Focus of Proposal

As proposed by Davidson Souza, Owner of Floresta's Repository, the project is eager to change from the Rust Async-std to Tokio due to better maintenance, performance and features that Tokio have nowadays.

The "crates" that will be affected by this proposal:

-

  ### florestad

Located at `/florestad`.

The Floresta node itself uses Async 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`.

A Electrum server adapted to floresta node exposing a proper API to communicate with Electrum wallets. 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.

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:

```rust
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 Stdeisync;

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 Stdeisync{
    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(Stdeisync);
    println!("print one billion using tokio functions:");
    let _ =  agnostic_function(TokioRuntime).await;

}
```

See that in `agnostic_function()` we can use the

Code versioning planning

Good to read (fonts):

Async-std

Tokio

smol-rs

Runtime agsnostic async crates by Zeeshan Ali.

Summer of Bitcoin website proposal

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