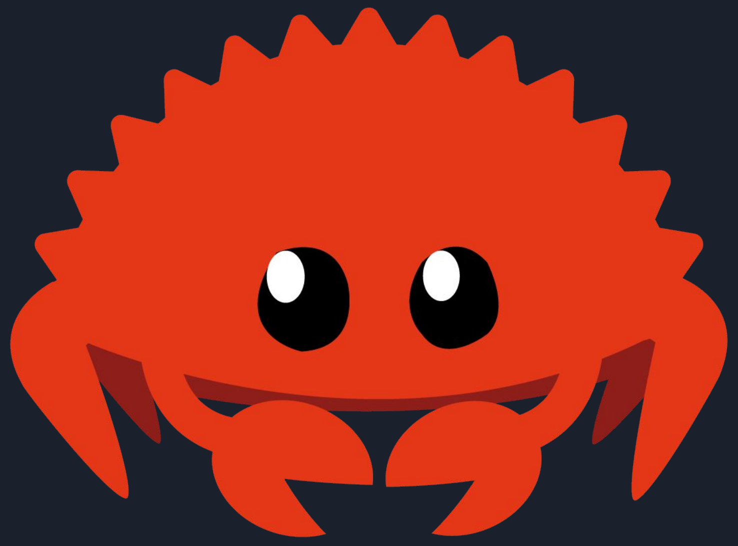


RustConf'24 - Montreal

Actors and Factories in Rust

Usages for distributed processing overload protection

Sean Lawlor
Software Engineer
Aka "a ractor actor"

Pedro Rittner
Production Engineer

Meta

Overview

- Motivation
  - Thrift for RPC @ Meta
  - Rust Thrift Servers @ Meta
  - Why Actors?
- Ractor - a framework
  - Factories
  - Overload protection
- Put it all together

A Very Brief History of Rust @ Meta

Rust @ Meta - Beginnings

- First “serious” Rust usage at Meta started ~2017 with Mononoke in Source Control.
- Rust disadvantaged due to limited interoperability with existing C++ code.
  - `cxx` didn’t exist yet, `bindgen` only worked with C.
- **Problem:** How to enable more Rust adoption?

fbthrift RPC @ Meta

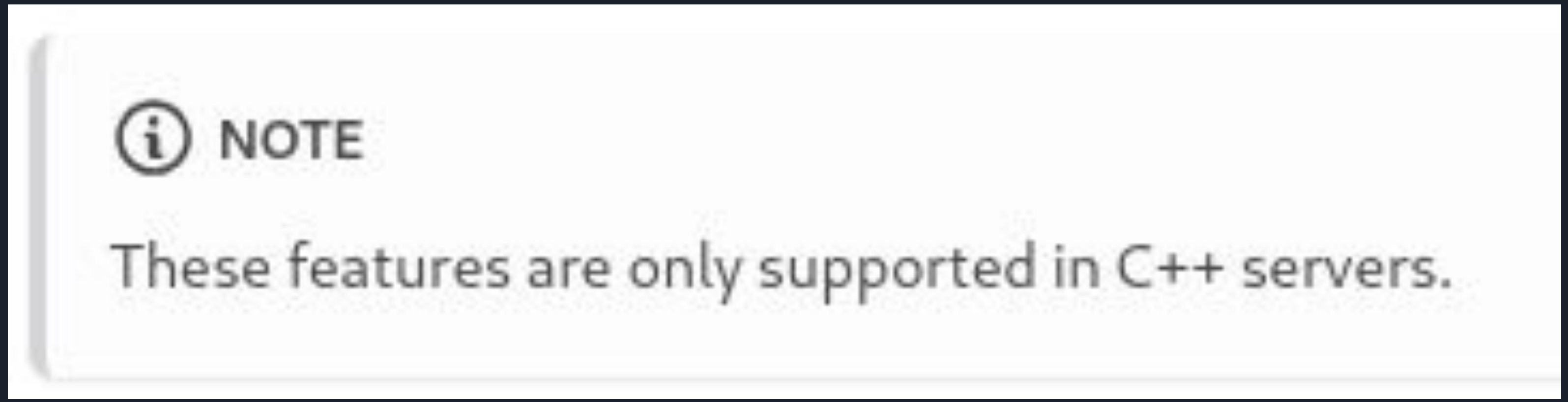
- Many Meta backend services were (are) written in C++ or Python using fbthrift, an IDL and RPC framework forked from Apache Thrift.
    - Scribe - Buffered Distributed Queuing
    - ZippyDB - Distributed Key-Value Database
- **Idea**: Add fbthrift support for Rust, leverage existing language-agnostic frameworks/libraries to build Rust service mesh clients and servers.

YOU GET A THRIFT SERVICE! YOU GET A THRIFT SERVICE!
EVERYBODY GETS A THRIFT SERVICE!
imgflip.com

fbthrift RPC @ Meta

Problem:

Many features added to fbthrift in response to infrastructure needs (e.g. backpressure, overload protection) implemented as *C++ Server middleware.*



ⓘ NOTE

These features are only supported in C++ servers.

Back to square one?

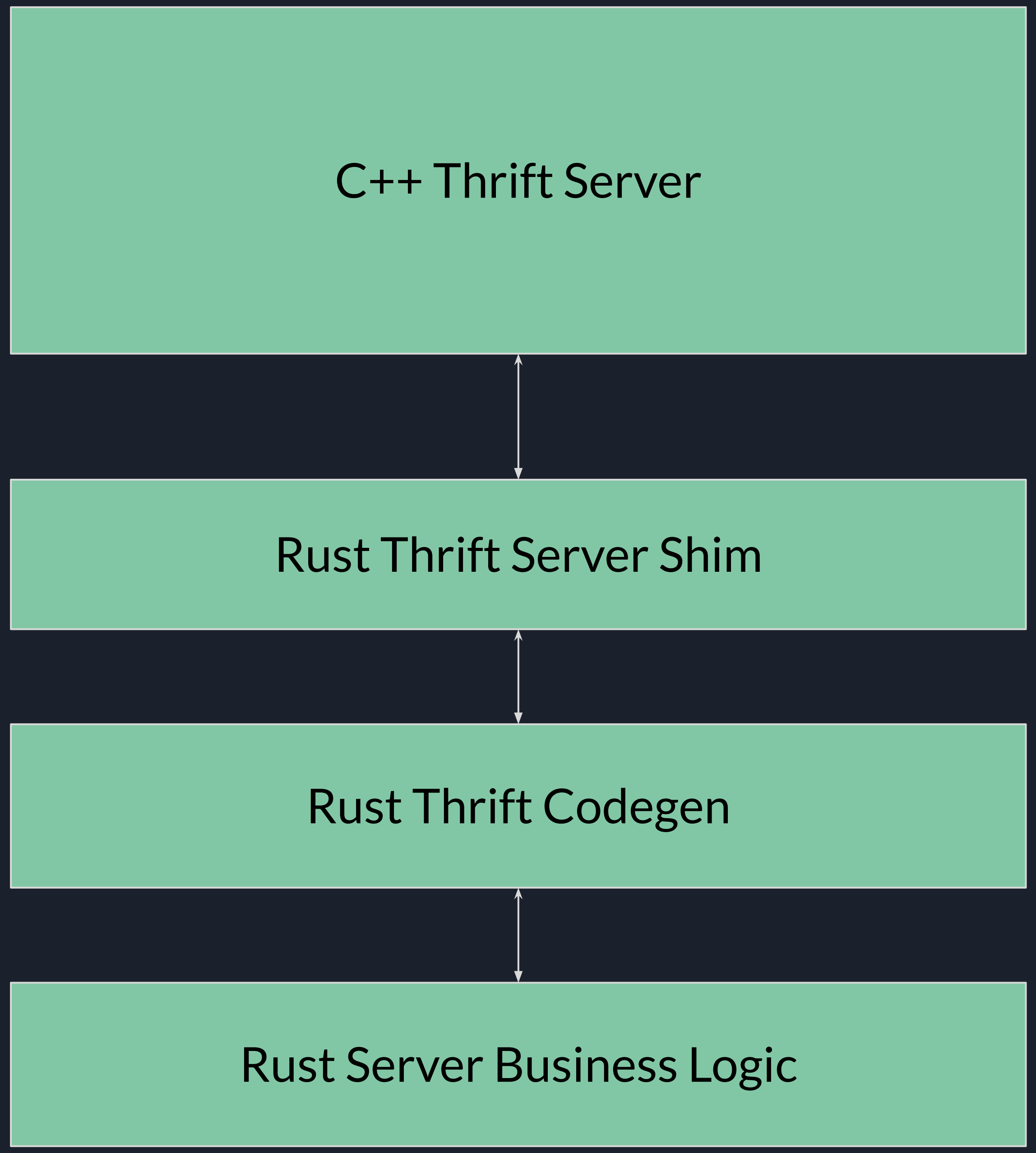
fbthrift + Rust: Initial Approach

- **Idea:** Use `bindgen` (later `cxx`) to manually implement *just enough* FFI “glue” code to offload most work to existing C++ fbthrift server.
  - Codegen for Rust still needed but easier lift (can reuse existing templating code in fbthrift compiler)
  - New features added to C++ fbthrift servers *should* “just work” for Rust fbthrift servers!
- **Problem:** How to integrate Rust and C++?

Basic Rust Thrift Services: Sketch

- **Idea**: Write a “shim” layer between the user’s business logic and the raw C++ Thrift server.
- **C++ handles:** serving traffic on the socket(s) using own threadpools and event loops to do I/O.
- **Rust handles**: request (de)serialization and calling user code by spawning tasks in the `tokio` runtime.

Problem: How to bridge C++ and Rust runtimes?



C++ `folly::coro` vs Rust `async`

C++

- Mix of coroutines, futures, and callbacks
- Futures start executing immediately
- Manage multiple threadpool-based executors
- `libevent` -based event base for I/O operations

Rust

- Futures “hidden” behind `async` keyword
- Futures must be polled to make progress
- Runtimes manage and abstract threadpools
- `mio` -based “reactor” for I/O operations

Problem: How do we reconcile these two approaches/models?

C++ `folly::coro` vs Rust `async`

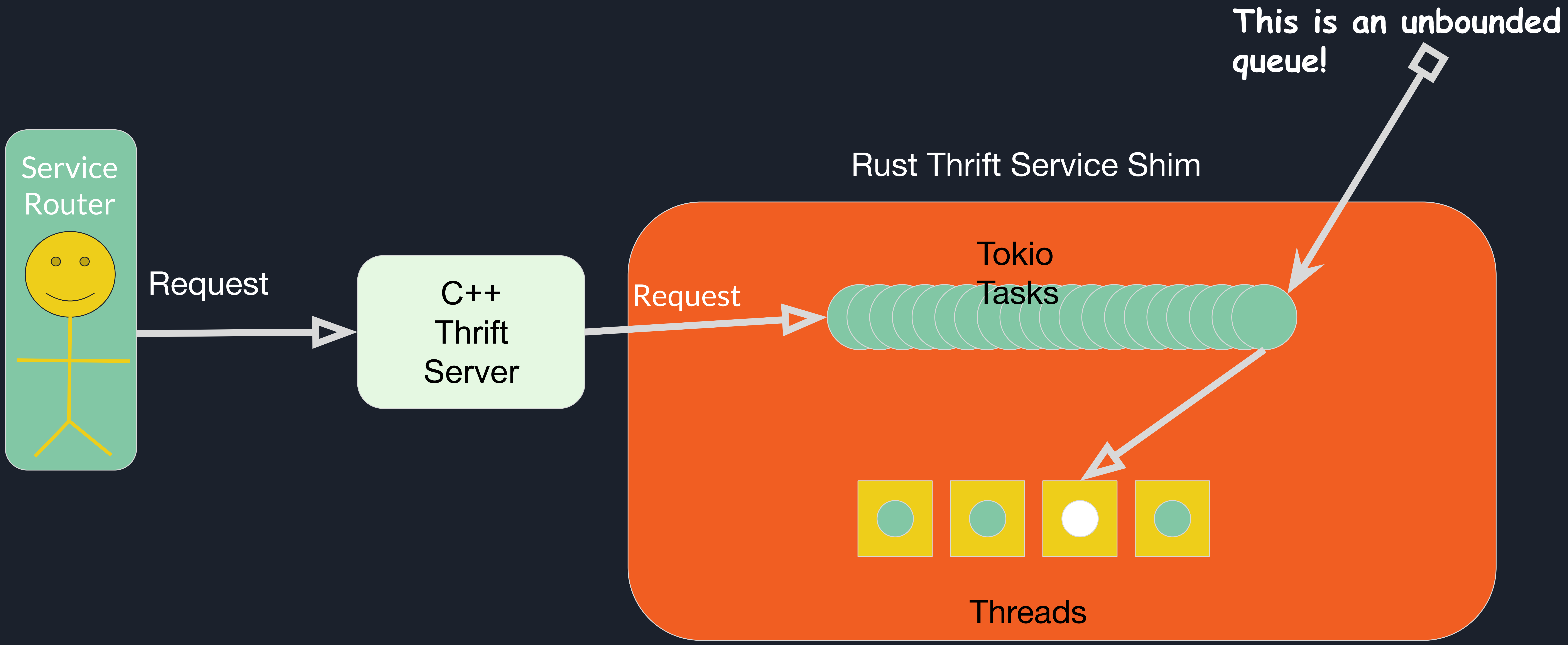
- Mix of cor
- Futures uting immediat
- Manag threadpool-based executo
- `libeve` event base for I/O operations

ust

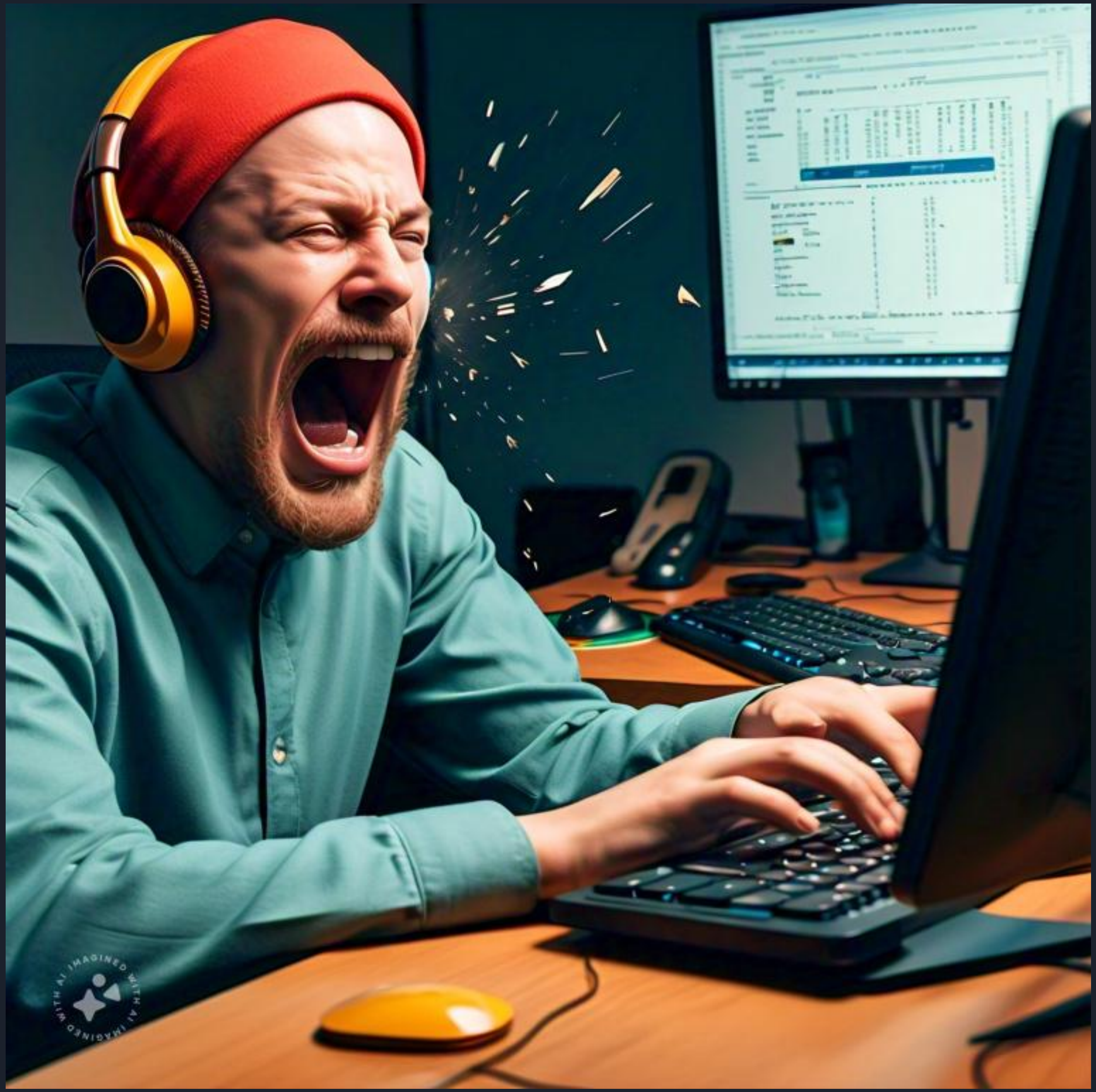
- Futures “hidden” be eyword
- Futures must be polled to gress
- times manage and abst dpools
- eactor” for ions

Pass function objects between them and hope it works!

Basic Rust Thrift Services: Naive Solution



Naive Solution = Many Overload Outages!



“There has to be a better way to do this!”

Enter: Actors

Actors have a great history in large-scale distributed workloads, as well as solving many memory management headaches.

> “An Actor is a computational entity that, in response to a message it receives, can concurrently:
>
> - send a finite number of messages to other Actors;
> - create a finite number of new Actors;
> - designate the behavior to be used for the next message it receives.”
>
> - Carl Hewitt 1973

Why Actors?

- **Isolation and Concurrency:** Actors provide a natural way to isolate concurrent operations, allowing multiple requests to be processed simultaneously.
- **Fault Tolerance:** Actor systems can be designed to handle failures in individual actors, preventing a single failure from bringing down the entire system.
- **Scalability**: Actors can be easily distributed across multiple nodes, making it simple to scale the system horizontally as needed.
- **Simplified State Management:** Actors encapsulate their own state, reducing the complexity of managing global state and making it easier to reason about the system's behavior.

Ractor:
A New Framework

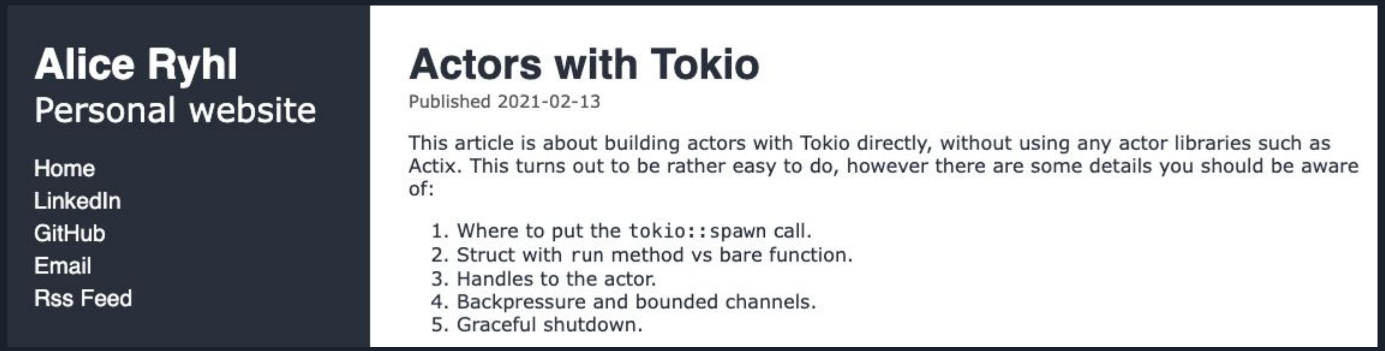
Why a new framework?

There are already multiple frameworks for actors in rust

Actix

BASTION

Riker



Alice Ryhl
Personal website

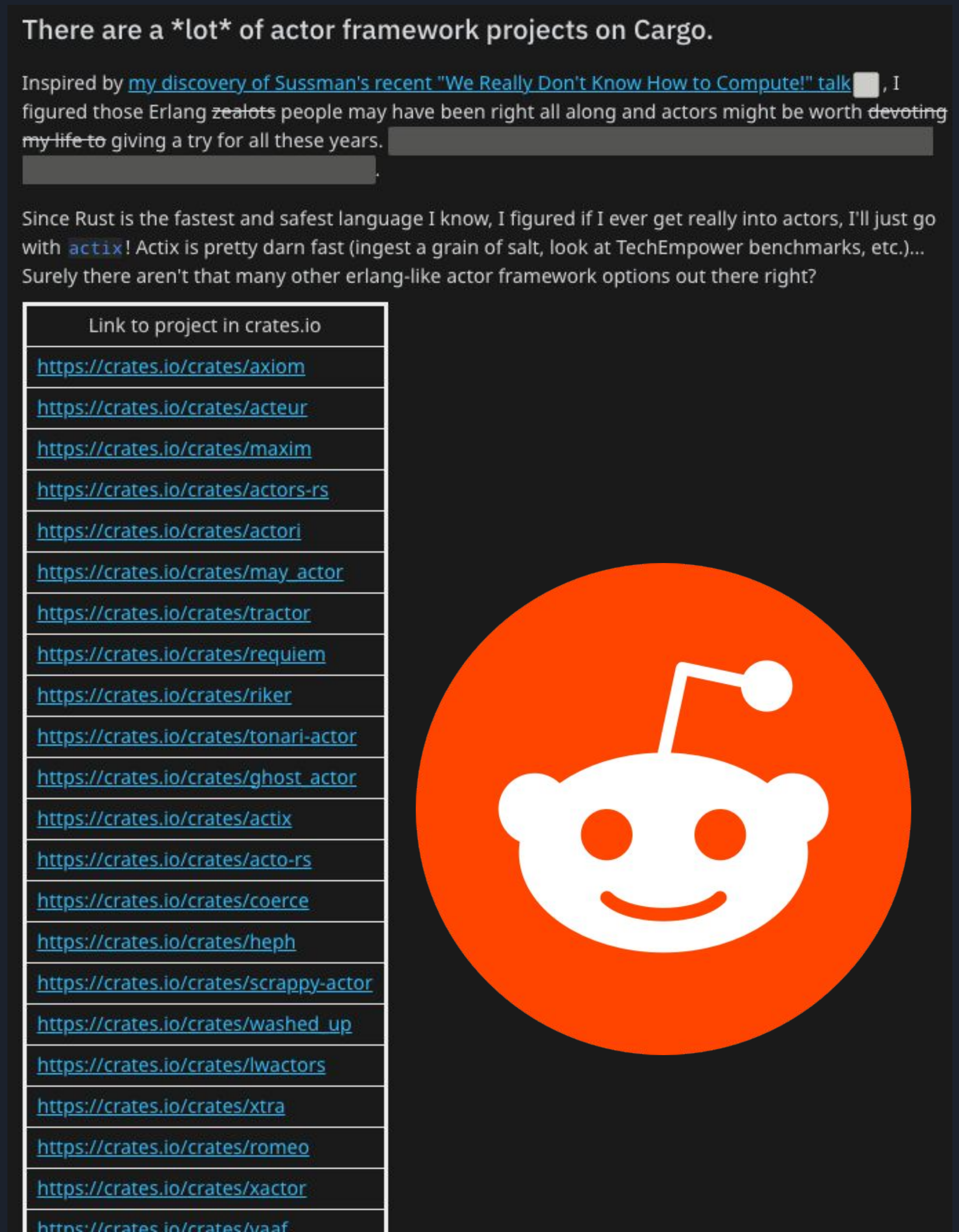
Home
LinkedIn
GitHub
Email
Rss Feed

Actors with Tokio

Published 2021-02-13

This article is about building actors with Tokio directly, without using any actor libraries such as Actix. This turns out to be rather easy to do, however there are some details you should be aware of:

1. Where to put the `tokio::spawn` call.
2. Struct with run method vs bare function.
3. Handles to the actor.
4. Backpressure and bounded channels.
5. Graceful shutdown.



There are a *lot* of actor framework projects on Cargo.

Inspired by my discovery of Sussman's recent "We Really Don't Know How to Compute!" talk, I figured those Erlang ~~zealots~~ people may have been right all along and actors might be worth ~~devoting my life to~~ giving a try for all these years. [redacted].

Since Rust is the fastest and safest language I know, I figured if I ever get really into actors, I'll just go with `actix`! Actix is pretty darn fast (ingest a grain of salt, look at TechEmpower benchmarks, etc.)... Surely there aren't that many other erlang-like actor framework options out there right?

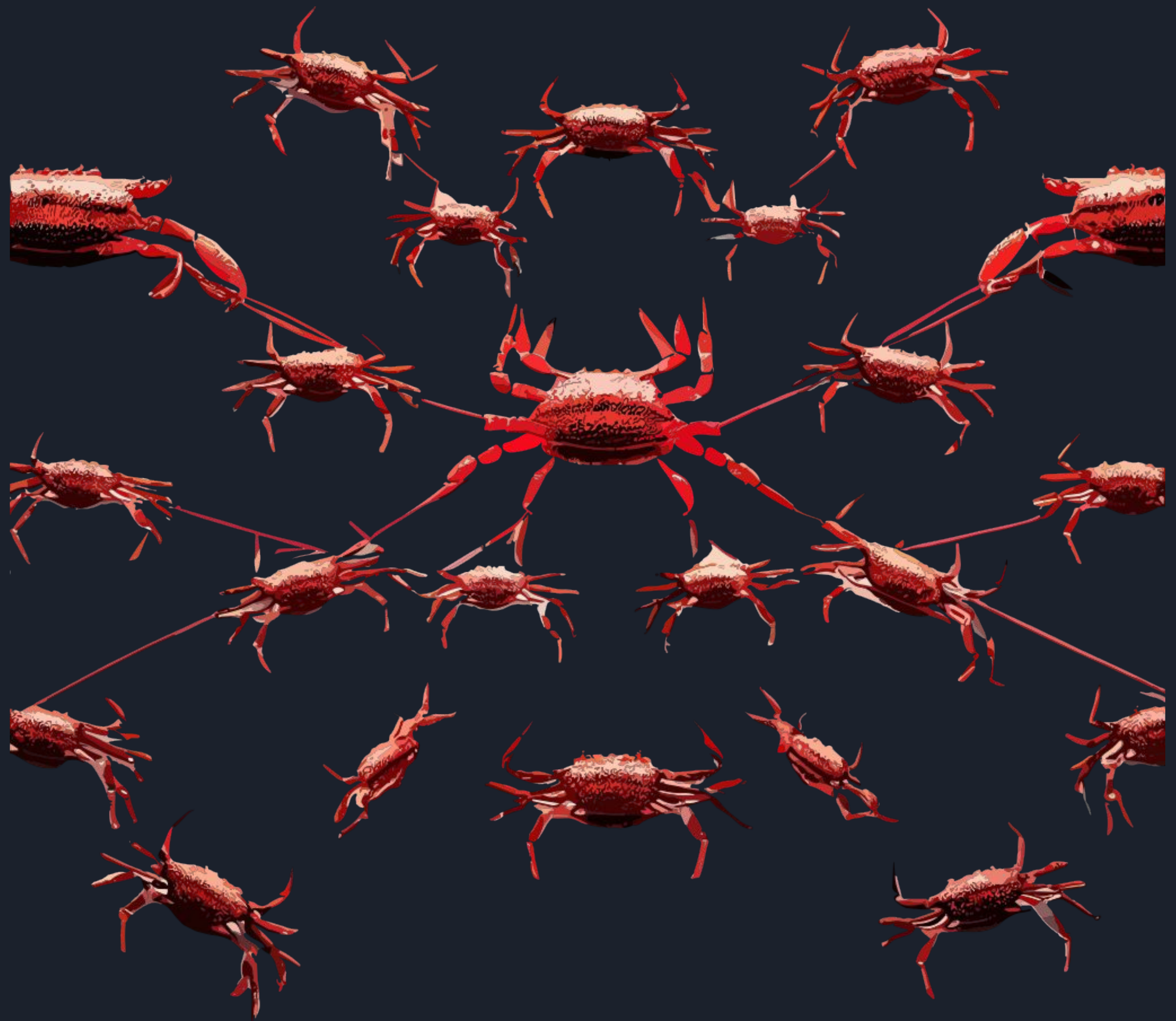
Link to project in crates.io
https://crates.io/crates/axiom
https://crates.io/crates/acteur
https://crates.io/crates/maxim
https://crates.io/crates/actors-rs
https://crates.io/crates/actori
https://crates.io/crates/may_actor
https://crates.io/crates/tractor
https://crates.io/crates/requiem
https://crates.io/crates/riker
https://crates.io/crates/tonari-actor
https://crates.io/crates/ghost_actor
https://crates.io/crates/actix
https://crates.io/crates/acto-rs
https://crates.io/crates/coerce
https://crates.io/crates/heph
https://crates.io/crates/scrappy-actor
https://crates.io/crates/washed_up
https://crates.io/crates/lwactors
https://crates.io/crates/xtra
https://crates.io/crates/romeo
https://crates.io/crates/xactor
https://crates.io/crates/yaaf

Why a new framework?

The problems with these crates that we've found is that they're…

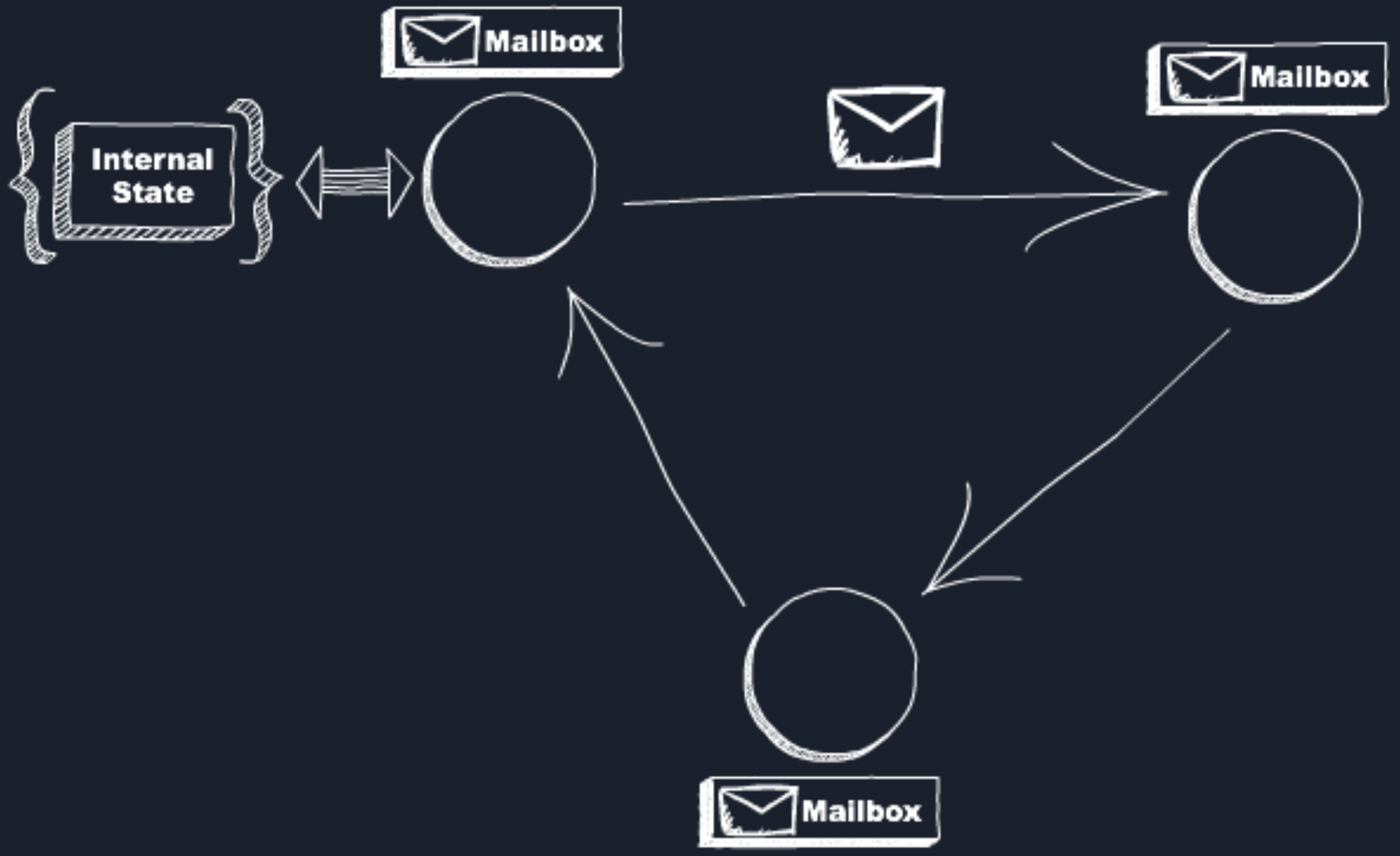
- Dead/unsupported
- Customized a lot of workflows away from Erlang's principles
- Not flexible enough
- Have custom runtimes / don't play nice with Tokio.

So… **Ractor!**



What is ractor all about?

- Isolated actor processes with optional state
- Actor Supervision trees through a basic trait
- Remote procedure call - **mod** rpc
- Timers - **mod** time
- Named registry - **mod** registry
- Process groups - **mod** pg
- **Factories** - **mod** factory



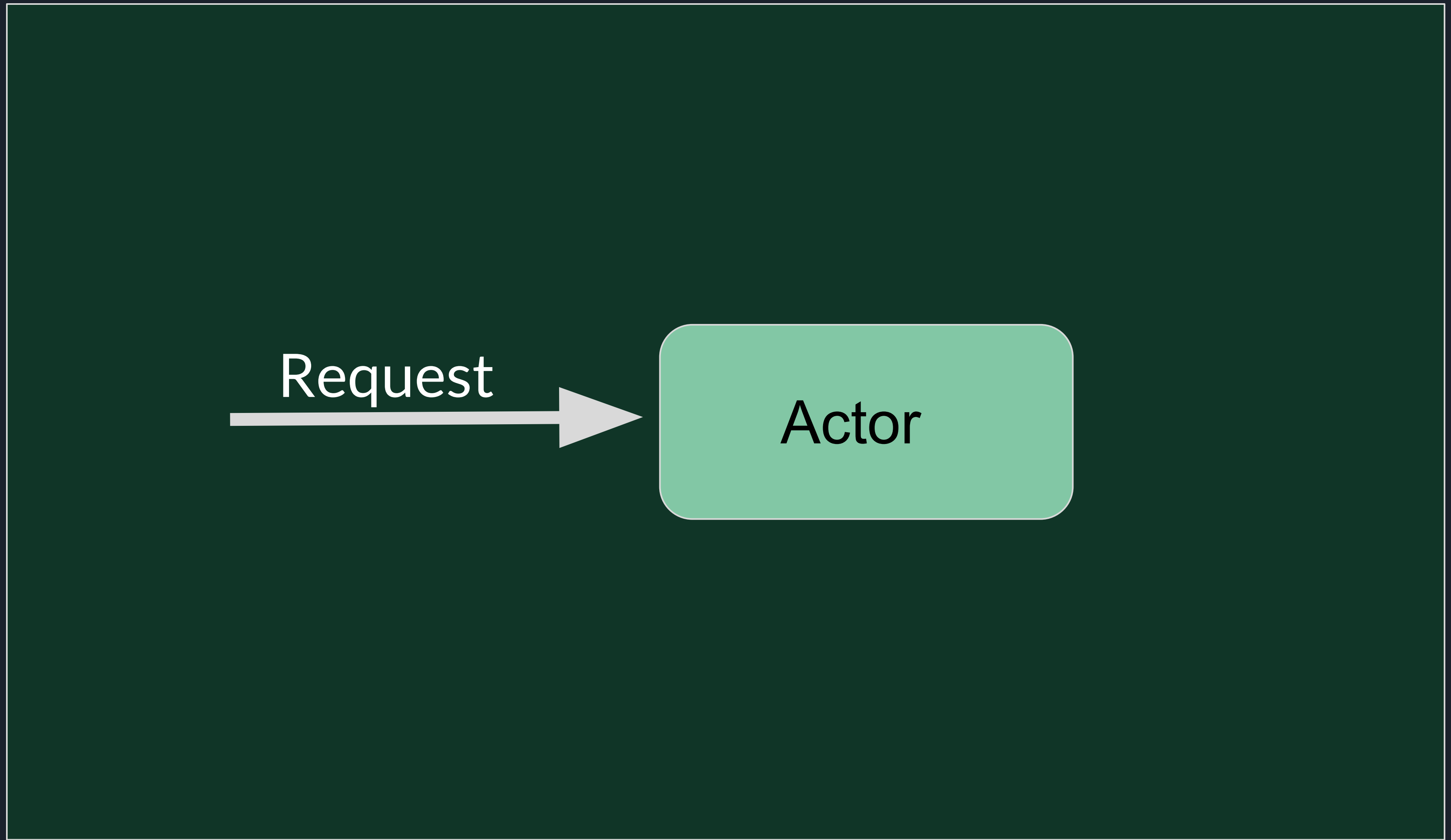
Factories

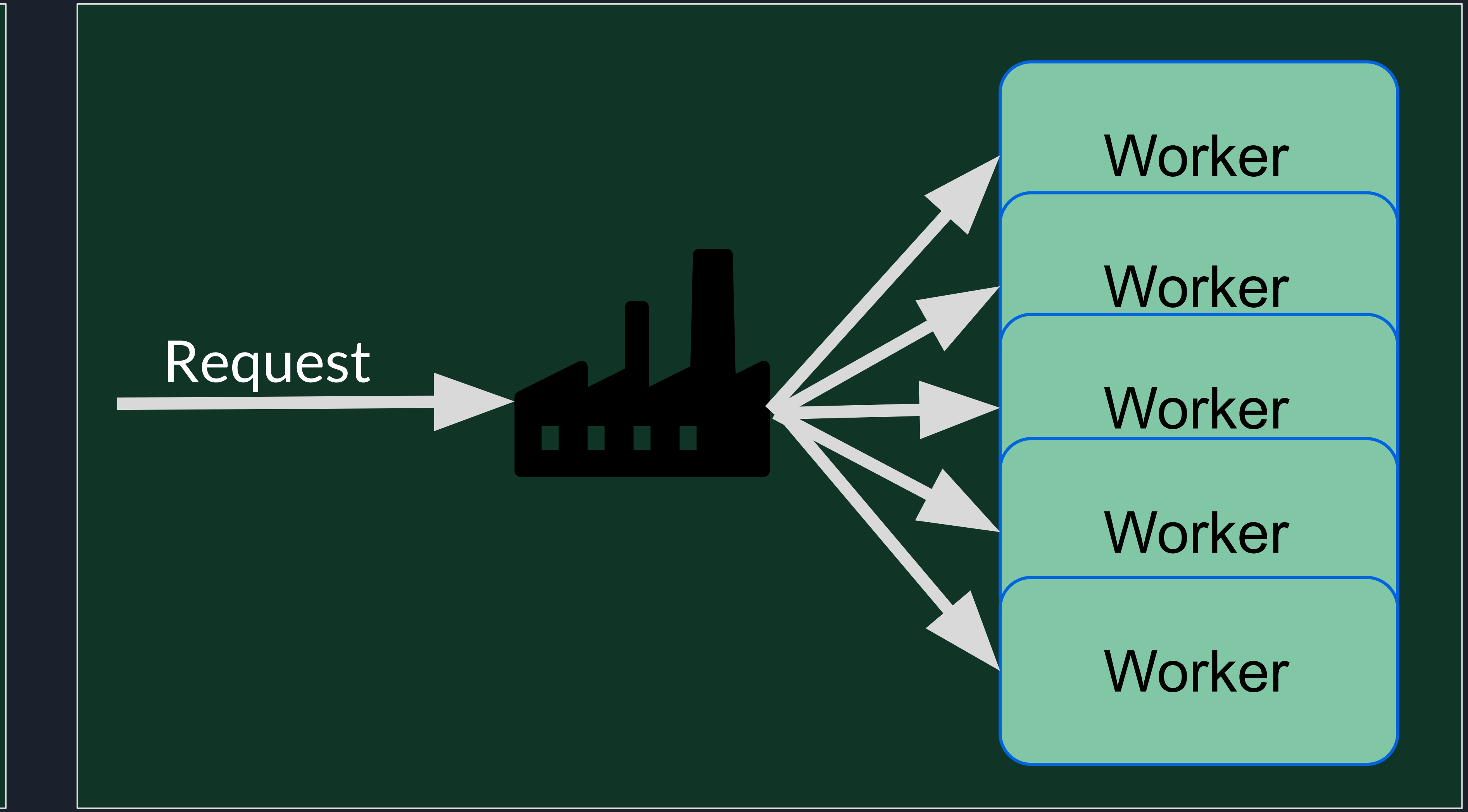
Not a factory method!

Factories

What happens if a single actor can't handle the load?

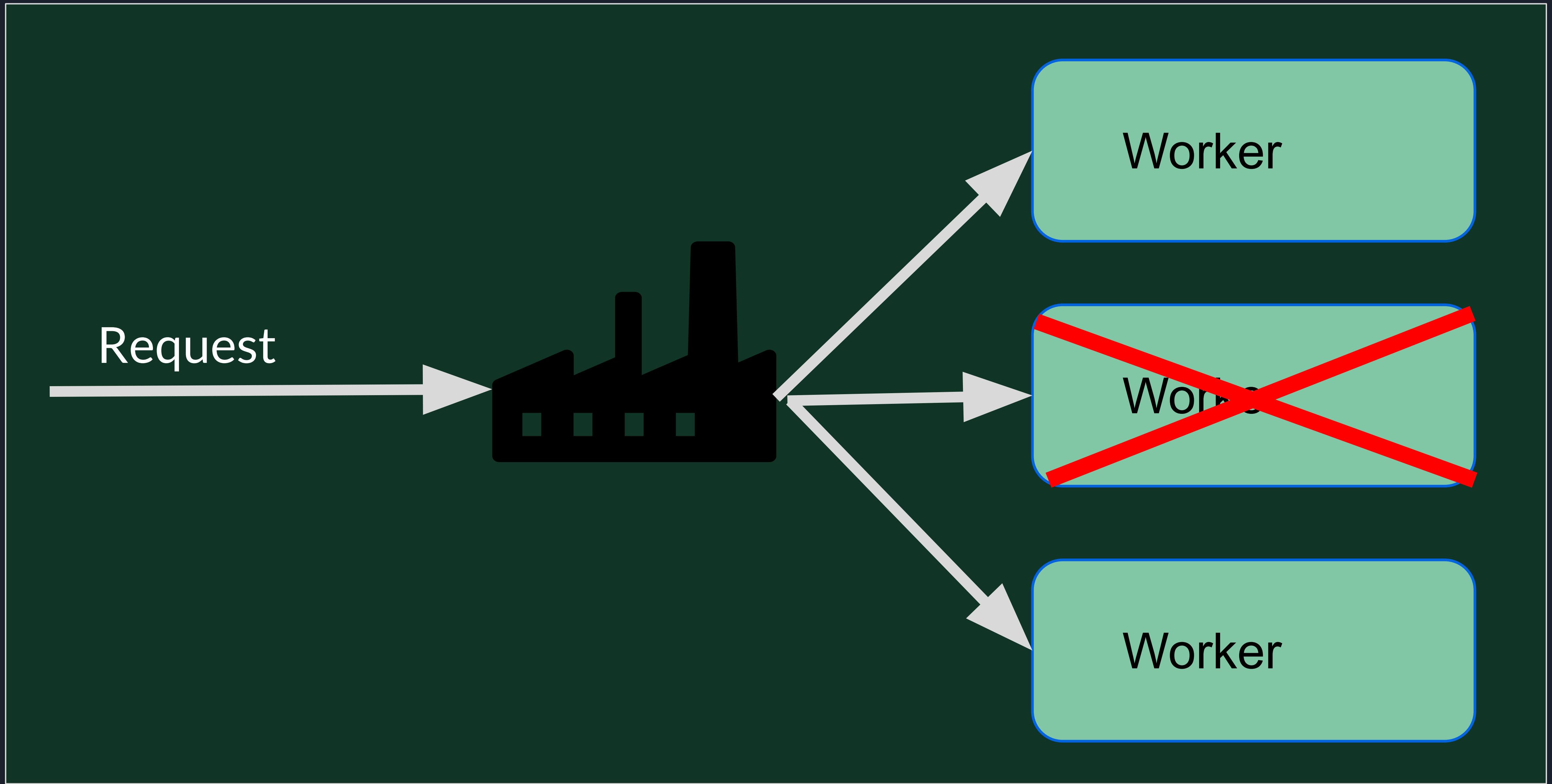
Work distribution





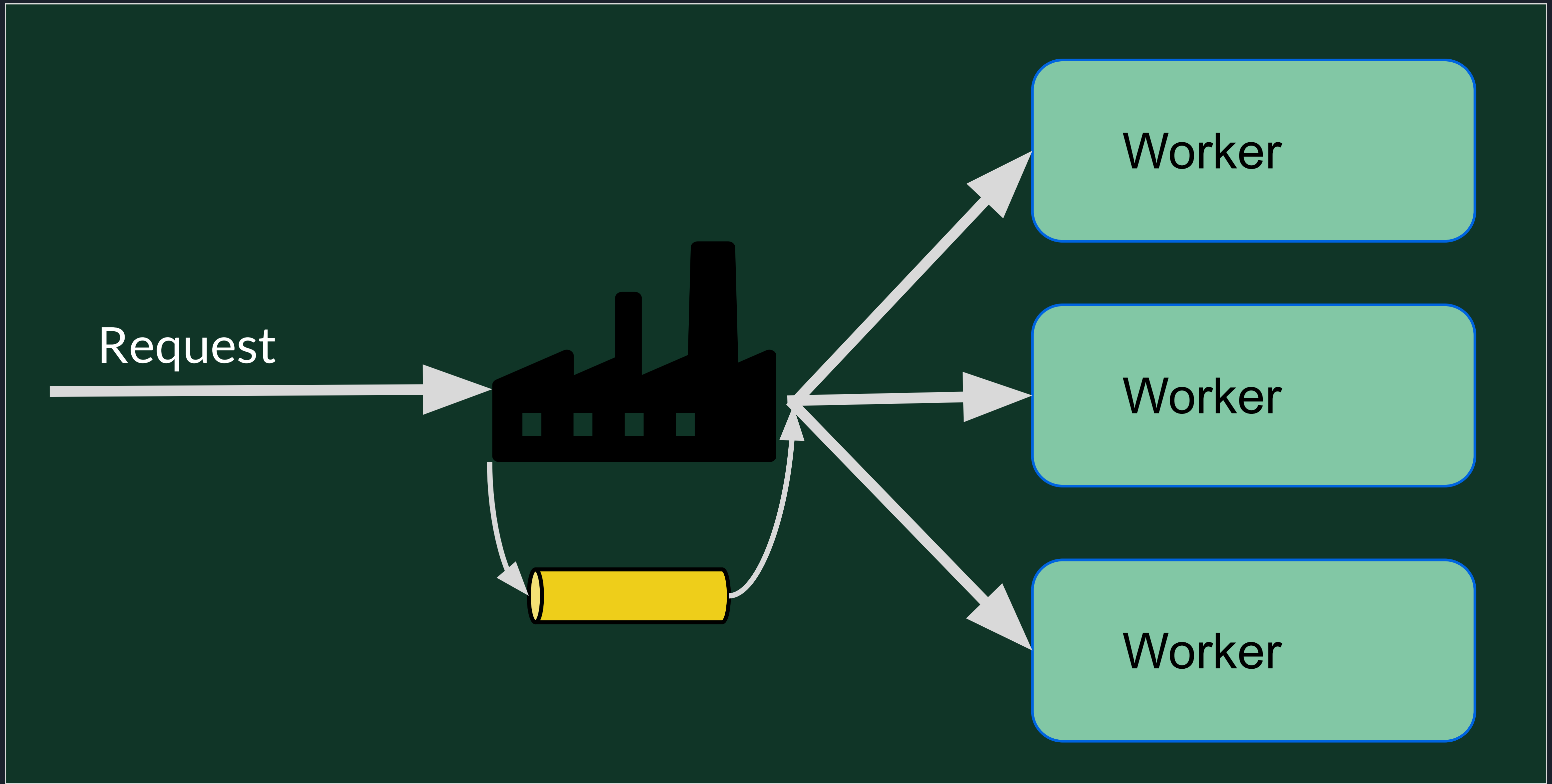
Factories

Worker lifecycle management



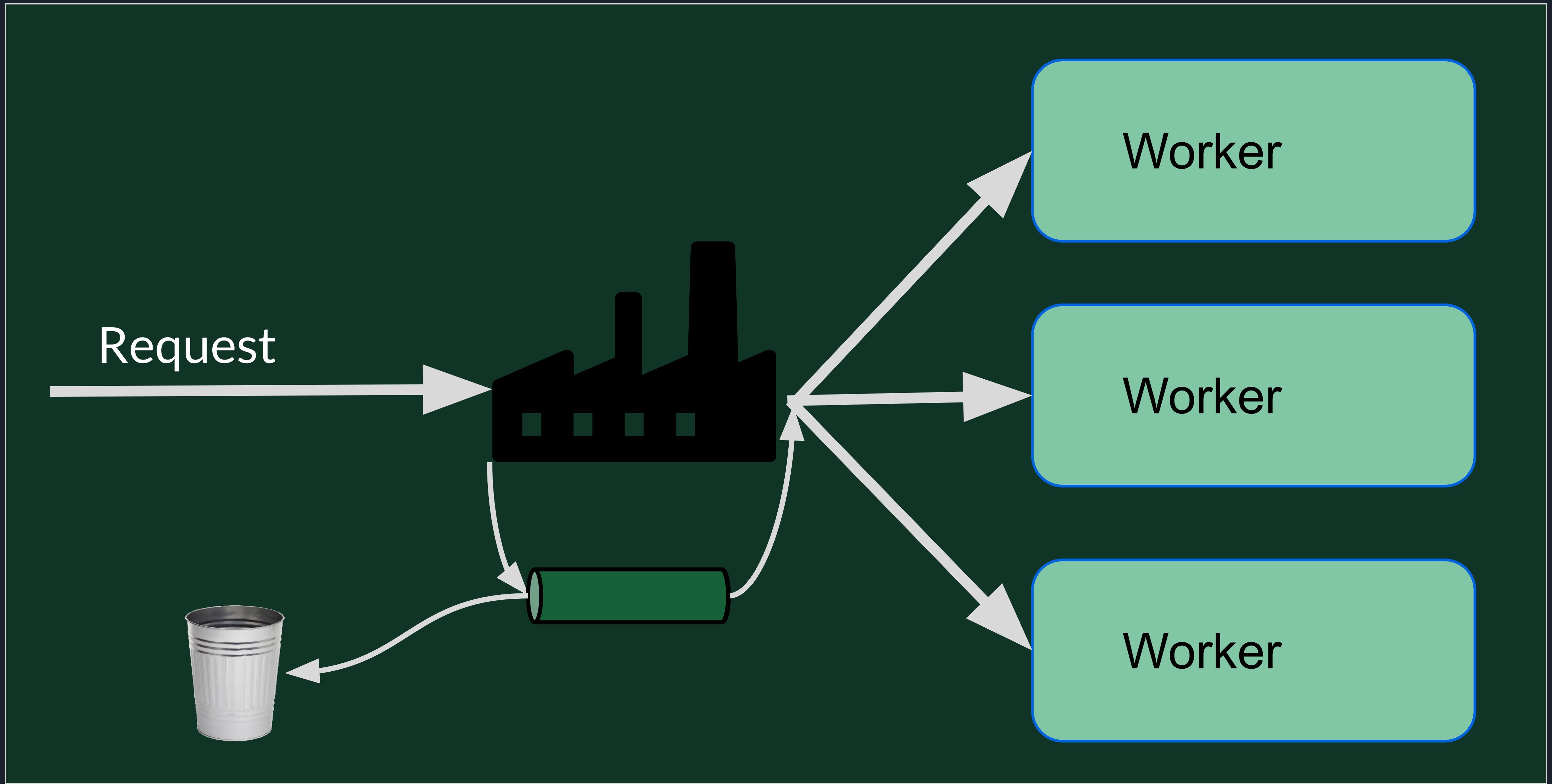
Factories

Queue backlog management



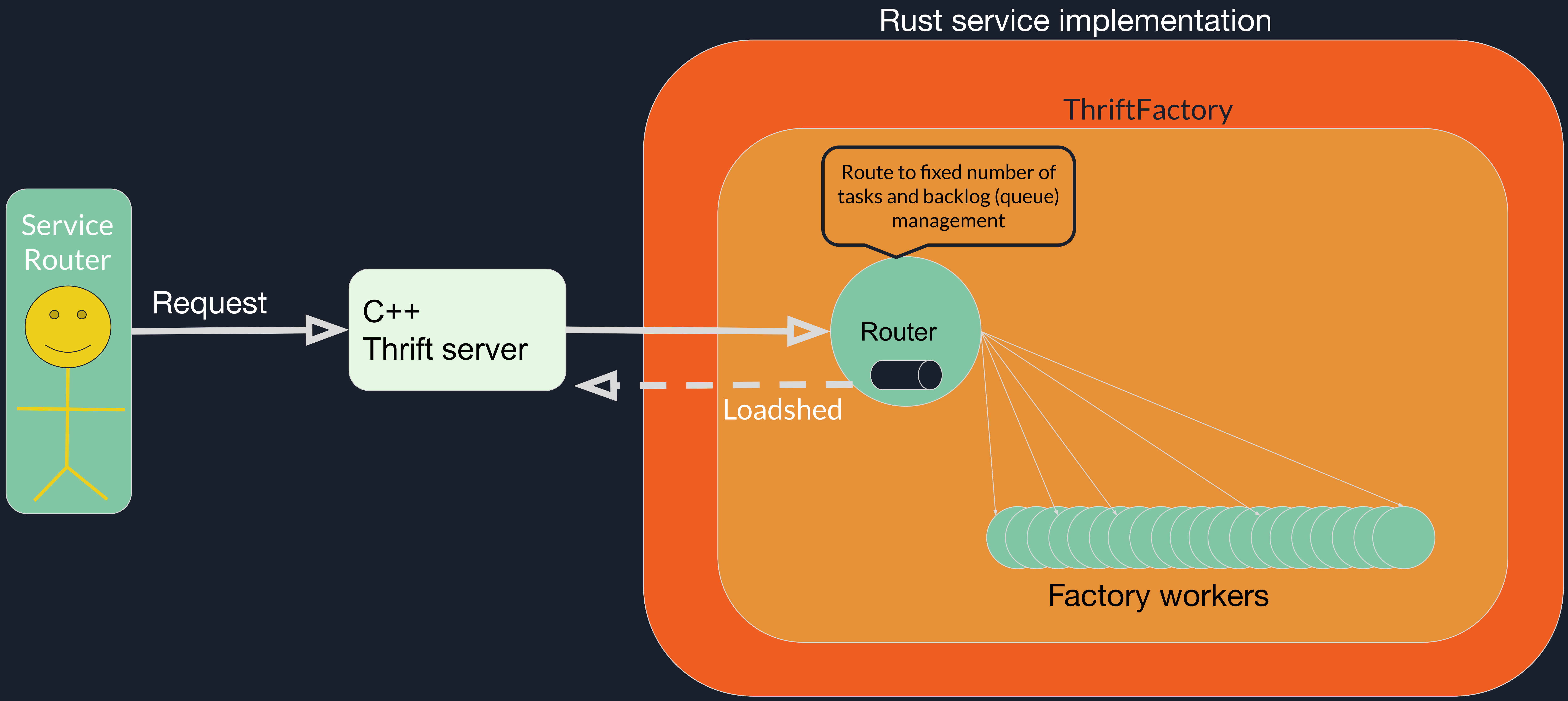
Factories

Job discarding / rejection



Factories and Thrift Servers

Concurrency limited Rust Thrift Server


Rust service implementation
ThriftFactory
Route to fixed number of tasks and backlog (queue) management
Service Router
Request
C++
Thrift server
Router
Loadshed
Factory workers

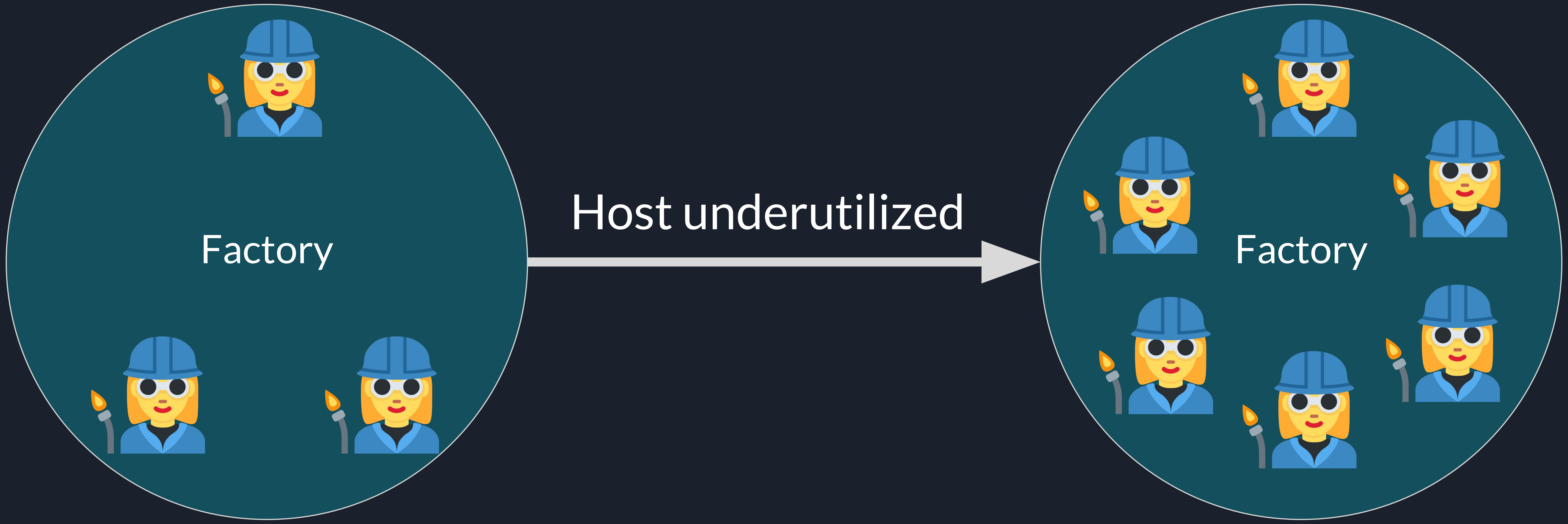
Controlling queue depth

- Jobs sitting in the thrift queue increase overall job processing delays.
- If the host is overloaded/slow then messages may be needlessly delayed when other hosts are free

A common metric to use for queue depth management is **message latency**

Controlling processing concurrency

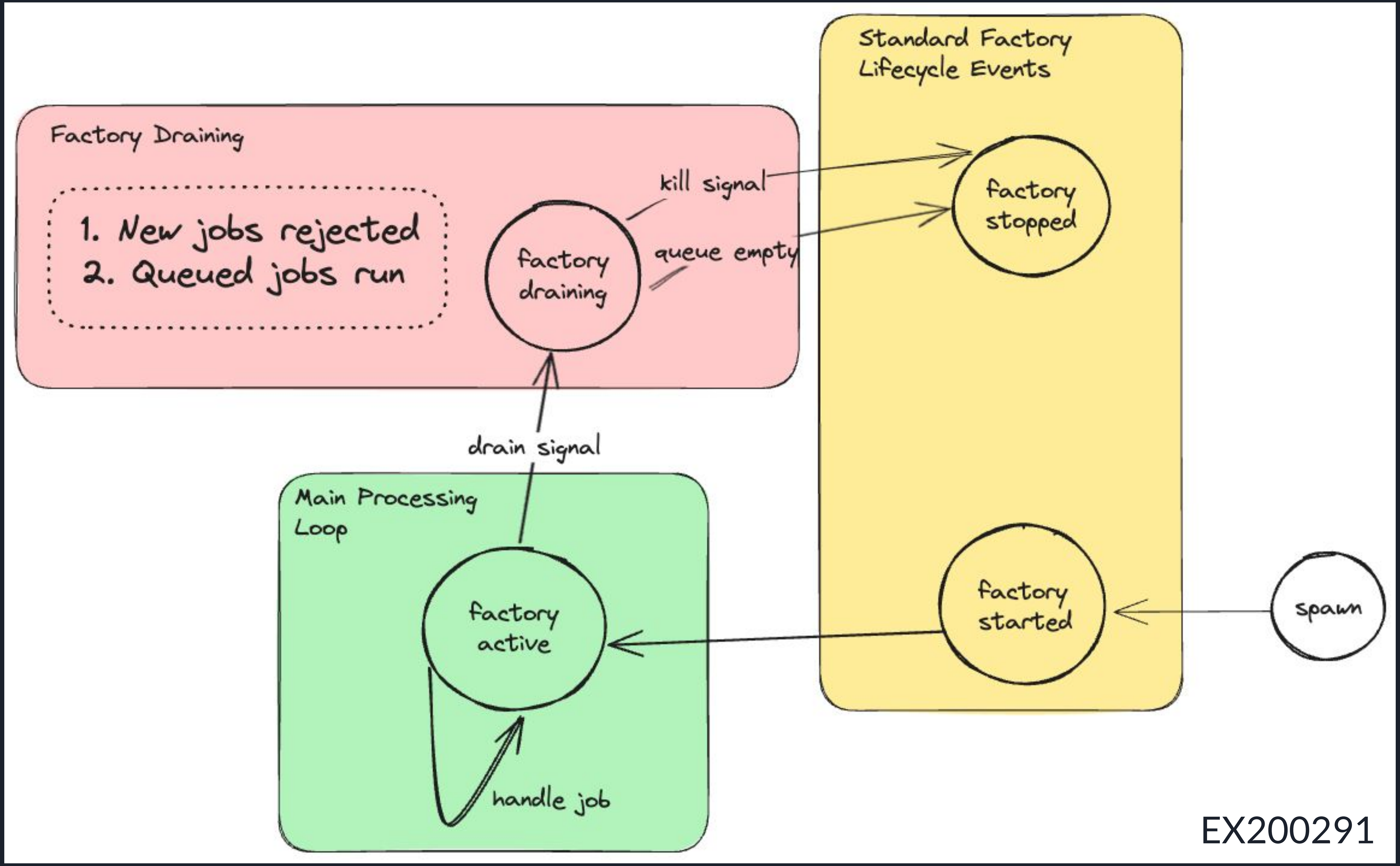
- If the host is under/over-loaded the host should scale the worker count to target a goal resource utilization
- Common metrics used for host utilization
  - CPU
  - Memory
  - Network
  - Or a combination



Request draining

What happens when a running service is updated and there are current requests being processed?

Request draining & Lifecycle management



EX200291

Conclusion

- Meta has a heavy service-to-service dependency on Thrift
- Rust had to adapt to the existing framework to survive in Meta
- Due to implications of threading models, we couldn’t use the existing loadshedding frameworks
- Built with Tokio in-mind, *ractor* adds safe concurrency for processing Thrift requests
- Factories help scale outside of a single task loop *safely*

Links

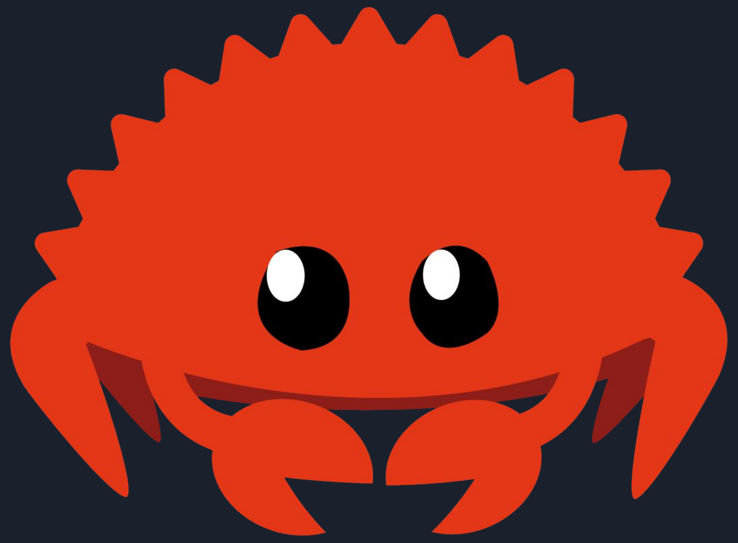
Ractor: actors for Rust - https://github.com/slawlor/ractor

Facebook Thrift - https://github.com/facebook/fbthrift

Rust-shed extensions - https://github.com/facebookexperimental/rust-shed

Factories - Introduced by WhatsApp for Erlang

- https://www.youtube.com/watch?v=c12cYAUTXXs



Meta