py-libp2p Documentation

Release 0.1.5

The Ethereum Foundation

Apr 27, 2024

GENERAL

The Python implementation of the libp2p networking stack

CHAPTER

ONE

CONTENTS

1.1 Introduction

py-libp2p is the Python implementation of the libp2p networking stack. It hopes to someday be fully compatible with other implementations of libp2p.

This project is still in its early stages and is not yet ready for production use.

1.1.1 Further Reading

- libp2p main site
- Tutorial: Introduction to libp2p

1.2 Quickstart

TODO - add quickstart instructions

1.3 Release Notes

1.3.1 libp2p v0.1.5 (2020-03-25)

Features

- Dial all multiaddrs stored for a peer when attempting to connect (not just the first one in the peer store). (#386)
- Migrate transport stack to trio-compatible code. Merge in #404. (#396)
- Migrate network stack to trio-compatible code. Merge in #404. (#397)
- Migrate host, peer and protocols stacks to trio-compatible code. Merge in #404. (#398)
- Migrate muxer and security transport stacks to trio-compatible code. Merge in #404. (#399)
- Migrate pubsub stack to trio-compatible code. Merge in #404. (#400)
- Fix interop tests w/ new trio-style code. Merge in #404. (#401)
- Fix remainder of test code w/ new trio-style code. Merge in #404. (#402)
- Add initial infrastructure for *noise* security transport. (#405)

- Add *PatternXX* of *noise* security transport. (#406)
- The *msg_id* in a pubsub message is now configurable by the user of the library. (#410)

Bugfixes

- Use *sha256* when calculating a peer's ID from their public key in Kademlia DHTs. (#385)
- Store peer ids in `set` instead of `list` and check if peer id exists in `dict` before accessing to prevent `KeyError`. (#387)
- Do not close a connection if it has been reset. (#394)

Internal Changes - for py-libp2p Contributors

- Add support for *fastecdsa* on windows (and thereby supporting windows installation via *pip*) (#380)
- Prefer f-string style formatting everywhere except logging statements. (#389)
- Mark *lru* dependency as third-party to fix a windows inconsistency. (#392)
- Bump *multiaddr* dependency to version *0.0.9* so that multiaddr objects are hashable. (#393)
- Remove incremental mode of mypy to disable some warnings. (#403)

1.3.2 libp2p v0.1.4 (2019-12-12)

Features

- Added support for Python 3.6 (#372)
- Add signing and verification to pubsub (#362)

Internal Changes - for py-libp2p Contributors

- Refactor and cleanup gossipsub (#373)

1.3.3 libp2p v0.1.3 (2019-11-27)

Bugfixes

- Handle Stream* errors (like `StreamClosed`) during calls to `stream.write()` and `stream.read()` (#350)
- Relax the protobuf dependency to play nicely with other libraries. It was pinned to 3.9.0, and now permits v3.10 up to (but not including) v4. (#354)
- Fixes KeyError when peer in a stream accidentally closes and resets the stream, because handlers for both will try to `del streams[stream_id]` without checking if the entry still exists. (#355)

Improved Documentation

- Use Sphinx & autodoc to generate docs, now available on py-libp2p.readthedocs.io (#318)

Internal Changes - for py-libp2p Contributors

- Added Makefile target to test a packaged version of libp2p before release. (#353)
- Move helper tools from `tests/` to `libp2p/tools/`, and some mildly-related cleanups. (#356)

Miscellaneous changes

- #357

1.3.4 v0.1.2

Welcome to the great beyond, where changes were not tracked by release…

1.4 examples package

1.4.1 Subpackages

examples.chat package

Submodules

examples.chat.chat module

examples.chat.chat.**main**() → None

async examples.chat.chat.**read_data**(*stream:* INetStream) → None

async examples.chat.chat.**run**(*port: int*, *destination: str*) → None

async examples.chat.chat.**write_data**(*stream:* INetStream) → None

Module contents

1.4.2 Module contents

1.5 libp2p package

1.5.1 Subpackages

libp2p.crypto package

Subpackages

libp2p.crypto.pb package

Submodules

libp2p.crypto.pb.crypto_pb2 module

Generated protocol buffer code.

class libp2p.crypto.pb.crypto_pb2.**PrivateKey**

Bases: Message, Message

DESCRIPTOR = <google._upb._message.Descriptor object>

class libp2p.crypto.pb.crypto_pb2.**PublicKey**

Bases: Message, Message

DESCRIPTOR = <google._upb._message.Descriptor object>

Module contents

Submodules

libp2p.crypto.authenticated_encryption module

class libp2p.crypto.authenticated_encryption.**EncryptionParameters**(*cipher_type: str*, *hash_type: str*, *iv: bytes*, *mac_key: bytes*, *cipher_key: bytes*)

Bases: object

cipher_key: bytes

cipher_type: str

hash_type: str

iv: bytes

mac_key: bytes

exception libp2p.crypto.authenticated_encryption.**InvalidMACException**

Bases: Exception

class libp2p.crypto.authenticated_encryption.**MacAndCipher**(*parameters:* EncryptionParameters)

Bases: object

authenticate(*data: bytes*) → bytes

decrypt_if_valid(*data_with_tag: bytes*) → bytes

encrypt(*data: bytes*) → bytes

libp2p.crypto.authenticated_encryption.**initialize_pair**(*cipher_type: str*, *hash_type: str*, *secret: bytes*) → Tuple[*EncryptionParameters*, *EncryptionParameters*]

Return a pair of `Keys` for use in securing a communications channel with authenticated encryption derived from the `secret` and using the requested `cipher_type` and `hash_type`.

libp2p.crypto.ecc module

class libp2p.crypto.ecc.**ECCPrivateKey**(*impl: int*, *curve: <MagicMock name='mock.curve.Curve' id='140457861788048'>*)

Bases: *PrivateKey*

get_public_key() → *PublicKey*

get_type() → *KeyType*

Returns the `KeyType` for `self`.

classmethod new(*curve: str*) → *ECCPrivateKey*

sign(*data: bytes*) → bytes

to_bytes() → bytes

Returns the byte representation of this key.

class libp2p.crypto.ecc.**ECCPublicKey**(*impl: <MagicMock name='mock.point.Point' id='140457861800048'>*, *curve: <MagicMock name='mock.curve.Curve' id='140457861788048'>*)

Bases: *PublicKey*

classmethod from_bytes(*data: bytes*, *curve: str*) → *ECCPublicKey*

get_type() → *KeyType*

Returns the `KeyType` for `self`.

to_bytes() → bytes

Returns the byte representation of this key.

verify(*data: bytes*, *signature: bytes*) → bool

Verify that `signature` is the cryptographic signature of the hash of `data`.

libp2p.crypto.ecc.**create_new_key_pair**(*curve: str*) → *KeyPair*

Return a new ECC keypair with the requested `curve` type, e.g. “P-256”.

libp2p.crypto.ecc.**infer_local_type**(*curve: str*) → <MagicMock name='mock.curve.Curve' id='140457861788048'>

Convert a `str` representation of some elliptic curve to a representation understood by the backend of this module.

libp2p.crypto.ed25519 module

class libp2p.crypto.ed25519.**Ed25519PrivateKey**(*impl: PrivateKey*)

Bases: *PrivateKey*

classmethod from_bytes(*data: bytes*) → *Ed25519PrivateKey*

get_public_key() → *PublicKey*

get_type() → *KeyType*

Returns the KeyType for `self`.

classmethod new(*seed: bytes | None = None*) → *Ed25519PrivateKey*

sign(*data: bytes*) → bytes

to_bytes() → bytes

Returns the byte representation of this key.

class libp2p.crypto.ed25519.**Ed25519PublicKey**(*impl: PublicKey*)

Bases: *PublicKey*

classmethod from_bytes(*key_bytes: bytes*) → *Ed25519PublicKey*

get_type() → *KeyType*

Returns the KeyType for `self`.

to_bytes() → bytes

Returns the byte representation of this key.

verify(*data: bytes*, *signature: bytes*) → bool

Verify that `signature` is the cryptographic signature of the hash of `data`.

libp2p.crypto.ed25519.**create_new_key_pair**(*seed: bytes | None = None*) → *KeyPair*

libp2p.crypto.exceptions module

exception libp2p.crypto.exceptions.**CryptographyError**

Bases: *BaseLibp2pError*

exception libp2p.crypto.exceptions.**MissingDeserializerError**

Bases: *CryptographyError*

Raise if the requested deserialization routine is missing for some type of cryptographic key.

libp2p.crypto.key_exchange module

libp2p.crypto.key_exchange.**create_ephemeral_key_pair**(*curve_type: str*) → Tuple[*PublicKey*, Callable[[bytes], bytes]]

Facilitates ECDH key exchange.

libp2p.crypto.keys module

class libp2p.crypto.keys.**Key**

Bases: ABC

A Key represents a cryptographic key.

abstract get_type() → *KeyType*

Returns the KeyType for self.

abstract to_bytes() → bytes

Returns the byte representation of this key.

class libp2p.crypto.keys.**KeyPair**(*private_key:* libp2p.crypto.keys.PrivateKey, *public_key:* libp2p.crypto.keys.PublicKey)

Bases: object

private_key: *PrivateKey*

public_key: *PublicKey*

class libp2p.crypto.keys.**KeyType**(*value*)

Bases: Enum

An enumeration.

ECC_P256 = 4

ECDSA = 3

Ed25519 = 1

RSA = 0

Secp256k1 = 2

class libp2p.crypto.keys.**PrivateKey**

Bases: *Key*

A PrivateKey represents a cryptographic private key.

classmethod deserialize_from_protobuf(*protobuf_data: bytes*) → *PrivateKey*

abstract get_public_key() → *PublicKey*

serialize() → bytes

Return the canonical serialization of this Key.

abstract sign(*data: bytes*) → bytes

class libp2p.crypto.keys.**PublicKey**

Bases: *Key*

A PublicKey represents a cryptographic public key.

classmethod deserialize_from_protobuf(*protobuf_data: bytes*) → *PublicKey*

serialize() → bytes

Return the canonical serialization of this Key.

abstract verify(*data: bytes*, *signature: bytes*) → bool

Verify that signature is the cryptographic signature of the hash of data.

libp2p.crypto.rsa module

class libp2p.crypto.rsa.**RSAPrivateKey**(*impl: RsaKey*)

Bases: *PrivateKey*

get_public_key() → *PublicKey*

get_type() → *KeyType*

Returns the KeyType for `self`.

classmethod new(*bits: int = 2048*, *e: int = 65537*) → *RSAPrivateKey*

sign(*data: bytes*) → bytes

to_bytes() → bytes

Returns the byte representation of this key.

class libp2p.crypto.rsa.**RSAPublicKey**(*impl: RsaKey*)

Bases: *PublicKey*

classmethod from_bytes(*key_bytes: bytes*) → *RSAPublicKey*

get_type() → *KeyType*

Returns the KeyType for `self`.

to_bytes() → bytes

Returns the byte representation of this key.

verify(*data: bytes*, *signature: bytes*) → bool

Verify that `signature` is the cryptographic signature of the hash of `data`.

libp2p.crypto.rsa.**create_new_key_pair**(*bits: int = 2048*, *e: int = 65537*) → *KeyPair*

Returns a new RSA keypair with the requested key size (`bits`) and the given public exponent `e`.

Sane defaults are provided for both values.

libp2p.crypto.secp256k1 module

class libp2p.crypto.secp256k1.**Secp256k1PrivateKey**(*impl: PrivateKey*)

Bases: *PrivateKey*

classmethod deserialize(*data: bytes*) → *Secp256k1PrivateKey*

classmethod from_bytes(*data: bytes*) → *Secp256k1PrivateKey*

get_public_key() → *PublicKey*

get_type() → *KeyType*

Returns the KeyType for `self`.

classmethod new(*secret: bytes | None = None*) → *Secp256k1PrivateKey*

sign(*data: bytes*) → bytes

to_bytes() → bytes

Returns the byte representation of this key.

class libp2p.crypto.secp256k1.**Secp256k1PublicKey**(*impl: PublicKey*)

Bases: *PublicKey*

classmethod deserialize(*data: bytes*) → *Secp256k1PublicKey*

classmethod from_bytes(*data: bytes*) → *Secp256k1PublicKey*

get_type() → *KeyType*

Returns the KeyType for self.

to_bytes() → bytes

Returns the byte representation of this key.

verify(*data: bytes*, *signature: bytes*) → bool

Verify that signature is the cryptographic signature of the hash of data.

libp2p.crypto.secp256k1.**create_new_key_pair**(*secret: bytes | None = None*) → *KeyPair*

Returns a new Secp256k1 keypair derived from the provided secret, a sequence of bytes corresponding to some integer between 0 and the group order.

A valid secret is created if None is passed.

libp2p.crypto.serialization module

libp2p.crypto.serialization.**deserialize_private_key**(*data: bytes*) → *PrivateKey*

libp2p.crypto.serialization.**deserialize_public_key**(*data: bytes*) → *PublicKey*

Module contents

libp2p.host package

Submodules

libp2p.host.basic_host module

class libp2p.host.basic_host.**BasicHost**(*network:* INetworkService, *default_protocols: OrderedDict[TProtocol, StreamHandlerFn] = None*)

Bases: *IHost*

BasicHost is a wrapper of a *INetwork* implementation.

It performs protocol negotiation on a stream with multistream-select right after a stream is initialized.

async close() → None

async connect(*peer_info:* PeerInfo) → None

Ensure there is a connection between this host and the peer with given *peer_info.peer_id*. connect will absorb the addresses in peer_info into its internal peerstore. If there is not an active connection, connect will issue a dial, and block until a connection is opened, or an error is returned.

Parameters

peer_info (peer.peerinfo.PeerInfo) – peer_info of the peer we want to connect to

async disconnect(*peer_id:* ID) → None

get_addrs() → List[Multiaddr]

Returns
all the multiaddr addresses this host is listening to

get_id() → *ID*

Returns
peer_id of host

get_mux() → *Multiselect*

Returns
mux instance of host

get_network() → *INetworkService*

Returns
network instance of host

get_peerstore() → *IPeerStore*

Returns
peerstore of the host (same one as in its network instance)

get_private_key() → *PrivateKey*

Returns
the private key belonging to the peer

get_public_key() → *PublicKey*

Returns
the public key belonging to the peer

multiselect: *Multiselect*

multiselect_client: *MultiselectClient*

async new_stream(*peer_id:* ID, *protocol_ids:* *Sequence[TProtocol]*) → *INetStream*

Parameters

- **peer_id** – peer_id that host is connecting
- **protocol_ids** – available protocol ids to use for stream

Returns
stream: new stream created

peerstore: *IPeerStore*

run(*listen_addrs:* *Sequence[Multiaddr]*) → AsyncIterator[None]

Run the host instance and listen to `listen_addrs`.

Parameters
listen_addrs – a sequence of multiaddrs that we want to listen to

set_stream_handler(*protocol_id: TProtocol*, *stream_handler: Callable[[INetStream], Awaitable[None]]*) → None

Set stream handler for given *protocol_id*

Parameters

- **protocol_id** – protocol id used on stream
- **stream_handler** – a stream handler function

libp2p.host.defaults module

libp2p.host.defaults.**get_default_protocols**(*host:* IHost) → OrderedDict[TProtocol, StreamHandlerFn]

libp2p.host.exceptions module

exception libp2p.host.exceptions.**ConnectionFailure**

Bases: *HostException*

exception libp2p.host.exceptions.**HostException**

Bases: *BaseLibp2pError*

A generic exception in *IHost*.

exception libp2p.host.exceptions.**StreamFailure**

Bases: *HostException*

libp2p.host.host_interface module

class libp2p.host.host_interface.**IHost**

Bases: ABC

abstract async close() → None

abstract async connect(*peer_info:* PeerInfo) → None

Ensure there is a connection between this host and the peer with given peer_info.peer_id. connect will absorb the addresses in peer_info into its internal peerstore. If there is not an active connection, connect will issue a dial, and block until a connection is opened, or an error is returned.

Parameters

peer_info (peer.peerinfo.PeerInfo) – peer_info of the peer we want to connect to

abstract async disconnect(*peer_id:* ID) → None

abstract get_addrs() → List[Multiaddr]

Returns

all the multiaddr addresses this host is listening to

abstract get_id() → *ID*

Returns

peer_id of host

abstract get_mux() → Any

Returns
mux instance of host

abstract get_network() → *INetworkService*

Returns
network instance of host

abstract get_private_key() → *PrivateKey*

Returns
the private key belonging to the peer

abstract get_public_key() → *PublicKey*

Returns
the public key belonging to the peer

abstract async new_stream(*peer_id:* ID, *protocol_ids: Sequence[TProtocol]*) → *INetStream*

Parameters

- **peer_id** – peer_id that host is connecting
- **protocol_ids** – available protocol ids to use for stream

Returns
stream: new stream created

abstract run(*listen_addrs: Sequence[Multiaddr]*) → AsyncContextManager[None]

Run the host instance and listen to `listen_addrs`.

Parameters
listen_addrs – a sequence of multiaddrs that we want to listen to

abstract set_stream_handler(*protocol_id: TProtocol*, *stream_handler: Callable[[*INetStream*], Awaitable[None]]*) → None

Set stream handler for host.

Parameters

- **protocol_id** – protocol id used on stream
- **stream_handler** – a stream handler function

libp2p.host.ping module

async libp2p.host.ping.**handle_ping**(*stream:* INetStream) → None

Respond to incoming ping requests until one side errors or closes the `stream`.

libp2p.host.routed_host module

class libp2p.host.routed_host.**RoutedHost**(*network:* INetworkService, *router:* IPeerRouting)

Bases: *BasicHost*

async **connect**(*peer_info:* PeerInfo) → None

Ensure there is a connection between this host and the peer with given *peer_info.peer_id*. See (basic_host).connect for more information.

RoutedHost's Connect differs in that if the host has no addresses for a given peer, it will use its routing system to try to find some.

Parameters

peer_info (peer.peerinfo.PeerInfo) – peer_info of the peer we want to connect to

Module contents

libp2p.identity package

Subpackages

libp2p.identity.identify package

Subpackages

libp2p.identity.identify.pb package

Submodules

libp2p.identity.identify.pb.identify_pb2 module

Generated protocol buffer code.

class libp2p.identity.identify.pb.identify_pb2.**Identify**

Bases: Message, Message

DESCRIPTOR = <google._upb._message.Descriptor object>

Module contents

Submodules

libp2p.identity.identify.protocol module

libp2p.identity.identify.protocol.**identify_handler_for**(*host:* IHost) → Callable[[*INetStream*], Awaitable[None]]

Module contents

Module contents

libp2p.io package

Submodules

libp2p.io.abc module

class libp2p.io.abc.**Closer**

Bases: ABC

abstract async close() → None

class libp2p.io.abc.**EncryptedMsgReadWriter**

Bases: *MsgReadWriteCloser*, *Encrypter*

Read/write message with encryption/decryption.

class libp2p.io.abc.**Encrypter**

Bases: ABC

abstract decrypt(*data: bytes*) → bytes

abstract encrypt(*data: bytes*) → bytes

class libp2p.io.abc.**MsgReadWriteCloser**

Bases: *MsgReader*, *MsgWriter*, *Closer*

class libp2p.io.abc.**MsgReader**

Bases: ABC

abstract async read_msg() → bytes

class libp2p.io.abc.**MsgWriter**

Bases: ABC

abstract async write_msg(*msg: bytes*) → None

class libp2p.io.abc.**ReadCloser**

Bases: *Reader*, *Closer*

class libp2p.io.abc.**ReadWriteCloser**

Bases: *Reader*, *Writer*, *Closer*

class libp2p.io.abc.**ReadWriter**

Bases: *Reader*, *Writer*

class libp2p.io.abc.**Reader**

Bases: ABC

abstract async read(*n: int | None = None*) → bytes

class libp2p.io.abc.**WriteCloser**

Bases: *Writer*, *Closer*

class libp2p.io.abc.**Writer**

Bases: ABC

abstract async write(*data: bytes*) → None

libp2p.io.exceptions module

exception libp2p.io.exceptions.**DecryptionFailedException**

Bases: *MsgioException*

exception libp2p.io.exceptions.**IOException**

Bases: *BaseLibp2pError*

exception libp2p.io.exceptions.**IncompleteReadError**

Bases: *IOException*

Fewer bytes were read than requested.

exception libp2p.io.exceptions.**MessageTooLarge**

Bases: *MsgioException*

exception libp2p.io.exceptions.**MissingLengthException**

Bases: *MsgioException*

exception libp2p.io.exceptions.**MissingMessageException**

Bases: *MsgioException*

exception libp2p.io.exceptions.**MsgioException**

Bases: *IOException*

libp2p.io.msgio module

msgio is an implementation of *https://github.com/libp2p/go-msgio*.

from that repo: "a simple package to r/w length-delimited slices."

NOTE: currently missing the capability to indicate lengths by "varint" method.

class libp2p.io.msgio.**BaseMsgReadWriter**(*read_write_closer:* ReadWriteCloser)

Bases: *MsgReadWriteCloser*

async close() → None

abstract encode_msg(*msg: bytes*) → bytes

abstract async next_msg_len() → int

async read_msg() → bytes

read_write_closer: *ReadWriteCloser*

size_len_bytes: int

async write_msg(*msg: bytes*) → None

class libp2p.io.msgio.**FixedSizeLenMsgReadWriter**(*read_write_closer:* ReadWriteCloser)

Bases: *BaseMsgReadWriter*

encode_msg(*msg: bytes*) → bytes

async next_msg_len() → int

size_len_bytes: int

class libp2p.io.msgio.**VarIntLengthMsgReadWriter**(*read_write_closer:* ReadWriteCloser)

Bases: *BaseMsgReadWriter*

encode_msg(*msg: bytes*) → bytes

max_msg_size: int

async next_msg_len() → int

libp2p.io.msgio.**encode_msg_with_length**(*msg_bytes: bytes*, *size_len_bytes: int*) → bytes

async libp2p.io.msgio.**read_length**(*reader:* Reader, *size_len_bytes: int*) → int

libp2p.io.trio module

class libp2p.io.trio.**TrioTCPStream**(*stream: SocketStream*)

Bases: *ReadWriteCloser*

async close() → None

async read(*n: int | None = None*) → bytes

read_lock: Lock

stream: SocketStream

async write(*data: bytes*) → None

Raise *RawConnError* if the underlying connection breaks.

write_lock: Lock

libp2p.io.utils module

async libp2p.io.utils.**read_exactly**(*reader:* Reader, *n: int*, *retry_count: int = 100*) → bytes

NOTE: relying on exceptions to break out on erroneous conditions, like EOF

Module contents

libp2p.network package

Subpackages

libp2p.network.connection package

Submodules

libp2p.network.connection.exceptions module

exception libp2p.network.connection.exceptions.**RawConnError**

Bases: *IOException*

libp2p.network.connection.net_connection_interface module

class libp2p.network.connection.net_connection_interface.**INetConn**

Bases: *Closer*

event_started: Event

abstract get_streams() → Tuple[*INetStream*, ...]

muxed_conn: *IMuxedConn*

abstract async new_stream() → *INetStream*

libp2p.network.connection.raw_connection module

class libp2p.network.connection.raw_connection.**RawConnection**(*stream:* ReadWriteCloser, *initiator: bool*)

Bases: *IRawConnection*

async close() → None

is_initiator: bool

async read(*n: int | None = None*) → bytes

Read up to n bytes from the underlying stream. This call is delegated directly to the underlying `self.reader`.

Raise *RawConnError* if the underlying connection breaks

stream: *ReadWriteCloser*

async write(*data: bytes*) → None

Raise *RawConnError* if the underlying connection breaks.

libp2p.network.connection.raw_connection_interface module

class libp2p.network.connection.raw_connection_interface.**IRawConnection**

Bases: *ReadWriteCloser*

A Raw Connection provides a Reader and a Writer.

is_initiator: bool

libp2p.network.connection.swarm_connection module

class libp2p.network.connection.swarm_connection.**SwarmConn**(*muxed_conn:* IMuxedConn, *swarm:* Swarm)

Bases: *INetConn*

async close() → None

event_closed: Event

get_streams() → Tuple[*NetStream*, ...]

property is_closed: bool

muxed_conn: *IMuxedConn*

async new_stream() → *NetStream*

remove_stream(*stream:* NetStream) → None

async start() → None

streams: Set[*NetStream*]

swarm: *Swarm*

Module contents

libp2p.network.stream package

Submodules

libp2p.network.stream.exceptions module

exception libp2p.network.stream.exceptions.**StreamClosed**

Bases: *StreamError*

exception libp2p.network.stream.exceptions.**StreamEOF**

Bases: *StreamError*, EOFError

exception libp2p.network.stream.exceptions.**StreamError**

Bases: *IOException*

exception libp2p.network.stream.exceptions.**StreamReset**

Bases: *StreamError*

libp2p.network.stream.net_stream module

class libp2p.network.stream.net_stream.**NetStream**(*muxed_stream:* IMuxedStream)

Bases: *INetStream*

async close() → None

Close stream.

get_protocol() → TProtocol

Returns
protocol id that stream runs on

muxed_stream: *IMuxedStream*

protocol_id: **TProtocol | None**

async read(*n: int | None = None*) → bytes

Read from stream.

Parameters
n – number of bytes to read

Returns
bytes of input

async reset() → None

Close both ends of the stream.

set_protocol(*protocol_id: TProtocol*) → None

Parameters
protocol_id – protocol id that stream runs on

async write(*data: bytes*) → None

Write to stream.

Returns
number of bytes written

libp2p.network.stream.net_stream_interface module

class libp2p.network.stream.net_stream_interface.**INetStream**

Bases: *ReadWriteCloser*

abstract get_protocol() → TProtocol

Returns
protocol id that stream runs on

muxed_conn: *IMuxedConn*

abstract async reset() → None

Close both ends of the stream.

abstract set_protocol(*protocol_id: TProtocol*) → None

Parameters
protocol_id – protocol id that stream runs on

Module contents

Submodules

libp2p.network.exceptions module

exception libp2p.network.exceptions.**SwarmException**

Bases: *BaseLibp2pError*

libp2p.network.network_interface module

class libp2p.network.network_interface.**INetwork**

Bases: ABC

abstract async close() → None

abstract async close_peer(*peer_id:* ID) → None

connections: Dict[*ID*, *INetConn*]

abstract async dial_peer(*peer_id:* ID) → *INetConn*

dial_peer try to create a connection to peer_id.

Parameters
peer_id – peer if we want to dial

Raises
SwarmException – raised when an error occurs

Returns
muxed connection

abstract get_peer_id() → *ID*

Returns
the peer id

abstract async listen(**multiaddrs: Sequence[Multiaddr]*) → bool

Parameters
multiaddrs – one or many multiaddrs to start listening on

Returns
True if at least one success

listeners: Dict[str, *IListener*]

abstract async new_stream(*peer_id:* ID) → *INetStream*

Parameters

- **peer_id** – peer_id of destination
- **protocol_ids** – available protocol ids to use for stream

Returns
net stream instance

peerstore: *IPeerStore*

abstract register_notifee(*notifee:* INotifee) → None

Parameters
notifee – object implementing Notifee interface

Returns
true if notifee registered successfully, false otherwise

abstract set_stream_handler(*stream_handler: Callable[[*INetStream*], Awaitable[None]]*) → None
Set the stream handler for all incoming streams.

class libp2p.network.network_interface.**INetworkService**
Bases: *INetwork*, ServiceAPI

libp2p.network.notifee_interface module

class libp2p.network.notifee_interface.**INotifee**
Bases: ABC

abstract async closed_stream(*network:* INetwork, *stream:* INetStream) → None

Parameters
- **network** – network the stream was closed on
- **stream** – stream that was closed

abstract async connected(*network:* INetwork, *conn:* INetConn) → None

Parameters
- **network** – network the connection was opened on
- **conn** – connection that was opened

abstract async disconnected(*network:* INetwork, *conn:* INetConn) → None

Parameters
- **network** – network the connection was closed on
- **conn** – connection that was closed

abstract async listen(*network:* INetwork, *multiaddr: Multiaddr*) → None

Parameters
- **network** – network the listener is listening on
- **multiaddr** – multiaddress listener is listening on

abstract async listen_close(*network:* INetwork, *multiaddr: Multiaddr*) → None

Parameters
- **network** – network the connection was opened on
- **multiaddr** – multiaddress listener is no longer listening on

abstract async opened_stream(*network:* INetwork, *stream:* INetStream) → None

Parameters

- **network** – network the stream was opened on
- **stream** – stream that was opened

libp2p.network.swarm module

class libp2p.network.swarm.**Swarm**(*peer_id:* ID, *peerstore:* IPeerStore, *upgrader:* TransportUpgrader, *transport:* ITransport)

Bases: Service, *INetworkService*

async add_conn(*muxed_conn:* IMuxedConn) → *SwarmConn*

Add a *IMuxedConn* to *Swarm* as a *SwarmConn*, notify "connected", and start to monitor the connection for its new streams and disconnection.

async close() → None

async close_peer(*peer_id:* ID) → None

common_stream_handler: Callable[[*INetStream*], Awaitable[None]]

connections: Dict[*ID*, *INetConn*]

async dial_addr(*addr: Multiaddr*, *peer_id:* ID) → *INetConn*

Try to create a connection to peer_id with addr.

Parameters

- **addr** – the address we want to connect with
- **peer_id** – the peer we want to connect to

Raises

SwarmException – raised when an error occurs

Returns

network connection

async dial_peer(*peer_id:* ID) → *INetConn*

Try to create a connection to peer_id.

Parameters

peer_id – peer if we want to dial

Raises

SwarmException – raised when an error occurs

Returns

muxed connection

event_listener_nursery_created: Event

get_peer_id() → *ID*

Returns

the peer id

async listen(**multiaddrs: Multiaddr*) → bool

Parameters
multiaddrs – one or many multiaddrs to start listening on

Returns
true if at least one success

For each multiaddr

- Check if a listener for multiaddr exists already
- If listener already exists, continue
- Otherwise:
  - Capture multiaddr in conn handler
  - Have conn handler delegate to stream handler
  - Call listener listen with the multiaddr
  - Map multiaddr to listener

listener_nursery: Nursery | None

listeners: Dict[str, *IListener*]

async new_stream(*peer_id:* ID) → *INetStream*

Parameters
peer_id – peer_id of destination

Raises
SwarmException – raised when an error occurs

Returns
net stream instance

notifees: List[*INotifee*]

async notify_closed_stream(*stream:* INetStream) → None

async notify_connected(*conn:* INetConn) → None

async notify_disconnected(*conn:* INetConn) → None

async notify_listen(*multiaddr: Multiaddr*) → None

async notify_listen_close(*multiaddr: Multiaddr*) → None

async notify_opened_stream(*stream:* INetStream) → None

peerstore: *IPeerStore*

register_notifee(*notifee:* INotifee) → None

Parameters
notifee – object implementing Notifee interface

Returns
true if notifee registered successfully, false otherwise

remove_conn(*swarm_conn:* SwarmConn) → None

Simply remove the connection from Swarm's records, without closing the connection.

async run() → None

Primary entry point for all service logic.

Note: This method should **not** be directly invoked by user code.

Services may be run using the following approaches.

self_id: *ID*

set_stream_handler(*stream_handler: Callable[[*INetStream*], Awaitable[None]]*) → None

Set the stream handler for all incoming streams.

transport: *ITransport*

upgrader: *TransportUpgrader*

libp2p.network.swarm.**create_default_stream_handler**(*network:* INetworkService) → Callable[[*INetStream*], Awaitable[None]]

Module contents

libp2p.peer package

Submodules

libp2p.peer.addrbook_interface module

class libp2p.peer.addrbook_interface.**IAddrBook**

Bases: ABC

abstract add_addr(*peer_id:* ID, *addr: Multiaddr*, *ttl: int*) → None

Calls add_addrs(peer_id, [addr], ttl)

Parameters

- **peer_id** – the peer to add address for
- **addr** – multiaddress of the peer
- **ttl** – time-to-live for the address (after this time, address is no longer valid)

abstract add_addrs(*peer_id:* ID, *addrs: Sequence[Multiaddr]*, *ttl: int*) → None

Adds addresses for a given peer all with the same time-to-live. If one of the addresses already exists for the peer and has a longer TTL, no operation should take place. If one of the addresses exists with a shorter TTL, extend the TTL to equal param ttl.

Parameters

- **peer_id** – the peer to add address for
- **addr** – multiaddresses of the peer
- **ttl** – time-to-live for the address (after this time, address is no longer valid

abstract addrs(*peer_id:* ID) → List[Multiaddr]

Parameters
peer_id – peer to get addresses of

Returns
all known (and valid) addresses for the given peer

abstract clear_addrs(*peer_id:* ID) → None

Removes all previously stored addresses.

Parameters
peer_id – peer to remove addresses of

abstract peers_with_addrs() → List[*ID*]

Returns
all of the peer IDs stored with addresses

libp2p.peer.id module

class libp2p.peer.id.**ID**(*peer_id_bytes: bytes*)

Bases: object

classmethod from_base58(*b58_encoded_peer_id_str: str*) → *ID*

classmethod from_pubkey(*key:* PublicKey) → *ID*

pretty() → str

to_base58() → str

to_bytes() → bytes

to_string() → str

property xor_id: int

class libp2p.peer.id.**IdentityHash**

Bases: object

digest() → bytes

update(*input: bytes*) → None

libp2p.peer.id.**sha256_digest**(*data: str | bytes*) → bytes

libp2p.peer.peerdata module

class libp2p.peer.peerdata.**PeerData**

Bases: *IPeerData*

add_addrs(*addrs: Sequence[Multiaddr]*) → None

Parameters
addrs – multiaddresses to add

add_privkey(*privkey:* PrivateKey) → None

Parameters
privkey

add_protocols(*protocols: Sequence[str]*) → None

Parameters
protocols – protocols to add

add_pubkey(*pubkey:* PublicKey) → None

Parameters
pubkey

addrs: List[Multiaddr]

clear_addrs() → None
Clear all addresses.

get_addrs() → List[Multiaddr]

Returns
all multiaddresses

get_metadata(*key: str*) → Any

Parameters
key – key in KV pair

Returns
val for key

Raises
PeerDataError – key not found

get_privkey() → *PrivateKey*

Returns
private key of the peer

Raises
PeerDataError – if private key not found

get_protocols() → List[str]

Returns
all protocols associated with given peer

get_pubkey() → *PublicKey*

Returns
public key of the peer

Raises
PeerDataError – if public key not found

metadata: Dict[Any, Any]

privkey: *PrivateKey*

protocols: List[str]

pubkey: *PublicKey*

put_metadata(*key: str*, *val: Any*) → None

Parameters

- **key** – key in KV pair
- **val** – val to associate with key

set_protocols(*protocols: Sequence[str]*) → None

Parameters
protocols – protocols to set

exception libp2p.peer.peerdata.**PeerDataError**

Bases: KeyError

Raised when a key is not found in peer metadata.

libp2p.peer.peerdata_interface module

class libp2p.peer.peerdata_interface.**IPeerData**

Bases: ABC

abstract add_addrs(*addrs: Sequence[Multiaddr]*) → None

Parameters
addrs – multiaddresses to add

abstract add_privkey(*privkey:* PrivateKey) → None

Parameters
privkey

abstract add_protocols(*protocols: Sequence[str]*) → None

Parameters
protocols – protocols to add

abstract add_pubkey(*pubkey:* PublicKey) → None

Parameters
pubkey

abstract clear_addrs() → None

Clear all addresses.

abstract get_addrs() → List[Multiaddr]

Returns
all multiaddresses

abstract get_metadata(*key: str*) → *IPeerMetadata*

Parameters
key – key in KV pair

Returns
val for key

Raises
PeerDataError – key not found

abstract get_privkey() → *PrivateKey*

Returns
private key of the peer

Raises
PeerDataError – if private key not found

abstract get_protocols() → List[str]

Returns
all protocols associated with given peer

abstract get_pubkey() → *PublicKey*

Returns
public key of the peer

Raises
PeerDataError – if public key not found

abstract put_metadata(*key: str*, *val: Any*) → None

Parameters

- **key** – key in KV pair
- **val** – val to associate with key

abstract set_protocols(*protocols: Sequence[str]*) → None

Parameters
protocols – protocols to set

libp2p.peer.peerinfo module

exception libp2p.peer.peerinfo.**InvalidAddrError**

Bases: ValueError

class libp2p.peer.peerinfo.**PeerInfo**(*peer_id: ID*, *addrs: Sequence[Multiaddr]*)

Bases: object

addrs: List[Multiaddr]

peer_id: *ID*

libp2p.peer.peerinfo.**info_from_p2p_addr**(*addr: Multiaddr*) → *PeerInfo*

libp2p.peer.peermetadata_interface module

class libp2p.peer.peermetadata_interface.**IPeerMetadata**

Bases: ABC

abstract get(*peer_id:* ID, *key: str*) → Any

Parameters

- **peer_id** – peer ID to lookup key for
- **key** – key to look up

Returns
value at key for given peer

Raises
Exception – peer ID not found

abstract put(*peer_id:* ID, *key: str*, *val: Any*) → None

Parameters

- **peer_id** – peer ID to lookup key for
- **key** – key to associate with peer
- **val** – value to associated with key

Raises
Exception – unsuccessful put

libp2p.peer.peerstore module

class libp2p.peer.peerstore.**PeerStore**

Bases: *IPeerStore*

add_addr(*peer_id:* ID, *addr: Multiaddr*, *ttl: int*) → None

Parameters

- **peer_id** – peer ID to add address for
- **addr**
- **ttl** – time-to-live for the this record

add_addrs(*peer_id:* ID, *addrs: Sequence[Multiaddr]*, *ttl: int*) → None

Parameters

- **peer_id** – peer ID to add address for
- **addrs**
- **ttl** – time-to-live for the this record

add_key_pair(*peer_id:* ID, *key_pair:* KeyPair) → None

Parameters

- **peer_id** – peer ID to add private key for
- **key_pair**

add_privkey(*peer_id:* ID, *privkey:* PrivateKey) → None

Parameters

- **peer_id** – peer ID to add private key for
- **privkey**

Raises
PeerStoreError – if peer ID or peer privkey not found

add_protocols(*peer_id:* ID, *protocols:* *Sequence[str]*) → None

Parameters

- **peer_id** – peer ID to add protocols for
- **protocols** – protocols to add

add_pubkey(*peer_id:* ID, *pubkey:* PublicKey) → None

Parameters

- **peer_id** – peer ID to add public key for
- **pubkey**

Raises
PeerStoreError – if peer ID and pubkey does not match

addrs(*peer_id:* ID) → List[Multiaddr]

Parameters
peer_id – peer ID to get addrs for

Returns
list of addrs

Raises
PeerStoreError – if peer ID not found

clear_addrs(*peer_id:* ID) → None

Parameters
peer_id – peer ID to clear addrs for

get(*peer_id:* ID, *key:* *str*) → Any

Parameters

- **peer_id** – peer ID to get peer data for
- **key** – the key to search value for

Returns
value corresponding to the key

Raises
PeerStoreError – if peer ID or value not found

get_protocols(*peer_id:* ID) → List[str]

Parameters
peer_id – peer ID to get protocols for

Returns
protocols (as list of strings)

Raises
PeerStoreError – if peer ID not found

peer_data_map: Dict[*ID*, *PeerData*]

peer_ids() → List[*ID*]

Returns
all of the peer IDs stored in peer store

peer_info(*peer_id:* ID) → *PeerInfo*

Parameters
peer_id – peer ID to get info for

Returns
peer info object

peers_with_addrs() → List[*ID*]

Returns
all of the peer IDs which has addrs stored in peer store

privkey(*peer_id:* ID) → *PrivateKey*

Parameters
peer_id – peer ID to get private key for

Returns
private key of the peer

Raises
PeerStoreError – if peer ID or peer privkey not found

pubkey(*peer_id:* ID) → *PublicKey*

Parameters
peer_id – peer ID to get public key for

Returns
public key of the peer

Raises
PeerStoreError – if peer ID or peer pubkey not found

put(*peer_id:* ID, *key: str*, *val: Any*) → None

Parameters

- **peer_id** – peer ID to put peer data for
- **key**
- **value**

set_protocols(*peer_id:* ID, *protocols: Sequence[str]*) → None

Parameters

- **peer_id** – peer ID to set protocols for
- **protocols** – protocols to set

exception libp2p.peer.peerstore.**PeerStoreError**

Bases: KeyError

Raised when peer ID is not found in peer store.

libp2p.peer.peerstore_interface module

class libp2p.peer.peerstore_interface.**IPeerStore**

Bases: *IAddrBook*, *IPeerMetadata*

abstract **add_addr**(*peer_id:* ID, *addr: Multiaddr*, *ttl: int*) → None

Parameters

- **peer_id** – peer ID to add address for
- **addr**
- **ttl** – time-to-live for the this record

abstract **add_addrs**(*peer_id:* ID, *addrs: Sequence[Multiaddr]*, *ttl: int*) → None

Parameters

- **peer_id** – peer ID to add address for
- **addrs**
- **ttl** – time-to-live for the this record

abstract **add_key_pair**(*peer_id:* ID, *key_pair:* KeyPair) → None

Parameters

- **peer_id** – peer ID to add private key for
- **key_pair**

Raises

PeerStoreError – if peer ID already has pubkey or privkey set

abstract **add_privkey**(*peer_id:* ID, *privkey:* PrivateKey) → None

Parameters

- **peer_id** – peer ID to add private key for
- **privkey**

Raises

PeerStoreError – if peer ID already has privkey set

abstract **add_protocols**(*peer_id:* ID, *protocols: Sequence[str]*) → None

Parameters

- **peer_id** – peer ID to add protocols for
- **protocols** – protocols to add

abstract **add_pubkey**(*peer_id:* ID, *pubkey:* PublicKey) → None

Parameters

- **peer_id** – peer ID to add public key for

- **pubkey**

Raises
PeerStoreError – if peer ID already has pubkey set

abstract addrs(*peer_id:* ID) → List[Multiaddr]

Parameters
peer_id – peer ID to get addrs for

Returns
list of addrs

abstract clear_addrs(*peer_id:* ID) → None

Parameters
peer_id – peer ID to clear addrs for

abstract get(*peer_id:* ID, *key: str*) → Any

Parameters

- **peer_id** – peer ID to get peer data for
- **key** – the key to search value for

Returns
value corresponding to the key

Raises
PeerStoreError – if peer ID or value not found

abstract get_protocols(*peer_id:* ID) → List[str]

Parameters
peer_id – peer ID to get protocols for

Returns
protocols (as list of strings)

Raises
PeerStoreError – if peer ID not found

abstract peer_ids() → List[*ID*]

Returns
all of the peer IDs stored in peer store

abstract peer_info(*peer_id:* ID) → *PeerInfo*

Parameters
peer_id – peer ID to get info for

Returns
peer info object

abstract peers_with_addrs() → List[*ID*]

Returns
all of the peer IDs which has addrs stored in peer store

abstract privkey(*peer_id:* ID) → *PrivateKey*

Parameters
peer_id – peer ID to get private key for

Returns
private key of the peer

Raises
PeerStoreError – if peer ID not found

abstract pubkey(*peer_id:* ID) → *PublicKey*

Parameters
peer_id – peer ID to get public key for

Returns
public key of the peer

Raises
PeerStoreError – if peer ID not found

abstract put(*peer_id:* ID, *key: str*, *val: Any*) → None

Parameters

- **peer_id** – peer ID to put peer data for
- **key**
- **value**

abstract set_protocols(*peer_id:* ID, *protocols: Sequence[str]*) → None

Parameters

- **peer_id** – peer ID to set protocols for
- **protocols** – protocols to set

Module contents

libp2p.protocol_muxer package

Submodules

libp2p.protocol_muxer.exceptions module

exception libp2p.protocol_muxer.exceptions.**MultiselectClientError**

Bases: *BaseLibp2pError*

Raised when an error occurs in protocol selection process.

exception libp2p.protocol_muxer.exceptions.**MultiselectCommunicatorError**

Bases: *BaseLibp2pError*

Raised when an error occurs during read/write via communicator.

exception libp2p.protocol_muxer.exceptions.**MultiselectError**

Bases: *BaseLibp2pError*

Raised when an error occurs in multiselect process.

libp2p.protocol_muxer.multiselect module

class libp2p.protocol_muxer.multiselect.**Multiselect**(*default_handlers: Dict[TProtocol, Callable[[INetStream], Awaitable[None]]] = None*)

Bases: *IMultiselectMuxer*

Multiselect module that is responsible for responding to a multiselect client and deciding on a specific protocol and handler pair to use for communication.

add_handler(*protocol: TProtocol*, *handler: Callable[[INetStream], Awaitable[None]]*) → None

Store the handler with the given protocol.

Parameters

- **protocol** – protocol name
- **handler** – handler function

handlers: `Dict[TProtocol, Callable[[INetStream], Awaitable[None]]]`

async handshake(*communicator:* IMultiselectCommunicator) → None

Perform handshake to agree on multiselect protocol.

Parameters
communicator – communicator to use

Raises
MultiselectError – raised when handshake failed

async negotiate(*communicator:* IMultiselectCommunicator) → Tuple[TProtocol, Callable[[*INetStream*], Awaitable[None]]]

Negotiate performs protocol selection.

Parameters
stream – stream to negotiate on

Returns
selected protocol name, handler function

Raises
MultiselectError – raised when negotiation failed

libp2p.protocol_muxer.multiselect.**is_valid_handshake**(*handshake_contents: str*) → bool

Determine if handshake is valid and should be confirmed.

Parameters
handshake_contents – contents of handshake message

Returns
true if handshake is complete, false otherwise

libp2p.protocol_muxer.multiselect_client module

class libp2p.protocol_muxer.multiselect_client.**MultiselectClient**

Bases: *IMultiselectClient*

Client for communicating with receiver's multiselect module in order to select a protocol id to communicate over.

async handshake(*communicator:* IMultiselectCommunicator) → None

Ensure that the client and multiselect are both using the same multiselect protocol.

Parameters
stream – stream to communicate with multiselect over

Raises
MultiselectClientError – raised when handshake failed

async select_one_of(*protocols: Sequence[TProtocol]*, *communicator:* IMultiselectCommunicator) → TProtocol

For each protocol, send message to multiselect selecting protocol and fail if multiselect does not return same protocol. Returns first protocol that multiselect agrees on (i.e. that multiselect selects)

Parameters

- **protocol** – protocol to select
- **stream** – stream to communicate with multiselect over

Returns
selected protocol

Raises
MultiselectClientError – raised when protocol negotiation failed

async try_select(*communicator:* IMultiselectCommunicator, *protocol: TProtocol*) → TProtocol

Try to select the given protocol or raise exception if fails.

Parameters

- **communicator** – communicator to use to communicate with counterparty
- **protocol** – protocol to select

Raises
MultiselectClientError – raised when protocol negotiation failed

Returns
selected protocol

libp2p.protocol_muxer.multiselect_client.**is_valid_handshake**(*handshake_contents: str*) → bool

Determine if handshake is valid and should be confirmed.

Parameters
handshake_contents – contents of handshake message

Returns
true if handshake is complete, false otherwise

libp2p.protocol_muxer.multiselect_client_interface module

class libp2p.protocol_muxer.multiselect_client_interface.**IMultiselectClient**

Bases: ABC

Client for communicating with receiver's multiselect module in order to select a protocol id to communicate over.

abstract async handshake(*communicator:* IMultiselectCommunicator) → None

Ensure that the client and multiselect are both using the same multiselect protocol.

Parameters
stream – stream to communicate with multiselect over

Raises
Exception – multiselect protocol ID mismatch

abstract async select_one_of(*protocols: Sequence[TProtocol]*, *communicator:* IMultiselectCommunicator) → TProtocol

For each protocol, send message to multiselect selecting protocol and fail if multiselect does not return same protocol. Returns first protocol that multiselect agrees on (i.e. that multiselect selects)

Parameters

- **protocol** – protocol to select
- **stream** – stream to communicate with multiselect over

Returns
selected protocol

abstract async try_select(*communicator:* IMultiselectCommunicator, *protocol: TProtocol*) → TProtocol

Try to select the given protocol or raise exception if fails.

Parameters

- **communicator** – communicator to use to communicate with counterparty
- **protocol** – protocol to select

Raises
Exception – error in protocol selection

Returns
selected protocol

libp2p.protocol_muxer.multiselect_communicator module

class libp2p.protocol_muxer.multiselect_communicator.**MultiselectCommunicator**(*read_writer:* ReadWriteCloser)

Bases: *IMultiselectCommunicator*

async read() → str

Raises
MultiselectCommunicatorError – raised when failed to read from underlying reader

read_writer: *ReadWriteCloser*

async write(*msg_str: str*) → None

Raises
MultiselectCommunicatorError – raised when failed to write to underlying reader

libp2p.protocol_muxer.multiselect_communicator_interface module

class libp2p.protocol_muxer.multiselect_communicator_interface.**IMultiselectCommunicator**

Bases: ABC

Communicator helper class that ensures both the client and multistream module will follow the same multistream protocol, which is necessary for them to work.

abstract async read() → str

Reads message from stream until EOF.

abstract async write(*msg_str: str*) → None

Write message to stream.

Parameters
msg_str – message to write

libp2p.protocol_muxer.multiselect_muxer_interface module

class libp2p.protocol_muxer.multiselect_muxer_interface.**IMultiselectMuxer**

Bases: ABC

Multiselect module that is responsible for responding to a multiselect client and deciding on a specific protocol and handler pair to use for communication.

abstract add_handler(*protocol: TProtocol*, *handler: Callable[[INetStream], Awaitable[None]]*) → None

Store the handler with the given protocol.

Parameters

- **protocol** – protocol name
- **handler** – handler function

get_protocols() → Tuple[TProtocol, ...]

handlers: Dict[TProtocol, Callable[[*INetStream*], Awaitable[None]]]

abstract async negotiate(*communicator:* IMultiselectCommunicator) → Tuple[TProtocol, Callable[[*INetStream*], Awaitable[None]]]

Negotiate performs protocol selection.

Parameters
stream – stream to negotiate on

Returns
selected protocol name, handler function

Raises
Exception – negotiation failed exception

Module contents

libp2p.pubsub package

Subpackages

libp2p.pubsub.pb package

Submodules

libp2p.pubsub.pb.rpc_pb2 module

Generated protocol buffer code.

class libp2p.pubsub.pb.rpc_pb2.**ControlGraft**

Bases: Message, Message

DESCRIPTOR = <google._upb._message.Descriptor object>

class libp2p.pubsub.pb.rpc_pb2.**ControlIHave**

Bases: Message, Message

DESCRIPTOR = <google._upb._message.Descriptor object>

class libp2p.pubsub.pb.rpc_pb2.**ControlIWant**

Bases: Message, Message

DESCRIPTOR = <google._upb._message.Descriptor object>

class libp2p.pubsub.pb.rpc_pb2.**ControlMessage**

Bases: Message, Message

DESCRIPTOR = <google._upb._message.Descriptor object>

class libp2p.pubsub.pb.rpc_pb2.**ControlPrune**

Bases: Message, Message

DESCRIPTOR = <google._upb._message.Descriptor object>

class libp2p.pubsub.pb.rpc_pb2.**Message**

Bases: Message, Message

DESCRIPTOR = <google._upb._message.Descriptor object>

class libp2p.pubsub.pb.rpc_pb2.**RPC**

Bases: Message, Message

DESCRIPTOR = <google._upb._message.Descriptor object>

class SubOpts

Bases: Message, Message

DESCRIPTOR = <google._upb._message.Descriptor object>

class libp2p.pubsub.pb.rpc_pb2.**TopicDescriptor**

Bases: Message, Message

class AuthOpts

Bases: Message, Message

DESCRIPTOR = <google._upb._message.Descriptor object>

DESCRIPTOR = <google._upb._message.Descriptor object>

class EncOpts

Bases: Message, Message

DESCRIPTOR = <google._upb._message.Descriptor object>

Module contents

Submodules

libp2p.pubsub.abc module

class libp2p.pubsub.abc.IPubsub

Bases: ServiceAPI

abstract property my_id: *ID*

abstract property protocols: Tuple[TProtocol, ...]

abstract async publish(*topic_id: str*, *data: bytes*) → None

abstract remove_topic_validator(*topic: str*) → None

abstract set_topic_validator(*topic: str*, *validator: Callable[[ID, Message], bool] | Callable[[ID, Message], Awaitable[bool]]*, *is_async_validator: bool*) → None

abstract async subscribe(*topic_id: str*) → *ISubscriptionAPI*

abstract property topic_ids: KeysView[str]

abstract async unsubscribe(*topic_id: str*) → None

abstract async wait_until_ready() → None

class libp2p.pubsub.abc.IPubsubRouter

Bases: ABC

abstract add_peer(*peer_id: ID*, *protocol_id: TProtocol*) → None

Notifies the router that a new peer has been connected.

Parameters
peer_id – id of peer to add

abstract attach(*pubsub: Pubsub*) → None

Attach is invoked by the PubSub constructor to attach the router to a freshly initialized PubSub instance.

Parameters
pubsub – pubsub instance to attach to

abstract get_protocols() → List[TProtocol]

Returns
the list of protocols supported by the router

abstract async handle_rpc(*rpc:* RPC, *sender_peer_id:* ID) → None

Invoked to process control messages in the RPC envelope. It is invoked after subscriptions and payload messages have been processed TODO: Check if this interface is ok. It's not the exact same as the go code, but the go code is really confusing with the msg origin, they specify *rpc.from* even when the rpc shouldn't have a from :param rpc: rpc message

abstract async join(*topic: str*) → None

Join notifies the router that we want to receive and forward messages in a topic. It is invoked after the subscription announcement.

Parameters
topic – topic to join

abstract async leave(*topic: str*) → None

Leave notifies the router that we are no longer interested in a topic. It is invoked after the unsubscription announcement.

Parameters
topic – topic to leave

abstract async publish(*msg_forwarder:* ID, *pubsub_msg:* Message) → None

Invoked to forward a new message that has been validated.

Parameters

- **msg_forwarder** – peer_id of message sender
- **pubsub_msg** – pubsub message to forward

abstract remove_peer(*peer_id:* ID) → None

Notifies the router that a peer has been disconnected.

Parameters
peer_id – id of peer to remove

class libp2p.pubsub.abc.**ISubscriptionAPI**

Bases: AsyncContextManager[*ISubscriptionAPI*], AsyncIterable[*Message*]

abstract async get() → *Message*

abstract async unsubscribe() → None

libp2p.pubsub.exceptions module

exception libp2p.pubsub.exceptions.**NoPubsubAttached**

Bases: *PubsubRouterError*

exception libp2p.pubsub.exceptions.**PubsubRouterError**

Bases: *BaseLibp2pError*

libp2p.pubsub.floodsub module

class libp2p.pubsub.floodsub.**FloodSub**(*protocols: Sequence[TProtocol]*)

Bases: *IPubsubRouter*

add_peer(*peer_id:* ID, *protocol_id: TProtocol*) → None

Notifies the router that a new peer has been connected.

Parameters
peer_id – id of peer to add

attach(*pubsub:* Pubsub) → None

Attach is invoked by the PubSub constructor to attach the router to a freshly initialized PubSub instance.

Parameters
pubsub – pubsub instance to attach to

get_protocols() → List[TProtocol]

Returns
the list of protocols supported by the router

async handle_rpc(*rpc:* RPC, *sender_peer_id:* ID) → None

Invoked to process control messages in the RPC envelope. It is invoked after subscriptions and payload messages have been processed.

Parameters
rpc – rpc message

async join(*topic: str*) → None

Join notifies the router that we want to receive and forward messages in a topic. It is invoked after the subscription announcement.

Parameters
topic – topic to join

async leave(*topic: str*) → None

Leave notifies the router that we are no longer interested in a topic. It is invoked after the unsubscription announcement.

Parameters
topic – topic to leave

protocols: List[TProtocol]

async publish(*msg_forwarder:* ID, *pubsub_msg:* Message) → None

Invoked to forward a new message that has been validated. This is where the "flooding" part of floodsub happens.

With flooding, routing is almost trivial: for each incoming message, forward to all known peers in the topic. There is a bit of logic, as the router maintains a timed cache of previous messages, so that seen messages are not further forwarded. It also never forwards a message back to the source or the peer that forwarded the message. :param msg_forwarder: peer ID of the peer who forwards the message to us :param pubsub_msg: pubsub message in protobuf.

pubsub: *Pubsub*

remove_peer(*peer_id:* ID) → None

Notifies the router that a peer has been disconnected.

Parameters
peer_id – id of peer to remove

libp2p.pubsub.gossipsub module

class libp2p.pubsub.gossipsub.**GossipSub**(*protocols: Sequence[TProtocol]*, *degree: int*, *degree_low: int*, *degree_high: int*, *time_to_live: int*, *gossip_window: int = 3*, *gossip_history: int = 5*, *heartbeat_initial_delay: float = 0.1*, *heartbeat_interval: int = 120*)

Bases: *IPubsubRouter*, Service

add_peer(*peer_id:* ID, *protocol_id: TProtocol*) → None

Notifies the router that a new peer has been connected.

Parameters

- **peer_id** – id of peer to add
- **protocol_id** – router protocol the peer speaks, e.g., floodsub, gossipsub

attach(*pubsub:* Pubsub) → None

Attach is invoked by the PubSub constructor to attach the router to a freshly initialized PubSub instance.

Parameters
pubsub – pubsub instance to attach to

degree: int

degree_high: int

degree_low: int

async emit_control_message(*control_msg:* ControlMessage, *to_peer:* ID) → None

async emit_graft(*topic: str*, *to_peer:* ID) → None

Emit graft message, sent to to_peer, for topic.

async emit_ihave(*topic: str*, *msg_ids: Any*, *to_peer:* ID) → None

Emit ihave message, sent to to_peer, for topic and msg_ids.

async emit_iwant(*msg_ids: Any*, *to_peer:* ID) → None

Emit iwant message, sent to to_peer, for msg_ids.

async emit_prune(*topic: str*, *to_peer:* ID) → None

Emit graft message, sent to to_peer, for topic.

fanout: Dict[str, Set[*ID*]]

fanout_heartbeat() → None

get_protocols() → List[TProtocol]

Returns
the list of protocols supported by the router

gossip_heartbeat() → DefaultDict[*ID*, Dict[str, List[str]]]

async handle_graft(*graft_msg:* ControlGraft, *sender_peer_id:* ID) → None

async handle_ihave(*ihave_msg:* ControlIHave, *sender_peer_id:* ID) → None

Checks the seen set and requests unknown messages with an IWANT message.

async handle_iwant(*iwant_msg:* ControlIWant, *sender_peer_id:* ID) → None

Forwards all request messages that are present in mcache to the requesting peer.

async handle_prune(*prune_msg:* ControlPrune, *sender_peer_id:* ID) → None

async handle_rpc(*rpc:* RPC, *sender_peer_id:* ID) → None

Invoked to process control messages in the RPC envelope. It is invoked after subscriptions and payload messages have been processed.

Parameters

- **rpc** – RPC message
- **sender_peer_id** – id of the peer who sent the message

async heartbeat() → None

Call individual heartbeats.

Note: the heartbeats are called with awaits because each heartbeat depends on the state changes in the preceding heartbeat

heartbeat_initial_delay: *float*

heartbeat_interval: *int*

async join(*topic: str*) → None

Join notifies the router that we want to receive and forward messages in a topic. It is invoked after the subscription announcement.

Parameters

topic – topic to join

async leave(*topic: str*) → None

Leave notifies the router that we are no longer interested in a topic. It is invoked after the unsubscription announcement.

Parameters

topic – topic to leave

mcache: *MessageCache*

mesh: *Dict[str, Set[ID]]*

mesh_heartbeat() → Tuple[DefaultDict[*ID*, List[str]], DefaultDict[*ID*, List[str]]]

pack_control_msgs(*ihave_msgs: List[*ControlIHave*]*, *graft_msgs: List[*ControlGraft*]*, *prune_msgs: List[*ControlPrune*]*) → *ControlMessage*

peer_protocol: *Dict[ID, TProtocol]*

protocols: *List[TProtocol]*

async publish(*msg_forwarder:* ID, *pubsub_msg:* Message) → None

Invoked to forward a new message that has been validated.

pubsub: *Pubsub*

remove_peer(*peer_id:* ID) → None

Notifies the router that a peer has been disconnected.

Parameters
peer_id – id of peer to remove

async run() → None

Primary entry point for all service logic.

Note: This method should **not** be directly invoked by user code.

Services may be run using the following approaches.

static select_from_minus(*num_to_select: int*, *pool: Iterable[Any]*, *minus: Iterable[Any]*) → List[Any]

Select at most num_to_select subset of elements from the set (pool - minus) randomly. :param num_to_select: number of elements to randomly select :param pool: list of items to select from (excluding elements in minus) :param minus: elements to be excluded from selection pool :return: list of selected elements

time_to_live: int

libp2p.pubsub.mcache module

class libp2p.pubsub.mcache.**CacheEntry**(*mid: Tuple[bytes, bytes]*, *topics: Sequence[str]*)

Bases: object

mid: Tuple[bytes, bytes]

topics: List[str]

A logical representation of an entry in the mcache's _history_.

class libp2p.pubsub.mcache.**MessageCache**(*window_size: int*, *history_size: int*)

Bases: object

get(*mid: Tuple[bytes, bytes]*) → *Message* | None

Get a message from the mcache.

Parameters
mid – (seqno, from_id) of the message to get.

Returns
The rpc message associated with this mid

history: List[List[*CacheEntry*]]

history_size: int

msgs: Dict[Tuple[bytes, bytes], *Message*]

put(*msg:* Message) → None

Put a message into the mcache.

Parameters
msg – The rpc message to put in. Should contain seqno and from_id

shift() → None

Shift the window over by 1 position, dropping the last element of the history.

window(*topic: str*) → List[Tuple[bytes, bytes]]

Get the window for this topic.

Parameters
topic – Topic whose message ids we desire.

Returns
List of mids in the current window.

window_size: int

libp2p.pubsub.pubsub module

class libp2p.pubsub.pubsub.**Pubsub**(*host: ~libp2p.host.host_interface.IHost, router: IPubsubRouter, cache_size: int = None, strict_signing: bool = True, msg_id_constructor: ~typing.Callable[[~libp2p.pubsub.pb.rpc_pb2.Message], bytes] = <function get_peer_and_seqno_msg_id>*)

Bases: Service, *IPubsub*

async continuously_read_stream(*stream:* INetStream) → None

Read from input stream in an infinite loop. Process messages from other nodes.

Parameters
stream – stream to continously read from

counter: int

dead_peer_receive_channel: trio.MemoryReceiveChannel[*ID*]

event_handle_dead_peer_queue_started: Event

event_handle_peer_queue_started: Event

get_hello_packet() → *RPC*

Generate subscription message with all topics we are subscribed to only send hello packet if we have subscribed topics.

get_msg_validators(*msg:* Message) → Tuple[*TopicValidator*, ...]

Get all validators corresponding to the topics in the message.

Parameters
msg – the message published to the topic

async handle_dead_peer_queue() → None

Continuously read from dead peer channel and close the stream between that peer and remove peer info from pubsub and pubsub router.

async handle_peer_queue() → None

Continuously read from peer queue and each time a new peer is found, open a stream to the peer using a supported pubsub protocol pubsub protocols we support.

handle_subscription(*origin_id:* ID, *sub_message:* SubOpts) → None

Handle an incoming subscription message from a peer. Update internal mapping to mark the peer as subscribed or unsubscribed to topics as defined in the subscription message.

Parameters

- **origin_id** – id of the peer who subscribe to the message
- **sub_message** – RPC.SubOpts

host: *IHost*

async message_all_peers(*raw_msg: bytes*) → None

Broadcast a message to peers.

Parameters
raw_msg – raw contents of the message to broadcast

property my_id: *ID*

notify_subscriptions(*publish_message:* Message) → None

Put incoming message from a peer onto my blocking queue.

Parameters
publish_message – RPC.Message format

peer_receive_channel: **trio.MemoryReceiveChannel[***ID***]**

peer_topics: **Dict[str, Set[***ID***]]**

peers: **Dict[***ID***, *INetStream*]**

property protocols: **Tuple[TProtocol, ...]**

async publish(*topic_id: str*, *data: bytes*) → None

Publish data to a topic.

Parameters

- **topic_id** – topic which we are going to publish the data to
- **data** – data which we are publishing

async push_msg(*msg_forwarder:* ID, *msg:* Message) → None

Push a pubsub message to others.

Parameters

- **msg_forwarder** – the peer who forward us the message.
- **msg** – the message we are going to push out.

remove_topic_validator(*topic: str*) → None

Remove the validator from the given topic.

Parameters
topic – the topic to remove validator from

router: *IPubsubRouter*

async run() → None

Primary entry point for all service logic.

Note: This method should **not** be directly invoked by user code.

Services may be run using the following approaches.

seen_messages: LRU

set_topic_validator(*topic: str*, *validator: Callable[[*ID, Message*], bool] | Callable[[*ID, Message*], Awaitable[bool]]*, *is_async_validator: bool*) → None

Register a validator under the given topic. One topic can only have one validtor.

Parameters

- **topic** – the topic to register validator under
- **validator** – the validator used to validate messages published to the topic
- **is_async_validator** – indicate if the validator is an asynchronous validator

sign_key: *PrivateKey*

async stream_handler(*stream:* INetStream) → None

Stream handler for pubsub. Gets invoked whenever a new stream is created on one of the supported pubsub protocols.

Parameters

stream – newly created stream

strict_signing: bool

async subscribe(*topic_id: str*) → *ISubscriptionAPI*

Subscribe ourself to a topic.

Parameters

topic_id – topic_id to subscribe to

subscribed_topics_receive: Dict[str, *TrioSubscriptionAPI*]

subscribed_topics_send: Dict[str, trio.MemorySendChannel[*rpc_pb2.Message*]]

property topic_ids: KeysView[str]

topic_validators: Dict[str, *TopicValidator*]

async unsubscribe(*topic_id: str*) → None

Unsubscribe ourself from a topic.

Parameters

topic_id – topic_id to unsubscribe from

async validate_msg(*msg_forwarder:* ID, *msg:* Message) → None

Validate the received message.

Parameters

- **msg_forwarder** – the peer who forward us the message.
- **msg** – the message.

async wait_until_ready() → None

class libp2p.pubsub.pubsub.**TopicValidator**(*validator*, *is_async*)

Bases: NamedTuple

is_async: bool

Alias for field number 1

validator: Callable[[*ID*, *Message*], bool] | Callable[[*ID*, *Message*], Awaitable[bool]]

Alias for field number 0

libp2p.pubsub.pubsub.**get_content_addressed_msg_id**(*msg:* Message) → bytes

libp2p.pubsub.pubsub.**get_peer_and_seqno_msg_id**(*msg:* Message) → bytes

libp2p.pubsub.pubsub_notifee module

class libp2p.pubsub.pubsub_notifee.**PubsubNotifee**(*initiator_peers_queue: trio.MemorySendChannel[ID]*, *dead_peers_queue: trio.MemorySendChannel[ID]*)

Bases: *INotifee*

async closed_stream(*network:* INetwork, *stream:* INetStream) → None

Parameters

- **network** – network the stream was closed on
- **stream** – stream that was closed

async connected(*network:* INetwork, *conn:* INetConn) → None

Add peer_id to initiator_peers_queue, so that this peer_id can be used to create a stream and we only want to have one pubsub stream with each peer.

Parameters

- **network** – network the connection was opened on
- **conn** – connection that was opened

dead_peers_queue: trio.MemorySendChannel[*ID*]

async disconnected(*network:* INetwork, *conn:* INetConn) → None

Add peer_id to dead_peers_queue, so that pubsub and its router can remove this peer_id and close the stream inbetween.

Parameters

- **network** – network the connection was opened on
- **conn** – connection that was opened

initiator_peers_queue: trio.MemorySendChannel[*ID*]

async listen(*network:* INetwork, *multiaddr: Multiaddr*) → None

Parameters

- **network** – network the listener is listening on
- **multiaddr** – multiaddress listener is listening on

async listen_close(*network:* INetwork, *multiaddr: Multiaddr*) → None

Parameters

- **network** – network the connection was opened on
- **multiaddr** – multiaddress listener is no longer listening on

async opened_stream(*network:* INetwork, *stream:* INetStream) → None

Parameters

- **network** – network the stream was opened on
- **stream** – stream that was opened

libp2p.pubsub.subscription module

class libp2p.pubsub.subscription.**BaseSubscriptionAPI**

Bases: *ISubscriptionAPI*

class libp2p.pubsub.subscription.**TrioSubscriptionAPI**(*receive_channel: trio.MemoryReceiveChannel[*rpc_pb2.Message*]*, *unsubscribe_fn: Callable[[], Awaitable[None]]*)

Bases: *BaseSubscriptionAPI*

async get() → *Message*

receive_channel: trio.MemoryReceiveChannel[*rpc_pb2.Message*]

async unsubscribe() → None

unsubscribe_fn: Callable[[], Awaitable[None]]

libp2p.pubsub.typing module

libp2p.pubsub.validators module

libp2p.pubsub.validators.**signature_validator**(*msg:* Message) → bool

Verify the message against the given public key.

Parameters

- **pubkey** – the public key which signs the message.
- **msg** – the message signed.

Module contents

libp2p.routing package

Submodules

libp2p.routing.interfaces module

class libp2p.routing.interfaces.**IContentRouting**

Bases: ABC

abstract find_provider_iter(*cid: bytes*, *count: int*) → Iterable[*PeerInfo*]

Search for peers who are able to provide a given key returns an iterator of peer.PeerInfo.

abstract provide(*cid: bytes*, *announce: bool = True*) → None

Provide adds the given cid to the content routing system.

If announce is True, it also announces it, otherwise it is just kept in the local accounting of which objects are being provided.

class libp2p.routing.interfaces.**IPeerRouting**

Bases: ABC

abstract async find_peer(*peer_id:* ID) → *PeerInfo*

Find specific Peer FindPeer searches for a peer with given peer_id, returns a peer.PeerInfo with relevant addresses.

Module contents

libp2p.security package

Subpackages

libp2p.security.insecure package

Subpackages

libp2p.security.insecure.pb package

Submodules

libp2p.security.insecure.pb.plaintext_pb2 module

Generated protocol buffer code.

class libp2p.security.insecure.pb.plaintext_pb2.**Exchange**

Bases: Message, Message

DESCRIPTOR = <google._upb._message.Descriptor object>

Module contents

Submodules

libp2p.security.insecure.transport module

class libp2p.security.insecure.transport.**InsecureSession**(*, *local_peer:* ID, *local_private_key:* PrivateKey, *remote_peer:* ID, *remote_permanent_pubkey:* PublicKey, *is_initiator:* *bool*, *conn:* ReadWriteCloser)

Bases: *BaseSession*

async close() → None

async read(*n:* *int* | *None* = *None*) → bytes

async write(*data:* *bytes*) → None

class libp2p.security.insecure.transport.**InsecureTransport**(*local_key_pair: ~libp2p.crypto.keys.KeyPair, secure_bytes_provider: ~typing.Callable[[int], bytes] = <function default_secure_bytes_provider>*)

Bases: *BaseSecureTransport*

Provides the "identity" upgrader for a `IRawConnection`, i.e. the upgraded transport does not add any additional security.

async secure_inbound(*conn:* IRawConnection) → *ISecureConn*

Secure the connection, either locally or by communicating with opposing node via conn, for an inbound connection (i.e. we are not the initiator)

Returns

secure connection object (that implements secure_conn_interface)

async secure_outbound(*conn:* IRawConnection, *peer_id:* ID) → *ISecureConn*

Secure the connection, either locally or by communicating with opposing node via conn, for an inbound connection (i.e. we are the initiator)

Returns

secure connection object (that implements secure_conn_interface)

class libp2p.security.insecure.transport.**PlaintextHandshakeReadWriter**(*read_write_closer:* ReadWriteCloser)

Bases: *VarIntLengthMsgReadWriter*

max_msg_size: int = 65536

libp2p.security.insecure.transport.**make_exchange_message**(*pubkey:* PublicKey) → *Exchange*

async libp2p.security.insecure.transport.**run_handshake**(*local_peer:* ID, *local_private_key:* PrivateKey, *conn:* IRawConnection, *is_initiator:* *bool*, *remote_peer_id:* ID) → *ISecureConn*

Raise *HandshakeFailure* when handshake failed.

Module contents

libp2p.security.noise package

Subpackages

libp2p.security.noise.pb package

Submodules

libp2p.security.noise.pb.noise_pb2 module

Generated protocol buffer code.

class libp2p.security.noise.pb.noise_pb2.**NoiseHandshakePayload**

Bases: Message, Message

DESCRIPTOR = <google._upb._message.Descriptor object>

Module contents

Submodules

libp2p.security.noise.exceptions module

exception libp2p.security.noise.exceptions.**HandshakeHasNotFinished**

Bases: *NoiseFailure*

exception libp2p.security.noise.exceptions.**InvalidSignature**

Bases: *NoiseFailure*

exception libp2p.security.noise.exceptions.**NoiseFailure**

Bases: *HandshakeFailure*

exception libp2p.security.noise.exceptions.**NoiseStateError**

Bases: *NoiseFailure*

Raised when anything goes wrong in the noise state in *noiseprotocol* package.

exception libp2p.security.noise.exceptions.**PeerIDMismatchesPubkey**

Bases: *NoiseFailure*

libp2p.security.noise.io module

class libp2p.security.noise.io.**BaseNoiseMsgReadWriter**(*conn:* IRawConnection, *noise_state:* *NoiseConnection*)

Bases: *EncryptedMsgReadWriter*

The base implementation of noise message reader/writer.

encrypt and *decrypt* are not implemented here, which should be implemented by the subclasses.

async close() → None

noise_state: NoiseConnection

prefix: bytes = b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

async read_msg(*prefix_encoded: bool = False*) → bytes

read_writer: *MsgReadWriteCloser*

async write_msg(*data: bytes*, *prefix_encoded: bool = False*) → None

class libp2p.security.noise.io.**NoiseHandshakeReadWriter**(*conn:* IRawConnection, *noise_state: NoiseConnection*)

Bases: *BaseNoiseMsgReadWriter*

decrypt(*data: bytes*) → bytes

encrypt(*data: bytes*) → bytes

class libp2p.security.noise.io.**NoisePacketReadWriter**(*read_write_closer:* ReadWriteCloser)

Bases: *FixedSizeLenMsgReadWriter*

size_len_bytes: int = 2

class libp2p.security.noise.io.**NoiseTransportReadWriter**(*conn:* IRawConnection, *noise_state: NoiseConnection*)

Bases: *BaseNoiseMsgReadWriter*

decrypt(*data: bytes*) → bytes

encrypt(*data: bytes*) → bytes

libp2p.security.noise.messages module

class libp2p.security.noise.messages.**NoiseHandshakePayload**(*id_pubkey:* libp2p.crypto.keys.PublicKey, *id_sig: bytes*, *early_data: bytes = None*)

Bases: object

classmethod deserialize(*protobuf_bytes: bytes*) → *NoiseHandshakePayload*

early_data: bytes = None

id_pubkey: *PublicKey*

id_sig: bytes

serialize() → bytes

libp2p.security.noise.messages.**make_data_to_be_signed**(*noise_static_pubkey:* PublicKey) → bytes

libp2p.security.noise.messages.**make_handshake_payload_sig**(*id_privkey:* PrivateKey, *noise_static_pubkey:* PublicKey) → bytes

libp2p.security.noise.messages.**verify_handshake_payload_sig**(*payload:* NoiseHandshakePayload, *noise_static_pubkey:* PublicKey) → bool

Verify if the signature

1. is composed of the data *SIGNED_DATA_PREFIX ++ noise_static_pubkey* and
2. signed by the private key corresponding to *id_pubkey*

libp2p.security.noise.patterns module

class libp2p.security.noise.patterns.**BasePattern**

Bases: *IPattern*

create_noise_state() → NoiseConnection

early_data: bytes

libp2p_privkey: *PrivateKey*

local_peer: *ID*

make_handshake_payload() → *NoiseHandshakePayload*

noise_static_key: *PrivateKey*

protocol_name: bytes

class libp2p.security.noise.patterns.**IPattern**

Bases: ABC

abstract async handshake_inbound(*conn:* IRawConnection) → *ISecureConn*

abstract async handshake_outbound(*conn:* IRawConnection, *remote_peer:* ID) → *ISecureConn*

class libp2p.security.noise.patterns.**PatternXX**(*local_peer:* ID, *libp2p_privkey:* PrivateKey, *noise_static_key:* PrivateKey, *early_data:* *bytes* | *None* = *None*)

Bases: *BasePattern*

async handshake_inbound(*conn:* IRawConnection) → *ISecureConn*

async handshake_outbound(*conn:* IRawConnection, *remote_peer:* ID) → *ISecureConn*

libp2p.security.noise.transport module

class libp2p.security.noise.transport.**Transport**(*libp2p_keypair:* KeyPair, *noise_privkey:* PrivateKey | *None* = *None*, *early_data:* *bytes* | *None* = *None*, *with_noise_pipes:* *bool* = *False*)

Bases: *ISecureTransport*

early_data: bytes

get_pattern() → *IPattern*

libp2p_privkey: *PrivateKey*

local_peer: *ID*

noise_privkey: *PrivateKey*

async secure_inbound(*conn:* IRawConnection) → *ISecureConn*

Secure the connection, either locally or by communicating with opposing node via conn, for an inbound connection (i.e. we are not the initiator)

Returns
secure connection object (that implements secure_conn_interface)

async secure_outbound(*conn:* IRawConnection, *peer_id:* ID) → *ISecureConn*

Secure the connection, either locally or by communicating with opposing node via conn, for an inbound connection (i.e. we are the initiator)

Returns
secure connection object (that implements secure_conn_interface)

with_noise_pipes: bool

Module contents

libp2p.security.secio package

Subpackages

libp2p.security.secio.pb package

Submodules

libp2p.security.secio.pb.spipe_pb2 module

Generated protocol buffer code.

class libp2p.security.secio.pb.spipe_pb2.**Exchange**

Bases: Message, Message

DESCRIPTOR = <google._upb._message.Descriptor object>

class libp2p.security.secio.pb.spipe_pb2.**Propose**

Bases: Message, Message

DESCRIPTOR = <google._upb._message.Descriptor object>

Module contents

Submodules

libp2p.security.secio.exceptions module

exception libp2p.security.secio.exceptions.**IncompatibleChoices**

Bases: *SecioException*

exception libp2p.security.secio.exceptions.**InconsistentNonce**

Bases: *SecioException*

exception libp2p.security.secio.exceptions.**InvalidSignatureOnExchange**

Bases: *SecioException*

exception libp2p.security.secio.exceptions.**PeerMismatchException**

Bases: *SecioException*

exception libp2p.security.secio.exceptions.**SecioException**

Bases: *HandshakeFailure*

exception libp2p.security.secio.exceptions.**SedesException**

Bases: *SecioException*

exception libp2p.security.secio.exceptions.**SelfEncryption**

Bases: *SecioException*

Raised to indicate that a host is attempting to encrypt communications with itself.

libp2p.security.secio.transport module

class libp2p.security.secio.transport.**EncryptionParameters**

Bases: object

cipher_type: *str*

curve_type: *str*

ephemeral_public_key: *PublicKey*

hash_type: *str*

permanent_public_key: *PublicKey*

class libp2p.security.secio.transport.**Proposal**(*nonce: bytes*, *public_key:* PublicKey, *exchanges: str = 'P-256'*, *ciphers: str = 'AES-128'*, *hashes: str = 'SHA256'*)

Bases: object

Represents the set of session parameters one peer in a pair of peers attempting to negotiate a *secio* channel prefers.

calculate_peer_id() → *ID*

ciphers: *str* = 'AES-128'

classmethod deserialize(*protobuf_bytes: bytes*) → *Proposal*

exchanges: *str* = 'P-256'

hashes: *str* = 'SHA256'

nonce: *bytes*

public_key: *PublicKey*

serialize() → bytes

class libp2p.security.secio.transport.**SecioMsgReadWriter**(*local_encryption_parameters: EncryptionParameters, remote_encryption_parameters: EncryptionParameters, read_writer: SecioPacketReadWriter*)

Bases: *EncryptedMsgReadWriter*

async close() → None

decrypt(*data: bytes*) → bytes

encrypt(*data: bytes*) → bytes

async read_msg() → bytes

read_writer: *SecioPacketReadWriter*

async write_msg(*msg: bytes*) → None

class libp2p.security.secio.transport.**SecioPacketReadWriter**(*read_write_closer: ReadWriteCloser*)

Bases: *FixedSizeLenMsgReadWriter*

size_len_bytes: *int* = 4

class libp2p.security.secio.transport.**SessionParameters**

Bases: object

local_encryption_parameters: *EncryptionParameters*

local_peer: *ID*

order: *int*

remote_encryption_parameters: *EncryptionParameters*

remote_peer: *ID*

shared_key: *bytes*

class libp2p.security.secio.transport.**Transport**(*local_key_pair: ~libp2p.crypto.keys.KeyPair, secure_bytes_provider: ~typing.Callable[[int], bytes] = <function default_secure_bytes_provider>*)

Bases: *BaseSecureTransport*

Provide a security upgrader for a IRawConnection, following the *secio* protocol defined in the libp2p specs.

get_nonce() → bytes

async secure_inbound(*conn:* IRawConnection) → *ISecureConn*

Secure the connection, either locally or by communicating with opposing node via conn, for an inbound connection (i.e. we are not the initiator)

Returns
secure connection object (that implements secure_conn_interface)

async secure_outbound(*conn:* IRawConnection, *peer_id:* ID) → *ISecureConn*

Secure the connection, either locally or by communicating with opposing node via conn, for an inbound connection (i.e. we are the initiator)

Returns
secure connection object (that implements secure_conn_interface)

async libp2p.security.secio.transport.**create_secure_session**(*local_nonce: bytes*, *local_peer:* ID, *local_private_key:* PrivateKey, *conn:* IRawConnection, *remote_peer:* ID | *None* = *None*) → *ISecureConn*

Attempt the initial *secio* handshake with the remote peer.

If successful, return an object that provides secure communication to the `remote_peer`. Raise *SecioException* when *conn* closed. Raise *InconsistentNonce* when handshake failed

Module contents

Submodules

libp2p.security.base_session module

class libp2p.security.base_session.**BaseSession**(*, *local_peer:* ID, *local_private_key:* PrivateKey, *remote_peer:* ID, *remote_permanent_pubkey:* PublicKey, *is_initiator: bool*)

Bases: *ISecureConn*

`BaseSession` is not fully instantiated from its abstract classes as it is only meant to be used in clases that derive from it.

get_local_peer() → *ID*

get_local_private_key() → *PrivateKey*

get_remote_peer() → *ID*

get_remote_public_key() → *PublicKey* | None

local_peer: *ID*

local_private_key: *PrivateKey*

remote_peer: *ID*

remote_permanent_pubkey: *PublicKey*

libp2p.security.base_transport module

class libp2p.security.base_transport.**BaseSecureTransport**(*local_key_pair: ~libp2p.crypto.keys.KeyPair, secure_bytes_provider: ~typing.Callable[[int], bytes] = <function default_secure_bytes_provider>*)

Bases: *ISecureTransport*

BaseSecureTransport is not fully instantiated from its abstract classes as it is only meant to be used in clases that derive from it.

Clients can provide a strategy to get cryptographically secure bytes of a given length. A default implementation is provided using the secrets module from the standard library.

libp2p.security.base_transport.**default_secure_bytes_provider**(*n: int*) → bytes

libp2p.security.exceptions module

exception libp2p.security.exceptions.**HandshakeFailure**

Bases: *BaseLibp2pError*

libp2p.security.secure_conn_interface module

class libp2p.security.secure_conn_interface.**AbstractSecureConn**

Bases: ABC

abstract get_local_peer() → *ID*

abstract get_local_private_key() → *PrivateKey*

abstract get_remote_peer() → *ID*

abstract get_remote_public_key() → *PublicKey*

class libp2p.security.secure_conn_interface.**ISecureConn**

Bases: *AbstractSecureConn*, *IRawConnection*

libp2p.security.secure_session module

class libp2p.security.secure_session.**SecureSession**(*, *local_peer:* ID, *local_private_key:* PrivateKey, *remote_peer:* ID, *remote_permanent_pubkey:* PublicKey, *is_initiator: bool*, *conn:* EncryptedMsgReadWriter)

Bases: *BaseSession*

buf: BytesIO

async close() → None

high_watermark: int

low_watermark: *int*

async read(*n: int | None = None*) → bytes

async write(*data: bytes*) → None

libp2p.security.secure_transport_interface module

class libp2p.security.secure_transport_interface.**ISecureTransport**

Bases: ABC

abstract async secure_inbound(*conn:* IRawConnection) → *ISecureConn*

Secure the connection, either locally or by communicating with opposing node via conn, for an inbound connection (i.e. we are not the initiator)

Returns
secure connection object (that implements secure_conn_interface)

abstract async secure_outbound(*conn:* IRawConnection, *peer_id:* ID) → *ISecureConn*

Secure the connection, either locally or by communicating with opposing node via conn, for an inbound connection (i.e. we are the initiator)

Returns
secure connection object (that implements secure_conn_interface)

libp2p.security.security_multistream module

class libp2p.security.security_multistream.**SecurityMultistream**(*secure_transports_by_protocol: Mapping[TProtocol,* ISecureTransport*]*)

Bases: ABC

SSMuxer is a multistream stream security transport multiplexer.

Go implementation: github.com/libp2p/go-conn-security-multistream/ssms.go

add_transport(*protocol: TProtocol*, *transport:* ISecureTransport) → None

Add a protocol and its corresponding transport to multistream- select(multiselect). The order that a protocol is added is exactly the precedence it is negotiated in multiselect.

Parameters

- **protocol** – the protocol name, which is negotiated in multiselect.
- **transport** – the corresponding transportation to the `protocol`.

multiselect: *Multiselect*

multiselect_client: *MultiselectClient*

async secure_inbound(*conn:* IRawConnection) → *ISecureConn*

Secure the connection, either locally or by communicating with opposing node via conn, for an inbound connection (i.e. we are not the initiator)

Returns
secure connection object (that implements secure_conn_interface)

async secure_outbound(*conn:* IRawConnection, *peer_id:* ID) → *ISecureConn*

Secure the connection, either locally or by communicating with opposing node via conn, for an inbound connection (i.e. we are the initiator)

Returns
secure connection object (that implements secure_conn_interface)

async select_transport(*conn:* IRawConnection, *is_initiator:* *bool*) → *ISecureTransport*

Select a transport that both us and the node on the other end of conn support and agree on.

Parameters

- **conn** – conn to choose a transport over
- **is_initiator** – true if we are the initiator, false otherwise

Returns
selected secure transport

transports: OrderedDict[TProtocol, *ISecureTransport*]

Module contents

libp2p.stream_muxer package

Subpackages

libp2p.stream_muxer.mplex package

Submodules

libp2p.stream_muxer.mplex.constants module

class libp2p.stream_muxer.mplex.constants.**HeaderTags**(*value*)

Bases: Enum

An enumeration.

CloseInitiator = 4

CloseReceiver = 3

MessageInitiator = 2

MessageReceiver = 1

NewStream = 0

ResetInitiator = 6

ResetReceiver = 5

libp2p.stream_muxer.mplex.datastructures module

class libp2p.stream_muxer.mplex.datastructures.**StreamID**(*channel_id*, *is_initiator*)

Bases: NamedTuple

channel_id: int

Alias for field number 0

is_initiator: bool

Alias for field number 1

libp2p.stream_muxer.mplex.exceptions module

exception libp2p.stream_muxer.mplex.exceptions.**MplexError**

Bases: *MuxedConnError*

exception libp2p.stream_muxer.mplex.exceptions.**MplexStreamClosed**

Bases: *MuxedStreamClosed*

exception libp2p.stream_muxer.mplex.exceptions.**MplexStreamEOF**

Bases: *MuxedStreamEOF*

exception libp2p.stream_muxer.mplex.exceptions.**MplexStreamReset**

Bases: *MuxedStreamReset*

exception libp2p.stream_muxer.mplex.exceptions.**MplexUnavailable**

Bases: *MuxedConnUnavailable*

libp2p.stream_muxer.mplex.mplex module

class libp2p.stream_muxer.mplex.mplex.**Mplex**(*secured_conn:* ISecureConn, *peer_id:* ID)

Bases: *IMuxedConn*

reference: https://github.com/libp2p/go-mplex/blob/master/multiplex.go

async accept_stream() → *IMuxedStream*

Accept a muxed stream opened by the other end.

async close() → None

Close the stream muxer and underlying secured connection.

event_closed: Event

event_shutting_down: Event

event_started: Event

async handle_incoming() → None

Read a message off of the secured connection and add it to the corresponding message buffer.

property is_closed: bool

Check connection is fully closed.

Returns

true if successful

property is_initiator: *bool*

If this connection is the initiator.

new_stream_receive_channel: *trio.MemoryReceiveChannel[IMuxedStream]*

new_stream_send_channel: *trio.MemorySendChannel[IMuxedStream]*

next_channel_id: *int*

async open_stream() → *IMuxedStream*

Create a new muxed_stream.

Returns
a new MplexStream

peer_id: *ID*

async read_message() → Tuple[int, int, bytes]

Read a single message off of the secured connection.

Returns
stream_id, flag, message contents

secured_conn: *ISecureConn*

async send_message(*flag:* HeaderTags, *data:* *bytes* | *None*, *stream_id:* StreamID) → int

Send a message over the connection.

Parameters

- **flag** – header to use
- **data** – data to send in the message
- **stream_id** – stream the message is in

async start() → None

Start the multiplexer.

streams: *Dict[StreamID, MplexStream]*

streams_lock: *Lock*

streams_msg_channels: *Dict[StreamID, trio.MemorySendChannel[bytes]]*

async write_to_stream(*_bytes:* *bytes*) → None

Write a byte array to a secured connection.

Parameters
_bytes – byte array to write

Returns
length written

libp2p.stream_muxer.mplex.mplex_stream module

class libp2p.stream_muxer.mplex.mplex_stream.**MplexStream**(*name: str*, *stream_id:* StreamID, *muxed_conn:* Mplex, *incoming_data_channel: trio.MemoryReceiveChannel[bytes]*)

Bases: *IMuxedStream*

reference: https://github.com/libp2p/go-mplex/blob/master/stream.go

async close() → None

Closing a stream closes it for writing and closes the remote end for reading but allows writing in the other direction.

close_lock: Lock

event_local_closed: Event

event_remote_closed: Event

event_reset: Event

incoming_data_channel: trio.MemoryReceiveChannel[bytes]

property is_initiator: bool

muxed_conn: *Mplex*

name: str

async read(*n: int | None = None*) → bytes

Read up to n bytes. Read possibly returns fewer than *n* bytes, if there are not enough bytes in the Mplex buffer. If *n is None*, read until EOF.

Parameters
n – number of bytes to read

Returns
bytes actually read

read_deadline: int

async reset() → None

Close both ends of the stream tells this remote side to hang up.

set_deadline(*ttl: int*) → bool

Set deadline for muxed stream.

Returns
True if successful

set_read_deadline(*ttl: int*) → bool

Set read deadline for muxed stream.

Returns
True if successful

set_write_deadline(*ttl: int*) → bool

Set write deadline for muxed stream.

Returns
True if successful

stream_id: *StreamID*

async write(*data: bytes*) → None

Write to stream.

Returns
number of bytes written

write_deadline: int

Module contents

Submodules

libp2p.stream_muxer.abc module

class libp2p.stream_muxer.abc.**IMuxedConn**(*conn:* ISecureConn, *peer_id:* ID)

Bases: ABC

reference: https://github.com/libp2p/go-stream-muxer/blob/master/muxer.go

abstract async accept_stream() → *IMuxedStream*

Accept a muxed stream opened by the other end.

abstract async close() → None

Close connection.

event_started: Event

abstract property is_closed: bool

Check connection is fully closed.

Returns
true if successful

abstract property is_initiator: bool

If this connection is the initiator.

abstract async open_stream() → *IMuxedStream*

Create a new muxed_stream.

Returns
a new IMuxedStream stream

peer_id: *ID*

abstract async start() → None

Start the multiplexer.

class libp2p.stream_muxer.abc.**IMuxedStream**

Bases: *ReadWriteCloser*

muxed_conn: *IMuxedConn*

abstract async reset() → None

Close both ends of the stream tells this remote side to hang up.

abstract set_deadline(*ttl: int*) → bool

Set deadline for muxed stream.

Returns
a new stream

libp2p.stream_muxer.exceptions module

exception libp2p.stream_muxer.exceptions.**MuxedConnError**

Bases: *BaseLibp2pError*

exception libp2p.stream_muxer.exceptions.**MuxedConnUnavailable**

Bases: *MuxedConnError*

exception libp2p.stream_muxer.exceptions.**MuxedStreamClosed**

Bases: *MuxedStreamError*

exception libp2p.stream_muxer.exceptions.**MuxedStreamEOF**

Bases: *MuxedStreamError*, EOFError

exception libp2p.stream_muxer.exceptions.**MuxedStreamError**

Bases: *BaseLibp2pError*

exception libp2p.stream_muxer.exceptions.**MuxedStreamReset**

Bases: *MuxedStreamError*

libp2p.stream_muxer.muxer_multistream module

class libp2p.stream_muxer.muxer_multistream.**MuxerMultistream**(*muxer_transports_by_protocol: Mapping[TProtocol, Type[IMuxedConn]]*)

Bases: object

MuxerMultistream is a multistream stream muxed transport multiplexer.

go implementation: github.com/libp2p/go-stream-muxer-multistream/multistream.go

add_transport(*protocol: TProtocol*, *transport: Type[IMuxedConn]*) → None

Add a protocol and its corresponding transport to multistream- select(multiselect). The order that a protocol is added is exactly the precedence it is negotiated in multiselect.

Parameters

- **protocol** – the protocol name, which is negotiated in multiselect.
- **transport** – the corresponding transportation to the protocol.

multiselect: *Multiselect*

multiselect_client: *MultiselectClient*

async new_conn(*conn:* ISecureConn, *peer_id:* ID) → *IMuxedConn*

async select_transport(*conn:* IRawConnection) → Type[*IMuxedConn*]

Select a transport that both us and the node on the other end of conn support and agree on.

Parameters
conn – conn to choose a transport over

Returns
selected muxer transport

transports: OrderedDict[TProtocol, Type[*IMuxedConn*]]

Module contents

libp2p.tools package

Subpackages

libp2p.tools.pubsub package

Submodules

libp2p.tools.pubsub.dummy_account_node module

class libp2p.tools.pubsub.dummy_account_node.**DummyAccountNode**(*pubsub:* Pubsub)

Bases: Service

Node which has an internal balance mapping, meant to serve as a dummy crypto blockchain.

There is no actual blockchain, just a simple map indicating how much crypto each user in the mappings holds

classmethod create(*number: int*) → AsyncIterator[Tuple[*DummyAccountNode*, ...]]

Create a new DummyAccountNode and attach a libp2p node, a floodsub, and a pubsub instance to this new node.

We use create as this serves as a factory function and allows us to use async await, unlike the init function

get_balance(*user: str*) → int

Get balance in crypto for a particular user.

Parameters
user – user to get balance for

Returns
balance of user

async handle_incoming_msgs() → None

Handle all incoming messages on the CRYPTO_TOPIC from peers.

handle_send_crypto(*source_user: str*, *dest_user: str*, *amount: int*) → None

Handle incoming send_crypto message.

Parameters

- **source_user** – user to send crypto from
- **dest_user** – user to send crypto to

- **amount** – amount of crypto to send

handle_set_crypto(*dest_user: str*, *amount: int*) → None

Handle incoming set_crypto message.

Parameters

- **dest_user** – user to set crypto for
- **amount** – amount of crypto

property host: *IHost*

async publish_send_crypto(*source_user: str*, *dest_user: str*, *amount: int*) → None

Create a send crypto message and publish that message to all other nodes.

Parameters

- **source_user** – user to send crypto from
- **dest_user** – user to send crypto to
- **amount** – amount of crypto to send

async publish_set_crypto(*user: str*, *amount: int*) → None

Create a set crypto message and publish that message to all other nodes.

Parameters

- **user** – user to set crypto for
- **amount** – amount of crypto

pubsub: *Pubsub*

async run() → None

Primary entry point for all service logic.

Note: This method should **not** be directly invoked by user code.

Services may be run using the following approaches.

libp2p.tools.pubsub.floodsub_integration_test_settings module

async libp2p.tools.pubsub.floodsub_integration_test_settings.**perform_test_from_obj**(*obj*, *pubsub_factory*) → None

Perform pubsub tests from a test object, which is composed as follows:

```
{
    "supported_protocols": ["supported/protocol/1.0.0",...],
    "adj_list": {
        "node1": ["neighbor1_of_node1", "neighbor2_of_node1", ...],
        "node2": ["neighbor1_of_node2", "neighbor2_of_node2", ...],
        ...
    },
    "topic_map": {
```

(continues on next page)

(continued from previous page)

```
        "topic1": ["node1_subscribed_to_topic1", "node2_subscribed_to_topic1", ...]
    },
    "messages": [
        {
            "topics": ["topic1_for_message", "topic2_for_message", ...],
            "data": b"some contents of the message (newlines are not supported)",
            "node_id": "message sender node id"
        },
        ...
    ]
}
```

Note: In adj_list, for any neighbors A and B, only list B as a neighbor of A or B as a neighbor of A once. Do NOT list both A: ["B"] and B:["A"] as the behavior is undefined (even if it may work)

libp2p.tools.pubsub.utils module

async libp2p.tools.pubsub.utils.**connect_some**(*hosts: Sequence[IHost]*, *degree: int*) → None

async libp2p.tools.pubsub.utils.**dense_connect**(*hosts: Sequence[IHost]*) → None

libp2p.tools.pubsub.utils.**make_pubsub_msg**(*origin_id: ID*, *topic_ids: Sequence[str]*, *data: bytes*, *seqno: bytes*) → *Message*

async libp2p.tools.pubsub.utils.**one_to_all_connect**(*hosts: Sequence[IHost]*, *central_host_index: int*) → None

Module contents

Submodules

libp2p.tools.constants module

class libp2p.tools.constants.**GossipsubParams**(*degree*, *degree_low*, *degree_high*, *time_to_live*, *gossip_window*, *gossip_history*, *heartbeat_initial_delay*, *heartbeat_interval*)

Bases: NamedTuple

degree: int

Alias for field number 0

degree_high: int

Alias for field number 2

degree_low: int

Alias for field number 1

gossip_history: int

Alias for field number 5

gossip_window: int

Alias for field number 4

heartbeat_initial_delay: float

Alias for field number 6

heartbeat_interval: float

Alias for field number 7

time_to_live: int

Alias for field number 3

libp2p.tools.factories module

class libp2p.tools.factories.**DummyRouter**

Bases: *IPeerRouting*

async **find_peer**(*peer_id:* ID) → *PeerInfo*

Find specific Peer FindPeer searches for a peer with given peer_id, returns a peer.PeerInfo with relevant addresses.

class libp2p.tools.factories.**FloodsubFactory**(***kwargs*)

Bases: Factory

protocols = ('/floodsub/1.0.0',)

class libp2p.tools.factories.**GossipsubFactory**(***kwargs*)

Bases: Factory

degree = 10

degree_high = 11

degree_low = 9

gossip_history = 5

gossip_window = 3

heartbeat_initial_delay = 0.1

heartbeat_interval = 0.5

protocols = ('/meshsub/1.0.0',)

time_to_live = 30

class libp2p.tools.factories.**HostFactory**(***kwargs*)

Bases: Factory

classmethod **create_batch_and_listen**(*number: int*, *security_protocol: TProtocol = None*, *muxer_opt: Mapping[TProtocol, Type[IMuxedConn]] = None*) → AsyncIterator[Tuple[*BasicHost*, ...]]

network = <factory.declarations.LazyAttribute object>

class libp2p.tools.factories.**IDFactory**(***kwargs*)

Bases: Factory

peer_id_bytes = <factory.declarations.LazyFunction object>

class libp2p.tools.factories.**PubsubFactory**(***kwargs*)

Bases: Factory

cache_size = None

classmethod create_and_start(*host: IHost, router: IPubsubRouter, cache_size: int, strict_signing: bool, msg_id_constructor: Callable[[Message], bytes] = None*) → AsyncIterator[*Pubsub*]

classmethod create_batch_with_floodsub(*number: int, cache_size: int = None, strict_signing: bool = False, protocols: ~typing.Sequence[TProtocol] = None, security_protocol: TProtocol = None, muxer_opt: ~typing.Mapping[TProtocol, ~typing.Type[~libp2p.stream_muxer.abc.IMuxedConn]] = None, msg_id_constructor: ~typing.Callable[[~libp2p.pubsub.pb.rpc_pb2.Message], bytes] = <function get_peer_and_seqno_msg_id>*) → AsyncIterator[Tuple[*Pubsub*, ...]]

classmethod create_batch_with_gossipsub(*number: int, *, cache_size: int = None, strict_signing: bool = False, protocols: ~typing.Sequence[TProtocol] = None, degree: int = 10, degree_low: int = 9, degree_high: int = 11, time_to_live: int = 30, gossip_window: int = 3, gossip_history: int = 5, heartbeat_interval: float = 0.5, heartbeat_initial_delay: float = 0.1, security_protocol: TProtocol = None, muxer_opt: ~typing.Mapping[TProtocol, ~typing.Type[~libp2p.stream_muxer.abc.IMuxedConn]] = None, msg_id_constructor: ~typing.Callable[[~libp2p.pubsub.pb.rpc_pb2.Message], bytes] = <function get_peer_and_seqno_msg_id>*) → AsyncIterator[Tuple[*Pubsub*, ...]]

host = <factory.declarations.SubFactory object>

router = None

strict_signing = False

class libp2p.tools.factories.**RoutedHostFactory**(***kwargs*)

Bases: Factory

classmethod create_batch_and_listen(*number: int, security_protocol: TProtocol = None, muxer_opt: Mapping[TProtocol, Type[IMuxedConn]] = None*) → AsyncIterator[Tuple[*RoutedHost*, ...]]

network = <factory.declarations.LazyAttribute object>

router = <factory.declarations.LazyFunction object>

class libp2p.tools.factories.**SwarmFactory**(***kwargs*)

Bases: Factory

classmethod create_and_listen(*key_pair:* KeyPair = *None*, *security_protocol: TProtocol = None*, *muxer_opt: Mapping[TProtocol, Type[*IMuxedConn*]] = None*) → AsyncIterator[*Swarm*]

classmethod create_batch_and_listen(*number: int*, *security_protocol: TProtocol = None*, *muxer_opt: Mapping[TProtocol, Type[*IMuxedConn*]] = None*) → AsyncIterator[Tuple[*Swarm*, ...]]

peer_id = <factory.declarations.LazyAttribute object>

peerstore = <factory.declarations.LazyAttribute object>

transport = <factory.declarations.LazyFunction object>

upgrader = <factory.declarations.LazyAttribute object>

libp2p.tools.factories.**default_key_pair_factory**() → *KeyPair*

libp2p.tools.factories.**default_muxer_transport_factory**() → Mapping[TProtocol, Type[*IMuxedConn*]]

libp2p.tools.factories.**host_pair_factory**(*security_protocol: TProtocol = None*, *muxer_opt: Mapping[TProtocol, Type[*IMuxedConn*]] = None*) → AsyncIterator[Tuple[*BasicHost*, *BasicHost*]]

libp2p.tools.factories.**initialize_peerstore_with_our_keypair**(*self_id:* ID, *key_pair:* KeyPair) → *PeerStore*

libp2p.tools.factories.**mplex_conn_pair_factory**(*security_protocol: TProtocol = None*) → AsyncIterator[Tuple[*Mplex*, *Mplex*]]

libp2p.tools.factories.**mplex_stream_pair_factory**(*security_protocol: TProtocol = None*) → AsyncIterator[Tuple[*MplexStream*, *MplexStream*]]

libp2p.tools.factories.**mplex_transport_factory**() → Mapping[TProtocol, Type[*IMuxedConn*]]

libp2p.tools.factories.**net_stream_pair_factory**(*security_protocol: TProtocol = None*, *muxer_opt: Mapping[TProtocol, Type[*IMuxedConn*]] = None*) → AsyncIterator[Tuple[*INetStream*, *INetStream*]]

libp2p.tools.factories.**noise_conn_factory**(*nursery: Nursery*) → AsyncIterator[Tuple[*ISecureConn*, *ISecureConn*]]

libp2p.tools.factories.**noise_handshake_payload_factory**() → *NoiseHandshakePayload*

libp2p.tools.factories.**noise_static_key_factory**() → *PrivateKey*

libp2p.tools.factories.**noise_transport_factory**(*key_pair:* KeyPair) → *ISecureTransport*

libp2p.tools.factories.**plaintext_transport_factory**(*key_pair:* KeyPair) → *ISecureTransport*

libp2p.tools.factories.**raw_conn_factory**(*nursery: Nursery*) → AsyncIterator[Tuple[*IRawConnection*, *IRawConnection*]]

libp2p.tools.factories.**secio_transport_factory**(*key_pair:* KeyPair) → *ISecureTransport*

libp2p.tools.factories.**security_options_factory_factory**(*protocol_id: TProtocol | None = None*) → Callable[[*KeyPair*], Mapping[TProtocol, *ISecureTransport*]]

libp2p.tools.factories.**swarm_conn_pair_factory**(*security_protocol: TProtocol = None*, *muxer_opt: Mapping[TProtocol, Type[*IMuxedConn*]] = None*) → AsyncIterator[Tuple[*SwarmConn*, *SwarmConn*]]

libp2p.tools.factories.**swarm_pair_factory**(*security_protocol: TProtocol = None*, *muxer_opt: Mapping[TProtocol, Type[*IMuxedConn*]] = None*) → AsyncIterator[Tuple[*Swarm*, *Swarm*]]

libp2p.tools.utils module

async libp2p.tools.utils.**connect**(*node1:* IHost, *node2:* IHost) → None

Connect node1 to node2.

async libp2p.tools.utils.**connect_swarm**(*swarm_0:* Swarm, *swarm_1:* Swarm) → None

libp2p.tools.utils.**create_echo_stream_handler**(*ack_prefix: str*) → Callable[[*INetStream*], Awaitable[None]]

Module contents

libp2p.transport package

Subpackages

libp2p.transport.tcp package

Submodules

libp2p.transport.tcp.tcp module

class libp2p.transport.tcp.tcp.**TCP**

Bases: *ITransport*

create_listener(*handler_function: Callable[[*ReadWriteCloser*], Awaitable[None]]*) → *TCPListener*

Create listener on transport.

Parameters

handler_function – a function called when a new connection is received that takes a connection as argument which implements interface-connection

Returns

a listener object that implements listener_interface.py

async dial(*maddr: Multiaddr*) → *IRawConnection*

Dial a transport to peer listening on multiaddr.

Parameters

maddr – multiaddr of peer

Returns
RawConnection if successful

Raises
OpenConnectionError – raised when failed to open connection

class libp2p.transport.tcp.tcp.**TCPListener**(*handler_function: Callable[[ReadWriteCloser], Awaitable[None]]*)

Bases: *IListener*

async close() → None

get_addrs() → Tuple[Multiaddr, ...]

Retrieve list of addresses the listener is listening on.

Returns
return list of addrs

async listen(*maddr: Multiaddr*, *nursery: Nursery*) → None

Put listener in listening mode and wait for incoming connections.

Parameters
maddr – maddr of peer

Returns
return True if successful

listeners: List[SocketListener]

Module contents

Submodules

libp2p.transport.exceptions module

exception libp2p.transport.exceptions.**MuxerUpgradeFailure**

Bases: *UpgradeFailure*

exception libp2p.transport.exceptions.**OpenConnectionError**

Bases: *BaseLibp2pError*

exception libp2p.transport.exceptions.**SecurityUpgradeFailure**

Bases: *UpgradeFailure*

exception libp2p.transport.exceptions.**UpgradeFailure**

Bases: *BaseLibp2pError*

libp2p.transport.listener_interface module

class libp2p.transport.listener_interface.**IListener**

Bases: ABC

abstract async close() → None

abstract get_addrs() → Tuple[Multiaddr, ...]

Retrieve list of addresses the listener is listening on.

Returns
return list of addrs

abstract async listen(*maddr: Multiaddr*, *nursery: Nursery*) → bool

Put listener in listening mode and wait for incoming connections.

Parameters
maddr – multiaddr of peer

Returns
return True if successful

libp2p.transport.transport_interface module

class libp2p.transport.transport_interface.**ITransport**

Bases: ABC

abstract create_listener(*handler_function: Callable[[*ReadWriteCloser*], Awaitable[None]]*) → *IListener*

Create listener on transport.

Parameters
handler_function – a function called when a new conntion is received that takes a connection as argument which implements interface-connection

Returns
a listener object that implements listener_interface.py

abstract async dial(*maddr: Multiaddr*) → *IRawConnection*

Dial a transport to peer listening on multiaddr.

Parameters

- **multiaddr** – multiaddr of peer
- **self_id** – peer_id of the dialer (to send to receiver)

Returns
list of multiaddrs

libp2p.transport.typing module

libp2p.transport.upgrader module

class libp2p.transport.upgrader.**TransportUpgrader**(*secure_transports_by_protocol: Mapping[TProtocol, ISecureTransport], muxer_transports_by_protocol: Mapping[TProtocol, Type[IMuxedConn]]*)

Bases: object

muxer_multistream: *MuxerMultistream*

security_multistream: *SecurityMultistream*

async upgrade_connection(*conn:* ISecureConn, *peer_id:* ID) → *IMuxedConn*

Upgrade secured connection to a muxed connection.

upgrade_listener(*transport:* ITransport, *listeners:* IListener) → None

Upgrade multiaddr listeners to libp2p-transport listeners.

async upgrade_security(*raw_conn:* IRawConnection, *peer_id:* ID, *is_initiator: bool*) → *ISecureConn*

Upgrade conn to a secured connection.

Module contents

1.5.2 Submodules

1.5.3 libp2p.exceptions module

exception libp2p.exceptions.**BaseLibp2pError**

Bases: Exception

exception libp2p.exceptions.**MultiError**

Bases: *BaseLibp2pError*

Raised with multiple exceptions.

exception libp2p.exceptions.**ParseError**

Bases: *BaseLibp2pError*

exception libp2p.exceptions.**ValidationError**

Bases: *BaseLibp2pError*

Raised when something does not pass a validation check.

1.5.4 libp2p.typing module

1.5.5 libp2p.utils module

async libp2p.utils.**decode_uvarint_from_stream**(*reader:* Reader) → int
https://en.wikipedia.org/wiki/LEB128.

libp2p.utils.**encode_delim**(*msg: bytes*) → bytes

libp2p.utils.**encode_uvarint**(*number: int*) → bytes
Pack *number* into varint bytes.

libp2p.utils.**encode_varint_prefixed**(*msg_bytes: bytes*) → bytes

async libp2p.utils.**read_delim**(*reader:* Reader) → bytes

async libp2p.utils.**read_varint_prefixed_bytes**(*reader:* Reader) → bytes

1.5.6 Module contents

libp2p.**generate_new_rsa_identity**() → *KeyPair*

libp2p.**generate_peer_id_from**(*key_pair:* KeyPair) → *ID*

libp2p.**new_host**(*key_pair:* KeyPair | *None* = *None*, *muxer_opt: Mapping[TProtocol, Type[*IMuxedConn*]]* | *None* = *None*, *sec_opt: Mapping[TProtocol,* ISecureTransport*]* | *None* = *None*, *peerstore_opt:* IPeerStore | *None* = *None*, *disc_opt:* IPeerRouting | *None* = *None*) → *IHost*

Create a new libp2p host based on the given parameters.

Parameters

- **key_pair** – optional choice of the KeyPair
- **muxer_opt** – optional choice of stream muxer
- **sec_opt** – optional choice of security upgrade
- **peerstore_opt** – optional peerstore
- **disc_opt** – optional discovery

Returns
return a host instance

libp2p.**new_swarm**(*key_pair:* KeyPair | *None* = *None*, *muxer_opt: Mapping[TProtocol, Type[*IMuxedConn*]]* | *None* = *None*, *sec_opt: Mapping[TProtocol,* ISecureTransport*]* | *None* = *None*, *peerstore_opt:* IPeerStore | *None* = *None*) → *INetworkService*

Create a swarm instance based on the parameters.

Parameters

- **key_pair** – optional choice of the KeyPair
- **muxer_opt** – optional choice of stream muxer
- **sec_opt** – optional choice of security upgrade
- **peerstore_opt** – optional peerstore

Returns
return a default swarm instance

1.6 Contributing

1.6.1 Development

To get started, fork the repository to your own github account, then clone it to your development machine:

```
git clone git@github.com:your-github-username/py-libp2p.git
```

then install the development dependencies. We recommend using a virtual environment, such as virtualenv.

```
cd py-libp2p
virtualenv -p python venv
. venv/bin/activate
python -m pip install -e ".[dev]"
pre-commit install
```

Dependencies

On Debian Linux you will need to ensure that you have the GNU Multiprecision Arithmetic Library installed since it is a dependency of the fastecdsa package. You can install it using the following command:

```
sudo apt-get install libgmp-dev
```

Requirements

The protobuf description in this repository was generated by `protoc` at version `25.3`.

1.6.2 Testing

Running that tests is a great way to explore the codebase.

You can run all the tests with `pytest tests`.

At this time, the interop tests are not passing. You can run just the internal tests with `pytest tests/core`.

1.6.3 Code Style

We use pre-commit to maintain consistent code style. Once installed, it will run automatically with every commit. You can also run it manually with:

```
make lint
```

If you need to make a commit that skips the pre-commit checks, you can do so with `git commit --no-verify`.

This project uses `mypy` for static type checking, though it is not yet complete. All new code should be fully typed, and we are working to add types to the existing codebase.

1.6.4 Releasing

Releases are typically done from the `main` branch, except when releasing a beta (in which case the beta is released from `main`, and the previous stable branch is released from said branch).

Final test before each release

Before releasing a new version, build and test the package that will be released:

```
git checkout main && git pull

make package-test
```

This will build the package and install it in a temporary virtual environment. Follow the instructions to activate the venv and test whatever you think is important.

You can also preview the release notes:

```
towncrier --draft
```

Build the release notes

Before bumping the version number, build the release notes. You must include the part of the version to bump (see below), which changes how the version number will show in the release notes.

```
make notes bump=$$VERSION_PART_TO_BUMP$$
```

If there are any errors, be sure to re-run make notes until it works.

Push the release to github & pypi

After confirming that the release package looks okay, release a new version:

```
make release bump=$$VERSION_PART_TO_BUMP$$
```

This command will:

- Bump the version number as specified in `.pyproject.toml` and `setup.py`.
- Create a git commit and tag for the new version.
- Build the package.
- Push the commit and tag to github.
- Push the new package files to pypi.

Which version part to bump

`$$VERSION_PART_TO_BUMP$$` must be one of: `major`, `minor`, `patch`, `stage`, or `devnum`.

The version format for this repo is `{major}.{minor}.{patch}` for stable, and `{major}.{minor}.{patch}-{stage}.{devnum}` for unstable (`stage` can be alpha or beta).

If you are in a beta version, `make release bump=stage` will switch to a stable.

To issue an unstable version when the current version is stable, specify the new version explicitly, like `make release bump="--new-version 4.0.0-alpha.1"`

You can see what the result of bumping any particular version part would be with `bump-my-version show-bump`

1.7 History

Prior to 2023, this project was graciously sponsored by the Ethereum Foundation through Wave 5 of their Grants Program.

The creators and original maintainers of this project are:

- @zixuanzh
- @alexh
- @stuckinaboot
- @robzajac
- @carver

1.8 Code of Conduct

The libp2p project operates under the IPFS Code of Conduct

tl;dr:

- Be respectful.
- We're here to help: abuse@ipfs.io
- Abusive behavior is never tolerated.
- Violations of this code may result in swift and permanent expulsion from the IPFS [and libp2p] community.
- "Too long, didn't read" is not a valid excuse for not knowing what is in this document.

PYTHON MODULE INDEX

e

l

INDEX

C

D

E

F

G

H

J

K

L

M

N

O

P

R

S

T

U

V

W

X