Configuration

Overview

libp2p is a modular networking stack. It's designed to be able to suit a variety of project needs. The configuration of libp2p is a key part of its structure. It enables you to bring exactly what you need, and

only what you need. This document is a guide on how to configure libp2p for your particular project. Check out the Configuration examples section if you're just looking to leverage an existing configuration.

Regardless of how you configure libp2p, the top level API will always remain the same. **Note**: if some modules are not configured, like Content Routing, using those methods will throw errors.

To get a high-level overview of the js-libp2p architecture, please read the Architecture document.

Modules

`js-libp2p` acts as the composer for this modular p2p networking stack using libp2p compatible modules as its subsystems. For getting an instance of `js-libp2p` compliant with all types of networking requirements, it is possible to specify the following subsystems:

- Transports
- Multiplexers
- Connection encryption mechanisms
- Peer discovery protocols
- Content routing protocols
- Peer routing protocols
- DHT implementation
- Pubsub router

The libp2p ecosystem contains at least one module for each of these subsystems. The user should install and import the modules that are relevant for their requirements. Moreover, thanks to the existing interfaces it is easy to create a libp2p compatible module and use it.

After selecting the modules to use, it is also possible to configure each one according to your needs.

Bear in mind that a **transport** and **connection encryption** module are **required**, while all the other subsystems are optional.

Transport

> In a p2p system, we need to interact with other peers in the network. Transports are used to establish connections between peers. The libp2p transports to use should be decided according to the environment where your node will live, as well as other requirements that you might have.

Some available transports are:

- @libp2p/tcp (not available in browsers)
- @libp2p/webrtc
- @libp2p/websockets
- @libp2p/webtransport

If none of the available transports fulfils your needs, you can create a libp2p compatible transport. A libp2p transport just needs to be compliant with the Transport Interface.

If you want to know more about libp2p transports, you should read the following content:

- https://docs.libp2p.io/concepts/transports/overview
- https://github.com/libp2p/specs/tree/master/connections

Stream Multiplexing

> Libp2p peers will need to communicate with each other through several protocols during their life. Stream multiplexing allows multiple independent logical streams to share a common underlying transport medium, instead of creating a new connection with the same peer per needed protocol.

Some available stream multiplexers are:

- @chainsafe/libp2p-yamux

Some transports such as WebRTC and WebTransport come with their own built-in stream multiplexing capabilities.

If none of the available stream multiplexers fulfills your needs, you can create a libp2p compatible stream multiplexer. A libp2p multiplexer just needs to be compliant with the Stream Muxer Interface.

If you want to know more about libp2p stream multiplexing, you should read the following content:

- https://docs.libp2p.io/concepts/stream-multiplexing
- https://github.com/libp2p/specs/tree/master/connections
- https://github.com/libp2p/specs/tree/master/yamux

Muxer Selection

If you configure multiple muxers for use in your application, js-libp2p will choose the first muxer in the list. Therefore, ordering matters.

Connection Encryption

> A connection encryption mechanism must be used, in order to ensure all exchanged data between two peers is encrypted.

Some available connection encryption protocols:

- @chainsafe/libp2p-noise
- @libp2p/plaintext (Not for production use)

If none of the available connection encryption mechanisms fulfills your needs, you can create a libp2p compatible one. A libp2p connection encryption protocol just needs to be compliant with the Connection Encrypter Interface.

If you want to know more about libp2p connection encryption, you should read the following content:

- https://docs.libp2p.io/concepts/secure-comms
- https://github.com/libp2p/specs/tree/master/connections

Peer Discovery

> In a p2p network, peer discovery is critical to a functioning system.

Some available peer discovery modules are:

- @libp2p/mdns (spec)
- @libp2p/kad-dht (spec)
- @libp2p/bootstrap (typically used together with @libp2p/kad-dht)
- @chainsafe/discv5 (spec)

If none of the available peer discovery protocols fulfills your needs, you can create a libp2p compatible one. A libp2p peer discovery protocol just needs to be compliant with the Peer Discovery Interface.

Content Routing

> Content routing provides a way to find where content lives in the network. It works in two steps: 1) Peers provide (announce) to the network that they are holders of specific content and 2) Peers issue queries to find where that content lives. A Content Routing mechanism could be as complex as a DHT or as simple as a registry somewhere in the network.

Some available content routing modules are:

- @libp2p/kad-dht
- @helia/delegated-routing-v1-http-api-client
- @libp2p/delegated-content-routing

> [!NOTE]
> The `@helia/delegated-routing-v1-http-api-client` module is a client for the IPFS Delegated Routing V1 HTTP API. It is not a libp2p module, but it can be used in conjunction with libp2p to provide content and peer routing functionality.
> For most purposes, `@helia/delegated-routing-v1-http-api-client` should be favoured over `@libp2p/delegated-content-routing` for delegated routing, as it is more broadly adopted by the ecosystem and doesn't rely on Kubo specific APIs.

If none of the available content routing protocols fulfil your needs, you can create a libp2p compatible one. A libp2p content routing protocol just needs to be compliant with the Content Routing Interface.

Peer Routing

> Peer Routing offers a way to find other peers in the network by issuing queries using a Peer Routing algorithm, which may be iterative or recursive. If the algorithm is unable to find the target peer, it will return the peers that are "closest" to the target peer, using a distance metric defined by the algorithm.

Some available peer routing modules are:

- @libp2p/kad-dht
- @helia/delegated-routing-v1-http-api-client
- @libp2p/delegated-peer-routing
  If none of the available peer routing protocols fulfills your needs, you can create a libp2p

compatible one. A libp2p peer routing protocol just needs to be compliant with the Peer Routing Interface.

> [!NOTE]
> The `@helia/delegated-routing-v1-http-api-client` module is a client for the IPFS Delegated Routing V1 HTTP API. It is not a libp2p module, but it can be used in conjunction with libp2p to provide content and peer routing functionality.
> For most purposes, `@helia/delegated-routing-v1-http-api-client` should be favoured over `@libp2p/delegated-content-routing` for delegated routing, as it is more broadly adopted by the ecosystem and doesn't rely on Kubo specific APIs.

DHT

> A DHT can provide content and peer routing capabilities in a p2p system, as well as peer discovery capabilities.

The DHT implementation currently available is @libp2p/kad-dht. This implementation is largely based on the Kademlia white paper, augmented with notions from S/Kademlia, Coral and mainlineDHT.

If this DHT implementation does not fulfill your needs and you want to create or use your own implementation, please get in touch with us through a github issue. We plan to work on improving the ability to bring your own DHT in a future release.

If you want to know more about libp2p DHT, you should read the following content:

- https://docs.libp2p.io/concepts/fundamentals/protocols/#kad-dht
- https://github.com/libp2p/specs/pull/108

Pubsub

> Publish/Subscribe is a system where peers congregate around topics they are interested in. Peers interested in a topic are said to be subscribed to that topic and should receive the data published on it from other peers.

Some available pubsub routers are:

- @chainsafe/libp2p-gossipsub
- @libp2p/floodsub (Not for production use)

If none of the available pubsub routers fulfills your needs, you can create a libp2p compatible one. A libp2p pubsub router just needs to be created on top of @libp2p/pubsub, which ensures `js-libp2p` API expectations.

If you want to know more about libp2p pubsub, you should read the following content:

- https://docs.libp2p.io/concepts/publish-subscribe
- https://github.com/libp2p/specs/tree/master/pubsub

Customizing libp2p

When creating a libp2p node, there are a number of services which are optional but will probably be needed for your use case such as the kademlia, peer discovery and pubsub services for example. These are passed into the `services` object upon creating a node. You can also pass in custom services that will be used to create the node. This is done by providing your custom implementation to the `services` object, which should have the following structure:

```
const modules = {
  transports: [],
  streamMuxers: [],
  connectionEncrypters: [],
  contentRouting: [],
  peerRouting: [],
  peerDiscovery: [],
  services: {
    myService: myServiceImplementation
  }
}
```

Moreover, the majority of the modules can be customized via option parameters. This way, it is also possible to provide this options through a `config` object. This config object should have the property name of each building block to configure, the same way as the modules specification.

Besides the `modules` and `config`, libp2p allows other internal options and configurations:

- `datastore`: an instance of ipfs/interface-datastore modules.
  - This is used in modules such as the DHT. If it is not provided, `js-libp2p` will use an in memory datastore.
- `peerId`: the identity of the node, an instance of libp2p/js-peer-id.
  - This is particularly useful if you want to reuse the same `peer-id`, as well as for modules like `libp2p-delegated-content-routing`, which need a `peer-id` in their instantiation.
- `addresses`: an object containing `listen`, `announce` and `announceFilter`:
  - `listen` addresses will be provided to the libp2p underlying transports for listening on them.
  - `announce` addresses will be used to compute the advertises that the node should advertise to the network.
  - `announceFilter`: filter function used to filter announced addresses programmatically: `(ma: Array<multiaddr>) => Array<multiaddr>`. Default: returns all addresses. `libp2p-utils` provides useful multiaddr utilities to create your filters.

It's important to note that some services depend on others in order to function optimally, this is further explained in the examples below.

Examples

Basic setup

```
// Creating a libp2p node with:
//   listen on tcp ports 9001 and 9002 on all interfaces
//   transport: websockets + tcp
//   stream-muxing: yamux
//   crypto-channel: noise
//   discovery: multicast-dns
//   dht: kad-dht
//   pubsub: gossipsub

import { createLibp2p } from 'libp2p'
import { tcp } from '@libp2p/tcp'
import { webSockets } from '@libp2p/websockets'
import { noise } from '@chainsafe/libp2p-noise'
import { mdns } from '@libp2p/mdns'
import { kadDHT } from '@libp2p/kad-dht'
import { gossipsub } from 'libp2p-gossipsub'
import { yamux } from '@chainsafe/libp2p-yamux'

const node = await createLibp2p({
  addresses: {
    listen: [
      '/ip4/0.0.0.0/tcp/9001/ws',
      '/ip4/0.0.0.0/tcp/9002',
    ],
  },
  transports: [
    tcp(),
    webSockets()
  ],
  streamMuxers: [yamux()],
  connectionEncrypters: [noise()],
  peerDiscovery: [mdns()],
  services: {
    dht: kadDHT(),
    pubsub: gossipsub()
  }
})
```

Customizing Peer Discovery

```
import { createLibp2p } from 'libp2p'
import { tcp } from '@libp2p/tcp'
import { yamux } from '@chainsafe/libp2p-yamux'
import { noise } from '@chainsafe/libp2p-noise'
import { mdns } from '@libp2p/mdns'
import { bootstrap } from '@libp2p/bootstrap'

const node = await createLibp2p({
  transports: [tcp()],
```

```
  streamMuxers: [yamux()],
  connectionEncrypters: [noise()],
  peerDiscovery: [
    mdns({
      interval: 1000
    }),
    bootstrap({
      list: [ // A list of bootstrap peers to connect to starting up the node

"/ip4/104.131.131.82/tcp/4001/ipfs/QmaCpDMGvV2BGHeYERUEnRQAwe3N8SzbUtfsmvsqQLuvuJ",

"/dnsaddr/bootstrap.libp2p.io/ipfs/QmNnooDu7bfjPFoTZYxMNLWUQJyrVwtbZg5gBMjTezGAJN",

"/dnsaddr/bootstrap.libp2p.io/ipfs/QmQCU2EcMqAqQPR2i9bChDtGNJchTbq5TbXJJ16u19uLTa",
      ]
    })
  ]
})

node.addEventListener('peer:discovery', (event) => {
  console.log('Discovered new peer:', event.detail.id.toString())
  node.dial(event.detail.multiaddrs)
})
```

Note the `bootstrap` peer discovery module will automatically dial the bootstrap peers when the node starts up, while `mdns` will only trigger the `peer:discovery` event when a new peer is discovered.

Customizing Pubsub

Before a peer can subscribe to a topic it must find other peers and establish network connections with them. The pub/sub system doesn't have any way to discover peers by itself. Instead, it relies upon the application to find new peers on its behalf, a process called ambient peer discovery.

This means that pubsub requires the identify service to be configured in order to exchange peer information with other peers, including lists of supported protocols.

Potential methods for discovering peers include:

- Distributed hash tables
- Local network broadcasts
- Centralized trackers or rendezvous points
- Lists of bootstrap peers

```
import { createLibp2p } from 'libp2p'
import { tcp } from '@libp2p/tcp'
import { yamux } from '@chainsafe/libp2p-yamux'
```

```
import { noise } from '@chainsafe/libp2p-noise'
import { gossipsub } from 'libp2p-gossipsub'
import { SignaturePolicy } from '@libp2p/interface'
import { identify } from '@libp2p/identify'

const node = await createLibp2p({
    transports: [
      tcp()
    ],
    streamMuxers: [
      yamux()
    ],
    connectionEncrypters: [
      noise()
    ],
    services: {
      identify: identify(),
      pubsub: gossipsub({
        emitSelf: false,                                // whether the node
should emit to self on publish
        globalSignaturePolicy: SignaturePolicy.StrictSign // message signing
policy
      })
    }
  })
```

Customizing DHT

As explained in previous sections the kad-dht is a Distributed Hash Table based on the Kademlia routing algorithm, with some modifications.

libp2p uses the DHT as an implementation of its peer routing and content routing functionality.

The kadDHT service requires the Identify service to discover other peers that support the protocol and which allows it to use them to make network queries.

```
import { createLibp2p } from 'libp2p'
import { tcp } from '@libp2p/tcp'
import { noise } from '@chainsafe/libp2p-noise'
import { kadDHT } from '@libp2p/kad-dht'
import { identify } from '@libp2p/identify'

const node = await createLibp2p({
  transports: [
    tcp()
  ],
  streamMuxers: [
    yamux()
  ],
  connectionEncrypters: [
```

```
    noise()
  ],
  services: {
    identify: identify(),
    dht: kadDHT({
      kBucketSize: 20,
      clientMode: false           // Whether to run the WAN DHT in client or
server mode (default: client mode)
    })
  }
})
```

Setup with Delegated Content and Peer Routing

```
import { createLibp2p } from 'libp2p'
import { tcp } from '@libp2p/tcp'
import { yamux } from '@chainsafe/libp2p-yamux'
import { noise } from '@chainsafe/libp2p-noise'
import { createDelegatedRoutingV1HttpApiClient } from '@helia/delegated-
routing-v1-http-api-client'
const node = await createLibp2p({
  transports: [
    tcp()
  ],
  streamMuxers: [
    yamux()
  ],
  connectionEncrypters: [
    noise()
  ],
  services: {
    delegatedRouting: () =>
createDelegatedRoutingV1HttpApiClient('https://delegated-ipfs.dev'),
  }
})
```

Setup with Relay

Circuit Relay, is a protocol for tunneling traffic through relay peers when two peers are unable to connect to each other directly.

When a peer to be available to be connected to via a relay, it first needs to find a peer that supports the Circuit Relay protocol.

It can search the network for providers of the service and/or it can rely on ambient discovery via the identify protocol, during which peers exchange lists of protocols they support.

Thus, it is recommended to include the Identify service in your services configuration when you hope to find a relay peer that supports the Circuit Relay protocol.

```
import { createLibp2p } from 'libp2p'
import { tcp } from '@libp2p/tcp'
import { yamux } from '@chainsafe/libp2p-yamux'
import { noise } from '@chainsafe/libp2p-noise'
import { circuitRelayTransport, circuitRelayServer } from '@libp2p/circuit-
relay-v2'
import { identify } from '@libp2p/identify'


const node = await createLibp2p({
  addresses: {
    listen: {
      // discover a relay using the routing
      '/p2p-circuit'
    }
  },
  transports: [
    tcp(),
    circuitRelayTransport()
  ],
  streamMuxers: [
    yamux()
  ],
  connectionEncrypters: [
    noise()
  ],
  connectionGater: {
    // used by the server - return true to deny a reservation to the remote
peer
    denyInboundRelayReservation: (source: PeerId) => Promise<boolean>

    // used by the server - return true to deny a relay connection request from
the source to the destination peer
    denyOutboundRelayedConnection: (source: PeerId, destination: PeerId) =>
Promise<boolean>

    // used by the client - return true to deny a relay connection from the
remote relay and peer
    denyInboundRelayedConnection: (relay: PeerId, remotePeer: PeerId) =>
Promise<boolean>
  },
  services: {
    identify: identify(),
    relay: circuitRelayServer({ // makes the node function as a relay server
      hopTimeout: 30 * 1000, // incoming relay requests must be resolved within
this time limit
      advertise: true,
      reservations: {
```

```
        maxReservations: 15 // how many peers are allowed to reserve relay
slots on this server
        reservationClearInterval: 300 * 1000 // how often to reclaim stale
reservations
        applyDefaultLimit: true // whether to apply default data/duration
limits to each relayed connection
        defaultDurationLimit: 2 * 60 * 1000 // the default maximum amount of
time a relayed connection can be open for
        defaultDataLimit: BigInt(2 << 7) // the default maximum number of bytes
that can be transferred over a relayed connection
        maxInboundHopStreams: 32 // how many inbound HOP streams are allow
simultaneously
        maxOutboundHopStreams: 64 // how many outbound HOP streams are allow
simultaneously
      }
    }),
  }
})
```

Setup with Automatic Reservations

In this configuration the libp2p node will search the network for one relay with a free reservation slot. When it has found one and negotiated a relay reservation, the relayed address will appear in the output of `libp2p.getMultiaddrs()`.

```
import { createLibp2p } from 'libp2p'
import { tcp } from '@libp2p/tcp'
import { yamux } from '@chainsafe/libp2p-yamux'
import { noise } from '@chainsafe/libp2p-noise'
import { circuitRelayTransport } from '@libp2p/circuit-relay-v2'

const node = await createLibp2p({
  addresses: {
    listen: [
      '/p2p-circuit'
    ]
  },
  transports: [
    tcp(),
    circuitRelayTransport()
  ],
  streamMuxers: [
    yamux()
  ],
  connectionEncrypters: [
    noise()
  ]
})
```

Setup with Pre-configured Reservations

In this configuration the libp2p node is a circuit relay client which connects to a relay, `/ip4/123.123.123.123/p2p/QmRelay` which has been configured to have slots available.

```
import { createLibp2p } from 'libp2p'
import { tcp } from '@libp2p/tcp'
import { noise } from '@chainsafe/libp2p-noise'
import { circuitRelayTransport } from '@libp2p/circuit-relay-v2'

const node = await createLibp2p({
  transports: [
    tcp(),
    circuitRelayTransport()
  ],
  addresses: {
    listen: [
      '/ip4/123.123.123.123/p2p/QmRelay/p2p-circuit' // a known relay node with
reservation slots available
    ]
  },
  streamMuxers: [
    yamux()
  ],
  connectionEncrypters: [
    noise()
  ]
})
```

Setup with Keychain

Libp2p allows you to setup a secure keychain to manage your keys. The keychain configuration object should have the following properties:

Name	Type	Description
pass	`string`	Passphrase to use in the keychain (minimum of 20 characters).
dek	`DEKConfig`	the default options for generating the derived encryption key, which, along with the passphrase are input to the PBKDF2 function. For more info see: https://github.com/libp2p/js-libp2p-keychain

The keychain will store keys encrypted in the datastore which default is an in memory datastore. If you want to store the keys on disc you need to initialize libp2p with a suitable datastore implementation.

```
import { createLibp2p } from 'libp2p'
import { tcp } from '@libp2p/tcp'
import { yamux } from '@chainsafe/libp2p-yamux'
```

```
import { noise } from '@chainsafe/libp2p-noise'
import { FsDatastore } from 'datastore-fs';


const datastore = new FsDatastore('path/to/store')
await datastore.open()

const node = await createLibp2p({
  transports: [
    tcp()
  ],
  streamMuxers: [
    yamux()
  ],
  connectionEncrypters: [
    noise()
  ],
  keychain: {
    pass: 'not-safe-password-123456789',
  },
  datastore
})
```

Configuring Connection Manager

The Connection Manager manages connections to peers in libp2p. It controls opening closing connections but also pruning connections when certain limits are exceeded. If Metrics are enabled, you can also configure the Connection Manager to monitor the bandwidth of libp2p and prune connections as needed. You can read more about what Connection Manager does at ./CONNECTION_MANAGER.md. The configuration values below show the defaults for Connection Manager.

See the API docs for a full run list and discussion of all Connection Manager options.

```
import { createLibp2p } from 'libp2p'
import { tcp } from '@libp2p/tcp'
import { yamux } from '@chainsafe/libp2p-yamux'
import { noise } from '@chainsafe/libp2p-noise'

const node = await createLibp2p({
  transports: [
    tcp()
  ],
  streamMuxers: [
    yamux()
  ],
  connectionEncrypters: [
    noise()
  ],
  connectionManager: {
    maxConnections: Infinity
```

```
  }
})
```

Configuring Connection Gater

The Connection Gater allows us to prevent making incoming and outgoing connections to peers and storing
multiaddrs in the address book.

The order in which methods are called is as follows:

Outgoing connections

1. `connectionGater.denyDialPeer(...)`
2. `connectionGater.denyDialMultiaddr(...)`
3. `connectionGater.denyOutboundConnection(...)`
4. `connectionGater.denyOutboundEncryptedConnection(...)`
5. `connectionGater.denyOutboundUpgradedConnection(...)`

Incoming connections

1. `connectionGater.denyInboundConnection(...)`
2. `connectionGater.denyInboundEncryptedConnection(...)`
3. `connectionGater.denyInboundUpgradedConnection(...)`

```
const node = await createLibp2p({
  // .. other config
  connectionGater: {
    /**
     * denyDialMultiaddr tests whether we're permitted to Dial the
     * specified peer.
     *
     * This is called by the dialer.connectToPeer implementation before
     * dialling a peer.
     *
     * Return true to prevent dialing the passed peer.
     */
    denyDialPeer: (peerId: PeerId) => Promise<boolean>

    /**
     * denyDialMultiaddr tests whether we're permitted to dial the specified
     * multiaddr for the given peer.
     *
     * This is called by the dialer.connectToPeer implementation after it has
     * resolved the peer's addresses, and prior to dialling each.
     *
     * Return true to prevent dialing the passed peer on the passed multiaddr.
     */
```

```
    denyDialMultiaddr: (multiaddr: Multiaddr) => Promise<boolean>

    /**
     * denyInboundConnection tests whether an incipient inbound connection is
allowed.
     *
     * This is called by the upgrader, or by the transport directly (e.g. QUIC,
     * Bluetooth), straight after it has accepted a connection from its socket.
     *
     * Return true to deny the incoming passed connection.
     */
    denyInboundConnection: (maConn: MultiaddrConnection) => Promise<boolean>

    /**
     * denyOutboundConnection tests whether an incipient outbound connection is
allowed.
     *
     * This is called by the upgrader, or by the transport directly (e.g. QUIC,
     * Bluetooth), straight after it has created a connection with its socket.
     *
     * Return true to deny the incoming passed connection.
     */
    denyOutboundConnection: (peerId: PeerId, maConn: MultiaddrConnection) =>
Promise<boolean>

    /**
     * denyInboundEncryptedConnection tests whether a given connection, now
encrypted,
     * is allowed.
     *
     * This is called by the upgrader, after it has performed the security
     * handshake, and before it negotiates the muxer, or by the directly by the
     * transport, at the exact same checkpoint.
     *
     * Return true to deny the passed secured connection.
     */
    denyInboundEncryptedConnection: (peerId: PeerId, maConn:
MultiaddrConnection) => Promise<boolean>

    /**
     * denyOutboundEncryptedConnection tests whether a given connection, now
encrypted,
     * is allowed.
     *
     * This is called by the upgrader, after it has performed the security
     * handshake, and before it negotiates the muxer, or by the directly by the
     * transport, at the exact same checkpoint.
     *
     * Return true to deny the passed secured connection.
     */
    denyOutboundEncryptedConnection: (peerId: PeerId, maConn:
MultiaddrConnection) => Promise<boolean>
```

```
    /**
     * denyInboundUpgradedConnection tests whether a fully capable connection
is allowed.
     *
     * This is called after encryption has been negotiated and the connection
has been
     * multiplexed, if a multiplexer is configured.
     *
     * Return true to deny the passed upgraded connection.
     */
    denyInboundUpgradedConnection: (peerId: PeerId, maConn:
MultiaddrConnection) => Promise<boolean>

    /**
     * denyOutboundUpgradedConnection tests whether a fully capable connection
is allowed.
     *
     * This is called after encryption has been negotiated and the connection
has been
     * multiplexed, if a multiplexer is configured.
     *
     * Return true to deny the passed upgraded connection.
     */
    denyOutboundUpgradedConnection: (peerId: PeerId, maConn:
MultiaddrConnection) => Promise<boolean>

    /**
     * Used by the address book to filter passed addresses.
     *
     * Return true to allow storing the passed multiaddr for the passed peer.
     */
    filterMultiaddrForPeer: (peer: PeerId, multiaddr: Multiaddr) =>
Promise<boolean>
  }
})
```

Configuring Transport Manager

The Transport Manager is responsible for managing the libp2p transports life cycle. This includes starting listeners for the provided listen addresses, closing these listeners and dialing using the provided transports. By default, if a libp2p node has a list of multiaddrs for listening on and there are no valid transports for those multiaddrs, libp2p will throw an error on startup and shutdown. However, for some applications it is perfectly acceptable for libp2p nodes to start in dial only mode if all the listen multiaddrs failed. This error tolerance can be enabled as follows:

```
import { createLibp2p } from 'libp2p'
import { tcp } from '@libp2p/tcp'
import { yamux } from '@chainsafe/libp2p-yamux'
import { noise } from '@chainsafe/libp2p-noise'
```

```
import { FaultTolerance } from '@libp2p/interface-transport'

const node = await createLibp2p({
  transports: [
    tcp()
  ],
  streamMuxers: [
    yamux()
  ],
  connectionEncrypters: [
    noise()
  ],
  transportManager: {
    faultTolerance: FaultTolerance.NO_FATAL
  }
})
```

Configuring Metrics

Metrics are disabled in libp2p by default. You can enable and configure them as follows:

Name	Type	Description
enabled	`boolean`	Enabled metrics collection.
computeThrottleMaxQueueSize	`number`	How many messages a stat will queue before processing.
computeThrottleTimeout	`number`	Time in milliseconds a stat will wait, after the last item was added, before processing.
movingAverageIntervals	`Array<number>`	The moving averages that will be computed.
maxOldPeersRetention	`number`	How many disconnected peers we will retain stats for.

The below configuration example shows how the metrics should be configured. Aside from enabled being `false` by default, the following default configuration options are listed below:

```
import { createLibp2p } from 'libp2p'
import { tcp } from '@libp2p/tcp'
import { yamux } from '@chainsafe/libp2p-yamux'
import { noise } from '@chainsafe/libp2p-noise'

const node = await createLibp2p({
  transports: [
    tcp()
  ],
  streamMuxers: [
    yamux()
```

```
  ],
  connectionEncrypters: [
    noise()
  ],
  metrics: {
    enabled: true,
    computeThrottleMaxQueueSize: 1000,
    computeThrottleTimeout: 2000,
    movingAverageIntervals: [
      60 * 1000, // 1 minute
      5 * 60 * 1000, // 5 minutes
      15 * 60 * 1000 // 15 minutes
    ],
    maxOldPeersRetention: 50
  }
})
```

Configuring PeerStore

PeerStore persistence is disabled in libp2p by default. You can enable and configure it as follows. Aside from enabled being `false` by default, it will need an implementation of a datastore. Take into consideration that using the memory datastore will be ineffective for persistence.

The threshold number represents the maximum number of "dirty peers" allowed in the PeerStore, i.e. peers that are not updated in the datastore. In this context, browser nodes should use a threshold of 1, since they might not "stop" properly in several scenarios and the PeerStore might end up with un-flushed records when the window is closed.

Name	Type	Description
persistence	`boolean`	Is persistence enabled.
threshold	`number`	Number of dirty peers allowed.

The below configuration example shows how the PeerStore should be configured. Aside from persistence being `false` by default, the following default configuration options are listed below:

```
import { createLibp2p } from 'libp2p'
import { tcp } from '@libp2p/tcp'
import { yamux } from '@chainsafe/libp2p-yamux'
import { noise } from '@chainsafe/libp2p-noise'
import { LevelDatastore } from 'datastore-level'

const datastore = new LevelDatastore('path/to/store')
await datastore.open() // level database must be ready before node boot

const node = await createLibp2p({
  datastore, // pass the opened datastore
  transports: [
```

```
    tcp()
  ],
  streamMuxers: [
    yamux()
  ],
  connectionEncrypters: [
    noise()
  ],
  peerStore: {
    persistence: true,
    threshold: 5
  }
})
```

Customizing Transports

Some Transports can be passed additional options when they are created. For example, webRTC accepts optional DataChannel Options. In addition to libp2p passing itself and an `Upgrader` to handle connection upgrading, libp2p will also pass the options, if they are provided, from `config.transport`.

```
import { createLibp2p } from 'libp2p'
import { yamux } from '@chainsafe/libp2p-yamux'
import { noise } from '@chainsafe/libp2p-noise'
import { webRTC } from '@libp2p/webrtc'


const node = await createLibp2p({
  transports: [
    webRTC({
      dataChannel: {
        maxMessageSize: 10
      }
    })
  ],
  streamMuxers: [
    yamux()
  ],
  connectionEncrypters: [
    noise()
  ]
})
```

During Libp2p startup, transport listeners will be created for the configured listen multiaddrs. Some transports support custom listener options and you can set them using the `listenerOptions` in the transport configuration. For example, webRTC transport listener supports the configuration of ice servers (STUN/TURN) config as follows:

```
const node = await createLibp2p({
  transports: [
    webRTC({
      rtcConfiguration: {
        iceServers:[{
          urls: [
            'stun:stun.l.google.com:19302',
            'stun:global.stun.twilio.com:3478'
          ]
        }]
    }
})
  ],
  streamMuxers: [
    yamux()
  ],
  connectionEncrypters: [
    noise()
  ],
  addresses: {
    listen: ['/webrtc'] // your webrtc dns multiaddr
  }
})
```

Configuring AutoNAT

In order for a node to have confidence that it is publicly dialable, the AutoNAT protocol can be used to instruct remote peers to dial the node on the addresses that it believes to be public.

If enough peers report that this address is dialable, the node is free to change it's relationship to the rest of the network; for example, it could become a DHT server or fulfil some other public role.

For more information see https://docs.libp2p.io/concepts/nat/autonat/#what-is-autonat

```
import { createLibp2p } from 'libp2p'
import { autoNAT } from '@libp2p/autonat'

const node = await createLibp2p({
  services: {
    nat: autoNAT({
      protocolPrefix: 'my-node', // this should be left as the default value to
ensure maximum compatibility
      timeout: 30000, // the remote must complete the AutoNAT protocol within
this timeout
      maxInboundStreams: 1, // how many concurrent inbound AutoNAT protocols
streams to allow on each connection
      maxOutboundStreams: 1 // how many concurrent outbound AutoNAT protocols
streams to allow on each connection
    })
```

```
  }
})
```

Configuring UPnP NAT Traversal

Network Address Translation (NAT) is a function performed by your router to enable multiple devices on your local network to share a single IPv4 address. It's done transparently for outgoing connections, ensuring the correct response traffic is routed to your computer, but if you wish to accept incoming connections some configuration is necessary.

Some home routers support UPnP NAT which allows network devices to request traffic to be forwarded from public facing ports that would otherwise be firewalled.

If your router supports this, libp2p can be configured to use it as follows:

```
import { createLibp2p } from 'libp2p'
import { uPnPNATService } from '@libp2p/upnp-nat'

const node = await createLibp2p({
  services: {
    nat: uPnPNATService({
      description: 'my-node', // set as the port mapping description on the
router, defaults the current libp2p version and your peer id
      gateway: '192.168.1.1', // leave unset to auto-discover
      externalIp: '80.1.1.1', // leave unset to auto-discover
      localAddress: '129.168.1.123', // leave unset to auto-discover
      ttl: 7200, // TTL for port mappings (min 20 minutes)
      keepAlive: true, // Refresh port mapping after TTL expires
    })
  }
})
```

Browser support

Browsers cannot open TCP ports or send the UDP datagrams necessary to configure external port mapping - to accept incoming connections in the browser please use a WebRTC transport.

UPnP and NAT-PMP

By default under nodejs libp2p will attempt to use UPnP to configure your router to allow incoming connections to any TCP transports that have been configured.

NAT-PMP is a feature of some modern routers which performs a similar job to UPnP. NAT-PMP is disabled by default, if enabled libp2p will try to use NAT-PMP and will fall back to UPnP if it fails.

Configuring protocol name

Changing the protocol name prefix can isolate default public network (IPFS) for custom purposes.

```
import { createLibp2p } from 'libp2p'
import { identify } from '@libp2p/identify'
import { ping } from 'libp2p/@ping'

const node = await createLibp2p({
  services: {
    identify: identify({
      protocolPrefix: 'ipfs' // default
    }),
    ping: ping({
      protocolPrefix: 'ipfs' // default
    })
  }
})
/*
protocols: [
  "/ipfs/id/1.0.0", // identify service protocol (if we have multiplexers)
  "/ipfs/id/push/1.0.0", // identify service push protocol (if we have
multiplexers)
  "/ipfs/ping/1.0.0", // ping protocol
]
*/
```

Configuration examples

As libp2p is designed to be a modular networking library, its usage will vary based on individual project needs. We've included links to some existing project configurations for your reference, in case you wish to replicate their configuration:

- libp2p-Helia-nodejs - libp2p configuration used by Helia when running in Node.js
- libp2p-Helia-browser - libp2p configuration used by Helia when running in a Browser

If you have developed a project using `js-libp2p`, please consider submitting your configuration to this list so that it can be found easily by other users.

The examples repo is also a good source of help for finding a configuration that suits your needs.

Limits

Configuring the various limits of your node is important to protect it when it is part of hostile of adversarial networks. See LIMITS.md for a full breakdown of the various built in protections and safeguards.