Rust Ecosystem

The State of Async Rust: Runtimes

Recently, I found myself returning to a compelling series of blog posts titled Zero-cost futures in Rust by Aaron Turon about what would become the foundation of Rust's async ecosystem and the Tokio runtime.

This series stands as a cornerstone in writings about Rust. People like Aaron are the reason why I wanted to be part of the Rust community in the first place.

While 2016 evokes nostalgic memories of excitement and fervor surrounding async Rust, my sentiments regarding the current state of its ecosystem are now somewhat ambivalent.

Why Bother?

Through this series, I hope to address two different audiences:

- Newcomers to async Rust, seeking to get an overview of the current state of the ecosystem.
- Library maintainers and contributors to the async ecosystem, in the hope that my perspective can be a basis for discussion about the future of async Rust.

In the first article, we will focus on the current state of async Rust runtimes, their design choices, and their implications on the broader Rust async ecosystem.

One True Runtime

An inconvenient truth about async Rust is that libraries still need to be written against individual runtimes. Writing your async code in a runtime-agnostic fashion requires conditional compilation, compatibility layers and handling edge-cases.

This is the rationale behind most libraries gravitating towards the One True Runtime — Tokio.

Executor coupling is a big problem for async Rust as it breaks the ecosystem into silos. Documentation and examples for one runtime don't work with the other runtimes.

Moreover, much of the existing documentation on async Rust feels outdated or incomplete. For example, the async book remains in draft, with concepts like `FuturesUnordered` yet to be covered. (There is an open pull request, though.)

That leaves us with a situation that is unsatisfactory for everyone involved:

- For new users, it is a big ask to navigate this space and make future-proof decisions.
- For experienced users and library maintainers, supporting multiple runtimes is an additional burden. It's no surprise that popular crates like `reqwest` simply insist on Tokio as a runtime.

This close coupling, recognized by the async working group, has me worried about its potential long-term impact on the ecosystem.

The case of `async-std`

`async-std` was an attempt to create an alternative runtime that is closer to the Rust standard library. Its promise was that you could almost use it as a drop-in replacement.

Take, for instance, this straightforward synchronous file-reading code:

```rust
use std::fs::File;
use std::io::Read;

fn main() -> std::io::Result<()> {
    let mut file = File::open("foo.txt")?;
    let mut data = vec![];
    file.read_to_end(&mut data)?;
    Ok(())
}
```

In `async-std`, it is an async operation instead:

```rust
use async_std::prelude::*;
use async_std::fs::File;
use async_std::io;

async fn read_file(path: &str) -> io::Result<()> {
    let mut file = File::open(path).await?;
    let mut data = vec![];
    file.read_to_end(&mut data).await?;
    Ok(())
}
```

The only difference is the `await` keyword.

While the name might suggest it, `async-std` is not a drop-in replacement for the standard library as there are many subtle differences between the two.

It is hard to create a runtime that is fully compatible with the standard library. Here are some examples of issues that are still open:

- New thread is spawned for every I/O request
- OpenOptionsExt missing for Windows?
- Spawned task is stuck during flushing in File.drop()

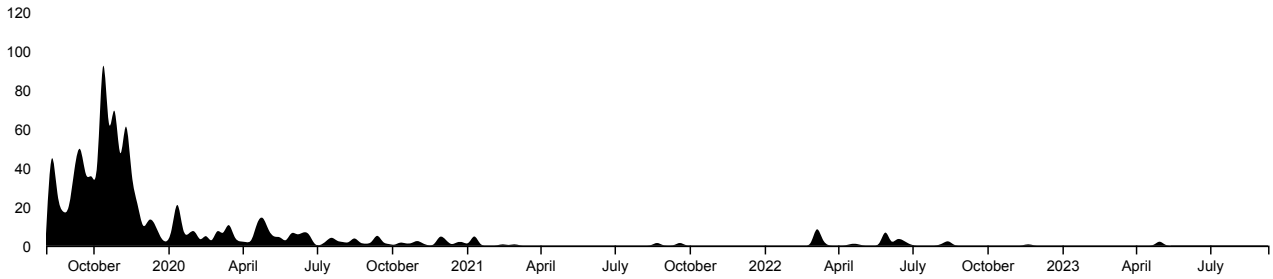
It is an enormous effort to replicate the standard library and it is not clear to me if it is worth it.

Even if it *were* a drop-in replacement, I'd still ponder its actual merit. Rust is a language that values explicitness. This is especially true for reasoning about runtime behavior, such as allocations and blocking operations. The async-

std's teams proposal to "Stop worrying about blocking" was met with noticeable community skepticism and later retracted.

As of this writing, 1754 public crates have a dependency on `async-std` and there are companies that rely on it in production.

However, looking at the commits over time `async-std` is essentially abandoned as there is no active development anymore:



This leaves those reliant on the `async-std` API – be it for concurrency mechanisms, extension traits, or otherwise – in an unfortunate situation, as is the case for libraries developed on top of `async-std`, such as `surf`. The core of `async-std` is now powered by `smol`, but it is probably best to use it directly for new projects.

Can't we just embrace Tokio?

Tokio stands as Rust's canonical async runtime. But to label Tokio merely as a runtime would be an understatement. It has extra modules for fs, io, net, process- and signal handling and more. That makes it more of a framework for asynchronous programming than just a runtime.

Partially, this is because Tokio had a pioneering role in async Rust. It explored the design space as it went along. And while you could exclusively use the runtime and ignore the rest, it is easier and more common to buy into the entire ecosystem.

Yet, my main concern with Tokio is that it makes a lot of assumptions about how async code should be written and where it runs.

For example, at the beginning of the Tokio documentation, they state:

"The easiest way to get started is to enable all features. Do this by enabling the `full` feature flag":

```
tokio = { version = "1", features = ["full"] }
```

By doing so, one would set up a multi-threaded runtime which mandates that types are `Send` and `'static` and makes it necessary to use synchronization primitives such as `Arc` and `Mutex` for all but the most trivial applications.

> *The Original Sin of Rust async programming is making it multi-threaded by default. If premature optimization is the root of all evil, this is the mother of all premature optimizations, and it curses all your code with the unholy `Send + 'static`, or worse yet `Send + Sync + 'static`, which just kills all the joy of actually writing Rust.*
>
> — *Maciej Hirsz*

Any time we reach for an `Arc` or a `Mutex` it's good idea to stop for a moment and think about the future implications of that decision.

The choice to use Arc or Mutex might be indicative of a design that hasn't fully embraced the ownership and borrowing principles that Rust emphasizes. It's worth reconsidering if the shared state is genuinely necessary or if there's an alternative design that could minimize or eliminate the need for shared mutable state.

The problem, of course, is that Tokio imposes this design on you. It's not your choice to make.

Beyond the complexities of architecting async code atop these synchronization mechanisms, they carry a performance cost: Locking means runtime overhead and additional memory usage; in embedded environments, these mechanisms are often not available at all.

Multi-threaded-by-default runtimes cause accidental complexity completely unrelated to the task of writing async code.

Ideally, we'd lean on an explicit `spawn::async` instead of `spawn::blocking`.

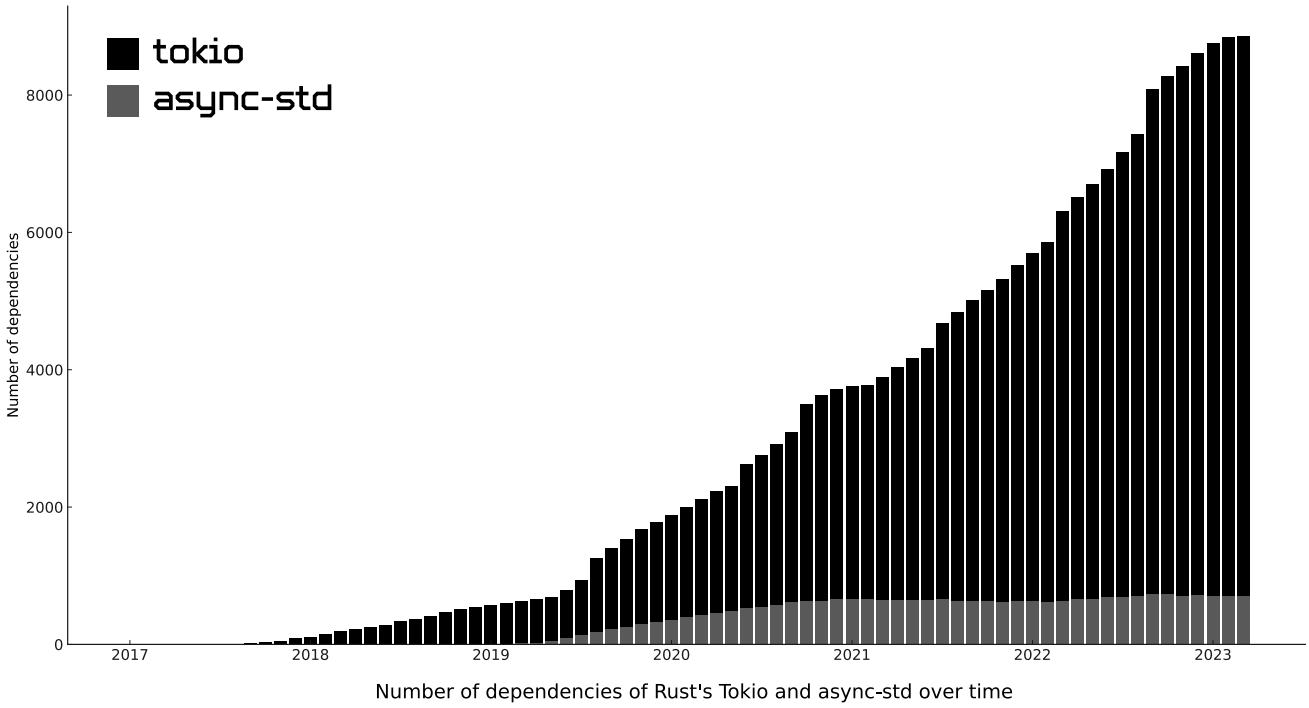
Futures should be designed for brief, scoped lifespans rather than the 'static lifetime.

Maciej suggested to use a local async runtime which is single-threaded by default and does **not** require types to be `Send` and `'static`.

I fully agree.

However, I have little hope that the Rust community will change course at this point. Tokio's roots run deep within the ecosystem and it feels like for better or worse we're stuck with it.

In the realms of networking and web operations, it's likely that one of your dependencies integrates Tokio, effectively nudging you towards its adoption. At the time of writing, Tokio is used at runtime in 20,768 crates (of which 5,245 depend on it optionally).



In spite of all this, we should not stop innovating in the async space!

Other Runtimes

Going beyond Tokio, several other runtimes deserve more attention:

- smol: A small async runtime, which is easy to understand. The entire executor is around 1000 lines of code with other parts of the ecosystem being similarly small.
- embassy: An async runtime for embedded systems.
- glommio: An async runtime for I/O-bound workloads, built on top of io_uring and using a thread-per-core model.

These runtimes are important, as they explore alternative paths or open up new use cases for async Rust. Drawing a parallel with, Rust's error handling story, the hope is that competing designs will lead to a more robust foundation overall. Especially, iterating on smaller runtimes that are less invasive and single-threaded by default can help improve Rust's async story.

Async vs Threads

Regardless of runtime choice, we end up doing part of the kernel's job in user space.

If you allow me a play on Greenspun's tenth rule:

> *Any sufficiently advanced async Rust program contains an ad hoc, informally-specified, potentially bug-ridden implementation of half of an operating system's scheduler.*

Modern operating systems come with highly optimized schedulers that are excellent at multitasking and support async I/O through io_uring and splice. We should make better use of these capabilities.

Let's finally address the elephant in the room: Threads, with their familiarity, present a path to make synchronous code faster with minimal adjustments.

For example, take our sync code to read a file from above and put it into a function:

```
fn read_contents<T: AsRef<Path>>(file: T) -> Result<String, Box<dyn Error>> {
    let mut file = File::open(file)?;
    let mut contents = String::new();
```

```
    file.read_to_string(&mut contents)?;
    return Ok(contents);
}
```

We can call this function inside the new scoped threads:

```
use std::error::Error;
use std::fs::File;
use std::io::Read;
use std::path::Path;
use std::{thread, time};

fn main() {
    thread::scope(|scope| {
        // worker thread 1
        scope.spawn(|| {
            let contents = read_contents("foo.txt");
            // do something with contents
        });

        // worker thread 2
        scope.spawn(|| {
            let contents = read_contents("bar.txt");
            // ...
        });

        // worker thread 3
        scope.spawn(|| {
            let contents = read_contents("baz.txt");
            // ...
        });
    });

    // No join; threads get joined
    // automatically once the scope ends
}
```

(Link to playground)

That code looks almost identical to the single-threaded version! Notably, there are no `.await` calls.

Were `read_contents` part of a public API, it could be used by both async and sync callers, eliminating the need for an asynchronous runtime.

Async Rust might be more memory-efficient than threads, at the cost of

complexity and worse ergonomics. As an example, if the function were *async* and you called it *outside* of a runtime, it would compile, but not run. Futures do nothing unless being polled. This is a common footgun for newcomers.

```
async fn read_contents<T: AsRef<Path>>(file: T) -> Result<String, Box<dyn Error>> {
    // ...
}

#[tokio::main]
async fn main() {
    // This will print a warning, but compile and do nothing at runtime
    read_contents("foo.txt");
}
```

(Link to playground)

In a recent benchmark, traditional threading outperformed the async approach in scenarios with a limited number of threads. This underscores the core premise that, in real-world applications, the performance difference between the two approaches is often negligible, if not slightly favoring threads. Thus, it's crucial not to gravitate towards async Rust driven solely by anticipated performance gains.

Thread-based frameworks, like the now-inactive iron, showcased the capability to effortlessly handle tens of thousands of requests per second. This is further complemented by the fact modern Linux systems can manage tens of thousands of threads.

Turns out, computers are pretty good at doing multiple things at once!

As an important caveat, threads are not available or feasible in all environments, such as embedded systems. My context for this article is primarily conventional server-side applications that run on top of platforms like Linux or Windows.

I would like to add that threaded code in Rust undergoes the same stringent safety checks as the rest of your Rust code: It is protected from data races, null dereferences, and dangling references, ensuring a level of thread safety that prevents many common pitfalls found in concurrent programming, Since there is no garbage collector, there never will be any stop-the-world pause to reclaim memory. Traditional arguments against threads simply don't apply to

Rust — fearless concurrency is your friend!

If you find yourself needing to share state between threads, consider using a channel:

```
use std::error::Error;
use std::path::Path;
use std::sync::mpsc;
use std::thread;

// Our error type needs to be `Send` to be used in a channel
// That's the only change we need to make to our original code
fn read_contents<T: AsRef<Path>>(file: T) -> Result<String, Box<dyn Error + Send>> {
    todo!()
}

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::scope(|scope| {
        scope.spawn(|| {
            let contents = read_contents("foo.txt");
            tx.send(contents).unwrap();
        });
        scope.spawn(|| {
            let contents = read_contents("bar.txt");
            tx.send(contents).unwrap();
        });
        scope.spawn(|| {
            let contents = read_contents("baz.txt");
            tx.send(contents).unwrap();
        });
    });

    // Receive messages from the channel
    for received in rx {
        println!("Got: {:?}", received);
    }
}
```

Summary

Use Async Rust Sparingly

My original intention was to advise newcomers to sidestep async Rust for now, giving the ecosystem time to mature. However, I since acknowledged that this is not feasible, given that a lot of libraries are async-first and new users will encounter async Rust one way or another.

Instead, I would recommend to use async Rust only when you really need it. Learn how to write good synchronous Rust first and then, if necessary, transition to async Rust. Learn to walk before you run.

If you have to use async Rust, stick to Tokio and well-established libraries like reqwest and sqlx.

While it may seem surprising given the context of this article, we can't overlook Tokio's stronghold within the ecosystem. A vast majority of libraries are tailored specifically for it. Navigating compatibility crates can pose challenges, and sidestepping Tokio doesn't guarantee your dependencies won't bring it in. I'm hoping for a future shift towards leaner runtimes, but for now, Tokio stands out as the pragmatic choice for real-world implementations.

However, it's valuable to know that there are alternatives to Tokio and that they are worth exploring.

Consider The Alternatives

At its core, Rust and its standard library offer just the absolute essentials for `async/await`. The bulk of the work is done in crates developed by the Rust community. We should make more use of the ability to iterate on async Rust and experiment with different designs before we settle on a final solution.

In binary crates, think twice if you really need to use async. It's probably easier to just spawn a thread and get away with blocking I/O. In case you have a CPU-bound workload, you can use rayon to parallelize your code.

> *If you don't need async for performance reasons, threads can often be the simpler alternative. — the Async Book*

Isolate Async Code

If async is truly indispensable, consider isolating your async code from the

rest of your application.

Keep your domain logic synchronous and only use async for I/O and external services. Following these guidelines will make your code more composable and accessible. On top of that, the error messages of sync Rust are much easier to reason about than those of async Rust.

In your public library code, avoid async-only interfaces to make downstream integration easier.

Keep It Simple

Async Rust feels like a different dialect, significantly more brittle than the rest of the language.

The default mode for writing Rust should be *synchronous*. Freely after Stroustup:
Inside Rust, there is a smaller, simpler language that is waiting to get out. It is this language that most Rust code should be written in.

Revision notes: In an earlier version of this article, I discussed async web frameworks. However, to maintain focus, I've opted to address web frameworks in a dedicated follow-up article. Furthermore, I've added some clarifications regarding the performance characteristics of async Rust after a discussion on Hacker News.

Published: 2023-09-13
Author: Matthias Endler
Editor: Simon Brüggen