Grin networking

Past, present, futures?

@hashmap

QTUM

SPARK POOL

Outline

- Blocking and non-blocking IO
- Thread execution models
- What Rust offers
- Grin messaging protocol
- Evolution of Grin networking
- Async Rust strikes back
- Future(s)

Blocking and non-blocking IO

Thread

- Instruction 1
- Instruction 2
- read
- … waiting …
- … waiting …
- Instruction 4

Blocking IO

- Good old read()
- Blocks a thread
- Simple sequential logic
- What should we do while we are waiting?

Non-blocking IO

- *Call Me Maybe* (Carly Rae Jepsen)
- Doesn’t block a thread
- Notifies when the operation is completed (if supported)
- Not that simple

Thread execution models

N:N execution model

- N connections
- N threads
- Simple
- Not very scalable
- Supported out of the box in most of the languages

M:N execution model

- N connections
- M threads
- M < N or M << N
- Complex runtime
- Few languages support it out of the box

What Rust offers

Rust approach

- No runtime
- Abstractions without hidden overhead
- What you don't use, you don't pay for. What you do use, you couldn't hand code any better [Stroustrup, 1994]
- Blocking IO
- Non-blocking IO without notification support
- N:N model
- M:N model was removed pre Rust 1.0

MIO and Tokio

- MIO implements non-blocking IO with notifications
- Tokio provides high level abstractions on top of MIO
- M:N thread model
- Future
- Rust Future is a bit different (poll model)
- Zero cost abstraction
- Plus runtime

Futures are fun to work with

```
error[E0599]: no method named `map_err` found for type `futures::AndThen<tokio_t ungstenite::AcceptAsync<tokio_core::net::TcpStream, tungstenite::handshake::serv
er::NoCallback>, futures::AndThen<std::boxed::Box<futures::Future<Error=std::io: :Error, Item=std::vec::Vec<u8>>>, std::result::Result<(), std::io::Error>, [clos
ure@src/main.rs:17:44: 20:8 address:_]>, [closure@src/main.rs:16:54: 22:6 handle _clone:_, address:_]>` in the current scope
  --> src/main.rs:23:6
   |
23 |     .map_err(|e| std::io::Error::new(std::io::ErrorKind::Other, e))
   |      ^^^^^^^
   |
   = note: the method `map_err` exists but the following trait bounds were not satisfied:
           `futures::AndThen<tokio_tungstenite::AcceptAsync<tokio_core::net::Tcp Stream, tungstenite::handshake::server::NoCallback>, futures::AndThen<std::boxed
::Box<futures::Future<Error=std::io::Error, Item=std::vec::Vec<u8>>>, std::resul t::Result<(), std::io::Error>, [closure@src/main.rs:17:44: 20:8 address:_]>, [cl
osure@src/main.rs:16:54: 22:6 handle_clone:_, address:_]> : futures::Future` `&mut futures::AndThen<tokio_tungstenite::AcceptAsync<tokio_core::net
::TcpStream, tungstenite::handshake::server::NoCallback>, futures::AndThen<std:: boxed::Box<futures::Future<Error=std::io::Error, Item=std::vec::Vec<u8>>>, std::
result::Result<(), std::io::Error>, [closure@src/main.rs:17:44: 20:8 address:_]> , [closure@src/main.rs:16:54: 22:6 handle_clone:_, address:_]> : futures::Stream
`
           `&mut futures::AndThen<tokio_tungstenite::AcceptAsync<tokio_core::net ::TcpStream, tungstenite::handshake::server::NoCallback>, futures::AndThen<std::
boxed::Box<futures::Future<Error=std::io::Error, Item=std::vec::Vec<u8>>>, std:: result::Result<(), std::io::Error>, [closure@src/main.rs:17:44: 20:8 address:_]>
, [closure@src/main.rs:16:54: 22:6 handle_clone:_, address:_]> : futures::Future
```

Grin messaging protocol

Types of protocols:

- Request - Response (HTTP) - ordered
- Messaging - no particular order

Grin tried both approaches (AFAIK)

Grin messaging protocol

- 2 almost independent connections
- Used to be request-response (testnet1)
- Handshake is request-response (messages **Hand** and **Shake**)
- **GetXXX** is an async request
- **XXX** is an async response or just a message
- **GetHeader** <-> **Header** or <- **Header**
- **TxHashSetArchive** is checked if it was requested

```
enum_from_primitive! {
    #[derive(Debug, Clone, Copy, PartialEq)]
    pub enum Type {
        Error = 0,
        Hand = 1,
        Shake = 2,
        Ping = 3,
        Pong = 4,
        GetPeerAddrs = 5,
        PeerAddrs = 6,
        GetHeaders = 7,
        Header = 8,
        Headers = 9,
        GetBlock = 10,
        Block = 11,
        GetCompactBlock = 12,
        CompactBlock = 13,
        StemTransaction = 14,
        Transaction = 15,
        TxHashSetRequest = 16,
        TxHashSetArchive = 17,
        BanReason = 18,
        GetTransaction = 19,
        TransactionKernel = 20,
        KernelDataRequest = 21,
        KernelDataResponse = 22,
    }
}
```

p2p/src/msg.rs

Evolution of Grin networking

What we tried

1.	Blocking IO, M:N	doesn’t make sense
2.	Non-blocking IO, M:N (Tokio)	testnet1
3.	Non-blocking IO, N:N, tesntet(2,3,4)	grin 1.x, 2.0
4.	Blocking IO, N:N (actually 2N:N)	grin 2.1+

Why not Non-blocking IO, M:N

Rewrite peer-to-peer logic to replace tokio with simple threading #664

Merged ignopeverell merged 8 commits into `mimblewimble:master` from `unknown repository` on Feb 2, 2018

Conversation 13 Commits 8 Checks 0 Files changed 19

ignopeverell commented on Jan 30, 2018 Member

Tokio and futures have proven very difficult to work with in our specific case (peer-to-peer interactions), making the code far more unwieldy and complex than it should. The result is hard to read, hard for any new contributor to get into and hard to change when new logic needs to be added. All of this for very little benefit as a peer in a network should never have to handle extremely high traffic levels (like a chat or HTTP server would, for example).

This PR rewrites most of the `p2p` crate as well as the seeding logic with simple threading. Each peer have a thread and the TCP server has its own as well. Resulting code is a lot easier to understand.

Only part left is fixing the tests in the `grin` crate.

Why not Non-blocking IO, N:N

- High cpu load (no read readiness events)

  Loop:

    Read from socket

    Write to socket (if needed)

    Check stop channel

    Short sleep
- read_exact and write_exact have to be reimplemented and also

  CPU-intensive

Why not Blocking IO, N:N

- 2 x peer number threads

  Loop1:

    Read from socket (block for 1 sec or more)

    Check stop channel

  Loop2:

    Check outgoing queue

    Write to socket (blocking syscall)

- Still some low level IO which is error prone

Future(s)

But we run out of options

- Oh, wait, **.await**
- Rust 1.39 brings **async/.await** functionality
- High level async abstractions
- Almost like sequential logic

Again Non-blocking IO, M:N **?**

Pros:
Most scalable
High level enough
All cool kids do it
Lots of work

Cons:
Ecosystem is not there yet
Less control
Errors may be puzzling
Lots of work