libp2p

Modular Peer-to-Peer Networking Stack for Rust

Erhan Tezcan

Blockchain Dev. @ Dria

4.5.2025

Introduction

whoami

- Building a peer-to-peer network (in Rust) at **Dria**[1].
- Interested in EVM & Solidity, and all kinds of languages!
- Loves open-source, contributed to a few zero-knowledge cryptography & LLM libraries.
- Worked on GPU programming (CUDA) at Koç University.
- Using Rust for ~2 years, can't go back to **anything else**.
- See more at **erhant.me**[2] or `github.com/erhant`.
- This presentation is written in **Typst**, a typesetting language built in Rust!

---

[1] formerly known as **FirstBatch**

[2] note to self: rebuild with `ratzilla`

What is libp2p?

libp2p[1] is a modular peer-to-peer networking stack, driven by well-designed specifications with several implementations (Go, JavaScript, **Rust**, C).

Rust implementation of libp2p in particular is being used in notable projects like **IPFS** client, **Lighthouse** Ethereum consensus client, **Filecoin** client, **Substrate** (of Polkadot), and many more.

[1] https://github.com/libp2p/rust-libp2p

Protocol Stack

Connection

- TCP, QUIC, WebSocket
- Multiplexing (**Yamux**, **mplex**)
- Security (**Noise**)

Communication

- GossipSub
- Request-Response
- DCutR

Discovery

- Kademlia DHT
- Multicast DNS (mDNS)
- Rendezvous

What is a Peer?

A peer is a node in a distributed network that can communicate with other nodes. In `libp2p`, peers are identified by their unique `PeerIds` and `Multiaddrs`.

A peer ID is a cryptographic identifier of a peer in the network, derived from the public key of the peer. LibP2P has two types of keys:

- **Secp256k1**: A widely used curve for ECDSA, often associated with Bitcoin and Ethereum.
- **Ed25519**: A modern curve used for EdDSA, with better speed and security.

What is a Peer?

Multi-addressing is a flexible addressing scheme used in `libp2p` to represent network addresses. It allows peers to specify multiple protocols and transport layers in a single address format.

- `/ip4/3.7.24.28/tcp/1234`
- `/ip4/6.18.23.23/tcp/4001/p2p/QmYyQSo1c1Ym7orWxLYvCrM2EmxFTANf8wXmmE7DWjhx5N`

Peer Discovery

Kademlia DHT

Kademlia is a Distributed Hash Table (DHT) protocol used for peer discovery and data storage in decentralized networks. It allows peers to efficiently locate other peers and resources in the network.

Multicast DNS (mDNS)

mDNS is a zero-configuration protocol that allows devices on a local network to discover each other, just via the router.

Rendezvous

A server simply collects peer information and newly-joined peers get all that information from this server.

Communication

LibP2P has two notable communication methods:

- **GossipSub**: Nodes use "gossip protocols" to distribute messages across the network efficiently; useful for broadcasting messages to multiple peers.
- **Request-Response**: Nodes can send requests to each other and receive responses, allowing for more structured communication patterns.

It also provides a method to bypass **NAT** problems between peers.

GossipSub

GossipSub is a publish-subscribe protocol that allows nodes to efficiently disseminate messages across the network.

Uses a combination of gossiping and topic-based subscriptions to ensure that messages reach interested peers while minimizing network overhead, while keeping track of peer scores to prioritize messages from trusted peers.

Request/Response Protocol

Allow nodes to send requests to each other and receive responses (one-to-one). Useful for scenarios where a node needs to query another node for specific information or perform a remote procedure call (RPC).

NAT & DCutR

LibP2P provides several mechanisms for NAT traversal, allowing peers behind NATs to communicate with each other.

DCutR (Direct Connection Upgrade through Relay) protocol is designed to facilitate direct connections between peers behind NATs.

- **Hole punching**: DCutR uses hole punching techniques to establish direct connections between peers, even if they are behind NATs.
- **Relay servers**: In cases where direct connections cannot be established, DCutR can use relay servers to facilitate communication between peers.

Usage

Behaviour

All modules we have talked about so far belong to a “behaviour” struct.

```
use libp2p::{gossipsub, identify,
mdns};
use libp2p::swarm::NetworkBehaviour;

#[derive(NetworkBehaviour)]
struct MyBehaviour {
    pub gossipsub:
gossipsub::Behaviour,
    pub mdns: mdns::tokio::Behaviour,
    pub identify: identify::Behaviour,
    // ...
}
```

This `derive` macro automatically implements the `NetworkBehaviour` trait for our struct, allowing us to use it as a behaviour in our libp2p application.

We then provide the behaviours to a `Swarm` struct, which is responsible for managing the network connections and communication between peers.

```rust
use libp2p::swarm::Swarm;

pub struct MyClient {
    swarm: Swarm<MyBehaviour>,
    // ...
}
```

Swarm then provides us `behaviour()` and `behaviour_mut()` methods to access the behaviours, e.g. `swarm.behaviour_mut().gossipsub`.

Connection Configuration

Connection configuration is done through the `SwarmBuilder` struct. You can configure the transport, identity, and other parameters of the connection.

```
let keypair = libp2p::identity::Keypair::generate_secp256k1();
let swarm = libp2p::SwarmBuilder::with_existing_identity(keypair)
    .with_tokio()
    .with_tcp(
        tcp::Config::default(),
        noise::Config::new,
        yamux::Config::default,
    )?
    .with_behaviour(|key| Ok(ChatBehaviour::new(key).unwrap()))
    .unwrap().build();
```

Events

LibP2P handles everything behind the scenes, and all providers communicate with each other through `SwarmEvents`:

```
  match self.swarm.select_next_some().await {
    SwarmEvent::ConnectionClosed { peer_id, cause, .. } => {
      println!("Disconnected from peer: {:?}, cause: {:?}", peer_id,
cause);
    }
    SwarmEvent::Behaviour(MyBehaviourEvent::GossipSub(gossip_event))
=> {}
    SwarmEvent::Behaviour(MyBehaviourEvent::Mdns(mdns_event)) => {}
    // and so on...
  }
```

Cancellation

Using a cancellation token is a good way to handle graceful shutdowns:

```
loop {
  _ = cancellation.cancelled() => break,
  event = self.swarm.select_next_some() => {
    // ...
  }
}
```

Metrics

Suppose you want to launch an HTTP server on the side to expose metrics about your P2P client. You can use an `Arc<RwLock<MyMetrics>>` and share it with the HTTP server thread and the P2P thread.

```
tokio::select! {
    event = self.swarm.select_next_some() => {/*...*/}
    _ = metrics_interval.tick() => {
        let mut metrics = self.metrics.write().await;
        // ...
    }
}
```

Then within your HTTP server thread, you can read the metrics:

```
let metrics = metrics.read().await;
// ...
```

Caveats

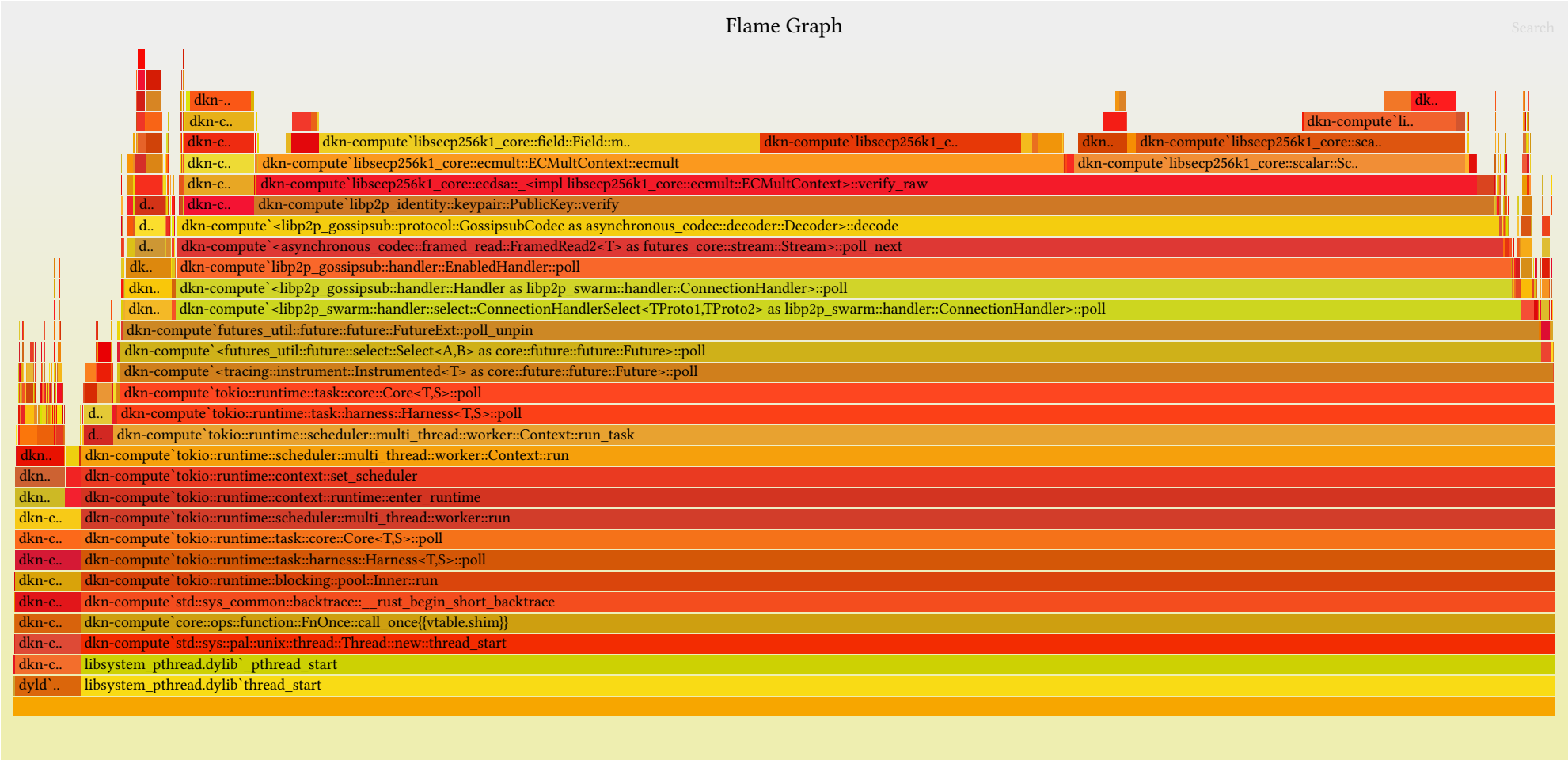
Authentication in GossipSub

GossipSub has several message authentication methods, with these enum variants:

- `Signed(Keypair)`
- `Author(PeerId)`
- `RandomAuthor`
- `Anonymous`

`Signed` is the most secure one, enabling a cryptographic guarantee that the message received originates from that peer. However...

Authentication in GossipSub



LibP2P will use a lot of file descriptors when the network is scaled up, so you may need to increase the `ulimit` on your system.

```
ulimit -n 12345
```

You will likely need to do this on the client code automatically:

```
let (soft, hard) = rlimit::Resource::NOFILE
      .get()
      .unwrap_or((DEFAULT_SOFT_LIMIT, DEFAULT_HARD_LIMIT));
let target_soft = hard / 10;
if soft < target_soft {
    if let Err(e) = rlimit::Resource::NOFILE.set(target_soft, hard) {
        // log...
    }
}
```

Memory Backpressure

Suppose you have a single-thread for handling `SwarmEvents`, and you are doing heavy work on each received `Request` event. Swarm events will keep coming, and the memory will keep growing until it runs out of memory.

To prevent this, you can use a `mpsc` channel to send the events to a worker thread for processing. This way, the main thread can continue to receive events without blocking.

```
pub struct MyClient {
    swarm: Swarm<MyBehaviour>,
    reqres_tx: mpsc::Sender<(PeerId, MyMessage)>,
    cmd_rx: mpsc::Receiver<MyCommand>,
}
```

Memory Backpressure

`Swarm` can't be sent between threads! We need to do some inter-thread communication if we ever need to use Swarm. We give an `mpsc` receiver to the `MyClient` struct, and from a different thread we can send "commands" to it, paired with a `oneshot` channel to get the result back.

```
let (sender, receiver) = oneshot::channel();
self.sender
      .send(DriaP2PCommand::IsConnected { peer_id, sender })
      .await?;
let is_connected = receiver.await?;
```

FFI

Why?

You love rust-libp2p, but you want to use it in a different language? You can do that with the FFI (Foreign Function Interface) bindings. Simply add the following to your Cargo.toml, and it will output shared library:

```
[lib]
crate-type = [
  "cdylib", # allows C/C++ to use this library
  "rlib",   # allows Rust to use this library
]
```

Why?

You love rust-libp2p, but you want to use it in a different language? You can do that with the FFI (Foreign Function Interface) bindings. Simply add the following to your Cargo.toml, and it will output shared library:

```
[lib]
crate-type = [
  "cdylib", # allows C/C++ to use this library
  "rlib",   # allows Rust to use this library
]
```

We will use the following types for our pointer type that lives in C.

```
typedef struct libp2p_chat libp2p_chat_t;
typedef struct libp2p_chat_handle libp2p_chat_handle_t;
```

Allocations


```rust
// extern libp2p_chat_t *libp2p_chat_new(void);
#[unsafe(no_mangle)]
pub extern "C" fn libp2p_chat_new() -> *mut ChatClient {
    let client =
        ChatClient::new(CancellationToken::new()).unwrap();
    Box::into_raw(Box::new(client))
}
// extern void libp2p_chat_free(libp2p_chat_t *ptr);
#[unsafe(no_mangle)]
pub extern "C" fn libp2p_chat_free(chat_ptr: *mut ChatClient) {
    if chat_ptr.is_null() { return; }
    unsafe {
        drop(Box::from_raw(chat_ptr));
    }
}
```

Starting

```
// extern libp2p_chat_handle_t* libp2p_chat_start(libp2p_chat_t* ptr, unsigned short
port);
#[unsafe(no_mangle)]
pub extern "C" fn libp2p_chat_start(client_ptr: *mut ChatClient, port: u16) -> *mut
JoinHandle<()> {
    let client = unsafe {
        assert!(!client_ptr.is_null());
        &mut *client_ptr
    };

    let rt = tokio::runtime::Builder::new_multi_thread()
        .enable_all().build().unwrap();
    let handle = std::thread::spawn(move || {
        rt.block_on(async { client.run(port).await.expect("could not run the
client") });
    });
    Box::into_raw(Box::new(handle))
}
```

Stopping


```rust
// extern void libp2p_chat_stop(libp2p_chat_t *ptr, libp2p_chat_handle_t *handle_ptr);
#[unsafe(no_mangle)]
pub extern "C" fn libp2p_chat_stop(
    client_ptr: *mut ChatClient,
    handle_ptr: *mut JoinHandle<()>,
) -> i32 {
    let client = unsafe {
        assert!(!client_ptr.is_null());
        &mut *client_ptr
    };
    let handle = unsafe {
        assert!(!handle_ptr.is_null());
        Box::from_raw(handle_ptr)
    };
    client.cancel();
    match handle.join() {
        Ok(_) => 0,
        Err(err) => -1
    }
}
```

Receive


```
// extern int libp2p_chat_receive(libp2p_chat_t *ptr, void *buf, size_t buf_size);
#[unsafe(no_mangle)]
pub fn libp2p_chat_receive(client_ptr: *mut ChatClient, buf: *const u8, buf_size: usize)
-> i32 {
    let client = unsafe {
        assert!(!client_ptr.is_null());
        &mut *client_ptr
    };
    if let Some((_, msg)) = client.received.pop_front() {
        let msg_len: usize = msg.len();
        if msg_len == 0 { 0 }
        else if buf_size < msg_len { -1 }
        else {
            unsafe {
                std::ptr::copy_nonoverlapping(msg.as_ptr(), buf as *mut u8, msg_len);
            }
            msg_len as i32
        }
    } else { 0 }
}
```

Demo?

Thank You!