Floresta: Move Async-std to Tokio

Concepts Preview

The proposal is about changing the async runtime from `async-std` to `tokio` in the Floresta project, but more how they work on Floresta. The change is proposed by the lead developer and maintainer of the project, Davidson Souza. The main goal is to improve performance and enhance features that `async-std` at the moment does not provide. Here some information about the concepts that will be used in this proposal:

Floresta

A open source lightweight Bitcoin full node implementation written in Rust.

Rust

Rust is a programming language focused on memory safety and performance. Its `Async` features was developed to be in the standard library but the `Runtime` was outsourced to the community.

Rust Crates

Crates are how the Rust modules and libraries are called. Every piece of code that has its own Cargo.toml can be called a crate.

Async and Runtime

Async is a programming paradigm that allows the execution of tasks concurrently. In Rust, to use Async, a `Runtime` is needed to manage the tasks.

Tokio and Async-std

`Tokio` and `Async-std` are two of the most used `Runtimes` in Rust. They provide the necessary tools to manage async tasks. But `Async-std` is not as performant as `Tokio` and can be considered a bit of deprecated.

If remains any doubt about the concepts, check the Source of Information section.

First Phase of Proposal

As proposed by Davidson Souza, lead developer and maintainer of `Floresta`, the project is eager to offer `tokio` as the primary async runtime. Currently, Floresta uses `async-std`. Potential advantages are performance improvements and enhanced features from `tokio` that `async-std` cannot provide.

The “crates” that will be affected by this proposal:

`florestad`

located at `/florestad`.

The Floresta node itself uses `async` operations to resolve shutdown calls, communicate with the Electrum server and I/O functions.

- Notable used `async-std` features:
  - `sync::{Rwlock, Block_on}`

`floresta-electrum`

located at `/crates/floresta-electrum`.

An Electrum server adapted to floresta node exposing a proper API to communicate with Electrum wallets. Async features are used to provide Tcp connections, message based channels between tasks and I/O functions.

- Notable used `async-std` features:
  - `channel::{unbounded, Receiver, Sender}`
  - `io::{BufReader}`
  - `net::{TcpListener, TcpStream, ToSocketAddrs}`
  - `sync::{Rwlock}`
  - `task::{spawn}`

`floresta-wire`

Located at `/crates/floresta-wire`.

Api to find and discover new blocks, it has have p2p protocol and utreexod's JSON-rpc implemented. Async feature is heavily present on p2p communication.

- Notable used Async-std features:
  - sync::{RwLock}
  - Channel::{bounded, receiver, sender, sendError}
  - net::{TcpStream}

Regarding the use of asynchronous, the future trait is used in the mentioned crates to manage an asynchronous approach to some tasks, even not being inherited directly from the Async-std Its outsourcing to Tokio can be evaluated since the discovery of a different approach to use Tokio that can reward more performance and trust in its execution.

The Challenge

To fit Tokio in Floresta some parameters have to be evaluated before the change execution.

1. The performance can't be worse than the alternative that is currently being used. (Async-std)
2. The code complexity is not supposed to increase. Even if the Tokio implementation needs rewriting, that is probably what will be, the rewriting needs to follow a similar way to deal with tasks that the actual code already has.
3. The tests that are related to the affected crates by the runtime change need to stay intact and untouched, showing that all functionalities are all working fine and as expected.

Steps Planning

The next steps covers the main purpose of the proposal, extras and possibilities that will come with the dependency change will be described and explored at "Second Phase of Proposal".

1. **Floresta-Wire**, **Floresta-Electrum** and **Florestad** Rust test battery for internal functions and Python test battery for mocking external cases and performance cover, focused on async functionalities. Deeper understanding of the project. **(1 - 2 weeks).**
2. **Floresta-Wire** async functionalities dependencies from Async-std to Tokio. **(1 - 2 weeks).**
3. **Floresta-Electrum** async functionalities dependencies from Async-std to Tokio. **(8 - 12 days).**
4. **Florestad** async functionalities dependencies from Async-std to Tokio. **(8 - 12 days).**

The estimated work time may vary depending on problems encountered during the execution of the proposal even if, in this document, defined for organized work, properly documented, Error handling and covering possible errors. Considering that the start of the work in the Floresta's project would begin at 15 May 2024 and is safely expected by the middle/end of july 2024.

Second Phase of Proposal

After the successful integration of Tokio, the good practices in code versioning (see code versioning planning) can introduce us to an opportunity to integrate a good feature to Floresta portability, Agnostic Runtime.

Agnostic runtime

Agnostic, outside the context of the religious meaning, can infer something that is “unattached” of another thing. In this project context we can “unattach” the Async runtime that rust outsourced to the community in a trade for just a little workaround and some “redesign” in how the async functions works for floresta.

This is a rust representation of how things could work (not final work):

```
use std::{future::Future, process::Output};
use anyhow::Result;
use std::marker::Send;
use async_std::task::{self as std_task};
use tokio::task::{spawn as tokio_spawn};


trait Asyncfunctions {
    async fn task<F, T>(&self,t: F) -> T where F: Future<Output = T> + Send
+ 'static,
    T: Send + 'static;
}
struct StdAsync;

struct TokioRuntime;

impl Asyncfunctions for TokioRuntime{
    async fn task<F, T>(&self,t: F) -> T where F: Future<Output = T> + Send +
'static,
    T: Send + 'static{
        tokio_spawn(t).await.unwrap()
    }
}
impl Asyncfunctions for StdAsync{
    async fn task<F, T>(&self,t: F) -> T where F: Future<Output = T> + Send
+ 'static,
    T: Send + 'static{
        std_task::spawn(t).await
    }
}

async fn agnostic_function<F: Asyncfunctions> (runtime: F) -> Result<()>
    { let task = runtime.task( async {
            let mut i = 0;
            for j in 0..1_000_000_000 {
                i += 1;
            }
            println!("one billion is reached. i:{}", i );
    });
    task.await;
    Ok(())
}
```

```
#[tokio::main]
async fn main() {

    println!("print one billion using Async-std funtions:");
    let _ =  agnostic_function(StdAsync);
    println!("print one billion using tokio functions:");
    let _ =  agnostic_function(TokioRuntime).await;

}
```

See that in `agnostic_function()` we can use Dependency injection, a technique that allow the use of dependencies as parameters and calling functionalities from that parameters, to make functions that need async use just the library that we want to inherit the async functions, in this case, `Async-std` and `Tokio` are used. both functions work as expected using each other runtime just printing:

```
print one billion using Async-std funtions:
one billion is reached. i:1000000000 print one
billion using tokio functions: one billion is
reached. i:1000000000
```

In the example, using Tokio runtime.

Is good to note the Rust feature that is being use in the design of code, including a "`wrapper`" function to async functionalities as `traits` and them making structs for the Async libraries that can be called.

Note the use of aliases when inheriting async functions from their respective libraries in this way, the compiler knows exactly what to call when building the binary.

```
use async_std::task::{self as std_task};
use tokio::task::{spawn as tokio_spawn};
```

Even if the code in question calls both libraries, in a oficial implementation is better to be precise and include this use of aliases

The runtime needs to be declared in the code, this can be achieved with cargo features and Rust macros to change the desired runtime in the compile time. Example:

```
cargo build --features "async-std-runtime"
```

or

```
cargo build --features "tokio-runtime"
```

and using types in Rust

```
#[cfg(feature = "async-std-runtime")]
type Runtime = StdAsync;

#[cfg(feature = "tokio-runtime")]
type Runtime = TokioRuntime;
```

The idea about "Runtime Agnostic" was mentioned by Davidson Souza at the Summer of Bitcoin

> "A stretch goal would be making it runtime agnostic, rather than tied to tokio alone."
>
> — Davidson at Floresta Project Ideas

The code design can be better discussed, but in first idea, the code could rely in modulating only the used async functions by the **libFloresta** and **florestad** make better use and reuse of the code.

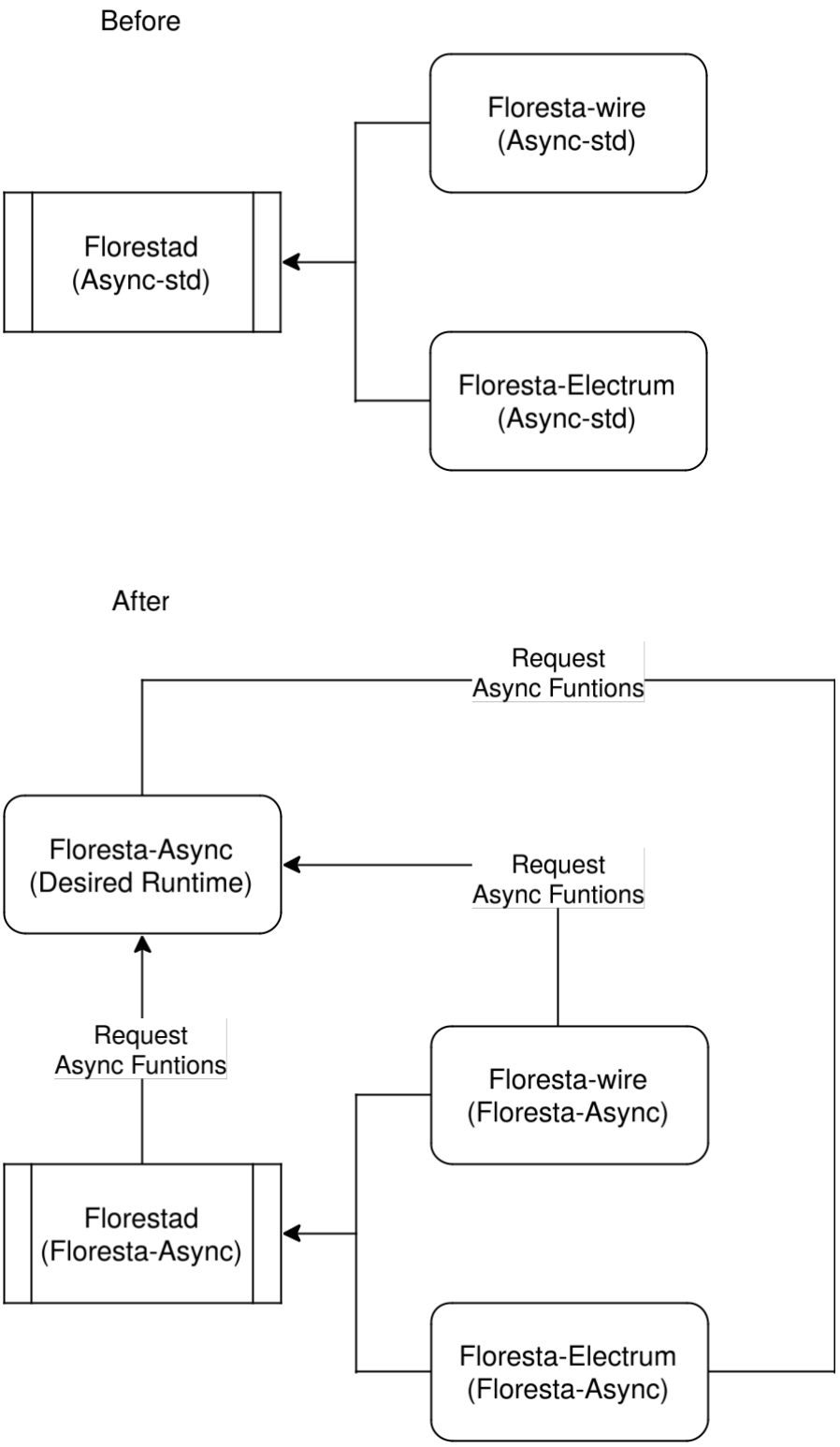

Figure 1: Before and After the “Agnostic Runtime” implementation in Floresta.

Using modularization to a point that the async functions used by Floresta are all together in a single module can expand and simplify the possibilities of features, one of them being the runtime change without taking down the whole Node and can simplify the code in the other modules by just calling the async functions from the module.

With this technique of “Agnostic Runtime” the Floresta node can fit or can easily be modified to fit in any device if the “Main runtime” can be a problem. For more low-end devices and environments

with scarce computing resources, the use of `smol-rs` can be a good fit and will need less work to implement it in the project than if the project was using only one runtime.

Depending on Mentor's will or ideas, the Agnostic Runtime code design and technique can be changed before the implementation.

Time expectation of implementation: **(3 + weeks).**

Code versioning planning

Code Versioning can help to decrease the implementation time and prevent errors since a good definition of the work objectives at the beginning of the work.

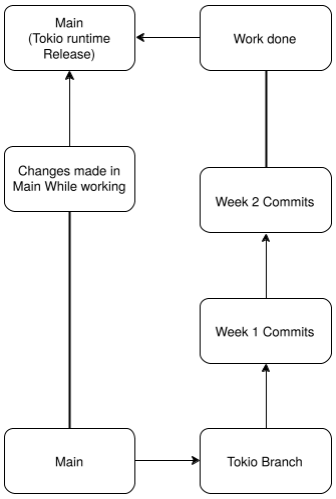

Figure 3: Versioning representation without
Agnostic runtime.

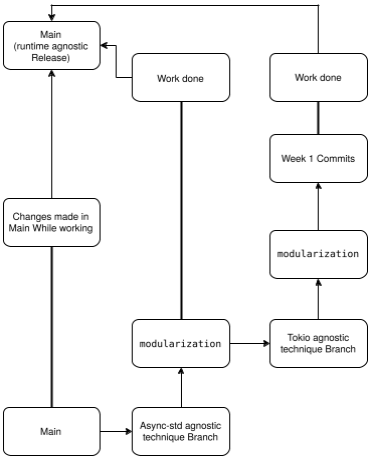

Figure 4: Versioning representation with
Agnostic runtime.

As represented in the images above, the agnostic modularization can start being implemented since the beginning of the work in the project.

A Promising Feature for Bitcoin and Rust Community

After the success of the proposal work, we'll stand with a more portable and flexible bitcoin project that has the great potential to be the `"Must Use"` bitcoin node for low-end devices, even because the Rust programming language is promising in this environments, and for it enthusiast that love to fingering with the code. The Floresta project attack in the bitcoin most requested feature: The `Adoption`, when evolving the project to be a fit for any device. The Rust community, in general, can make a good use of a future `Floresta-Async`, a possible runtime agnostic rust crate made for Floresta project since the highly mentioned technique is quite reproducible for other projects. To follow the lightweight ideal of Floresta the implementation of `smol-rs`, a lightweight async Rust runtime with permissive license, will bring the project closer to be perfect and a icing on the cake to work with after the conclusion of the proposal.

The Writer

The writer of this proposal is João Leal, a Brazilian Science Computer student in his first semester that tries to run out of obvious but never from the simple. The writer became a Rust adopter to learn things using modern and efficient tools. Programming modernity can't be explained without talking about the Bitcoin solution to the global society money problem, and maybe ower best piece of code for ourselves as humans, the poor, the rich, the minority and the majority, they are all the same for Bitcoin and Bitcoin is the same for them. If anything on this proposal sounds good for you, and you want more of it, you can talk with the writer:

- Full name: João Gabriel Leal de Andrade Barbosa
- E-mail: jgleal@protonmail.com
- Github: João Leal's Github.
- Discord: jleall
- Nationality:Brazilian
- Timezone: São Paulo - SP (GMT-3)

Source of Information:

The Rust Programming Language

Runtime at Wikipedia

Async-std

Tokio

Dependency injection

smol-rs at Github

Runtime agnostic async crates by Zeeshan Ali.

Summer of Bitcoin website proposal

#144 [SoB]: Move Async-std to Tokio