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. The project is a fully-validating Bitcoin node in Rust. 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` cannot provide. Here some bullet points about the concepts that will be used in the proposal:

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.

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, go to the Good to read section.

Focus of Proposal

As proposed by Davidson Souza, lead developer and maintainer of `Floresta`, a fully-validating Bitcoin node in Rust, the project is eager to offer `tokio` as the primary async runtime. Currently, the `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 wavkvllets. Async presence is 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 that have p2p protocol and utreexod's JSON-rpc implemented. Async feature is heavily used 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 Problematic

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 After Party.

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 midlle/end of july 2024.

After Party

After the sucessfull integration of Tokio, the good pratices 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 outsorced to the

community in a trade for just a little boilerplate and some “redesign” in how the async functions works for floresta.

This is a rust example of how things could 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 the the Dependency injection technique to make async funtions use the library that we want to henrerit the funcions, in this case, `Async-std` and `Tokio` are used. both functions work as expected using eachother 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 exemple, using Tokio runtime.

Since the runtime needs to be declared in the code and 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's Project Ideas

The code design can be better discussed, but in first idea, the code could rely in modulating the async functions for each crate present in the libFloresta to make better use and reuse of the code.

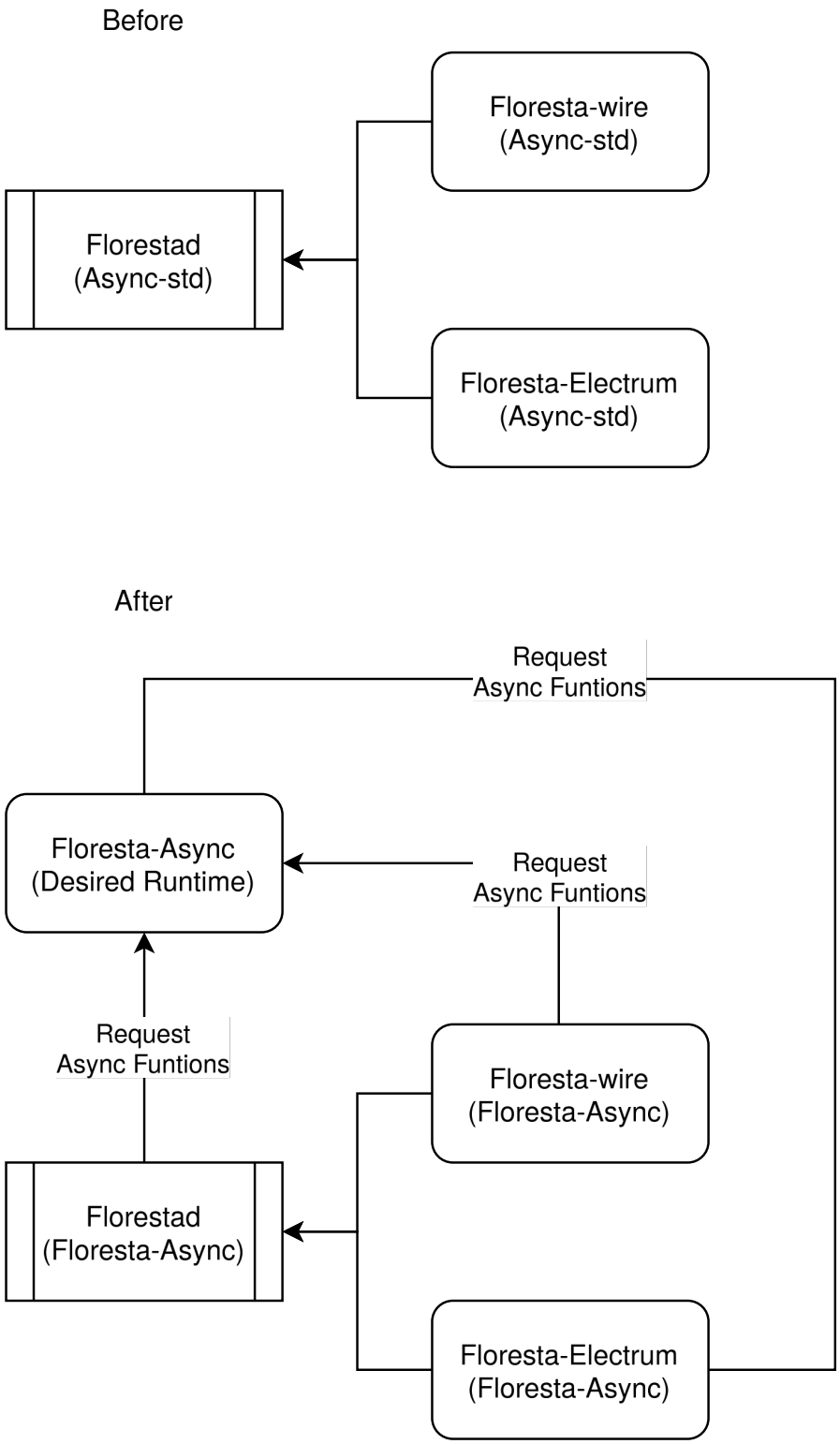

Figure 1: Before and After the "Agnostic Runtime" implementation in Floresta.

With this technique of "Agnostic Runtime" the Floresta node can fit or can easily modified to fit in any device if the "Main runtime" can be a problem. For more "portable" devices, 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 `Tokio` or `Async-std`.

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

Time expectation to implementation: **(1 - 2 weeks).**

Code versioning planning

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

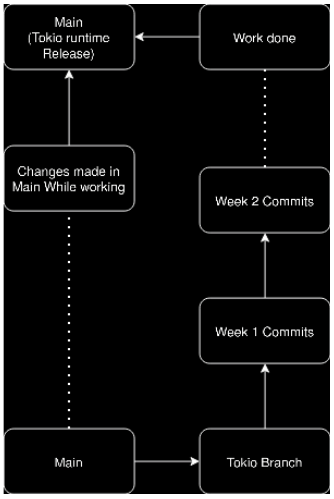

Figure 3: Versioning representation without Agnostic runtime.

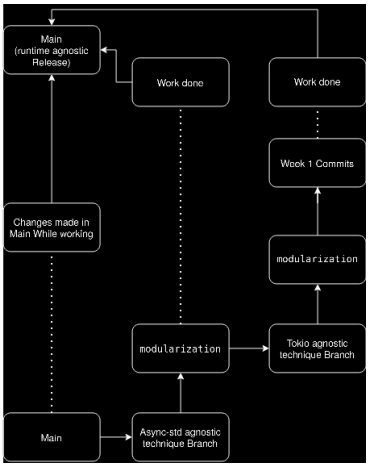

Figure 4: Versioning representation with Agnostic runtime.

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

Good to read (fonts):

The Rust Programming Language

Runtime at Wikipedia

Async-std

Tokio

smol-rs at Github

Runtime agsnostic async crates by Zeeshan Ali.

Summer of Bitcoin website proposal

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