1 RFC-0001: RFC Life Cycle, Process and Structure

- RFC Number: 0001
- Title: RFC Life Cycle, Process and Structure
- Status: Raw
- Author(s): Qianchen Yu (@QYuQianchen), Tino Breddin (@tolbrino)
- Created: 2025-02-20
- Updated: 2025-08-20
- Version: v0.2.0 (Raw)
- Supersedes: none
- Related Links: none

1. Abstract

This RFC defines the life cycle, contribution process, versioning system, governance model, and document structure for RFCs at HOPR. It outlines stages, naming conventions, validation rules, and formatting standards that MUST be followed to ensure consistency and clarity across all RFC submissions. The process ensures iterative development with feedback loops and transparent updates with pull requests (PR).

2. Motivation

HOPR project requires a clear and consistent process for managing technical proposals, documenting protocol architecture. A well-defined life cycle MUST be established to maintain coherence, ensure quality, and streamline future development.

3. Terminology

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [01].

Draft: An RFC is considered a draft from the moment it is proposed for review. A draft MUST include a clear summary, context, and initial technical details. Drafts MUST follow the v0.x.x versioning scheme, with each version being independently implementable. A draft version is assigned as soon as the first PR is created.

4. Specification

4.1. RFC Life Cycle Stages

4.1.1. Mermaid Diagram for RFC Life Cycle Stages

4.1.2. Stage Descriptions:

- Raw: The RFC MUST begin as a raw draft reflecting initial ideas. The draft MAY contain incomplete details but MUST provide a clear objective.
- Discussion: Upon submission of the initial PR, the RFC number and `v0.1.0` version are assigned. Feedback SHALL be gathered via PRs, with iterative updates reflected in version increments `(v0.x.x)`.
- Review: The RFC MUST undergo at least one review cycle. The draft SHOULD incorporate significant feedback and each iteration MUST be independently implementable.
- Draft: The RFC moves into active development and refinement. Each update SHALL increment the version (`v0.x.x`) to indicate progress.
- Implementation: Merging to the main branch signifies readiness for practical use, triggering the finalization process.

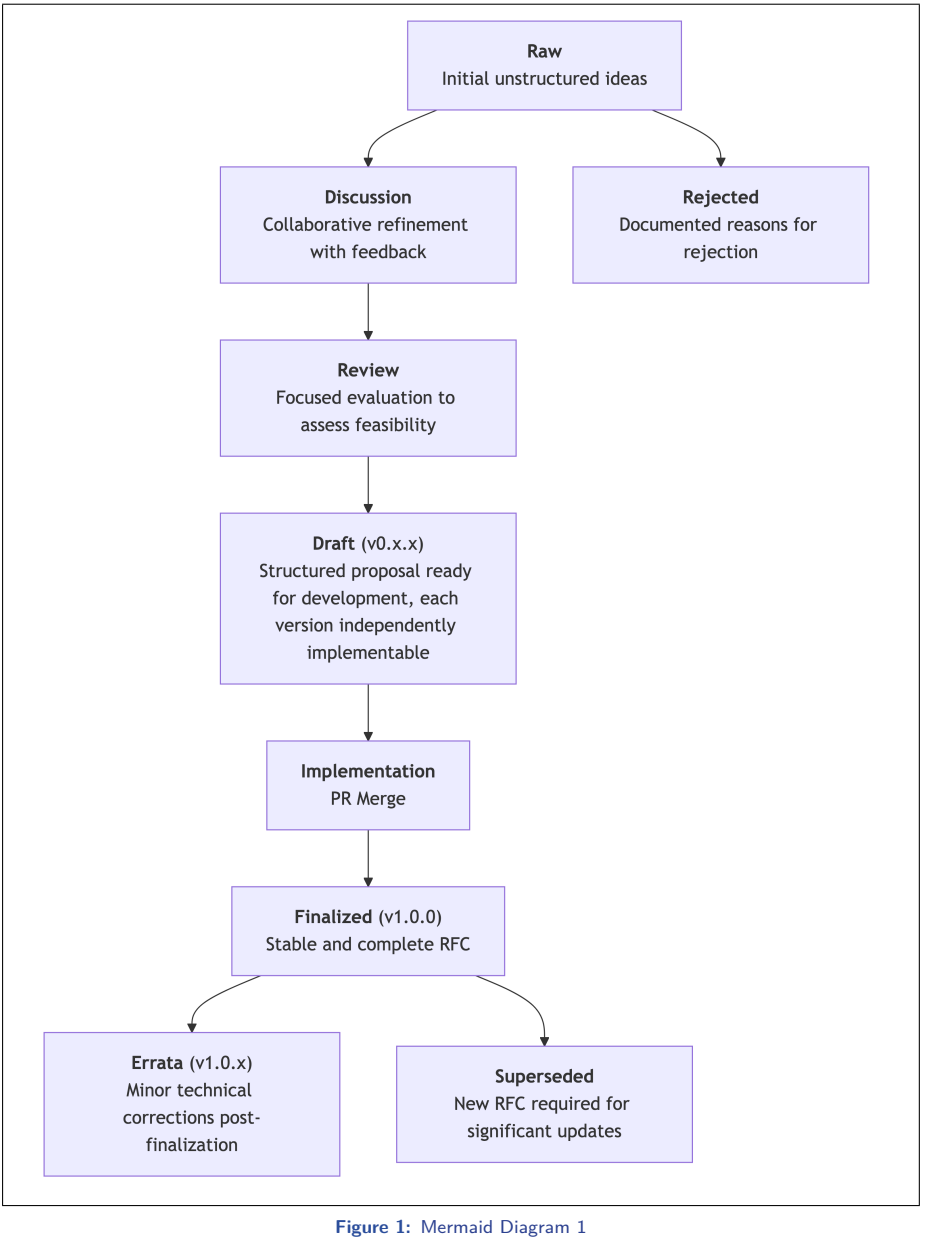


Figure 1: Mermaid Diagram 1

- Finalized: The RFC is considered stable and complete, with version `v1.0.0` assigned. Only errata modifications are permitted afterward.
- Errata: Minor technical corrections post-finalization MUST be documented and result in a patch version increment (`v1.0.x`). Errata are technical corrections or factual updates made after an RFC has been finalized. They MUST NOT alter the intended functionality or introduce new features.
- Superseded: Significant updates requiring functionality changes MUST be documented in a new RFC, starting at `v2.0.0` or higher. The original RFC must include information that it has been superseded, accompanied with a link to the new RFC that supersedes it.
- Rejected: If an RFC does not progress past the discussion stage, reasons MUST be documented.

4.2. File Structure

```
RFC-0001-rfc-life-cycle-process/
│
├── 0001-rfc-life-cycle-process.md
├── errata/
│   └── 0001-v1.0.1-erratum.md
└── assets/
    └── life-cycle-overview.png
```

4.3. Validation Rules

- Directory MUST be prefixed with uppercased “RFC”, followed by its RFC number, and a succinct title all in lowercase joined by hyphens. E.g. `RFC-0001-rfc-life-cycle-process`
- Main file MUST be prefixed with its RFC number and a succinct title all in lowercase joined by hyphens. E.g. `0001-rfc-life-cycle-process.md`

- All assets MUST reside in the `assets/` folder.
- Errata MUST reside in the `errata/` folder.

4.4. RFC Document Structure

All RFCs MUST follow a consistent document structure to ensure readability and maintainability.

4.4.1. Metadata Preface

Every RFC MUST begin with the following metadata structure:

```
\# RFC-XXXX: [Title]

- **RFC Number:** XXXX
- **Title:** [Title in Title Case]
- **Status:** Raw | Discussion | Review | Draft | Implementation | Finalized
↪  | Errata | Rejected | Superseded
- **Author(s):** [Name (GitHub Handle)]
- **Created:** YYYY-MM-DD
- **Updated:** YYYY-MM-DD
- **Version:** vX.X.X (Status)
- **Supersedes:** RFC-YYYY (if applicable) | N/A
- **Related Links:** [RFC-XXXX](../RFC-XXXX-[slug]/XXXX-[slug].md) | none
```

4.4.2. Reference Styles

RFCs MUST use two distinct reference styles:

4.4.2.1. RFC-to-RFC References

- RFC references to other HOPR RFCs MUST be listed in the metadata's Related Links: field
- Format: `[RFC-XXXX](../RFC-XXXX-[slug]/XXXX-[slug].md)`
- Multiple references SHALL be separated by commas
- If no RFC references exist, the field MUST contain "none"
- Example: `[RFC-0002](../RFC-0002-mixnet-keywords/0002-mixnet-keywords.md),[RFC-0004](../RFC-0004-hopr-packet-protocol/0004-hopr-packet-protocol.md)`

4.4.2.2. External References

- External references MUST be listed in a dedicated `##References` section at the end of the document
- References MUST use sequential numbering with zero-padding: [01], [02], etc.
- In-text citations MUST use the numbered format: "as described in [01]"
- Reference format SHOULD follow academic citation style:

```
[XX] Author(s). (Year). [Title](URL). \_Publication\_, Volume(Issue),
↪  pages.
```

- Example:

```
[01] Chaum, D. (1981). [Untraceable Electronic Mail, Return Addresses,
↪  and Digital Pseudonyms](https://www.freehaven.net/anonbib/cache/c
↪  haum-mix.pdf). \_Communications of the ACM, 24\_(2), 84-90.
```

4.4.3. Required Sections

All RFCs MUST include the following sections:

1. Metadata Preface (as defined in 4.4.1)
2. Abstract - Brief summary of the RFC's purpose and scope
3. References - External citations (if any)

5. Design Considerations

- Modular RFCs SHOULD be preferred.
- PR system MUST be the primary mechanism for contribution, review, and errata handling.

6. Compatibility

- New RFCs MUST maintain backward compatibility unless explicitly stated.
- Errata MUST NOT introduce backward-incompatible changes.
- Breaking changes MUST be reflected in a major version increment (`v2.0.0`).

7. Security Considerations

- Security review phase MUST be included before finalization.
- Errata MUST undergo security review if impacting critical components.

8. Drawbacks

- Strict naming conventions MAY limit creative flexibility.

9. Alternatives

- Collaborative document editing tools, e.g. hackmd.

10. Unresolved Questions

- Handling emergency RFCs
- Enforcing cross-RFC dependencies
- Formal approval timeline for errata

11. Future Work

- Automated validation tools
- CI/CD integration for automated versioning and errata checks
- Web interface for publishing RFCs

12. References

[01] Bradner, S. (1997). Key words for use in RFCs to Indicate Requirement Levels. IETF RFC 2119.

[02] RFC Editor Style Guide. RFC Editor.

[03] Rust RFC Process. Rust Language Team.

[04] ZeroMQ RFC Process. ZeroMQ Community.

[05] VACP2P RFC Index. Vac Research.

2 RFC-0002: Common mixnet terms and keywords

- RFC Number: 0002
- Title: Common mixnet terms and keywords
- Status: Draft
- Author(s): Tino Breddin (@tolbrino)
- Created: 2025-08-01
- Updated: 2025-09-04
- Version: v0.1.0 (Draft)
- Supersedes: none
- Related Links: none

1. Abstract

This RFC provides a glossary of common terms and keywords related to mixnets and the HOPR protocol specifically. It aims to establish a shared vocabulary for developers, researchers, and users involved in the HOPR project.

2. Motivation

The HOPR project involves a diverse community of people with different backgrounds and levels of technical expertise. A shared vocabulary is essential for clear communication and a common understanding of the concepts and technologies used in the project. This RFC aims to provide a single source of truth for the terminology used in the HOPR ecosystem.

3. Terminology

- Mixnet: Also known as a Mix network is a routing protocol that creates hard-to-trace communications by using a chain of proxy servers known as mixes which take in messages from multiple senders, shuffle them, and send them back out in random order to the next destination.
- Node: A process which implements the HOPR protocol and participates in the mixnet. Nodes can be run by anyone. A node can be a sender, destination or a relay node which helps to relay messages through the network. Also referred to as "peer" [01, 02].
- Sender: The node that initiates communication by sending out a packet through the mixnet. This is typically an application which wants to send a message anonymously [01, 02].
- Destination: The node that receives a message sent through the mixnet. Also referred to as "receiver" in some contexts [01, 02].
- Peer: A node that is connected to another node in the p2p network. Each peer has a unique identifier and can communicate with other peers. The terms "peer" and "node" are often used interchangeably.
- Cover Traffic: Artificial data packets introduced into the network to obscure traffic patterns with adaptive noise. These data packets can be generated on any node and are used to make it harder to distinguish between real user traffic and dummy traffic [01, 03].
- Path: The route a message takes through the mixnet, defined as a sequence of hops between sender and destination. A path can be direct from sender to destination, or it can go through multiple relay nodes before reaching the destination. Also referred to as "message path" [01, 02].
- Forward Path: A path that is used to deliver a packet only in the direction from the sender to the destination.
- Return Path: A path that is used to deliver a packet in the opposite direction than the forward path. The return path MAY be disjoint with the forward path.
- Relay Node: A node that forwards messages from one node to another in the mixnet. Relay nodes help to obscure the sender's identity by routing

messages through multiple nodes [01, 02].

- Hop: A relay node in the message path that is neither the sender nor the destination. E.g. a 0-hop message is sent directly from the sender to the destination, while a 1-hop message goes through one relay node before reaching the destination. The terms "hop" and "relay" are often used interchangeably [01, 02]. More hops in the path generally increase the anonymity of the message, but also increase latency and cost.
- Mix Nodes: These are the proxy servers that make up the mixnet. They receive messages from multiple senders, shuffle them, and then send them back out in a random order [01].
- Layered Encryption: A technique where a message is wrapped in successive layers of encryption. Each intermediary node (or hop) can only decrypt its corresponding layer, revealing the next destination in the path [01, 04].
- Metadata: Data that provides information about other data. In the context of mixnets, this includes things like the sender's and destination's IP addresses, the size of the message, and the time it was sent or received. Mixnets work to shuffle this metadata to protect user privacy [01, 06].
- Onion Routing: A technique for anonymous communication over a network. It involves encrypting messages in layers, analogous to the layers of an onion, which are then routed through a series of network nodes [04].
- Public Key Cryptography: A cryptographic system that uses pairs of keys: public keys, which may be disseminated widely, and private keys, which are known only to the owner. This is used to encrypt messages sent through the mixnet [01].
- Sphinx: A packet format that ensures unlinkability and layered encryption. It uses a fixed-size packet structure to resist traffic analysis [02].
- Symmetric Encryption: A type of encryption where the same key is used to both encrypt and decrypt data [05].
- Traffic Analysis: The process of intercepting and examining messages in order to deduce information from patterns in communication. Mixnets are designed to make traffic analysis very difficult [01].
- Forward Message: A packet that is sent along the forward path. Also referred to as "forward packet".

- Reply Message: A packet that is sent along the return path. Also referred to as "reply packet".
- HOPR Network: The decentralized network of HOPR nodes that relay messages to provide privacy-preserving communications with economic incentives.
- HOPR Node: A participant in the HOPR network that implements the full HOPR protocol stack and can send, receive, and relay messages while participating in the payment system.
- Session: An established communication channel between two HOPR nodes for exchanging multiple messages with state management and reliability features.
- Proof of Relay: A cryptographic proof that demonstrates a HOPR node has correctly relayed a message and is eligible to receive payment for the relay service.
- Channel: A payment channel between two HOPR nodes that enables efficient micropayments for relay services without requiring blockchain transactions for each payment.
- Mixer: A HOPR protocol component that introduces random delays and batching to packets to break timing correlation attacks and enhance traffic analysis resistance.

4. References

[01] Chaum, D. (1981). Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. Communications of the ACM, 24(2), 84-90.

[02] Danezis, G., & Goldberg, I. (2009). Sphinx: A Compact and Provably Secure Mix Format. 2009 30th IEEE Symposium on Security and Privacy, 262-277.

[03] K. Sampigethaya and R. Poovendran, A Survey on Mix Networks and Their Secure Applications. Proceedings of the IEEE, vol. 94, no. 12, pp. 2142-2181, Dec. 2006.

[04] Reed, M. G., Syverson, P. F., & Goldschlag, D. M. (1998). Anonymous Con-

nections and Onion Routing. IEEE Journal on Selected Areas in Communications, 16(4), 482-494.

[05] Shannon, C. E. (1949). Communication Theory of Secrecy Systems. Bell System Technical Journal, 28(4), 656-715. DOI: 10.1002/j.1538-7305.1949.tb00928.x

[06] Cheu, A., Smith, A., Ullman, J., Zeber, D., & Zhilyaev, M. (2019, April). Distributed differential privacy via shuffling. In Annual international conference on the theory and applications of cryptographic techniques (pp. 375-403). Cham: Springer International Publishing.

3 RFC-0003: HOPR Overview

- RFC Number: 0003
- Title: HOPR Overview
- Status: Draft
- Author(s): Tino Breddin (@tolbrino)
- Created: 2025-09-11
- Updated: 2025-09-11
- Version: v0.1.0 (Draft)
- Supersedes: none
- Related Links: RFC-0002, RFC-0004, RFC-0005, RFC-0007, RFC-0008, RFC-0009

1. Abstract

This RFC provides an introductory overview of the HOPR network (sometimes referred to as HOPRnet) and protocol stack. HOPR is a decentralized, incentivized mixnet that enables privacy-preserving communication by routing messages through multiple relay nodes.

HOPR's innovation includes the proof-of-relay mechanism, which solves the challenge of creating economically sustainable anonymous communication networks. HOPR enables scalable privacy infrastructure that grows stronger with increased adoption, unlike volunteer-based networks that struggle with sustainability and performance.

This document serves as the primary entry point for understanding the HOPR ecosystem, providing detailed architectural explanations while referencing specialized RFCs for implementation-specific details. It targets researchers, developers, and infrastructure providers seeking to understand or implement privacy-preserving communication solutions.

2. Motivation

In today's digital landscape, privacy-preserving communication is increasingly important for protecting user data, enabling free speech, and maintaining confidentiality in business and personal communications. Traditional internet protocols provide insufficient privacy protection, as metadata and traffic patterns can be analyzed to reveal sensitive information about users and their communications.

HOPR addresses these privacy challenges by implementing a decentralized mixnet that:

- Provides metadata privacy: Unlike traditional networks that expose communication patterns, HOPR obscures sender-receiver relationships through traffic mixing and onion routing [01, 02]
- Offers economic incentives: Node operators receive payment for relaying traffic, creating a sustainable ecosystem for privacy infrastructure
- Ensures decentralization: No single entity controls the network, preventing censorship and single points of failure
- Maintains accessibility: Applications can integrate privacy features without requiring users to understand complex cryptographic concepts

The HOPR protocol is designed to be transport-agnostic, allowing it to operate over standard internet infrastructures while providing strong privacy guarantees. By combining proven cryptographic techniques with novel incentive mechanisms, HOPR creates a practical solution for privacy-preserving communications at scale.

3. Terminology

For all terminology used in this document, including both general mixnet concepts and HOPR-specific terms, refer to RFC-0002.

4. Network Overview

The HOPR network is a decentralized, peer-to-peer network that provides privacy-preserving communication. The network architecture consists of several key components working together to ensure metadata privacy and incentivize participation.

4.1 Network Architecture

The HOPR network is composed of:

- Entry Nodes: Nodes that initiate communication sessions and send messages into the network
- Relay Nodes: Intermediate nodes that forward messages along routing paths and receive payment for their services
- Exit Nodes: Final relay nodes that deliver messages to their intended destinations
- Payment Infrastructure: On-chain payment channels that enable micro-transactions between nodes

4.2 Path Construction

Messages in the HOPR network are routed through multi-hop paths to provide privacy protection:

1. Path Discovery: Nodes discover available relay nodes through automated mechanisms detailed in RFC-0010
2. Path Selection: Senders choose routing paths based on privacy requirements, latency, and cost considerations
3. Onion Routing: Messages are encrypted in multiple layers, with each relay node able to decrypt only the information necessary to forward the message to the next hop [02]

4.3 Economic Incentives

The HOPR network uses economic incentives to ensure sustainable operation:

- Micropayments: Relay nodes receive small payments for each message they forward
- Proof of Relay: Cryptographic proofs ensure that relay nodes actually forward messages before receiving payment
- Payment Channels: Direct payment channels between nodes enable efficient microtransactions without high blockchain fees [04]

4.4 Privacy Properties

The network architecture provides several privacy guarantees:

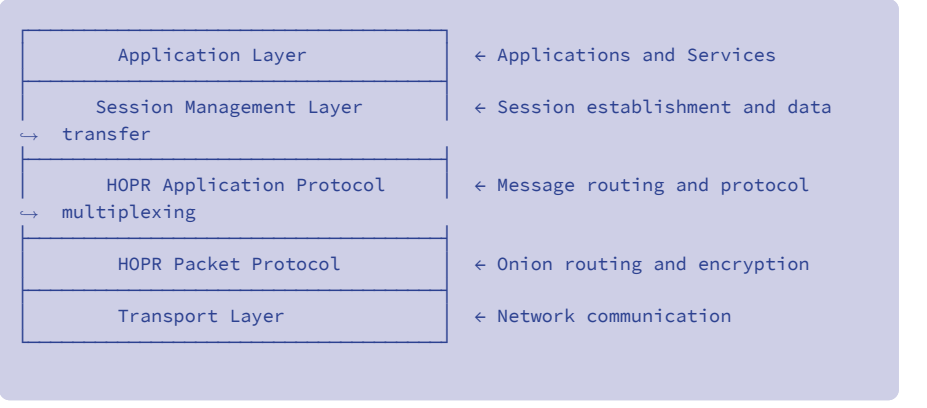
- Sender Anonymity: Relay nodes cannot determine the original sender of a message [05]
- Receiver Anonymity: Intermediate nodes cannot identify the final recipient [05]
- Unlinkability: Observers cannot link multiple messages from the same sender or to the same receiver [05]
- Traffic Analysis Resistance: Random delays and packet mixing prevent timing correlation attacks [06]

5. Protocol Overview

The HOPR protocol stack consists of multiple layers that work together to provide privacy-preserving communication with economic incentives. This section provides a high-level overview of the protocol components and their interactions.

5.1 Protocol Architecture

The HOPR protocol is organized into several layers:



5.2 Core Protocol Components

5.2.1 HOPR Packet Protocol

The HOPR Packet Protocol (RFC-0004) defines the fundamental packet format and processing rules:

- Onion Encryption: Multi-layer encryption that allows each relay node to decrypt only the information needed to forward the packet
- Sphinx-based Design: Based on the Sphinx packet format with extensions for incentivization [03]
- Fixed Packet Size: All packets have the same size to prevent traffic analysis based on packet size [06]

5.2.2 Proof of Relay

The Proof of Relay mechanism (RFC-0005) ensures that relay nodes actually forward packets:

- Cryptographic Proofs: Mathematical proofs that a node has correctly processed and forwarded a packet
- Payment Integration: Proofs are required before relay nodes receive payment for their services
- Fraud Prevention: Detects and prevents nodes from claiming payment without providing relay services

5.2.3 Traffic Mixing

The HOPR Mixer (RFC-0006) provides traffic analysis resistance:

- Temporal Mixing: Introduces random delays to break timing correlations between packets [01, 06]
- Batching: Groups packets together before forwarding to obscure traffic patterns
- Configurable Strategies: Multiple mixing strategies for different privacy/latency trade-offs

5.2.4 Session Management

Session protocols provide higher-level communication primitives:

- Session Establishment: RFC-0009 defines how nodes establish communication sessions
- Data Transfer: RFC-0008 provides reliable and unreliable data transmission modes
- Connection Management: Session lifecycle management and error handling

5.2.5 Economic System

The economic reward system (RFC-0007) incentivizes participation:

- Token Economics: Native token rewards for staked funds
- Payment Channels: Efficient micropayment infrastructure [04]
- Fair Distribution: Ensures equitable reward distribution based amount of staked funds

5.3 Protocol Flow

A typical message transmission through the HOPR network follows this flow:

1. Path Discovery: Sender discovers available relay nodes and constructs a routing path
2. Session Establishment: If required, sender establishes a session with the recipient
3. Packet Construction: Message is encrypted in multiple layers and packaged into HOPR packets
4. Routing: Packets are forwarded through the selected path, with each relay node:
    - Decrypting one layer to reveal the next hop
    - Applying traffic mixing delays
    - Generating proofs of relay
    - Receiving micropayments for the service
5. Delivery: Final relay node delivers the packet to the intended recipient

5.4 Integration Points

The HOPR protocol is designed to support various applications and use cases:

- Application Protocol: RFC-0011 defines how higher-level protocols can utilize HOPR services

- Transport Independence: Protocol can operate over different network transports (TCP, UDP, etc.)
- API Compatibility: Through Sessions familiar networking APIs are provided to ease application integration
- Extensibility: Modular design allows for protocol extensions and improvements

6. References

[01] Chaum, D. (1981). Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. Communications of the ACM, 24(2), 84-90.

[02] Reed, M. G., Syverson, P. F., & Goldschlag, D. M. (1998). Anonymous Connections and Onion Routing. IEEE Journal on Selected Areas in Communications, 16(4), 482-494.

[03] Danezis, G., & Goldberg, I. (2009). Sphinx: A Compact and Provably Secure Mix Format. 2009 30th IEEE Symposium on Security and Privacy, 262-277.

[04] Poon, J., & Dryja, T. (2016). The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. Lightning Network Whitepaper.

[05] Pfitzmann, A., & Köhntopp, M. (2001). Anonymity, Unobservability, and Pseudonymity—A Proposal for Terminology. In Designing Privacy Enhancing Technologies (pp. 1-9). Springer.

[06] Raymond, J. F. (2001). Traffic Analysis: Protocols, Attacks, Design Issues, and Open Problems. In Designing Privacy Enhancing Technologies (pp. 10-29). Springer.

4 RFC-0004 HOPR Packet Protocol

- RFC Number: 0004
- Title: HOPR Packet Protocol
- Status: Draft
- Author(s): Lukas Pohanka (@NumberFour8)
- Created: 2025-03-19
- Updated: 2025-08-27
- Version: v0.9.0 (Draft)
- Supersedes: none
- Related Links: RFC-0002, RFC-0005, RFC-0006, RFC-0011

1. Abstract

This RFC describes the wire format of a HOPR packet and its encoding and decoding protocol. The HOPR packet format is heavily based on the Sphinx packet format [01], as it aims to fulfil the similiar set of goals: to provide anonymous indistinguishable packets, hiding the path length and unlinkability of messages. Moreover, the HOPR packet format adds additional information to the header, which allows incentivization of individual relay nodes via Proof of Relay.

The Proof of Relay (PoR) is described in the separate RFC-0005.

2. Introduction

The HOPR packet format is the fundamental building block of the HOPR protocol, allowing to build the HOPR mixnet. The format is designed to create indistinguishable packets sent between source and destination using a set of relays over a path RFC-0002, thereby achieving unlinkability of messages between sender and destination. In HOPR protocol, the relays SHOULD also perform packet mixing, as described in RFC-0006. The format is built using the Sphinx

packet format [01] but adds additional information for each hop to allow incentivization of the hops (except the last one) for the relaying duties. The incentivization of the last hop is exempt from the HOPR packet format itself and is subject to a separate RFC-0007.

The HOPR packet format does not require a reliable underlying transport or in-order delivery. The packet payloads are encrypted, however, payload authenticity and integrity is not assured and MAY be ensured by the overlay protocol. In addition, the packet format is aimed to minimize overhead and maximize payload capacity.

The HOPR packet consists of two primary parts:

- Meta packet (also called the Sphinx packet) that carries the necessary routing information for the selected path and the encrypted payload. This will be described in the following sections.
- Ticket, which contains payout (incentivization) information for the next hop on the path. The structure of Tickets is described in the separate RFC-0005.

This document describes version 1.0.0 of the HOPR Packet format and protocol.

2.1. Conventions and terminology

The keywords “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [02] when, and only when, they appear in all capitals, as shown here.

Terms defined in RFC-0002 are used, as well as some following additional terms:

peer public/private key (also pubkey or privkey): part of a cryptographic keypair owned by a peer.

extended path: a forward or return path which in addition contains the destination or sender respectively.

pseudonym: a randomly generated identifier of the sender. The pseudonym MAY be prefixed with a static prefix. The length such static prefix MUST NOT exceed half of the entire pseudonym's size. The pseudonym used in the forward message MUST be the same as the pseudonym used in the reply message.

public key identifier: a reasonably short identifier of each peer's public key. The size of such an identifier SHOULD be strictly smaller than the size of the corresponding public key.

|x|: denotes the binary representation length of `x` in bytes.

2.2. Global packet format parameters

The HOPR packet format requires certain cryptographic primitives in place, namely:

- an Elliptic Curve (EC) group where the Elliptic Curve Diffie-Hellman Problem (ECDLP) is hard. The peer public keys correspond to points on the chosen EC. The peer private keys correspond to scalars of the corresponding finite field.
- Pseudo-Random Permutation (PRP), commonly represented by a symmetric cipher
- Pseudo-Random Generator (PRG), commonly represented by a stream cipher or a block cipher in a stream-mode.
- One-time authenticator `OA(K,M)` where K denotes a one-time key and M is the message being authenticated
- a Key Derivation Function (KDF) allowing to:
  - generate secret key material from a high-entropy pre-key K, context string C, and a salt S: `KDF(C,K,S)`. KDF will perform the necessary expansion to match the size required by the output. The Salt `S` argument is optional and MAY be omitted.
  - if the above is applied to an EC point as `K`, the point MUST be in its compressed form.

- Hash to Field (Scalar) operation `HS(S,T)` which computes a field element of the elliptic curve from RFC-0005, given the secret `S` and a tag `T`.

The concrete instantiations of these primitives are discussed in Appendix 1. All the primitives MUST have corresponding security bounds (e.g., they all have 128-bit security) and the generated key material MUST also satisfy the required bounds of the primitives.

The global value of `PacketMax` is the maximum size of the data in bytes allowed inside the packet payload.

3. Forward packet creation

The REQUIRED inputs for the packet creation are as follows:

- User's Packet payload (as a sequence of bytes)
- Sender pseudonym (as a sequence of bytes)
- forward path and an OPTIONAL list of one or more return paths

The input MAY also contain:

- unique bidirectional map between peer pubkeys and public key identifiers (mapper)

Note that the mapper MAY only contain public key identifiers mappings of pubkeys from forward and return paths.

The packet payload MUST be between 0 to `PacketMax` bytes-long.

The Sender pseudonym MUST be randomly generated for each packet header but MAY contain a static prefix.

The forward and return paths MAY be represented by public keys of individual hops. Alternatively, the paths MAY be represented by public key identifiers and mapped using the mapper as needed.

The size of the forward and return paths (number of hops) MUST be between 0 and 3.

3.1. Partial Ticket creation

The creation of the HOPR packet starts with the creation of the partial Ticket structure as defined in RFC-0005. If Ticket creation fails at this point, the packet creation process MUST be terminated.

The Ticket is created almost completely, apart from the Challenge field, which can be populated only after the Proof of Relay values have been fully created for the packet.

3.2. Generating the Shared secrets

In the next step, shared secrets for individual hops on the forward path are generated, as described in Section 2.2 in [01]:

Assume the length of the path is `N` (between 0 and 3) and each hop's public key is `Phop_i`. The public key of the destination is `Pdst`.

Let the extended path be a list of `Phop_i` and `Pdst` (for `i=1..N`). For `N=0`, the extended path consists of just `Pdst`.

1. A new random ephemeral key pair is generated, `Epriv` and `Epub` respectively.
2. Set `Alpha` = `Epub` and `Coeff` = `Epriv`
3. For each (i-th) public key `P_i` the Extended path:
    - `SharedPreSecret_i` = `Coeff` * `P_i`
    - `SharedSecret_i` = KDF("HASH_KEY_SPHINX_SECRET", `SharedPreSecret_i`, `P_i`)
    - if `i==N`, quit the loop
    - `B_i=KDF("HASH_KEY_SPHINX_BLINDING",SharedPreSecret_i,Alpha)`
    - `Alpha=B_iAlpha`
    - `Coeff=B_iCoeff`
4. Return `Alpha` and the list of `SharedSecret_i`

For path of length `N`, the list length of the Shared secrets is `N+1`.

In some instantiations, an invalid elliptic curve point may be encountered anywhere during step 3. In such case the computation MUST fail with an error. The process then MAY restart from step 1.

After KDF_expand``,theB_i' MAY be additionally transformed so that it conforms to a valid field scalar. Shall that operation fail, the computation MUST fail with an error and the process then MAY restart from step 1.

The returned Alpha value MAY be encoded to an equivalent representation (such as using elliptic curve point compression), so that space is preserved.

3.3. Generating the Proof of Relay

The packet generation continues with per-hop proof generation of relay values, Ticket challenge, and Acknowledgement challenge for the first downstream node. This generation is done for each hop on the path.

This is described in RFC-0005 and is a two-step process.

The first step uses the List of shared secrets for the extended path as input. As a result, there is a list of length N, where each entry contains:

- Ticket challenge for the hop i+1 on the extended path
- Hint value for the i-th hop

Both values in each list entry are elliptic curve points. The Ticket challenge value MAY be transformed via a one-way cryptographic hash function, whose output MAY be truncated. See RFC-0005 on how such representation is instantiated.

This list consists of PoRStrings_i entries.

In the second step of the PoR generation, the input is the first Shared secret from the List and optionally the second Shared secret (if the extended path is longer than 1). It outputs additional two entries:

- Acknowledgement challenge for the first hop

- Ticket challenge for the first ticket

Also, here, both values are EC points, where the latter MAY be represented via the same one-way representation.

This tuple is called `PoRValues` and is used to finalize the partial Ticket: the Ticket challenge fills in the missing part in the `Ticket`.

3.4. Forward Meta Packet creation

At this point, there is enough information to generate the Meta packet, which is a logical construct that does not contain the `Ticket` yet.

The Meta Packet consists of the following components:

- `Alpha` value
- `Header` (an instantiation of the Sphinx mix header)
- padded and encrypted payload `EncPayload`

The above order of these components is canonical MUST be followed when a packet is serialized to its binary form. The definitions of the above components follow in the next sections.

The `Alpha` value is obtained from the Shared secrets generation phase.

The `Header` is created differently depending on whether this packet is a forward packet or a reply packet.

The creation of the `EncPayload` depends on whether the packet is routed via the forward path or return path.

3.4.1. Header creation

The header creation also closely follows [01] Section 3.2. Its creation is almost identical whether it is being created for the forward or return path.

The input for the header creation is:

- Extended path (of peer public keys `P_i`)
- Shared secrets from previous steps (`SharedSecret_i`)
- PoRStrings (each entry denoted a `PoRString_i` of equal lengths)
- Sender pseudonym (represented as a sequence of bytes)

Let `HeaderPrefix_i` be a single byte, where:

- The first 3 most significant bits indicate the version, and currently MUST be set to `001`.
- The 4th most significant bit indicates the `NoAckFlag`. It MUST be set to 1 when the recipient SHOULD NOT acknowledge the packet.
- The 5th most significant bit indicates the `ReplyFlag` and MUST be set to 1 if the header is created for the return path, otherwise it MUST be zero.
- The last remaining 3 bits represent the number `i`, in most significant bits first format.

For example, the binary representation of `HeaderPrefix_3` with `ReplyFlag` set and `NoAckFlag` not set looks like this:

```
HeaderPrefix\_3 = 0 0 1 0 1 0 1 1
```

The `HeaderPrefix_i` MUST not be computed for `i>7`.

Let `ID_i` be a public key identifier of `P_i` (by using the mapper), and `|T|` denote the output's size of a chosen one-time authenticator. Since `ID_i` MUST be all of equal lengths for each `i`, denote this length `|ID|`. Similarly, `|PoRString_i|` MUST have also all equal lengths of `|PoRString|`.

Let `RoutingInfoLen` be equal to `1+|ID|+|T|+|PoRString|`.

Allocate a zeroized `HdrExt` buffer of `1+|Pseudonym|+4*RoutingInfoLen` bytes and another zeroed buffer `OATag` of `|T|` bytes.

For each i = 1 up to N+1 do:

1. Initialize PRG with `SharedSecret_{N-i+2}`
2. If i is equal to 1
    - Set `HdrExt[0]` to `HeaderPrefix_0`

- Copy all bytes of `Pseudonym` to `HdrExt` at offset 1
- Fill `HdrExt` from offset `1+|Pseudonym|` up to `(5-N)*RoutingInfoLen` with uniformly randomly generated bytes.
- Perform an exclusive-OR (XOR) of bytes generated by the PRG with HdrExt, starting from offset 0 up to `1+|Pseudonym|+(5-N)*RoutingInfoLen`
- If N > 0, generate filler bytes given the list of Shared secrets as follows:
  - Allocate a zeroed buffer Filler of `(N-1)*RoutingInfoLen`
  - For each j from 1 to N-1:
    * Initialize a new PRG instance with `SharedSecret_j`
    * Seek the PRG to position `1+|Pseudonym|+(4-j)*RoutingInfoLen`
    * XOR RoutingInfoLen bytes of the PRG to Filler from offset 0 up to `j*RoutingInfoLen`
    * Destroy the PRG instance
- Copy the Filler bytes to `HdrExt` at offset `1+|Pseudonym|+(5-N)*RoutingInfoLen`

3. If i is greater than 1:
   - Copy bytes of `HdrExt` from offset 0 up to `1+|Pseudonym|+3*RoutingInfoLen` to offset `RoutingInfoLen` in `HdrExt`
   - Set `HdrExt[0]` to `HeaderPrefix_{i-1}`
   - Copy `ID_{N-i+2}` to `HdrExt` starting at offset 1
   - Copy `OATag` to `HdrExt` starting at offset `1+|ID|`
   - Copy bytes of `PoRString_{N-i+2}` to `HdrExt` starting at offset `1+|ID|+|T|`
   - XOR PRG bytes to `HdrExt` from offset 0 up to `1+|Pseudonym|+3*RoutingInfoLen`
4. Compute `K_tag` = KDF("HASH*KEY_TAG", `SharedSecret*{N-i+2}`)
5. Compute `OA(K_tag,HdrExt[0..1+|Pseudonym|+3*RoutingInfoLen)` and copy its output of `|T|` bytes to `OATag`

The output is the contents of `HdrExt` from offset 0 up to `1+|Pseudonym|+3*RoutingInfoLen` and the `OATag`:

```
Header \{
  header: [u8; 1 + |Pseudonym| + 3 * RoutingInfoLen]
  oa\_tag: [u8; |T|]
\}
```

3.4.2. Forward payload creation

The packet payload consists of the User payload given at the beginning of section 2. However, if any non-zero number of return paths has been given as well, the packet payload MUST consist of that many Single Use Reply Blocks (SURBs) that are prepended to the User payload.

The total size of the packet payload MUST not exceed `PacketMax` bytes, and therefore the size of the User payload and the number of SURBs is bounded.

A packet MAY only contain SURBs and no User payload. There MUST NOT be more than 15 SURBs in a single packet. The packet MAY contain additional packet signals for the recipient, typically the upper 4 bits of the SURB count field MAY serve this purpose.

For the above reasons, the forward payload MUST consist of:

- the number of SURBs
- all SURBs (if the number was non-zero)
- User's payload

```
PacketPayload \{
  signals: u4,
  num\_surbs: u4,
  surbs: [Surb; num\_surbs]
  user\_payload: [u8; <variable length>]
\}
```

The `signals` and `num_surbs` fields MAY be encoded as a single byte, where the most-significant 4 bits represent the `signals` and the least-significant 4

bits represent the num_surbs. When no signals are passed, the signals field MUST be zero.

The user payload usually consists of the Application layer protocol as described in RFC-0011, but it can be arbitrary.

3.4.3. Generating SURBs

The Single Use Reply Block is always generated by the Sender for its chosen pseudonym. Its purpose is to allow reply packet generation sent on the return path from the recipient back to sender.

The process of generating a single SURB is very similar to the process of creating the forward packet header.

As SURB is sent to the packet recipient, it also has its counterpart, called ReplyOpener. The ReplyOpener is generated alongside the SURB and is stored at the Sender (indexed by its Pseudonym) and used later to decrypt the reply packet delivered to the Sender using the associated SURB.

Both SURB and the ReplyOpener are always bound to the chosen Sender pseudonym.

Inputs for creating a SURB and the ReplyOpener:

- return path
- sender pseudonym

OPTIONALLY, also a unique bidirectional map between peer pubkeys and public key identifiers (mapper) is given.

The generation of SURB and its corresponding ReplyOpener is as follows:

Assume the length of the return path is N (between 0 and 3) and each hop's public key is Phop_i. The public key of the sender is Psrc.

Let the extended return path be a list of Phop_i and Psrc (for i = 1 .. N). For N = 0, the Extended return path consists of just Psrc.

1. generate a Shared secret list (`SharedSecret_i`) for the extended return path and the corresponding `Alpha` value as given in section 3.2.
2. generate PoR for the given extended return path: list of `PoRStrings_i` and `PoRValues`
3. generate Reply packet `Header` for the extended return path as in section 3.4.1:
   - The list of `PoRStrings_i` and list of `SharedSecret_i` from step 1 and 2 are used
   - The 5th bit of the `HeaderPrefix` is set to 1 (see section 3.4.1)
4. generate a random cryptographic key material, for at least the selected security boundary (`SenderKey` as a sequence of bytes)

`SURB` MUST consist of:

- `SenderKey`
- `Header` (for the return path)
- public key identifier of the first return path hop
- `PoRValues`
- `Alpha` value (for the return path)

```
SURB \{
  alpha: Alpha,
  header: Header,
  sender\_key: [u8; <variable length>]
  first\_hop\_ident: [u8; <variable length>]
  por\_values: PoRValues
\}
```

The corresponding `ReplyOpener` MUST consist of:

- `SenderKey`
- Shared secret list (`SharedSecret_i`)

```
ReplyOpener \{
  sender\_key: [u8; <variable length>]
  rp\_shared\_secrets: [SharedSecret; N+1]
\}
```

The Sender keeps the `ReplyOpener` (MUST be indexed by the chosen pseudonym), and puts the `SURB` in the forward packet payload.

3.4.4. Payload padding

The packet payload MUST be padded in accordance to [01] to exactly `PacketMax+|PaddingTag|` bytes.

The process works as follows:

The payload MUST always be pre-pended with a `PaddingTag`. The `PaddingTag` SHOULD be 1 byte long.

If the length of the payload is still less than `PacketMax+|PaddingTag|` bytes, zero bytes MUST be prepended until the length is exactly `PacketMax+|PaddingTag|` bytes.

```
PaddedPayload \{
  zeros: [0u8; PacketMax - |PacketPayload|],
  padding\_tag: u8,
  payload: PacketPayload
\}
```

3.4.5. Payload encryption

The encryption of the padded payload follows the same procedure from [01].

For each i=1 up to N:

1. Generate `Kprp` = KDF("HASH_KEY_PRP", `SharedSecret_i`)
2. Transform the `PaddedPayload` using PRP:

```
EncPayload = PRP(Kprp, PaddedPayload)
```

The Meta packet is formed from `Alpha`, `Header`, and `EncPayload`.

3.5. Final forward packet overview

The final structure of the HOPR packet format MUST consist of the logical Meta packet with the `Ticket` attached:

```
HOPR\_Packet \{
  alpha: Alpha,
  header: Header,
  encrypted\_payload: EncPayload,
  ticket: Ticket
\}
```

The packet is then sent to the peer represented by the first public key of the forward path.

Note that the size of the packet is exactly `|HOPR_Packet|=|Alpha|+|Header|+|PacketMax|+|PaddingTag|+|Ticket|`. It can be also referred to the size of the logical Meta packet plus `|Ticket|`.

4. Reply packet creation

Upon receiving a forward packet, the forward packet recipient SHOULD create a reply packet using one of a SURB. This is possible only if the Recipient received a SURB (with this or any previous forward packets) from an equal pseudonym.

The Recipient MAY use any SURB with the same pseudonym; however, in such a case the SURBs MUST be used in the reverse order in which they were received.

The Sender of the forward packet MAY use a fixed random prefix of the pseudonym to identify itself across multiple forward packets. In such a case, the SURBs in-

dexed with pseudonyms with the same prefix SHOULD be used in random order to construct reply packets.

The following inputs are REQUIRED to create the reply packet:

- User's Packet payload (as a sequence of bytes)
- Pseudonym of the forward packet sender
- Single Use Reply Block (`SURB`) corresponding to the above pseudonym

OPTIONALLY, a unique bidirectional map between peer pubkeys and public key identifiers (mapper) is also given.

The final reply packet is a `HOPR_Packet` and the means of getting the values needed for its construction are given in the next sections.

4.1. Reply packet ticket creation

The `PoRValues` and first reply hop key identifiers are extracted from the used `SURB`.

The mapper is used to map the key identifier (`first_hop_ident`) to the public key of the first reply hop, which is then used to retrieve required ticket information.

The Challenge from the `PoRValues` (`por_values`) in the `SURB` is used to construct the complete `Ticket` for the first hop.

4.2. Reply meta packet creation

The `Alpha` value (`alpha` field) and the packet `Header` (`header` field) are extracted from the used `SURB`.

4.2.1. Reply payload creation

The reply payload is constructed as `PacketPayload` in section 3.4.2. However, the reply payload MUST not contain any SURBs.

```
PacketPayload \{
  signals: u4,
  num\_surbs: u4,   // = zero
  surbs: [Surb; 0] // empty
  user\_payload: [u8; <variable length>]
\}
```

The `PacketPayload` then MUST be padded to get `PaddedPayload` as described in section 3.4.4.

4.2.2. Reply payload encryption

The `SenderKey` (`sender_key` field) is extracted from the used `SURB`.

The `PaddedPayload` of the reply packet MUST be encrypted as follows:

1. Generate `Kprp_reply` = KDF("HASH_KEY_REPLY_PRP", `SenderKey`, `Pseudonym`)
2. Transform the `PaddedPayload` as using PRP:

```
EncPayload = PRP(Kprp\_reply, PaddedPayload)
```

This finalizes all the fields of the `HOPR_Packet` of for the reply. The `HOPR_Packet` is sent to the peer represented by a public key, corresponding to `first_hop_ident` extracted from the `SURB` (that is the first peer on the return path). For this operation, the mapper MAY be used to get the actual public key to route the packet.

5. Packet processing

This section describes the behavior of processing a `HOPR_Packet` instance, when received by a peer (hop). Let `Phop_priv` be the private key corresponding to the public key `Phop` of the peer processing the packet.

Upon reception of a byte-sequence that is at least `|HOPR_Packet|` bytes-long, the `|Ticket|` is separated from the sequence. As per section 2.4, the order of the fields in `HOPR_Packet` is canonical, thefore the `Ticket` starts exactly at |HOPR_Packet| - |Ticket| byte-offset.

The resulting Meta packet is processed first, and if this processing is successful, the `Ticket` is validated as well, as defined in `RFC-0005`.

If any of the operations fail, the packet MUST be rejected, and subsequently, it MUST be acknowledged. See Section 5.4.

5.1. Advancing the Alpha value

To recover the `SharedSecret_i`, the `Alpha` value MUST be transformed using the following transformation:

1. Compute `SharedPreKey_i` = `Phop_priv` * `Alpha`
2. `SharedSecret_i` = KDF("HASH_KEY_SPHINX_SECRET", `SharedPreKey_i`, `Phop`)
3. `B_i` = KDF("HASH_KEY_SPHINX_BLINDING", `SharedPreKey_i`, `Alpha`)
4. `Alpha` = `B_i` * `Alpha`

Similarly, as in section 3.2, the `B_i` in step 3 MAY be additionally transformed so that it conforms to a valid field scalar usable in step 4.

Shall the process fail in any of these steps (due to invalid EC point or field scalar), the process MUST terminate with an error and the entire packet MUST be rejected.

Also derive the ReplayTag = KDF("HASH_KEY_PACKET_TAG", SharedSecret_i). Verify that ReplayTag has not yet been seen by this node, and if yes, the packet MUST be rejected.

5.2. Header processing

In the next steps, the Header (field header) processed using the derived SharedSecret_i .

As per section 3.4.1, the Header consists of two byte sequences of fixed length: the header and oa_tag. Let |T| be the fixed byte-length of oa_tag and |Header| be the fixed byte-length of header. Also denote |PoRString_i|, which have equal for all i, as |PoRString|. Likewise, |ID_i| for all i as |ID|.

1. Generate K_tag = KDF("HASH_KEY_TAG", 0, SharedSecret_i)
2. Compute oa_tag_c = OA(K_tag, header)
3. If oa_tag_c != oa_tag, the entire packet MUST be rejected.
4. Initialize PRG with SharedSecret_i and XOR PRG bytes to header
5. The first byte of the transformed header represents the HeaderPrefix:
    - Verify that the first 3 most significant bits represent the supported version (001), otherwise the entire packet MUST be rejected.
    - If 3 least significant bits are not all zeros (meaning this node not the recipient):
        - Let i be the 3 least significant bits of HeaderPrefix
        - Set ID_i = header[|HeaderPrefix|..|HeaderPrefix|+|ID|]
    - Tag_i = header[|HeaderPrefix|+|ID|..|HeaderPrefix|+|ID|+|T|]
    - PoRString_i = header[|HeaderPrefix|+|ID|+|T|..|HeaderPrefix|+|ID|+|PoRString|]
    - Shift header by |HeaderPrefix|+|ID|+|T|..|HeaderPrefix|+|ID|+|PoRString| bytes left (discarding those bytes)
    - Seek the PRG to the position |Header|
    - Apply the PRG keystream to header
    - Otherwise, if all 3 least significant bits are all zeroes, it means this node is the recipient:

- Recover pseudonym as header[|HeaderPrefix|..|HeaderPrefix|+|Pseudonym|]
- Recover the 5th and 4th most significant bit (NoAckFlag and ReplyFlag)

5.3. Packet processing

In the next step, the encrypted_payload is decrypted:

1. Generate Kprp = KDF("HASH_KEY_PRP", SharedSecret_i)
2. Transform the encrypted_payload using PRP:

```
new\_payload = PRP(Kprp, encrypted\_payload)
```

5.3.1. Forwarded packet

If the processed header indicated that the packet is destined for another node, the new_payload is the encrypted_payload:EncryptedPayload . The updated header and alpha values from the previous steps are used to construct the forwarded packet. A new ticket structure is created for the recipient (as described in RFC-0005), while the current ticket structure MUST be verified (as also described in RFC-0005).

The forwarded packet MUST have the identical structure :

```
HOPR\_Packet \{
  alpha: Alpha,
  header: Header,
  encrypted\_payload: EncPayload,
  ticket: Ticket
\}
```

5.3.2. Final packet

If the processed header indicated that this node is the final destination of the packet, the `ReplyFlag` is used to indicate subsequent processing.

5.3.2.1. Forward packet

If the `ReplyFlag` is set to 0, the packet is a forward (not a reply) packet.

The `new_payload` MUST be the `PaddedPayload`.

5.3.2.2. Reply packet

If the `ReplyFlag` is set to 1, it indicates that this is a reply packet that requires further processing. The `pseudonym` extracted during header processing is used to find the corresponding `ReplyOpener`. If it is not found, the packet MUST be rejected.

Once the `ReplyOpener` is found, the `rp_shared_secrets` are used to decrypt the `new_payload`:

For each `SharedSecret_k` in `rp_shared_secrets` do:

1. Generate `Kprp` = KDF("HASH_KEY_PRP", `SharedSecret_k`)
2. Transform the `new_payload` using PRP:

```
new\_payload = PRP(Kprp, new\_payload)
```

This will invert the PRP transformations done at each forwarding hop. Finally, the additional reply PRP transformation has to be inverted (using `sender_key` from the `ReplyOpener` and `pseudonym`):

1. Generate `Kprp_reply` = KDF("HASH_KEY_REPLY_PRP", `sender_key`, `pseudonym`)

2. Transform the new_payload as using PRP:

```
new\_payload = PRP(Kprp\_reply, new\_payload)
```

The new_payload now MUST be PaddedPayload.

5.3.3. Interpreting the payload

In any case, the new_payload is PaddedPayload.

The zeros are removed until the padding_tag is found. If it cannot be found, the packet MUST be rejected. The payload:PacketPayload is extracted. If num_surbs > 0, the contained SURBs SHOULD be stored to be used for future reply packet creation, indexed by the pseudonym extracted during header processing.

The user_payload can then be used by the upper protocol layer.

5.4. Ticket verification and acknowledgement

In the next step the ticket MUST be pre-verified using the SharedSecret_i, as defined in RFC-0005. If the packet was not destined for this node (not final) OR the packet is final and the NoAckFlag is 0, the packet MUST be acknowledged.

The acknowledgement of the successfully processed packet is created as per RFC-0005 using SharedKey_i+1_ack = HS(SharedSecret_i,"HASH_ACK_KEY"). The SharedKey_i+1_ack is the scalar in the field of the elliptic curve chosen in RFC-0005. The acknowledgement is sent back to the previous hop.

This is done creating and sending a standard forward packet directly to the node the original packet was received from. The NoAckFlag on this packet MUST be set. The user_payload of the packet contains the encoded Ackno

wledgement structure as defined in RFC-0005. The num_surbs of this packet MUST be set to 0.

If the packet processing was not successful at any point, a random acknowledgement MUST be generated (as defined in RFC-0005) and sent to the previous hop.

6. Appendix A

The current version is instantiated using the following cryptographic primitives:

- Curve25519 elliptic curve with the corresponding scalar field
- PRP is instantiated using Lioness wide-block cipher [04] over Chacha20 and Blake3
- PRG is instantiated using Chacha20 [02]
- OA is instantiated with Poly1305 [02]
- KDF is instantiated using Blake3 in KDF mode, where the optional salt S is prepended to the key material K: KDF(C,K,S)=blake3_kdf(C,S||K). If S is omitted: KDF(C,K)=blake3_kdf(C,K).
- HS is instantiated via hash_to_field using secp256k1_XMD:SHA3-256_SSWU_RO_ as defined in [04]. S is used a the secret input, and T as an additional domain separator.

7. References

[01] Danezis, G., & Goldberg, I. (2009). Sphinx: A Compact and Provably Secure Mix Format. 2009 30th IEEE Symposium on Security and Privacy, 262-277.

[02] Bradner, S. (1997). Key words for use in RFCs to Indicate Requirement Levels. IETF RFC 2119.

[03] Nir, Y., & Langley, A. (2015). ChaCha20 and Poly1305 for IETF Protocols. IETF RFC 7539.

[04] Faz-Hernandez, A., et al. (2023). Hashing to Elliptic Curves. IETF RFC 9380.

[05] Anderson, R., & Biham, E. (1996). Two practical and provably secure block ciphers: BEAR and LION. In International Workshop on Fast Software Encryption (pp. 113-120). Berlin, Heidelberg: Springer Berlin Heidelberg.

5 RFC-0005: Proof of Relay

- RFC Number: 0005
- Title: Proof of Relay
- Status: Implementation
- Author(s): Lukas Pohanka (@NumberFour8), Qianchen Yu (@QYuQianchen)
- Created: 2025/04/02
- Updated: 2025/08/28
- Version: v0.9.0 (Draft)
- Supersedes: none
- Related Links: RFC-0002, RFC-0004

1. Abstract

This RFC describes the structures and protocol for establishing a Proof of Relay (PoR) of HOPR packets sent between two peers over a relay. In addition, such PoR can be used to unlock incentives for the node relaying the packets to the destination.

2. Motivation

This RFC aims to solve the assurance of packet delivery between two peers inside a mixnet. In particular, when data are sent from a sender (peer A) using node B as a relay node to deliver the packet to the destination node C, the assurance is established that:

1. node A has guarantees that node B delivered A's packets to node C
2. after successful relaying to C, node B possesses a cryptographic proof of the delivery
3. node B can use such proof to claim a reward from node A
4. the identity of node A is not revealed to node C

3. Terminology

This document builds upon standard terminology established in RFC-0002. Mentions to "HOPR packets" or "mixnet packets" refer to a particular structure (`HOPR_Packet`) defined in RFC-0004.

In addition, this document also uses the following terms:

- Channel (or Payment channel): a unidirectional directed relation of two parties (source node and destination node) that holds a monetary balance, that can be paid out by source to the destination, if certain conditions are met.
- Ticket: a structure that holds cryptographic material allowing probabilistic fund transfer within the Payment channel.
- domainSeparator: To prevent replay attacks across different domains (e.g., contracts, chains) where the ledger that stores channel states MAY be deployed, all cryptographic signatures in the HOPR protocol are bound to a specific execution context using a domain separator.
- Notice period (T_closure): Minimum elapsed time required for an outgoing channel to transit from `PENDING_TO_CLOSE` to `CLOSED`

The above terms are formally defined in the following sections.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [01].

3.1. Cryptographic and security parameters

This document makes use of certain cryptographic and mathematical terms. A security parameter `L` is chosen, and corresponding cryptographic primitives are used in a concrete instantiation of this RFC. The specific instantiation of the current version of this protocol is given in Appendix 1.

The security parameter `L` SHALL NOT be less than 2^128 - meaning the chosen

cryptographic primitives instantiations below SHALL NOT have less than 128-bits of security.

- EC group refers to a specific elliptic curve `E` group over a finite field, where computational Diffie-Hellman problem is AT LEAST as difficult as the chosen security parameter `L`. The elements of the field are denoted using lower-case letter, whereas the elements (also referred to as elliptic curve points, or EC points) of the EC group are denoted using upper-case letters.
- MUL(a,B) represents a multiplication of an EC point `B` by a scalar `a` from the corresponding finite field.
- ADD(A,B) represents an addition of two EC points `A` and `B` from the corresponding finite field.
- Public key refers to a non-identity EC group element (or its equivalent) of a large order.
- Private key refers to a scalar from a finite field of the chosen EC group. It represents a private key for a certain public key.
- Hash `H(x)` refers to a cryptographic hash function taking an input of any size, and returning a fixed length output. Security of `H` against cryptographic attacks SHALL NOT be less than `L`.
- Verifiable Random Function (VRF) produces a pseudo-random value that is publicly verifiable but cannot be forged or precomputed.

Nodes and clients MUST implement handling for each of the above to ensure compliance and fault tolerance within the HOPR PoR protocol.

The concrete choices of the above cryptographic primitives for the implementation of version 1.0 are given in Appendix 1.

4. Payment channels

Let A, B and C be peers participating in the mixnet. Each node is in possesion of its own private key (`Kpriv_A`, `Kpriv_B`, `Kpriv_C`) and the corresponding public key (`P_A`, `P_B`, `P_C`). The public keys of participating nodes are publicly exposed.

The public keys MUST be from an elliptic curve cryptosystem represented by an elliptic curve `E`.

Assume that node A wishes to communicate with node C, using node B as a relay. Node A then opens a logical payment channel with node B (denoted A -> B), staking some funds into this channel. Such channel will hold the current balance and additional state information shared between A and B and is strictly directed in the direction A -> B.

For the purpose of this RFC, the amount of funds MUST be strictly greater than 0 and MUST be strictly less than 2^96.

There MUST NOT be more than a single payment channel between any two nodes A and B in this direction. Since channel is uni-directional, there MAY BE channel A -> B and also B -> A at the same time.

Each channel has a unique, deterministic identifier, which is channel ID. The channel ID for A  B MUST be computed as: `channel_id=H(f(P_A)||f(P_B))` where `||` stands for byte-wise concatenation. This construction is directional (A first, then B).

The channel MUST always be in one of the 3 logical states:

1. Open
2. Pending to close
3. Closed

Such state can be described using `ChannelStatus` enumeration:

```
ChannelStatus \{ OPEN, PENDING\_TO\_CLOSE, CLOSED \}
```

There is a structure called `Channel` that MUST contain at least the following fields:

1. `source`: public key of the source node (A in this case)
2. `destination`: public key of the destination node (beneficiary, B in this case)
3. `balance` : an unsigned 96-bit integer

4. `ticket_index`: an unsigned 48-bit integer
5. `channel_epoch`: an unsigned 24-bit non-zero integer
6. `status`: one of the `ChannelStatus` values

```
Channel \{
    source: [u8; |P\_A|],
    destination: [u8; |P\_B|],
    balance: u96,
    ticket\_index: u48,
    channel\_epoch: u24,
    status: ChannelStatus
\}
```

Such structure is sufficient to describe the payment channel A -> B.

Channels are uniquely identified by the `channel_id` above. The fixed-length byte string returned by the function is called `ChannelId`.

4.1. Payment channel life-cycle

A payment channel between nodes A -> B MUST always be initiated by node A. It MUST be initialized with a non-zero `balance`, a `ticket_index` equal to `0`, `channel_epoch` equal to `1` and `status` equal to `Open`. To prevent spamming, the funding `balance` MUST be larger than `MIN_USED_BALANCE` and smaller than `MAX_USED_BALANCE`.

In such state, the node A is allowed communicate with node C via B and the node B can claim certain fixed amounts of `balance` to be paid out to it in return - as a reward for the relaying work. This will described in the later sections.

At any point in time, the channel initiator A can initiate a closure of the channel A -> B. Such transition MUST change the `status` field to `PENDING_TO_CLOSE` and this change MUST be communicated to B. In such state, the node A MUST NOT be allowed to communicate with C via B, but B MUST be allowed to still claim any unclaimed rewards from the channel. However, B MUST NOT be allowed to claim any rewards after `T_closure` has elapsed since the transition to

PENDING_TO_CLOSE. `T_closure` MUST be measured in block timestamps, and both parties MUST derive it from the same source.

After each claim is done by B, the `ticket_index` field MUST be incremented by 1, and such change MUST be communicated to both A and B. The increment MAY be done by an independent trusted third party supervising the reward claims.

The initiator A SHALL transition the channel state to `CLOSED` (changing the `status` to `CLOSED`). Such transition MUST NOT be possible before `T_closure` has elapsed. The transition MUST be communicated to B. In such state, the node A MUST NOT be allowed to communicate with C via B, and B MUST NOT be allowed to claim any unclaimed rewards from the channel. The `balance` in the channel A -> B MUST be reset to `0` and its `channel_epoch` MUST be incremented by `1`.

At any point of time when the channel is at the state other than `CLOSED`, the channel destination B MAY unilaterally transition the channel A -> B to state `CLOSED`. Node B SHALL claim unclaimed rewards before the state transition, because any unclaimed rewards becomes unclaimable after the state transit, resulting a lost for node B. To prevent spamming, the reward amount MUST be larger than `MIN_USED_BALANCE` and smaller than `MAX_USED_BALANCE`.

5. Tickets

Tickets are always created by a node that is the source (`A`) of an existing channel. It is created whenever `A` wishes to send a HOPR packet to a certain destination (`C`), while having the existing channel's destination (`B`) act as a relay.

Their creation MAY happen at the same time as the HOPR packet, or MAY be precomputed in advance when usage of a certain path is known in-prior.

A Ticket:

1. MUST be tied (via a cryptographic challenge) to a single HOPR packet (from RFC-0004)

2. the cryptographic challenge MUST be solvable by the ticket recipient (`B`) once it delivers the corresponding HOPR packet to `C`
3. the solution of the cryptographic challenge MAY unlock a reward for ticket's recipient `B` at expense of `A`
4. MUST NOT contain information about packet's destination (`C`)

5.1. Ticket structure encoding

The Ticket has the following structure:

```
Ticket \{
    channel\_id: ChannelId,
    amount: u96,
    index: u48,
    index\_offset: u32,
    encoded\_win\_prob: u56,
    channel\_epoch: u24,
    challenge: ECPoint,
    signature: ECDSASignature
\}
```

All multi-byte unsigned integers MUST use the big-endian encoding when serialized.

The `ECPoint` is an encoding of an Elliptic curve point on the chosen curve `E` that corresponds to a cryptographic challenge. Such challenge is later solved by the ticket recipient once it forwards the attached packet to the next downstream node.

The encoding (for serialization) of the `ECPoint` MUST be unique and MAY be irreversible, in a sense, that the original elliptic point on the curve `E` is not recoverable, but the encoding uniquely identifies the said point.

The `ECDSASignature` SHOULD use the ERC-2098 encoding, the public key recovery bit is stored in the most significant bit of the `s` value (which is guaranteed to be unused). Both `r` and `s` use big-endian encoding when serialized.

```
ECDSASignature \{
    r: u256
    s: u256
\}
```

The ECDSA signature of the ticket MUST be computed over the EIP-712 hash `H_ticket` of the `Ticket` typed-data using `domainSeparator` (`dst`):

```
H\_1 = H(channel\_id || amount || index || index\_offset || channel\_epoch ||
↪  encoded\_win\_prob || challenge)
H\_2 = H(0xfcb7796f0000000000000000000000000000000000000000000000000000000000000000
↪  || H\_1)`
H\_ticket = H(0x1901 || dst || H\_2)
```

The `Ticket` signature MUST be done over the same elliptic curve `E` using the private key of the ticket creator (issuer).

5.2. Construction of Proof-of-Relay (PoR) secrets

This section uses terms defined in Section 2.2 in RFC-0004, namely the `SharedSecret_i` generated for `i`-th node on the path (`i` ranges from 0 (sender node) up to `n` (destination node), i.e. `n` is equal to the path length). Note, that for 0-hop path (a direct packet from sender to destination), `n` = 1.

In the PoR mechanism, a cryptographic secret is established between relay nodes and their adjacent nodes on the route.

Upon packet creation, the Sender node creates two structures:

1. the list of `ProofOfRelayString_i` for each `i`-th node on the path for i > 0 up to `n-1`. For `n=1`, the list will be empty
2. the `ProofOfRelayValues` structure

Each `ProofOfRelayString_i` contains the `challenge` for the ticket for node the `i+1`-th and the `hint` value fort the same node. The `hint` value is later used

by the `i+1`-th node to validate that the `challenge` is not bogus, before it delivers the packet to the next hop.

Due to this later verification, the `hint` MUST use an encoding useful for EC group computations on `E` (here denoted as `RawECPoint`).

```
ProofOfRelayString\_i \{
    challenge: ECPoint,
    hint: RawECPoint
\}
```

The `ProofOfRelayValues` structure contains the `challenge` and `hint` to the first relayer on the path, plus it MUST contain information about the path length. This information is later used to set the correct price of the first ticket.

Path length MUST be always less than 4 (i.e. maximum 3 hops).

```
ProofOfRelayValues \{
    challenge: ECPoint,
    hint: RawECPoint,
    path\_len: u8
\}
```

5.2.1. Creation of Proof of Relay strings and values

Let `HS` be the Hash to Field operation defined in `RFC-0004` over the field of the chosen `E`.

The generation process of `ProofOfRelayString_i` proceeds as follows for each `i` from 0 to `n-1` :

1. The `SharedKey_i+1_ack` is derived from the shared secret (`SharedSecret_i`) provided during the HOPR packet construction. `SharedKey_i+1_ack` denotes the secret acknowledgement key for the next downstream node (`i+1`).

- if i < n : SharedKey_i+1_ack=HS(SharedKey_i,"HASH_KEY_ACK_KEY")
- if i = n : the SharedKey_i+1_ack MUST be generated as a uniformly random byte-string with the byte-length of E's field elements.

2. The own shared secret SharedKey_i_own from SharedSecret_i is generated as: SharedKey_i_own=HS(SharedKey_i,"HASH_KEY_OWN_KEY")

3. The hint value is computed:
   - if i = 0: hint=HS(SharedKey_0,"HASH_KEY_ACK_KEY")
   - if i > 0: hint=SharedKey_i+1_ack (from step 1)
4. For i > 0, the ProofOfRelayString_i is composed and added to the list:
   - challenge is computed as: challenge=MUL(SharedKey_i_own+SharedKey_i+1_ack,G) and encoded as ECPoint
   - hint is used from step 3.
5. For i = 0, the ProofOfRelayValues is created:
   - challenge is computed as: challenge=MUL(SharedKey_i_own+SharedKey_i+1_ack,G) and encoded as ECPoint
   - hint is used from step 3.
   - path_length is set to n

5.3 Creation of the ticket for the first relayer

The first ticket MUST be created by the packet Sender and MUST contain the challenge field equal to the challenge in the ProofOfRelayValues from the previous step.

Multi-hop ticket: for n > 1

In this situation, the Channel between the Sender and the next hop MUST exist and be in the OPEN state.

1. The field channel_id MUST be set according to the Channel leading from the Sender to the first packet relayer.

2. The `amount` field SHOULD be set according to an expected packet price times the number of hops on the path (that is `n` - 1).
3. The `index` field MUST be set to the `ticket_index` + 1 from the corresponding `Channel`.
4. The `index_offset` MUST be set to 1 in the current implementation.
5. The `encoded_win_prob` SHOULD be set according to expected ticket winning probability in the network.
6. The `channel_epoch` MUST be set to the `channel_epoch` from the corresponding `Channel`.

`Zero-hop ticket: n = 1`

This is a specific case when the packet is 0-hop (`n` = 1, it is sent directly from the Sender to the Recipient). If the `Channel` between the Sender and Recipient does exist, it MUST be ignored.

The `Ticket` is still created:

1. The `channel_id` MUST be set to `H(P_S||P_R)` where `P_S` and `P_R` are public keys (or their encoding) of Sender and Recipient respectively.
2. The `amount`, `index` and `channel_epoch` MUST be 0
3. The `index_offset` MUST be 1
4. The `encoded_win_prob` MUST be set to a value equivalent to the 0 winning probability

In any case, once the `Ticket` structure is complete, it MUST be signed by the Sender, who MUST be always the first ticket's issuer.

As described in Section 2.5 in RFC-0004, the complete encoded `Ticket` structure becomes part of the outgoing `HOPR_Packet`.

5.4. Ticket processing at a node

This is inherently part of the packet processing from the RFC-0004. Once a node receives a HOPR_Packet structure, the Ticket is separated and its processing is a two step process:

1. The ticket is pre-verified (this is already mentioned in section 4.4 of RFC 0003).
2. If the packet is to be forwarded to a next node, the ticket MUST be fully-verified
    - If successful, the ticket is replaced with a new ticket in the HOPR_Packet for the next hop

5.4.1. Ticket pre-verification

Failure to validate in any of the verification steps MUST result in discarding the ticket and the corresponding HOPR_Packet, and interrupting the processing further.

If the extracted Ticket structure cannot be deserialized, the corresponding HOPR_Packet MUST be discarded. It the Ticket has been issued for an unknown channel, or it does not correspond to the channel between the packet sender and the node where it is being processed, or the channel is in the CLOSED state, the corresponding HOPR_Packet MUST be discarded.

At this point, the node knows its SharedSecret_i with which it is able to decrypt the HOPR_Packet and the ProofOfRelayString_i has already been extracted from the packet header (see section 4.2 in RFC-0004).

1. SharedSecret_i is used to derive SharedSecret_i_own as per Section 4.2.1
2. The hint is extracted from the ProofOfRelayString_i
3. Compute challenge_check=ADD(SharedSecret_i_own,hint)
4. The HOPR_Packet MUST be rejected if encoding of challenge_check does not match challenge from the Ticket

If the pre-verification fails at any point, it still applies that the discarded HOPR_Packet MUST be acknowledged (as per section 4.2.3.1).

5.4.2. Ticket validation and replacement

Let corr_channel be the Channel that corresponds to the channel_id on the Ticket. This channel MUST exist and not be in the CLOSED state per previous section, otherwise the entire HOPR_Packet has been discarded.

If the packet is to be forwarded (as per section 4.3.1 in RFC-0004), the Ticket MUST be verified as follows:

1. the signature of the Ticket is verified - if the signature uses ERC-2098 encoding, the ticket issuer from the signature is recovered and compared to the public key of the packet sender (or its representation)
2. the amount MUST be checked, so that it is greater than some given minimum ticket amount (this SHOULD be done with respect to the path position)
3. the channel_epoch on the Ticket MUST be the current epoch of the corr_channel.
4. if MUST be checked that the packet sender has enough funds to cover the amount of the ticket

Once the above verifications have passed, verified ticket is stored as unaknowledged by the node and SHOULD be indexed by hint. The stored unaknowledged tickets are dealt with later (see 4.2.3).

A new Ticket for the packet forwarded to the next hop MUST be created.

The HeaderPrefix from the packet header contains the current path position, this information is further used to determine which type of ticket to create.

The path position is used to derive the number of remaining hops.

If the number of remaining hops is > 1, it MUST be checked if a Channel for the next hop exists from the current node, and if it is in the OPEN state. If not, the corresponding HOPR_Packet is discarded and the process is interrupted.

The process of Ticket creation from section 4.3 then applies, either with the Channel as the next hop channel in a multi-hop ticket (if the number of remaining hops > 1), or creates a zero-hop ticket if the number of remaining hops is 1.

The following applies in addition to 4.3:

- the amount on the ticket in the multi-hop case MAY be adjusted (typically amount from previous ticket is diminished by the packet price)
- the challenge MUST be set to challenge from the ProofOfRelayString_i extracted from the HOPR_Packet

If the ticket validation fails at any point, it still applies that the discarded HOPR_Packet MUST be acknowledged (as per section 4.2.3.1).

5.2.3. Ticket acknowledgement

The following sections first describe how acknowledgements are created when sent back to the original packet's Sender, and secondly how a received acknowledgement should be processed.

5.2.3.1. Sending acknowledgement

Per section 4.3.3 in RFC-0004, each packet without NoAckFlag set MUST be acknowledged. Such an acknowledgement becomes a payload of a 0-hop packet sent from the original packet's recipient to the original packet's sender.

```
Acknowledgement \{
    ack\_secret: ECScalar,
    signature: ECDSASignature
\}
```

There are two possibilities how the ack_secret field is calculated:

1. if the HOPR_Packet being acknowledged has been successfully processed (along with successfully validated ticket), the ack_secret MUST be calculated as:

```
ack_secret=HS(SharedSecret_i,"HASH_KEY_ACK_KEY")
```

This EC field element MUST be encoded as a big-endian integer (denoted as EC Scalar).

2. if the processing of the HOPR_Packet failed for any reason (either failure of the packet processing in RFC-0004 or during packet pre-verification or validation from Section 4.4): ack_secret is set to a randomly EC point on E.

This signature field contains the signature of the encoded ack_secret bytes. The signature done over H(ack_secret) using the private key of the acknowledging party. For this purpose the same EC cryptosystem for signing and verification as with Ticket SHOULD be used. The same encoding of the signature field is used as with the Ticket.

5.2.3.2. Receiving an acknowledgement

After the Ticket has been extracted and validated by the relay node, it awaits until the packet acknowledgement is received back from the next hop. The node SHOULD discard tickets that haven't been acknowledged for a certain given period of time.

Once an Acknowledgement is received the node MUST:

1. validate the signature of ack_secret. If invalid, the Acknowledgement MUST be discarded.
2. decode ack_secret calculate hint=MUL(ack_secret,G)

The node then searches for a previously stored unacknowledged Ticket with the corresponding hint as index.

- If a `Ticket` with corresponding `hint` is found, it MUST be marked as acknowledged and the `ack_secret` is then the missing part in the solution of the cryptographic challenge on that `Ticket` (which is corresponding to the packet that just has been acknowledged).

Let `SharedSecret_i_own` be the value from 1) in Section 4.4.1. The `response` to the `Ticket` challenge corresponding to the acknowledged packet is:

`response=ack_secret+SharedSecret_i_own`

The response is a field element of `E`.

- If no matching `Ticket` was found, the received `Acknowledgement` SHOULD be discarded.

5.2.3.3. Derivation of VRF parameters for an Acknowledged ticket

Once the ticket becomes acknowledged, the node then calculates the `vrf_V` value, that will be useful to determine if the ticket is suitable for value extraction.

Let `HC(msg,ctx)` be a suitable Hash to Curve function for `E`, where `msg` is an arbitrary binary message, `ctx` is a domain separator and whose output is a point on `E`. See Appendix 1 for a concrete choice of `HC`.

Let `P` be the ticket recipient's public key in the EC cryptosystem on `E`.

Let `a` be the corresponding private key as field element of `E`.

The field element MUST be representable as an unsigned big-endian integer so it could be used e.g. as an input to a hash function `H`. Similarly, `P` MUST be representable in an "uncompressed" form when given to a hash function as input.

Let `H_P` be an irreversible byte-representation of `P`.

Let `H_ticket` be the hash of a previously acknowledged ticket as per section 4.1.

Let `R` be a sequence of 64 uniformly randomly generated bytes using a CSPRNG.

```
B = HC(H\_P || H\_ticket, dst)
V = MUL(a, B)
r = HS(a || v || R, dst)
R\_v = MUL(r, B)
h = HS(P || V || R\_v || H\_ticket)
s = r + h * a
```

The `vrf_V` is the uncompressed representation of the EC point `V` as `X||Y`, where `X` and `Y` are big-endian unsigned integer representation of the EC point's coordinates.

6 Ticket and Channel interactions

6.1. Discovering acknowledged winning tickets

The aknowledged tickets are probabilistic in the sense that the monetary value represented by the `amount` MUST be claimable only if the aknowledged ticket is winning. This is determined using the `encoded_win_prob` field on the `Ticket`.

Let `luck` be an unsigned 56-bit integer in the big endian encoding created by truncating the output of the following hash output:

`H(H_ticket||response||vrf_V)`

The `H_ticket` is the hash of the `Ticket` as defined in section 4.1.

The `response` is a field element of `E` and MUST be encoded as big-endian unsigned integer (i.e. has the same encoding as `ECScalar`).

The `vrf_V` is a value computed by the ticket recipient during acknowledgement.

The `amount` on the `Ticket` MUST be claimable only if `luck` < `encoded_win_prob` on the `Ticket`. Such an acknowledged ticket is called winning ticket.

6.2. Claiming a winning ticket

The monetary value represented by the amount on a winning ticket can be claimable at some 3rd party which provides such service. Such a 3rd party MUST have the ability to modify global state of all the involved Channels.

Such amount SHOULD be claimable only if the Channel corresponding the winning ticket has enough balance >= amount.

Any holder of a winning ticket can claim the amount on the ticket by submitting the following:

- the entire encoded Ticket structure of the winning ticket
- response encoded as field element of E
- the public key P of the recipient of the ticket
- values V, h and s computed in Section 4.2.3.3

If the 3rd party wishes to verify the claim, it proceeds as follows. If any of the below check fails, the amount MUST not be claimable.

1. Compute H_ticket as per 4.1 and verify the ticket's signature
2. The Channel matching channel_id MUST exist, MUST NOT be CLOSED, its channel_epoch MUST match with the one on the ticket and SHOULD have balance >= amount.
3. The index on the ticket MUST be greater or equal to ticket_index on the Channel
4. The 3rd party applies appropriate encoding to obtain H_P from P. The performs the following computations:

```
B = HC(H\_P || H\_ticket, dst)
sB = MUL(s, B)
hV = MUL(h, V)
R = sB - hV
h\_check = HS(P || V || R || H\_ticket, dst)
```

Finally, the h_check MUST be equal to h.

5. The result of `MUL(response,G)` MUST be equal to the `challenge` from the `Ticket`. If unique encoding of `ECPoint` was used, their encoding MAY be compared instead.
6. The `luck` value computed using the given `V` MUST be less than the `encoded_win_prob` from the `Ticket`

To satisfy the claim, the 3rd party MAY also adjust the balance on a `Channel` that is in the opposite direction of the claim (ticket receiver -> ticket issuer), if such channel exist and is in an `OPEN` state.

Upon successful redemption, the 3rd party MUST make sure that:

1. The `balance` on `Channel` from which the claim has been made MUST be decreased by `amount`
2. The `ticket_index` on `Channel` is set to `index` + `index_offset` (where `index` and `index_offset` are from the claimed ticket)

7. Appendix 1

The current implementation of the Proof of Relay protocol (which is in correspondence with the HOPR Packet protocol from RFC-0004):

- Hash function `H` is Keccak256
- Elliptic curve `E` is chosen as secp256k1
- HS is instantiated via `hash_to_field` using `secp256k1_XMD:SHA3-256_SSWU_RO_` as defined in [02]
- HC is instantiated via `hash_to_curve` using `secp256k1_XMD:SHA3-256_SSWU_RO_` as defined in [02]
- The one-way encoding `ECPoint` is done as `Keccak256(P)` where `P` denotes secp256k1 point in uncompressed form. The output of the hash has the first 12 bytes removed, which leaves the length at 20 bytes.
- MIN_USED_BALANCE = `1e-18` HOPR.
- MAX_USED_BALANCE = `1e7` HOPR.

8. Appendix 2

This appendix describes the ticket states which are implementation specific for the current Proof Of Relay implementation as part of the HOPR protocol.

- Ticket (unsigned or signed, but not yet verified)
  - Contains all ticket fields (channel_id, amount, index, index_offset, winProb, channel_epoch, challenge, signature).
  - A Ticket without a signature MUST NOT be accepted by peers and MUST NOT be transmitted except for internal construction.
- VerifiedTicket (signed and verified)
  - The signature MUST verify against `get_hash(domainSeparator)` and recover the ticket issuer's address.
  - `verified_hash` MUST equal `Ticket::get_hash(domainSeparator)`; `verified_issuer` MUST equal the recovered signer.
- UnacknowledgedTicket (VerifiedTicket + own half-key)
  - Produced when the recipient binds its own PoR half-key to the VerifiedTicket while waiting for the downstream acknowledgement.
- AcknowledgedTicket (VerifiedTicket + PoR response)
  - Produced once the recipient learns the downstream half-key and reconstructs `Response`.
- RedeemableTicket (winning, issuer-verified, VRF-bound)
  - Produced from an AcknowledgedTicket by attaching VRF parameters derived with the redeemer's chain key and the `domainSeparator`.
  - A RedeemableTicket MUST be suitable for on-chain submission.
- TransferableWinningTicket (wire format for aggregation/transfer)
  - A compact, verifiable representation of a winning ticket intended for off-chain aggregation.

8.1. Allowed transitions

1. `Ticket--sign-->VerifiedTicket`
   - Pre-conditions:

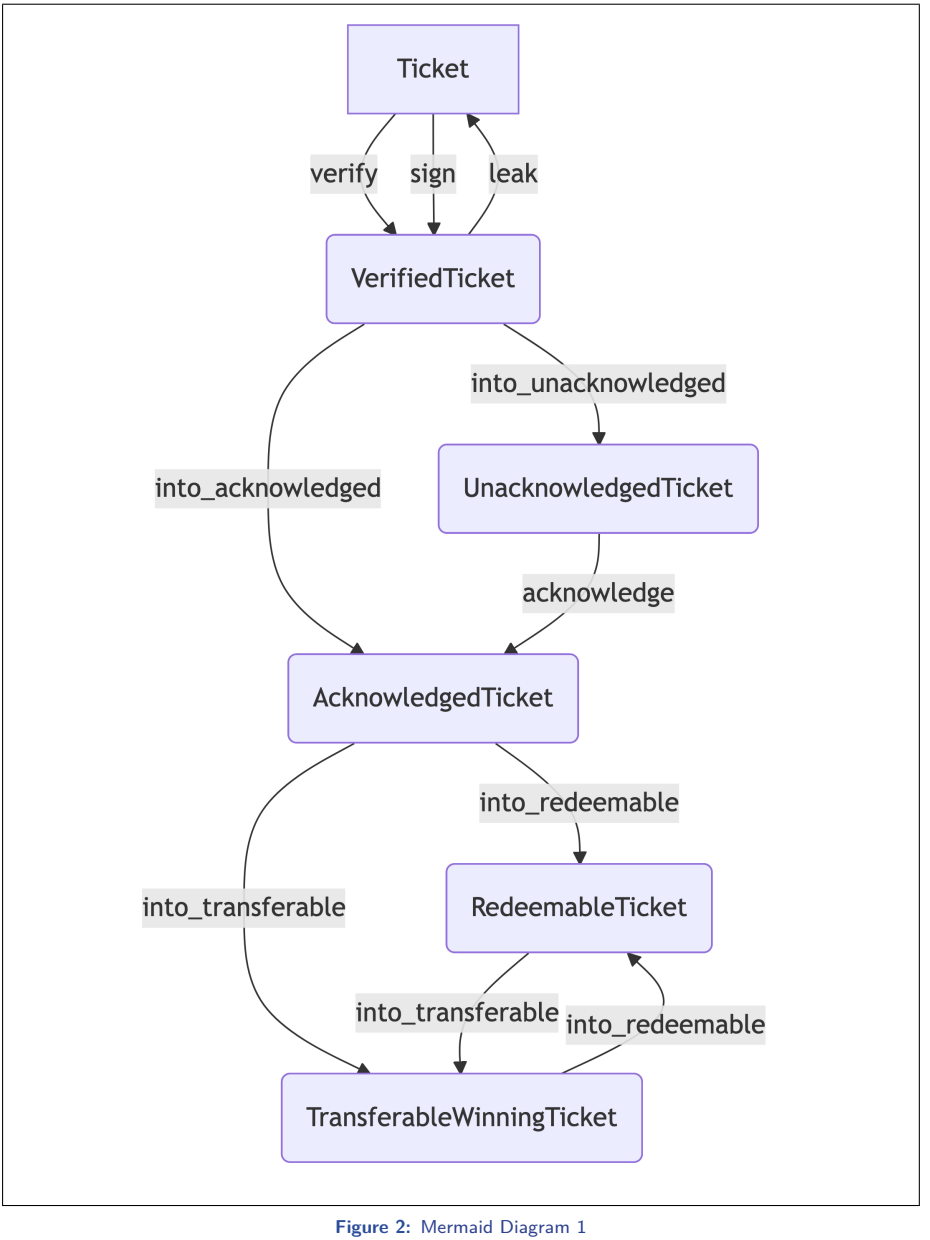


Figure 2: Mermaid Diagram 1

        - Ticket MUST include all mandatory fields and satisfy bounds (amount ≤ 10^25; index ≤ 2^48; index_offset ≥ 1; channel_epoch ≤ 2^24).
    - Post-conditions:
        - A valid ECDSA signature over `get_hash(domainSeparator)` is attached.
2. `Ticket--verify(issuer,domainSeparator)-->VerifiedTicket`
    - MUST recover `issuer` from `signature` over `get_hash(domainSeparator)`.
    - On failure, verification MUST be rejected.
3. `VerifiedTicket--into_unacknowledged(own_key)-->UnacknowledgedTicket`
    - Binds the recipient's PoR half-key. No additional checks REQUIRED.
4. `UnacknowledgedTicket--acknowledge(ack_key)-->AcknowledgedTicket`
    - Compute `Response=combine(own_key,ack_key)`.
    - The derived challenge `Response.to_challenge()` MUST equal `ticket.challenge`.
    - On mismatch, the transition MUST fail with `InvalidChallenge` and the ticket MUST remain unacknowledged.
5. `AcknowledgedTicket(Untouched)--into_redeemable(chain_keypair,domainSeparator)-->RedeemableTicket`
    - The caller (redeemer) MUST NOT be the ticket issuer (Loopback prevention).
    - Derive VRF parameters over `(verified_hash,redeemer,domainSeparator)`.
    - The resulting RedeemableTicket MAY be submitted on-chain if winning (see §3).
6. `AcknowledgedTicket(Untouched)--into_transferable(chain_keypair,domainSeparator)-->TransferableWinningTicket`
    - Equivalent to `into_redeemable` followed by conversion to transferable form; retains VRF and response.
7. `TransferableWinningTicket--into_redeemable(expected_issuer,domainSeparator)-->RedeemableTicket`

- MUST verify: `signer==expected_issuer` and the embedded signature over `get_hash(domainSeparator)`.
- MUST recompute "win" locally (see §3). On failure, MUST reject.

8. `VerifiedTicket--leak()-->Ticket`
   - Debug/escape hatch only. Implementations SHOULD avoid downgrading state in production flows.

9. Appendix 3

Domain separator (`dst`) for the current implementation (in Solidity) is derived as:

```
domainSeparator = keccak256(
  abi.encode(
    keccak256("EIP712Domain(string name,string version,uint256
    ↪  chainId,address verifyingContract)"),
    keccak256(bytes("HoprChannels")),
    keccak256(bytes(VERSION)),
    chainId,
    address(this)
  )
)
```

10. References

[01] Bradner, S. (1997). Key words for use in RFCs to Indicate Requirement Levels. IETF RFC 2119.

[02] Faz-Hernandez, A., et al. (2023). Hashing to Elliptic Curves. IETF RFC 9380.

6 RFC-0006: HOPR Mixer

- RFC Number: 0006
- Title: HOPR Mixer
- Status: Implementation
- Author(s): Tino Breddin (@tolbrino)
- Created: 2025-08-14
- Updated: 2025-09-04
- Version: v0.1.0 (Draft)
- Supersedes: none
- Related Links: RFC-0002, RFC-0004

1. Abstract

This RFC describes the HOPR Mixer component, a critical element of the HOPR mixnet that introduces temporal mixing to break timing correlation between incoming and outgoing packets. The mixer applies random delays to packets, effectively destroying temporal patterns that could be used for traffic analysis. This specification details the mixer's design, implementation requirements, and integration points to enable consistent implementations across different HOPR nodes.

2. Motivation

In mixnets, simply forwarding packets through multiple hops is insufficient to prevent traffic analysis attacks. Adversaries can correlate packets by observing timing patterns, even if packet contents are encrypted and routes are obscured. Without temporal mixing, an observer monitoring network traffic can potentially link incoming and outgoing packets based on their timing relationships.

The HOPR Mixer addresses this attack vector by:

- Breaking temporal correlations between packet arrival and departure times
- Providing configurable delay parameters to balance anonymity and performance
- Using an efficient queuing mechanism that maintains packet ordering based on release times
- All while supporting high-throughput scenarios without compromising mixing effectiveness

3. Terminology

Terms defined in RFC-0002 are used. Additional mixer-specific terms include:

mixing delay: A random time interval added to a packet's transit time through a node to prevent timing correlation attacks.

release timestamp: The calculated time when a delayed packet should be forwarded from the mixer.

mixing buffer: A priority queue that holds packets ordered by their release timestamps.

4. Specification

4.1. Overview

The HOPR Mixer follows a flow-based design which is split into these steps:

1. Accepts packets from upstream components
2. Assigns random delays to each packet
3. Stores packets in a time-ordered buffer
4. Releases packets when their delay expires

4.2. Configuration Parameters

The mixer accepts the following configuration parameters:

1. min_delay: Minimum delay applied to packets (default: 0ms)
2. delay_range: Range from minimum to maximum delay (default: 200ms)

The actual delay for each packet is randomly selected from a chosen distribution over the interval `[min_delay,min_delay+delay_range]`.

4.3. Core Components

4.3.1. Delay Assignment

When a packet arrives, the mixer:

1. Generates a random delay using a cryptographically secure random number generator
2. Calculates the release timestamp as `current_time+random_delay`
3. Wraps the packet with its release timestamp
4. Puts the wrapped packet into a buffer ordered by the release timestamp

Random delay generation:

- MUST use a CSPRNG with sufficient entropy
- MUST be independent per packet (no reuse/correlation across packets)
- SHOULD allow uniform distribution as the baseline; other distributions MAY be added via configuration

Note: Uniform distribution is a simple baseline. More advanced strategies like Poisson mixing (as used in Loopix [01]) can provide stronger anonymity properties by making packet timings less distinguishable from cover traffic patterns.

4.3.2. Mixing Buffer

The mixer maintains packets in a data structure where:

- Packets are ordered by their release timestamps
- The packet with the earliest release time is always at the top
- Insertion and extraction operations have O(log n) complexity
- If multiple packets share the same `release_time`, the ordering MUST be stable FIFO by insertion sequence

This ensures efficient processing even under high load conditions.

4.4. Operational Behavior

4.4.1. Packet Processing Flow

```
1. Packet arrives at mixer via Sender
2. Random delay is generated: delay ∈ [min\_delay, min\_delay + delay\_range]
3. Release timestamp calculated: release\_time = now() + delay
4. Packet wrapped with timestamp and inserted into buffer
5. Receiver woken if sleeping
5a. If the inserted packet has an earlier `release\_time` than the current
↪  head, re-arm the timer to the new head
6. When current\_time ≥ release\_time, packet is released to Receiver
6a. Upon wake (including after system sleep), release all packets with
↪  `release\_time` ≤ current\_time before sleeping again
```

4.4.2. Timer Management

The mixer requires a timer that is able to:

- Wake the mixer at the next packet's `release_time`
- Use minimal system calls and context switches

- Handle concurrent access safely
- Use a monotonic clock source (not wall-clock) for computing `release_time`
- Handle system sleep/clock adjustments by releasing all overdue packets immediately upon wake

NOTE: The need for a dedicated timer MAY be satisfied automatically when using a RTOS and its native waking mechanisms.

4.5. Special Cases

4.5.1. Zero Delay Configuration

When both `min_delay` and `delay_range` are zero:

- Packets pass through without mixing
- Original packet order is preserved
- Useful for testing or non-anonymous operation modes

5. Design Considerations

5.1. Performance Optimization

An implementation should prioritize:

- Minimal allocations: Pre-allocated buffer reduces memory pressure
- Efficient data structures: Binary heap provides O(log n) operations
- Lock minimization: Fine-grained locking for concurrent access
- Timer efficiency: Single shared timer reduces system overhead, including minimizing runtime system overhead by using a single thread

5.2. Abuse Resistance and Resource Limits

- Timing attacks: Random delays must use cryptographically secure randomness
- Statistical analysis: Uniform distribution is a simple baseline; stronger timing strategies (e.g., exponential/Poisson as in Loopix [01]) provide better resistance to pattern inference
- Queue bounds and DoS: The mixer MUST use a bounded buffer with backpressure. Implementations MUST define behavior when full (e.g., drop-tail oldest/newest, randomized drop, or reject upstream sends) and expose metrics/alerts to prevent memory exhaustion attacks.

5.3. Monitoring and Metrics

The mixer should track:

- Current queue size
- Average packet delay (over configurable window)

These metrics aid in:

- Performance tuning
- Detecting abnormal traffic patterns
- Capacity planning

6. Security Considerations

6.1. Threat Model

The mixer defends against:

- Timing correlation attacks: Randomized delays make linking input/output packets by timing significantly harder

- Statistical traffic analysis: Random delays reduce pattern predictability but do not eliminate all analysis
- Queue manipulation: Authenticated packet handling prevents injection attacks

6.2. Limitations

The mixer does not protect against:

- low volume spread traffic that does not produce sufficient amount of messages to be mixed within the delay window
- Global passive adversaries: With unlimited observation capability
- Active attacks: Packet dropping or delaying by malicious nodes
- Side channels: CPU, memory, or network-level information leaks

7. Drawbacks

- Increased latency: Every packet experiences additional delay
- Memory usage: Buffering packets requires memory proportional to traffic volume and queue size
- Complexity: Adds another component to the protocol stack which even makes node-local debugging harder
- Simplistic nature: The mixing does not account for the total count of elements in the buffer, with increasing amounts of messages in the mixer the generated delay can decrease without sacrificing the mixing properties.

8. Alternatives

Alternative mixing strategies considered:

- Batch mixing: Release packets in fixed-size batches (higher latency)

- Threshold mixing: Release when buffer reaches certain size (variable latency)
- Stop-and-go mixing: Fixed delays at each hop (predictable patterns)
- Poisson mixing: As implemented in Loopix [01], uses Poisson-distributed delays that make real traffic harder to distinguish from cover traffic. This can provide stronger anonymity properties but requires careful parameter tuning and integration with cover traffic.

The current continuous mixing approach with uniform distribution is a simple baseline that balances latency and anonymity while being easier to implement and analyze.

9. Unresolved Questions

- Optimal delay parameters for different network conditions
- Adaptive delay strategies based on traffic patterns
- Integration with node-local cover traffic generation
- Memory usage limits and robust overflow handling strategies

10. Future Work

- Poisson Mixing Implementation: Implement Poisson mixing (exponentially distributed per-packet delays derived from a Poisson process) as described in Loopix [01] to provide stronger anonymity properties when combined with cover traffic
- Performance optimizations for hardware acceleration

11. References

[01] Piotrowska, A. M., Hayes, J., Elahi, T., Meiser, S., & Danezis, G. (2017). The Loopix Anonymity System. 26th USENIX Security Symposium, 1199-1216.

7 RFC-0007: Economic Reward System

- RFC Number: 0007
- Title: Economic Reward System
- Status: Raw
- Author(s): Jean Demeusy (@jeandemeusy)
- Created: 2025-08-25
- Updated: 2025-08-25
- Version: v0.1.0 (Raw)
- Supersedes: none
- Related Links: none

1. Abstract

This RFC describes mechanisms around the economic reward system such as how the eligible peer set is constructed and how the rewards per peer are calculated

2. Motivation

The rewards calculation can be seen as an opaque procedure selecting who receives which amount. This RFC aims to raise the veil and clarify the reasoning behind it.

The economic reward system is a necessary component of the HOPR mixnet, as it incentivize node runners to keep their node running, in order to have a network topology as stable as possible. It must be a fair logic, to never favour or disadvantage a subset of node runners, that encourages sustainability without compromising decentralization. It must also incentivize node runners to be connected to other nodes in the network with channels. Isolated nodes are way less useful to the network than well intricately connected nodes.

3. Terminology

Terms defined in RFC-0002 are used. Additionally, this document uses the following economic system-specific terms:

- Subgraph: Off-chain data indexer (e.g., The Graph) for blockchain data (NFT holders, registered nodes, allocations, EOA balances).
- API: HOPR node HTTP API for live network data (topology, channel balances).
- EOA (Externally Owned Account): Blockchain account controlled by a private key.
- Safe: Smart contract wallet (e.g., Gnosis Safe) for holding tokens.
- CT Node: Node running the CT application.
- NFT Holder: Address holding a specific NFT.
- SessionToSocket: Object managing a UDP session and socket for a peer.
- MessageFormat: Class encoding message metadata and payload as bytes.

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [01].

4. System Overview

The HOPR CT system is designed to distribute rewards to eligible peers based on their participation and stake in the network. The process is composed of several key stages. First, the system collects and enriches peer data from a variety of sources, including both on-chain and off-chain information. Next, it applies a series of eligibility filters to determine which peers qualify for rewards. For those that are eligible, an economic model is used to calculate the number of reward units (messages) each peer should receive. Finally, the system manages the technical process of sending these messages to peers using UDP sessions, ensuring that the distribution is both fair and technically robust.

The following flowchart summarizes the overall process:

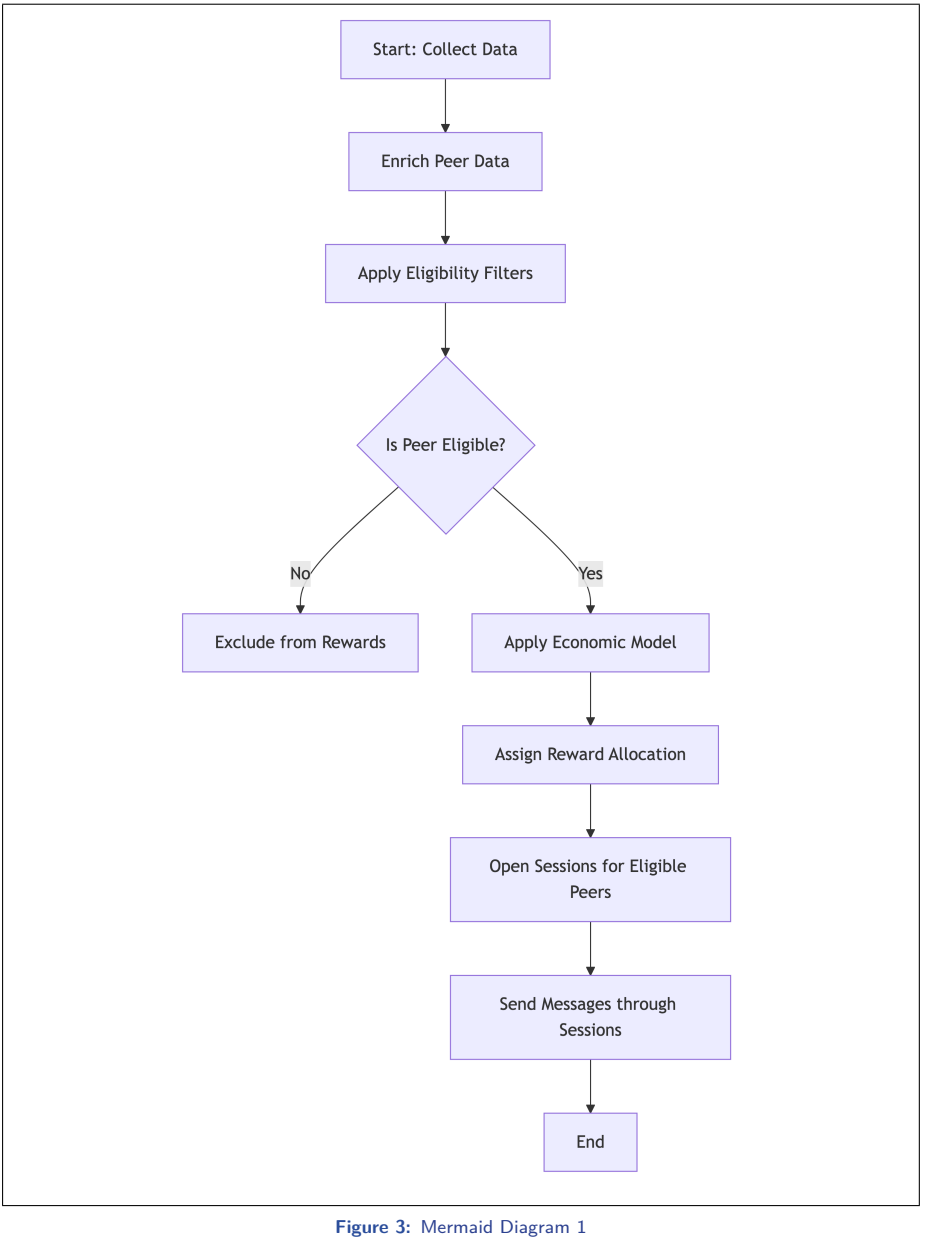


Figure 3: Mermaid Diagram 1

5. Data Collection and Enrichment

5.1 Data Sources

Data is gathered from multiple sources to build a comprehensive view of the network and its participants. The HOPR node API provides a list of currently visible peers and the network topology, including open payment channels and their balances. Subgraphs supply information about registered nodes and their associated Safes. Direct RPC calls are used to provide specific allocations to targeted accounts (which may increase a peer's effective stake) and to retrieve those accounts' EOA balances. Finally, a static list of NFT owners is used to allow rewards distribution to people holding a special "OG NFT". This combination of sources ensures that both the live state of the network and relevant historical or off-chain data are considered in the reward process.

5.2 Data Enrichment

Once collected, the data is used to enrich each peer object. Registered node information is used to associate each peer with a Gnosis Safe and other node metadata. Allocations and EOA balances are incorporated to adjust the peer's effective stake and balance, reflecting both on-chain and off-chain holdings. The network topology data is used to determine the peer's channel balance, which is important for both eligibility and reward calculation. It is important to note that NFT holder status and CT node status are not directly added to the peer object during enrichment; instead, these are checked during the eligibility filtering phase.

The following diagram illustrates the data enrichment process:

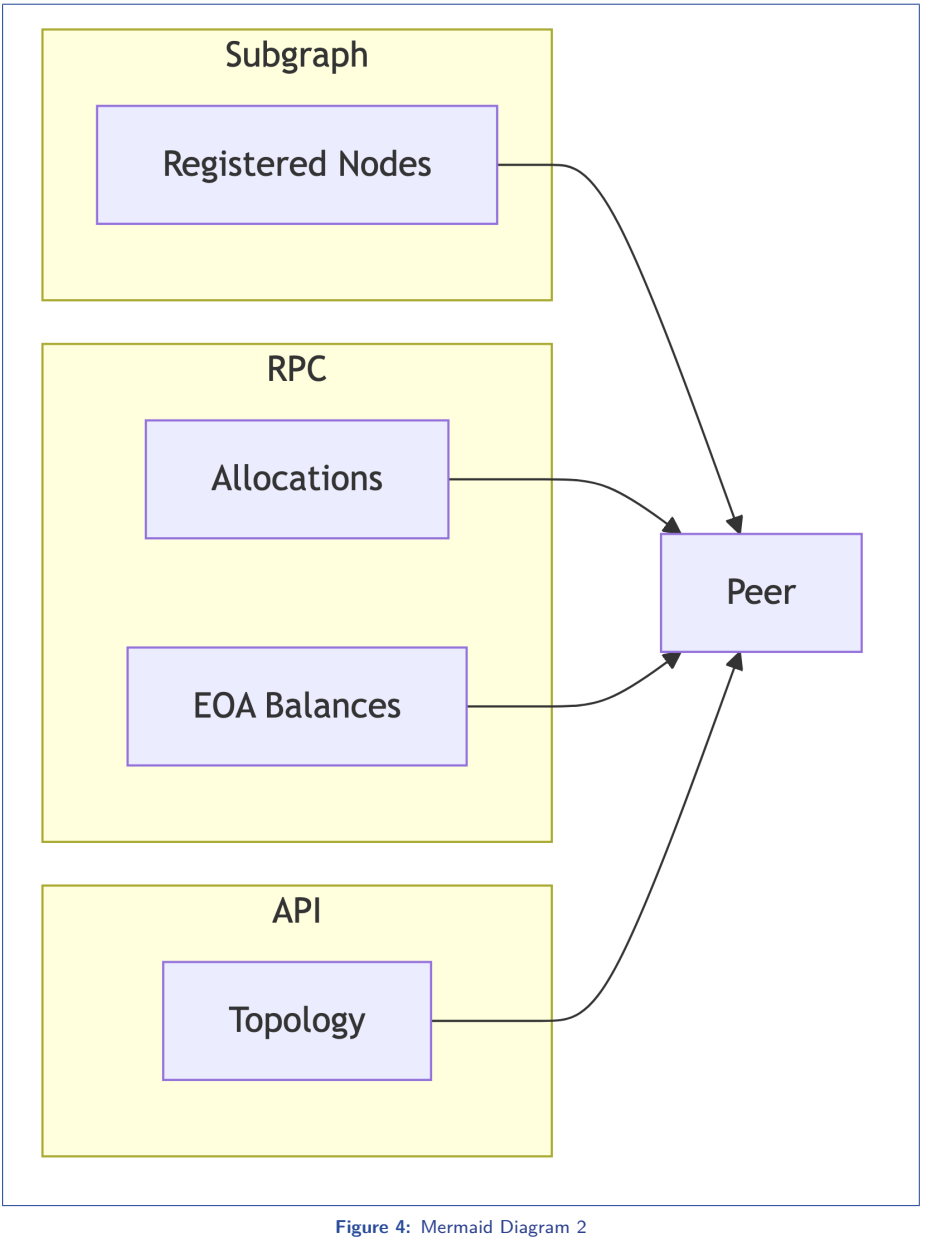


Figure 4: Mermaid Diagram 2

6. Peer Eligibility Filtering

The eligibility filtering process is designed to ensure that only peers who are meaningfully participating in the network and contributing resources are considered for rewards. The first filter checks that the peer's safe allowance meets a minimum threshold, ensuring that only active and funded peers are included. Next, the system excludes any peer that is also a CT node, to prevent self-rewarding. The NFT/stake requirement is then applied: if a peer is not an NFT holder, they must meet a higher minimum stake threshold, while NFT holders may be subject to a lower threshold. Finally, all peers must meet a minimum stake requirement, regardless of NFT status. Only those who pass all these checks are considered eligible for rewards.

The following flowchart details the filtering logic:

7. Economic Model Application

For each eligible peer, the system applies an economic model—such as a sigmoid or legacy model—to determine the number of messages (reward units) they should receive over the course of a year. The model takes into account the peer's individual stake, the total network stake, the network's capacity, and historical activity metrics such as message relay counts. The output of this model is the yearly message count for each peer, which directly determines their share of the rewards.

The following diagram shows the economic model application:

8. Message Timing and Delay Calculation

The timing between messages sent to each eligible peer is carefully calculated to ensure a fair and even distribution throughout the year. The base delay between two messages is computed as the total number of seconds in a non-leap year divided by the peer's yearly message count. To allow for efficient batching

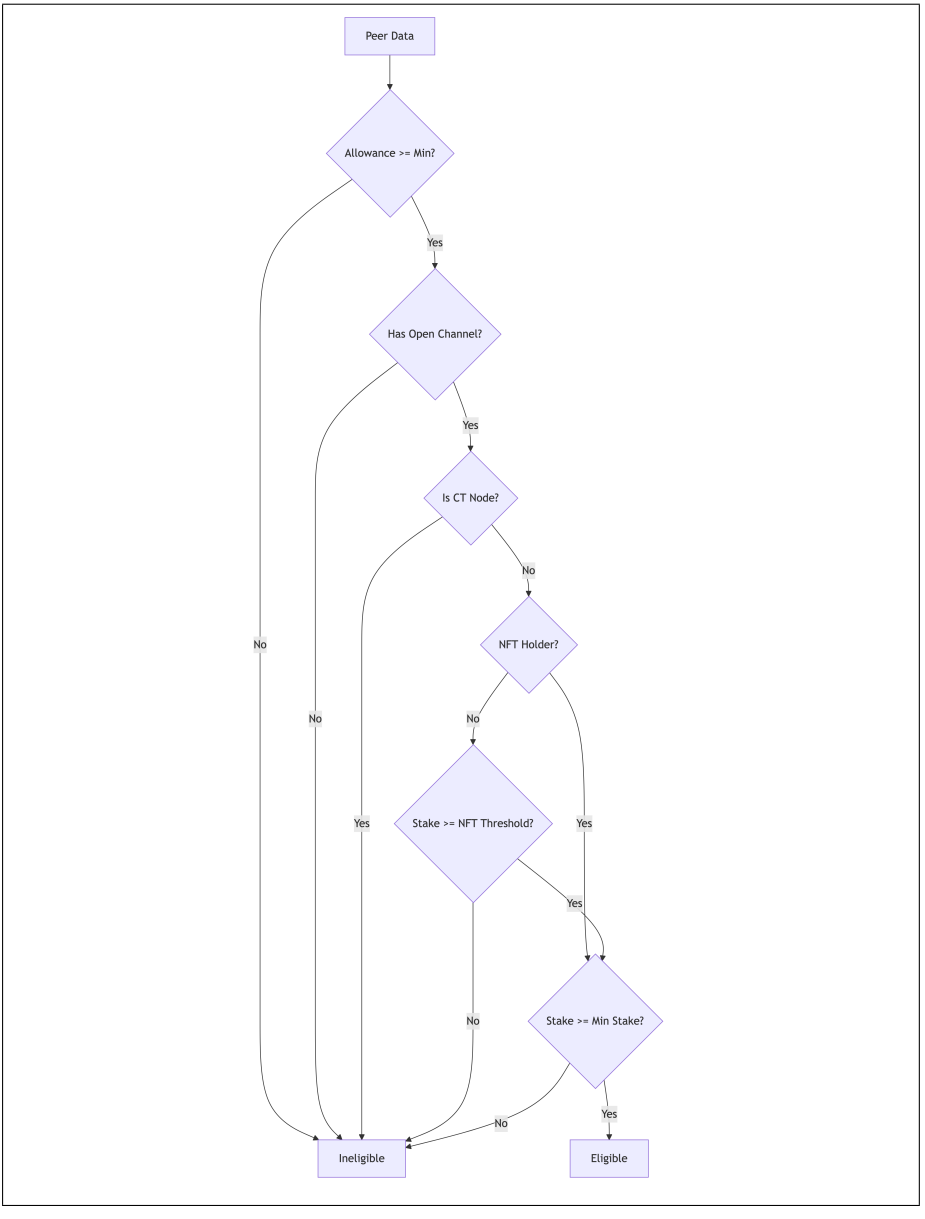


Figure 5: Mermaid Diagram 3

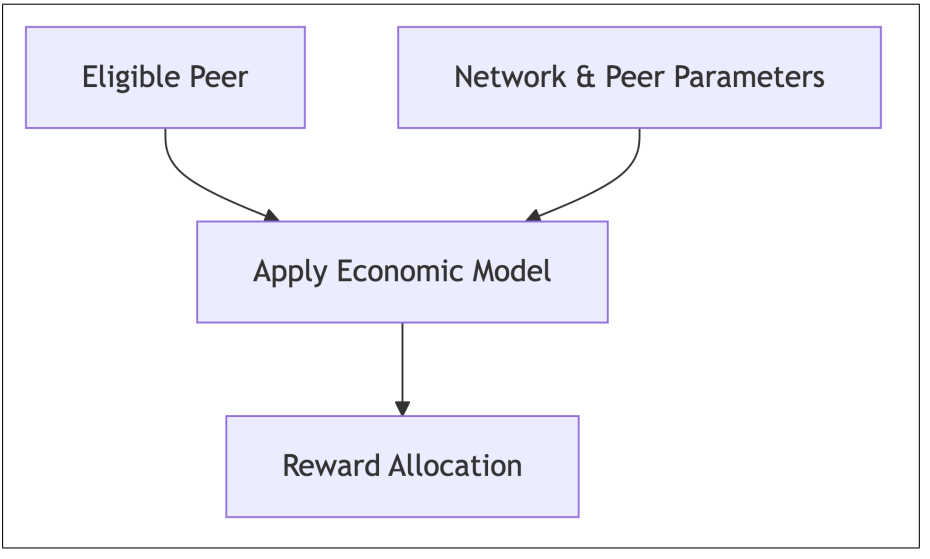


Figure 6: Mermaid Diagram 4

and aggregation, the system introduces two session parameters: `aggregated_packets` and `batch_size`. The actual sleep time between message batches is the product of the base delay, the number of aggregated packets, and the batch size. This approach allows the system to send bursts of messages followed by a pause, balancing throughput and network load. The values of these parameters can be tuned to optimize performance and reliability.

The `aggregated_packets` parameter specifies how many messages are grouped together and sent in a single relay operation, while `batch_size` determines how many such operations are performed before the system waits for the next delay interval. The product of these two parameters gives the total number of messages sent in each cycle, and the delay is applied after each cycle. This mechanism provides fine-grained control over the message sending pattern.

9. Message Sending Architecture

When it is time to send messages, the system first establishes a UDP session for each eligible peer, selecting a destination CT node at random (excluding the local node). Each session is managed by a `SessionToSocket` object, which handles both the session metadata and the underlying UDP socket. The socket is configured with appropriate buffer sizes and is closed when the session ends to prevent resource leaks.

Messages themselves are constructed using the `MessageFormat` class, which encodes all necessary metadata—such as sender, relayer, packet size, and indices—into a raw byte string. The message is padded to the required packet size and sent through the UDP socket to the destination node's address and port. The system can optionally wait for a response to measure round-trip time, which is useful for monitoring and diagnostics.

Batching multiple message sendings are handled according to the session parameters described earlier. Multiple messages can be sent in a batch, and after each batch, the system waits for the calculated delay before sending the next batch. This approach ensures that message delivery is both efficient and aligned with the reward allocation determined by the economic model.

The following flowchart summarizes the message sending process:

10. Security and Monitoring

Security and monitoring are integral to the HOPR CT reward distribution process. To ensure transparency and facilitate troubleshooting, all delays and message counts are tracked using Prometheus metrics. This allows operators and developers to monitor the system's performance in real time, detect anomalies, and analyze historical trends.

Resource management is also a key concern. The system is designed to manage sessions and sockets carefully, ensuring that resources are allocated and released appropriately. Sockets are closed when sessions end, and sessions

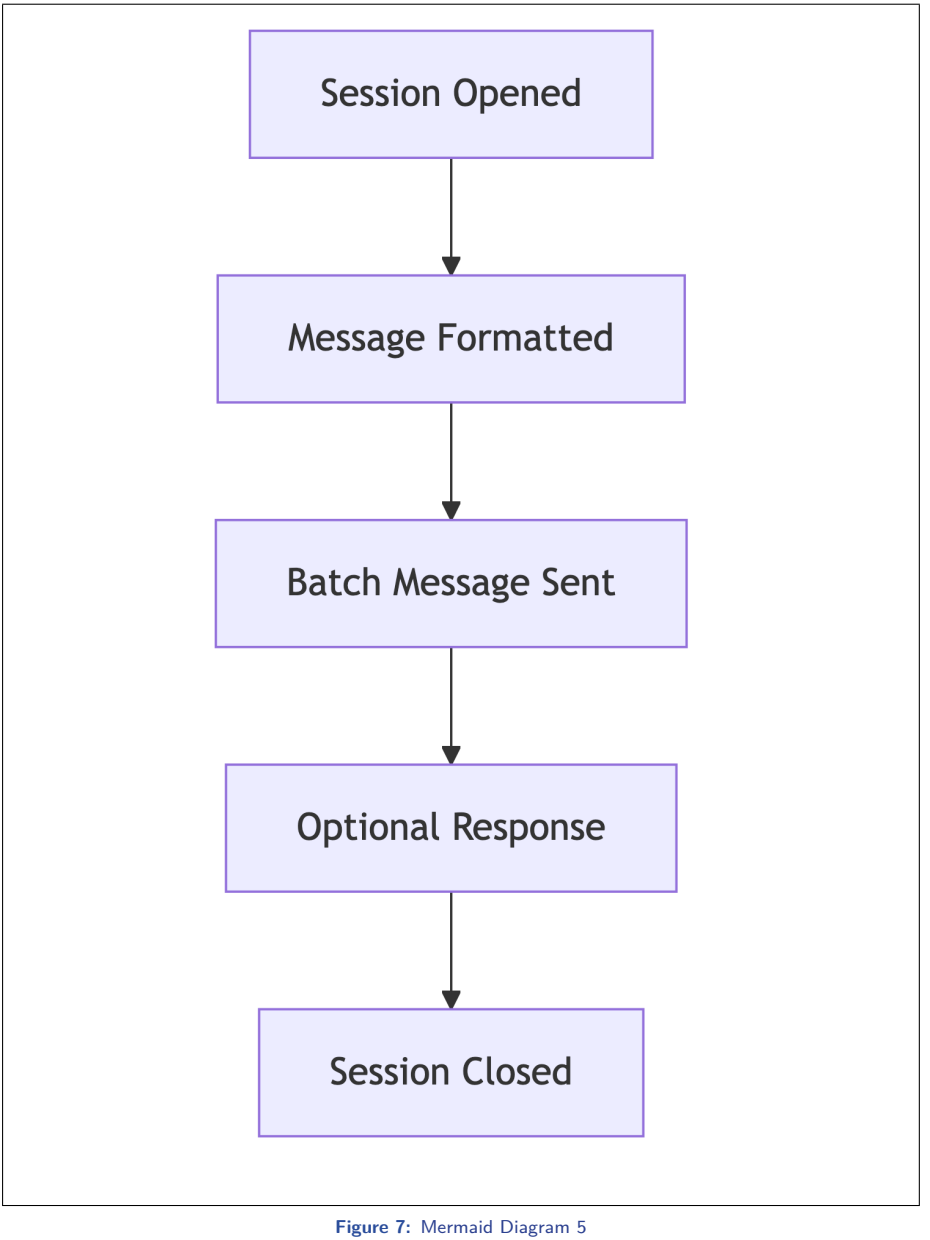


Figure 7: Mermaid Diagram 5

are only maintained for as long as they are needed. This approach helps prevent resource leaks, which could otherwise degrade system performance or cause failures over time.

Finally, the system enforces strict eligibility checks before sending messages. Only peers that have open payment channels and valid, active sessions are eligible to receive messages. This ensures that rewards are distributed only to those who are actively participating in the network and have met all necessary criteria, further enhancing the security and integrity of the reward process.

11. Appendix: Data Structures

Registered Node

Variable Name	Type	Purpose
address	str	Node's unique address
safe	Safe	Associated Gnosis Safe object
...	...	(Other metadata as provided by subgraph)

Safe

Variable Name	Type	Purpose
address	str	Safe contract address
balance	Balance	Total balance held in the safe
allowance	Balance	Allowance available for node operations
additional_balance	Balance	Extra balance from allocations/EOA
owners	list	List of owner addresses
...	...	(Other metadata as provided by subgraph)

Allocation

Variable Name	Type	Purpose
address	str	Allocation contract address
unclaimed_amount	Balance	Amount not yet claimed
linked_safes	set	Safes associated with this allocation
num_linked_safes	int	Number of safes linked

EOA Balance

Variable Name	Type	Purpose
address	str	EOA address
balance	Balance	Balance held by the EOA
linked_safes	set	Safes associated with this EOA
num_linked_safes	int	Number of safes linked

Topology Entry

Variable Name	Type	Purpose
address	str	Peer address
channels_balance	Balance	Total balance in outgoing channels

Peer (Enriched)

Variable Name	Type	Purpose
address	Address	Peer's unique address
version	Version	Peer's software version
safe	Safe	Associated safe object
channel_balance	Balance	Balance in outgoing channels
yearly_message_count	int/None	Calculated reward allocation
params	Parameters	Application parameters
running	bool	Is the peer currently active
...	...	(Other runtime attributes)

12. References

[01] Bradner, S. (1997). Key words for use in RFCs to Indicate Requirement Levels. IETF RFC 2119.

13. Changelog

- 2025-06-26: Initial draft.

8 RFC-0008: Session Data Protocol

- RFC Number: 0008
- Title: Session Data Protocol
- Status: Implementation
- Author(s): Tino Breddin (@tolbrino), Lukas Pohanka (@NumberFour8)
- Created: 2025-08-15
- Updated: 2025-09-05
- Version: v0.1.0 (Draft)
- Supersedes: none
- Related Links: RFC-0002, RFC-0004, RFC-0009

1. Abstract

This RFC specifies the HOPR Session Data Protocol, which provides reliable and unreliable data transmission capabilities over the HOPR mixnet. The protocol implements TCP-like [01] features including message segmentation, reassembly, acknowledgement, and retransmission while maintaining simplicity and efficiency. This protocol works in conjunction with the HOPR Session Start Protocol (see RFC-0009) to provide complete session management capabilities for applications within the HOPR mixnet ecosystem.

2. Motivation

The HOPR mixnet uses HOPR packets RFC-0004 to send data between nodes. This fundamental packet sending mechanisms however works, similar to UDP [03], as a fire-and-forget mechanisms and does not provide any higher-level features any application developer would expect. To ease adoption a HOPR node needs a way for existing applications to use it without having to implement TCP [01] or UDP all over again. Since HOPR protocol is not IP-based, such implementation would require IP protocol emulation.

The HOPR Session Data Protocol fills that gap by providing reliable and unreliable data transmission capabilities to applications. Session establishment and lifecycle management is handled by the HOPR Session Start Protocol RFC-0009, while this protocol focuses exclusively on data transmission.

3. Terminology

Terms defined in RFC-0002 are used. Additionally, this document defines the following session-specific terms:

- Frame: A logical unit of data transmission in the Session Protocol. Frames can be of arbitrary length and are identified by a unique Frame ID.
- Segment: A fixed-size fragment of a frame. Frames are split into segments for transmission, with each segment carrying metadata about its position within the frame.
- Frame ID: A 32-bit unsigned integer that uniquely identifies a frame within a session (1-indexed). Frame ID values are interpreted as big-endian unsigned integers.
- Sequence Number (SeqNum): An 8-bit unsigned integer indicating a segment's position within its frame (0-indexed).
- Sequence Flags (SeqFlags): An 8-bit value encoding additional segment sequence metadata.
- Session Socket: The endpoint abstraction that implements the Session Protocol, available in both reliable and unreliable variants.
- MTU (Maximum Transmission Unit): The maximum size of a single HOPR protocol message, denoted as `C` throughout this specification.
- Terminating Segment: A special segment that signals the end of a session.

4. Specification

4.1 Protocol Overview

The HOPR Session Data Protocol operates at version 1 and consists of three message types that work together to provide reliable or unreliable data transmission:

1. Segment Messages: Carry actual data fragments
2. Retransmission Request Messages: Request missing segments
3. Frame Acknowledgement Messages: Confirm successful frame receipt

The protocol supports two operational modes:

- Unreliable Mode: Fast, stateless operation similar to UDP [03]
- Reliable Mode: Stateful operation with acknowledgements and retransmissions

Session establishment and lifecycle management is handled by the HOPR Session Start Protocol. All multi-byte integer fields use network byte order (big-endian) encoding to ensure consistent interpretation across different architectures.

4.2 Session Data Protocol Message Format

All Session Data Protocol messages follow a common structure:

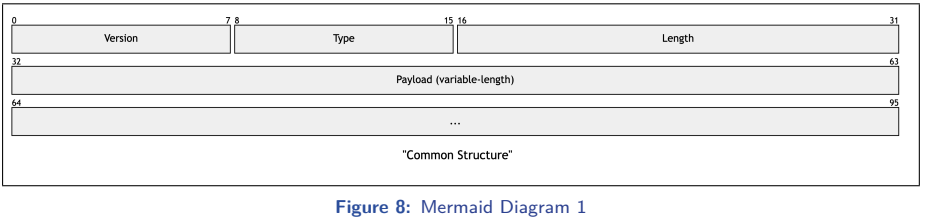


Figure 8: Mermaid Diagram 1

Field	Size	Description	Value
Version	1 byte	Protocol version	MUST be `0x01` for version 1
Type	1 byte	Message type discriminant	See Message Types table below
Length	2 bytes	Payload length in bytes	Maximum is `C-4`
Payload	Variable	Message-specific data	Format depends on message type

4.2.1 Message Types

Type Code	Name	Description
`0x00`	Segment	Carries actual data fragments
`0x01`	Retransmission Request	Requests missing segments
`0x02`	Frame Acknowledgement	Confirms successful frame receipt

4.2.2 Byte Order

All multi-byte integer fields and values in the Session Data Protocol MUST be encoded and interpreted in network byte order (big-endian). This applies to:

Protocol Message Fields:

- Length field (2 bytes) in the common message format
- Frame ID field (4 bytes) in Segment, Retransmission Request, and Frame Acknowledgement messages
- Any future numeric fields added to the protocol

This requirement ensures consistent interpretation across different architectures and prevents interoperability issues between implementations.

4.3 Segment Message

4.3.1 Segment Structure

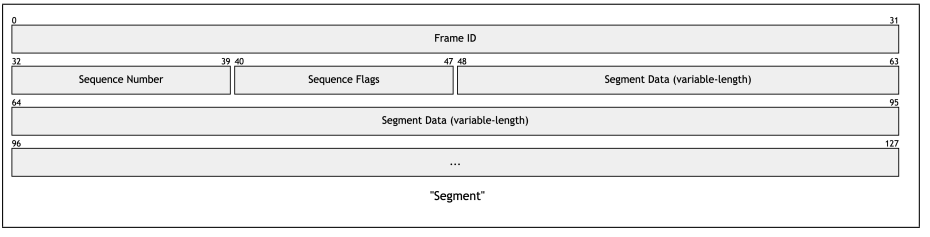


Figure 9: Mermaid Diagram 2

Field	Size	Description	Valid Range
Frame ID	4 bytes	Frame identifier	1 to 4,294,967,295
Sequence Number	1 byte	Segment position within frame (0-based)	0-63
Sequence Flags	1 byte	Segment metadata flags	See Sequence Flags table below
Segment Data	Variable	Payload data	0 to (`C-10`) bytes

4.3.2 Sequence Flags Bitmap

Bit	Flag Name	Description	Values
7	Termination Flag	Indicates terminating segment	`0` = Normal, `1` = Terminating
6	Reserved	Reserved for future use	MUST be `0`

Bit	Flag Name	Description	Values
5-0	Segment Count	Total segments in frame minus 1	0-63 (1-64 segments)

4.3.3 Segmentation Rules

Rule	Requirement	Description
Segmentation Threshold	MUST	Frames MUST be segmented when larger than (C-10) bytes
Maximum Segments	MUST	Maximum 64 segments per frame (6-bit sequence length field limit)
Segment Sizing	SHOULD	Each segment except the last SHOULD be of equal size
Empty Segments	MUST	Empty segments MUST be valid (used for terminating segments)
Frame ID Ordering	MUST	Frame IDs MUST be monotonically increasing within a session

4.3.4 Protocol Constants

Constant	Value	Description
Protocol Version	0x01	Current protocol version
Segment Overhead	10 bytes	Header overhead per segment (4 common + 6 segment)
Maximum Frame ID	4,294,967,295	Maximum 32-bit frame identifier
Maximum Segments	64	Maximum segments per frame
Maximum Payload Length	C-4 bytes	Maximum message payload size

4.4 Retransmission Request Message

4.4.1 Request Structure

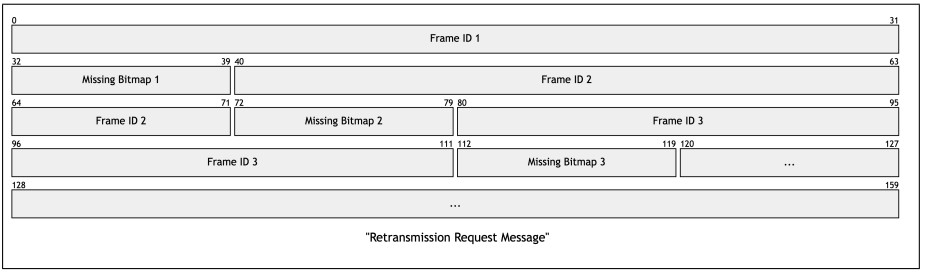


Figure 10: Mermaid Diagram 3

The message contains a sequence of 5-byte entries:

Field	Size	Description	Format
Frame ID	4 bytes	Frame identifier	1 to 4,294,967,295
Missing Bitmap	1 byte	Bitmap of missing segments	See Missing Bitmap table below

4.4.2 Missing Bitmap Format

Bit	Sequence Number	Description
0	Segment 0	`1` = Missing, `0` = Received
1	Segment 1	`1` = Missing, `0` = Received
2	Segment 2	`1` = Missing, `0` = Received
3	Segment 3	`1` = Missing, `0` = Received
4	Segment 4	`1` = Missing, `0` = Received
5	Segment 5	`1` = Missing, `0` = Received

Bit	Sequence Number	Description
6	Segment 6	`1` = Missing, `0` = Received
7	Segment 7	`1` = Missing, `0` = Received

Note: This message MUST be used only for frames with up to 8 segments (due to bitmap size limitation). Reliable sessions are limited to 7 segments per frame. Unreliable sessions SHOULD not have this limitation.

4.4.3 Request Rules

Rule	Requirement	Description
Ordering	MUST	Entries MUST be ordered by Frame ID (ascending)
Padding	MAY	Frame ID of `0` indicates padding (ignored)
Entry Limit	MUST	Maximum entries per message: `(C-4)/5`
Segment Limit	MUST	Only the first 8 segments per frame can be requested

4.5 Frame Acknowledgement Message

4.5.1 Acknowledgement Structure

Field	Size	Description	Rules
Frame ID List	4 bytes each	List of fully received frame identifiers	See Acknowledgement Rules table below

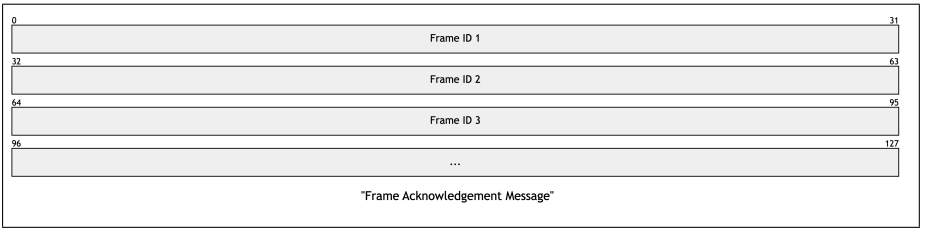


Figure 11: Mermaid Diagram 4

4.5.2 Acknowledgement Rules

Rule	Requirement	Description
Ordering	MUST	Frame IDs MUST be in ascending order
Padding	MAY	Frame ID of `0` indicates padding (ignored)
Entry Limit	MUST	Maximum frame IDs per message: `(C-4)/4`
Completeness	MUST	Only acknowledge frames that are fully received

4.6 Protocol State Machines

4.6.1 Unreliable Socket State Machine

4.6.2 Reliable Socket State Machine

4.7 Timing and Reliability Parameters

4.7.1 Unreliable Mode

- No acknowledgements or retransmissions
- Frames may be delivered out-of-order
- No delivery guarantees
- Suitable for real-time or loss-tolerant applications

4.7.2 Reliable Mode Parameters

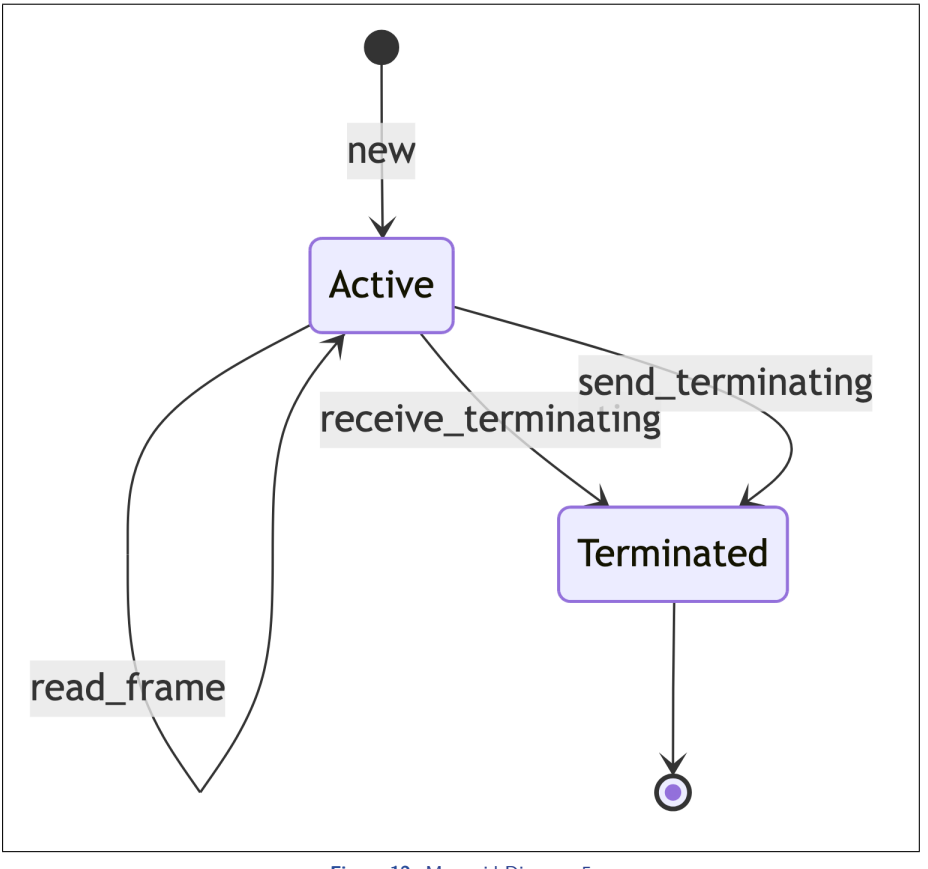


Figure 12: Mermaid Diagram 5

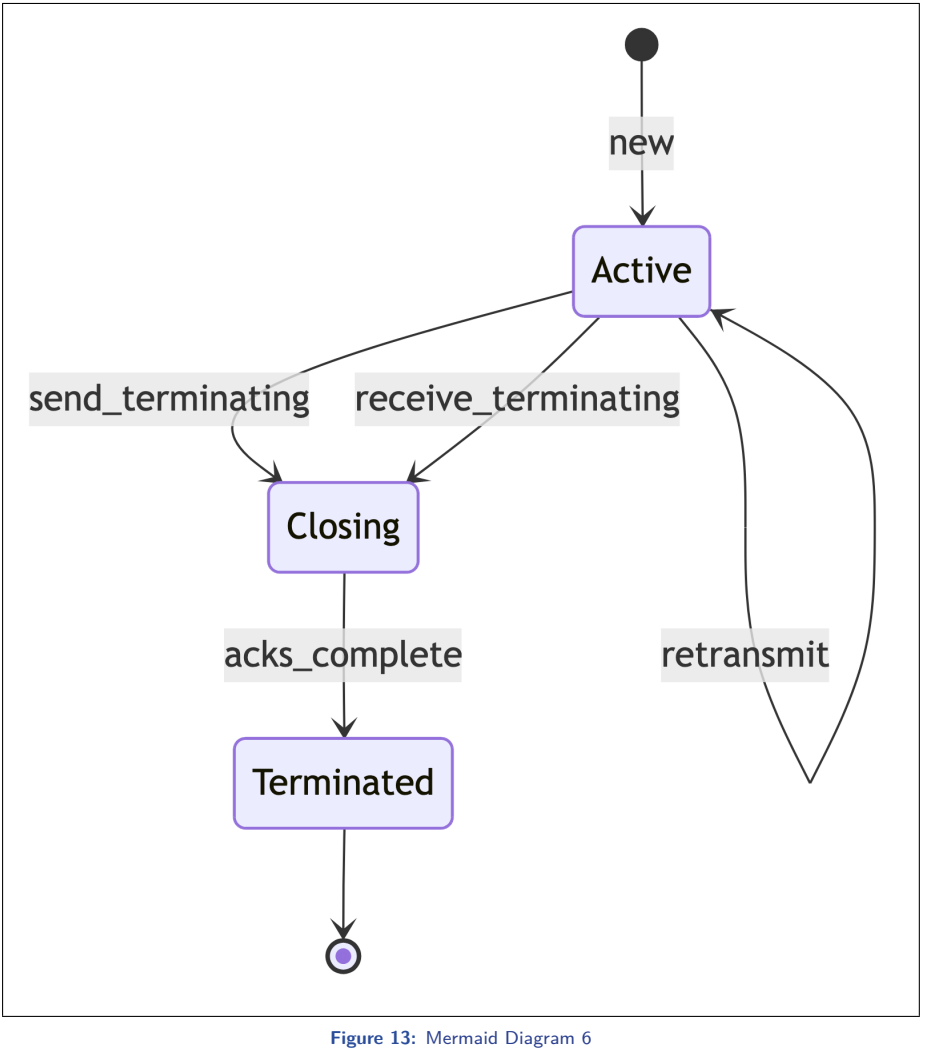


Figure 13: Mermaid Diagram 6

Parameter	Default Value	Description	Requirement
Frame Timeout	800ms	Time before requesting retransmission	SHOULD be configurable
Acknowledgement Window	255 frames	Maximum unacknowledged frames	MUST NOT exceed 255
Retransmission Limit	3 attempts	Maximum retransmission attempts	Implementation-defined
Acknowledgement Batching	100ms	Maximum delay for batching ACKs	SHOULD be configurable

4.8 Session Termination

1. Either party MAY send a terminating segment (empty segment with termination flag set)
2. Upon receiving a terminating segment:
   - Unreliable sockets SHOULD close immediately
   - Reliable sockets MUST complete pending acknowledgements before closing
3. No data frames MUST be sent after a terminating segment

4.9 Example Message Exchanges

All numeric values in the examples below are shown in their logical representation. Frame IDs and other multi-byte integers are encoded in big-endian format on the wire.

4.9.1 Simple Frame Transmission (Unreliable Mode)

Sending a 300-byte frame with MTU=256 (246 bytes available per segment after 10-byte overhead):

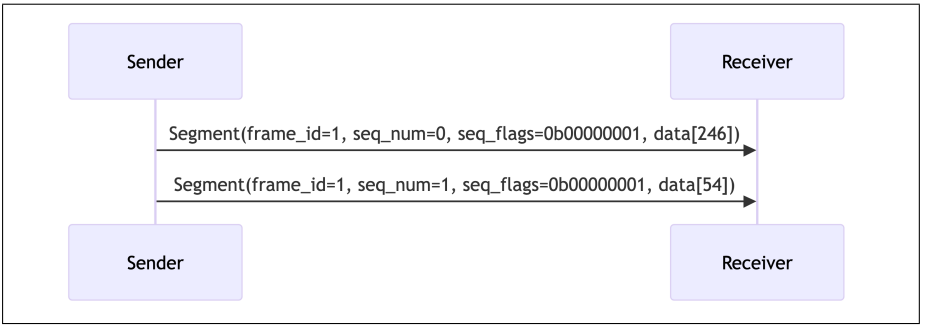


Figure 14: Mermaid Diagram 7

4.9.2 Frame with Retransmission (Reliable Mode)

Reliable transmission where the middle segment is lost and retransmitted:

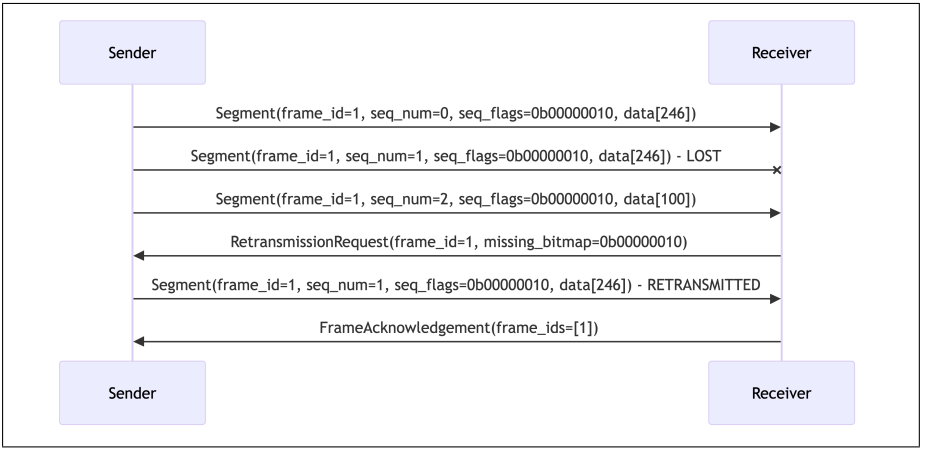


Figure 15: Mermaid Diagram 8

4.9.3 Multiple Frame Acknowledgement (Reliable Mode)

Efficiently acknowledging multiple received frames in a batch:

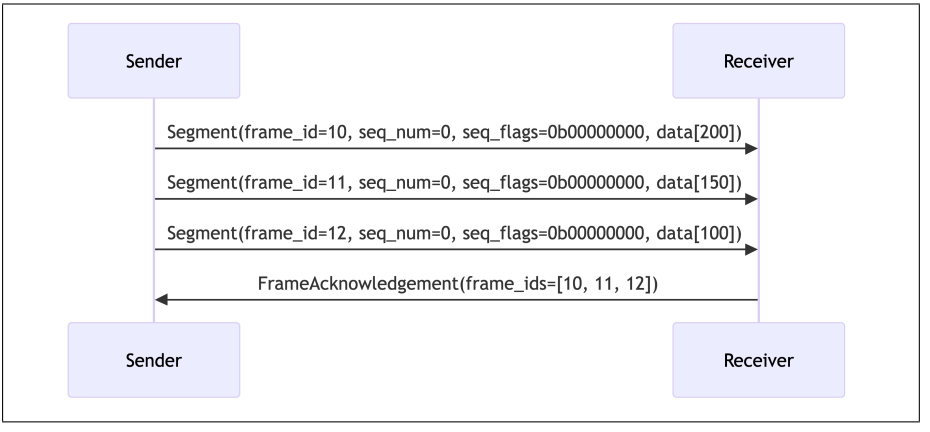


Figure 16: Mermaid Diagram 9

4.9.4 Session Termination (Reliable Mode)

Graceful session termination with acknowledgement:

4.9.5 Session Termination (Unreliable Mode)

Immediate session termination without acknowledgement:

5. Design Considerations

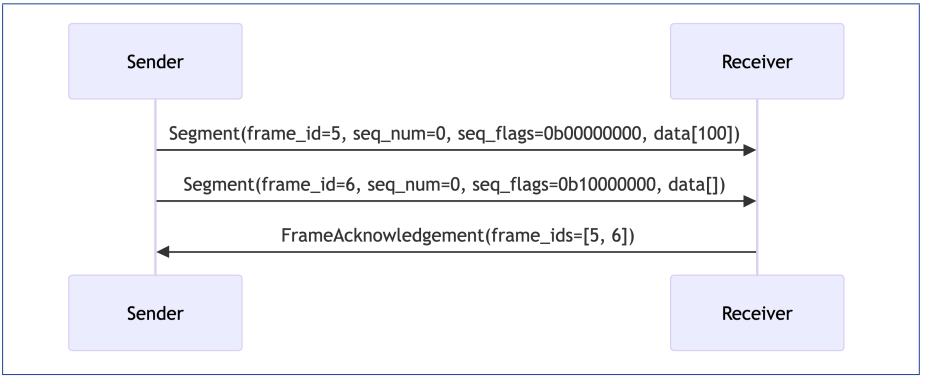


Figure 17: Mermaid Diagram 10

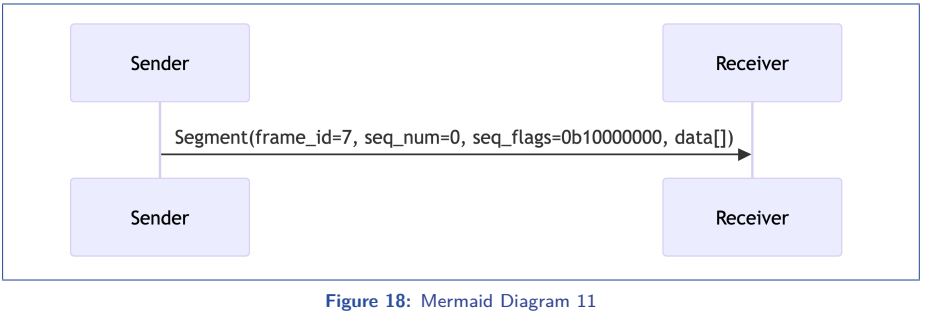


Figure 18: Mermaid Diagram 11

5.1 Maximum Segments Limitation

The protocol limits frames to 64 segments due to the 6-bit sequence length field. This provides a good balance between:

- Frame size flexibility (up to 64 × MTU)
- Protocol overhead (1 byte for sequence information)
- Implementation complexity (simple bitmap for retransmissions)

5.2 Frame ID Space

The 32-bit Frame ID space allows for over 4 billion frames per session. Frame IDs MUST be monotonically increasing to enable:

- Duplicate detection
- Out-of-order delivery handling
- Simple state management

The Session MUST terminate when Frame ID of 0 is encountered by the receiving side, indicating an overflow.

5.3 Retransmission Request Design

Limiting retransmission requests to the first 8 segments per frame:

- Keeps message format simple (1-byte bitmap)
- Covers the common case (most frames have ≤8 segments)
- Frames requiring >8 segments can use smaller frame sizes

5.4 Protocol Overhead

- Minimum overhead per segment: 10 bytes (4 header + 6 segment header)
- Maximum protocol efficiency: (C - 10) / C

- For C = 1024: ~99% efficiency
- For C = 256: ~96% efficiency

6. Compatibility

6.1 Version Compatibility

- Version 1 is the initial Session Data protocol version
- Future versions MUST use different version numbers
- Implementations MUST reject messages with unknown versions
- Version negotiation is out of scope for this specification

6.2 Transport Requirements

- Requires bidirectional communication channel
- No assumptions about ordering or reliability

7. Security Considerations

7.1 Protocol Security

- The protocol provides NO encryption or authentication
- Security MUST be provided by the underlying transport
- Frame IDs are predictable and MUST NOT be used for security

8. Future Work

- Enhanced acknowledgement schemes for better efficiency
- Forward error correction for high-loss environments

9. Implementation Notes

9.1 Testing Recommendations

- Test with various MTU sizes (256, 512, 1024, 1500, 9000)
- Simulate packet loss, reordering, and duplication
- Verify termination handling under all conditions
- Stress test with maximum frame sizes and counts

10. References

[01] Postel, J. (1981). Transmission Control Protocol. IETF RFC 793.

[02] Bormann, C. & Hoffman, P. (2013). Concise Binary Object Representation (CBOR). IETF RFC 7049.

[03] Postel, J. (1980). User Datagram Protocol. IETF RFC 768.

9 RFC-0009: Session Start Protocol

- RFC Number: 0009
- Title: Session Start Protocol
- Status: Implementation
- Author(s): Tino Breddin (@tolbrino), Lukas Pohanka (@NumberFour8)
- Created: 2025-08-20
- Updated: 2025-09-05
- Version: v0.1.0 (Draft)
- Supersedes: none
- Related Links: RFC-0002, RFC-0004, RFC-0008, RFC-0011

1. Abstract

This RFC specifies the HOPR Session Start Protocol, a handshake protocol for establishing sessions between peers in the HOPR mixnet. The protocol manages session establishment, lifecycle, and capability negotiation using HOPR packets as transport. It provides a standardized method for initiating communication sessions, exchanging session parameters, and maintaining session state through keep-alive mechanisms.

2. Motivation

The HOPR mixnet requires a standardized mechanism for establishing communication sessions between nodes. While the Session Data Protocol (see RFC-0008) handles data transmission, there needs to be a separate protocol for:

- Establishing sessions with capability negotiation
- Exchanging session identifiers and targets
- Managing session lifecycle and state
- Providing error handling for session establishment failures

- Maintaining session liveness through keep-alive mechanisms

The Session Start Protocol fills this gap by providing a lightweight, transport-agnostic handshake mechanism specifically designed for the HOPR ecosystem.

3. Terminology

Terms defined in RFC-0002 are used. Additionally, this document defines the following session start protocol-specific terms:

- Challenge: A 64-bit random value used to correlate requests and responses in the handshake process. Challenge values are interpreted as big-endian unsigned integers.
- Session Target: The destination or purpose of a session, typically an address or service identifier, encoded in CBOR format.
- Session Capabilities: A bitmap of session features and options negotiated during session establishment.
- Session ID: A unique identifier assigned by the responder to identify the established session.
- Entry Node: The node that initiates a session establishment request.
- Exit Node: The node that receives and responds to a session establishment request.
- CBOR (Concise Binary Object Representation): A binary data serialization format defined in RFC 7049 [01], used for encoding session identifiers and targets.

4. Specification

4.1 Protocol Overview

The Session Start Protocol operates at version 2 and consists of four message types that manage the complete lifecycle of session establishment:

1. StartSession: Initiates a new session
2. SessionEstablished: Confirms session establishment
3. SessionError: Reports session establishment failure
4. KeepAlive: Maintains session liveness

The protocol uses HOPR packets as the underlying transport mechanism and supports both successful and failed session establishment scenarios. All multi-byte integer fields use network byte order (big-endian) encoding to ensure consistent interpretation across different architectures.

4.2 Message Format

All Session Start Protocol messages follow a common structure:

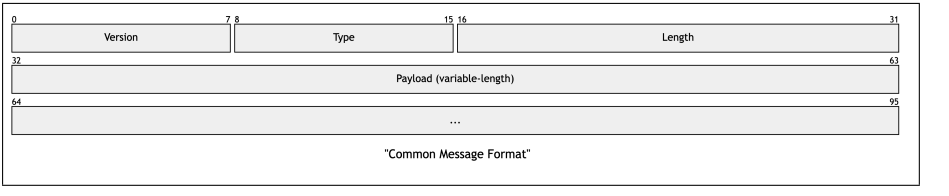


Figure 19: Mermaid Diagram 1

Field	Size	Description	Value
Version	1 byte	Protocol version	MUST be `0x02` for version 2
Type	1 byte	Message type discriminant	See Message Types table below
Length	2 bytes	Payload length in bytes	0-65535
Payload	Variable	Message-specific data	CBOR-encoded where applicable

4.2.1 Message Types

Type Code	Name	Description
0x00	StartSession	Initiates a new session
0x01	SessionEstablished	Confirms session establishment
0x02	SessionError	Reports session establishment failure
0x03	KeepAlive	Maintains session liveness

4.2.2 Byte Order

All multi-byte integer fields and values in the Session Start Protocol MUST be encoded and interpreted in network byte order (big-endian). This applies to:

Protocol Message Fields:

- Length field (2 bytes) in the common message format
- Challenge field (8 bytes) in StartSession, SessionEstablished, and SessionError messages
- Additional Data field (4 bytes) in StartSession messages
- Additional Data field (8 bytes) in KeepAlive messages
- Session ID suffix (64-bit) in HOPR Session ID format
- Any future numeric fields added to the protocol

This requirement ensures consistent interpretation across different architectures and prevents interoperability issues between implementations.

4.3 StartSession Message

Initiates a new session with the remote peer.

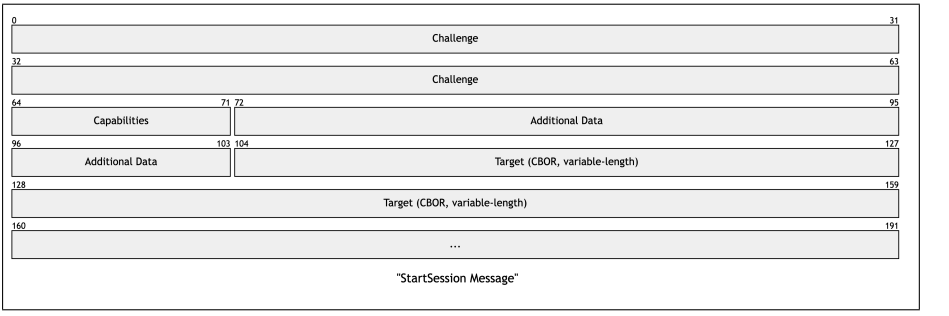


Figure 20: Mermaid Diagram 2

Field	Size	Description	Notes
Challenge	8 bytes	Random challenge for correlating responses	MUST use CSPRNG
Capabilities	1 byte	Session capabilities bitmap	See Capability Flags table
Additional Data	4 bytes	Capability-dependent options	Set to `0x00000000` to ignore
Target	Variable	CBOR-encoded session target	e.g., “127.0.0.1:1234”

4.3.1 Capability Flags

Bit	Flag Name	Description
0	Reserved	Reserved for future use
1	Reserved	Reserved for future use
2	Reserved	Reserved for future use
3	Reserved	Reserved for future use
4	Reserved	Reserved for future use
5	Reserved	Reserved for future use
6	Reserved	Reserved for future use
7	Reserved	Reserved for future use

Bit	Flag Name	Description

4.4 SessionEstablished Message

Confirms successful session establishment.

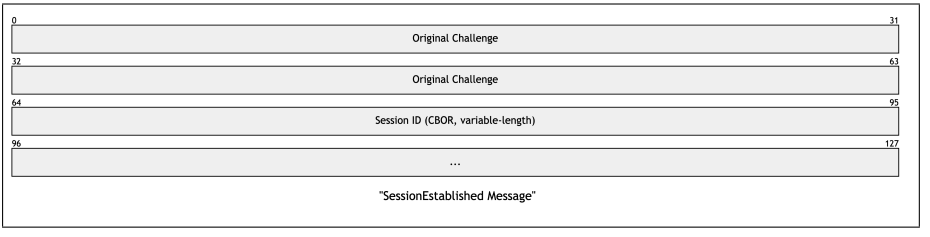


Figure 21: Mermaid Diagram 3

Field	Size	Description	Notes
Original Challenge	8 bytes	Challenge from StartSession message	MUST match original challenge
Session ID	Variable	CBOR-encoded session identifier	Assigned by responder, MUST be unique

4.5 SessionError Message

Reports session establishment failure.

Field	Size	Description	Notes
Challenge	8 bytes	Challenge from StartSession message	MUST match original challenge
Reason	1 byte	Error reason code	See Error Codes table below

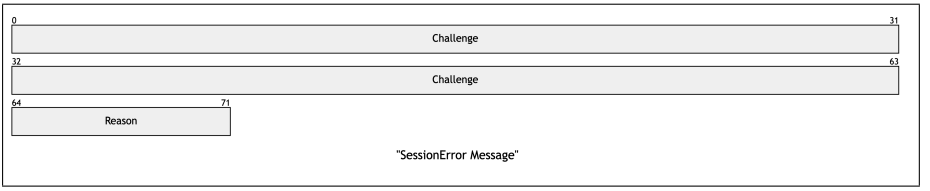


Figure 22: Mermaid Diagram 4

4.5.1 Error Codes

Code	Name	Description	Recommended Action
0x00	Unknown Error	Unspecified error condition	Retry with different parameters or node
0x01	No Slots Available	Exit node has no available session slots	Retry later or try different node
0x02	Busy	Exit node is temporarily busy	Retry after brief delay

4.6 KeepAlive Message

Maintains session liveness.

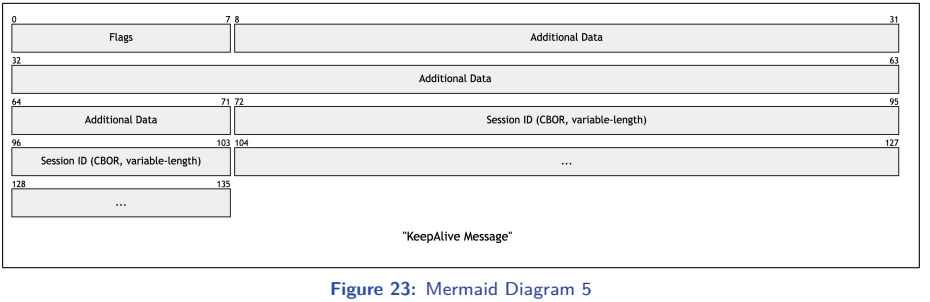


Figure 23: Mermaid Diagram 5

Field	Size	Description	Notes
Flags	1 byte	Reserved for future use	MUST be 0x00
Additional Data	8 bytes	Flag-dependent options	Set to 0x0000000000000000 to ignore

Field	Size	Description	Notes
Session ID	Variable	CBOR-encoded session identifier	MUST match established session

4.7 Protocol Flow

4.8 Protocol Constants

Constant	Value	Description
Protocol Version	`0x02`	Current protocol version
Default Timeout	30 seconds	Default session establishment timeout
Challenge Size	8 bytes	Fixed size for challenge field
Max Payload Length	65535 bytes	Maximum message payload size

4.9 Protocol Rules

Rule	Requirement Level	Description
Challenge Generation	MUST	Challenge values MUST be randomly generated using CSPRNG
Session ID Uniqueness	MUST	Session IDs MUST be unique per responder
Byte Order	MUST	All multi-byte integer fields MUST use network byte order
CBOR Encoding	MUST	Targets and Session IDs use CBOR encoding [01]

Rule	Requirement Level	Description
Payload Limits	MUST	Messages MUST fit within HOPR packet payload limits
Keep-Alive Frequency	SHOULD	KeepAlive messages SHOULD be sent periodically
Error Handling	MUST	Implementations MUST handle all defined error conditions gracefully
Timeout Configuration	SHOULD	Session establishment timeouts SHOULD be configurable (default: 30s)

4.10 Example Message Exchanges

4.10.1 Successful Session Establishment

Complete session establishment with immediate keep-alive:

4.10.2 Session Establishment with Error

Session establishment failing due to resource exhaustion:

4.10.3 Session Establishment Timeout

Session establishment with no response from Exit Node:

4.10.4 Long-Running Session with Periodic Keep-Alives

Maintaining an established session over time:

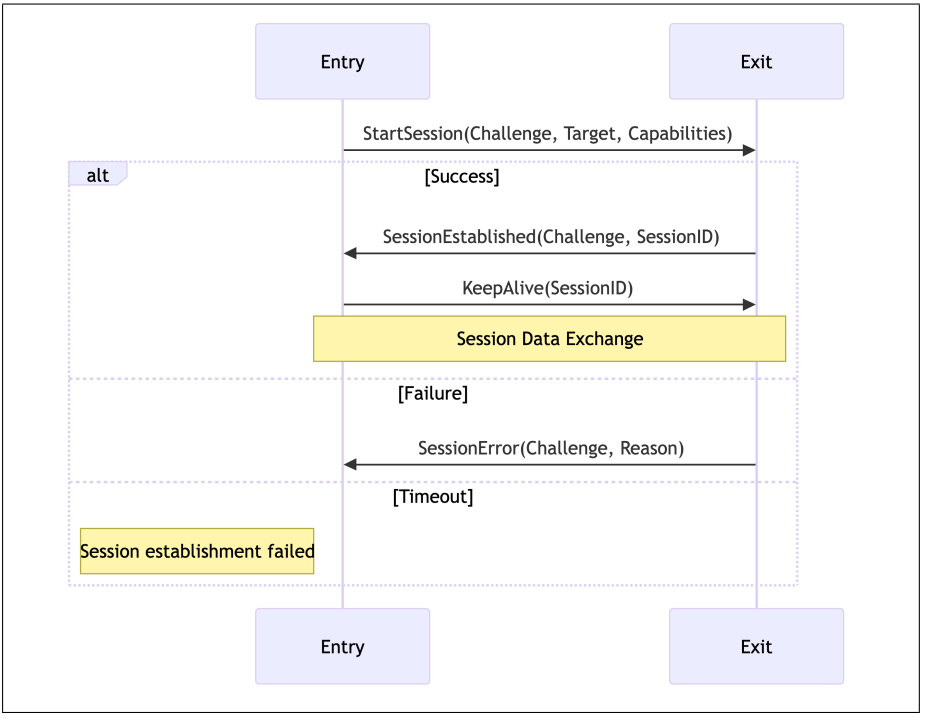


Figure 24: Mermaid Diagram 6

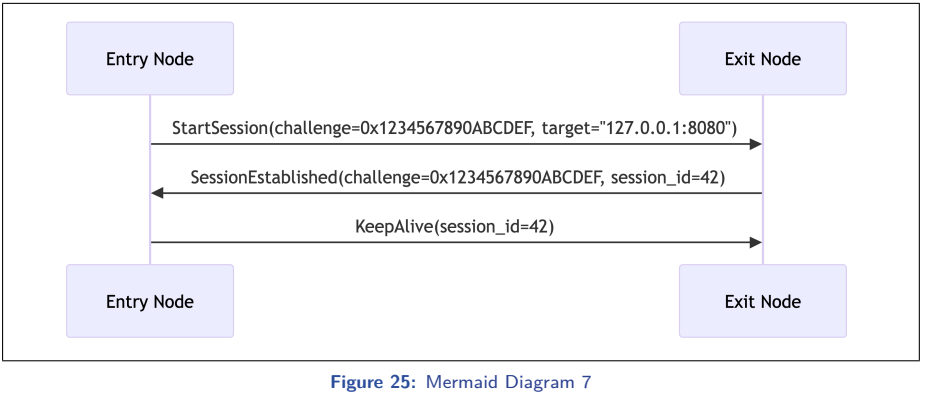


Figure 25: Mermaid Diagram 7

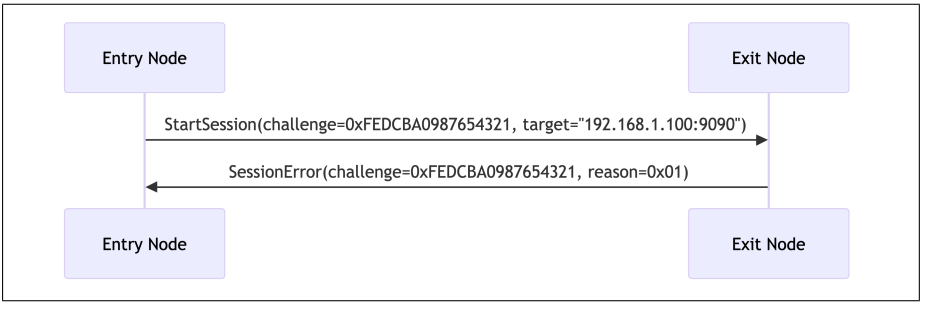


Figure 26: Mermaid Diagram 8

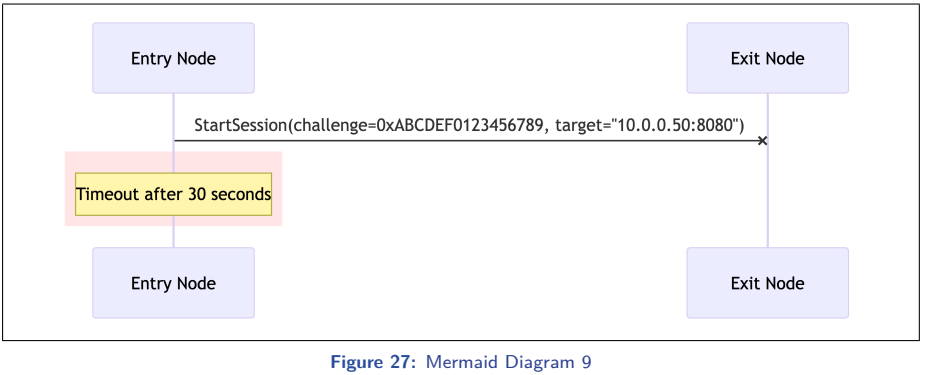


Figure 27: Mermaid Diagram 9

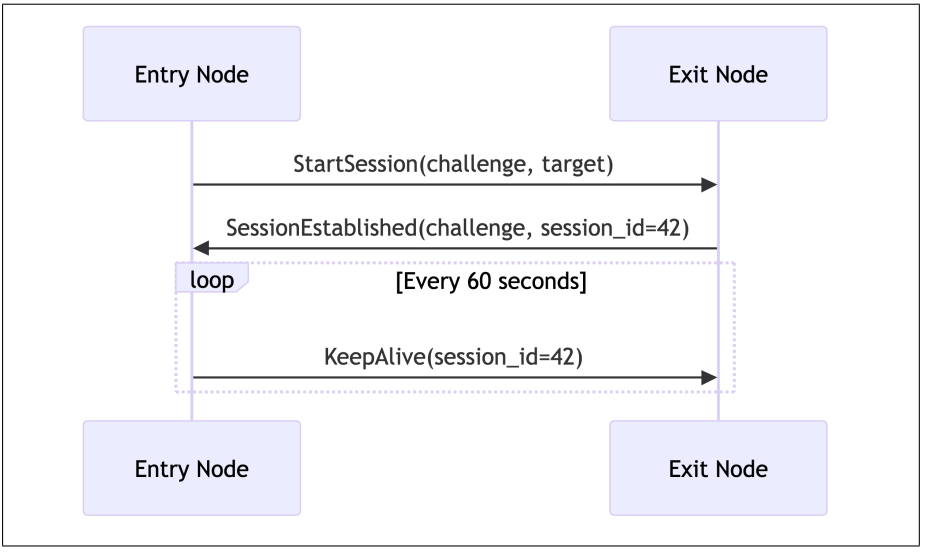


Figure 28: Mermaid Diagram 10

5. Design Considerations

5.1 CBOR Encoding

The use of CBOR (Concise Binary Object Representation) for Session IDs and Targets provides:

- Flexible data types without fixed-size constraints
- Compact binary encoding
- Language-agnostic serialization
- Support for complex session identifiers

5.2 Challenge-Response Design

The 64-bit challenge provides:

- Correlation between requests and responses
- Protection against replay attacks (when combined with transport security)
- Simple state tracking for pending sessions

5.3 Capability Negotiation

The single-byte capability field allows:

- Up to 8 independent capability flags
- Future protocol extensions
- Backward compatibility through ignored bits

5.4 Transport Independence

The Session Start protocol is transport-agnostic:

- Works over any packet-based transport
- Designed for HOPR packets but not limited to them
- No assumptions about ordering or reliability

5.5 Error Handling

The protocol provides structured error reporting:

- Specific error codes for common failure scenarios
- Challenge correlation for error attribution
- Graceful handling of resource exhaustion

6. Compatibility

6.1 Version Compatibility

- Version 2 is the initial protocol version
- Future versions MUST use different version numbers
- Implementations MUST reject messages with unknown versions
- Version negotiation is out of scope for this specification

6.2 Transport Requirements

- Requires bidirectional communication channel
- No assumptions about ordering or reliability
- Compatible with any transport providing packet delivery
- Designed for HOPR mixnet but not limited to it

6.3 Integration with HOPR Session Data Protocol

- HOPR Session Start Protocol establishes sessions for use by HOPR Session Data Protocol (see RFC-0008)
- Session IDs from this protocol are used to identify data sessions
- Protocol operates independently but provides foundation for data exchange

7. Security Considerations

7.1 Protocol Security

- The protocol provides NO encryption or authentication
- Security MUST be provided by the underlying transport
- Session IDs SHOULD be unpredictable to prevent session hijacking
- Challenges MUST use cryptographically secure random number generation

7.2 Attack Vectors

- Replay attacks possible without additional timestamp or nonce mechanisms
- Man-in-the-middle attacks not prevented by protocol alone
- Session targets may expose service information if not encrypted at transport
- Resource exhaustion through excessive session establishment requests

7.3 Mitigation Strategies

- Use transport-level security (e.g., HOPR packet encryption)
- Implement rate limiting for session establishment requests
- Use unpredictable session identifiers
- Consider implementing session timeout mechanisms

8. Future Work

- Session parameter renegotiation mechanisms
- Performance optimizations for high-frequency session establishment

9. Implementation Notes

9.1 Testing Recommendations

- Test with various session target formats
- Simulate network failures and timeouts
- Verify challenge uniqueness and correlation
- Test capability negotiation edge cases
- Validate CBOR encoding/decoding correctness

10. References

[01] Bormann, C. & Hoffman, P. (2013). Concise Binary Object Representation (CBOR). IETF RFC 7049.

11. Related Links

- RFC-0011 Application Layer protocol

12. Appendix 1

Within HOPR protocol a Session is identified uniquely via HOPR Session ID, this consists of a 10-byte pseudorandom bytes as prefix and 64-bit unsigned integer as suffix. The 64-bit suffix is encoded and interpreted as a big-endian unsigned integer.

In human readable format, a HOPR Session ID has the following syntax:

`0xabcdefabcdefabcdefab:123456`

The prefix represents a fixed pseudonym prefix of in the HOPR Packet protocol (as in RFC-0004). The suffix represents an application tag that identifies Sessions within the reserved range in the Application protocol RFC-0011.

10 RFC-0010: Automatic path discovery

- RFC Number: 0010
- Title: Automatic path discovery
- Status: Raw
- Author(s): @Teebor-Choka
- Created: 2025-02-25
- Updated: 2025-07-21
- Version: v0.0.1 (Raw)
- Supersedes: none
- Related Links: RFC-0002, RFC-0004, RFC-0005, RFC-0008, RFC-0009

1. Abstract

This RFC provides a description of an automatic path discovery mechanism necessary for the HOPR protocol to be usable inside a dynamic ad-hoc peer-to-peer network. The outlined solutions aim to allow the HOPR protocol message sender to remain anonymous, while ensuring optimal message delivery through the network by actively probing various network nodes with the goal of establishing compliance with the HOPR protocol specified functionality and non-adversarial behavior.

2. Motivation

Effective end-to-end communication over the HOPR protocol requires the communication sender to select viable paths across the network:

- From sender to destination for unidirectional communication
- Additionally, from destination to sender using the Return Path mechanism for bidirectional communication

The HOPR protocol does not define communication flow control, as this is handled by upper protocol layers. This design decision places responsibility of every network element to keep track of peer and network status to allow establishing stable propagation paths with consistent transport link properties.

In the mixnet architecture, both forward and return paths MUST be constructed by the sender to preserve anonymity . Consequently, the sender MUST maintain an accurate and current view of the network topology to create effective forward and return path pools.

Relayers and destinations must also discover the network in order to make sure the incentivized layer and network transport are aligned.

3. Terminology

Terms defined in RFC-0002 are used.

4. Specification

4.1 Overview

This specification defines multiple complementary graph search algorithms for topology discovery. Implementations MUST support both algorithms and employ them in concert, as complete topology discovery becomes computationally prohibitive as network size increases.

4.2 Network probing

The network discovery algorithms SHOULD make the following assumptions about the network:

1. the network topology is not static

    - the network topology can change as individual nodes peer preferences or open/close channels
    - for peers that require a relay the disappearance of the relay can cause topology reconfiguration
2. every other node can be unreliable
    - rooted deeply in the physical network infrastructure performance
3. every other node can be malicious
    - any behavior resembling malicious behavior should be considered malicious and appropriately flagged

Given these assumptions, the network probing algorithms for topology discovery SHOULD use multiple complementary mechanisms: a breadth-first and a depth-first algorithm.

Initially, implementers SHALL perform a general network discovery using primarily the breadth-first approach.

Once a statistically sufficient topology is identified to support path randomization, the depth-first approach SHOULD be employed to probe specific topology paths of interest (e.g., exit node peers).

The advantage of the depth-first approach is that its results can be combined with the breadth-first approach to identify potentially unreliable or malicious peers more efficiently, while allowing focus on specific peers in the path as static anchors (for QoS, exit behavior functionality, etc.).

The network topology is an oriented graph structure consisting of nodes performing the probing data relay functionality. Each edge corresponds to a combination of properties defined by the physical transport and the HOPR protocol that MUST be present:

1. Existence of a HOPR staking channel RFC-0005 from the node in the path in the direction of the relayer
2. Presence of a physical transport connection allowing data transfer

While property 1 is known from the incentive mechanism RFC-0007, property 2 MUST be discovered on the physical network and is subject to network probing. The only exception to property 1 in the HOPR protocol is the last hop (i.e., the

last relayer to the destination), where a staking channel is not required for data delivery. The network probing mechanism, abstracting transport interactions completely, consists of 3 components:

1. Path generating probing algorithm
2. Evaluation mechanism
3. Retention and slashing mechanism

4.2.1 Path generating probing algorithm

The primary responsibility of the path generating component is to apply different algorithms to prepare pre-generated paths that would offer insights in algorithm selected sections of the network with the goal of collecting path viability information.

The algorithm MUST use a loopback form of communication to conceal the nature of the probing traffic from relayers. Loopback MAY be realized via the Session protocol RFC-0008 or via an equivalent ephemeral mechanism; Sessions are OPTIONAL. In this approach, the probing node functions as both sender and receiver of the probing traffic, effectively designating each node in the path as a probed relayer and each edge between consecutive relayers as a probed connection. While this approach does not guarantee extraction of all relevant information from a single probing attempt, when combined with results from multiple probing attempts, it enables construction of a comprehensive view of network topology and dynamics.

A combination of breadth-first and depth-first algorithms SHALL be employed to ensure the probing process neither anneals too slowly to a usable network topology nor focuses exclusively on small sub-topologies due to network size constraints. Loopback probing methods with respect to the sender:

1. Immediate 0-hop: Observe only whether acknowledgment was received from the counterparty and measure response latency, using indistinguishable payloads (“junk” data indistinguishable from application data via padding and AEAD) with acknowledgements produced by the destination and authenticated before acceptance

2. 1-hop to self: First-order checks of immediate peer connections - functionally equivalent to option 1 but executed in a manner that conceals probing activity
3. 2-hop to self: Checks second-order communication paths, MAY replace some 3-hop paths to reduce total probing paths
4. 3-hop to self: Full path bidirectional channel probing for 1-hop connections

Algorithm:

- Discovery algorithm SHALL operate in complementary modes: breadth-first and depth-first
- Basic operations:
    1. Discover immediate peers
    2. Generate paths for n-hop connections (referential probing with low frequency)
    3. for sessions RFC-0008, prepopulate the cache from sufficiently recent historical knowledge of successful paths
    4. perform higher frequency probing checks

4.2.1.1 Breadth-first algorithm (BFA)

Breadth-First Search (BFS) is a graph traversal algorithm used to systematically explore nodes and edges in a graph. It MUST start at the sender and explores the neighboring nodes at the current depth level before moving on to nodes at the next depth level.

BFA SHOULD primarily be used for the initial network topology discovery with the goal of identifying a statistically significant minimum number of peers with the desired QoS and connectability properties.

This algorithm SHOULD be primarily implemented in terms of the 1-hop to self.

Given a network topology around the node A (Fig. 1):

Fig. 1: Network topology for BFA inspired network probing

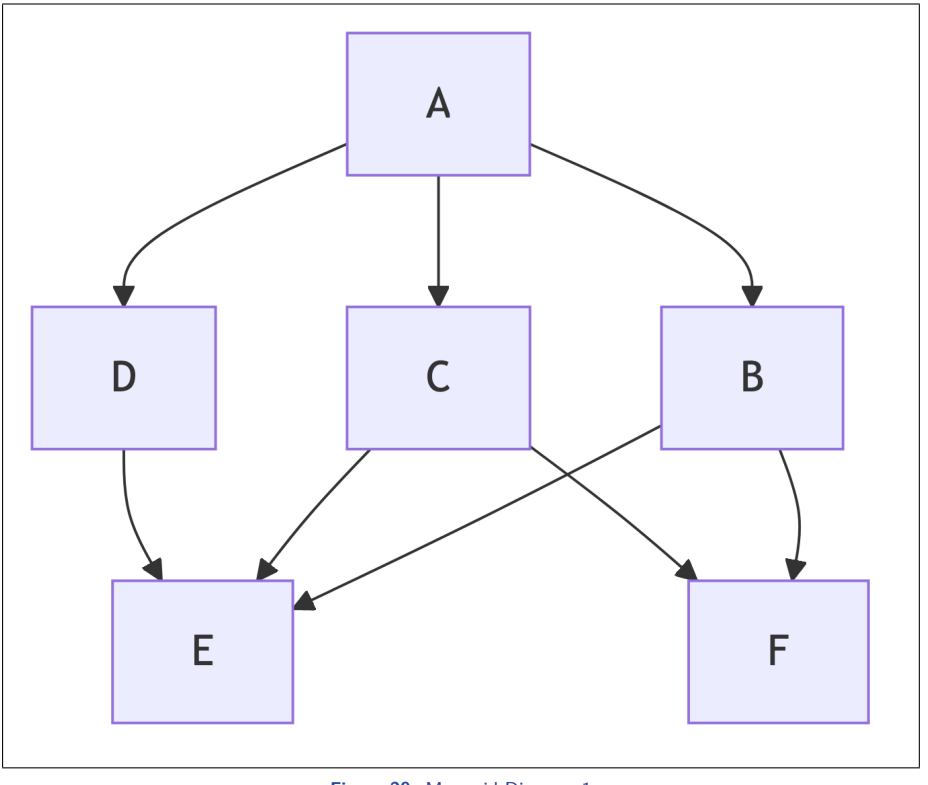


Figure 29: Mermaid Diagram 1

The probing traffic from node A would follow the BFA pattern of establishing the telemetry from the immediate vicinity of A using a 1-hop probing traffic:

```
A -> B -> A
A -> C -> A
A -> D -> A
```

Once the immediate vicinity is probed, a larger share of the probing traffic SHOULD use the depth-first algorithm phasing the BFA into smaller proportion.

4.2.1.2 Depth-first algorithm (DFA)

Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It MUST start at the current node to explore each branch of the graph deeply before moving to another branch.

DFS is particularly useful for solving problems related to maze exploration and pathfinding.

This algorithm SHOULD be primarily implemented in terms of the n-hop to self, where n>1 and n<MAX_HOPR_SUPPORTED_PATH_LENGTH (a network parameter defined in RFC-0004), with each edge probed as soon as feasible, but at the same time not at the expense of other edges in the topology. n SHOULD be chosen randomly, but MUST conform with the minimum requirement for edge traversal.

Given a network topology around the node A (Fig. 2):

Fig. 2: Network topology for DFA inspired network probing

The probing traffic from node A would follow the DFA pattern of establishing the telemetry to the furthest interesting point in the network using an n-hop probing traffic with n generated randomly:

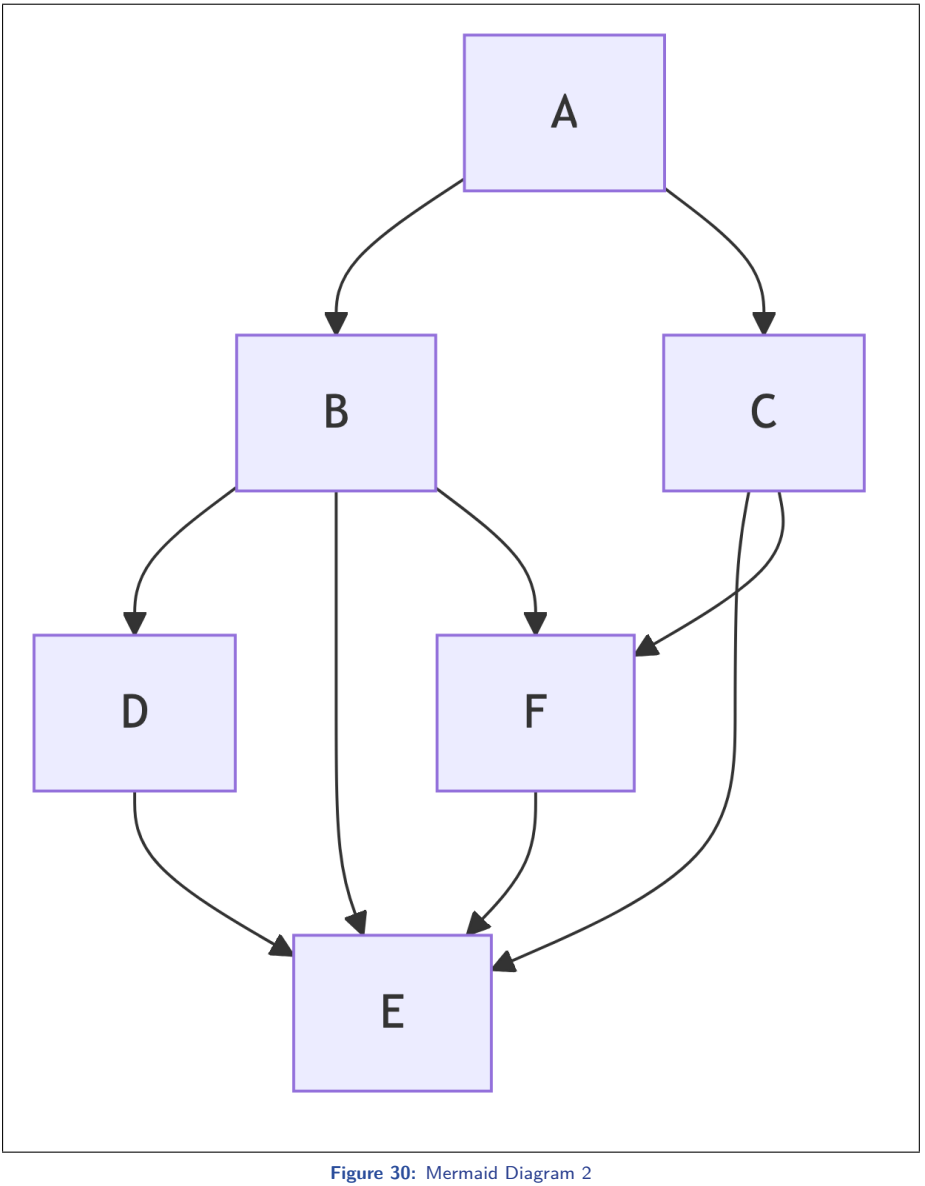


Figure 30: Mermaid Diagram 2

```
A -> B -> F -> A
A -> C -> F -> E -> A
A -> B -> D -> A
```

4.2.1.3 BFA and DFA interactions

Average values calculated over the differences of various observations can be used to establish individual per node properties. From the previous example, given multiple averaged telemetry values over the path it is possible to establish ensemble information about the topology.

Example: With average path latencies observed over these paths as:

```
A -> B -> A = 421ms
A -> B -> F -> A = 545ms
```

It is possible to establish the average latency of introducing the node `F` into the path as `A->B->F->A` - `A->B->A`= 545 - 421 = 124ms.

Assuming artificial mixer delays introducing additional anonymity, repeated observations of this value averaged over longer windows would provide an average expected latency introduced by element `F`.

4.2.2 Evaluation mechanism

Evaluation mechanism SHOULD have short-term memory and equally reward and penalize probe success and failures.

4.2.3 Retention and slashing mechanism

Nodes MAY implement a slashing mechanism based on failed probes to avoid using the relay nodes in non-probing communication and avoid dropped messages.

4.2.4 Throughput considerations

Paths SHOULD be used by the discovery mechanism in a way that would allow sustained throughput, i.e. the maximum achievable packet rate:

- Calculate load balancing over paths based on the min stake on the path
- Actual throughput as measured by the real traffic

4.3. Telemetry

Refers to data and metadata collected by the probing mechanism about the traversed transport path.

4.3.1 Next-hop telemetry

Supplemental per path telemetry (PPT) MUST be used as a source of information for a possibly channel opening and closing strategy responsible for reorganizing the first hop connections from the current node.

The PPT SHOULD provide the basic evaluation of the transport channel in the absence of an open onchain channel and MUST provide at least these transport channel observations using 0-hop as specified in the HOPR protocol RFC-0004:

1. latency

    - duration between a send-message and a corresponding acknowledgement
2. packet drop
    - track ratio of missing/all expected acknowledgements for each message on the channel

The PPT MAY be utilized as an information source by other mechanisms, e.g. the channel manipulation strategy optimizing the outgoing network topology.

4.3.2 Non-probing telemetry

The non-probing telemetry MAY track the next-hop telemetry targets with the goal of adding more relevant channel information for the nearest 0-hop.

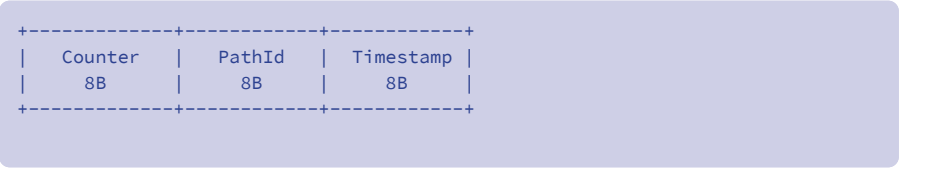
Each outgoing message should be tracked for the same set of telemetry as the PPT on the per message basis.

4.3.3 Probing telemetry

Telemetry data pertains to the content of the probing message sent over the network. All multi-byte integer fields MUST be transmitted in network byte order (big endian).

The content of the probing message:

- Iterating counter to verify the mixing property over a path
    - an iterated `uint64` equivalent value
- Path identification for attribution
    - a unique value identifying a single specific path in the graph using a `uint64` equivalent value
- Timestamp of packet creation for channel latency observations
    - formatted as an 8-byte (64-bit) `UNIXtimeinnanoseconds`



The total packet size is 24 bytes.

4.4 Component placement

The network probing functionality, with the exception of the PPT mechanism, MUST be implemented using HOPR loopback communication.

Implementation requirements:

- The concept of channel graph quality based on network observations SHALL be removed
  - Only the onchain channel information SHALL be retained
- Implementations MUST provide processes to:
  - Generate a low-rate continuous stream of network path probes
  - Generate session-specific paths for session path selection obfuscation RFC-0008
- A new path graph system SHALL be derived from these processes
- Paths SHALL be cached for a configurable minimum time window
- Session metrics SHALL incorporate:
  - Session-level performance metrics
  - Session-specific path probing data
  - Session-derived cover traffic for exploratory network traversal

5. Design considerations

Each sender SHOULD:

- Be able to identify a sufficiently large number of network nodes to ensure privacy through path pool diversity

- Be capable of detecting unstable, malicious, or adversarial nodes
- Be able to establish basic propagation metrics for Quality of Service (QoS) estimation

Given the capabilities described above, the message sender SHOULD be able to construct a functional representation of the network topology, state, and constraints, enabling optimal selection and exclusion of message propagation paths.

The multihop probing traffic and measurement packets MUST be indistinguishable from ordinary traffic to ensure accurate recording of network node propagation characteristics. Due to the dynamic nature of decentralized peer-to-peer networks, the message sender SHOULD employ adaptive mechanisms for establishing and maintaining topological awareness.

For both unidirectional and bidirectional communication to adapt to changing network conditions, the sender MUST actively probe the network in a continuous manner.

The measurement traffic itself SHOULD adhere to economic feasibility constraints, i.e., it SHOULD be proportional to actual message traffic and MAY be incorporated as part of the Cover Traffic (CT) RFC-0008.

Any measurements obtained from the probing traffic SHOULD be node-specific and MUST NOT be subject to data or topology exchange with other nodes.

The collected telemetry for measured paths:

- MUST contain path passability data
  - Path traversability by single or multiple messages
- MAY include additional information
  - Telemetry transferred as message content

By designing probing traffic to be indistinguishable from actual message propagation in the mixnet, direct verification of immediate peer properties becomes infeasible. For this purpose, a separate mechanism not described in this document SHOULD exist.

The nearest one-hop probing mechanism MAY NOT comply with the anonymity requirement, since it:

1. mimics the 0-hop session RFC-0008 which does not fully benefit from relaying mechanisms
2. could be used as a first layer for relayers to discover viable candidates for future channel openings

The network probing mechanism SHALL utilize graph-based algorithms to efficiently discover and maintain network topology information.

6. Compatibility

This feature affects only a single node in the network and MAY be modified without impacting overall network operation.

The network probing mechanism MAY be compatible with the loopback session mechanism

7. Security Considerations

The probing traffic consumes both physical resources and value at various levels of the HOPR protocol stack.

Security considerations related to resource utilization include:

1. In highly volatile networks, adversarial behavior may cause excessive resource expenditure, potentially enabling resource depletion attacks.
2. The PPT mechanism MAY serve as an attack vector for Denial of Service (DoS) attempts.
3. Nodes MAY implement any security risk mitigation strategy

8. Drawbacks

The network probing mechanism has several inherent limitations:

1. Probing activity consumes resources; implementations MUST carefully balance probing and data transmission activities to maintain reasonable resource utilization ratios.
2. Complete real-time probing of large networks is computationally prohibitive; algorithms SHOULD operate within bounded subnetworks where they can provide reasonable network visibility guarantees.
3. Prior knowledge of target nodes is advantageous to minimize initialization time before establishing a sufficient network view for informed path selection.

9. Alternatives

No alternative mechanisms exist that simultaneously preserve anonymity, maintain trustless properties, and consolidate probing control under the communication source.

10. Unresolved Questions

None

11. Future Work

Future development SHOULD focus on:

1. Improving the ability to collect additional network metrics primarily by extending the data payload transmitted along the loopback path

2. Developing new path generating strategies allowing statistical inference of information from the path section overlaps
3. Improve metric evaluation mechanism
4. Add proper slashing mechanism with equation based logic

12. References

None.

11 RFC-0011 Application Layer protocol

- RFC Number: 0011
- Title: Application Layer protocol
- Status: Draft
- Author(s): Lukas Pohanka (@NumberFour8)
- Created: 2025-08-22
- Updated: 2025-08-22
- Version: v0.1.0 (Draft)
- Supersedes: none
- Related Links: RFC-0002, RFC-0004, RFC-0008, RFC-0009, RFC-0010

1. Abstract

This RFC describes the Application layer protocol used in the HOPR project. Typically, this protocol is used in between the HOPR Packet protocol RFC-0004 and some higher-level protocol, such as the Session protocol RFC-0008 or Start protocol RFC-0009. The goal of this protocol is for a HOPR node to make distinction between different protocol running on top of the HOPR packet protocol.

It can be seen similar to how standard TCP or UDP protocols distinguishes between applications using port numbers.

2. Motivation

The HOPR network supports multiple upper layer protocols that serve different purposes. Without a standardized method to distinguish between these protocols, nodes would be unable to properly route and handle packets intended for specific applications. The Application layer protocol solves this by providing a lightweight tagging mechanism similar to port numbers in TCP/UDP.

3. Terminology

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [01] when, and only when, they appear in all capitals, as shown here.

Terms defined in RFC-0002 might be also used.

4. Introduction

The HOPR network can host multitude of upper layer protocols, that serve different purposes. Some of those are described in other RFCs, such as RFC-0008, RFC-0009 or RFC-0010. The Application layer protocol described in this RFC creates a thin layer between the HOPR Packet protocol from RFC-0004 and these upper layer protocols.

The Application layer protocol primarily serves two purposes:

1. node should be able to distinguish between upper protocols and dispatch their packets the respective protocol interpreters
2. create an inter-protocol communication link for signals between the HOPR Packet protocol and the upper layer protocol

5. Specification

The Application layer protocol acts as a wrapper to arbitrary upper layer `data` and adds a `Tag` that determineds the type of the upper-layer protocol:

```
ApplicationData \{
    tag: Tag,
    data: [u8; <length>]
    flags: u8
\}
```

The `Tag` itself MUST be represented by 64 bits and the 3 upper most significant bits MUST be always set to 0 in the current version. The remaining 61 bits represent a unique identifier of the upper layer protocol.

The `Tag` range SHOULD be split as follows:

- `0x0000000000000000` identifies the Probing protocol (see RFC-0010).
- `0x0000000000000001` identifies the Start protocol (see RFC-0009).
- `0x0000000000000002` - `0x000000000000000d` identifies range for user protocols
- `0x000000000000000e` identifies a catch-all for unknown protocols
- `0x000000000000000f` - `0x1fffffffffffffff` identifes a space reserved for the Session protocol (see RFC-0008).

5.1 Wire format encoding

The individual fields of `ApplicationData` MUST be encoded in the following order:

1. `tag`: unsigned 8 bytes, big-endian order, the 3 most significant bits MUST be cleared
2. `data`: opaque bytes, the length MUST be most the size of the HOPR protocol packet, the upper layer protocol SHALL be responsible for the framing
3. `field`: MUST NOT be serialized, it is a transient, implementation-local, per-packet field

The upper layer protocol MAY use the 4 most significant bits in `flags` to pass arbitrary signaling to the HOPR Packet protocol. Conversely, the HOPR packet protocol MAY use the 4 least significant bits in `flags` to pass arbibrary signalling to the upper-layer protocol.

The interpretation of `flags` is entirely implementation specific and MAY be ignored by either sides.

6. Appendix 1

HOPR packet protocol signals in the current implementation

The version 1 of the HOPR packet protocol (as in RFC-0004) MAY currently pass the following signals to the upper-layer protocol:

1. `0x01`: SURB distress signal. Indicates that the level of SURBs at the counterparty has gone below a certain pre-defined threshold.
2. `0x03`: Out of SURBs signal. Indicates that the received packet has used the last SURB available to the Sender.

It is OPTIONAL for any upper-layer protocol to react to these signals if they are passed to them.

7. References

[01] Bradner, S. (1997). Key words for use in RFCs to Indicate Requirement Levels. IETF RFC 2119.

12 RFC-0013: Return path incentivization

- RFC Number: 0013
- Title: Return path incentivization
- Status: Raw
- Author(s): [Name (@GitHubHandle)]
- Created: YYYY-MM-DD
- Updated: YYYY-MM-DD
- Version: v0.1.0 (Raw)
- Supersedes: none
- Related Links: none

1. Abstract

This RFC is a placeholder for future work on return path incentivization in the HOPR protocol. The specification is currently in development and will define mechanisms to incentivize relay nodes on return paths for bidirectional communication.

2. Motivation

[This section to be completed during development]

3. Terminology

[This section to be completed during development]

4. Specification

Comprehensive description of the proposed solution, including:

- Protocol overview
- Technical details (data formats, APIs, endpoints)
- Supported use cases
- Diagrams (stored in `assets/` and referenced as `![Diagram](assets/diagram-name.png)`)

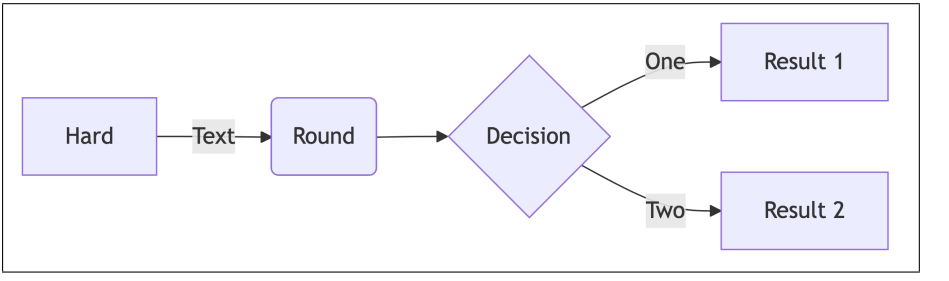


Figure 31: Mermaid Diagram 1

5. Design Considerations

Discuss critical design decisions, trade-offs, and justification for chosen approaches over alternatives.

6. Compatibility

Address backward compatibility, migration paths, and impact on existing systems.

7. Security Considerations

Identify potential security risks, threat models, and mitigation strategies.

8. Drawbacks

Discuss potential downsides, risks, or limitations associated with the proposed solution.

9. Alternatives

Outline alternative approaches that were considered and reasons for their rejection.

10. Unresolved Questions

Highlight questions or issues that remain open for discussion.

11. Future Work

Suggest potential areas for future exploration, enhancements, or iterations.

12. References

None.