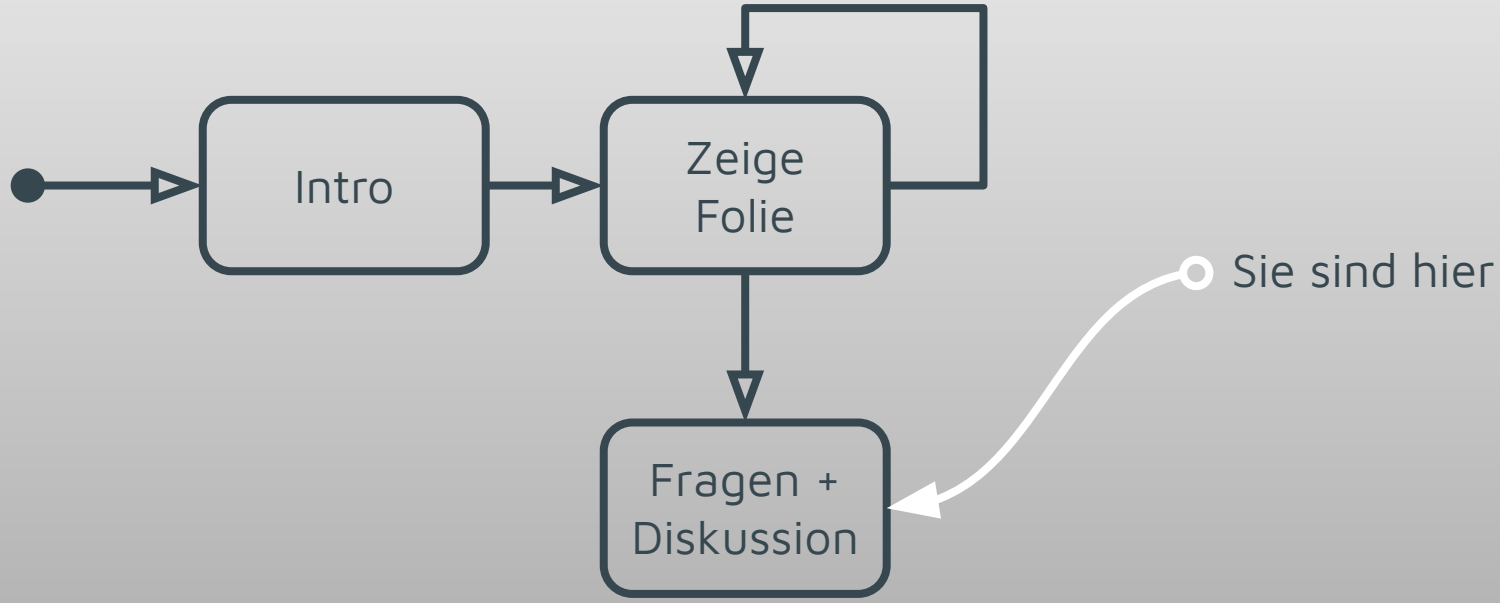
SELECT



GO/DETACH/SPAWN



go statement considered harmful?


Intro
Zeige Folie
Fragen + Diskussion
Sie sind hier


C++