Ethereum 2.0  
Guide

RLPx Protocol and Repository Specification

Walkthrough

Publisher: Justin L. Mitchell - Ethereum Developer

# February 20, 2021 (update)

This documentation is a walkthrough of the RLPx Protocol Specifications and GitHub Repository.

This specification defines the RLPx transport protocol, a TCP-based transport protocol

used for communication among Ethereum nodes. The protocol carries encrypted messages

belonging to one or more 'capabilities' which are negotiated during connection

establishment. RLPx is named after the [RLP] serialization format. The name is not an

acronym and has no particular meaning.

The current protocol version is \*\*5\*\*. You can find a list of changes in past versions at

the end of this document.

## Notation

`X || Y`\

denotes concatenation of X and Y.\

`X ^ Y`\

is byte-wise XOR of X and Y.\

`X[:N]`\

denotes an N-byte prefix of X.\

`[X, Y, Z, ...]`\

denotes recursive encoding as an RLP list.\

`keccak256(MESSAGE)`\

is the Keccak256 hash function as used by Ethereum.\

`ecies.encrypt(PUBKEY, MESSAGE, AUTHDATA)`\

is the asymmetric authenticated encryption function as used by RLPx.\

AUTHDATA is authenticated data which is not part of the resulting ciphertext,\

but written to HMAC-256 before generating the message tag.\

`ecdh.agree(PRIVKEY, PUBKEY)`\

is elliptic curve Diffie-Hellman key agreement between PRIVKEY and PUBKEY.

## ECIES Encryption

ECIES (Elliptic Curve Integrated Encryption Scheme) is an asymmetric encryption method

used in the RLPx handshake. The cryptosystem used by RLPx is

- The elliptic curve secp256k1 with generator `G`.

- `KDF(k, len)`: the NIST SP 800-56 Concatenation Key Derivation Function

- `MAC(k, m)`: HMAC using the SHA-256 hash function.

- `AES(k, iv, m)`: the AES-128 encryption function in CTR mode.

Alice wants to send an encrypted message that can be decrypted by Bobs static private key

<code>k<sub>B</sub></code>. Alice knows about Bobs static public key

<code>K<sub>B</sub></code>.

To encrypt the message `m`, Alice generates a random number `r` and corresponding elliptic

curve public key `R = r \* G` and computes the shared secret <code>S = P<sub>x</sub></code>

where <code>(P<sub>x</sub>, P<sub>y</sub>) = r \* K<sub>B</sub></code>. She derives key

material for encryption and authentication as

<code>k<sub>E</sub> || k<sub>M</sub> = KDF(S, 32)</code> as well as a random

initialization vector `iv`. Alice sends the encrypted message `R || iv || c || d` where

<code>c = AES(k<sub>E</sub>, iv , m)</code> and

<code>d = MAC(sha256(k<sub>M</sub>), iv || c)</code> to Bob.

For Bob to decrypt the message `R || iv || c || d`, he derives the shared secret

<code>S = P<sub>x</sub></code> where

<code>(P<sub>x</sub>, P<sub>y</sub>) = k<sub>B</sub> \* R</code> as well as the encryption and

authentication keys <code>k<sub>E</sub> || k<sub>M</sub> = KDF(S, 32)</code>. Bob verifies

the authenticity of the message by checking whether

<code>d == MAC(sha256(k<sub>M</sub>), iv || c)</code> then obtains the plaintext as

<code>m = AES(k<sub>E</sub>, iv || c)</code>.

## Node Identity

All cryptographic operations are based on the secp256k1 elliptic curve. Each node is

expected to maintain a static secp256k1 private key which is saved and restored between

sessions. It is recommended that the private key can only be reset manually, for example,

by deleting a file or database entry.

## Initial Handshake

An RLPx connection is established by creating a TCP connection and agreeing on ephemeral

key material for further encrypted and authenticated communication. The process of

creating those session keys is the 'handshake' and is carried out between the 'initiator'

(the node which opened the TCP connection) and the 'recipient' (the node which accepted it).

1. initiator connects to recipient and sends its `auth` message

2. recipient accepts, decrypts and verifies `auth` (checks that recovery of signature ==

`keccak256(ephemeral-pubk)`)

3. recipient generates `auth-ack` message from `remote-ephemeral-pubk` and `nonce`

4. recipient derives secrets and sends the first encrypted frame containing the [Hello] message

5. initiator receives `auth-ack` and derives secrets

6. initiator sends its first encrypted frame containing initiator [Hello] message

7. recipient receives and authenticates first encrypted frame

8. initiator receives and authenticates first encrypted frame

9. cryptographic handshake is complete if MAC of first encrypted frame is valid on both sides

Either side may disconnect if authentication of the first framed packet fails.

Handshake messages:

auth = auth-size || enc-auth-body

auth-size = size of enc-auth-body, encoded as a big-endian 16-bit integer

auth-vsn = 4

auth-body = [sig, initiator-pubk, initiator-nonce, auth-vsn, ...]

enc-auth-body = ecies.encrypt(recipient-pubk, auth-body || auth-padding, auth-size)

auth-padding = arbitrary data

ack = ack-size || enc-ack-body

ack-size = size of enc-ack-body, encoded as a big-endian 16-bit integer

ack-vsn = 4

ack-body = [recipient-ephemeral-pubk, recipient-nonce, ack-vsn, ...]

enc-ack-body = ecies.encrypt(initiator-pubk, ack-body || ack-padding, ack-size)

ack-padding = arbitrary data

Implementations must ignore any mismatches in `auth-vsn` and `ack-vsn`. Implementations

must also ignore any additional list elements in `auth-body` and `ack-body`.

Secrets generated following the exchange of handshake messages:

static-shared-secret = ecdh.agree(privkey, remote-pubk)

ephemeral-key = ecdh.agree(ephemeral-privkey, remote-ephemeral-pubk)

shared-secret = keccak256(ephemeral-key || keccak256(nonce || initiator-nonce))

aes-secret = keccak256(ephemeral-key || shared-secret)

mac-secret = keccak256(ephemeral-key || aes-secret)

## Framing

All messages following the initial handshake are framed. A frame carries a single

encrypted message belonging to a capability.

The purpose of framing is multiplexing multiple capabilites over a single connection.

Secondarily, as framed messages yield reasonable demarcation points for message

authentication codes, supporting an encrypted and authenticated stream becomes

straight-forward. Frames are encrypted and authenticated via key material generated during

the handshake.

The frame header provides information about the size of the message and the message's

source capability. Padding is used to prevent buffer starvation, such that frame

components are byte-aligned to block size of cipher.

frame = header-ciphertext || header-mac || frame-ciphertext || frame-mac

header-ciphertext = aes(aes-secret, header)

header = frame-size || header-data || header-padding

header-data = [capability-id, context-id]

capability-id = integer, always zero

context-id = integer, always zero

header-padding = zero-fill header to 16-byte boundary

frame-ciphertext = aes(aes-secret, frame-data || frame-padding)

frame-padding = zero-fill frame-data to 16-byte boundary

See the [Capability Messaging] section for definitions of `frame-data` and `frame-size.`

### MAC

Message authentication in RLPx uses two keccak256 states, one for each direction of

communication. The `egress-mac` and `ingress-mac` keccak states are continuously updated

with the ciphertext of bytes sent (egress) or received (ingress). Following the initial

handshake, the MAC states are initialized as follows:

Initiator:

egress-mac = keccak256.init((mac-secret ^ recipient-nonce) || auth)

ingress-mac = keccak256.init((mac-secret ^ initiator-nonce) || ack)

Recipient:

egress-mac = keccak256.init((mac-secret ^ initiator-nonce) || ack)

ingress-mac = keccak256.init((mac-secret ^ recipient-nonce) || auth)

When a frame is sent, the corresponding MAC values are computed by updating the

`egress-mac` state with the data to be sent. The update is performed by XORing the header

with the encrypted output of its corresponding MAC. This is done to ensure uniform

operations are performed for both plaintext MAC and ciphertext. All MACs are sent

cleartext.

header-mac-seed = aes(mac-secret, keccak256.digest(egress-mac)[:16]) ^ header-ciphertext

egress-mac = keccak256.update(egress-mac, header-mac-seed)

header-mac = keccak256.digest(egress-mac)[:16]

Computing `frame-mac`:

egress-mac = keccak256.update(egress-mac, frame-ciphertext)

frame-mac-seed = aes(mac-secret, keccak256.digest(egress-mac)[:16]) ^ keccak256.digest(egress-mac)[:16]

egress-mac = keccak256.update(egress-mac, frame-mac-seed)

frame-mac = keccak256.digest(egress-mac)[:16]

Verifying the MAC on ingress frames is done by updating the `ingress-mac` state in the

same way as `egress-mac` and comparing to the values of `header-mac` and `frame-mac` in

the ingress frame. This should be done before decrypting `header-ciphertext` and

`frame-ciphertext`.

# Capability Messaging

All messages following the initial handshake are associated with a 'capability'. Any

number of capabilities can be used concurrently on a single RLPx connection.

A capability is identified by a short ASCII name and version number. The capabilities

supported on either side of the connection are exchanged in the [Hello] message belonging

to the 'p2p' capability which is required to be available on all connections.

## Message Encoding

The initial [Hello] message is encoded as follows:

frame-data = msg-id || msg-data

frame-size = length of frame-data, encoded as a 24bit big-endian integer

where `msg-id` is an RLP-encoded integer identifying the message and `msg-data` is an RLP

list containing the message data.

All messages following Hello are compressed using the Snappy algorithm.

frame-data = msg-id || snappyCompress(msg-data)

frame-size = length of frame-data encoded as a 24bit big-endian integer

Note that the `frame-size` of compressed messages refers to the compressed size of

`msg-data`. Since compressed messages may inflate to a very large size after

decompression, implementations should check for the uncompressed size of the data before

decoding the message. This is possible because the [snappy format] contains a length

header. Messages carrying uncompressed data larger than 16 MiB should be rejected by

closing the connection.

## Message ID-based Multiplexing

While the framing layer supports a `capability-id`, the current version of RLPx doesn't

use that field for multiplexing between different capabilities. Instead, multiplexing

relies purely on the message ID.

Each capability is given as much of the message-ID space as it needs. All such

capabilities must statically specify how many message IDs they require. On connection and

reception of the [Hello] message, both peers have equivalent information about what

capabilities they share (including versions) and are able to form consensus over the

composition of message ID space.

Message IDs are assumed to be compact from ID 0x11 onwards (0x00-0x10 is reserved for the

"p2p" capability) and given to each shared (equal-version, equal-name) capability in

alphabetic order. Capability names are case-sensitive. Capabilities which are not shared

are ignored. If multiple versions are shared of the same (equal name) capability, the

numerically highest wins, others are ignored.

## "p2p" Capability

The "p2p" capability is present on all connections. After the initial handshake, both

sides of the connection must send either [Hello] or a [Disconnect] message. Upon receiving

the [Hello] message a session is active and any other message may be sent. Implementations

must ignore any difference in protocol version for forward-compatibility reasons. When

communicating with a peer of lower version, implementations should try to mimic that

version.

At any time after protocol negotiation, a [Disconnect] message may be sent.

### Hello (0x00)

`[protocolVersion: P, clientId: B, capabilities, listenPort: P, nodeKey: B\_64, ...]`

First packet sent over the connection, and sent once by both sides. No other messages may

be sent until a Hello is received. Implementations must ignore any additional list elements

in Hello because they may be used by a future version.

- `protocolVersion` the version of the "p2p" capability, \*\*5\*\*.

- `clientId` Specifies the client software identity, as a human-readable string (e.g.

"Ethereum(++)/1.0.0").

- `capabilities` is the list of supported capabilities and their versions:

`[[cap1, capVersion1], [cap2, capVersion2], ...]`.

- `listenPort` specifies the port that the client is listening on (on the interface that

the present connection traverses). If 0 it indicates the client is not listening.

- `nodeId` is the secp256k1 public key corresponding to the node's private key.

### Disconnect (0x01)

`[reason: P]`

Inform the peer that a disconnection is imminent; if received, a peer should disconnect

immediately. When sending, well-behaved hosts give their peers a fighting chance (read:

wait 2 seconds) to disconnect to before disconnecting themselves.

`reason` is an optional integer specifying one of a number of reasons for disconnect:

| Reason | Meaning |

|--------|:-------------------------------------------------------------|

| `0x00` | Disconnect requested |

| `0x01` | TCP sub-system error |

| `0x02` | Breach of protocol, e.g. a malformed message, bad RLP, ... |

| `0x03` | Useless peer |

| `0x04` | Too many peers |

| `0x05` | Already connected |

| `0x06` | Incompatible P2P protocol version |

| `0x07` | Null node identity received - this is automatically invalid |

| `0x08` | Client quitting |

| `0x09` | Unexpected identity in handshake |

| `0x0a` | Identity is the same as this node (i.e. connected to itself) |

| `0x0b` | Ping timeout |

| `0x10` | Some other reason specific to a subprotocol |

### Ping (0x02)

`[]`

Requests an immediate reply of [Pong] from the peer.

### Pong (0x03)

`[]`

Reply to the peer's [Ping] packet.

# Change Log

### Known Issues in the current version

- The frame encryption/MAC scheme is considered 'broken' because `aes-secret` and

`mac-secret` are reused for both reading and writing. The two sides of a RLPx connection

generate two CTR streams from the same key, nonce and IV. If an attacker knows one

plaintext, they can decrypt unknown plaintexts of the reused keystream.

- General feedback from reviewers has been that the use of a keccak256 state as a MAC

accumulator and the use of AES in the MAC algorithm is an uncommon and overly complex

way to perform message authentication but can be considered safe.

- The frame encoding provides `capability-id` and `context-id` fields for multiplexing

purposes, but these fields are unused.

### Version 5 (EIP-706, September 2017)

[EIP-706] added Snappy message compression.

### Version 4 (EIP-8, December 2015)

[EIP-8] changed the encoding of `auth-body` and `ack-body` in the initial handshake to

RLP, added a version number to the handshake and mandated that implementations should

ignore additional list elements in handshake messages and [Hello].

# References

- Elaine Barker, Don Johnson, and Miles Smid. NIST Special Publication 800-56A Section 5.8.1,

Concatenation Key Derivation Function. 2017.\

URL <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-56ar.pdf>

- Victor Shoup. A proposal for an ISO standard for public key encryption, Version 2.1. 2001.\

URL <http://www.shoup.net/papers/iso-2\_1.pdf>

- Mike Belshe and Roberto Peon. SPDY Protocol - Draft 3. 2014.\

URL <http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3>

- Snappy compressed format description. 2011.\

URL <https://github.com/google/snappy/blob/master/format\_description.txt>

Copyright &copy; 2014 Alex Leverington.

<a rel="license" href="http://creativecommons.org/licenses/by-nc-sa/4.0/">

This work is licensed under a

Creative Commons Attribution-NonCommercial-ShareAlike-4.0 International License</a>.

This repository contains specifications for the peer-to-peer networking protocols used by Ethereum.

Ethereum Node Records, our node metadata format.

Node Discovery Protocol v4

Node Discovery Protocol v5 (Draft Specification)

RLPx transport protocol (version 5) and several RLPx-based capabilities:

Ethereum Wire Protocol (version 65)

Ethereum Snapshot Protocol (version 1)

Light Ethereum Subprotocol (version 3)

Parity Light Protocol (version 1)

Ethereum Witness Protocol (version 0)

The issue tracker here is for discussions of protocol changes. It's OK to open an issue if you just have a question. You can also get in touch through our Gitter channel.

Protocol level security issues are valuable! Please report serious issues responsibly through the Ethereum Foundation Bounty Program.

The Mission

devp2p is a set of network protocols which form the Ethereum peer-to-peer network. 'Ethereum network' is meant in a broad sense, i.e. devp2p isn't specific to a particular blockchain, but should serve the needs of any networked application associated with the Ethereum umbrella.

We aim for an integrated system of orthogonal parts, implemented in multiple programming environments. The system provides discovery of other participants throughout the Internet as well as secure communication with those participants.

The network protocols in devp2p should be easy to implement from scratch given only the specification, and must work within the limits of a consumer-grade Internet connection. We usually design protocols in a 'specification first' approach, but any specification proposed must be accompanied by a working prototype or implementable within reasonable time.

Relationship with libp2p

The libp2p project was started at about the same time as devp2p and seeks to be a collection of modules for assembling a peer-to-peer network from modular components. Questions about the relationship between devp2p and libp2p come up rather often.

It's hard to compare the two projects because they have different scope and are designed with different goals in mind. devp2p is an integrated system definition that wants to serve Ethereum's needs well (although it may be a good fit for other applications, too) while libp2p is a collection of programming library parts serving no single application in particular.

That said, both projects are very similar in spirit and devp2p is slowly adopting parts of libp2p as they mature.

Implementations

devp2p is part of most Ethereum clients. Implementations include:

C#: Nethermind https://github.com/NethermindEth/nethermind

C++: Aleth https://github.com/ethereum/aleth

C: Breadwallet https://github.com/breadwallet/breadwallet-core

Elixir: Exthereum https://github.com/exthereum/ex\_wire

Go: go-ethereum/geth https://github.com/ethereum/go-ethereum

Java: Tuweni RLPx library https://github.com/apache/incubator-tuweni/tree/master/rlpx

Java: Besu https://github.com/hyperledger/besu

JavaScript: EthereumJS https://github.com/ethereumjs/ethereumjs-devp2p

Kotlin: Tuweni Discovery library https://github.com/apache/incubator-tuweni/tree/master/devp2p

Nim: Nimbus nim-eth https://github.com/status-im/nim-eth

Python: Trinity https://github.com/ethereum/trinity

Ruby: Ciri https://github.com/ciri-ethereum/ciri

Ruby: ruby-devp2p https://github.com/cryptape/ruby-devp2p

Rust: rust-devp2p https://github.com/rust-ethereum/devp2p

Rust: openethereum https://github.com/openethereum/openethereum

WireShark dissectors are available here: https://github.com/ConsenSys/ethereum-dissectors

Releases (No releases published) 2/20/2021

Ethereum Wire Protocol (ETH)

'eth' is a protocol on the [RLPx] transport that facilitates exchange of Ethereum

blockchain information between peers. The current protocol version is \*\*eth/65\*\*. See end

of document for a list of changes in past protocol versions.

### Basic Operation

Once a connection is established, a [Status] message must be sent. Following the reception

of the peer's Status message, the Ethereum session is active and any other message may be

sent.

Within a session, three high-level tasks can be performed: chain synchronization, block

propagation and transaction exchange. These tasks use disjoint sets of protocol messages

and clients typically perform them as concurrent activities on all peer connections.

Client implementations should enforce limits on protocol message sizes. The underlying

RLPx transport limits the size of a single message to 16.7 MiB. The practical limits for

the eth protocol are lower, typically 10 MiB. If a received message is larger than the

limit, the peer should be disconnected.

In addition to the hard limit on received messages, clients should also impose 'soft'

limits on the requests and responses which they send. The recommended soft limit varies

per message type. Limiting requests and responses ensures that concurrent activity, e.g.

block synchronization and transaction exchange work smoothly over the same peer

connection.

### Chain Synchronization

Nodes participating in the eth protocol are expected to have knowledge of the complete

chain of all blocks from the genesis block to current, latest block. The chain is obtained

by downloading it from other peers.

Upon connection, both peers send their [Status] message, which includes the Total

Difficulty (TD) and hash of their 'best' known block.

The client with the worst TD then proceeds to download block headers using the

[GetBlockHeaders] message. It verifies proof-of-work values in received headers and

fetches block bodies using the [GetBlockBodies] message. Received blocks are executed

using the Ethereum Virtual Machine, recreating the state tree and receipts.

Note that header downloads, block body downloads and block execution may happen

concurrently.

### State Synchronization (a.k.a. "fast sync")

Protocol versions eth/63 and later also allow synchronizing transaction execution results

(i.e. state tree and receipts). This may be faster than re-executing all historical

transactions but comes at the expense of some security.

State synchronization typically proceeds by downloading the chain of block headers,

verifying their proof-of-work values. Block bodies are requested as in the Chain

Synchronization section but block transactions aren't executed. Instead, the client picks

a block near the head of the chain and downloads merkle tree nodes and contract code

incrementally by requesting the root node, its children, grandchildren, ... using

[GetNodeData] until the entire tree is synchronized.

### Block Propagation

Newly-mined blocks must be relayed to all nodes. This happens through block propagation,

which is a two step process. When a [NewBlock] announcement message is received from a

peer, the client first verifies the basic validity of the block and checks that the

proof-of-work value is valid. It then sends the block to a small fraction of connected

peers (usually the square root of the total number of peers) using the [NewBlock] message.

After the proof-of-work check, the client imports the block into its local chain by

executing all transactions contained in the block, computing the block's 'post state'. The

block's state root hash must match the computed post state root. Once the block is fully

processed the client sends a [NewBlockHashes] message about the block to all peers which

it didn't notify earlier. Those peers may request the full block later if they fail to

receive it via [NewBlock] from anyone else.

A node should never send a block announcement back to a peer which previously announced

the same block. This is usually achieved by remembering a large set of block hashes

recently relayed to or from each peer.

The reception of a block announcement may also trigger chain synchronization if the block

is not the immediate successor of the client's current latest block.

### Transaction Exchange

All nodes must exchange pending transactions in order to relay them to miners, which will

pick them for inclusion into the blockchain. Client implementations keep track of the set

of pending transactions in the 'transaction pool'. The pool is subject to client-specific

limits and can contain many (i.e. thousands of) transactions.

When a new peer connection is established, the transaction pools on both sides need to be

synchronized. Initially, both ends should send [NewPooledTransactionHashes] messages

containing all transaction hashes in the local pool to start the exchange.

On receipt of a NewPooledTransactionHashes announcement, the client filters the received

set, collecting transaction hashes which it doesn't yet have in its own local pool. It can

then request the transactions using the [GetPooledTransactions] message.

When new transactions appear in the client's pool, it should propagate them to the network

using the [Transactions] and [NewPooledTransactionHashes] messages. The Transactions

message relays complete transaction objects and is typically sent to a small, random

fraction of connected peers. All other peers receive a notification of the transaction

hash and can request the complete transaction object if it is unknown to them. The

dissemination of complete transactions to a fraction of peers usually ensures that all

nodes receive the transaction and won't need to request it.

A node should never send a transaction back to a peer that it can determine already knows

of it (either because it was previously sent or because it was informed from this peer

originally). This is usually achieved by remembering a set of transaction hashes recently

relayed by the peer.

Transactions must be validated before re-propagating them. Relaying an invalid transaction

results in peer disconnection.

## Protocol Messages

### Status (0x00)

`[protocolVersion: P, networkId: P, td: P, bestHash: B\_32, genesisHash: B\_32, forkID]`

Inform a peer of its current state. This message should be sent just after the connection

is established and prior to any other eth protocol messages.

- `protocolVersion`: the current protocol version

- `networkId`: Integer identifying the blockchain, see table below

- `td`: total difficulty of the best chain. Integer, as found in block header.

- `bestHash`: The hash of the best (i.e. highest TD) known block.

- `genesisHash`: The hash of the Genesis block.

- `number`: The block number of the latest block in the chain.

- `forkID`: An [EIP-2124] fork identifier, encoded as `[forkHash, forkNext]`.

This table lists common Network IDs and their corresponding networks. Other IDs exist

which aren't listed, i.e. clients should not require that any particular network ID is

used. Note that the Network ID may or may not correspond with the EIP-155 Chain ID used

for transaction replay prevention.

| ID | chain |

|----|-------------------------------|

| 0 | Olympic (disused) |

| 1 | Frontier (now mainnet) |

| 2 | Morden (disused) |

| 3 | Ropsten (current PoW testnet) |

| 4 | [Rinkeby] |

For a community curated list of chain IDs, see <https://chainid.network>.

### NewBlockHashes (0x01)

`[[hash\_0: B\_32, number\_0: P], [hash\_1: B\_32, number\_1: P], ...]`

Specify one or more new blocks which have appeared on the network. To be maximally

helpful, nodes should inform peers of all blocks that they may not be aware of. Including

hashes that the sending peer could reasonably be considered to know (due to the fact they

were previously informed of because that node has itself advertised knowledge of the

hashes through NewBlockHashes) is considered bad form, and may reduce the reputation of

the sending node. Including hashes that the sending node later refuses to honour with a

proceeding [GetBlockHeaders] message is considered bad form, and may reduce the reputation

of the sending node.

### Transactions (0x02)

`[[nonce: P, receivingAddress: B\_20, value: P, ...], ...]`

Specify transactions that the peer should make sure is included on its transaction queue.

The items in the list are transactions in the format described in the main Ethereum

specification. Transactions messages must contain at least one (new) transaction, empty

Transactions messages are discouraged and may lead to disconnection.

Nodes must not resend the same transaction to a peer in the same session and must not

relay transactions to a peer they received that transaction from. In practice this is

often implemented by keeping a per-peer bloom filter or set of transaction hashes which

have already been sent or received.

### GetBlockHeaders (0x03)

`[block: {P, B\_32}, maxHeaders: P, skip: P, reverse: P in {0, 1}]`

Require peer to return a [BlockHeaders] message. Reply must contain a number of block

headers, of rising number when `reverse` is `0`, falling when `1`, `skip` blocks apart,

beginning at block `block` (denoted by either number or hash) in the canonical chain, and

with at most `maxHeaders` items.

### BlockHeaders (0x04)

`[blockHeader\_0, blockHeader\_1, ...]`

Reply to [GetBlockHeaders]. The items in the list (following the message ID) are block

headers in the format described in the main Ethereum specification, previously asked for

in a GetBlockHeaders message. This may validly contain no block headers if none of the

requested block headers were found. The number of headers that can be requested in a

single message may be subject to implementation-defined limits.

The recommended soft limit for BlockHeaders responses is 2 MiB.

### GetBlockBodies (0x05)

`[hash\_0: B\_32, hash\_1: B\_32, ...]`

Require peer to return a [BlockBodies] message. Specify the set of blocks that we're

interested in with the hashes. The number of blocks that can be requested in a single

message may be subject to implementation-defined limits.

### BlockBodies (0x06)

`[[transactions\_0, uncles\_0] , ...]`

Reply to [GetBlockBodies]. The items in the list are some of the blocks, minus the header,

in the format described in the main Ethereum specification, previously asked for in a

GetBlockBodies message. This may be empty if no blocks were available for the last

GetBlockBodies query.

The recommended soft limit for BlockBodies responses is 2 MiB.

### NewBlock (0x07)

`[[blockHeader, transactionList, uncleList], totalDifficulty]`

Specify a single block that the peer should know about. The composite item in the list

(following the message ID) is a block in the format described in the main Ethereum

specification.

- `totalDifficulty` is the total difficulty of the block (aka score).

### NewPooledTransactionHashes (0x08)

`[hash\_0: B\_32, hash\_1: B\_32, ...]`

This message announces one or more transactions that have appeared in the network and

which have not yet been included in a block. To be maximally helpful, nodes should inform

peers of all transactions that they may not be aware of.

The recommended soft limit for this message is 4096 hashes (128 KiB).

Nodes should only announce hashes of transactions that the remote peer could reasonably be

considered not to know, but it is better to return more transactions than to have a nonce

gap in the pool.

### GetPooledTransactions (0x09)

`[hash\_0: B\_32, hash\_1: B\_32, ...]`

This message requests transactions from the recipient's transaction pool by hash.

The recommended soft limit for GetPooledTransactions requests is 256 hashes (8 KiB). The

recipient may enforce an arbitrary limit on the response (size or serving time), which

must not be considered a protocol violation.

### PooledTransactions (0x0a)

`[[nonce: P, receivingAddress: B\_20, value: P, ...], ...]`

This is the response to GetPooledTransactions, returning the requested transactions from

the local pool. The items in the list are transactions in the format described in the main

Ethereum specification.

The transactions must be in same order as in the request, but it is OK to skip

transactions which are not available. This way, if the response size limit is reached,

requesters will know which hashes to request again (everything starting from the last

returned transaction) and which to assume unavailable (all gaps before the last returned

transaction).

It is permissible to first announce a transaction via NewPooledTransactionHashes, but then

to refuse serving it via PooledTransactions. This situation can arise when the transaction

is included in a block (and removed from the pool) in between the announcement and the

request.

A peer may respond with an empty list iff none of the hashes match transactions in its

pool.

### GetNodeData (0x0d)

`[hash\_0: B\_32, hash\_1: B\_32, ...]`

Require peer to return a [NodeData] message containing state tree nodes or contract code

matching the requested hashes.

### NodeData (0x0e)

`[value\_0: B, value\_1: B, ...]`

Provide a set of state tree nodes or contract code blobs which correspond to previously

requested hashes from [GetNodeData]. Does not need to contain all; best effort is fine. This

message may be an empty list if the peer doesn't know about any of the previously

requested hashes. The number of items that can be requested in a single message may be

subject to implementation-defined limits.

The recommended soft limit for NodeData responses is 2 MiB.

### GetReceipts (0x0f)

`[blockHash\_0: B\_32, blockHash\_1: B\_32, ...]`

Require peer to return a [Receipts] message containing the receipts of the given block

hashes. The number of receipts that can be requested in a single message may be subject to

implementation-defined limits.

### Receipts (0x10)

`[[receipt\_0, receipt\_1], ...]`

Provide a set of receipts which correspond to block hashes in a previous [GetReceipts]

message.

The recommended soft limit for Receipts responses is 2 MiB.

## Change Log

### eth/65 ([EIP-2464], January 2020)

Version 65 improved transaction exchange, introducing three additional messages:

[NewPooledTransactionHashes], [GetPooledTransactions], and [PooledTransactions].

Prior to version 65, peers always exchanged complete transaction objects. As activity and

transaction sizes increased on the Ethereum mainnet, the network bandwidth used for

transaction exchange became a significant burden on node operators. The update reduced the

required bandwidth by adopting a two-tier transaction broadcast system similar to block

propagation.

### eth/64 ([EIP-2364], November 2019)

Version 64 changed the [Status] message to include the [EIP-2124] ForkID. This allows

peers to determine mutual compatibility of chain execution rules without synchronizing the

blockchain.

### eth/63 (2016)

Version 63 added the [GetNodeData], [NodeData], [GetReceipts] and [Receipts] messages

which allow synchronizing transaction execution results.

### eth/62 (2015)

In version 62, the [NewBlockHashes] message was extended to include block numbers

alongside the announced hashes. The block number in [Status] was removed. Messages

GetBlockHashes (0x03), BlockHashes (0x04), GetBlocks (0x05) and Blocks (0x06) were

replaced by messages that fetch block headers and bodies. The BlockHashesFromNumber (0x08)

message was removed.

Previous encodings of the reassigned/removed message codes were:

- GetBlockHashes (0x03): `[hash: B\_32, maxBlocks: P]`

- BlockHashes (0x04): `[hash\_0: B\_32, hash\_1: B\_32, ...]`

- GetBlocks (0x05): `[hash\_0: B\_32, hash\_1: B\_32, ...]`

- Blocks (0x06): `[[blockHeader, transactionList, uncleList], ...]`

- BlockHashesFromNumber (0x08): `[number: P, maxBlocks: P]`

### eth/61 (2015)

Version 61 added the BlockHashesFromNumber (0x08) message which could be used to request

blocks in ascending order. It also added the latest block number to the [Status] message.

### eth/60 and below

Version numbers below 60 were used during the Ethereum PoC development phase.

- `0x00` for PoC-1

- `0x01` for PoC-2

- `0x07` for PoC-3

- `0x09` for PoC-4

- `0x17` for PoC-5

- `0x1c` for PoC-6

[RLPx]: ../rlpx.md

[Status]: #status-0x00

[NewBlockHashes]: #newblockhashes-0x01

[Transactions]: #transactions-0x02

[GetBlockHeaders]: #getblockheaders-0x03

[BlockHeaders]: #blockheaders-0x04

[GetBlockBodies]: #getblockbodies-0x05

[BlockBodies]: #blockbodies-0x06

[NewBlock]: #newblock-0x07

[NewPooledTransactionHashes]: #newpooledtransactionhashes-0x08

[GetPooledTransactions]: #getpooledtransactions-0x09

[PooledTransactions]: #pooledtransactions-0x0a

[GetNodeData]: #getnodedata-0x0d

[NodeData]: #nodedata-0x0e

[GetReceipts]: #getreceipts-0x0f

[Receipts]: #receipts-0x10

[Rinkeby]: https://rinkeby.io

[EIP-2124]: https://eips.ethereum.org/EIPS/eip-2124

[EIP-2364]: https://eips.ethereum.org/EIPS/eip-2364

[EIP-2464]: https://eips.ethereum.org/EIPS/eip-2464

Light Ethereum Subprotocol (LES)

The Light Ethereum Subprotocol (LES) is the protocol used by "light" clients, which only

download block headers as they appear and fetch other parts of the blockchain on-demand.

They provide full functionality in terms of safely accessing the blockchain, but do not

mine and therefore do not take part in the consensus process. Full and archive nodes can

also support the 'les' protocol besides 'eth' in order to be able to serve light nodes.

The current protocol version is \*\*les/3\*\*. See end of document for a list of changes in

past protocol versions. Some of the les protocol messages are similar to of the [Ethereum

Wire Protocol], with the addition of a few new fields.

## Canonical Hash Trie

Canonical Hash Trie (CHT) structures are used by LES for quick initial syncing and secure

on-demand retrieval of canonical hash mappings, block headers and total difficulty (TD)

values.

A CHT is a Merkle trie (specifically '[Merkle Patricia Trie]' as used for Ethereum state)

that contains `blockNumber -> [blockHash, TD]` mappings where keys are binary big endian

encoded 64 bit integers and values are RLP-encoded `[hash, number]` pairs.

CHTs are generated by LES servers for every 32768 blocks, `CHT[i]` containing data for

blocks `0..(i+1) \* 32768 - 1`. If a client knows the root hash of `CHT[i]` and wants to fetch

header number `N` (where `N < (i+1) \* 32768`), it can obtain the header and the corresponding

Merkle proof of the CHT with a [GetHelperTrieProofs] request.

CHTs are only generated after 2048 confirmations, making it sure they will not be changed

by a chain reorg. In the current version of the light client there is a hard-coded

`[chtNumber, chtRoot]` pair associated with the genesis block hash of both the mainnet and

the testnet. A trustless validation algorithm is planned for later protocol versions.

## BloomBits

The BloomBits data structure optimizes log searching by doing a bitwise transformation

that makes it cheaper to retrieve bloom filter data relevant to a specific filter.

When searching in a long section of the block history, we are checking three specific bits

of each bloom filter per queried address/topic. In order to do that, LES must retrieve a

~550 byte block header per filtered block.

The BloomBits structure optimizes bloom filter lookups through a "bitwise 90 degree

rotation" of the bloom filters. Blocks are grouped into fixed length sections (section

size for the LES BloomBits Trie is 32768 blocks), `BloomBits[bitIdx][sectionIdx]` is a

32768 bit (4096 byte) long bit vector that contains a single bit of each bloom filter from

the block range `sectionIdx\*SectionSize ... (sectionIdx+1)\*SectionSize-1`. Since bloom

filters are usually sparse, a simple data compression makes this structure even more

efficient, especially for on-demand retrieval. By reading and binary AND-ing three

BloomBits sections, we can filter for an address/topic in 32768 blocks at once ("1" bits

in the binary AND result mean bloom matches).

### Compression Algorithm

BloomBits data is stored in compressed form. The compression algorithm is optimized for

sparse input data which contains a lot of zero bytes. Decompression requires knowledge of

the decompressed data length.

The algorithm can be described with this pseudo-code:

if data only contains zeroes,

CompressBytes(data) == nil

otherwise if len(data) <= 1,

CompressBytes(data) == data

otherwise:

CompressBytes(data) == append(CompressBytes(nonZeroBitset(data)), nonZeroBytes(data)...)

where

nonZeroBitset(data) is a bit vector with len(data) bits (MSB first):

nonZeroBitset(data)[i/8] && (1 << (7-i%8)) != 0 if data[i] != 0

len(nonZeroBitset(data)) == (len(data)+7)/8

nonZeroBytes(data) contains the non-zero bytes of data in the same order

### BloomBits Trie

In order to make this data structure retrievable on-demand for the light client, we put

the generated vectors in a trie. Parts of this trie can be retrieved with the

[GetHelperTrieProofs] message. Currently the trie root is part of the trusted syncing

checkpoint but trustless validation of the BloomBits trie is part of the development

plans. The trie consists of the compressed bit vectors as values stored at keys

constructed from the the bloom bit index encoded as a 2-byte big endian, followed by the

section index encoded as an 8-byte big endian. Since all-zero bit vectors have a zero

length when compressed, these vectors are not added to the trie at all.

BloomBits tries are generated for each new section of transformed bloom filter data by

adding the vectors belonging to the latest section index to the previous trie.

## Client Side Flow Control

Any node which takes on a server role in the the LES protocol needs to be able to somehow

limit the amount of work it does for each client peer during a given time period. They can

always just serve requests slowly if they are overloaded, but it is beneficial to give

some sort of flow control feedback to the clients. This way, clients could (and would have

incentive to) behave nicely and not send requests too quickly in the first place (and then

possibly timeout and resend while the server is still working on them). They could also

distribute requests better between multiple servers they are connected to. And if clients

can do this, servers can expect them to do this and throttle or drop them if they break

the flow control rules.

### The Model

Let us assume that serving each request has a cost (depending on type and parameters) for

the server. This cost is determined by the server, but it has an upper limit for any valid

request. The server assigns a "buffer" for each client from which the cost of each request

is deduced. The buffer has an upper limit (the "buffer limit") and a recharge rate (cost

per second). The server can decide to recharge it more quickly at any time if it has more

free resources, but there is a guaranteed minimum recharge rate. If a request is received

that would drain the client's buffer below zero, the client has broken the flow control

rules and is throttled or disconnected.

### The Protocol

The server announces three parameters in the [Status] message:

- `"flowControl/BL"`: Buffer Limit, an integer value

- `"flowControl/MRR"`: Minimum Rate of Recharge, an integer value

- `"flowControl/MRC"`: Maximum Request Cost table. The value of this parameter is a

table assigning cost values to every on-demand retrieval message in the LES protocol.

The table is encoded as a list of integer triples: `[[MsgCode, BaseCost, ReqCost], ...]`

On the server side:

When a client connects, the server sets the initial Buffer Value (`BV`) of the client to

`BL` and announces `BL` in [Status]. When a request is received from the client, it

calculates the cost according to its own estimates (but not higher than `MaxCost`, which

equals `BaseCost + ReqCost \* N`, where `N` is the number of individual elements asked in

the request), then deducts it from `BV`. If `BV` goes negative, drops the peer, otherwise

starts serving the request. The reply message contains a `BV` value that is the previously

calculated `BV` plus the amount recharged during the time spent serving. Note that since

the server can always determine any cost up to `MaxCost` for a request (and a client

should not assume otherwise), it can reject a message without processing it if received

while `BV < MaxCost` because that's already a flow control breach.

On the client side:

The client always has a lowest estimate for its current `BV`, called `BLE`. It

- sets `BLE` to `BL` received in [Status]

- doesn't send any request to the server when `BLE < MaxCost`

- deduces `MaxCost` when sending a request

- recharges `BLE` at the rate of `MRR` when less than `BL`

When a reply message with a new `BV` value is received, it sets `BLE` to `BV -

Sum(MaxCost)`, summing the `MaxCost` values of requests sent after the one belonging to

this reply.

#### Buffer underrun

Before \*\*les/3\*\* buffer underruns always resulted in immediate disconnection. Now it is

possible and recommended to send a [StopMsg] instead and then a [ResumeMsg] when the

buffer has been at least partially recharged. This allows clients to treat the buffer

feedback as an optional performance optimization hint instead of a mandatory mechanism

and allows simple implementations that do not care about the buffer at all.

## Request ID

Every on-demand request message contains a `reqID` field, which is simply returned by the

server in the corresponding reply message. This helps matching replies for requests on the

client side so that each reply doesn't need to be matched against each pending request.

## Protocol Messages

### Status (0x00)

`[[key\_0, value\_0], [key\_1, value\_1], ...]`

Inform a peer of the sender's current LES state. This message should be sent just after

the connection is established and prior to any other LES messages. The following keys

are required (value types are noted after the key string):

- `"protocolVersion"` `P`: is 1 for protocol version one.

- `"networkId"` `P`: should be 0 for testnet, 1 for mainnet.

- `"headTd"` `P`: Total Difficulty of the best chain. Integer, as found in block header.

- `"headHash"` `B\_32`: the hash of the best (i.e. highest TD) known block.

- `"headNum"` `P`: the number of the best (i.e. highest TD) known block.

- `"genesisHash"` `B\_32`: the hash of the Genesis block.

There are several optional key/value pairs which can be set:

- `"announceType"` `P`: set by clients, this field affects the [Announce] messages of the

server. Allowed integer values are:

- none (`0`): no [Announce] messages are sent, i.e. the client is not interested in announcements.

- simple (`1`): Default. [Announce] messages use the \*\*les/1\*\* format.

- signed (`2`): there is a `"sign"` key in the key/value list of [Announce] messages. The

associated value is a signature of an RLP encoded `[headHash: B\_32, headNumber: P, headTd: P]`

structure by the server's node key.

- `"serveHeaders"` (empty value): present if the peer can serve header chain downloads.

- `"serveChainSince"` `P`: present if the peer can serve Body/Receipts ODR requests

starting from the given block number.

- `"serveRecentChain"` `P`: if present then the availability of chain data is only guaranteed

for the given number of recent blocks. If the node serves chain data then `"serveChainSince"`

should always be present while `"serveRecentChain"` is optional. Chain availability can

be assumed for blocks with `blockNumber >= MAX(serveChainSince, headNumber-serveRecentChain+1)`.

- `"serveStateSince"` `P`: present if the peer can serve Proof/Code ODR requests starting

from the given block number.

- `"serveRecentState"` `P`: if present then the availability of state data is only guaranteed

for the given number of recent blocks. If the node serves state data then `"serveStateSince"`

should always be present while `"serveRecentState"` is optional. State availability can

be assumed for blocks with `blockNumber >= MAX(serveStateSince, headNumber-serveRecentState+1)`.

- `"txRelay"` (no value): present if the peer can relay transactions to the ETH network.

- `"flowControl/BL"`, `"flowControl/MRC"`, `"flowControl/MRR"`: see [Client Side Flow Control]

Unknown keys should be ignored by both sides. This allows announcing additional

capabilities while staying compatible with past protocol versions.

### Announce (0x01)

`[headHash: B\_32, headNumber: P, headTd: P, reorgDepth: P, [[key\_0, value\_0], [key\_1, value\_1], ...]]`

Announce a new chain head and optionally also a change to some of the values announced at

handshake. A restrictive change of server capabilities (for example, an increase of

`"serveStateSince"` due to state pruning) should be announced at least 10 seconds prior to

actually restricting those capabilities in order to avoid asynchronous problems. Changes

to unknown keys should be ignored. Changes to known keys that make no sense lead to

disconnection.

Announcing a head with a lower or equal TD than previously announced or a head that the

sending node later refuses to honor with a proceeding [GetBlockHeaders] message (with

number and TD also matching) is considered bad form, and may lead to disconnection or

reduce the reputation of the sending node.

The field `reorgDepth` contains the number of blocks to be rolled back from the last head

announced by the same node in order to find the last common ancestor of the last and

current heaviest chain. Adding this field helps the client to minimize the number of

requests and the amount of bandwidth required to fetch new headers.

### GetBlockHeaders (0x02)

`[reqID: P, [block: {P, B\_32}, maxHeaders: P, skip: P, reverse: P in {0, 1}]]`

Require peer to return a [BlockHeaders] message. Reply must contain a number of block

headers, of rising number when `reverse` is `0`, falling when `1`, `skip` blocks apart,

beginning at block `block` (denoted by either number or hash) in the canonical chain, and

with at most `maxHeaders` items.

### BlockHeaders (0x03)

`[reqID: P, BV: P, [blockHeader\_0, blockHeader\_1, ...]]`

Reply to [GetBlockHeaders]. The items in the list (following the message ID) are block

headers in the format described in the main Ethereum specification, previously asked for

in a [GetBlockHeaders] message. The list may be empty if none of the requested block

headers were available on the server side.

### GetBlockBodies (0x04)

`[reqID: P, [hash\_0: B\_32, hash\_1: B\_32, ...]]`

Require peer to return a [BlockBodies] message. Specify the set of blocks that we're

interested in with the hashes.

### BlockBodies (0x05)

`[reqID: P, BV: P, [[transactions\_0, uncles\_0] , ...]]`

Reply to [GetBlockBodies]. The items in the list (following the message ID) are some of

the blocks, minus the header, in the format described in the main Ethereum specification,

previously asked for in a [GetBlockBodies] message.

### GetReceipts (0x06)

`[reqID: P, [hash\_0: B\_32, hash\_1: B\_32, ...]]`

Require peer to return a [Receipts] message.

### Receipts (0x07)

`[reqID: P, BV: P, [[receipt\_0, receipt\_1, ...], ...]]`

Provide a set of receipts which correspond to the block hashes previously asked for in

[GetReceipts].

### GetProofs (0x08)

`[reqID: P, [[blockhash: B\_32, key: B\_32, key2: B\_32, fromLevel: P], ...]]`

Require peer to return a [Proofs] message, containing one or more Merkle proofs, each

proving the value of index `key` from the state trie of the given block (if `key2` is

empty), or the storage value of index `key2` from the storage trie referenced in the

account at `key`. If `fromLevel` is greater than zero, the given number of trie nodes

closest to the root can be omitted from the proof.

This message was deprecated in \*\*les/2\*\*, use [GetProofsV2] instead.

### Proofs (0x09)

`[reqID: P, BV: P, [[node\_1, node\_2, ...], ...]]`

Return a set of Merkle proofs, each consisting of a set of nodes that must be processed in

order to access the trie entry value (or prove the absence of an entry) requested in

[GetProofs].

### GetContractCodes (0x0a)

`[reqID: P, [[blockhash: B\_32, key: B\_32], ...]]`

Require peer to return a [ContractCodes] message.

### ContractCodes (0x0b)

`[reqID: P, BV: P, [value\_0: B, value\_1: B, ...]]`

Provide a set of contract codes which correspond to the block hashes and account keys

previously asked in [GetContractCodes].

### GetHeaderProofs (0x0d)

`[reqID: P, [[chtNumber: P, blockNumber: P, fromLevel: P], ...]]`

Require peer to return a [HeaderProofs] message, containing one or more canonical block

headers (of block number `blockNumber`) and corresponding Merkle proofs of the [CHT]

(Canonical Hash Trie) identified by `chtNumber`. If `fromLevel` is greater than zero, the

given number of trie nodes closest to the root can be omitted from the proof.

This message was deprecated in \*\*les/2\*\*, use [GetHelperTrieProofs] instead.

### HeaderProofs (0x0e)

`[reqID: P, BV: P, [[blockHeader, [node\_1, node\_2...]], ...]]`

Return a set of structures, each containing a block header and a Merkle proof proving the

header hash and belonging TD against a given CHT requested in [GetHeaderProofs].

### SendTx (0x0c)

`[txdata\_1, txdata\_2, ...]`

Require peer to add a set of transactions into its transaction pool and relay them to the

ETH network.

This message was deprecated in \*\*les/2\*\*, use [SendTxV2] instead.

### GetProofsV2 (0x0f)

`[reqID: P, [[blockhash: B\_32, key: B\_32, key2: B\_32, fromLevel: P], ...]]`

Require peer to return a [ProofsV2] message, containing a single (and smallest possible)

set of trie nodes that proves for each request the value of index `key` from the state

trie of the given block (if `key2` is empty), or the storage value of index `key2` from

the storage trie referenced in the account at `key`. If `fromLevel` is greater than zero,

the given number of trie nodes closest to the root can be omitted from the proof.

### ProofsV2 (0x10)

`[reqID: P, BV: P, [node\_1, node\_2, ...]]`

Return the smallest set of trie nodes required to access the trie entry value (or prove

the absence of an entry) requested in [GetProofsV2]. This set will be called a \*proof

set\*. Compared to [Proofs], this message contains a single list of nodes satisfying all

requested proofs. The list shouldn't contain duplicate nodes.

### GetHelperTrieProofs (0x11)

`[reqID: P, [[subType: P, sectionIdx: P, key: B, fromLevel: P, auxReq: P], ...]]`

Require peer to return a [HelperTrieProofs] message, containing a \*proof set\* and optional

auxiliary data for each request.

Note: this request is a generalization of the \*\*les/1\*\* [GetHeaderProofs] message. It

retrieves Merkle proofs from different types of "helper tries" which are generated for

every fixed-length section of the canonical chain. `subType` identifies the helper trie

that is being requested for the section marked by `sectionIdx`. `key` and `fromLevel` are

interpreted like in case of proof requests.

If `auxReq` is greater than zero then auxiliary data is requested too. If `auxReq` is 1

then the root hash of the specified trie (according to the server) is returned and no trie

nodes are added to the proof set. This special request will be required for trustless

validation of helper tries. The interpretation of `auxReq` values greater than 1 is

subject to `subType`.

The following `subType` integer values are allowed in \*\*les/2\*\*:

- CHT (`0`): request a key from the [Canonical Hash Trie]. If `auxReq` is 2 then the

belonging header is returned as `auxData`. `key` is the block number encoded as an

8-byte big endian. Note that the section size for CHTs has been raised to 32k instead of

4k blocks so for example a `sectionIdx` of 100 equals a `chtNumber` of 807 in case of

the \*\*les/1\*\* [GetHeaderProofs] message.

- BloomBits (`1`): request a key from the [BloomBits Trie]. In this trie `key` is 10 bytes

long, it consists of the bloom bit index encoded as a 2-byte big endian, followed by the

section index encoded as an 8-byte big endian. The returned value is the corresponding

compressed bloom bit vector.

### HelperTrieProofs (0x12)

`[reqID: P, BV: P, [[node\_1, node\_2...], [auxData\_0, auxData\_1, ...]]]`

Return a proof set and a set of `auxData` requested in [GetHelperTrieProofs]. The length

of the `auxData` list equals the number of requests with a non-zero `auxReq`.

### SendTxV2 (0x13)

`[reqID: P, [txdata\_1, txdata\_2, ...]]`

Require peer to add a set of transactions into its transaction pool and relay them to the

ETH network, then return a [TxStatus] message containing the status of the sent

transactions.

### GetTxStatus (0x14)

`[reqID: P, [txHash\_1, txHash\_2, ...]]`

Require peer to return a [TxStatus] message containing the status of the referenced

transactions. This message is intended for inquiry about past transactions sent by the

client. Note that the server is not required to make every transaction available

indefinitely.

### TxStatus (0x15)

`[reqID: P, BV: P, [[status: P, data: B], ...]]`

Return the current status of the sent/queried transactions. Possible `status` values are:

- Unknown (`0`): transaction is unknown

- Queued (`1`): transaction is queued (not processable yet)

- Pending (`2`): transaction is pending (processable)

- Included (`3`): transaction is already included in the canonical chain. `data` contains

an RLP-encoded `[blockHash: B\_32, blockNumber: P, txIndex: P]` structure.

- Error (`4`): transaction sending failed. `data` contains a text error message.

### StopMsg (0x16)

Instruct the client to temporarily stop sending requests and to not expect responses to those requests it did not already receive a reply for.

Implementer's note: this message can be used to handle transient server overloads or individual client flow control buffer underruns. The server should avoid sending [StopMsg] too often though if the client also avoids buffer underruns. It should try to regulate its own utilization (and thereby also the frequency of transient overload occurences) with the flow control feedback. Receiving [StopMsg] more than once every few minutes in long term average or not receiving [ResumeMsg] in a few seconds can be considered bad service quality by the clients.

### ResumeMsg (0x17)

`[BV: P]`

Update flow control buffer and allow sending requests again. Note that the requests not answered before [StopMsg] were permanently canceled and will not be answered after [ResumeMsg]. If a [ResumeMsg] is received without a preceding [StopMsg] then it should be treated as a simple flow control buffer update (assuming that the server has already deducted the cost of the previously answered messages).

## Change Log

### les/3 (May 2019)

- Keys `"serveRecentChain"` and `"serveRecentState"` were added to the [Status] message.

- Messages [StopMsg] and [ResumeMsg] were added to improve handling transient overloads

and flow control buffer underruns.

### les/2 (November 2017)

- The `"announceType"` key was added to the [Status] message.

- The BloomBits Trie and associated messages [GetHelperTrieProofs], [HelperTrieProofs]

were added to facilitate server-assisted log search. \*\*les/1\*\* clients would frequently

download large ranges of receipts to search for specific logs.

- Messages [GetProofsV2], [ProofsV2] were added to de-duplicate result nodes when

requesting multiple proofs at the same time.

- Messages [SendTxV2], [GetTxStatus] and [TxStatus] were added to allow querying for past

transactions and to enable user-lever error reporting for non-includable transactions at

the time of submission.

- The [GetHeaderProofs], [HeaderProofs], [GetProofs], [Proofs] and [SendTx] messages from

\*\*les/1\*\* are no longer supported in \*\*les/2\*\*.

[Client Side Flow Control]: #client-side-flow-control

[Canonical Hash Trie]: #canonical-hash-trie

[CHT]: #canonical-hash-trie

[BloomBits Trie]: #bloombits-trie

[Status]: #status-0x00

[Announce]: #announce-0x01

[GetBlockHeaders]: #getblockheaders-0x02

[BlockHeaders]: #blockheaders-0x03

[GetBlockBodies]: #getblockbodies-0x04

[BlockBodies]: #blockbodies-0x05

[GetReceipts]: #getreceipts-0x06

[Receipts]: #receipts-0x07

[GetProofs]: #getproofs-0x08

[Proofs]: #proofs-0x09

[GetContractCodes]: #getcontractcodes-0x0a

[ContractCodes]: #contractcodes-0x0b

[GetHeaderProofs]: #getheaderproofs-0x0d

[HeaderProofs]: #headerproofs-0x0e

[SendTx]: #sendtx-0x0c

[GetProofsV2]: #getproofsv2-0x0f

[ProofsV2]: #proofsv2-0x10

[GetHelperTrieProofs]: #gethelpertrieproofs-0x11

[HelperTrieProofs]: #helpertrieproofs-0x12

[SendTxV2]: #sendtxv2-0x13

[GetTxStatus]: #gettxstatus-0x14

[TxStatus]: #txstatus-0x15

[StopMsg]: #stopmsg-0x16

[ResumeMsg]: #resumemsg-0x17

[Ethereum Wire Protocol]: ./eth.md

[Merkle Patricia Trie]: <https://github.com/ethereum/wiki/wiki/Patricia-Tree>

# Parity Light Protocol (PIP)

The Parity Light Protocol is a variation of LES designed and implemented by Parity Tech

for the Parity Ethereum client. Please refer to the [LES specification] for information on

the purpose of the light client protocol.

Like LES, PIP adopts a flow-control mechanism closely analogous to a [token-bucket rate

limiter] where the client is expected to mirror the server token-bucket state (as

exceeding the 'burstiness' depth is a violation that results in disconnection). PIP

utilises [Canonical Hash Tries] \(CHTs), which are also described in the LES documentation.

Unlike LES, a PIP CHT is generated once every 2048 blocks. One 32-byte trie root is stored

for every range of 2048 blocks.

The current version is \*\*pip/1\*\*. This specification was derived from the official

specification at `https://wiki.parity.io`. However, the official specification has since

been deleted.

## Notation

Throughout this document, and in accordance with other devp2p documents, when referring to

wire message formats the following symbols apply:

`[ .. , .. , .. ]` means an RLP list

`a || b` means concatenation of `a` and `b`

`...` means additional list elements

## Handshake

After the initial RLPx handshake, the first message that must be communicated is from the

server to the light peer and is a status message. Updates to information in the status

message are supplied with announcements.

### Status (0x00)

`[[key0, value0], [key1, value1], ...]`

Keys are strings. Mandatory keys and values are as follows:

- `"protocol\_version"` 1 for this PIP/1 protocol version.

- `"network\_id"` 0 for testnet, 1 for mainnet

- `"total\_difficulty"` integer total difficulty of the best chain as found in the block header.

- `"head\_blockhash"` the hash of the best (i.e. highest total difficulty) known block.

- `"head\_blocknum"` the number of the best (i.e. highest total difficulty) known block.

- `"genesisHash""` the hash of the genesis block.

Optional keys and values are as follows:

- `"serve\_headers"` any value and key-pair present if the peer can serve header chain

downloads.

- `"serve\_chain\_since"` present if the peer can serve Body/Receipts ODR requests starting

from the given block number.

- `"serve\_state\_since"` present if the peer can serve Proof/Code ODR requests starting

from the given block number.

- `"tx\_relay"` present if the peer can relay transactions to the network.

- `"flow\_control\_bl"` max credits (positive integer describing the burst-depth of the

token bucket),

- `"flow\_control\_mrc"` the initial cost table (see below)

- `"flow\_control\_mrr"` rate of recharge (positive integer of credits recharged per second)

#### Cost Table

The cost table includes a mapping of individual [PIP Request/Response Messages] to costs,

which are applied in the token-bucket rate limiter. The [Headers] and [Execution] request

messages are special cases where the cost is multiplied by the maximum number of requested

header or gas requested, respectively. The table also includes a base cost, which is

applied for every [Request Batch].

cost\_table = [base\_cost, [id,cost],...]

base\_cost = positive integer cost applied to a request batch.

id = identifier of an individual PIP message type

cost = positive integer to apply to cost calculations for this message type

### Announcement (0x01)

`[head\_blockhash, head\_blocknum, total\_difficulty, reorg\_depth, [key0, value0], [key1, value1], ...]`

- `reorg\_depth` is positive integer containing the reorganization depth to the common

ancestor of the new head and the last announced head.

- Other elements have the same meaning as in the [Status] message with the exception of

`reorg\_depth`.

### Request Batch (0x02)

`[request-id, [req1, ...]]`

where

- `request-id` is a unique scalar request identifier for request-reply correlation.

- `[req1, ...]` is the list of request messages, as described in the [PIP Request/Response Messages]

section.

This message, sent from client to server, requests that the given request messages should

be executed. The server responds with a Response Batch.

### Response Batch (0x03)

`[request-id, cr, [resp1, ...]]`

where

- `request-id1` is the unique scalar correlating with a previously received request message.

- `cr` is an updated amount of request credits prior to recharge events at the time of

processing on the server (please see throttling below).

- `[resp1, ...]` is the list of response messages.

There must be a response message for each request contained in the corresponding request batch.

The individual responses must supply all elements of the response message specifications.

The PIP protocol considers messages missing any of these elements \*incomplete\*.

### UpdateCreditParameters (0x04)

`[max, recharge, cost\_table]`

where

- `max` is a positive integer, the new maximum credit depth for the token bucket.

- `recharge` a positive integer, the new recharge rate in credits per second.

- `cost\_table` is the updated [Cost Table].

The server may periodically update the token-bucket parameters, such as depth, message

cost and recharge rate, for the particular client. Received updates must be acknowledged

with an AcknowledgeUpdate message.

### AcknowledgeUpdate (0x05)

This message acknowledges receipt of updated credit parameters and has no payload.

### RelayTransactions (0x06)

`[tx1, tx2, ...]`

where

`tx1`, `tx2` are RLP encoded transactions as per [ETH] documentation.

This message requests that the given transactions should be relayed to the

to the eth network.

## PIP Request/Response Messages

PIP request and response messages are batched and cannot be sent individually. Unlike LES,

PIP batches may contain multiple messages of different types. The [Request Batch] is used

to send messages of the types described below to the server.

Each message type also specifies its corresponding response message (referred to as

\*outputs\*). Response messages are sent as a [Response Batch] by the server when requests

have executed.

PIP tries to further optimise client-server round trips by allowing the individual

requests in the batch to include references to what their responses would contain if

processed sequentially. For clarification, an example PIP batch request could contain two

request messages in order, where the second message specifies that an input is a specific

'output' of the first message, where 'output' means the server response to that request.

Referencing a field in a response to a batched request is achieved with \*loose inputs\* and

\*reusable outputs\*. Response message fields are documented as being \*\*reusable as `n`\*\*

where `n` is an identifier labelling the field in the response message body.

\*Loose inputs\* may be a back-reference to a \*reusable output\* or may be hard data.

loose\_input = [raw\_flag, input]

raw\_flag = is 0 or 1 (a.k.a. 'discriminant')

input = if raw\_flag is 0, this is the RLP encoded value

if raw\_flag is 1, this is back\_reference

back\_reference = [request\_message\_index, reusable\_output]

request\_message\_index = the 0-based position of a prior message in the request batch

reusable\_output = the unsigned integer identifying the corresponding response message field

The following are the individual messages, paired as requests and their responses.

### Headers (0x00)

Request and retrieve block headers from the server.

#### Request

`[message-id, [start, skip, max, reverse]]`

- `start` Loose, of type either 32byte hash (block hash), or unsigned integer block number

- `skip` unsigned integer N, specifying the server should return every Nth block

- `max` unsinged integer, the maximum number of blocks to return

- `reverse` 0 if the block numbers should be increasing, 1 to return in reverse order

#### Response

`[message-id, [header1, header2, ...]]`

- `header1, header2, ...` the requested block headers

### HeaderProof (0x01)

Request for a header proof.

#### Request

`[message-id, [block]]`

- `block` Loose, of type unsigned integer, referring to the block number

#### Response

`[message-id, [cht\_inclusion\_proof, block\_hash, total\_difficulty]]`

- `cht\_inclusion\_proof` is `[[node1, node2, ...], ...]`

- `node1` merkle tree node as byte array

- `block\_hash` hash of the requested block \*\*reusable as 0\*\*

- `total\_difficulty` unsigned integer, the requested block total difficulty

### TransactionIndex (0x02)

Request for transaction inclusion information by transaction hash.

#### Request

`[message-id, [hash]]`

- `hash` Loose, of type 32 byte hash, referring to the transaction hash.

#### Response

`[message-id, [block\_number, block\_hash, index]]`

- `block\_number` the block number of the block containing the transaction \*\*reusable as 0\*\*

- `block\_hash` hash of the requested block \*\*reusable as 1\*\*

- `index` index in the block

### BlockReceipts (0x03)

Request for a block's receipts.

#### Request

`[message-id, [hash]]`

- `hash` Loose, of type 32 byte hash, referring to the block hash.

#### Response

`[message-id, [receipts]]`

- `receipts` is `[receipt1, receipt2, ...]`

- `receipt1` a receipt, as per ETH spec.

### BlockBody (0x04)

Request for a block's transactions.

#### Request

`[message-id, [hash]]`

- `hash` Loose, of type 32 byte hash, referring to the transaction hash

#### Response

`[message-id, [transactions, uncles]]`

- `transactions` is `[tx1, tx2, ...]`

- `tx1` a transaction, as per ETH spec

- `uncles` is `[header1, header2,...]`

- `header1` an uncle block header as per ETH spec

### Account (0x05)

Request for proof of specific account in the state.

#### Request

`[message-id , [block\_hash, address\_hash]]`

- `block\_hash` Loose, of type 32 byte hash, referring to the block hash

- `address\_hash` Loose, of type 32 byte hash, referring to the account address hash

#### Response

`[message-id, [cht\_inclusion\_proof, nonce, balance, code\_hash, storage\_root]]`

- `cht\_inclusion\_proof` is `[[node1, node2, ...], ...]`

- `node1` merkle tree node as byte array

- `nonce` the block nonce (unsigned integer)

- `balance` the account balance (unsigned integer)

- `code\_hash` 32 byte hash \*\*reusable as 0\*\*

- `storage\_root` 32 byte storage root hash \*\*reusable as 1\*\*

### Storage (0x06)

Request for a proof of contract storage.

#### Request

`[message-id, [block\_hash, address\_hash, storage\_key\_hash]]`

- `block\_hash` Loose, of type 32 byte hash, referring to the block hash

- `address\_hash` Loose, of type 32 byte hash, referring to the account address hash

- `storage\_key\_hash` Loose, of type 32 byte hash, referring to the storage key

#### Response

`[message-id, [cht\_inclusion\_proof, storage\_value]]`

- `cht\_inclusion\_proof` is `[[node1, node2, ...], ...]`

- `node1` merkle tree node as byte array

- `storage\_value` 32 byte hash \*\*reusable as 0\*\*

### Code (0x07)

Request for contract code.

#### Request

`[message-id, [block\_hash, code\_hash]]`

- `block\_hash` Loose, of type 32 byte hash, identifying the block.

- `code\_hash` Loose, of type 32 byte hash, identifying the code.

#### Response

`[message-id, [bytecode]]`

- `bytecode` byte array of the contract code

### Execution (0x08)

Request for Merkle proofs of a contract execution.

#### Request

`[message-id, [block\_hash, from\_address, call\_or\_create\_address, gas\_to\_prove, gas\_price, value, data]]`

- `block\_hash` Loose, of type 32 byte hash, identifying the block

- `from\_address` Type 32 byte hash, referring to the caller account address hash

- `call\_or\_create\_address` 32 byte hash, call contract if address, otherwise create contract if empty

- `gas\_to\_prove` 32 byte unsigned integer of gas to prove

- `gas\_price` 32 byte unsigned integer of gas price

- `value` 32 byte unsigned integer of value to transfer

- `data` byte array of relevant data

#### Response

`[message-id, [proof]]`

- `proof` is `[[node1, node2, ...], ...]`, the necessary execution proof

- `node1` merkle tree node as byte array

[LES specification]: ./les.md

[ETH]: ./eth.md

[Cost Table]: #cost-table

[Canonical Hash Tries]: ./les.md#canonical-hash-trie

[token-bucket rate limiter]: https://en.wikipedia.org/wiki/Token\_bucket

[Status]: #status-0x00

[Request Batch]: #request-batch-0x02

[Response Batch]: #response-batch-0x03

[PIP Request/Response Messages]: #pip-requestresponse-messages

[Headers]: #headers-0x00

[Execution]: #execution-0x08

# Ethereum Snapshot Protocol (SNAP)

The `snap` protocol runs on top of [RLPx], facilitating the exchange of Ethereum state

snapshots between peers. The protocol is an optional extension for peers supporting (or

caring about) the dynamic snapshot format.

The current version is `snap/1`.

## Overview

The `snap` protocol is designed for semi real-time data retrieval. It's goal is to make

dynamic snapshots of recent states available for peers. The `snap` protocol does not take

part in chain maintenance (block and transaction propagation); and it is \*\*meant to be run

side-by-side with the `eth` protocol\*\*, not standalone (e.g. chain progression is

announced via `eth`).

The protocol itself is simplistic by design (take note, the supporting implementation is

everything but simple). In its crux, `snap` supports retrieving a contiguous segment of

accounts from the Ethereum state trie, or a contiguous segment of storage slots from one

particular storage trie. Both replies are Merkle proven for immediate verification. In

addition batches of bytecodes can also be retrieved similarly to the `eth` protocol.

The synchronization mechanism the protocol enables is for peers to retrieve and verify all

the account and storage data without downloading intermediate Merkle trie nodes. The final

state trie is reassembled locally. An additional complexity nodes must be aware of, is

that state is ephemeral and moves with the chain, so syncers need to support reassembling

partially consistent state segments. This is supported by trie node retrieval similar to

`eth`, which can be used to heal trie inconsistencies (more on this later).

The `snap` protocol permits downloading the entire Ethereum state without having to

download all the intermediate Merkle proofs, which can be regenerated locally. This

reduces the networking load enormously:

- Ingress bandwidth is reduced from `O(accounts \* log account + SUM(states \* log states))`

(Merkle trie nodes) to `O(accounts + SUM(states))` (actual state data).

- Egress bandwidth is reduced from `O(accounts \* log account + SUM(states \* log states)) \*

32 bytes` (Merkle trie node hashes) to `O(accounts + SUM(states)) / 100000 bytes`

(number of 100KB chucks to cover the state).

- Round trip time is reduced from `O(accounts \* log account + SUM(states \* log states)) /

384` (states retrieval packets) to `O(accounts + SUM(states)) / 100000 bytes` (number of

100KB chucks to cover the state).

### Expected results

To put some numbers on the above abstract orders of magnitudes, synchronizing Ethereum

mainnet state (i.e. ignoring blocks and receipts, as those are the same) with `eth` vs.

the `snap` protocol:

Block ~#11,177,000:

- Accounts: 107,598,788 @ 19.70GiB

- Byte codes: 319,654 @ 1.48GiB

- Storage slots: 365,787,020 @ 49.88GiB

- Trie nodes: 617,045,138

| | Time | Upload | Download | Packets | Serving disk reads\* |

|:------:|:------:|:-------:|:--------:|:--------:|:-------------------:|

| `eth` | 10h50m | 20.38GB | 43.8GB | 1607M | 15.68TB |

| `snap` | 2h6m | 0.15GB | 20.44GB | 0.099M | 0.096TB |

| | -80.6% | -99.26% | -53.33% | -99.993% | -99.39% |

\*\\*Also accounts for other peer requests during the time span.\*

Post snap state heal:

- Additional trie nodes: 541,260 @ 160.44MiB

- Additional byte codes: 34 @ 234.98KiB

## Relation to `eth`

The `snap` protocol is a \*dependent satellite\* of `eth` (i.e. to run `snap`, you need to

run `eth` too), not a fully standalone protocol. This is a deliberate design decision:

- `snap` is meant to be a bootstrap aid for newly joining full nodes. By enforcing all

`snap` peers to also speak `eth`, we can avoid non-full nodes from lingering attached to

`snap` indefinitely.

- `eth` already contains well established chain and fork negotiation mechanisms, as well

as remote peer staleness detection during sync. By running both protocols side-by-side,

`snap` can benefit of all these mechanisms without having to duplicate them.

This \*satellite\* status may be changed later, but it's better to launch with a more

restricted protocol first and then expand if need be vs. trying to withdraw depended-upon

features.

The `snap` protocol is not an extension / next version of `eth` as it relies on the

availability of a \*snapshot\* acceleration structure that can iterate accounts and storage

slots linearly. Its purpose is also one specific sync method that might not be suitable

for all clients. Keeping `snap` as a separate protocol permits every client to decide to

pursue it or not, without hindering their capacity to participate in the `eth` protocol.

## Synchronization algorithm

The crux of the snapshot synchronization is making contiguous ranges of accounts and

storage slots available for remote retrieval. The sort order is the same as the state trie

iteration order, which makes it possible to not only request N subsequent accounts, but

also to Merkle prove them. Some important properties of this simple algorithm:

- Opposed to \*fast sync\*, we only need to transfer the useful leaf data from the state

trie and can reconstruct internal nodes locally.

- Opposed to \*warp sync\*, we can download small chunks of accounts and storage slots and

immediately verify their Merkle proofs, making junk attacks impossible.

- Opposed to \*warp sync\*, random account ranges can be retrieved, thus synchronization

concurrency is totally dependent on client implementation and is not forced by the

protocol.

The gotcha of the snapshot synchronization is that serving nodes need to be able to

provide \*\*fast\*\* iterable access to the state of the most recent `N` (128) blocks.

Iterating the Merkle trie itself might be functional, but it's not viable (iterating the

state trie at the time of writing takes 9h 30m on an idle machine). Geth introduced

support for [dynamic snapshots], which allows iterating all the accounts in 7m

(see [blog for more]). Some important properties of the dynamic snapshots:

- Serving a contiguous range of accounts or storage slots take `O(n)` operations, and more

importantly, it's the same for disk access too, being stored contiguously on disk (not

counting the database read amplification).

- Maintaining a live dynamic snapshot means:

- Opposed to \*warp sync\*, syncing nodes can always get the latest data, thus they don't

need to process days' worth of blocks afterwards.

- Opposed to \*warp sync\*, there is no pre-computation to generate a snapshot (it's

updated live), so there's no periodic burden on the nodes to iterate the tries (there

it an initial burden to create the first snapshot after sync though).

- Providing access to 128 recent snapshots permits `O(1)` direct access to any account

and state, which can be used during EVM execution for `SLOAD`.

The caveat of the snapshot synchronization is that as with \*fast sync\* (and opposed to

\*warp sync\*), the available data constantly moves (as new blocks arrive). The probability

of finishing sync before the 128 block window (15m) moves out is asymptotically zero. This

is not a problem, because we can self-heal. It is fine to import state snapshot chunks

from different tries, because the inconsistencies can be fixed by running a

\*fast-sync-style-state-sync\* on top of the assembled semi-correct state afterwards. Some

important properties of the self-healing:

- Synchronization can be aborted at any time and resumed later. It might cause

self-healing to run longer, but it will fix the data either way.

- Synchronization on slow connections is guaranteed to finish too (as long as the node can

download data faster than it's being produced by the network), the data cannot disappear

from the network (opposed to warp sync).

## Data format

The accounts in the `snap` protocol are analogous to the Ethereum RLP consensus encoding

(same fields, same order), but in a \*\*slim\*\* format:

- The code hash is `empty list` instead of `Keccak256("")`

- The root hash is `empty list` instead of `Hash(<empty trie>)`

This is done to avoid having to transfer the same 32+32 bytes for all plain accounts over

the network.

## Protocol Messages

### GetAccountRange (0x00)

`[reqID: P, rootHash: B\_32, startingHash: B\_32, responseBytes: P]`

Requests an unknown number of accounts from a given account trie, starting at the

specified account hash and capped by the maximum allowed response size in bytes. The

intended purpose of this message is to fetch a large number of subsequent accounts from a

remote node and reconstruct a state subtrie locally.

- `reqID`: Request ID to match up responses with

- `rootHash`: Root hash of the account trie to serve

- `startingHash`: Account hash of the first to retrieve

- `responseBytes`: Soft limit at which to stop returning data

Notes:

- Nodes \*\*must\*\* always respond to the query.

- If the node does \*\*not\*\* have the state for the requested state root, it \*\*must\*\* return

an empty reply. It is the responsibility of the caller to query an state not older than

128 blocks.

- The responding node is allowed to return \*\*less\*\* data than requested (own QoS limits),

but the node \*\*must\*\* return at least one account, unless no account exists in the

requested range.

- The responding node \*\*must\*\* Merkle prove the starting hash (even if it does not exist)

and the last returned account (if any exists after the starting hash).

Rationale:

- The starting account is identified deliberately by hash and not by address. As the

accounts in the Ethereum Merkle trie are sorted by hash, the address is irrelevant. In

addition, there is no consensus requirement for full nodes to be aware of the address

pre-images.

- The response is capped by byte size and not by number of accounts, because it makes the

network traffic more deterministic. As the state density is unknowable, it's also

impossible to delimit the query with an ending hash.

Caveats:

- When requesting accounts from a starting hash, malicious nodes may skip ahead and return

a gaped reply. Such a reply would cause sync to finish early with a lot of missing data.

Proof of non-existence for the starting hash prevents this attack, completely covering

the range from start to end.

- No special signaling is needed if there are no more accounts after the last returned

one, as the attached Merkle proof for the last account will have all trie nodes right of

the proven path zero.

### AccountRange (0x01)

`[reqID: P, accounts: [[accHash: B\_32, accBody: B], ...], proof: [node\_1: B, node\_2, ...]]`

Returns a number of consecutive accounts and the Merkle proofs for the entire range

(boundary proofs). The left-side proof must be for the requested origin hash (even if an

associated account does not exist) and the right-side proof must be for the last returned

account.

- `reqID`: ID of the request this is a response for

- `accounts`: List of consecutive accounts from the trie

- `accHash`: Hash of the account address (trie path)

- `accBody`: Account body in slim format

- `proof`: List of trie nodes proving the account range

Notes:

- If the account range is the entire state (requested origin was `0x00..0` and all

accounts fit into the response), no proofs should be sent along the response. This is

unlikely for accounts, but since it's a common situation for storage slots, this clause

keeps the behavior the same across both.

### GetStorageRanges (0x02)

`[reqID: P, rootHash: B\_32, accountHashes: [B\_32], startingHash: B, responseBytes: P]`

Requests the storage slots of multiple accounts' storage tries. Since certain contracts

have huge state, the method can also request storage slots from a single account, starting

at a specific storage key hash. The intended purpose of this message is to fetch a large

number of subsequent storage slots from a remote node and reconstruct a state subtrie

locally.

- `reqID`: Request ID to match up responses with

- `rootHash`: Root hash of the account trie to serve

- `accountHashes`: Account hashes of the storage tries to serve

- `startingHash`: Storage slot hash of the first to retrieve

- `responseBytes`: Soft limit at which to stop returning data

Notes:

- Nodes \*\*must\*\* always respond to the query.

- If the node does \*\*not\*\* have the state for the requested state root or for \*\*any\*\*

requested account hash, it \*\*must\*\* return an empty reply. It is the responsibility of

the caller to query an state not older than 128 blocks; and the caller is expected to

only ever query existing accounts.

- The responding node is allowed to return \*\*less\*\* data than requested (serving QoS

limits), but the node \*\*must\*\* return at least one slot, unless none exists.

- If multiple accounts' storage is requested, serving nodes should reply with the entire

storage ranges (thus no Merkle proofs needed), up to the first contract which exceeds

the packet limit. If the last included storage range does not fit entirely, a Merkle

proof \*\*must\*\* be attached to that and \*\*only\*\* that.

- If a single account's storage is requested, serving nodes should only return slots

starting with the requested starting hash, up to the last one or until the packet fills

up. It the entire storage range is not being returned, a Merkle proof \*\*must\*\* be

attached.

- If a proof is attached, the responding node \*\*must\*\* Merkle prove the starting hash

(even if it does not exist) and the last returned slot (if any exists after the starting

hash).

Rationale:

- The response is capped by byte size and not by number of slots, because it makes the

network traffic more deterministic.

- The request supports querying multiple contracts at the same time as most storage tries

are in the order of 100s of bytes. Querying these individually would produce a lot of

network round trips.

Caveats:

- When requesting storage slots from a starting hash, malicious nodes may skip ahead and

return a prefix-gapped reply. Such a reply would cause sync to finish early with a lot

of missing data. Proof of non-existence for the starting hash prevents this attack,

completely covering the range from start to end.

- Although serving nodes should respect the response limit requested by the caller, it is

valuable to slightly force the limit (consider it soft only) when adding the last

contract to avoid having to split it and prove it.

- No special signaling is needed if there are no more slots after the last returned one,

as the attached Merkle proof for the last account will have all trie nodes right of the

proven path zero.

### StorageRanges (0x03)

`[reqID: P, slots: [[[slotHash: B\_32, slotData: B], ...], ...], proof: [node\_1: B, node\_2, ...]]`

Returns a number of consecutive storage slots for the requested account (i.e. list of list

of slots) and optionally the Merkle proofs for the last range (boundary proofs) if it only

partially covers the storage trie. The left-side proof must be for the requested origin

slots (even if it does not exist) and the right-side proof must be for the last returned

slots.

- `reqID`: ID of the request this is a response for

- `slots`: List of list of consecutive slots from the trie (one list per account)

- `slotHash`: Hash of the storage slot key (trie path)

- `slotData`: Data content of the slot

- `proof`: List of trie nodes proving the slot range

Notes:

- If the slot range is the entire storage state, no proofs will be sent along the response.

### GetByteCodes (0x04)

`[reqID: P, hashes: [hash1: B\_32, hash2: B\_32, ...], bytes: P]`

Requests a number of contract byte-codes by hash. This is analogous to the `eth/63`

`GetNodeData`, but restricted to only bytecode to break the generality that causes issues

with database optimizations. The intended purpose of this request is to allow retrieving

the code associated with accounts retrieved via GetAccountRange, but it's needed during

healing too.

- `reqID`: Request ID to match up responses with

- `hashes`: Code hashes to retrieve the code for

- `bytes`: Soft limit at which to stop returning data

\*This functionality was duplicated into `snap` from `eth/65` to permit `eth` long term to

become a chain maintenance protocol only and move synchronization primitives out into

satellite protocols only.\*

Notes:

- Nodes \*\*must\*\* always respond to the query.

- The returned codes \*\*must\*\* be in the request order.

- The responding node is allowed to return \*\*less\*\* data than requested (serving QoS

limits), but the node \*\*must\*\* return at least one bytecode, unless none requested are

available, in which case it \*\*must\*\* answer with an empty response.

- If a bytecode is unavailable, the node \*\*must\*\* skip that slot and proceed to the next

one. The node \*\*must not\*\* return `nil` or other placeholders.

Rationale:

- The response is capped by byte size and not by number of slots, because it makes the

network traffic more deterministic, as contract sizes can vary randomly up to 24KB with

current consensus rules.

- By retaining the original request order and skipping unavailable bytecodes, the

requesting node can differentiate between unavailable data (gaps in the hashes) and QoS

limitations (missing suffix).

Caveats:

- Implementations are free to request as many or as few bytecodes in a single request, but

they should keep in mind that requesting too few results in wasted time due to network

latency; but requesting too many results in wasted bandwidth if the response doesn't

fit. Average (unique) contract size on mainnet is about 5-6KB, so `bytes / 6KB` is a

good heuristic for the number of codes to request in a single packet (e.g. for 512KB

desired response size, 80-100 bytecodes per request is a good choice).

### ByteCodes (0x05)

`[reqID: P, codes: [code1: B, code2: B, ...]]`

Returns a number of requested contract codes. The order is the same as in the request, but

there might be gaps if not all codes are available or there might be fewer is QoS limits

are reached.

### GetTrieNodes (0x06)

`[reqID: P, rootHash: B\_32, paths: [[accPath: B, slotPath1: B, slotPath2: B, ...]...], bytes: P]`

Requests a number of state (either account or storage) Merkle trie nodes \*\*by path\*\*. This

is analogous in functionality to the `eth/63` `GetNodeData`, but restricted to only tries

and queried by path, to break the generality that causes issues with database

optimizations.

- `reqID`: Request ID to match up responses with

- `rootHash`: Root hash of the account trie to serve

- `paths`: Trie paths to retrieve the nodes for, grouped by account

- `bytes`: Soft limit at which to stop returning data

The `paths` is one array of trie node paths to retrieve per account (i.e. list of list of

paths). Each list in the array special cases the first element as the path in the account

trie and the remaining elements as paths in the storage trie. To address an account node,

the inner list should have a length of 1 consisting of only the account path. Partial

paths (<32 bytes) should be compact encoded per the Ethereum wire protocol, full paths

should be plain binary encoded.

\*This functionality was mutated into `snap` from `eth/65` to permit `eth` long term to

become a chain maintenance protocol only and move synchronization primitives out into

satellite protocols only.\*

Notes:

- Nodes \*\*must\*\* always respond to the query.

- The returned nodes \*\*must\*\* be in the request order.

- If the node does \*\*not\*\* have the state for the requested state root or for \*\*any\*\*

requested account paths, it \*\*must\*\* return an empty reply. It is the responsibility of

the caller to query an state not older than 128 blocks; and the caller is expected to

only ever query existing trie nodes.

- The responding node is allowed to return \*\*less\*\* data than requested (serving QoS

limits), but the node \*\*must\*\* return at least one trie node.

Rationale:

- The response is capped by byte size and not by number of slots, because it makes the

network traffic more deterministic. Although opposed to the previous request types

(accounts, slots, codes), trie nodes are relatively deterministic (100-500B), the

protocol remains cleaner if all packets follow the same traffic shaping rules.

- A naive way to represent trie nodes would be a simple list of `account || storage` path

segments concatenated, but that would be very wasteful on the network as it would

duplicate the account hash for every storage trie node.

### TrieNodes (0x07)

`[reqID: P, nodes: [node1: B, node2: B, ...]]`

Returns a number of requested state trie nodes. The order is the same as in the request,

but there might be fewer is QoS limits are reached.

## Change Log

### snap/1 (November 2020)

Version 1 was the introduction of the snapshot protocol.

[RLPx]: ../rlpx.md

[dynamic snapshots]: https://github.com/ethereum/go-ethereum/pull/20152

[blog for more]: https://blog.ethereum.org/2020/07/17/ask-about-geth-snapshot-acceleration/

# Ethereum Witness Protocol (wit)

The `wit` protocol runs on top of [RLPx], facilitating the exchange of Ethereum state

witnesses between peers. The protocol is an optional extension for peers supporting (or

caring about) the state witnesses for Ethereum blocks.

The current version is `wit/0`.

### Overview

The `wit` protocol is designed to assist clients in syncing up to the tip of the chain.

Eventually, it also aspires to assist in stateless client operation. The `wit` protocol

does not take part in chain maintenance (block and transaction propagation); and it is

\*\*meant to be run side-by-side with the `eth` protocol\*\*, not standalone (e.g. chain

progression is announced via `eth`). (like the `snap` protocol)

Despite the name, version 0 will not provide actual witnesses. It will provide meta-data

about the witness, which can be used to download the witness over the `eth` protocol.

For now, the known use case is to assist [Beam Syncing] peers. By requesting witness

metadata, these peers will keep up with the tip of the network and become fully-synced

nodes faster.

Using the `wit` protocol, peers ask each other for the list of trie node hashes read

during the execution of a particular block. This includes the following data:

- Storage nodes

- Bytecodes

- Account nodes

- Read during EVM execution

- Read during transaction validation

- Read during block reward calculation

- Nodes read when generating the final state root (i.e. sometimes deleting data requires a

trie refactor that reads nearby trie nodes)

The trie node hashes which are generated at the end of the block from existing data are

\*not\* included. For example, the final state root hash is not included.

### Relation to `eth`

The `wit` protocol follows the same pattern as `snap`. It is a \*dependent satellite\* of

`eth` (i.e. to run `wit`, you need to run `eth` too), not a fully standalone protocol.

This is a deliberate design decision:

- `wit` is meant to be a bootstrap aid for newly joining full nodes. By enforcing all

`wit` peers to also speak `eth`, we can avoid non-full nodes from lingering attached to

`wit` indefinitely.

- `eth` already contains well established chain and fork negotiation mechanisms, as well

as remote peer staleness detection during sync. By running both protocols side-by-side,

`wit` can benefit of all these mechanisms without having to duplicate them.

This \*satellite\* status may be changed later, but it's better to launch with a more

restricted protocol first and then expand if need be vs. trying to withdraw depended-upon

features.

In order to follow the `wit` protocol, clients must generate witness metadata when

executing blocks. For now, its primary purpose is also one specific sync method that might

not be suitable for all clients. Keeping `wit` as a separate protocol permits every client

to decide to pursue it or not, without hindering their capacity to participate in the

`eth` protocol.

### Accelerating Beam Sync

At its most naive, Beam Sync needs to download any missing state one trie node at a time.

According to a recent test, after Beam Syncing for 22 hours, the median block still

required more than 300 new trie nodes. At an optimistic 100ms round-trip time, that means

30 seconds per block of data download. This is where witness metadata can help

tremendously.

If a client can request the trie node hashes used by a block up front, those 300 trie

nodes can likely be accessed in a fraction of a second. That's easily enough to keep

synced with mainnet.

Unfortunately, the list of trie node hashes cannot be verified before the block is

imported. This would be a huge problem for a stateless client, which would be permanently

at risk to a DoS attack where peers feed it a long list of incorrect hashes. But Beam

Syncing clients are only vulnerable until they've finished downloading the full network

state, so the payoff for such an attack is smaller.

## Protocol Messages

### RESERVED (0x00)

This command is undefined, held in place for a possible future Status message.

### GetBlockWitnessHashes (0x01)

`[reqID: P, blockHash: B\_32]`

Requests a list of trie node hashes used by a given block.

- `reqID`: Request ID to match up responses with

- `blockHash`: Hash of the header to request the witness hashes for

Notes:

- Nodes \*\*must\*\* always respond to the query.

- If the node does \*\*not\*\* have the trie hashes requested block, it \*\*must\*\* return an

empty reply.

### BlockWitnessHashes (0x02)

`[reqID: P, witnessHashes: [trieNodeHash: B\_32, ...]]`

Returns a list of the trie node hashes that were read during execution and validation of

the given block.

- `reqID`: ID of the request this is a response for

- `witnessHashes`: List of trie node hashes

## Change Log

### wit/0 (October 2020)

Version 0 was the introduction of the witness protocol.

[RLPx]: ../rlpx.md

[Beam Syncing]: https://github.com/ethereum/stateless-ethereum-specs/blob/master/beam-sync-phase0.md

# Node Discovery Protocol

This specification defines the Node Discovery protocol version 4, a Kademlia-like DHT that

stores information about Ethereum nodes. The Kademlia structure was chosen because it is

an efficient way to organize a distributed index of nodes and yields a topology of low

diameter.

The current protocol version is \*\*4\*\*. You can find a list of changes in past protocol

versions at the end of this document.

## Node Identities

Every node has a cryptographic identity, a key on the secp256k1 elliptic curve. The public

key of the node serves as its identifier or 'node ID'.

The 'distance' between two node keys is the bitwise exclusive or on the hashes of the

public keys, taken as the number.

distance(n₁, n₂) = keccak256(n₁) XOR keccak256(n₂)

## Node Records

Participants in the Discovery Protocol are expected to maintain a [node record] \(ENR\)

containing up-to-date information. All records must use the "v4" identity scheme. Other

nodes may request the local record at any time by sending an [ENRRequest] packet.

To resolve the current record of any node public key, perform a Kademlia lookup using

[FindNode] packets. When the node is found, send ENRRequest to it and return the record

from the response.

## Kademlia Table

Nodes in the Discovery Protocol keep information about other nodes in their neighborhood.

Neighbor nodes are stored in a routing table consisting of 'k-buckets'. For each `0 ≤ i <

256`, every node keeps a k-bucket for nodes of distance between `2i` and `2i+1` from

itself.

The Node Discovery Protocol uses `k = 16`, i.e. every k-bucket contains up to 16 node

entries. The entries are sorted by time last seen — least-recently seen node at the head,

most-recently seen at the tail.

Whenever a new node N₁ is encountered, it can be inserted into the corresponding bucket.

If the bucket contains less than `k` entries N₁ can simply be added as the first entry. If

the bucket already contains `k` entries, the least recently seen node in the bucket, N₂,

needs to be revalidated by sending a [Ping] packet. If no reply is received from N₂ it is

considered dead, removed and N₁ added to the front of the bucket.

## Endpoint Proof

To prevent traffic amplification attacks, implementations must verify that the sender of a

query participates in the discovery protocol. The sender of a packet is considered

verified if it has sent a valid [Pong] response with matching ping hash within the last 12

hours.

## Recursive Lookup

A 'lookup' locates the `k` closest nodes to a node ID.

The lookup initiator starts by picking `α` closest nodes to the target it knows of. The

initiator then sends concurrent [FindNode] packets to those nodes. `α` is a system-wide

concurrency parameter, such as 3. In the recursive step, the initiator resends FindNode to

nodes it has learned about from previous queries. Of the `k` nodes the initiator has heard

of closest to the target, it picks `α` that it has not yet queried and resends [FindNode]

to them. Nodes that fail to respond quickly are removed from consideration until and

unless they do respond.

If a round of FindNode queries fails to return a node any closer than the closest already

seen, the initiator resends the find node to all of the `k` closest nodes it has not

already queried. The lookup terminates when the initiator has queried and gotten responses

from the `k` closest nodes it has seen.

## Wire Protocol

Node discovery messages are sent as UDP datagrams. The maximum size of any packet is 1280

bytes.

packet = packet-header || packet-data

Every packet starts with a header:

packet-header = hash || signature || packet-type

hash = keccak256(signature || packet-type || packet-data)

signature = sign(packet-type || packet-data)

The `hash` exists to make the packet format recognizable when running multiple protocols

on the same UDP port. It serves no other purpose.

Every packet is signed by the node's identity key. The `signature` is encoded as a byte

array of length 65 as the concatenation of the signature values `r`, `s` and the 'recovery

id' `v`.

The `packet-type` is a single byte defining the type of message. Valid packet types are

listed below. Data after the header is specific to the packet type and is encoded as an

RLP list. Implementations should ignore any additional elements in the `packet-data` list

as well as any extra data after the list.

### Ping Packet (0x01)

packet-data = [version, from, to, expiration, enr-seq ...]

version = 4

from = [sender-ip, sender-udp-port, sender-tcp-port]

to = [recipient-ip, recipient-udp-port, 0]

The `expiration` field is an absolute UNIX time stamp. Packets containing a time stamp

that lies in the past are expired may not be processed.

The `enr-seq` field is the current ENR sequence number of the sender. This field is

optional.

When a ping packet is received, the recipient should reply with a [Pong] packet. It may

also consider the sender for addition into the local table. Implementations should ignore

any mismatches in version.

If no communication with the sender has occurred within the last 12h, a ping should be

sent in addition to pong in order to receive an endpoint proof.

### Pong Packet (0x02)

packet-data = [to, ping-hash, expiration, enr-seq, ...]

Pong is the reply to ping.

`ping-hash` should be equal to `hash` of the corresponding ping packet. Implementations

should ignore unsolicited pong packets that do not contain the hash of the most recent

ping packet.

The `enr-seq` field is the current ENR sequence number of the sender. This field is

optional.

### FindNode Packet (0x03)

packet-data = [target, expiration, ...]

A FindNode packet requests information about nodes close to `target`. The `target` is a

65-byte secp256k1 public key. When FindNode is received, the recipient should reply with

[Neighbors] packets containing the closest 16 nodes to target found in its local table.

To guard against traffic amplification attacks, Neighbors replies should only be sent if

the sender of FindNode has been verified by the endpoint proof procedure.

### Neighbors Packet (0x04)

packet-data = [nodes, expiration, ...]

nodes = [[ip, udp-port, tcp-port, node-id], ...]

Neighbors is the reply to [FindNode].

### ENRRequest Packet (0x05)

packet-data = [expiration]

When a packet of this type is received, the node should reply with an ENRResponse packet

containing the current version of its [node record].

To guard against amplification attacks, the sender of ENRRequest should have replied to a

ping packet recently (just like for FindNode). The `expiration` field, a UNIX timestamp,

should be handled as for all other existing packets i.e. no reply should be sent if it

refers to a time in the past.

### ENRResponse Packet (0x06)

packet-data = [request-hash, ENR]

This packet is the response to ENRRequest.

- `request-hash` is the hash of the entire ENRRequest packet being replied to.

- `ENR` is the node record.

The recipient of the packet should verify that the node record is signed by the public key

which signed the response packet.

# Change Log

## Known Issues in the Current Version

The `expiration` field present in all packets is supposed to prevent packet replay. Since

it is an absolute time stamp, the node's clock must be accurate to verify it correctly.

Since the protocol's launch in 2016 we have received countless reports about connectivity

issues related to the user's clock being wrong.

The endpoint proof is imprecise because the sender of FindNode can never be sure whether

the recipient has seen a recent enough pong. Geth handles it as follows: If no

communication with the recipient has occurred within the last 12h, initiate the procedure

by sending a ping. Wait for a ping from the other side, reply to it and then send

FindNode.

## EIP-868 (October 2019)

[EIP-868] adds the [ENRRequest] and [ENRResponse] packets. It also modifies [Ping] and

[Pong] to include the local ENR sequence number.

## EIP-8 (December 2017)

[EIP-8] mandated that implementations ignore mismatches in Ping version and any additional

list elements in `packet-data`.

[Ping]: #ping-packet-0x01

[Pong]: #pong-packet-0x02

[FindNode]: #findnode-packet-0x03

[Neighbors]: #neighbors-packet-0x04

[ENRRequest]: #enrrequest-packet-0x05

[ENRResponse]: #enrresponse-packet-0x06

[EIP-8]: https://eips.ethereum.org/EIPS/eip-8

[EIP-868]: https://eips.ethereum.org/EIPS/eip-868

[node record]: ./enr.md

# Node Discovery Protocol v5 - Rationale

\*\*Protocol version v5.1\*\*

Note that this specification is a work in progress and may change incompatibly without

prior notice.

This document explains the design requirements and security needs of Discovery v5. In

addition, the document tries to gather the various vulnerabilities and threats that

pertain to Kademlia-like p2p networks. Our aim is to make it plain which issues are

addressed and how they are mitigated, so that the design of the [wire protocol] may be

verified.

# Design Requirements

## Basic Goals

#### 1.1.1 Replace the Discovery v4 Endpoint Proof

The existing mutual endpoint verification process is unreliable because either side may

forget about a previously performed endpoint proof. If node A assumes that node B already

knows about a recent PING/PONG interaction and sends FINDNODE, the request may fail.

Implementations of Discovery v4 may guard against this flaw using retries, but retrying is

really slow and usually not done.

#### 1.1.2 Require knowledge of destination node ID for communication

Make it expensive to obtain the logical node ID from discovery communications. In

Discovery v4, any node can provoke responses knowing IP alone, and obtain information

about a node without knowing its ID. This encourages sloppy implementations to not perform

proper validation of FINDNODE results and increases the risk of DHT misuse for DDoS

purposes.

#### 1.1.3 Support more than one node ID cryptosystem

Ensure the DHT can accomodate ENR's with multiple identity systems. This will allow

identity cryptosystems other than \*secp256k1/keccak256\*.

#### 1.1.4 Replace node information tuples with ENRs

ENRs include discovery information and more. These signed, versioned records fulfill

multiple requirements, such as permitting capability advertisement and transport

negotiation.

#### 1.1.5 Guard against Kademlia implementation flaws

Discovery v4 trusts other nodes to return neighbors according to an agreed distance

metric. Mismatches in implementation can make it hard for nodes to join the network, or

lead to network fragmentation.

#### 1.1.6 Secondary topic-based node index

The protocol must support discovery of nodes via an arbitrary topic identifier. Finding

nodes belonging to a topic should be as fast or faster than finding a node with a certain

ID.

#### 1.1.7 Change replay prevention

The use of timestamps as a replay prevention mechanism in Discovery v4 has led to many

complaints about connectivity when the host's clock was wrong. The protocol should be

independent of the clock.

#### 1.1.8 Message obfuscation

The protocol should obfuscate traffic to prevent accidental packet mangling or trivial

sniffing. It must also avoid inclusion of obvious markers to prevent naive blocking of

discovery traffic using hard-coded packet signatures. Defense against advanced traffic

analysis systems, e.g. using inter-packet timing is a secondary concern.

## Security Goals

Individual potential vulnerabilities are identified below. These each represent their own

risk mitigation goal.

#### 1.2.1 Replay of the handshake

The handshake, if successfully replayed from an older session, would allow a malicious

node to occupy a former IP location, or pollute the routing table with old information.

#### 1.2.2 Replay NODES

A NODES response, if successfully replayed, would pollute the routing table with stale

information.

#### 1.2.3 Replay PONG

A PONG, if successfully replayed, could convince a node that a node is live and

participating when it isn't.

#### 1.2.4 Kademlia redirection

A FindNode response contains false endpoint information intended at directing traffic at a

victim / polluting the routing table. A topic query results in fake endpoint information,

directing traffic at a victim.

#### 1.2.5 Kademlia redirection + self-propagation

As 1.2.3 but the responses attempt to replicate the malicious node throughout the routing

table, to amplify the source of pollution and traffic.

#### 1.2.6 Unsolicited replies

A malicious node is attempting to spam a node with fake responses to typical requests.

These messages may be replayed from previous communications, or may be new messages with

spoofed source endpoints. The aim is to disrupt weak implementations or have their

information be received as authentic, to pollute the recipient's routing table.

#### 1.2.7 Amplification

Malicious requests of small message size are sent from spoofed source IPs to direct larger

response messages at the victim.

#### 1.2.8 Kademlia direct validation

Direct validation of a newly discovered node can be an attack vector. A malicious node may

supply false node information with the IP of a victim. Validation traffic is then directed

at the victim.

#### 1.2.9 Kademlia ID count per address validations

There are various attacks facilitated by being able to associate multiple fake (or even

real) malicious node ids with a single IP endpoint. One mitigation method that is

sometimes considered is to globally limit the number of logical node IDs that can be

associated with an IP address. However, this is an attack vector. A malicious actor can

supply many logical node ids for a single IP address and thus prevent the correct node

from being able to join the network.

#### 1.2.10 Sybil/Eclipse attacks

These attacks rely on being able to create many real nodes, or spoof many logical node IDs

for a small number of physical endpoints, to form a large, isolated area of the network

under the control of the malicious actor. The victim's discovery findings are directed

into that part of the network, either to manipulate their traffic or to fully isolate them

from the network.

## Version Interoperability / Upgrade Paths

There are several considerations regarding the coexistence of v4 and v5 network members.

#### 1.3.1 Transition period during network formation

Discovery v4 clients should be able to serve as discovery v5 bootstrap nodes while the

number of new discovery v5 clients is still low.

#### 1.3.2 Circumvention of 1.1.2 with v4 PING

While a client supports both the old v4 and newer versions, it is possible for malicious

actors to pose as a v4 node and recover node IDs from arbitrary IP addresses. This should

somehow be avoided.

# Rationale

## Why UDP?

The wire protocol specification mandates the use of UDP. This may seem restrictive, but

use of UDP communication is an important part of the design. While there is no single

reason which ultimately dictates this choice, there are many reasons why the system as a

whole will function a lot better in the context of UDP.

For discovery to work, all nodes must be able to communicate with each other on equal

footing. The network won't form properly if some nodes can only communicate with certain

other nodes. Incooperative NAT in between the node and the Internet can cause

communication failure. UDP is fundamentally easier to work with when it comes to NAT

traversal. No explicit hole-punching is required if the NAT setup is capable of full-cone

translation, i.e. a single packet sent to any other node establishes a port mapping which

allows packets from others to reach the node behind NAT.

Unlike other DHT systems such as IPFS, the node discovery protocol mandates a single wire

protocol to be implemented by everyone. This avoids communication failures due to

incompatible transports and strengthens the DHT because all participants are guaranteed to

be reachable on the declared endpoint. It is also fundamentally simpler to reason about

and implement: the protocol either works in a certain context or it doesn't. If the

protocol cannot be used because the networking environment doesn't support UDP, another

discovery mechanism must be chosen.

Another reason for UDP is communication latency: participants in the discovery protocol

must be able to communicate with a large number of other nodes within a short time frame

to establish and maintain the neighbor set and must perform regular liveness checks on

their neighbors. For the topic advertisement system, registrants collect tickets and must

use them as soon as the ticket expires to place an ad in a topic queue.

These protocol interactions are difficult to implement in a TCP setting where connections

require multiple round-trips before application data can be sent and the connection

lifecycle needs to be maintained. An implementation of the wire protocol on a TCP-based

transport would either need permanent connection to hundreds of nodes, in which case the

application would be short on file descriptors, or establish many short-lived TCP

connections per second to communicate with specific nodes.

Yet another useful property of UDP is that packets aren't required to reach their

destination --- intermediaries may drop arbitrary packets. This strengthens the protocol

because it must be designed to function even under bad connectivity. Implementations may

exploit the possibility of packet loss to their advantage. A participant can never tell

whether a certain request wasn't answered in time because the recipient chose to ignore it

or because their own connection isn't working. An implementation that tries to minimize

traffic or CPU overhead could simply drop a certain amount of packets at application level

to stay within self-imposed limits.

## Why Kademlia?

Kademlia is a simple distributed hash table design proposed in 2002. It is commonly used

for file-sharing systems where content is stored by hash and distributed among

participants based on their 'proximity' according to the XOR distance metric.

Node discovery is a Kademlia-inspired system but doesn't store any files, only node

information is relayed. We chose Kademlia primarily because the algorithm is simple and

understandable while providing a distributed database that scales with the number of

participants. Our system also relies on the routing table to allow enumeration and random

traversal of the whole network, i.e. all participants can be found. Most importantly,

having a structured network with routing enables thinking about DHT 'address space' and

'regions of address space'. These concepts are used to build the [topic-based node index].

Kademlia is often criticized as a naive design with obvious weaknesses. We believe that

most issues with simple Kademlia can be overcome by careful programming and the benefits

of a simple design outweigh the cost and risks of maintaining a more complex system.

## Sybil and Eclipse Attacks

The well-known 'sybil attack' is based on the observation that creating node identities is

essentially free. In any system using a measure of proximity among node identities, an

adversary may place nodes close to a chosen node by generating suitable identities. For

basic node discovery through network enumeration, the 'sybil attack' poses no significant

challenge. Sybils are a serious issue for the topic-based node index, especially for

topics provided by few participants, because the index relies on node distance.

An 'eclipse attack' is usually based on generating sybil nodes with the goal of polluting

the victim node's routing table. Once the table is overtaken, the victim has no way to

find any other nodes but those controlled by the adversary. Even if creating sybil nodes

were somehow impossible, 'eclipsing' a node might still be achieved through other means

such as directing large amounts of traffic to the node. When the victim node is unable to

keep up regular communication with the rest of the network it may lose connection and be

forced into re-bootstrapping its routing table --- a situation in which it is most

vulnerable.

Both the 'sybil attack' and the 'eclipse attack' must be considered for any structured

overlay network, and there is no single optimal solution to fully protect against these

attacks. However, certain implementation decisions can make them more expensive or render

them ineffective.

As a general measure, implementations can place IP-based limits on the content of their

routing table. For example, limiting Kademlia table buckets to two nodes from every /24 IP

subnetwork and the whole table to 10 nodes per /24 IP subnetwork significantly increases

the number of hosts an attacker must control to overtake the routing table. Such limits

are effective because IPv4 addresses are a scarce resource. Subnetwork-based limits remain

effective even as IPv6 adoption progresses.

To counter being eclipsed via repeated contact by an adversary, implementations of the

Kademlia table should avoid taking on new members on incoming contact unless the table is

well-stocked from outbound queries. Readers of the original Kademlia paper may easily

assume that liveness checks on bucket members should be performed just when a new node

tries to enter the bucket, but doing so increases the risk of emptying the table through

DoS. We therefore recommend to perform liveness checks on a separate schedule which is

independent of incoming requests. Checks may also be paused or delayed when the node is

under high load. The number of past liveness checks performed on a bucket member is an

important indicator of its age: Implementations should favor long-lived nodes and may

relax liveness checks according to node age.

A well-researched countermeasure to sybil attacks is to make creation of identities

computationally expensive. While effective in theory, there are significant downsides to

this approach. Nodes on resource-constrained devices such as mobile phones may not be able

to solve the computational puzzle in time to join the network. Continuous advances in

hashing technology which speed up cryptocurrency proof-of-work algorithms show that this

way of securing the network requires constant adjustments to thresholds and can never beat

determined attackers.

Support for mixed ENR identity schemes, described later in this document, allows for an

escape hatch to introduce arbitrary optional constraints (including proof-of-work) on node

identities. Thus, while the issue is not directly addressed at wire protocol level, there

is no inherent blocker for solving it as the need arises.

## Node Records and Their Properties

In Discovery v5, all node information is exchanged using [node records]. Records are

self-signed by the node they describe and contain arbitrary key-value pairs. They also

contain a sequence number to determine which copy of the record is newer when multiple

copies are available. When a node record is changed by its owner, the sequence number

increases. The new record 'syncs' to neighboring nodes because they will request it during

liveness revalidation. The record is also 'pushed' on to newly seen nodes as part of the

handshake.

Signing records prevents any intermediary node from changing the content of a record. Any

node's information is either available in the exact form it was published or not at all.

To make the system secure, proper validation of records is important. Implementations must

verify the signature of all received records. Implementations should also avoid sharing

records containing no usable IP addresses or ports and check that Internet hosts do not

attempt to share records containing LAN IP addresses.

## On Encryption

An early draft of Discovery v5 integrated weak obfuscation based on XORing packet content

as an optional facility. As development of the protocol progressed, we understood that

traffic amplification, replay and packet authentication could all be solved by introducing

a real encryption scheme. The way the handshake and encryption works is primarily aimed at

these issues and is not supposed to ensure complete anonymity of DHT users. While it does

protect against passive observers, the handshake is not forward-secure and active protocol

participants can access node information by simply asking for it.

Node identities can use different kinds of keys depending on the identity scheme used in

the node record. This has implications on the handshake because it deals with the public

key used to derive the identity. Implementations of Discovery v5 must agree on the set of

supported identity schemes to keep the network interoperable and custom code to verify the

handshake is required for every new scheme. We believe this is an acceptable tradeoff

because introducing a new kind of node identity is a rare event.

Since the handshake performs complex cryptographic operations (ECDH, signature

verification) performance of the handshake is a big concern. Benchmarking the experimental

Go implementation shows that the handshake computation takes 500µs on a 2014-era laptop

using the default secp256k1/keccak256 identity scheme. That's a lot, but note the cost

amortizes because nodes commonly exchange multiple packets. Subsequent packets in the same

conversation can be decrypted and authenticated in just 2µs. The most common protocol

interaction is a FINDNODE or TOPICQUERY request on an unknown node with 4 NODES responses.

To put things into perspective: encryption and authentication in Discovery v5 is still a

significant improvement over the authentication scheme used in Discovery v4, which

performs secp256k1 signature 'recovery' (benchmark: ~170µs) on every packet. A FINDNODE

interaction with an unknown v4 node takes 7 packets (2x PING/PONG, FINDNODE, 2x NEIGHBORS)

and costs 1.2ms on each side for the crypto alone. In addition, the v5 handshake reduces

the risk of computational DoS because it costs as much to create as it costs to verify and

cannot be replayed.

## On Amplification and Replay

Any openly accessible packet-based system must consider misuse of the protocol for traffic

amplification purposes. There are two possible avenues of attack: In the first, an

adversary who wishes to attack a third-party host may send packets with 'spoofed' source

IP address to a node, attempting to make the node send a larger response to the victim

endpoint. In the second, the adversary attempts to install a node record containing the

victim's endpoint in the DHT, causing other nodes to direct packets to the victim.

The handshake handles the first kind of attack by responding with a small WHOAREYOU packet

whenever any request is received from an unknown endpoint. This is safe because the

adversary's packet is always larger than the WHOAREYOU response, removing the incentive

for the attack. To make the countermeasure work, implementations must keep session secrets

not just per node ID, but also per node IP.

The second kind of attack--- installing the victim as a node ---is handled by requiring

that implementations mustn't answer queries with nodes whose liveness hasn't been

verified. When a node is added to the Kademlia table, it must pass at least one check on

the IP declared in the node record before it can be returned in a NODES response.

An adversary may also try to replay previously sent/seen packets to impersonate a node or

disturb the operation of the protocol. Session keys per node-ID/IP generally prevent

replay across sessions. The `request-id`, mirrored in response packets, prevents replay of

responses within a session.

## The Topic Index

Using FINDNODE queries with appropriately chosen targets, the entire DHT can be sampled by

a random walk to find all other participants. When building a distributed application, it

is often desirable to restrict the search to participants which provide a certain service.

A simple solution to this problem would be to simply split up the network and require

participation in many smaller application-specific networks. However, such networks are

hard to bootstrap and also more vulnerable to attacks which could isolate nodes.

The topic index provides discovery by provided service in a different way. Nodes maintain

a single node table tracking their neighbors and advertise 'topics' on nodes found by

randomly walking the DHT. While the 'global' topic index can be also spammed, it makes

complete isolation a lot harder. To prevent nodes interested in a certain topic from

finding each other, the entire discovery network would have to be overpowered.

To make the index useful, searching for nodes by topic must be efficient regardless of the

number of advertisers. This is achieved by estimating the topic 'radius', i.e. the

percentage of all live nodes which are advertising the topic. Advertisement and search

activities are restricted to a region of DHT address space around the topic's 'center'.

We also want the index to satisfy another property: When a topic advertisement is placed,

it should last for a well-defined amount of time. This ensures nodes may rely on their

advertisements staying placed rather than worrying about keeping them alive.

Finally, the index should consume limited resources. Just as the node table is limited in

number and size of buckets, the size of the index data structure on each node is limited.

### Why should advertisers wait?

Advertisers must wait a certain amount of time before they can be registered. Enforcing

this time limit prevents misuse of the topic index because any topic must be important

enough to outweigh the cost of waiting. Imagine a group phone call: announcing the

participants of the call using topic advertisement isn't a good use of the system because

the topic exists only for a short time and will have very few participants. The waiting

time prevents using the index for this purpose because the call might already be over

before everyone could get registered.

### Dealing with Topic Spam

Our model is based on the following assumptions:

- Anyone can place their own advertisements under any topics and the rate of placing ads

is not limited globally. The number of active ads for any node is roughly proportional

to the resources (network bandwidth, mostly) spent on advertising.

- Honest actors whose purpose is to connect to other honest actors will spend an adequate

amount of efforts on registering and searching for ads, depending on the rate of newly

established connections they are targeting. If the given topic is used only by honest

actors, a few registrations per minute will be satisfactory, regardless of the size of

the subnetwork.

- Dishonest actors may want to place an excessive amount of ads just to disrupt the

discovery service. This will reduce the effectiveness of honest registration efforts by

increasing the topic radius and/or topic queue waiting times. If the attacker(s) can

place a comparable amount or more ads than all honest actors combined then the rate of

new (useful) connections established throughout the network will reduce proportionally

to the `honest / (dishonest + honest)` registration rates.

This adverse effect can be countered by honest actors increasing their registration and

search efforts. Fortunately, the rate of established connections between them will

increase proportionally both with increased honest registration and search efforts. If

both are increased in response to an attack, the required factor of increased efforts from

honest actors is proportional to the square root of the attacker's efforts.

### Detecting a useless registration attack

In the case of a symmetrical protocol, where nodes are both searching and advertising

under the same topic, it is easy to detect when most of the found ads turn out to be

useless and increase both registration and query frequency. It is a bit harder but still

possible with asymmetrical (client-server) protocols, where only clients can easily detect

useless registrations, while advertisers (servers) do not have a direct way of detecting

when they should increase their advertising efforts. One possible solution is for servers

to also act as clients just to test the server capabilities of other advertisers. It is

also possible to implement a feedback system between trusted clients and servers.

# References

- Petar Maymounkov and David Mazières.

\*Kademlia: A Peer-to-peer Information System Based on the XOR Metric.\* 2002.\

<https://www.scs.stanford.edu/~dm/home/papers/kpos.pdf>

- Atul Singh, Tsuen-Wan “Johnny” Ngan, Peter Druschel, Dan S. Wallach.

\*Eclipse Attacks on Overlay Networks: Threats and Defenses\*. 2006.\

<https://www.cs.rice.edu/~dwallach/pub/eclipse-infocom06.pdf>

- Ingmar Baumgart and Sebastian Mies.

\*S/Kademlia: A Practicable Approach Towards Secure Key-Based Routing.\* 2007.\

<https://telematics.tm.kit.edu/publications/Files/267/SKademlia\_2007.pdf>

- Xin Sun, Ruben Torres and Sanjay Rao. \*Feasiblity of DDoS Attacks with P2P Systems and

Prevention through Robust Membership Management.\* 2007.\

<https://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1357&context=ecetr>

- Erik Hjelmvik, Wolfgang John. \*Breaking and Improving Protocol Obfuscation.\* 2010.\

<https://internetstiftelsen.se/docs/hjelmvik\_breaking.pdf>

- Adam Langley, Wan-Teh Chang. \*QUIC Crypto\*. 2016.\

<https://docs.google.com/document/d/1g5nIXAIkN\_Y-7XJW5K45IblHd\_L2f5LTaDUDwvZ5L6g>

- W3C Credentials Community Group. \*Decentralized Identifiers (DIDs) Spec.\* 2017.\

<https://w3c-ccg.github.io/did-spec>

- Seoung Kyun Kim, Zane Ma, Siddharth Murali, Joshua Mason, Andrew Miller, Michael Bailey.

\*Measuring Ethereum Network Peers\*. 2018.\

<http://mdbailey.ece.illinois.edu/publications/imc18\_ethereum.pdf>

- Yuval Marcus, Ethan Heilman, Sharon Goldberg.

\*Low-Resource Eclipse Attacks on Ethereum’s Peer-to-Peer Network.\* 2018.\

<https://eprint.iacr.org/2018/236.pdf>

[wire protocol]: ./discv5-wire.md

[topic-based node index]: ./discv5-theory.md#topic-advertisement

[node records]: ../enr.md

# Node Discovery Protocol v5 - Theory

\*\*Protocol version v5.1\*\*

This document explains the algorithms and data structures used by the protocol.

## Nodes, Records and Distances

A participant in the Node Discovery Protocol is represented by a 'node record' as defined

in [EIP-778]. The node record keeps arbitrary information about the node. For the purposes

of this protocol, the node must at least provide an IP address (`"ip"` or `"ip6"` key) and

UDP port (`"udp"` key) in order to have it's record relayed in the DHT.

Node records are signed according to an 'identity scheme'. Any scheme can be used with

Node Discovery Protocol, and nodes using different schemes can communicate.

The identity scheme of a node record defines how a 32-byte 'node ID' is derived from the

information contained in the record. The 'distance' between two node IDs is the bitwise

XOR of the IDs, taken as the number.

distance(n₁, n₂) = n₁ XOR n₂

In many situations, the logarithmic distance (i.e. length of common prefix in bits) is

used in place of the actual distance.

logdistance(n₁, n₂) = log2(distance(n₁, n₂))

### Maintaining The Local Node Record

Participants should update their record, increase the sequence number and sign a new

version of the record whenever their information changes. This is especially important for

changes to the node's IP address and port. Implementations should determine the external

endpoint (the Internet-facing IP address and port on which the node can be reached) and

include it in their record.

If communication flows through a NAT device, the UPnP/NAT-PMP protocols or the mirrored

UDP envelope IP and port found in the [PONG] message can be used to determine the external

IP address and port.

If the endpoint cannot be determined (e.g. when the NAT doesn't support 'full-cone'

translation), implementations should omit IP address and UDP port from the record.

## Sessions

Discovery communication is encrypted and authenticated using session keys, established in

the handshake. Since every node participating in the network acts as both client and

server, a handshake can be initiated by either side of communication at any time.

### Handshake Steps

#### Step 1: Node A sends message packet

In the following definitions, we assume that node A wishes to communicate with node B,

e.g. to send a FINDNODE message. Node A must have a copy of node B's record in order to

communicate with it.

If node A has session keys from prior communication with B, it encrypts its request with

those keys. If no keys are known, it initiates the handshake by sending an ordinary

message packet with random message content.

A -> B FINDNODE message packet encrypted with unknown key

#### Step 2: Node B responds with challenge

Node B receives the message packet and extracts the source node ID from the packet header.

If node B has session keys from prior communication with A, it attempts to decrypt the

message data. If decryption and authentication of the message succeeds, there is no need

for a handshake and node B can simply respond to the request.

If node B does not have session keys or decryption is not successful, it must initiate a

handshake by responding with a [WHOAREYOU packet].

It first generates a unique `id-nonce` value and includes it in the packet. Node B also

checks if it has a copy of node A's record. If it does, it also includes the sequence

number of this record in the challenge packet, otherwise it sets the `enr-seq` field to

zero.

Node B must also store the A's record and the WHOAREYOU challenge for a short duration

after sending it to node A because they will be needed again in step 4.

A <- B WHOAREYOU packet including id-nonce, enr-seq

#### Step 3: Node A processes the challenge

Node A receives the challenge sent by node B, which confirms that node B is alive and is

ready to perform the handshake. The challenge can be traced back to the request packet

which solicited it by checking the `nonce`, which mirrors the request packet's `nonce`.

Node A proceeds with the handshake by re-sending the FINDNODE request as a [handshake

message packet]. This packet contains three parts in addition to the message:

`id-signature`, `ephemeral-pubkey` and `record`.

The handshake uses the unmasked WHOAREYOU challenge as an input:

challenge-data = masking-iv || static-header || authdata

Node A can now derive the new session keys. To do so, it first generates an ephemeral key

pair on the elliptic curve used by node B's identity scheme. As an example, let's assume

the node record of B uses the "v4" scheme. In this case the `ephemeral-pubkey` will be a

public key on the secp256k1 curve.

ephemeral-key = random private key generated by node A

ephemeral-pubkey = public key corresponding to ephemeral-key

The ephemeral key is used to perform Diffie-Hellman key agreement with node B's static

public key and the session keys are derived from it using the HKDF key derivation

function.

dest-pubkey = public key corresponding to node B's static private key

secret = ecdh(ephemeral-key, dest-pubkey)

kdf-info = "discovery v5 key agreement" || node-id-A || node-id-B

prk = HKDF-Extract(secret, challenge-data)

key-data = HKDF-Expand(prk, kdf-info)

initiator-key = key-data[:16]

recipient-key = key-data[16:]

Node A creates the `id-signature`, which proves that it controls the private key which

signed its node record. The signature also prevents replay of the handshake.

id-signature-text = "discovery v5 identity proof"

id-signature-input = id-signature-text || challenge-data || ephemeral-pubkey || node-id-B

id-signature = id\_sign(sha256(id-signature-input))

Finally, node A compares the `enr-seq` element of the WHOAREYOU challenge against its own

node record sequence number. If the sequence number in the challenge is lower, it includes

its record into the handshake message packet.

The request is now re-sent, with the message encrypted using the new session keys.

A -> B FINDNODE handshake message packet, encrypted with new initiator-key

#### Step 4: Node B receives handshake message

When node B receives the handshake message packet, it first loads the node record and

WHOAREYOU challenge which it sent and stored earlier.

If node B did not have the node record of node A, the handshake message packet must

contain a node record. A record may also be present if node A determined that its record

is newer than B's current copy. If the packet contains a node record, B must first

validate it by checking the record's signature.

Node B then verifies the `id-signature` against the identity public key of A's record.

After that, B can perform the key derivation using its own static private key and the

`ephemeral-pubkey` from the handshake packet. Using the resulting session keys, it

attempts to decrypt the message contained in the packet.

If the message can be decrypted and authenticated, Node B considers the new session keys

valid and responds to the message. In our example case, the response is a `NODES` message:

A <- B NODES encrypted with new recipient-key

#### Step 5: Node A receives response message

Node A receives the message packet response and authenticates/decrypts it with the new

session keys. If decryption/authentication succeeds, node B's identity is verified and

node A also considers the new session keys valid.

### Identity-Specific Cryptography in the Handshake

Establishment of session keys is dependent on the [identity scheme] used by the recipient

(i.e. the node which sends WHOAREYOU). Likewise, the signature over `id-sig-input` is made

by the identity key of the initiator. It is not required that initiator and recipient use

the same identity scheme in their respective node records. Implementations must be able to

perform the handshake for all supported identity schemes.

At this time, the only supported identity scheme is "v4".

`id\_sign(hash)` creates a signature over `hash` using the node's static private key. The

signature is encoded as the 64-byte array `r || s`, i.e. as the concatenation of the

signature values.

`ecdh(pubkey, privkey)` creates a secret through elliptic-curve Diffie-Hellman key

agreement. The public key is multiplied by the private key to create a secret ephemeral

key `eph = pubkey \* privkey`. The 33-byte secret output is `y || eph.x` where `y` is

`0x02` when `eph.y` is even or `0x03` when `eph.y` is odd.

### Handshake Implementation Considerations

Since a handshake may happen at any time, UDP packets may be reordered by transmitting

networking equipment, implementations must deal with certain subtleties regarding the

handshake.

In general, implementations should keep a reference to all sent request packets until the

request either times out, is answered by the corresponding response packet or answered by

WHOAREYOU. If WHOAREYOU is received as the answer to a request, the request must be

re-sent as a handshake packet.

If an implementation supports sending concurrent requests, multiple responses may be

pending when WHOAREYOU is received, as in the following example:

A -> B FINDNODE

A -> B PING

A -> B TOPICQUERY

A <- B WHOAREYOU (nonce references PING)

When this happens, all buffered requests can be considered invalid (the remote end cannot

decrypt them) and the packet referenced by the WHOAREYOU `nonce` (in this example: PING)

must be re-sent as a handshake. When the response to the re-sent is received, the new

session is established and other pending requests (example: FINDNODE, TOPICQUERY) may be

re-sent.

Note that WHOAREYOU is only ever valid as a response to a previously sent request. If

WHOAREYOU is received but no requests are pending, the handshake attempt can be ignored.

Another important issue is the processing of message packets while a challenge is

received: consider the case where node A has sent a packet that B cannot decrypt, and B

has responded with WHOAREYOU.

A -> B FINDNODE

A <- B WHOAREYOU

Node B is now waiting for a handshake message packet to complete the new session, but

instead receives another ordinary message packet.

A -> B ORDINARY MESSAGE PACKET

In this case, implementations should respond with a new WHOAREYOU challenge referencing

the message packet.

### Session Cache

Nodes should store session keys for communication with other recently-seen nodes. Since

sessions are ephemeral and can be re-established whenever necessary, it is sufficient to

store a limited number of sessions in an in-memory LRU cache.

To prevent IP spoofing attacks, implementations must ensure that session secrets and the

handshake are tied to a specific UDP endpoint. This is simple to implement by using the

node ID and IP/port as the 'key' into the in-memory session cache. When a node switches

endpoints, e.g. when roaming between different wireless networks, sessions will to be

re-established by handshaking again. This requires no effort on behalf of the roaming node

because the recipients of protocol messages will simply refuse to decrypt messages from

the new endpoint and reply with WHOAREYOU.

The number of messages which can be encrypted with a certain session key is limited

because encryption of each message requires a unique nonce for AES-GCM. In addition to the

keys, the session cache must also keep track of the count of outgoing messages to ensure

the uniqueness of nonce values. Since the wire protocol uses 96 bit AES-GCM nonces, it is

strongly recommended to generate them by encoding the current outgoing message count into

the first 32 bits of the nonce and filling the remaining 64 bits with random data

generated by a cryptographically secure random number generator.

## Node Table

Nodes keep information about other nodes in their neighborhood. Neighbor nodes are stored

in a routing table consisting of 'k-buckets'. For each `0 ≤ i < 256`, every node keeps a

k-bucket for nodes of `logdistance(self, n) == i`. The Node Discovery Protocol uses `k =

16`, i.e. every k-bucket contains up to 16 node entries. The entries are sorted by time

last seen — least-recently seen node at the head, most-recently seen at the tail.

Whenever a new node N₁ is encountered, it can be inserted into the corresponding bucket.

If the bucket contains less than `k` entries N₁ can simply be added as the first entry. If

the bucket already contains `k` entries, the liveness of the least recently seen node in

the bucket, N₂, needs to be revalidated. If no reply is received from N₂ it is considered

dead, removed and N₁ added to the front of the bucket.

Neighbors of very low distance are unlikely to occur in practice. Implementations may omit

k-buckets for low distances.

### Table Maintenance In Practice

Nodes are expected to keep track of their close neighbors and regularly refresh their

information. To do so, a lookup targeting the least recently refreshed bucket should be

performed at regular intervals.

Checking node liveness whenever a node is to be added to a bucket is impractical and

creates a DoS vector. Implementations should perform liveness checks asynchronously with

bucket addition and occasionally verify that a random node in a random bucket is live by

sending [PING]. When the PONG response indicates that a new version of the node record is

available, the liveness check should pull the new record and update it in the local table.

If a node's liveness has been verified many times, implementations may consider occasional

non-responsiveness permissible and assume the node is live.

When responding to FINDNODE, implementations must avoid relaying any nodes whose liveness

has not been verified. This is easy to achieve by storing an additional flag per node in

the table, tracking whether the node has ever successfully responded to a PING request.

In order to keep all k-bucket positions occupied even when bucket members fail liveness

checks, it is strongly recommended to maintain a 'replacement cache' alongside each

bucket. This cache holds recently-seen node which would fall into the corresponding bucket

but cannot become a member of the bucket because it is already at capacity. Once a bucket

member becomes unresponsive, a replacement can be chosen from the cache.

### Lookup

A 'lookup' locates the `k` closest nodes to a node ID.

The lookup initiator starts by picking `α` closest nodes to the target it knows of from

the local table. The initiator then sends [FINDNODE] requests to those nodes. `α` is an

implementation-defined concurrency parameter, typically `3`. As NEIGHBORS responses are

received, the initiator resends FINDNODE to nodes it has learned about from previous

queries. Of the `k` nodes the initiator has heard of closest to the target, it picks `α`

that it has not yet queried and sends FINDNODE to them. The lookup terminates when the

initiator has queried and gotten responses from the `k` closest nodes it has seen.

To improve the resilience of lookups against adversarial nodes, the algorithm may be

adapted to perform network traversal on multiple disjoint paths. Not only does this

approach benefit security, it also improves effectiveness because more nodes are visited

during a single lookup. The initial `k` closest nodes are partitioned into multiple

independent 'path' buckets, and ​concurrent FINDNODE​ requests executed as described above,

with one difference: results discovered on one path are not reused on another, i.e. each

path attempts to reach the closest nodes to the lookup target independently without

reusing intermediate results found on another path. Note that it is still necessary to

track previously asked nodes across all paths to keep the paths disjoint.

### Lookup Protocol

This section shows how the wire protocol messages can be used to perform a lookup

interaction against a single node.

Node `A` is looking for target `x`. It selects node `B` from the local table or

intermediate lookup results. To query for nodes close to `x` on `B`, node `A` computes the

query distance `d = logdistance(B, x)` and sends its request.

A -> B FINDNODE [d]

Node `B` responds with multiple nodes messages containing the nodes at the queried

distance.

A <- B NODES [N₁, N₂, N₃]

A <- B NODES [N₄, N₅]

Depending on the value of `d` and the content of `B`s table, the response to the initial

query might contain very few nodes or no nodes at all. Should this be the case, `A` varies

the distance to retrieve more nodes from adjacent k-buckets on `B`:

A -> B FINDNODE [d+1]

`B` responds with more nodes:

A <- B NODES [N₆, N₇]

Node `A` now sorts all received nodes by distance to the lookup target and proceeds by

repeating the lookup procedure on another, closer node.

## Topic Advertisement

The topic advertisement subsystem indexes participants by their provided services. A

node's provided services are identified by arbitrary strings called 'topics'. A node

providing a certain service is said to 'place an ad' for itself when it makes itself

discoverable under that topic. Depending on the needs of the application, a node can

advertise multiple topics or no topics at all. Every node participating in the discovery

protocol acts as an advertisement medium, meaning that it accepts topic ads from other

nodes and later returns them to nodes searching for the same topic.

### Topic Table

Nodes store ads for any number of topics and a limited number of ads for each topic. The

data structure holding advertisements is called the 'topic table'. The list of ads for a

particular topic is called the 'topic queue' because it functions like a FIFO queue of

limited length. The image below depicts a topic table containing three queues. The queue

for topic `T₁` is at capacity.

![topic table](./img/topic-queue-diagram.png)

The queue size limit is implementation-defined. Implementations should place a global

limit on the number of ads in the topic table regardless of the topic queue which contains

them. Reasonable limits are 100 ads per queue and 50000 ads across all queues. Since ENRs

are at most 300 bytes in size, these limits ensure that a full topic table consumes

approximately 15MB of memory.

Any node may appear at most once in any topic queue, that is, registration of a node which

is already registered for a given topic fails. Implementations may impose other

restrictions on the table, such as restrictions on the number of IP-addresses in a certain

range or number of occurrences of the same node across queues.

### Tickets

Ads should remain in the queue for a constant amount of time, the `target-ad-lifetime`. To

maintain this guarantee, new registrations are throttled and registrants must wait for a

certain amount of time before they are admitted. When a node attempts to place an ad, it

receives a 'ticket' which tells them how long they must wait before they will be accepted.

It is up to the registrant node to keep the ticket and present it to the advertisement

medium when the waiting time has elapsed.

The waiting time constant is:

target-ad-lifetime = 15min

The assigned waiting time for any registration attempt is determined according to the

following rules:

- When the table is full, the waiting time is assigned based on the lifetime of the oldest

ad across the whole table, i.e. the registrant must wait for a table slot to become

available.

- When the topic queue is full, the waiting time depends on the lifetime of the oldest ad

in the queue. The assigned time is `target-ad-lifetime - oldest-ad-lifetime` in this

case.

- Otherwise the ad may be placed immediately.

Tickets are opaque objects storing arbitrary information determined by the issuing node.

While details of encoding and ticket validation are up to the implementation, tickets must

contain enough information to verify that:

- The node attempting to use the ticket is the node which requested it.

- The ticket is valid for a single topic only.

- The ticket can only be used within the registration window.

- The ticket can't be used more than once.

Implementations may choose to include arbitrary other information in the ticket, such as

the cumulative wait time spent by the advertiser. A practical way to handle tickets is to

encrypt and authenticate them with a dedicated secret key:

ticket = aesgcm\_encrypt(ticket-key, ticket-nonce, ticket-pt, '')

ticket-pt = [src-node-id, src-ip, topic, req-time, wait-time, cum-wait-time]

src-node-id = node ID that requested the ticket

src-ip = IP address that requested the ticket

topic = the topic that ticket is valid for

req-time = absolute time of REGTOPIC request

wait-time = waiting time assigned when ticket was created

cum-wait = cumulative waiting time of this node

### Registration Window

The image below depicts a single ticket's validity over time. When the ticket is issued,

the node keeping it must wait until the registration window opens. The length of the

registration window is 10 seconds. The ticket becomes invalid after the registration

window has passed.

![ticket validity over time](./img/ticket-validity.png)

Since all ticket waiting times are assigned to expire when a slot in the queue opens, the

advertisement medium may receive multiple valid tickets during the registration window and

must choose one of them to be admitted in the topic queue. The winning node is notified

using a [REGCONFIRMATION] response.

Picking the winner can be achieved by keeping track of a single 'next ticket' per queue

during the registration window. Whenever a new ticket is submitted, first determine its

validity and compare it against the current 'next ticket' to determine which of the two is

better according to an implementation-defined metric such as the cumulative wait time

stored in the ticket.

### Advertisement Protocol

This section explains how the topic-related protocol messages are used to place an ad.

Let us assume that node `A` provides topic `T`. It selects node `C` as advertisement

medium and wants to register an ad, so that when node `B` (who is searching for topic `T`)

asks `C`, `C` can return the registration entry of `A` to `B`.

Node `A` first attempts to register without a ticket by sending [REGTOPIC] to `C`.

A -> C REGTOPIC [T, ""]

`C` replies with a ticket and waiting time.

A <- C TICKET [ticket, wait-time]

Node `A` now waits for the duration of the waiting time. When the wait is over, `A` sends

another registration request including the ticket. `C` does not need to remember its

issued tickets since the ticket is authenticated and contains enough information for `C`

to determine its validity.

A -> C REGTOPIC [T, ticket]

Node `C` replies with another ticket. Node `A` must keep this ticket in place of the

earlier one, and must also be prepared to handle a confirmation call in case registration

was successful.

A <- C TICKET [ticket, wait-time]

Node `C` waits for the registration window to end on the queue and selects `A` as the node

which is registered. Node `C` places `A` into the topic queue for `T` and sends a

[REGCONFIRMATION] response.

A <- C REGCONFIRMATION [T]

### Ad Placement And Topic Radius

Since every node may act as an advertisement medium for any topic, advertisers and nodes

looking for ads must agree on a scheme by which ads for a topic are distributed. When the

number of nodes advertising a topic is at least a certain percentage of the whole

discovery network (rough estimate: at least 1%), ads may simply be placed on random nodes

because searching for the topic on randomly selected will locate the ads quickly enough.

However, topic search should be fast even when the number of advertisers for a topic is

much smaller than the number of all live nodes. Advertisers and searchers must agree on a

subset of nodes to serve as advertisement media for the topic. This subset is simply a

region of node ID address space, consisting of nodes whose Kademlia address is within a

certain distance to the topic hash `sha256(T)`. This distance is called the 'topic

radius'.

Example: for a topic `f3b2529e...` with a radius of 2^240, the subset covers all nodes

whose IDs have prefix `f3b2...`. A radius of 2^256 means the entire network, in which case

advertisements are distributed uniformly among all nodes. The diagram below depicts a

region of address space with the topic hash `t` in the middle and several nodes close to

`t` surrounding it. Dots above the nodes represent entries in the node's queue for the

topic.

![diagram explaining the topic radius concept](./img/topic-radius-diagram.png)

To place their ads, participants simply perform a random walk within the currently

estimated radius and run the advertisement protocol by collecting tickets from all nodes

encountered during the walk and using them when their waiting time is over.

### Topic Radius Estimation

Advertisers must estimate the topic radius continuously in order to place their ads on

nodes where they will be found. The radius mustn't fall below a certain size because

restricting registration to too few nodes leaves the topic vulnerable to censorship and

leads to long waiting times. If the radius were too large, searching nodes would take too

long to find the ads.

Estimating the radius uses the waiting time as an indicator of how many other nodes are

attempting to place ads in a certain region. This is achieved by keeping track of the

average time to successful registration within segments of the address space surrounding

the topic hash. Advertisers initially assume the radius is 2^256, i.e. the entire network.

As tickets are collected, the advertiser samples the time it takes to place an ad in each

segment and adjusts the radius such that registration at the chosen distance takes

approximately `target-ad-lifetime / 2` to complete.

## Topic Search

Finding nodes that provide a certain topic is a continuous process which reads the content

of topic queues inside the approximated topic radius. This is a much simpler process than

topic advertisement because collecting tickets and waiting on them is not required.

To find nodes for a topic, the searcher generates random node IDs inside the estimated

topic radius and performs Kademlia lookups for these IDs. All (intermediate) nodes

encountered during lookup are asked for topic queue entries using the [TOPICQUERY] packet.

Radius estimation for topic search is similar to the estimation procedure for

advertisement, but samples the average number of results from TOPICQUERY instead of

average time to registration. The radius estimation value can be shared with the

registration algorithm if the same topic is being registered and searched for.

[EIP-778]: ../enr.md

[identity scheme]: ../enr.md#record-structure

[handshake message packet]: ./discv5-wire.md#handshake-message-packet-flag--2

[WHOAREYOU packet]: ./discv5-wire.md#whoareyou-packet-flag--1

[PING]: ./discv5-wire.md#ping-request-0x01

[PONG]: ./discv5-wire.md#pong-response-0x02

[FINDNODE]: ./discv5-wire.md#findnode-request-0x03

[REGTOPIC]: ./discv5-wire.md#regtopic-request-0x07

[REGCONFIRMATION]: ./discv5-wire.md#regconfirmation-response-0x09

[TOPICQUERY]: ./discv5-wire.md#topicquery-request-0x0a

# Test Vectors

This document provides a collection of test vectors for the Discovery v5 wire protocol

aimed to aid new implementations conform to the specification.

## Packet Encodings

This section provides test vectors for the different packet types. Your implementation

should load the `node-b-key` and then be able to decrypt and authenticate these as-is.

The secp256k1 private keys used here are:

node-a-key = 0xeef77acb6c6a6eebc5b363a475ac583ec7eccdb42b6481424c60f59aa326547f

node-b-key = 0x66fb62bfbd66b9177a138c1e5cddbe4f7c30c343e94e68df8769459cb1cde628

Ping message packet (flag 0):

# src-node-id = 0xaaaa8419e9f49d0083561b48287df592939a8d19947d8c0ef88f2a4856a69fbb

# dest-node-id = 0xbbbb9d047f0488c0b5a93c1c3f2d8bafc7c8ff337024a55434a0d0555de64db9

# nonce = 0xffffffffffffffffffffffff

# read-key = 0x00000000000000000000000000000000

# ping.req-id = 0x00000001

# ping.enr-seq = 2

00000000000000000000000000000000088b3d4342774649325f313964a39e55

ea96c005ad52be8c7560413a7008f16c9e6d2f43bbea8814a546b7409ce783d3

4c4f53245d08dab84102ed931f66d1492acb308fa1c6715b9d139b81acbdcc

WHOAREYOU packet (flag 1):

# src-node-id = 0xaaaa8419e9f49d0083561b48287df592939a8d19947d8c0ef88f2a4856a69fbb

# dest-node-id = 0xbbbb9d047f0488c0b5a93c1c3f2d8bafc7c8ff337024a55434a0d0555de64db9

# whoareyou.challenge-data = 0x000000000000000000000000000000006469736376350001010102030405060708090a0b0c00180102030405060708090a0b0c0d0e0f100000000000000000

# whoareyou.request-nonce = 0x0102030405060708090a0b0c

# whoareyou.id-nonce = 0x0102030405060708090a0b0c0d0e0f10

# whoareyou.enr-seq = 0

00000000000000000000000000000000088b3d434277464933a1ccc59f5967ad

1d6035f15e528627dde75cd68292f9e6c27d6b66c8100a873fcbaed4e16b8d

Ping handshake packet (flag 2):

# src-node-id = 0xaaaa8419e9f49d0083561b48287df592939a8d19947d8c0ef88f2a4856a69fbb

# dest-node-id = 0xbbbb9d047f0488c0b5a93c1c3f2d8bafc7c8ff337024a55434a0d0555de64db9

# nonce = 0xffffffffffffffffffffffff

# read-key = 0x4f9fac6de7567d1e3b1241dffe90f662

# ping.req-id = 0x00000001

# ping.enr-seq = 1

#

# handshake inputs:

#

# whoareyou.challenge-data = 0x000000000000000000000000000000006469736376350001010102030405060708090a0b0c00180102030405060708090a0b0c0d0e0f100000000000000001

# whoareyou.request-nonce = 0x0102030405060708090a0b0c

# whoareyou.id-nonce = 0x0102030405060708090a0b0c0d0e0f10

# whoareyou.enr-seq = 1

# ephemeral-key = 0x0288ef00023598499cb6c940146d050d2b1fb914198c327f76aad590bead68b6

# ephemeral-pubkey = 0x039a003ba6517b473fa0cd74aefe99dadfdb34627f90fec6362df85803908f53a5

00000000000000000000000000000000088b3d4342774649305f313964a39e55

ea96c005ad521d8c7560413a7008f16c9e6d2f43bbea8814a546b7409ce783d3

4c4f53245d08da4bb252012b2cba3f4f374a90a75cff91f142fa9be3e0a5f3ef

268ccb9065aeecfd67a999e7fdc137e062b2ec4a0eb92947f0d9a74bfbf44dfb

a776b21301f8b65efd5796706adff216ab862a9186875f9494150c4ae06fa4d1

f0396c93f215fa4ef524f1eadf5f0f4126b79336671cbcf7a885b1f8bd2a5d83

9cf8

Ping handshake message packet (flag 2, with ENR):

# src-node-id = 0xaaaa8419e9f49d0083561b48287df592939a8d19947d8c0ef88f2a4856a69fbb

# dest-node-id = 0xbbbb9d047f0488c0b5a93c1c3f2d8bafc7c8ff337024a55434a0d0555de64db9

# nonce = 0xffffffffffffffffffffffff

# read-key = 0x53b1c075f41876423154e157470c2f48

# ping.req-id = 0x00000001

# ping.enr-seq = 1

#

# handshake inputs:

#

# whoareyou.challenge-data = 0x000000000000000000000000000000006469736376350001010102030405060708090a0b0c00180102030405060708090a0b0c0d0e0f100000000000000000

# whoareyou.request-nonce = 0x0102030405060708090a0b0c

# whoareyou.id-nonce = 0x0102030405060708090a0b0c0d0e0f10

# whoareyou.enr-seq = 0

# ephemeral-key = 0x0288ef00023598499cb6c940146d050d2b1fb914198c327f76aad590bead68b6

# ephemeral-pubkey = 0x039a003ba6517b473fa0cd74aefe99dadfdb34627f90fec6362df85803908f53a5

00000000000000000000000000000000088b3d4342774649305f313964a39e55

ea96c005ad539c8c7560413a7008f16c9e6d2f43bbea8814a546b7409ce783d3

4c4f53245d08da4bb23698868350aaad22e3ab8dd034f548a1c43cd246be9856

2fafa0a1fa86d8e7a3b95ae78cc2b988ded6a5b59eb83ad58097252188b902b2

1481e30e5e285f19735796706adff216ab862a9186875f9494150c4ae06fa4d1

f0396c93f215fa4ef524e0ed04c3c21e39b1868e1ca8105e585ec17315e755e6

cfc4dd6cb7fd8e1a1f55e49b4b5eb024221482105346f3c82b15fdaae36a3bb1

2a494683b4a3c7f2ae41306252fed84785e2bbff3b022812d0882f06978df84a

80d443972213342d04b9048fc3b1d5fcb1df0f822152eced6da4d3f6df27e70e

4539717307a0208cd208d65093ccab5aa596a34d7511401987662d8cf62b1394

71

## Cryptographic Primitives

This section provides test vectors for the currently supported "v4" identity scheme.

### ECDH

The ECDH function takes the elliptic-curve scalar multiplication of a public key and a

private key. The wire protocol describes this process.

public-key = 0x039961e4c2356d61bedb83052c115d311acb3a96f5777296dcf297351130266231

secret-key = 0xfb757dc581730490a1d7a00deea65e9b1936924caaea8f44d476014856b68736

This output is the result of the ECDH function which will be used by the KDF.

shared-secret = 0x033b11a2a1f214567e1537ce5e509ffd9b21373247f2a3ff6841f4976f53165e7e

### Key Derivation

This test vector checks the complete key derivation as used by the handshake.

ephemeral-key = 0xfb757dc581730490a1d7a00deea65e9b1936924caaea8f44d476014856b68736

dest-pubkey = 0x0317931e6e0840220642f230037d285d122bc59063221ef3226b1f403ddc69ca91

node-id-a = 0xaaaa8419e9f49d0083561b48287df592939a8d19947d8c0ef88f2a4856a69fbb

node-id-b = 0xbbbb9d047f0488c0b5a93c1c3f2d8bafc7c8ff337024a55434a0d0555de64db9

challenge-data = 0x000000000000000000000000000000006469736376350001010102030405060708090a0b0c00180102030405060708090a0b0c0d0e0f100000000000000000

The expected outputs, resulting from the HKDF-EXPAND function.

initiator-key = 0xdccc82d81bd610f4f76d3ebe97a40571

recipient-key = 0xac74bb8773749920b0d3a8881c173ec5

### ID Nonce Signing

This test vector checks the ID signature as used by the handshake.

The `static-key` is the secp256k1 private key used for signing.

static-key = 0xfb757dc581730490a1d7a00deea65e9b1936924caaea8f44d476014856b68736

challenge-data = 0x000000000000000000000000000000006469736376350001010102030405060708090a0b0c00180102030405060708090a0b0c0d0e0f100000000000000000

ephemeral-pubkey = 0x039961e4c2356d61bedb83052c115d311acb3a96f5777296dcf297351130266231

node-id-B = 0xbbbb9d047f0488c0b5a93c1c3f2d8bafc7c8ff337024a55434a0d0555de64db9

The expected output is the `id-signature`. You can also apply this test vector in reverse

by verifying the signature against the inputs above.

id-signature = 0x94852a1e2318c4e5e9d422c98eaf19d1d90d876b29cd06ca7cb7546d0fff7b484fe86c09a064fe72bdbef73ba8e9c34df0cd2b53e9d65528c2c7f336d5dfc6e6

### Encryption/Decryption

This test vector demonstrates the `AES\_GCM` encryption/decryption used in the wire

protocol.

encryption-key: 0x9f2d77db7004bf8a1a85107ac686990b

nonce: 0x27b5af763c446acd2749fe8e

pt: 0x01c20101

ad: 0x93a7400fa0d6a694ebc24d5cf570f65d04215b6ac00757875e3f3a5f42107903

Note that the 16 byte MAC is prepended to the ciphertext.

message-ciphertext: 0xa5d12a2d94b8ccb3ba55558229867dc13bfa3648

# Node Discovery Protocol v5 - Wire Protocol

\*\*Protocol version v5.1\*\*

This document specifies the wire protocol of Node Discovery v5.

## Notation

Here we present the notation that is used throughout this document.

`[ .. , .. , .. ]`\

is recursive encoding as an RLP list\

`a || b`\

means binary concatenation of `a` and `b`\

`xor(a, b)`\

means binary XOR of `a` and `b`\

`sha256(x)`\

is the SHA256 digest of `x`\

`aesctr\_encrypt(key, iv, pt)`\

is unauthenticated AES/CTR symmetric encryption with the given `key` and `iv`.\

Size of `key` and `iv` is 16 bytes (AES-128).\

`aesgcm\_encrypt(key, nonce, pt, ad)`\

is AES-GCM encryption/authentication with the given `key`, `nonce` and additional\

authenticated data `ad`. Size of `key` is 16 bytes (AES-128), size of `nonce` 12 bytes.

## UDP Communication

Node discovery messages are sent as UDP datagrams. Since UDP is a lossy transport, packets

may be received in any order or not at all. Implementations should not re-send packets if

the recipient doesn't respond.

The maximum size of any packet is 1280 bytes. Implementations should not generate or

process packets larger than this size. Most messages are smaller than this limit by

definition, the exception being the NODES message. FINDNODE returns up to 16 records, plus

other data, and TOPICQUERY may also distribute a significantly long list of ENRs. As per

specification the maximum size of an ENR is 300 bytes. A NODES message containing all

FINDNODE response records would be at least 4800 bytes, not including additional data such

as the header. To stay below the size limit, NODES responses are sent as multiple messages

and specify the total number of responses in the message.

The minimum size of any Discovery v5 packet is 63 bytes. Implementations should reject

packets smaller than this size.

Since low-latency communication is expected, implementations should place short timeouts

on request/response interactions. Good timeout values are 500ms for a single

request/response and 1s for the handshake.

When responding to a request, the response should be sent to the UDP envelope address of

the request.

## Packet Encoding

The protocol deals with three distinct kinds of packets:

- Ordinary message packets, which carry an encrypted/authenticated message.

- WHOAREYOU packets, which are sent when the recipient of an ordinary message packet

cannot decrypt/authenticate the packet's message.

- Handshake message packets, which are sent following WHOAREYOU. These packets establish a

new session and carry handshake-related data in addition to the encrypted/authenticated

message.

In the following definitions, we assume that the sender of a packet has knowledge of its

own 256-bit node ID (`src-id`) and the node ID of the packet destination (`dest-id`). When

sending any packet except WHOAREYOU, the sender also generates a unique 96-bit `nonce`

value.

### Protocol Header

All discovery packets contain a header followed by an optional encrypted and authenticated

message.

Header information is 'masked' using symmetric encryption in order to avoid static

identification of the protocol by firewalls.

packet = masking-iv || masked-header || message

masked-header = aesctr\_encrypt(masking-key, masking-iv, header)

masking-key = dest-id[:16]

masking-iv = uint128 -- random data unique to packet

The `masked-header` contains the actual packet header, which starts with a fixed-size

`static-header`, followed by a variable-length `authdata` section (of size `authdata-size`).

header = static-header || authdata

static-header = protocol-id || version || flag || nonce || authdata-size

protocol-id = "discv5"

version = 0x0001

authdata-size = uint16 -- byte length of authdata

flag = uint8 -- packet type identifier

nonce = uint96 -- nonce of message

Decrypting the masked header data works as follows: The recipient constructs an AES/CTR

stream cipher using its own node ID (`dest-id`) as the key and taking the IV from the

packet. It can then decrypt the `static-header` and verify that `protocol-id` matches the

expected string. If it does, the recipient can read `authdata-size` and unmask the

remaining `authdata`.

Implementations should not respond to packets with mismatching `protocol-id`.

In ordinary message packets and handshake message packets, the packet contains an

authenticated message after the `authdata` section. For WHOAREYOU packets, the `message`

is empty. Implementations must generate a unique `nonce` value for every message packet.

message = aesgcm\_encrypt(initiator-key, nonce, message-pt, message-ad)

message-pt = message-type || message-data

message-ad = masking-iv || header

The `flag` field of the header identifies the kind of packet and determines the encoding

of `authdata`, which differs depending on the packet type.

### Ordinary Message Packet (`flag = 0`)

For message packets, the `authdata` section is just the source node ID.

authdata = src-id

authdata-size = 32

![message packet layout](./img/message-packet-layout.png)

### WHOAREYOU Packet (`flag = 1`)

In WHOAREYOU packets, the `authdata` section contains information for the identity

verification procedure. The `message` part of WHOAREYOU packets is always empty. The

`nonce` part of the packet must be set to the `nonce` of the message packet that caused

the WHOAREYOU response.

authdata = id-nonce || enr-seq

authdata-size = 24

id-nonce = uint128 -- random bytes

enr-seq = uint64 -- ENR sequence number of the requesting node

![whoareyou packet layout](./img/whoareyou-packet-layout.png)

### Handshake Message Packet (`flag = 2`)

For handshake message packets, the `authdata` section has variable size since public key

and signature sizes depend on the ENR identity scheme. For the "v4" identity scheme, we

assume 64-byte signature size and 33 bytes of (compressed) public key size.

`authdata` starts with a fixed-size `authdata-head` component, followed by the ID

signature, ephemeral public key and optional node record.

The `record` field may be omitted if the `enr-seq` of WHOAREYOU is recent enough, i.e.

when it matches the current sequence number of the sending node. If `enr-seq` is zero, the

record must be sent. Node records are encoded and verified as specified in [EIP-778].

Please refer to the [handshake section] for more information about the content of the

handshake packet.

authdata = authdata-head || id-signature || eph-pubkey || record

authdata-head = src-id || sig-size || eph-key-size

authdata-size = 34 + sig-size + eph-key-size + len(record)

sig-size = uint8 -- value: 64 for ID scheme "v4"

eph-key-size = uint8 -- value: 33 for ID scheme "v4"

![handshake packet layout](./img/handshake-packet-layout.png)

## Protocol Messages

This section lists all defined messages which can be sent and received. The hexadecimal

value in parentheses is the `message-type`.

The first element of every `message-data` list is the request ID. `request-id` is an RLP

byte array of length <= 8 bytes. For requests, this value is assigned by the requester.

The recipient of a message must mirror the value in the `request-id` element of the

response. The selection of appropriate values for request IDs is left to the implementation.

### PING Request (0x01)

message-data = [request-id, enr-seq]

message-type = 0x01

enr-seq = local ENR sequence number of sender

PING checks whether the recipient is alive and informs it about the sender's ENR sequence

number.

### PONG Response (0x02)

message-data = [request-id, enr-seq, recipient-ip, recipient-port]

message-type = 0x02

enr-seq = ENR sequence number of sender

recipient-ip = 16 or 4 byte IP address of the intended recipient

recipient-port = recipient UDP port, a 16-bit integer

PONG is the reply to PING.

### FINDNODE Request (0x03)

message-data = [request-id, [distance₁, distance₂, ..., distanceₙ]]

message-type = 0x03

distanceₙ = requested log2 distance, a positive integer

FINDNODE queries for nodes at the given logarithmic distances from the recipient's node

ID. When distance `0` is requested, the result set should contain the recipient's current

record.

The recipient should create the result set by collecting nodes from its local node table

according to the requested distances. Implementations should limit the number of nodes in

the result set. The recommended result limit for FINDNODE queries is 16 nodes.

### NODES Response (0x04)

message-data = [request-id, total, [ENR, ...]]

message-type = 0x04

total = total number of responses to the request

NODES is the response to a FINDNODE or TOPICQUERY message. Multiple NODES messages may be

sent as responses to a single query. Implementations may place a limit on the allowed

maximum for `total`. If exceeded, additional responses may be ignored.

When handling NODES as a response to FINDNODE, the recipient should verify that the

received nodes match the requested distances.

### TALKREQ Request (0x05)

message-data = [request-id, protocol, request]

message-type = 0x05

TALKREQ sends an application-level request. The purpose of this message is pre-negotiating

connections made through another application-specific protocol identified by `protocol`.

`protocol` and `request` are RLP byte arrays.

The recipient must respond with a TALKRESP message containing the response to the request.

If the `protocol` is unknown to the recipient, it must respond with a TALKRESP response

containing empty `response` data.

### TALKRESP Response (0x06)

message-data = [request-id, response]

message-type = 0x06

request-id = request-id of TALKREQ

TALKRESP is the response to TALKREQ. The `response` is a RLP byte array containing the

response data.

### REGTOPIC Request (0x07)

\*\*NOTE: the content and semantics of this message are not final.\*\*

\*\*Implementations should not respond to or send these messages.\*\*

message-data = [request-id, topic, ENR, ticket]

message-type = 0x07

node-record = current node record of sender

ticket = byte array containing ticket content

REGTOPIC attempts to register the sender for the given topic. If the requesting node has a

ticket from a previous registration attempt, it must present the ticket. Otherwise

`ticket` is the empty byte array (RLP: `0x80`). The ticket must be valid and its waiting

time must have elapsed before using the ticket.

REGTOPIC is always answered by a TICKET response. The requesting node may also receive a

REGCONFIRMATION response when registration is successful. It may take up to 10s for the

confirmation to be sent.

### TICKET Response (0x08)

\*\*NOTE: the content and semantics of this message are not final.\*\*

\*\*Implementations should not respond to or send these messages.\*\*

message-data = [request-id, ticket, wait-time]

message-type = 0x08

ticket = an opaque byte array representing the ticket

wait-time = time to wait before registering, in seconds

TICKET is the response to REGTOPIC. It contains a ticket which can be used to register for

the requested topic after `wait-time` has elapsed. See the [theory section on tickets] for

more information.

### REGCONFIRMATION Response (0x09)

\*\*NOTE: the content and semantics of this message are not final.\*\*

\*\*Implementations should not respond to or send these messages.\*\*

message-data = [request-id, topic]

message-type = 0x09

request-id = request-id of REGTOPIC

REGCONFIRMATION notifies the recipient about a successful registration for the given

topic. This call is sent by the advertisement medium after the time window for

registration has elapsed on a topic queue.

### TOPICQUERY Request (0x0A)

\*\*NOTE: the content and semantics of this message are not final.\*\*

\*\*Implementations should not respond to or send these messages.\*\*

message-data = [request-id, topic]

message-type = 0x0a

topic = 32-byte topic hash

TOPICQUERY requests nodes in the [topic queue] of the given topic. The recipient of this

request must send one or more NODES messages containing node records registered for the

topic.

## Test Vectors

A collection of test vectors for this specification can be found at

[discv5 wire test vectors].

[handshake section]: ./discv5-theory.md#handshake-steps

[topic queue]: ./discv5-theory.md#topic-table

[theory section on tickets]: ./discv5-theory.md#tickets

[EIP-778]: ../enr.md

[discv5 wire test vectors]: ./discv5-wire-test-vectors.md

# Node Discovery Protocol v5

\*\*Protocol version v5.1\*\*

Welcome to the Node Discovery Protocol v5 specification!

Note that this specification is a work in progress and may change incompatibly without

prior notice.

Node Discovery is a system for finding other participants in a peer-to-peer network. The

system can be used by any node, for any purpose, at no cost other than running the network

protocol and storing a limited number of other nodes' records. Any node can be used as an

entry point into the network.

The system's design is loosely inspired by the Kademlia DHT, but unlike most DHTs no

arbitrary keys and values are stored. Instead, the DHT stores and relays 'node records',

which are signed documents providing information about nodes in the network. Node

Discovery acts as a database of all live nodes in the network and performs three basic

functions:

- Sampling the set of all live participants: by walking the DHT, the network can be

enumerated.

- Searching for participants providing a certain service: Node Discovery v5 includes a

scalable facility for registering 'topic advertisements'. These advertisements can be

queried and nodes advertising a topic found.

- Authoritative resolution of node records: if a node's ID is known, the most recent

version of its record can be retrieved.

## Specification Overview

The specification has three parts:

- [discv5-wire.md] defines the wire protocol.

- [discv5-theory.md] describes the algorithms and data structures.

- [discv5-rationale.md] contains the design rationale.

## Comparison With Other Discovery Mechanisms

Systems such as MDNS/Bonjour allow finding hosts in a local-area network. The Node

Discovery Protocol is designed to work on the Internet and is most useful for applications

with a large number of participants spread across the Internet.

Systems using a rendezvous server: these systems are commonly used by desktop applications

or cloud services to connect participants to each other. While undoubtedly efficient, this

requires trust in the operator of the rendezvous server and these systems are prone to

censorship. Compared to a rendezvous server, The Node Discovery Protocol doesn't rely on a

single operator and places a small amount of trust in every participant. It becomes more

resistant to censorship as the size of the network increases and participants of multiple

distinct peer-to-peer networks can share the discovery network to further increase its

resilience.

The Achilles heel of the Node Discovery Protocol is the process of joining the network:

while any other node may be used as an entry point, such a node must first be located

through some other mechanism. Several approaches including scalable listing of initial

entry points in DNS or discovery of participants in the local network can be used for

reasonable secure entry into the network.

## Comparison With Node Discovery v4

- Topic advertisement was added.

- Arbitrary node metadata can be stored/relayed.

- Node identity crypto is extensible, use of secp256k1 keys isn't strictly required.

- The protocol no longer relies on the system clock.

- Communication is encrypted, protecting topic searches and record lookups against passive

observers.

[discv5-wire.md]: ./discv5-wire.md

[discv5-theory.md]: ./discv5-theory.md

[discv5-rationale.md]: ./discv5-rationale.md

# The "eth" ENR entry

This specification defines the "eth" ENR entry, which provides information

about the [eth capability] on a certain node.

## Entry Format

entry-key = "eth"

entry-value = [[ forkHash, forkNext ]]

At this time, the "eth" entry is a single element list containing an [EIP-2124] fork ID

value. Please see the EIP for definitions of `forkHash` and `forkNext`.

In order to be compatible with future versions of this specifications, implementations

should ignore any additional list elements in `entry-value`.

## Change Log

### EIP-2124 (May 2019)

The initial version of the "eth" entry was proposed in [EIP-2124].

[eth capability]: ../caps/eth.md

[EIP-2124]: https://eips.ethereum.org/EIPS/eip-2124

# Ethereum Node Records

This specification defines Ethereum Node Records, an open format for p2p connectivity

information. A node record usually contains the network endpoints of a node, i.e. the

node's IP addresses and ports. It also holds information about the node's purpose on the

network so others can decide whether to connect to the node.

## Record Structure

The components of a node record are:

- `signature`: cryptographic signature of record contents

- `seq`: The sequence number, a 64-bit unsigned integer. Nodes should increase the number

whenever the record changes and republish the record.

- The remainder of the record consists of arbitrary key/value pairs

A record's signature is made and validated according to an \*identity scheme\*. The identity

scheme is also responsible for deriving a node's address in the DHT.

The key/value pairs must be sorted by key and must be unique, i.e. any key may be present

only once. The keys can technically be any byte sequence, but ASCII text is preferred. Key

names in the table below have pre-defined meaning.

| Key | Value |

|:------------|:-------------------------------------------|

| `id` | name of identity scheme, e.g. "v4" |

| `secp256k1` | compressed secp256k1 public key, 33 bytes |

| `ip` | IPv4 address, 4 bytes |

| `tcp` | TCP port, big endian integer |

| `udp` | UDP port, big endian integer |

| `ip6` | IPv6 address, 16 bytes |

| `tcp6` | IPv6-specific TCP port, big endian integer |

| `udp6` | IPv6-specific UDP port, big endian integer |

All keys except `id` are optional, including IP addresses and ports. A record without

endpoint information is still valid as long as its signature is valid. If no `tcp6` /

`udp6` port is provided, the `tcp` / `udp` port applies to both IP addresses. Declaring

the same port number in both `tcp`, `tcp6` or `udp`, `udp6` should be avoided but doesn't

render the record invalid.

### RLP Encoding

The canonical encoding of a node record is an RLP list of `[signature, seq, k, v, ...]`.

The maximum encoded size of a node record is 300 bytes. Implementations should reject

records larger than this size.

Records are signed and encoded as follows:

content = [seq, k, v, ...]

signature = sign(content)

record = [signature, seq, k, v, ...]

### Text Encoding

The textual form of a node record is the base64 encoding of its RLP representation,

prefixed by `enr:`. Implementations should use the [URL-safe base64 alphabet]

and omit padding characters.

### "v4" Identity Scheme

This specification defines a single identity scheme to be used as the default until other

schemes are defined by further EIPs. The "v4" scheme is backwards-compatible with the

cryptosystem used by Node Discovery v4.

- To sign record `content` with this scheme, apply the keccak256 hash function (as used by

the EVM) to `content`, then create a signature of the hash. The resulting 64-byte

signature is encoded as the concatenation of the `r` and `s` signature values (the

recovery ID `v` is omitted).

- To verify a record, check that the signature was made by the public key in the

"secp256k1" key/value pair of the record.

- To derive a node address, take the keccak256 hash of the uncompressed public key.

## Rationale

The format is meant to suit future needs in two ways:

- Adding new key/value pairs: This is always possible and doesn't require implementation

consensus. Existing clients will accept any key/value pairs regardless of whether they

can interpret their content.

- Adding identity schemes: these need implementation consensus because the network won't

accept the signature otherwise. To introduce a new identity scheme, propose an EIP and

get it implemented. The scheme can be used as soon as most clients accept it.

The size of a record is limited because records are relayed frequently and may be included

in size-constrained protocols such as DNS. A record containing a IPv4 address, when signed

using the "v4" scheme occupies roughly 120 bytes, leaving plenty of room for additional

metadata.

You might wonder about the need for so many pre-defined keys related to IP addresses and

ports. This need arises because residential and mobile network setups often put IPv4

behind NAT while IPv6 traffic—if supported—is directly routed to the same host. Declaring

both address types ensures a node is reachable from IPv4-only locations and those

supporting both protocols.

## Test Vectors

This is an example record containing the IPv4 address `127.0.0.1` and UDP port `30303`.

The node ID is `a448f24c6d18e575453db13171562b71999873db5b286df957af199ec94617f7`.

enr:-IS4QHCYrYZbAKWCBRlAy5zzaDZXJBGkcnh4MHcBFZntXNFrdvJjX04jRzjzCBOonrkTfj499SZuOh8R33Ls8RRcy5wBgmlkgnY0gmlwhH8AAAGJc2VjcDI1NmsxoQPKY0yuDUmstAHYpMa2\_oxVtw0RW\_QAdpzBQA8yWM0xOIN1ZHCCdl8

The record is signed using the "v4" identity scheme using sequence number `1` and this

private key:

b71c71a67e1177ad4e901695e1b4b9ee17ae16c6668d313eac2f96dbcda3f291

The RLP structure of the record is:

[

7098ad865b00a582051940cb9cf36836572411a47278783077011599ed5cd16b76f2635f4e234738f30813a89eb9137e3e3df5266e3a1f11df72ecf1145ccb9c,

01,

"id",

"v4",

"ip",

7f000001,

"secp256k1",

03ca634cae0d49acb401d8a4c6b6fe8c55b70d115bf400769cc1400f3258cd3138,

"udp",

765f,

]

[URL-safe base64 alphabet]: https://tools.ietf.org/html/rfc4648#section-5

Lastly….

Quick Review and Re-Cap of RLPx Protocol

# The RLPx Transport Protocol

This specification defines the RLPx transport protocol, a TCP-based transport protocol

used for communication among Ethereum nodes. The protocol carries encrypted messages

belonging to one or more 'capabilities' which are negotiated during connection

establishment. RLPx is named after the [RLP] serialization format. The name is not an

acronym and has no particular meaning.

The current protocol version is \*\*5\*\*. You can find a list of changes in past versions at

the end of this document.

## Notation

`X || Y`\

denotes concatenation of X and Y.\

`X ^ Y`\

is byte-wise XOR of X and Y.\

`X[:N]`\

denotes an N-byte prefix of X.\

`[X, Y, Z, ...]`\

denotes recursive encoding as an RLP list.\

`keccak256(MESSAGE)`\

is the Keccak256 hash function as used by Ethereum.\

`ecies.encrypt(PUBKEY, MESSAGE, AUTHDATA)`\

is the asymmetric authenticated encryption function as used by RLPx.\

AUTHDATA is authenticated data which is not part of the resulting ciphertext,\

but written to HMAC-256 before generating the message tag.\

`ecdh.agree(PRIVKEY, PUBKEY)`\

is elliptic curve Diffie-Hellman key agreement between PRIVKEY and PUBKEY.

## ECIES Encryption

ECIES (Elliptic Curve Integrated Encryption Scheme) is an asymmetric encryption method

used in the RLPx handshake. The cryptosystem used by RLPx is

- The elliptic curve secp256k1 with generator `G`.

- `KDF(k, len)`: the NIST SP 800-56 Concatenation Key Derivation Function

- `MAC(k, m)`: HMAC using the SHA-256 hash function.

- `AES(k, iv, m)`: the AES-128 encryption function in CTR mode.

Alice wants to send an encrypted message that can be decrypted by Bobs static private key

<code>k<sub>B</sub></code>. Alice knows about Bobs static public key

<code>K<sub>B</sub></code>.

To encrypt the message `m`, Alice generates a random number `r` and corresponding elliptic

curve public key `R = r \* G` and computes the shared secret <code>S = P<sub>x</sub></code>

where <code>(P<sub>x</sub>, P<sub>y</sub>) = r \* K<sub>B</sub></code>. She derives key

material for encryption and authentication as

<code>k<sub>E</sub> || k<sub>M</sub> = KDF(S, 32)</code> as well as a random

initialization vector `iv`. Alice sends the encrypted message `R || iv || c || d` where

<code>c = AES(k<sub>E</sub>, iv , m)</code> and

<code>d = MAC(sha256(k<sub>M</sub>), iv || c)</code> to Bob.

For Bob to decrypt the message `R || iv || c || d`, he derives the shared secret

<code>S = P<sub>x</sub></code> where

<code>(P<sub>x</sub>, P<sub>y</sub>) = k<sub>B</sub> \* R</code> as well as the encryption and

authentication keys <code>k<sub>E</sub> || k<sub>M</sub> = KDF(S, 32)</code>. Bob verifies

the authenticity of the message by checking whether

<code>d == MAC(sha256(k<sub>M</sub>), iv || c)</code> then obtains the plaintext as

<code>m = AES(k<sub>E</sub>, iv || c)</code>.

## Node Identity

All cryptographic operations are based on the secp256k1 elliptic curve. Each node is

expected to maintain a static secp256k1 private key which is saved and restored between

sessions. It is recommended that the private key can only be reset manually, for example,

by deleting a file or database entry.

## Initial Handshake

An RLPx connection is established by creating a TCP connection and agreeing on ephemeral

key material for further encrypted and authenticated communication. The process of

creating those session keys is the 'handshake' and is carried out between the 'initiator'

(the node which opened the TCP connection) and the 'recipient' (the node which accepted it).

1. initiator connects to recipient and sends its `auth` message

2. recipient accepts, decrypts and verifies `auth` (checks that recovery of signature ==

`keccak256(ephemeral-pubk)`)

3. recipient generates `auth-ack` message from `remote-ephemeral-pubk` and `nonce`

4. recipient derives secrets and sends the first encrypted frame containing the [Hello] message

5. initiator receives `auth-ack` and derives secrets

6. initiator sends its first encrypted frame containing initiator [Hello] message

7. recipient receives and authenticates first encrypted frame

8. initiator receives and authenticates first encrypted frame

9. cryptographic handshake is complete if MAC of first encrypted frame is valid on both sides

Either side may disconnect if authentication of the first framed packet fails.

Handshake messages:

auth = auth-size || enc-auth-body

auth-size = size of enc-auth-body, encoded as a big-endian 16-bit integer

auth-vsn = 4

auth-body = [sig, initiator-pubk, initiator-nonce, auth-vsn, ...]

enc-auth-body = ecies.encrypt(recipient-pubk, auth-body || auth-padding, auth-size)

auth-padding = arbitrary data

ack = ack-size || enc-ack-body

ack-size = size of enc-ack-body, encoded as a big-endian 16-bit integer

ack-vsn = 4

ack-body = [recipient-ephemeral-pubk, recipient-nonce, ack-vsn, ...]

enc-ack-body = ecies.encrypt(initiator-pubk, ack-body || ack-padding, ack-size)

ack-padding = arbitrary data

Implementations must ignore any mismatches in `auth-vsn` and `ack-vsn`. Implementations

must also ignore any additional list elements in `auth-body` and `ack-body`.

Secrets generated following the exchange of handshake messages:

static-shared-secret = ecdh.agree(privkey, remote-pubk)

ephemeral-key = ecdh.agree(ephemeral-privkey, remote-ephemeral-pubk)

shared-secret = keccak256(ephemeral-key || keccak256(nonce || initiator-nonce))

aes-secret = keccak256(ephemeral-key || shared-secret)

mac-secret = keccak256(ephemeral-key || aes-secret)

## Framing

All messages following the initial handshake are framed. A frame carries a single

encrypted message belonging to a capability.

The purpose of framing is multiplexing multiple capabilites over a single connection.

Secondarily, as framed messages yield reasonable demarcation points for message

authentication codes, supporting an encrypted and authenticated stream becomes

straight-forward. Frames are encrypted and authenticated via key material generated during

the handshake.

The frame header provides information about the size of the message and the message's

source capability. Padding is used to prevent buffer starvation, such that frame

components are byte-aligned to block size of cipher.

frame = header-ciphertext || header-mac || frame-ciphertext || frame-mac

header-ciphertext = aes(aes-secret, header)

header = frame-size || header-data || header-padding

header-data = [capability-id, context-id]

capability-id = integer, always zero

context-id = integer, always zero

header-padding = zero-fill header to 16-byte boundary

frame-ciphertext = aes(aes-secret, frame-data || frame-padding)

frame-padding = zero-fill frame-data to 16-byte boundary

See the [Capability Messaging] section for definitions of `frame-data` and `frame-size.`

### MAC

Message authentication in RLPx uses two keccak256 states, one for each direction of

communication. The `egress-mac` and `ingress-mac` keccak states are continuously updated

with the ciphertext of bytes sent (egress) or received (ingress). Following the initial

handshake, the MAC states are initialized as follows:

Initiator:

egress-mac = keccak256.init((mac-secret ^ recipient-nonce) || auth)

ingress-mac = keccak256.init((mac-secret ^ initiator-nonce) || ack)

Recipient:

egress-mac = keccak256.init((mac-secret ^ initiator-nonce) || ack)

ingress-mac = keccak256.init((mac-secret ^ recipient-nonce) || auth)

When a frame is sent, the corresponding MAC values are computed by updating the

`egress-mac` state with the data to be sent. The update is performed by XORing the header

with the encrypted output of its corresponding MAC. This is done to ensure uniform

operations are performed for both plaintext MAC and ciphertext. All MACs are sent

cleartext.

header-mac-seed = aes(mac-secret, keccak256.digest(egress-mac)[:16]) ^ header-ciphertext

egress-mac = keccak256.update(egress-mac, header-mac-seed)

header-mac = keccak256.digest(egress-mac)[:16]

Computing `frame-mac`:

egress-mac = keccak256.update(egress-mac, frame-ciphertext)

frame-mac-seed = aes(mac-secret, keccak256.digest(egress-mac)[:16]) ^ keccak256.digest(egress-mac)[:16]

egress-mac = keccak256.update(egress-mac, frame-mac-seed)

frame-mac = keccak256.digest(egress-mac)[:16]

Verifying the MAC on ingress frames is done by updating the `ingress-mac` state in the

same way as `egress-mac` and comparing to the values of `header-mac` and `frame-mac` in

the ingress frame. This should be done before decrypting `header-ciphertext` and

`frame-ciphertext`.

# Capability Messaging

All messages following the initial handshake are associated with a 'capability'. Any

number of capabilities can be used concurrently on a single RLPx connection.

A capability is identified by a short ASCII name and version number. The capabilities

supported on either side of the connection are exchanged in the [Hello] message belonging

to the 'p2p' capability which is required to be available on all connections.

## Message Encoding

The initial [Hello] message is encoded as follows:

frame-data = msg-id || msg-data

frame-size = length of frame-data, encoded as a 24bit big-endian integer

where `msg-id` is an RLP-encoded integer identifying the message and `msg-data` is an RLP

list containing the message data.

All messages following Hello are compressed using the Snappy algorithm.

frame-data = msg-id || snappyCompress(msg-data)

frame-size = length of frame-data encoded as a 24bit big-endian integer

Note that the `frame-size` of compressed messages refers to the compressed size of

`msg-data`. Since compressed messages may inflate to a very large size after

decompression, implementations should check for the uncompressed size of the data before

decoding the message. This is possible because the [snappy format] contains a length

header. Messages carrying uncompressed data larger than 16 MiB should be rejected by

closing the connection.

## Message ID-based Multiplexing

While the framing layer supports a `capability-id`, the current version of RLPx doesn't

use that field for multiplexing between different capabilities. Instead, multiplexing

relies purely on the message ID.

Each capability is given as much of the message-ID space as it needs. All such

capabilities must statically specify how many message IDs they require. On connection and

reception of the [Hello] message, both peers have equivalent information about what

capabilities they share (including versions) and are able to form consensus over the

composition of message ID space.

Message IDs are assumed to be compact from ID 0x11 onwards (0x00-0x10 is reserved for the

"p2p" capability) and given to each shared (equal-version, equal-name) capability in

alphabetic order. Capability names are case-sensitive. Capabilities which are not shared

are ignored. If multiple versions are shared of the same (equal name) capability, the

numerically highest wins, others are ignored.

## "p2p" Capability

The "p2p" capability is present on all connections. After the initial handshake, both

sides of the connection must send either [Hello] or a [Disconnect] message. Upon receiving

the [Hello] message a session is active and any other message may be sent. Implementations

must ignore any difference in protocol version for forward-compatibility reasons. When

communicating with a peer of lower version, implementations should try to mimic that

version.

At any time after protocol negotiation, a [Disconnect] message may be sent.

### Hello (0x00)

`[protocolVersion: P, clientId: B, capabilities, listenPort: P, nodeKey: B\_64, ...]`

First packet sent over the connection, and sent once by both sides. No other messages may

be sent until a Hello is received. Implementations must ignore any additional list elements

in Hello because they may be used by a future version.

- `protocolVersion` the version of the "p2p" capability, \*\*5\*\*.

- `clientId` Specifies the client software identity, as a human-readable string (e.g.

"Ethereum(++)/1.0.0").

- `capabilities` is the list of supported capabilities and their versions:

`[[cap1, capVersion1], [cap2, capVersion2], ...]`.

- `listenPort` specifies the port that the client is listening on (on the interface that

the present connection traverses). If 0 it indicates the client is not listening.

- `nodeId` is the secp256k1 public key corresponding to the node's private key.

### Disconnect (0x01)

`[reason: P]`

Inform the peer that a disconnection is imminent; if received, a peer should disconnect

immediately. When sending, well-behaved hosts give their peers a fighting chance (read:

wait 2 seconds) to disconnect to before disconnecting themselves.

`reason` is an optional integer specifying one of a number of reasons for disconnect:

| Reason | Meaning |

|--------|:-------------------------------------------------------------|

| `0x00` | Disconnect requested |

| `0x01` | TCP sub-system error |

| `0x02` | Breach of protocol, e.g. a malformed message, bad RLP, ... |

| `0x03` | Useless peer |

| `0x04` | Too many peers |

| `0x05` | Already connected |

| `0x06` | Incompatible P2P protocol version |

| `0x07` | Null node identity received - this is automatically invalid |

| `0x08` | Client quitting |

| `0x09` | Unexpected identity in handshake |

| `0x0a` | Identity is the same as this node (i.e. connected to itself) |

| `0x0b` | Ping timeout |

| `0x10` | Some other reason specific to a subprotocol |

### Ping (0x02)

`[]`

Requests an immediate reply of [Pong] from the peer.

### Pong (0x03)

`[]`

Reply to the peer's [Ping] packet.

# Change Log

### Known Issues in the current version

- The frame encryption/MAC scheme is considered 'broken' because `aes-secret` and

`mac-secret` are reused for both reading and writing. The two sides of a RLPx connection

generate two CTR streams from the same key, nonce and IV. If an attacker knows one

plaintext, they can decrypt unknown plaintexts of the reused keystream.

- General feedback from reviewers has been that the use of a keccak256 state as a MAC

accumulator and the use of AES in the MAC algorithm is an uncommon and overly complex

way to perform message authentication but can be considered safe.

- The frame encoding provides `capability-id` and `context-id` fields for multiplexing

purposes, but these fields are unused.

### Version 5 (EIP-706, September 2017)

[EIP-706] added Snappy message compression.

### Version 4 (EIP-8, December 2015)

[EIP-8] changed the encoding of `auth-body` and `ack-body` in the initial handshake to

RLP, added a version number to the handshake and mandated that implementations should

ignore additional list elements in handshake messages and [Hello].

# References

- Elaine Barker, Don Johnson, and Miles Smid. NIST Special Publication 800-56A Section 5.8.1,

Concatenation Key Derivation Function. 2017.\

URL <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-56ar.pdf>

- Victor Shoup. A proposal for an ISO standard for public key encryption, Version 2.1. 2001.\

URL <http://www.shoup.net/papers/iso-2\_1.pdf>

- Mike Belshe and Roberto Peon. SPDY Protocol - Draft 3. 2014.\

URL <http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3>

- Snappy compressed format description. 2011.\

URL <https://github.com/google/snappy/blob/master/format\_description.txt>

Copyright &copy; 2014 Alex Leverington.

<a rel="license" href="http://creativecommons.org/licenses/by-nc-sa/4.0/">

This work is licensed under a

Creative Commons Attribution-NonCommercial-ShareAlike

4.0 International License</a>.

[Hello]: #hello-0x00

[Disconnect]: #disconnect-0x01

[Ping]: #ping-0x02

[Pong]: #pong-0x03

[Capability Messaging]: #capability-messaging

[EIP-8]: https://eips.ethereum.org/EIPS/eip-8

[EIP-706]: https://eips.ethereum.org/EIPS/eip-706

[RLP]: https://github.com/ethereum/wiki/wiki/RLP

[snappy format]: https://github.com/google/snappy/blob/master/format\_description.txt