Contents

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 lifecycle, contribution process, versioning system, governance model, and document structure for RFCs within the HOPR project. It specifies the stages RFCs progress through, along with the 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, transparent updates via pull requests (PRs), and clear criteria for advancing through each stage.

2. Motivation

The HOPR project requires a clear and consistent process for managing technical proposals and documenting protocol architecture. A well-defined lifecycle MUST be established and upheld to maintain coherence, ensure quality, streamline development, and provide clear expectations for contributors. This process serves multiple purposes:

- Quality assurance: ensuring that RFCs undergo appropriate review and refinement before implementation
- Transparency: making the development process visible and accessible to all stakeholders

- Version control: tracking changes and maintaining compatibility across protocol versions
- Coordination: allowing multiple contributors to work on related RFCs without conflicts or inconsistencies

3. Terminology

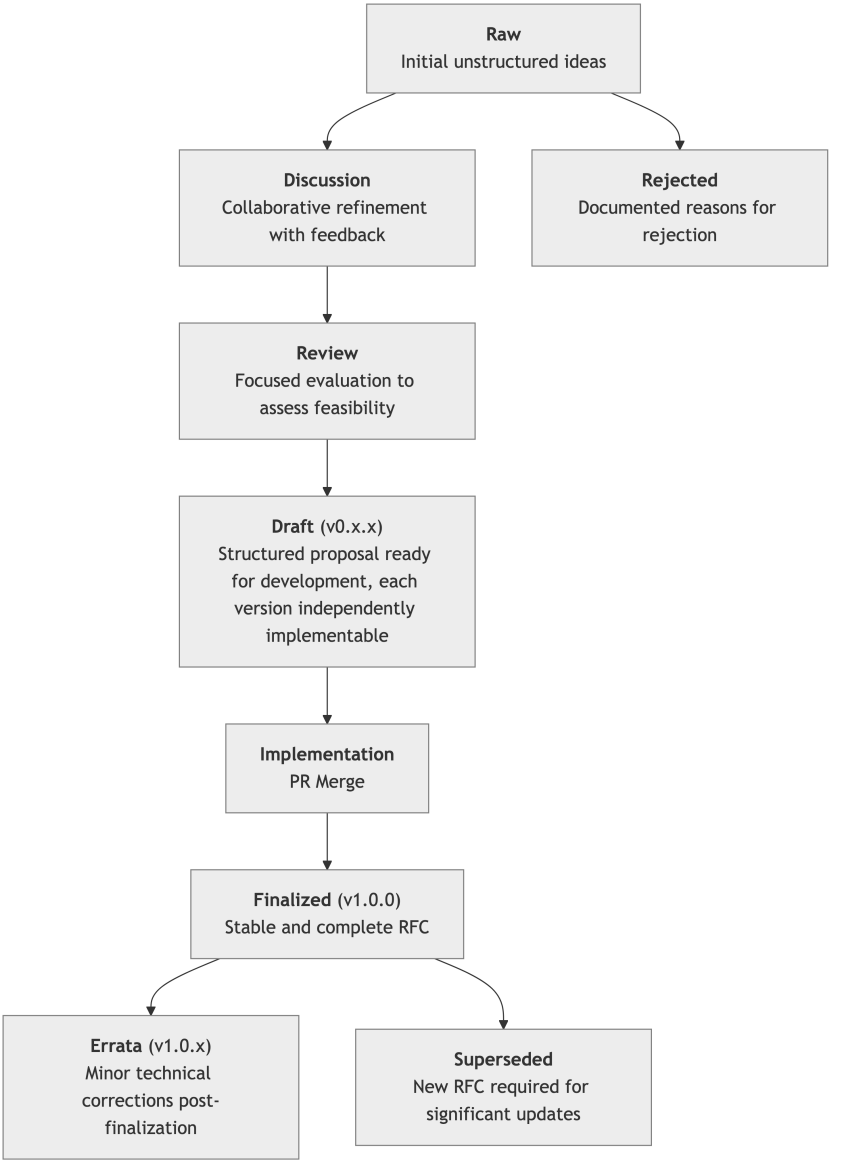
The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [01].

Draft: an RFC is considered a draft from the moment it is proposed for review. A draft MUST include a clear summary, contextual background, and initial technical details sufficient for evaluation. Drafts MUST follow the v0.x.x versioning scheme, with each version being independently reviewable and, where appropriate, independently implementable. A draft version (v0.1.0) is assigned as soon as the first PR is created and the RFC number is allocated.

4. Specification

4.1. RFC Life Cycle Stages

4.1.1. Mermaid Diagram for RFC Life Cycle Stages


Raw
Initial unstructured ideas
Discussion
Collaborative refinement with feedback
Rejected
Documented reasons for rejection
Review
Focused evaluation to assess feasibility
Draft (v0.x.x)
Structured proposal ready for development, each version independently implementable
Implementation
PR Merge
Finalized (v1.0.0)
Stable and complete RFC
Errata (v1.0.x)
Minor technical corrections post-finalization
Superseded
New RFC required for significant updates

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 finalisation process.
- Finalised: The RFC is considered stable and complete, with version `v1.0.0` assigned. Only errata modifications are permitted afterwards.
- Errata: Minor technical corrections post-finalisation 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 finalised. 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 by a link to the new RFC that supersedes it.
- Rejected: If an RFC does not progress past the discussion stage, the 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

- The 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`
- The 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]"
- References SHOULD be formatted in accordance with the following style, based on APA reference style with numeric labels in square brackets:

```
[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/chaum-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.
- The 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

- A security review phase MUST be completed before finalisation.
- 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): 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 that 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 that 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 that 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 to the forward path. The return path MAY be disjoint from 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. For example, 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: the proxy servers that make up the mixnet. They receive messages from multiple senders, shuffle them, and then send them back out in 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 such as 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 decentralised 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 Connections 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 (also referred to as HOPRnet) and its associated protocol stack. HOPR is a decentralised and incentivised mix network that enables privacy-preserving communication by routing messages through multiple relay nodes using onion routing.

HOPR's key innovation is the proof-of-relay mechanism, which addresses the challenge of establishing economically sustainable anonymous communication networks. By combining cryptographic proofs with economic incentives, HOPR enables scalable privacy infrastructure that becomes stronger with increased adoption, in contrast to volunteer-based networks that struggle with sustainability and performance issues.

This document serves as the primary entry point for understanding the HOPR network as outlined in these RFCs. It introduces the network architecture and protocol stack at a conceptual level and provides references to additional RFCs that define specific implementation details. The intended audience includes researchers, developers, and infrastructure operators seeking to understand or implement privacy-preserving communication systems based on the HOPR protocol.

2. Motivation

In the contemporary digital environment, privacy-preserving communication has become essential for safeguarding user data, supporting freedom of expression, and maintaining confidentiality in both personal and professional contexts. Conventional internet protocols provide inadequate protection for privacy, as metadata and traffic patterns can be analysed to infer sensitive information about users and their communications.

The HOPR protocol addresses these privacy challenges by implementing a decentralized mix network that:

- Provides metadata privacy: Unlike traditional communication networks that expose communication patterns, HOPR obscures sender-receiver relationships through traffic mixing and onion routing [01, 02]
- Offers economic incentives: Node operators are compensated for relaying traffic, thereby creating an economically sustainable privacy infrastructure
- Ensures decentralization: No single entity controls the network, mitigating the risks of censorship and eliminating single points of failure
- Maintains accessibility: Applications can integrate HOPR's privacy capabilities without requiring users to understand complex cryptographic concepts

The HOPR protocol is designed to be transport-agnostic, enabling operation over standard internet infrastructures while preserving robust privacy guarantees. By combining established cryptographic primitives with novel incentive mechanisms, HOPR offers a practical and scalable solution for privacy-preserving communication.

3. Terminology

All terminology used in this document, including general mix network concepts and HOPR-specific definitions, is provided in RFC-0002. That document serves as the authoritative reference for the terminology and conventions adopted across the HOPR RFC series.

4. Network Overview

The HOPR network is a decentralised, peer-to-peer mix network that provides privacy-preserving communication through multi-hop routing. The network architecture

consists of several key components that work together to ensure metadata privacy whilst incentivising participation through economic rewards.

4.1 Network Architecture

The HOPR network comprises different node roles based on their function in message routing:

- Entry nodes: nodes that initiate communication sessions and inject messages into the network
- Relay nodes: intermediate nodes that forward messages along routing paths and receive payment for their relay services
- Exit nodes: final relay nodes in a path that deliver messages to their intended destinations
- Payment infrastructure: on-chain payment channels that enable efficient microtransactions between nodes without requiring a blockchain transaction for each payment

Every HOPR node can simultaneously act as an entry node, relay node, and exit node depending on the context of different message flows. The distinction between these roles is functional rather than structural, being dependent on a node's position within a specific routing path.

4.2 Path Construction

Messages in the HOPR network are routed through multi-hop paths to provide privacy protection. Path construction involves three phases:

1. Path discovery: nodes discover available relay nodes through automated probing mechanisms detailed in RFC-0010. This process identifies which nodes are reachable, reliable, and have open payment channels.
2. Path selection: senders choose routing paths based on multiple criteria, including privacy requirements, expected latency, relay costs, and node reliability. The selection algorithm balances these trade-offs according to application needs.
3. Onion routing: messages are encrypted in multiple layers using the Sphinx packet format [02, 03], with each relay node able to decrypt only one layer to reveal the next hop whilst keeping the sender, final destination, and full path hidden.

4.3 Economic Incentives

The HOPR network employs economic incentives to ensure sustainable operation and encourage node participation:

- Micropayments: relay nodes receive small probabilistic payments for each message they forward. Payments are made through tickets that have a winning probability, enabling efficient micropayments without excessive on-chain transactions.
- Proof of relay: cryptographic proofs ensure that relay nodes actually forward messages before receiving payment. This mechanism is detailed in RFC-0005 and prevents nodes from claiming payment without providing service.
- Payment channels: unidirectional payment channels between nodes enable efficient microtransactions without high blockchain fees [04]. Channels are established on-chain but allow many off-chain payments, settling only periodically or when channels close.
- Staking rewards: nodes that stake tokens and maintain open payment channels receive additional rewards as described in RFC-0007, creating incentives for network participation beyond per-message payments.

4.4 Privacy Properties

The network architecture provides several key privacy guarantees through its layered security approach:

- Sender anonymity: relay nodes cannot determine the original sender of a message due to onion routing. Each node only knows the immediate previous hop, not the ultimate source [05].
- Receiver anonymity: intermediate nodes cannot identify the final recipient of a message. Only the exit node knows the final destination, but not the original sender [05].
- Unlinkability: observers cannot link multiple messages from the same sender or to the same receiver [05]. Different messages may take different paths, and the encryption prevents correlation.
- Traffic analysis resistance: random delays introduced by the mixer component (RFC-0006) and packet mixing prevent timing correlation attacks [06]. This ensures

that an observer cannot correlate incoming and outgoing packets based on timing patterns.

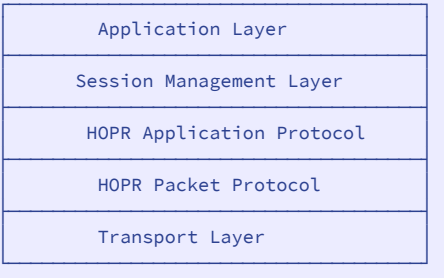
These properties hold even against an adversary who controls a subset of the network nodes, as long as at least one honest node exists in each routing path.

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 five layers, arranged as follows:



From top to bottom, these layers provide the following functionalities:

Application layer: Support for applications and services Session management layer: Session establishment and data transfer HOPR application protocol: Message routing and protocol multiplexing HOPR packet protocol: Onion routing and encryption Transport layer: Network communication

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 that enable onion routing:

- Onion encryption: multi-layer encryption ensures that each relay node can decrypt only one layer to reveal the next hop's address, maintaining sender and destination anonymity throughout the routing process.
- Sphinx-based design: based on the Sphinx packet format [03] with extensions for incentivisation. Sphinx provides compact headers and strong cryptographic guarantees about packet unlinkability.
- Fixed packet size: all packets have identical size (including header, payload, and proof-of-relay information) to prevent traffic analysis based on packet size [06]. To achieve this, variable-length messages are padded to the maximum size.
- Single-use reply blocks (SURBs): SURBs enable recipients to send reply messages back to anonymous senders without knowing their identity, supporting bidirectional communication whilst preserving anonymity.

5.2.2 Proof of Relay

The proof of relay mechanism (RFC-0005) ensures that relay nodes actually forward packets before receiving payment:

- Cryptographic proofs: each packet contains cryptographic challenges that can only be solved by a node that successfully delivers a packet to the next hop. The solution serves as mathematical proof that the relay service was performed.
- Payment integration: proofs are cryptographically bound to payment tickets. Relay nodes can only claim payment by presenting valid proofs, ensuring that compensation is tied to actual work performed.
- Fraud prevention: the mechanism detects and prevents nodes from claiming payment without providing relay services. Invalid proofs are rejected, and repeated fraud attempts can result in channel closure and stake slashing.

5.2.3 Traffic Mixing

The HOPR mixer (RFC-0006) provides traffic analysis resistance through temporal mixing:

- Temporal mixing: introduces random delays to packets before forwarding, breaking timing correlations between incoming and outgoing packets [01, 06]. This prevents attackers from linking packets based on timing patterns.
- Configurable delays: supports configurable minimum delay and delay range parameters, allowing nodes to balance privacy protection against latency requirements based on their threat model and application needs.
- Per-packet randomisation: each packet receives an independently generated random delay, ensuring that timing patterns cannot be exploited even when observing multiple packets.

The mixer operates as a priority queue ordered by release timestamps, efficiently managing packets even under high-load conditions.

5.2.4 Session Management

Session protocols provide higher-level communication primitives on top of the basic packet transport:

- Session establishment: RFC-0009 defines how nodes establish communication sessions with capability negotiation, session identifier exchange, and keep-alive mechanisms.
- Data transfer: RFC-0008 provides both reliable and unreliable data transmission modes. Reliable mode includes acknowledgements, retransmissions, and in-order delivery, whilst unreliable mode offers lower latency for applications that can tolerate packet loss.
- Message fragmentation: sessions handle segmentation of large messages into multiple packets and reassembly at the destination, transparently managing the fixed packet size constraint.
- Connection management: session lifecycle management including error handling, timeout management, and graceful termination.

5.2.5 Economic System

The economic reward system (RFC-0007) incentivises network participation through multiple mechanisms:

- Staking rewards: nodes that stake tokens receive rewards proportional to their stake, encouraging long-term network commitment and providing economic security.
- Payment channels: unidirectional payment channels enable efficient micropayments between nodes [04]. Channels are funded on-chain but support many off-chain transactions, minimising blockchain costs.
- Fair distribution: rewards are distributed equitably based on staked amounts and network participation, ensuring that nodes with open channels and good connectivity receive appropriate compensation.
- Quality-of-service incentives: the reward system considers node reliability and availability, incentivising operators to maintain high-quality service.

5.3 Protocol Flow

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

1. Path discovery: the sender discovers available relay nodes through active probing and constructs a routing path based on network topology, channel availability, and performance metrics.
2. Session establishment: if reliable delivery or bidirectional communication is required, the sender establishes a session with the recipient using the session start protocol. For simple one-way messages, this step may be skipped.
3. Packet construction: the message (possibly fragmented into multiple packets) is encrypted in multiple layers using onion encryption. Each layer includes routing information for one hop and cryptographic challenges for proof of relay.
4. Routing: packets are forwarded through the selected path, with each relay node:
    - Removing one layer of encryption to reveal the next hop's address
    - Applying random delays through the mixer component
    - Solving cryptographic challenges to generate proofs of relay
    - Claiming payment tickets upon successful delivery to the next hop
5. Delivery: the exit node delivers the packet to the intended recipient, who can decrypt the final layer to access the message content.

5.4 Integration Points

The HOPR protocol provides multiple integration points to support various applications and use cases:

- Application protocol: RFC-0011 defines a lightweight multiplexing layer that allows multiple higher-level protocols to coexist over the HOPR packet transport, similar to port numbers in TCP/UDP.
- Transport independence: the protocol can operate over different network transports (TCP, UDP, QUIC, etc.), making it deployable in various network environments without requiring specific infrastructure.
- API compatibility: through the session protocols, HOPR provides familiar networking APIs (stream-based and datagram-based) to ease application integration and lower the barrier to adoption.
- Extensibility: the modular design allows for protocol extensions and improvements without breaking existing implementations. New features can be negotiated during session establishment through capability flags.

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 protocols. The HOPR packet format is heavily based on the Sphinx packet format [01], as it aims to fulfil a similar set of goals: providing anonymous, indistinguishable packets that hide path length and ensure unlinkability of messages. The HOPR packet format extends Sphinx by adding information to support incentivisation of individual relay nodes through the Proof of Relay mechanism.

The Proof of Relay (PoR) mechanism is described in RFC-0005. This RFC focuses on the packet structure and cryptographic operations required for packet creation, forwarding, and processing.

2. Introduction

The HOPR packet format is the fundamental building block of the HOPR protocol, enabling the construction of the HOPR mix network. The format is designed to create indistinguishable packets sent between source and destination through a set of relay nodes, as defined in RFC-0002, thereby achieving unlinkability of messages between sender and destination.

In the HOPR protocol, relay nodes SHOULD perform packet mixing as described in RFC-0006 to provide additional protection against timing analysis. The packet format

is built on the Sphinx packet format [01] but adds per-hop information to enable incentivisation of relay nodes (except the last hop) for their relay services. Incentivisation of the final hop is handled separately through the economic reward system described in RFC-0007.

The HOPR packet format does not require a reliable underlying transport or in-order delivery, making it suitable for deployment over UDP or other connectionless protocols. Packet payloads are encrypted; however, payload authenticity and integrity are not guaranteed by this layer and MAY be provided by overlay protocols such as the session protocol (RFC-0008). The packet format is optimised to minimise overhead and maximise payload capacity within the fixed packet size constraint.

The HOPR packet consists of two primary parts:

1. Meta packet (also called the Sphinx packet): carries the routing information for the selected path and the encrypted payload. The meta packet includes:
   - An `Alpha` value (ephemeral public key) for establishing shared secrets
   - A `Header` containing routing information and per-hop instructions
   - An encrypted payload (`EncPayload`) containing the actual message data

   The meta packet structure and processing are described in detail in sections 3 and 5 of this RFC.
2. Ticket: contains payment and proof-of-relay information for the next hop on the path. The ticket structure enables probabilistic micropayments to incentivise relay nodes. Tickets are described in RFC-0005.

These two parts are concatenated to form the complete HOPR packet, which has a fixed size regardless of the actual payload length to prevent traffic analysis based on packet size. This fixed sized is achieved by padding payloads which fall below the maximum size in bytes.

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 throughout this document. Additionally, the following packet-protocol-specific terms are defined:

peer public/private key (also pubkey or privkey): part of a cryptographic key pair owned by a peer. The public key is used to establish shared secrets for onion encryption, whilst the private key is kept secret and used to decrypt packets destined for that peer.

extended path: a forward or return path that includes the final destination or original sender respectively. For a forward path of `N` hops, the extended path contains `N` relay nodes plus the destination node (`N+1` nodes total). For a return path, it contains `N` relay nodes plus the original sender.

pseudonym: a randomly generated identifier of the sender used to enable reply messages. The pseudonym MAY be prefixed with a static prefix to allow the sender to be identified across multiple messages whilst maintaining anonymity. The length of any static prefix MUST NOT exceed half of the entire pseudonym's size. The pseudonym used in the forward message MUST be identical to the pseudonym used in any reply message to enable proper routing.

public key identifier: a compact 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 to reduce header overhead. Implementations MAY use truncated hashes of public keys as identifiers.

|x|: denotes the binary representation length of `x` in bytes. This notation is used throughout the specification to indicate field sizes.

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 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:
  - generation of 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 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 identifier mappings of pubkeys from forward and return paths.

The packet payload MUST be between 0 and `PacketMax` bytes in length.

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`, the `B_i` MAY be additionally transformed so that it conforms to a valid field scalar. Should 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 finalise 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 and 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 are 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 the sender.

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

As the `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 the `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 steps 1 and 2 are used

    - The 5th bit of the `HeaderPrefix` is set to 1 (see section 3.4.1)
4. Generate 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 with [01] to exactly `PacketMax+|PaddingTag|` bytes.

The process works as follows:

The payload MUST always be prepended 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 also be referred to as 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 the SURBs. 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 indexed 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 the 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 using PRP:

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

This finalises all the fields of the HOPR_Packet 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 behaviour 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, therefore 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.

Should 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`) is 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 are 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|+|PoR String| 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 by 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 `use`

`r_payload` of the packet contains the encoded `Acknowledgement` 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) for HOPR packets sent between two peers via a relay node. The PoR mechanism provides cryptographic proof that a relay node has successfully delivered a packet to its destination, which can then be used to claim payment for the relay service. This solves the fundamental challenge of incentivising relay nodes in a trustless manner whilst preserving sender anonymity.

2. Motivation

The Proof of Relay mechanism addresses the challenge of ensuring reliable packet delivery in a privacy-preserving mixnet with economic incentives. When a sender (peer A) uses node B as a relay to deliver a packet to destination node C, the mechanism establishes that:

1. Node A has cryptographic guarantees that node B delivered A's packet to node C
2. After successful relaying to C, node B possesses a cryptographic proof of delivery
3. Node B can use this proof to claim a reward from node A through a payment channel
4. The identity of node A remains hidden from node C, preserving sender anonymity

Without such a mechanism, relay nodes could claim payment without actually forwarding packets, or senders would have to trust relay nodes without verification. The PoR

mechanism makes the payment conditional on proof of actual relay service, creating a trustless, incentive-compatible system.

3. Terminology

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

In addition, this document defines the following proof-of-relay-specific terms:

- channel (or payment channel): a unidirectional relation between two parties (source node and destination node) that holds a monetary balance. The source can pay out funds to the destination when certain conditions are met (specifically, when valid proof-of-relay tickets are presented).
- ticket: a cryptographic structure that enables probabilistic fund transfer within a payment channel. Tickets contain challenges that must be solved by the relay node to prove packet delivery.
- domainSeparator: a unique identifier that binds cryptographic signatures to a specific execution context (contract address, chain ID, etc.) to prevent replay attacks across different domains where the channel ledger may be deployed.
- Notice period (T_closure): the minimum elapsed time required for an outgoing channel to transition from the `PENDING_TO_CLOSE` state to the `CLOSED` state. This period allows relay nodes to claim pending rewards before channel closure.

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 uses certain cryptographic and mathematical terms. A security parameter `L` is defined, and corresponding cryptographic primitives are instantiated to achieve this security level. The specific instantiation for the current version of this protocol is provided in Appendix 1.

The security parameter `L` SHALL NOT be less than 2^128, meaning the chosen cryptographic primitive instantiations SHALL provide at least 128 bits of security against known attacks.

The following cryptographic primitives are required:

- EC group: a specific elliptic curve `E` group over a finite field, where the computational Diffie-Hellman problem has hardness at least equal to the security parameter `L`. Field elements are denoted using lowercase letters, whilst elliptic curve points (EC points) are denoted using uppercase letters.
- MUL(a,B): scalar multiplication of an EC point `B` by a scalar `a` from the corresponding finite field.
- ADD(A,B): addition of two EC points `A` and `B` on the elliptic curve.
- Public key: a non-identity EC group element of large order, used to identify a node and establish shared secrets.
- Private key: a scalar from the finite field of the chosen EC group, corresponding to a public key. Must be kept secret.
- Hash `H(x)`: a cryptographic hash function taking an input of any size and returning a fixed-length output. The security of `H` against preimage, collision, and second-preimage attacks SHALL be at least `L` bits.
- Verifiable Random Function (VRF): a function that produces a pseudo-random value along with a proof of correct computation. The output is publicly verifiable but cannot be forged or precomputed without the secret key.

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

Payment channels are the foundation of the HOPR incentive mechanism. They enable efficient micropayments between nodes without requiring a blockchain transaction for each packet relayed.

Let A, B, and C be peers participating in the mixnet. Each node possesses its own

private key (`Kpriv_A`, `Kpriv_B`, `Kpriv_C`) and the corresponding public key (`P_A`, `P_B`, `P_C`). Public keys are publicly exposed to enable packet routing and shared secret establishment.

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

When node A wishes to communicate with node C using node B as a relay, node A opens a unidirectional payment channel with node B (denoted A -> B), depositing funds into this channel on-chain. The channel holds the current balance and additional state information shared between A and B, and funds flow strictly in the direction A -> B.

Channel funds MUST be strictly greater than 0 and strictly less than 2^96 (to fit within the ticket structure's amount field).

There MUST NOT be more than one payment channel between any two nodes A and B in a given direction. Since channels are unidirectional, there MAY simultaneously exist both a channel A -> B and a channel B -> A.

Each channel has a unique, deterministic identifier: the channel ID. The channel ID for A 𝌆 B MUST be computed as: `channel_id=H(f(P_A)||f(P_B))` where `||` denotes byte-wise concatenation and `f` represents a deterministic encoding function for public keys (typically compressed EC point encoding). This construction is directional: the source node's public key appears first, followed by the destination node's public key.

Channels transition through three distinct lifecycle states:

1. OPEN: the channel is active and can be used for packet relay payments
2. PENDING_TO_CLOSE: the channel is in the process of closing; nodes can still claim pending rewards during the notice period
3. CLOSED: the channel is permanently closed; no further operations are possible

These states can be represented using the `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 to 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 be 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_cl`

osure 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 in 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 become unclaimable after the state transition, resulting in a loss 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 beforehand.

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 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 the 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(0xfcb7796f0000000000000000000000000000000000000000000000000000000000
↪  || 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 the `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 the `i+1`-th node and the `hint` value for 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 always be 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 the 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. If 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. it 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 unacknowledged by the node and SHOULD be indexed by `hint`. The stored unacknowledged 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 the `amount` from the 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 for how the `ack_secret` field is calculated:

1. if the `HOPR_Packet` being acknowledged has been successfully processed (along with a 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 `ECScalar`).

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 5.2): `ack_secret` is set to a random EC point on `E`.

The `signature` field contains the signature of the encoded `ack_secret` bytes. The signature is 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` and 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 corresponds to the packet that has just been acknowledged).

Let `SharedSecret_i_own` be the value from 1) in Section 5.2.1. 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, which 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 that it can 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 acknowledged tickets are probabilistic in the sense that the monetary value represented by the `amount` MUST be claimable only if the acknowledged 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 a 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 a winning ticket.

6.2. Claiming a winning ticket

The monetary value represented by the `amount` on a winning ticket can be claimable at some third party which provides such a service. Such a third party MUST have the ability

to modify the global state of all the involved `Channels`.

Such `amount` SHOULD be claimable only if the `Channel` corresponding to 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 a field element of `E`
- the public key `P` of the recipient of the ticket
- values `V`, `h` and `s` computed in Section 5.2.3.3

If the third party wishes to verify the claim, it proceeds as follows. If any of the checks below fail, 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 than or equal to `ticket_index` on the `Channel`
4. The third party applies appropriate encoding to obtain `H_P` from `P`. It then 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 third party MAY also adjust the balance on a `Channel` that is in the opposite direction of the claim (ticket receiver -> ticket issuer), if such a channel exists and is in an `OPEN` state.

Upon successful redemption, the third party MUST ensure that:

1. The `balance` on the `Channel` from which the claim has been made MUST be decreased by `amount`
2. The `ticket_index` on the `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):

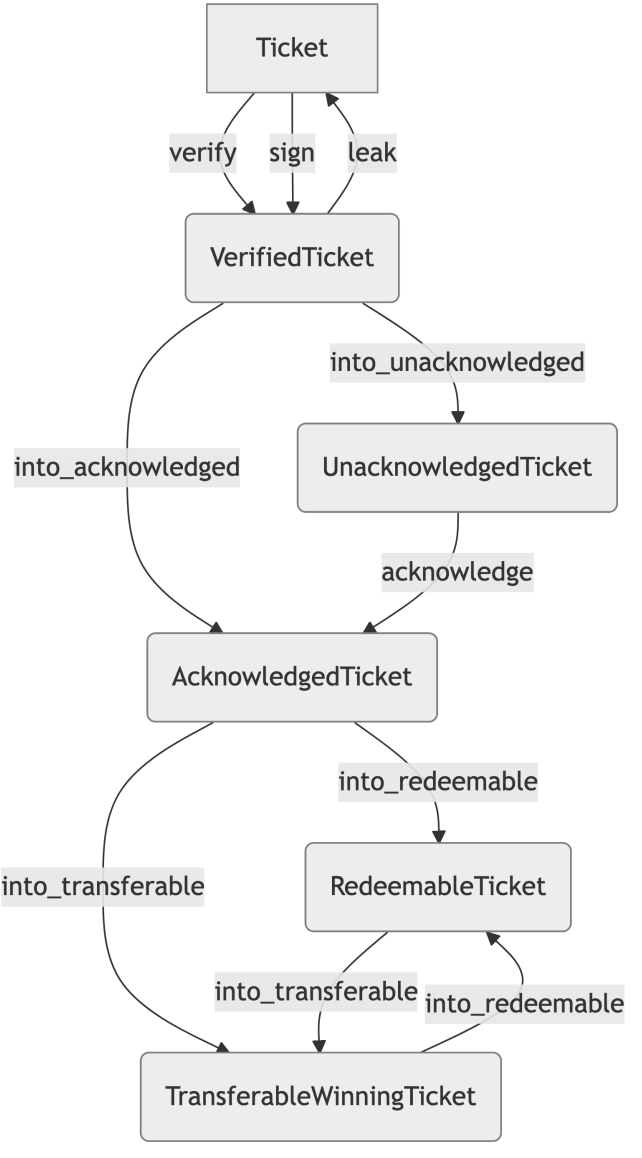
- Hash function `H` is Keccak256
- Elliptic curve `E` is chosen as secp256k1
- HS is instantiated via `hash_to_field` using `secp256k1_XMD:SHA3-256_SSWU_RO_` as defined in [02]
- HC is instantiated via `hash_to_curve` using `secp256k1_XMD:SHA3-256_SSWU_RO_` as defined in [02]
- The one-way encoding `ECPoint` 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


Ticket
verify
sign
leak
VerifiedTicket
into_unacknowledged
into_acknowledged
UnacknowledgedTicket
acknowledge
AcknowledgedTicket
into_redeemable
into_transferable
RedeemableTicket
into_transferable
into_redeemable
TransferableWinningTicket

1. `Ticket--sign-->VerifiedTicket`
   - Pre-conditions:
     - 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 correlations between incoming and outgoing packets. By applying random delays to packets, the mixer effectively destroys temporal patterns that could otherwise be exploited for traffic analysis attacks. This specification details the mixer's design, implementation requirements, and integration points to enable consistent implementations across different HOPR nodes whilst balancing anonymity protection against latency and throughput requirements.

2. Motivation

In mixnets, simply forwarding packets through multiple hops is insufficient to prevent traffic analysis attacks. Even with encrypted packet contents and obscured routing paths, adversaries can correlate packets by observing timing patterns. An observer monitoring network traffic at multiple points can potentially link incoming and outgoing packets based on their arrival and departure times. This technique is known as timing correlation or an intersection attack.

Without temporal mixing, an adversary who observes a packet arriving at node A at time t_1 and a packet leaving node A at time $t_2 \approx t_1$ can infer with high probability that these packets are the same, thus tracking packets through the network and potentially deanonymising communications.

The HOPR mixer addresses this attack vector by:

- Breaking temporal correlations: introducing random delays between packet arrival and departure times, making timing-based correlation significantly more difficult
- Configurable privacy-latency trade-offs: providing tunable delay parameters to balance anonymity protection against performance requirements
- Efficient implementation: using a priority queue that maintains packet ordering by release time, enabling `O(logn)` operations
- High-throughput support: maintaining mixing effectiveness even under high packet rates

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 at which 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 that is split into these steps:

1. Accept packets from upstream components
2. Assign random delays to each packet
3. Store packets in a time-ordered buffer
4. Release packets when their delay expires

4.2. Configuration Parameters

The mixer accepts the following configuration parameters that control the delay distribution:

1. `min_delay`: minimum delay applied to packets (default: 0 ms). This establishes the lower bound of the delay interval.
2. `delay_range`: the range from minimum to maximum delay (default: 200 ms). The maximum delay is `min_delay` + `delay_range`.

The actual delay for each packet is randomly selected from a probability distribution over the interval `[min_delay,min_delay+delay_range]`. The default implementation uses a uniform distribution, but implementations MAY support additional distributions (e.g., exponential, Poisson) for enhanced anonymity properties.

4.3. Core Components

4.3.1. Delay Assignment

When a packet arrives at the mixer, the following operations are performed:

1. A random delay is generated using a cryptographically secure random number generator (CSPRNG)
2. The release timestamp is calculated as `current_time+random_delay`
3. The packet is wrapped with its release timestamp metadata
4. The wrapped packet is inserted into the mixing buffer, ordered by release timestamp

To generate a satisfactory random delay, the following conditions MUST be met:

- MUST use a CSPRNG with sufficient entropy (at least 128 bits of entropy)
- MUST generate independent delays per packet, with no correlation or reuse across packets
- SHOULD use uniform distribution as the baseline; other distributions (e.g., exponential, Poisson) MAY be supported via configuration
- MUST NOT leak information about delay values through timing side channels

Different mixing strategies produce different results. A uniform distribution will provide a simple baseline that is easy to implement and analyse. 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, but require careful parameter tuning and integration with cover traffic generation.

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(logn)` 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 Behaviour

4.4.1. Packet processing flow

Packet processing SHOULD use the following 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 an 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 Optimisation

An implementation should prioritise:

- Minimal allocations: Pre-allocated buffer reduces memory pressure
- Efficient data structures: Binary heap provides `O(logn)` operations
- Lock minimisation: Fine-grained locking for concurrent access
- Timer efficiency: Single shared timer reduces system overhead, including minimising 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 behaviour 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 a sufficient number 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 numbers 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 a 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 analyse.

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 optimisations 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 the mechanisms of the HOPR economic reward system, specifically how the eligible peer set is constructed and how rewards are calculated and distributed among peers. The system ensures fair and sustainable incentivisation of node operators whilst preventing gaming and maintaining network decentralisation.

The reward system operates by collecting data from multiple sources (blockchain, subgraphs, node APIs), filtering for eligible peers based on stake and connectivity requirements, applying an economic model to calculate reward allocations, and distributing rewards through the HOPR network itself.

2. Motivation

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

The economic reward system is a necessary component of the HOPR mixnet, as it incentivises node runners to keep their nodes running in order to have a network topology that is as stable as possible. It must employ fair logic that never favours or disadvantages a subset of node runners, and that encourages sustainability without compromising decentralisation. It must also incentivise node runners to be connected to other nodes

in the network via channels. Isolated nodes are far less useful to the network than well-connected nodes.

3. Terminology

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

- Subgraph: an off-chain data indexer (such as The Graph protocol) that indexes blockchain events and provides queryable access to on-chain data including NFT holders, registered nodes, allocations, and EOA balances.
- API: the HOPR node HTTP API that provides real-time network data including topology information, peer connectivity, and channel balances.
- EOA (Externally Owned Account): a blockchain account directly controlled by a private key (as opposed to a smart contract account). EOAs can initiate transactions and hold token balances.
- Safe: a smart contract wallet (specifically Gnosis Safe) used for holding tokens with multi-signature security. Node operators typically use Safes to manage their staked funds.
- CT node: a node running the Cover Traffic (CT) application, which is used by the HOPR Association to distribute rewards. CT nodes are excluded from receiving rewards to prevent self-dealing.
- NFT Holder: an address holding a specific NFT that grants preferential treatment in the reward system (lower staking thresholds).
- SessionToSocket: an implementation object that manages a UDP session and socket for communicating with a specific peer.
- MessageFormat: a class responsible for encoding message metadata and payload as bytes for transmission over the network.

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].

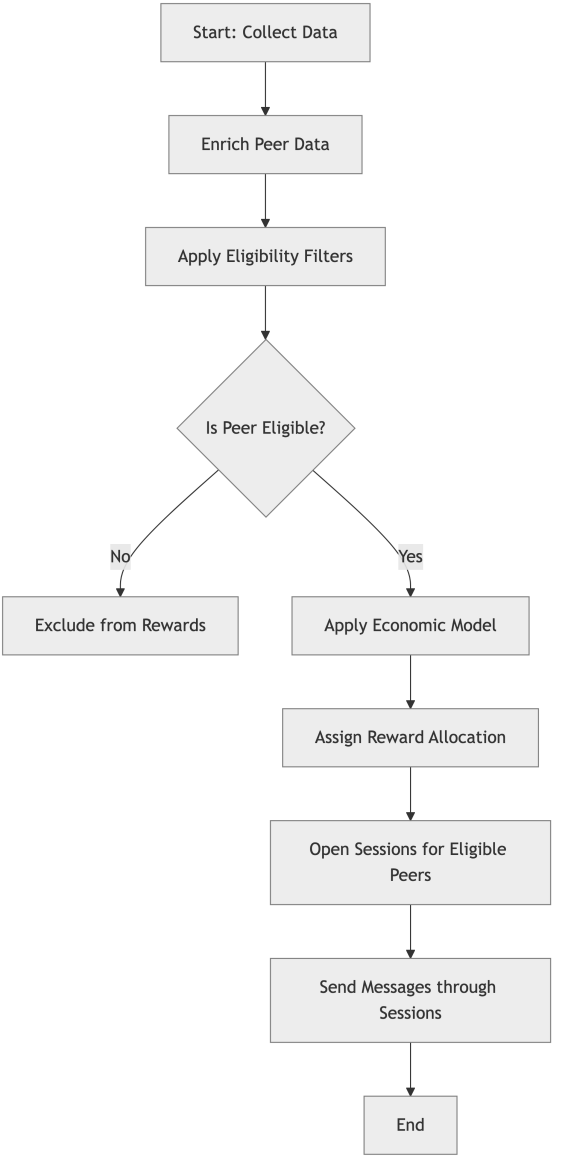
4. System Overview

The HOPR Cover Traffic (CT) system distributes rewards to eligible peers based on their participation and stake in the network. The reward distribution process consists of several key stages:

1. Data collection and enrichment: gathering peer data from multiple sources (blockchain, subgraphs, node APIs) and enriching each peer's profile with on-chain and off-chain information
2. Eligibility filtering: applying a series of filters to determine which peers qualify for rewards based on stake, connectivity, and participation criteria
3. Economic model application: calculating the reward allocation (measured in message count) for each eligible peer using an economic model that considers stake amounts, network connectivity, and contribution metrics
4. Message distribution: managing the technical process of sending reward messages to eligible peers via UDP sessions, ensuring fair and robust distribution

This multi-stage process ensures that rewards are distributed fairly, transparently, and in proportion to each peer's contribution to network stability and performance.

The following flowchart summarizes the overall process:


Start: Collect Data
Enrich Peer Data
Apply Eligibility Filters
Is Peer Eligible?
No
Yes
Exclude from Rewards
Apply Economic Model
Assign Reward Allocation
Open Sessions for Eligible Peers
Send Messages through Sessions
End

5. Data Collection and Enrichment

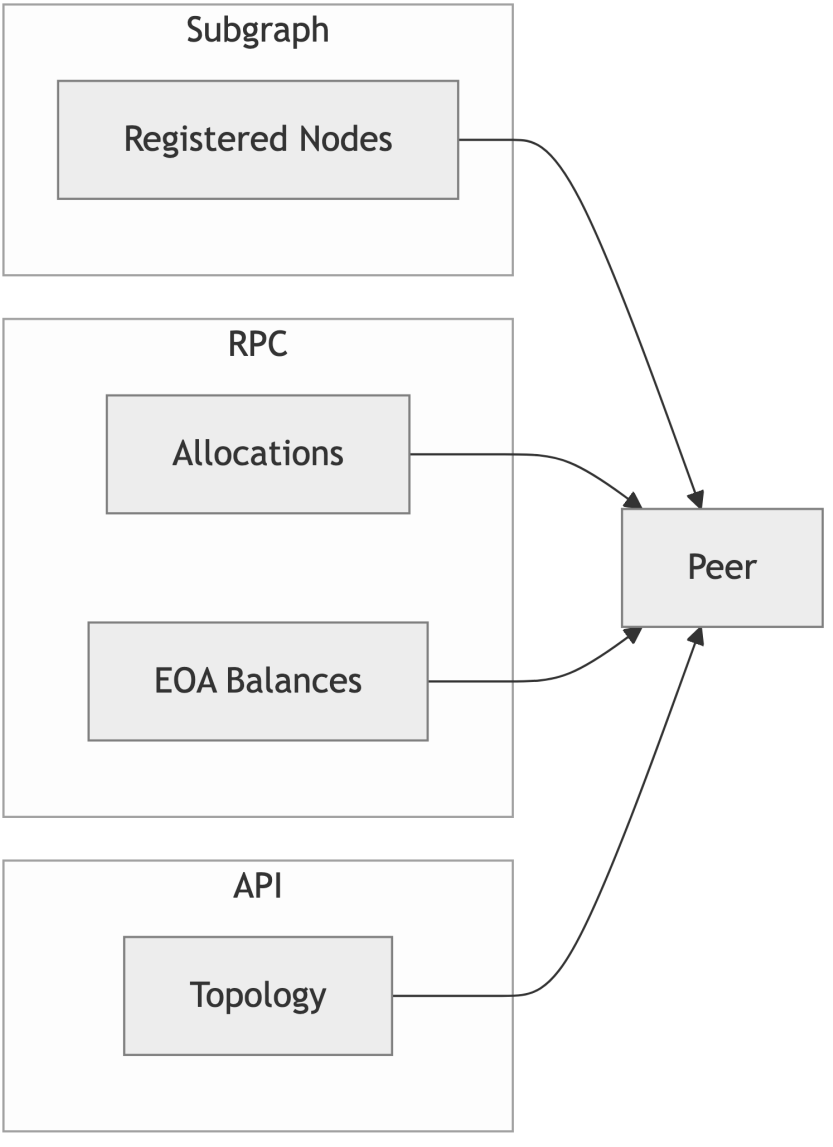
5.1 Data Sources

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 reward distribution to individuals 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.

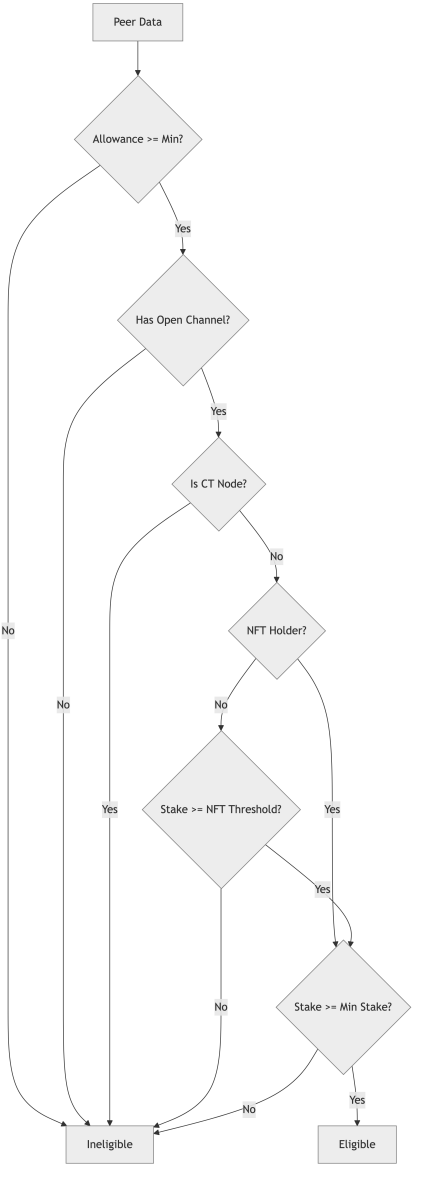
The following diagram illustrates the data enrichment process:


Subgraph
Registered Nodes
RPC
Allocations
EOA Balances
API
Topology
Peer

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.

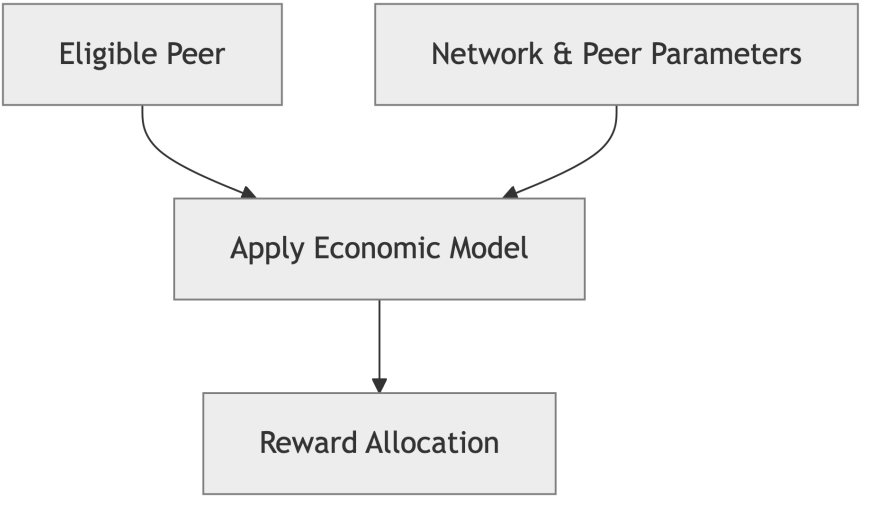
The following flowchart details the filtering logic:


Peer Data
Allowance >= Min?
Yes
No
Has Open Channel?
Yes
No
Is CT Node?
No
Yes
NFT Holder?
No
Yes
Stake >= NFT Threshold?
Yes
No
Yes
Stake >= Min Stake?
No
Yes
Ineligible
Eligible

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 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 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 optimise 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.

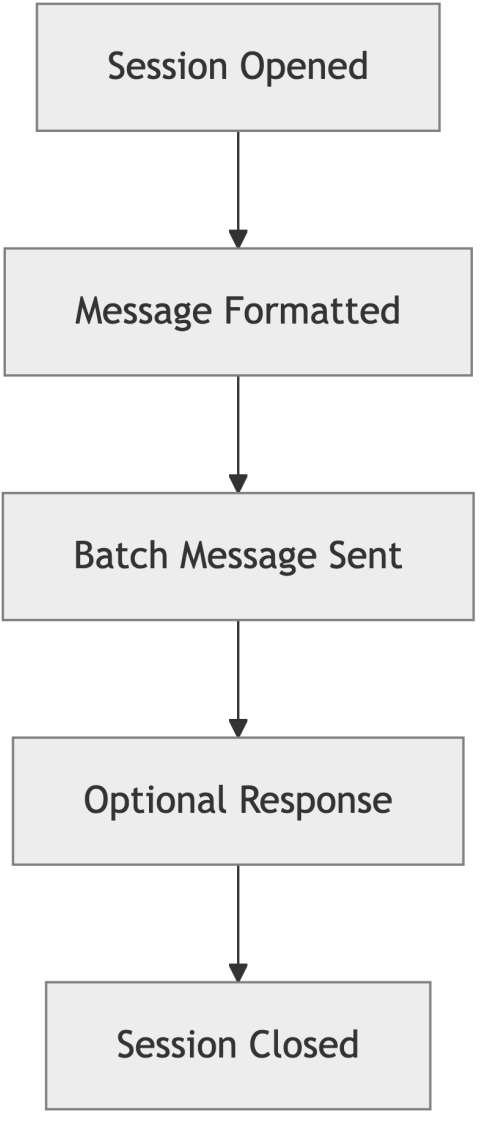
9. Message Sending Architecture

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.

The batching of multiple message sendings is 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:


Session Opened
Message Formatted
Batch Message Sent
Optional Response
Session Closed

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 analyse 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 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 features [01] including message segmentation, reassembly, acknowledgement, and retransmission, whilst maintaining simplicity and efficiency suitable for mixnet deployment. This protocol works in conjunction with the HOPR session start protocol (RFC-0009) to provide complete session management capabilities for applications within the HOPR ecosystem.

The protocol supports both reliable (TCP-like) and unreliable (UDP-like) transmission modes, allowing applications to choose the appropriate trade-off between reliability and latency for their use case.

2. Motivation

The HOPR mixnet uses HOPR packets (RFC-0004) to transport data between nodes. This fundamental packet-sending mechanism operates as a fire-and-forget transport similar to UDP [03], providing no guarantees of delivery, ordering, or message boundaries. Whilst this simplicity is appropriate for the packet layer, application developers typically require higher-level features such as:

- Reliable delivery: ensuring that messages are delivered or that the sender is notified of failures
- Message ordering: receiving messages in the order they were sent
- Message segmentation: handling messages larger than the fixed packet size
- Flow control: managing transmission rates to prevent overwhelming receivers

To ease adoption, HOPR nodes must provide a way for applications to use these features without reimplementing TCP [01] or UDP [02] from scratch. Since the HOPR protocol is not IP-based, implementing these protocols directly would require complex IP protocol emulation.

The HOPR session data protocol fills this gap by providing both reliable and unreliable data transmission modes directly over the HOPR packet transport. Session establishment and lifecycle management are handled by the HOPR session start protocol (RFC-0009), whilst this protocol focuses exclusively on efficient data transmission once a session is established.

3. Terminology

Terms defined in RFC-0002 are used. Additionally, this document defines the following session-protocol-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. Frames represent complete application messages that may span multiple packets.
- segment: a fixed-size fragment of a frame. Frames larger than the packet MTU are split into segments for transmission, with each segment carrying metadata about its position within the frame to enable reassembly.
- frame ID: a 32-bit unsigned integer that uniquely identifies a frame within a session (1-indexed, starting from 1). Frame ID values are interpreted as big-endian unsigned integers and increment sequentially with each new frame.
- Sequence Number (SeqNum): an 8-bit unsigned integer indicating a segment's position within its frame (0-indexed, starting from 0). This enables correct reassembly of frames from segments.
- Sequence Flags (SeqFlags): an 8-bit value encoding additional segment metadata, including whether the segment is the final segment of a frame and whether it

represents a terminating segment.

- session socket: the endpoint abstraction that implements the session protocol, available in both reliable and unreliable variants. Session sockets provide familiar send/receive APIs to applications.
- MTU (Maximum Transmission Unit): the maximum size of a single HOPR protocol message payload, denoted as `C` throughout this specification. This is determined by the packet format defined in RFC-0004.
- terminating segment: a special segment with the termination flag set that signals the graceful end of a session. Terminating segments carry no data payload.

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

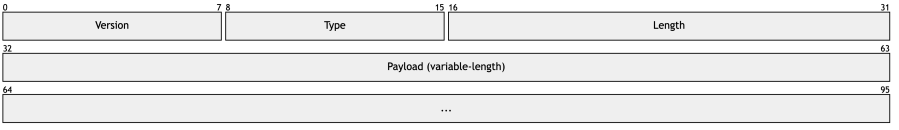
The protocol supports two operational modes:

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

Session establishment and lifecycle management are 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:



"Common Structure"

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:

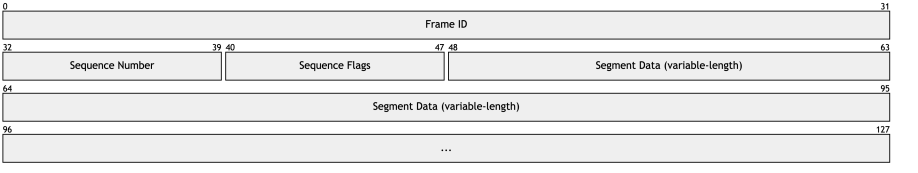
- Length field (2 bytes) in the common message format

- 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



"Segment"

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`
5-0	Segment Count	Total segments in frame minus 1	`0-63` (1-64 segments)

4.3.3 Segmentation Rules

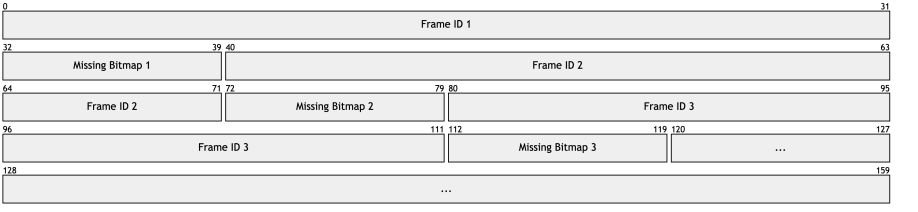
Rule	Requirement	Description
Segmentation Threshold	MUST	Frames MUST be segmented when larger than `(C-10)` bytes
Maximum Segments	MUST	Maximum 64 segments per frame (6-bit sequence length field limit)
Segment Sizing	SHOULD	Each segment except the last SHOULD be of equal size
Empty Segments	MUST	Empty 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



"Retransmission Request Message"

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

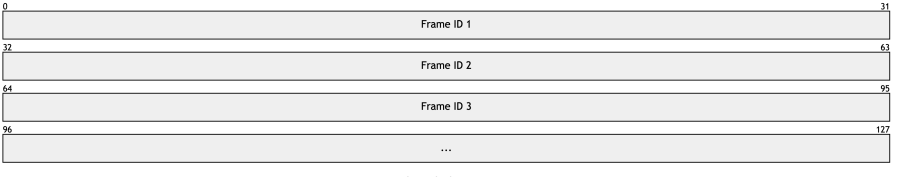
Note: This message MUST be used only for frames with up to 8 segments (due to bitmap size limitation). Reliable sessions are limited to 7 segments per frame. Unreliable sessions SHOULD not have this limitation.

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



"Frame Acknowledgement Message"

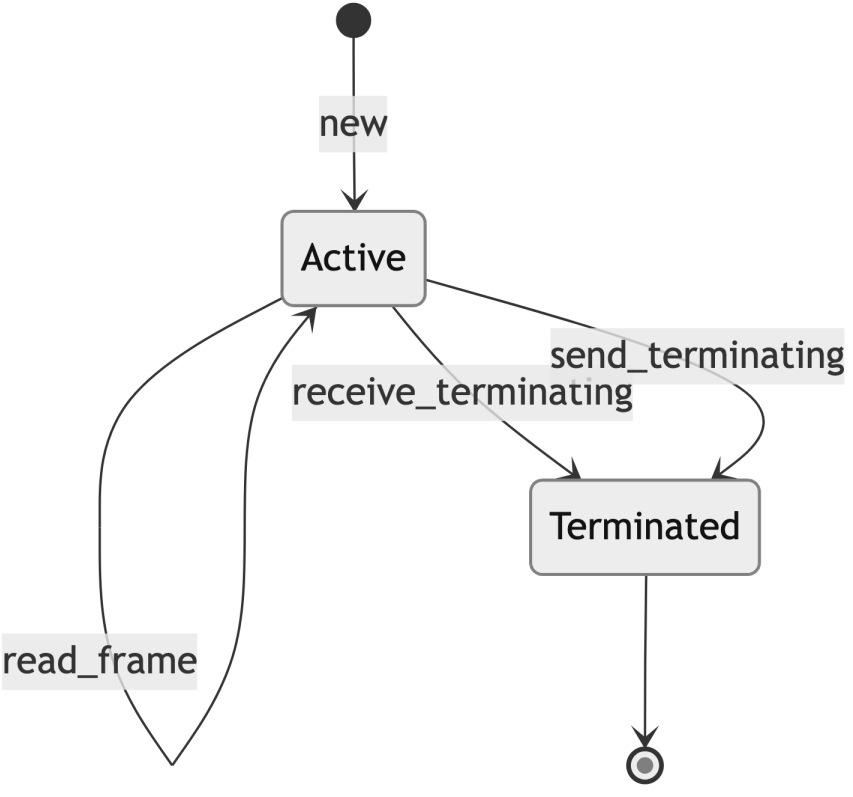
Field	Size	Description	Rules
Frame ID List	4 bytes each	List of fully received frame identifiers	See Acknowledgement Rules table below

4.5.2 Acknowledgement Rules

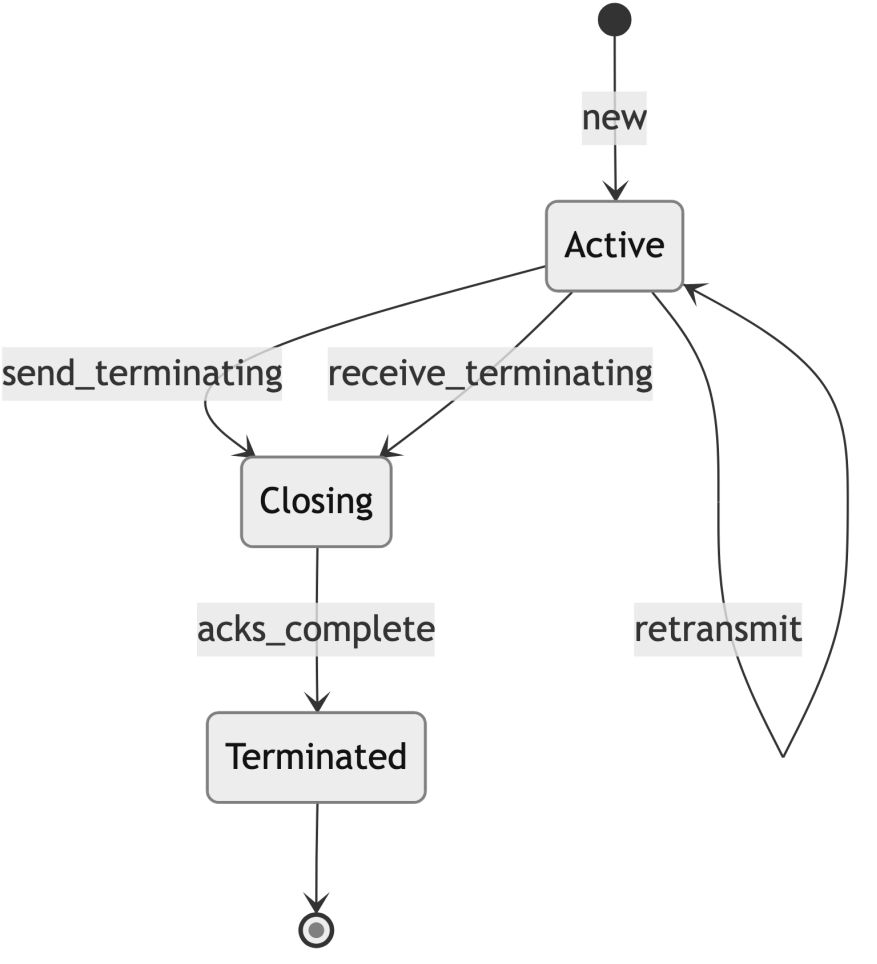
Rule	Requirement	Description
Ordering	MUST	Frame IDs MUST be in ascending order
Padding	MAY	Frame ID of `0` indicates padding (ignored)
Entry Limit	MUST	Maximum frame IDs per message: `(C-4)/4`
Completeness	MUST	Only acknowledge frames that are fully received

4.6 Protocol State Machines

4.6.1 Unreliable Socket State Machine


new
Active
send_terminating
receive_terminating
Terminated
read_frame

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

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

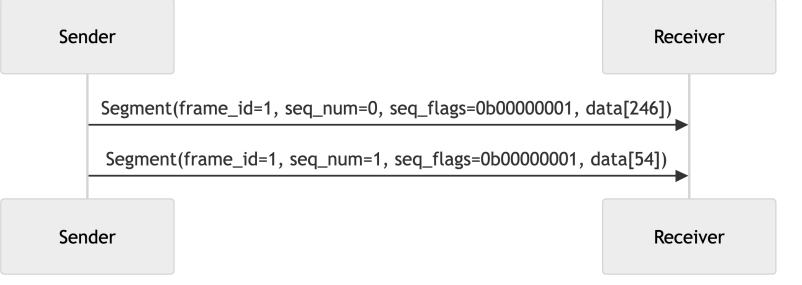
1. Either party MAY send a terminating segment (empty segment with the termination flag set)
2. Upon receiving a terminating segment:
    - Unreliable sockets SHOULD close immediately
    - Reliable sockets MUST complete pending acknowledgements before closing
3. No data frames MUST be sent after a terminating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

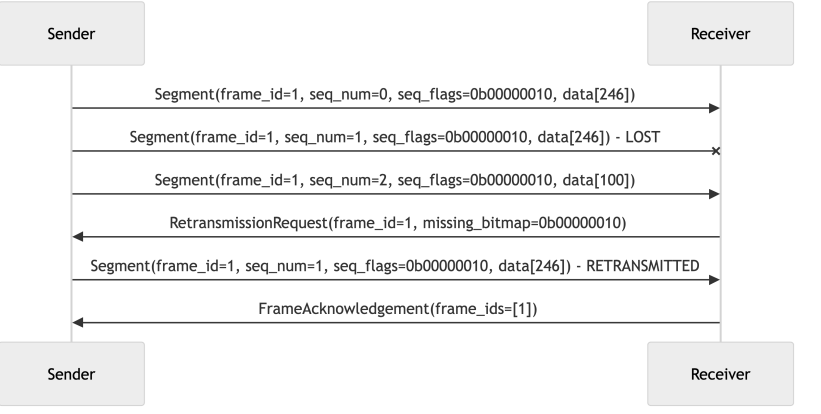
4.9.1 Simple Frame Transmission (Unreliable Mode)

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



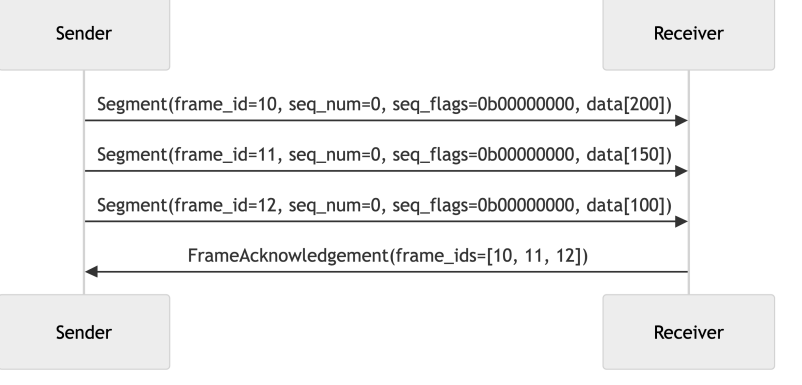
4.9.2 Frame with Retransmission (Reliable Mode)

Reliable transmission where the middle segment is lost and retransmitted:



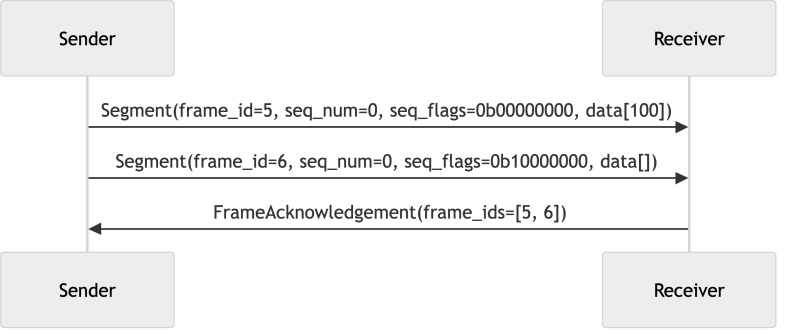
4.9.3 Multiple Frame Acknowledgement (Reliable Mode)

Efficiently acknowledging multiple received frames in a batch:



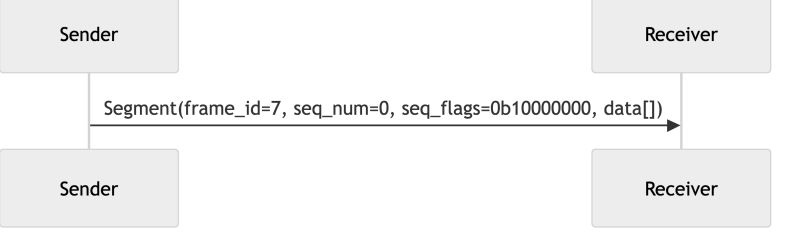
4.9.4 Session Termination (Reliable Mode)

Graceful session termination with acknowledgement:



4.9.5 Session Termination (Unreliable Mode)

Immediate session termination without acknowledgement:



5. Design Considerations

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 among:

- 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 a 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, which provides a handshake mechanism for establishing communication sessions between peers in the HOPR mixnet. The protocol manages session establishment, lifecycle management, and capability negotiation, using HOPR packets as the underlying transport layer. It defines a standardised method for initiating sessions, exchanging session parameters (identifiers, targets, and capabilities), and maintaining session state through periodic keep-alive messages.

The session start protocol operates independently of the session data protocol (RFC-0008), which handles actual data transmission once a session has been established. This separation allows the handshake mechanism to evolve independently from data transfer protocols.

2. Motivation

The HOPR mixnet requires a standardised mechanism for establishing communication sessions between nodes. While the session data protocol (RFC-0008) handles reliable and unreliable data transmission, a complementary protocol is needed for session initialisation. The session start protocol addresses the following requirements:

1. Session establishment: Provide a handshake mechanism to initiate sessions with capability negotiation, allowing peers to agree on session parameters before data exchange begins.
2. Session identification: Enable exchange of unique session identifiers and target endpoints, ensuring both peers can correctly route subsequent messages.
3. Lifecycle management: Define clear state transitions for session establishment, including timeout handling and graceful error reporting.
4. Error handling: Provide structured error reporting for common failure scenarios (e.g., resource exhaustion, busy nodes), enabling intelligent retry logic.
5. Liveness maintenance: Support keep-alive mechanisms to maintain long-lived sessions and detect peer failures.

The session start protocol is intentionally lightweight and transport-agnostic, making it suitable for use over various packet-based transports while being optimised for the HOPR mixnet.

3. Terminology

Terms defined in RFC-0002 are used throughout this specification. 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 MUST be generated using a cryptographically secure pseudo-random number generator (CSPRNG) and are interpreted as big-endian unsigned integers.
- session target: The destination or purpose of a session, typically representing an address or service identifier. Session targets are encoded using CBOR format [01] to allow flexible representation of various endpoint types (e.g., IPv4/IPv6 addresses with ports, service URIs).
- session capabilities: A bitmap of session features and options negotiated during session establishment. The capabilities field enables peers to agree on optional protocol features, with unrecognised bits being safely ignored to support backward compatibility.
- session ID: A unique identifier assigned by the responder to identify an established session. Session IDs are encoded using CBOR format and MUST be unique within the

responder's session namespace. Within HOPR, session IDs follow a specific format (see Appendix 1).

- entry node: The node that initiates a session establishment request. The entry node generates the initial challenge and specifies the desired session target and capabilities.
- exit node: The node that receives and responds to a session establishment request. The exit node validates the request, assigns a unique session ID upon success, and returns either a `SessionEstablished` or `SessionError` message.

4. Specification

4.1 Protocol Overview

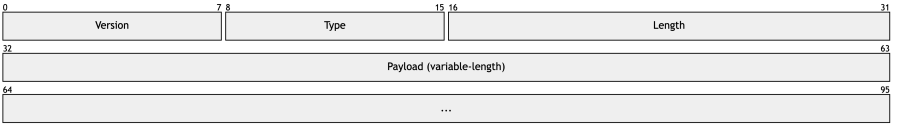
The session start protocol operates at version 2 and defines four message types that manage the complete lifecycle of session establishment and maintenance:

1. StartSession: Initiates a new session, carrying the challenge, target endpoint, and capability flags.
2. SessionEstablished: Confirms successful session establishment, returning the original challenge and newly assigned session ID.
3. SessionError: Reports session establishment failure with a specific error code and the original challenge for correlation.
4. KeepAlive: Maintains session liveness by periodically signalling that the session is still active.

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 and implementations.

4.2 Message Format

All session start protocol messages share a common header structure that enables protocol versioning, message type discrimination, and variable-length payloads:



"Common Message Format"

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 the following fields:

Protocol message fields:

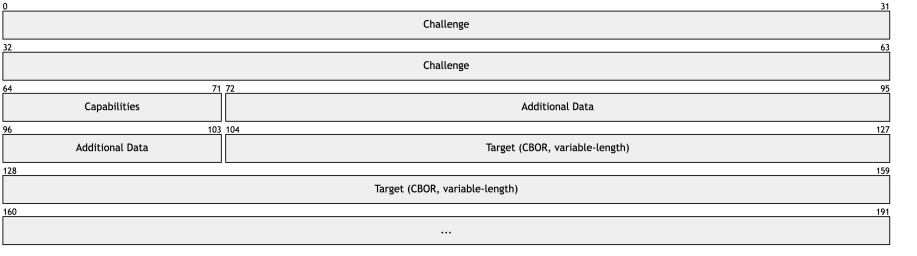
- Length field (2 bytes) in the common message format

- 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 (see Appendix 1)
- Any future numeric fields added to the protocol

This requirement ensures consistent interpretation across different architectures (e.g., x86, ARM, RISC-V) and prevents interoperability issues between implementations.

4.3 StartSession Message

The `StartSession` message initiates a new session with a remote peer. The entry node sends this message to request session establishment, specifying the desired target endpoint and capability flags.



"StartSession Message"

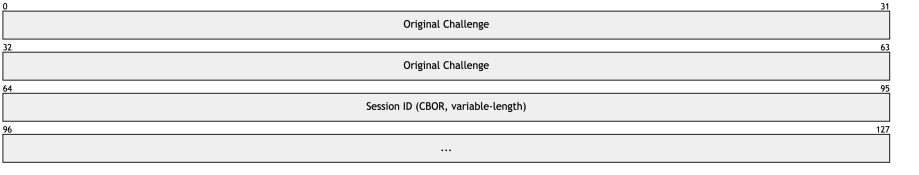
Field	Size	Description	Notes
Challenge	8 bytes	Random challenge for correlating responses	MUST be generated using CSPRNG to prevent prediction
Capabilities	1 byte	Session capabilities bitmap	See Capability Flags table; unrecognised bits SHOULD be ignored
Additional Data	4 bytes	Capability-dependent options	Set to `0x00000000` if unused; interpretation depends on capabilities
Target	Variable	CBOR-encoded session target	Examples: `"127.0.0.1:1234"`, `"wss://relay.example.com:443"`

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

4.4 SessionEstablished Message

The `SessionEstablished` message confirms successful session establishment. The exit node sends this message in response to a valid `StartSession` request, assigning a unique session ID that will be used for all subsequent communication in this session.

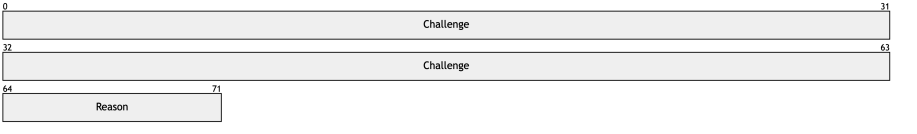


"SessionEstablished Message"

Field	Size	Description	Notes
Original Challenge	8 bytes	Challenge from `StartSession` message	MUST exactly match the challenge from the initiating `StartSession` request
Session ID	Variable	CBOR-encoded session identifier	Assigned by exit node; MUST be unique within exit node's session namespace

4.5 SessionError Message

The `SessionError` message reports session establishment failure. The exit node sends this message when it cannot establish a session, providing a specific error code to indicate the reason for failure. This enables the entry node to implement intelligent retry logic or select alternative exit nodes.



"SessionError Message"

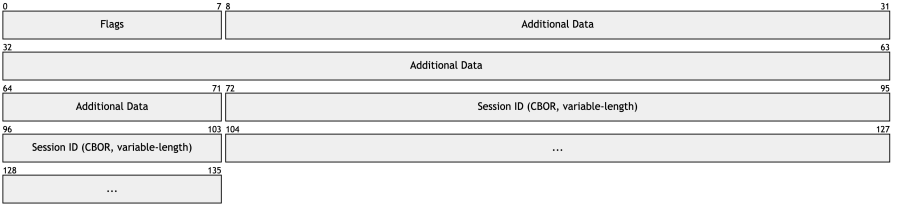
Field	Size	Description	Notes
Challenge	8 bytes	Challenge from `StartSession` message	MUST exactly match the challenge from the initiating `StartSession` request
Reason	1 byte	Error reason code	See Error Codes table below

4.5.1 Error Codes

Code	Name	Description	Recommended Action
`0x00`	Unknown Error	Unspecified error condition	Retry with different parameters or select alternative exit node
`0x01`	No Slots Available	Exit node has no available session slots	Retry after delay or select alternative exit node
`0x02`	Busy	Exit node is temporarily busy processing requests	Retry after brief exponential backoff delay

4.6 KeepAlive Message

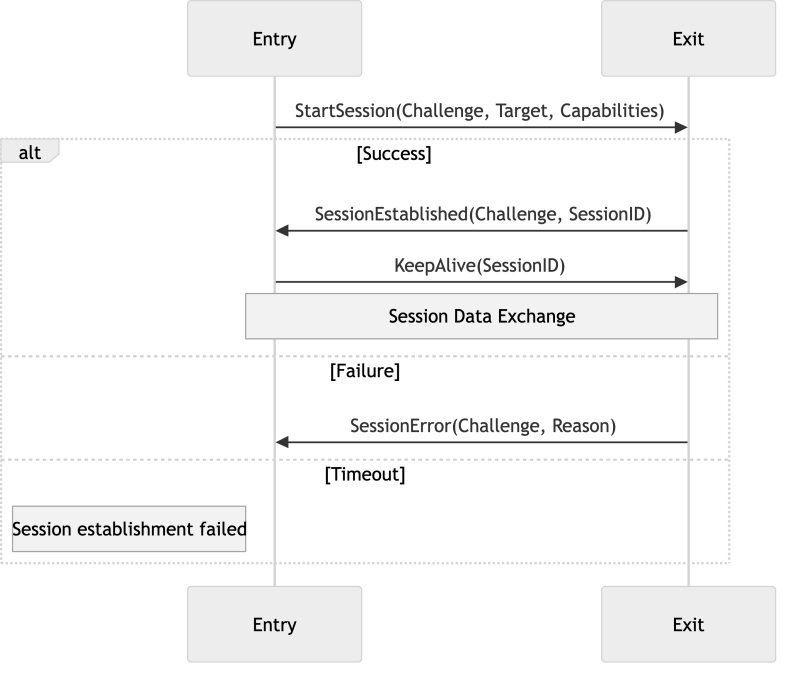
The `KeepAlive` message maintains session liveness. Either peer can send this message periodically to signal that the session is still active and prevent session timeout. The frequency of keep-alive messages depends on the session timeout policy of the peers.



"KeepAlive Message"

Field	Size	Description	Notes
Flags	1 byte	Reserved for future use	MUST be set to `0x00` by senders; SHOULD be ignored by receivers
Additional Data	8 bytes	Flag-dependent options	Set to `0x0000000000000000` if unused; interpretation may depend on future flags
Session ID	Variable	CBOR-encoded session identifier	MUST match an established session ID

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 (SHOULD be configurable)
Challenge Size	8 bytes	Fixed size for challenge field
Max Payload Length	65535 bytes	Maximum message payload size (limited by Length field)

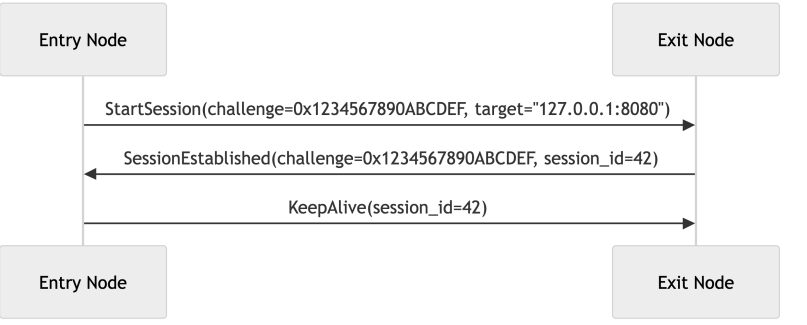
4.9 Protocol Rules

Rule	Requirement Level	Description
Challenge Generation	MUST	Challenge values MUST be randomly generated using a cryptographically secure PRNG
Session ID Uniqueness	MUST	Session IDs MUST be unique within the exit node's session namespace
Byte Order	MUST	All multi-byte integer fields MUST use network byte order (big-endian)
CBOR Encoding	MUST	Session targets and session IDs MUST use CBOR encoding [01]
Payload Limits	MUST	Messages MUST fit within HOPR packet payload limits (see RFC-0004)
Keep-Alive Frequency	SHOULD	`KeepAlive` messages SHOULD be sent periodically to maintain long-lived sessions
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

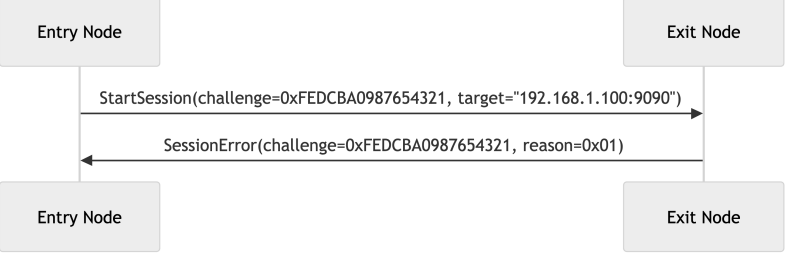
4.10.1 Successful Session Establishment

Complete successful session establishment with immediate keep-alive:



4.10.2 Session Establishment Failure

Session establishment failing due to resource exhaustion:

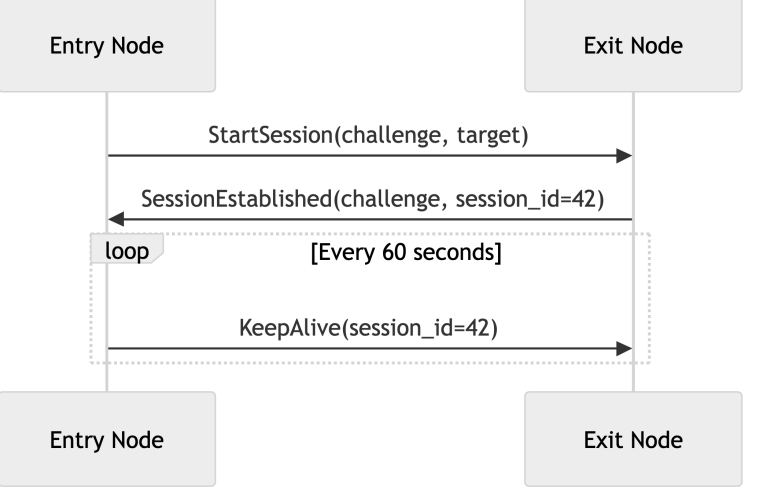


4.10.3 Session Establishment Timeout

Session establishment with no response from exit node, resulting in timeout:

4.10.4 Long-Running Session with Periodic Keep-Alives

Maintaining an established session over time:



5. Design Considerations

5.1 CBOR Encoding

The use of CBOR (Concise Binary Object Representation) [01] for session IDs and session targets provides several advantages:

- Flexible data types: Supports various data types without fixed-size constraints, enabling session IDs and targets to be represented as integers, strings, byte arrays, or structured data.
- Compact binary encoding: More efficient than text-based formats like JSON, reducing packet overhead in the constrained HOPR packet payload.
- Language-agnostic serialisation: Standardised format with implementations available in multiple programming languages, facilitating interoperability.
- Support for complex identifiers: Enables session identifiers to encode additional metadata when needed (e.g., node identifiers, timestamps, or routing hints).

5.2 Challenge-Response Design

The 64-bit challenge field serves multiple purposes in the session start protocol:

- Request-response correlation: Enables the entry node to match `SessionEstablished` or `SessionError` responses to the corresponding `StartSession` request, even when multiple requests are pending simultaneously.
- Protection against replay attacks: When combined with transport-level security, the unpredictable challenge prevents an attacker from replaying a captured `StartSession` message to establish unauthorised sessions.
- Simple state tracking: The challenge allows implementations to maintain minimal state for pending session establishment requests, using the challenge as a key in a hash table or similar data structure.
- Low collision probability: With 2^64 possible values and cryptographically secure random generation, the probability of challenge collisions is negligible even with many concurrent requests.

5.3 Capability Negotiation

The single-byte capability field provides a compact mechanism for protocol negotiation:

- Up to 8 independent flags: Each bit can represent a distinct capability, allowing peers to negotiate multiple features simultaneously.
- Future protocol extensions: As new session features are developed, capability bits can be assigned without changing the message format or breaking existing implementations.
- Backward compatibility: Implementations can safely ignore unrecognised capability bits, allowing newer implementations to interoperate with older ones that don't support new features.
- Minimal overhead: A single byte adds negligible overhead while providing sufficient flexibility for anticipated protocol evolution.

5.4 Transport Independence

The session start protocol is intentionally transport-agnostic, making it suitable for various network environments:

- Packet-based transport: Works over any packet-based transport layer that provides bidirectional communication.
- Designed for HOPR, not limited to it: While optimised for HOPR packets (RFC-0004), the protocol can be used over other transports such as raw UDP, WebSockets, or QUIC.
- No ordering assumptions: The protocol does not require ordered message delivery, making it suitable for unreliable transports.
- No reliability assumptions: The protocol does not depend on reliable delivery; implementations can add timeouts and retransmission logic as needed for their specific transport.

5.5 Error Handling

The protocol provides structured error reporting to enable intelligent failure handling:

- Specific error codes: Well-defined error codes (Unknown Error, No Slots Available, Busy) enable entry nodes to distinguish between different failure scenarios and adjust their behaviour accordingly.
- Challenge correlation: Including the original challenge in error messages ensures that entry nodes can correctly attribute errors to specific requests.

- Graceful resource exhaustion: The “No Slots Available” error allows exit nodes to signal capacity limits without dropping requests silently, enabling entry nodes to try alternative exit nodes.
- Temporary vs. permanent failures: The error code taxonomy distinguishes between temporary failures (Busy) that warrant retry and semi-permanent failures (No Slots Available) that suggest trying a different node.

6. Compatibility

6.1 Version Compatibility

- Version 2 (`0x02`) is the initial version of the session start protocol specified in this document.
- Future versions MUST use different version numbers to distinguish themselves from version 2.
- Implementations MUST reject messages with unknown or unsupported version numbers.
- Version negotiation mechanisms are out of scope for this specification; if needed, they should be addressed in future RFCs.

6.2 Transport Requirements

- The protocol requires a bidirectional communication channel between entry and exit nodes.
- No assumptions are made about message ordering; messages may arrive out of order.
- No assumptions are made about reliability; implementations should add timeout and retransmission logic as appropriate.
- Compatible with any transport that provides packet delivery (e.g., UDP, HOPR packets, QUIC, WebSockets).
- Designed for the HOPR mixnet but not limited to it; the protocol can be deployed over other privacy-preserving or traditional networks.

6.3 Integration with HOPR Session Data Protocol

- The session start protocol establishes sessions that are subsequently used by the session data protocol (RFC-0008) for reliable and unreliable data transmission.
- Session IDs assigned by this protocol are used to identify data sessions in the session data protocol.
- The two protocols operate independently: session start handles handshake and lifecycle, while session data handles message transmission.
- Session establishment MUST complete successfully before data transmission can begin.

7. Security Considerations

7.1 Protocol Security

- The session start protocol provides NO encryption or authentication by itself.
- Security properties (confidentiality, integrity, authenticity) MUST be provided by the underlying transport layer.
- Session IDs SHOULD be unpredictable to prevent session hijacking and enumeration attacks.
- Challenges MUST be generated using cryptographically secure random number generation to prevent prediction and replay attacks.

7.2 Attack Vectors

The following attack vectors exist when the protocol is used without adequate transport-level security:

- Replay attacks: Captured `StartSession` messages can be replayed without additional timestamp or nonce mechanisms. Mitigation requires transport-level encryption and authentication.
- Man-in-the-middle attacks: The protocol alone does not prevent an active attacker from intercepting and modifying messages. Transport-level security is required.
- Information disclosure: Session targets may expose service information (e.g., destination addresses) if not encrypted at the transport layer.

- Resource exhaustion: Attackers can flood exit nodes with excessive session establishment requests, potentially exhausting available session slots. Rate limiting is essential.
- Session hijacking: Predictable session IDs enable attackers to guess valid session identifiers and hijack established sessions. Session IDs MUST be generated unpredictably.

7.3 Mitigation Strategies

Implementations SHOULD employ the following strategies to mitigate security risks:

- Transport-level security: Use HOPR packet encryption (RFC-0004) or other transport-level encryption and authentication mechanisms to protect against replay, man-in-the-middle, and information disclosure attacks.
- Rate limiting: Implement rate limiting for incoming session establishment requests to prevent resource exhaustion attacks. Limits can be per-peer or global.
- Unpredictable session identifiers: Generate session IDs using cryptographically secure random number generators to prevent session hijacking and enumeration.
- Session timeout mechanisms: Implement session timeouts to automatically clean up stale sessions and free resources. Keep-alive messages can be used to maintain active sessions.
- Challenge expiration: Optionally expire challenges after a configurable timeout to limit the window for replay attacks.

8. Future Work

Potential areas for future protocol enhancements include:

- Session parameter renegotiation: Mechanisms to renegotiate session parameters (capabilities, targets) without tearing down and re-establishing the session.
- Performance optimisations: Techniques to reduce session establishment latency for high-frequency session creation scenarios, such as session pooling or 0-RTT establishment.
- Enhanced capability negotiation: More sophisticated capability negotiation mechanisms, including capability versioning and feature discovery.

- Heartbeat and health monitoring: Enhanced keep-alive mechanisms that can carry health status information or quality-of-service metrics.

9. Implementation Notes

9.1 Testing Recommendations

Implementations SHOULD include comprehensive tests covering:

- Session target format variations: Test with various session target formats (IPv4, IPv6, service URIs, edge cases) to ensure correct CBOR encoding and decoding.
- Network failure simulation: Simulate packet loss, delays, and timeouts to verify correct timeout handling and retransmission logic.
- Challenge uniqueness and correlation: Verify that challenges are generated uniquely and that responses are correctly correlated with requests, including handling of duplicate challenges.
- Capability negotiation edge cases: Test capability negotiation with various combinations of set and unset capability bits, including forward and backward compatibility scenarios.
- CBOR encoding correctness: Validate that CBOR encoding and decoding of session IDs and targets is correct and handles all expected data types.
- Error handling: Test all error codes and verify that error messages are correctly generated and handled.

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 the HOPR protocol, a session is identified uniquely via the HOPR session ID. This consists of 10 pseudo-random bytes as a prefix and a 64-bit unsigned integer as a 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 (`0xabcdefabcdefabcdefab`) represents a fixed pseudonym prefix in the HOPR packet protocol (as specified in RFC-0004). The suffix (`123456`) 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 specifies an automatic path discovery mechanism for the HOPR protocol, enabling it to function effectively within dynamic ad hoc peer-to-peer networks. The mechanism allows message senders to remain anonymous while ensuring optimal message delivery by actively probing network nodes to assess compliance with HOPR protocol functionality and detect non-adversarial behaviour. The specification defines complementary breadth-first and depth-first graph traversal algorithms for topology discovery, along with telemetry collection methods to support path selection and quality-of-service (QoS) assessment.

2. Motivation

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

1. Forward path: From sender to destination for unidirectional communication.
2. Return path: From destination back to sender for bidirectional communication, established using Single-Use Reply Blocks (SURBs) as defined in RFC-0004.

The HOPR protocol does not define flow control at the network layer, as this responsibility is delegated to upper protocol layers (see RFC-0008). This design places the

responsibility on each network node to track peer status and network conditions to establish 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 sender anonymity. Consequently, the sender MUST maintain an accurate and current view of the network topology to create effective forward and return path pools. Without topology knowledge, the sender cannot select paths that:

- Have adequate channel capacity and funding
- Provide acceptable latency and throughput
- Avoid unreliable or malicious relay nodes
- Maintain sufficient diversity for anonymity

Relay nodes and destinations also benefit from network discovery to ensure alignment between the incentivisation layer (payment channels) and the network transport layer (physical connectivity).

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 breadth-first and depth-first algorithms and employ them in concert, as exhaustive topology discovery becomes computationally prohibitive as network size increases. The combination of these algorithms enables efficient discovery of immediate peers (breadth-first) and deeper paths (depth-first) while managing resource consumption.

4.2 Network probing

The network discovery algorithms operate under the following assumptions about the network environment:

1. Dynamic topology: The network topology is not static and can change as individual nodes modify peer preferences, open or close payment channels, or go offline. For peers that require a relay for connectivity, the disappearance of the relay can cause topology reconfiguration.
2. Unreliable nodes: Any node in the network can be unreliable due to physical network infrastructure performance limitations, intermittent connectivity, or resource constraints.
3. Malicious nodes: Any node in the network can behave maliciously. Any behaviour resembling malicious activity SHOULD be considered malicious and appropriately flagged for exclusion from path selection.

Given these assumptions, the network probing algorithms for topology discovery employ multiple complementary mechanisms: a breadth-first algorithm (BFA) and a depth-first algorithm (DFA).

Initially, implementations SHALL perform general network discovery using primarily the breadth-first approach to identify immediate peers and build an initial topology view.

Once a statistically sufficient topology is identified to support path randomisation (typically when sufficient peer diversity exists for meaningful path construction), the depth-first approach SHOULD be employed to probe specific topology paths of interest, such as paths through particular relay nodes or to specific exit nodes.

The advantage of combining these approaches is that their results can be used together to identify potentially unreliable or malicious peers more efficiently, while allowing focus on specific peers in the path as static anchors (for QoS requirements, exit node functionality, etc.).

The network topology is modelled as a directed graph structure where nodes perform data relay functionality. Each directed edge in the graph represents a viable connection between two nodes and corresponds to a combination of properties defined by both the physical transport and the HOPR protocol. For an edge to be considered valid, the following properties MUST be present:

1. Payment channel existence: A HOPR payment channel (see RFC-0005) MUST exist from the source node to the destination node of the edge. This channel enables the proof of relay mechanism and provides economic incentives for packet forwarding.

2. Physical connectivity: A physical transport connection MUST exist allowing data transfer between the two nodes. This includes network reachability, NAT traversal (if applicable), and transport protocol compatibility.

While property 1 can be determined from on-chain data in the incentive mechanism (see RFC-0007), property 2 MUST be discovered through active probing on the physical network.

The only exception to property 1 in the HOPR protocol is the final hop (i.e., the connection from the last relay node to the destination), where a payment channel is not required for data delivery since no further relaying occurs.

The network probing mechanism abstracts transport interactions and consists of three core components:

1. Path-generating probing algorithm: Generates paths to probe based on breadth-first or depth-first strategies.
2. Evaluation mechanism: Assesses probe results to determine path viability and node reliability.
3. Retention and slashing mechanism: Maintains path quality information and removes unreliable paths from consideration.

4.2.1 Path-generating probing algorithm

The primary responsibility of the path-generating component is to apply different graph traversal algorithms to generate probe paths that offer insights into 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 probing nature of the traffic from relay nodes. Loopback communication means that the probing node functions as both sender and receiver, with packets traversing a multi-hop path before returning to the origin. Loopback MAY be realized via the session protocol (RFC-0008) or via an equivalent ephemeral mechanism; formal sessions are OPTIONAL for probing traffic.

In this approach, each node in the path is treated as a probed relay node, and each edge

between consecutive relays is treated as a probed connection. While a single probing attempt does not guarantee extraction of all relevant information, when combined with results from multiple probing attempts across different paths, 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 converges too slowly to a usable network topology nor focuses exclusively on small sub-topologies due to computational constraints.

Loopback probing methods:

The following loopback probing methods are defined in terms of hop count:

1. Immediate 0-hop: Directly observe whether an acknowledgement was received from the peer and measure response latency. Probes use indistinguishable payloads (data indistinguishable from application data via padding and AEAD encryption). Acknowledgements are produced by the destination and authenticated before acceptance. This method is suitable for next-hop telemetry (see Section 4.3.1).
2. 1-hop to self: Perform first-order checks of immediate peer connections by sending a packet through a single peer and back to self. Functionally equivalent to 0-hop but executed in a manner that conceals probing activity from the peer (since the peer cannot distinguish loopback traffic from regular forwarding).
3. 2-hop to self: Check second-order communication paths by traversing two hops before returning. This method MAY replace some 3-hop paths to reduce total probing overhead.
4. 3-hop to self: Perform full bidirectional path probing for 1-hop connections, traversing three hops (out, relay, and back). This represents a complete anonymising path in the HOPR network.

Discovery algorithm operations:

The discovery algorithm SHALL operate in complementary modes: breadth-first and depth-first. The basic operational steps are:

1. Discover immediate peers: Use 0-hop or 1-hop probes to identify directly connected peers and assess their basic connectivity.

2. Generate n-hop paths: Generate paths for multi-hop connections using referential probing with low frequency to explore deeper network topology.
3. Prepopulate path cache: For sessions (RFC-0008), prepopulate the path cache from sufficiently recent historical knowledge of successful paths to reduce session establishment latency.
4. Perform high-frequency probing: Execute higher frequency probing checks on paths of interest to maintain up-to-date viability information.

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. In the context of network probing, BFS MUST start at the probing node and explore neighbouring nodes at the current depth level before moving on to nodes at the next depth level.

The breadth-first algorithm (BFA) SHOULD primarily be used for initial network topology discovery with the goal of identifying a statistically significant minimum number of peers with desired QoS and connectivity properties. This approach provides rapid discovery of the immediate network neighbourhood before exploring deeper paths.

This algorithm SHOULD be primarily implemented using 1-hop to self probes to efficiently discover immediate peers while concealing probing activity.

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

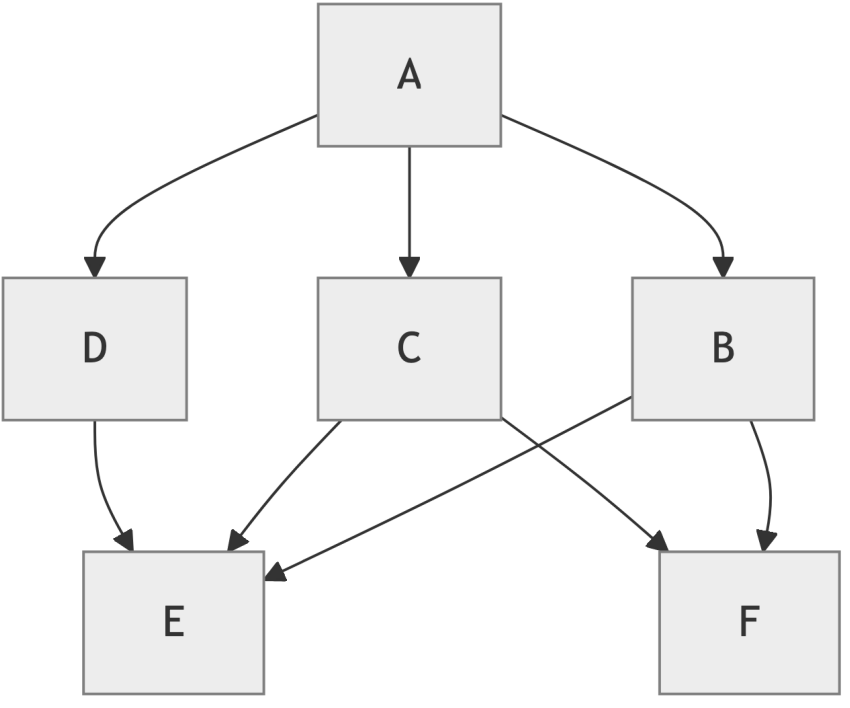

A
D
C
B
E
F

Fig. 1: Network topology for BFA-inspired network probing

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

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

Once the immediate vicinity is probed and a basic topology map is established, a larger share of the probing traffic SHOULD transition to using the depth-first algorithm, phasing the BFA into a smaller proportion of overall probing activity.

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. In the context of network probing, DFS MUST start at the probing node and explore each branch of the graph deeply before moving to another branch.

DFS is particularly useful for pathfinding and exploring specific routes through the network to assess end-to-end path viability.

This algorithm SHOULD be primarily implemented using n-hop to self probes, where $n>1$ and $n \leq$ `MAX_HOPR_SUPPORTED_PATH_LENGTH` (a network parameter defined in RFC-0004). Each edge SHOULD be probed as soon as feasible, but not at the expense of other edges in the topology (i.e., probing should be distributed across the topology). The value of n SHOULD be chosen randomly to prevent predictable probing patterns, but MUST conform with the minimum requirement for edge traversal (typically $n \geq 2$ for meaningful path diversity assessment).

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

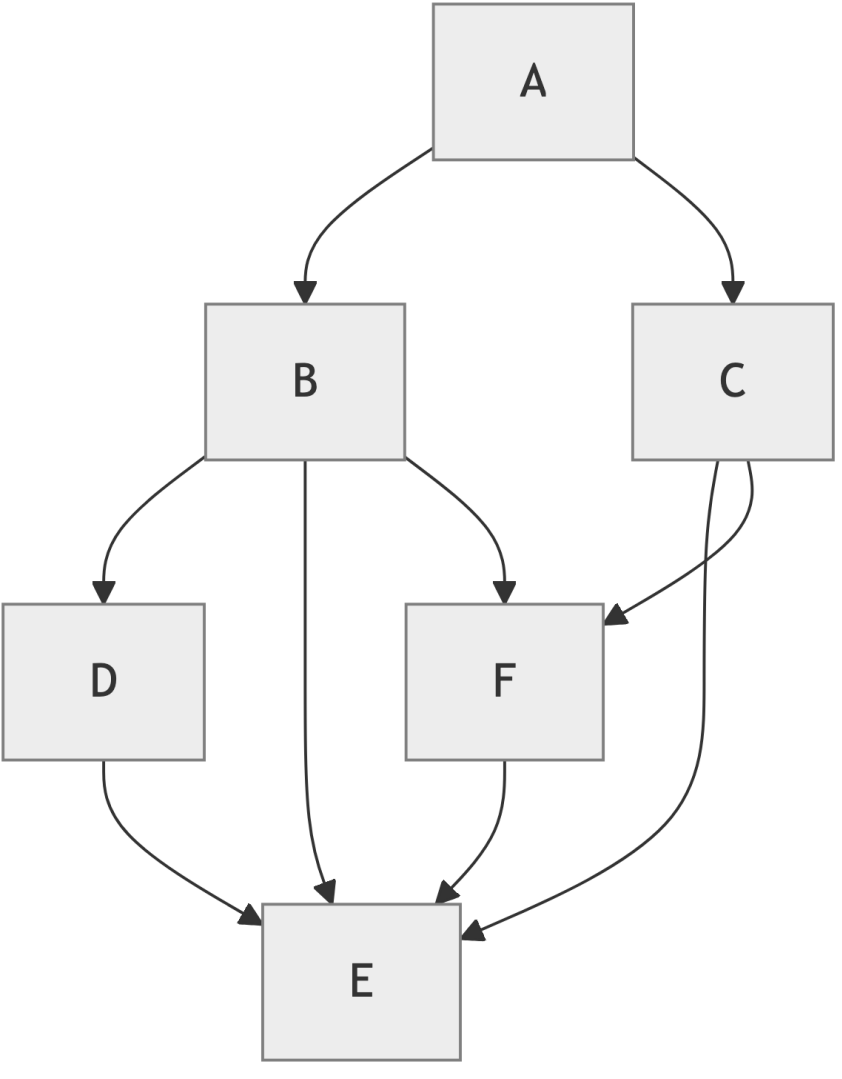

A
B
C
D
F
E

Fig. 2: Network topology for DFA-inspired network probing

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

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

These deep probes explore specific paths through the network and collect end-to-end path metrics.

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. By combining telemetry from breadth-first and depth-first probes, it is possible to derive statistical information about individual nodes and edges in the topology.

Example: Assume the following average path latencies are observed:

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

From these measurements, it is possible to estimate the average latency contribution of node F (and the edges involving F) as:

```
(A -> B -> F -> A) - (A -> B -> A) = 545ms - 421ms = 124ms
```

This difference represents the additional latency introduced by traversing through node F. Accounting for artificial mixer delays that introduce additional anonymity, repeated observations of this value averaged over longer time windows would provide an expected

latency contribution for node F. By aggregating such measurements across multiple paths, implementations can build a statistical model of individual node performance characteristics.

4.2.2 Evaluation mechanism

The evaluation mechanism processes probe results to assess path and node viability. The mechanism SHOULD maintain short-term memory of recent probe results and apply balanced scoring that equally rewards probe successes and penalises probe failures. This approach ensures that recent network conditions are given appropriate weight while preventing both overly optimistic and overly pessimistic assessments.

Implementations MAY use various evaluation strategies, such as exponentially weighted moving averages, sliding time windows, or Bayesian estimation, provided they meet the requirement of balanced success/failure treatment.

4.2.3 Retention and slashing mechanism

Nodes MAY implement a slashing mechanism based on failed probes to prevent using unreliable relay nodes in non-probing (production) communication, thereby avoiding dropped messages and improving overall communication reliability.

The slashing mechanism operates by temporarily or permanently removing nodes or paths from the usable path pool based on probe failure patterns. Implementations SHOULD consider:

- Failure threshold: The number or percentage of consecutive or recent failed probes that trigger slashing.
- Slashing duration: Whether nodes are removed permanently or temporarily (with exponential backoff for repeated failures).
- Recovery mechanism: Conditions under which previously slashed nodes can be re-evaluated and restored to the usable pool.

Slashing decisions SHOULD be made locally by each node based on its own probe observations, without coordination with other nodes.

4.2.4 Throughput considerations

Paths SHOULD be selected and used by the discovery mechanism in a manner that supports sustained throughput (i.e., the maximum achievable packet rate). Path selection SHOULD consider:

- Load balancing over paths: Distribute traffic across multiple paths based on the minimum stake (channel balance) on each path, ensuring paths with higher capacity receive proportionally more traffic.
- Measured throughput: Use actual throughput as observed in real traffic (not just probes) to refine path selection and avoid paths that perform poorly under load.

These considerations ensure that path discovery supports not only path viability assessment but also efficient utilisation of available network capacity.

4.3 Telemetry

Telemetry refers to the data and metadata collected by the probing mechanism about traversed transport paths. Telemetry enables nodes to assess path quality, detect failures, and make informed path selection decisions. This section defines the types of telemetry collected and their purposes.

4.3.1 Next-hop telemetry

Next-hop telemetry, also referred to as per-path telemetry (PPT), MUST be collected for each direct peer connection. This telemetry SHOULD be used to inform channel opening and closing strategies that optimise first-hop connections from the current node.

The PPT SHOULD provide basic evaluation of the transport channel, both in the presence and absence of an open on-chain payment channel. At a minimum, the PPT MUST provide the following observations for each 0-hop connection (as specified in RFC-0004):

1. Latency: Duration between sending a message and receiving the corresponding acknowledgement. This measures round-trip time to the immediate peer.
2. Packet drop rate: Track the ratio of missing acknowledgements to expected acknowledgements for messages sent on the channel. This indicates the reliability of the transport connection.

The PPT MAY be utilised by other mechanisms as an information source, such as channel management strategies that optimise the outgoing network topology by opening channels to high-performance peers and closing channels to unreliable peers.

4.3.2 Non-probing telemetry

Non-probing telemetry refers to telemetry collected from production (non-probe) traffic. This telemetry MAY track the same metrics as next-hop telemetry with the goal of adding more relevant channel information for 0-hop connections.

Each outgoing message SHOULD be tracked for the same set of telemetry as the PPT (latency, packet drop rate) on a per-message basis. This provides real-world performance data that complements probe-based observations and can reveal issues that only appear under actual traffic load.

4.3.3 Probing telemetry

Probing telemetry refers to structured data embedded within probe messages to facilitate path identification and performance measurement. All multi-byte integer fields MUST be transmitted in network byte order (big-endian) to ensure consistent interpretation across different architectures.

The probing message payload contains the following fields:

- Counter (8 bytes, `uint64`): An iterating counter used to verify the mixing property over a path and detect packet reordering or duplication. The counter increments for each probe sent.

- Path ID (8 bytes, `uint64`): A unique identifier for a single specific path in the graph, enabling attribution of probe results to the correct path.
- Timestamp (8 bytes, 64-bit `UNIXtimeinnanoseconds`): The timestamp of packet creation, used for end-to-end latency observations. The timestamp is recorded when the probe is generated and compared against the received timestamp upon loopback completion.

Wire format:

```
+-------------+------------+------------+
|   Counter   |   PathId   |  Timestamp |
|     8B      |     8B     |     8B     |
+-------------+------------+------------+
```

The total probing message payload size is 24 bytes.

4.4 Component placement

The network probing functionality, with the exception of the next-hop telemetry (PPT) mechanism, MUST be implemented using HOPR loopback communication to preserve anonymity and prevent relay nodes from distinguishing probe traffic from production traffic.

Implementation requirements:

- Remove channel graph quality: The concept of channel graph quality based on network observations SHALL be removed from implementations. Only on-chain channel information (stake, balance, status) SHALL be retained for path viability determination.
- Continuous probe generation: Implementations MUST provide processes to generate a low-rate continuous stream of network path probes to maintain up-to-date topology information.
- Session-specific paths: Implementations MUST generate session-specific paths for session path selection to provide cover traffic and obfuscate the real communication patterns (see RFC-0008).

- Path graph system: A new path graph system SHALL be derived from probe results, representing discovered topology and path quality metrics.
- Path caching: Paths SHALL be cached for a configurable minimum time window to amortise the cost of path discovery and reduce probe frequency.
- Session metrics incorporation: Session-level telemetry SHALL incorporate:
  - Session-level performance metrics (throughput, latency, packet loss)
  - Session-specific path probing data to refine path selection during active sessions
  - Session-derived cover traffic for exploratory network traversal without dedicated probe overhead

5. Design considerations

The automatic path discovery mechanism is designed to enable each sender to:

1. Identify sufficient network nodes: Discover a sufficiently large number of network nodes to ensure privacy through path pool diversity. A larger discovered topology enables greater path randomisation and reduces the risk of traffic analysis.
2. Detect problematic nodes: Identify unstable, malicious, or adversarial nodes through probe failure patterns and exclude them from path selection.
3. Estimate QoS metrics: Establish basic propagation metrics for quality-of-service (QoS) estimation, including latency, throughput, and reliability.

With these capabilities, the sender can construct a functional representation of the network topology, state, and constraints, enabling optimal selection and exclusion of message propagation paths.

Key design principles:

- Indistinguishability: Multi-hop probing traffic and measurement packets MUST be indistinguishable from ordinary traffic to ensure accurate recording of network node propagation characteristics. If relay nodes could distinguish probes from production traffic, they might handle them differently (e.g., prioritise or deprioritise probes), leading to inaccurate measurements.
- Adaptive mechanisms: Due to the dynamic nature of decentralised peer-to-peer networks, senders SHOULD employ adaptive mechanisms for establishing and

maintaining topological awareness. Static path selection would quickly become outdated as nodes join, leave, or change behaviour.

- Continuous probing: For both unidirectional and bidirectional communication to adapt to changing network conditions, senders MUST actively probe the network in a continuous manner, with probe frequency balanced against economic feasibility.
- Economic feasibility: Measurement traffic SHOULD adhere to economic feasibility constraints, i.e., it SHOULD be proportional to actual message traffic. Excessive probing wastes network resources and incurs unnecessary costs. Probe traffic MAY be incorporated as part of cover traffic (see RFC-0008) to serve dual purposes.
- No telemetry sharing: Any measurements obtained from probing traffic SHOULD be node-specific and MUST NOT be subject to data or topology exchange with other nodes. Sharing telemetry could compromise anonymity by revealing which nodes are being probed and what paths are being considered.

Telemetry requirements:

The collected telemetry for measured paths:

- MUST contain path passability data indicating whether paths are traversable by single or multiple messages.
- MAY include additional information such as latency, packet loss rate, and throughput, transmitted as message content in the probing payload.

Direct peer probing:

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 (next-hop telemetry, Section 4.3.1) exists that operates outside the anonymity requirement.

The 0-hop and 1-hop probing mechanisms MAY NOT fully comply with the anonymity requirement, since they:

1. Mimic the 0-hop session (RFC-0008), which does not benefit from multi-hop relaying mechanisms and may reveal the probing node to the immediate peer.
2. Could be used as a first layer for relay nodes to discover viable candidates for future channel openings, which is acceptable as it does not compromise sender anonymity in multi-hop paths.

The network probing mechanism SHALL utilise graph-based algorithms (breadth-first and depth-first) to efficiently discover and maintain network topology information while managing computational and economic costs.

6. Compatibility

The automatic path discovery mechanism is a local node feature that affects only the implementing node. Changes to path discovery algorithms or telemetry collection MAY be modified without impacting overall network operation, as long as the node continues to generate valid HOPR packets and respects protocol semantics.

The network probing mechanism is compatible with the loopback session mechanism defined in RFC-0008, allowing probes to leverage session infrastructure when available.

7. Security considerations

The probing traffic consumes both physical resources (bandwidth, compute) and economic value (payment channel balances) at various levels of the HOPR protocol stack. This resource consumption introduces several security considerations:

1. Resource depletion attacks: In highly volatile networks, adversarial behaviour may cause excessive resource expenditure through probe failures or artificial network instability. Attackers could deliberately fail probes to force nodes to increase probe frequency, potentially enabling resource depletion attacks. Implementations SHOULD implement rate limiting and adaptive probe frequency to mitigate this risk.
2. Denial-of-service via PPT: The next-hop telemetry (PPT) mechanism, which operates at 0-hop without full anonymity protection, MAY serve as an attack vector for denial-of-service (DoS) attempts. Attackers could flood a node with 0-hop telemetry requests to exhaust resources. Implementations SHOULD apply rate limiting to PPT requests.
3. Traffic analysis: Although probes are designed to be indistinguishable from production traffic, statistical analysis of traffic patterns might reveal probing behaviour if probe generation follows predictable patterns. Implementations SHOULD randomise probe timing and path selection to prevent traffic analysis.
4. Mitigation strategies: Nodes MAY implement any reasonable security risk mitigation strategy, including but not limited to:

- Rate limiting probe generation and reception
- Adaptive probe frequency based on network conditions
- Slashing mechanisms to exclude misbehaving nodes
- Resource quotas for probe traffic

8. Drawbacks

The network probing mechanism has several inherent limitations:

1. Resource consumption: Probing activity consumes network bandwidth, computational resources, and payment channel balances. Implementations MUST carefully balance probing and data transmission activities to maintain reasonable resource utilisation ratios. Excessive probing wastes resources; insufficient probing leads to outdated topology information.
2. Scalability limitations: Complete real-time probing of large networks is computationally prohibitive. As network size increases, the number of possible paths grows combinatorially, making exhaustive probing infeasible. Algorithms SHOULD operate within bounded subnetworks where they can provide reasonable network visibility guarantees within acceptable resource constraints.
3. Bootstrap requirements: Prior knowledge of target nodes (e.g., through external discovery mechanisms or bootstrap node lists) is advantageous to minimise initialisation time before establishing a sufficient network view for informed path selection. Nodes joining a network without any peer knowledge face a cold-start problem.

9. Alternatives

No known alternative mechanisms exist that simultaneously:

- Preserve sender anonymity by preventing relay nodes from distinguishing probes
- Maintain trustless properties without requiring nodes to share topology information
- Consolidate probing control under the communication source to enable informed path selection

Alternative approaches such as centralised topology databases or distributed topology sharing protocols would compromise either anonymity or trustlessness, making them

unsuitable for the HOPR protocol's threat model.

10. Unresolved questions

None.

11. Future work

Future development of the automatic path discovery mechanism SHOULD focus on the following areas:

1. Extended telemetry collection: Improve the ability to collect additional network metrics by extending the data payload transmitted along the loopback path. Additional metrics might include jitter, packet reordering, or relay node load indicators.
2. Advanced path generation strategies: Develop new path-generating strategies that enable statistical inference of information from path section overlaps. For example, using matrix completion techniques or Bayesian inference to estimate properties of un-probed edges from probed path combinations.
3. Enhanced evaluation mechanisms: Improve metric evaluation mechanisms with more sophisticated scoring functions, machine learning-based anomaly detection, or adaptive weighting schemes that respond to network conditions.
4. Formal slashing logic: Define a formal slashing mechanism with equation-based logic that specifies precise conditions for node removal, recovery, and reputation scoring.

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 HOPR application layer protocol, a thin multiplexing layer that sits between the HOPR packet protocol RFC-0004 and higher-level protocols such as the session protocol RFC-0008 or session start protocol RFC-0009. The application protocol enables HOPR nodes to distinguish between different upper-layer protocols running over the same packet transport, similar to how TCP and UDP use port numbers to multiplex multiple applications over IP.

The protocol consists of a simple tagging mechanism using 64-bit identifiers, allowing up to 2^61 distinct protocol types whilst reserving space for future extensions.

2. Motivation

The HOPR network supports multiple upper-layer protocols that serve different purposes, including session management, path discovery, and application data transport. Without a standardised method to distinguish between these protocols, nodes would be unable to properly route and handle packets intended for specific purposes. The application layer protocol solves this by providing a lightweight tagging mechanism similar to port numbers in TCP/UDP, enabling protocol multiplexing over the fixed-size HOPR packet format.

Additionally, the protocol provides a bidirectional signalling mechanism through flag bits, allowing the packet layer and upper layers to exchange control information (such as SURB availability notifications) without requiring separate packet types.

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 multiple upper-layer protocols that serve different purposes. Examples include session management (RFC-0008), session establishment (RFC-0009), and path discovery (RFC-0010). The application layer protocol described in this RFC creates a thin multiplexing layer between the HOPR packet protocol (RFC-0004) and these upper-layer protocols.

The application layer protocol serves two primary purposes:

1. Protocol multiplexing: enabling a node to distinguish between different upper-layer protocols and dispatch packets to the appropriate protocol handlers based on protocol tags
2. Inter-layer signalling: providing a bidirectional communication channel for control signals between the HOPR packet protocol and upper-layer protocols through flag bits (e.g., SURB availability notifications)

5. Specification

The application layer protocol acts as a wrapper for arbitrary upper-layer `data`, adding a `Tag` that identifies the upper-layer protocol type:

```
ApplicationData \{
    tag: Tag,           // 64-bit protocol identifier
    data: [u8; <length>]  // Variable-length protocol data
    flags: u8           // Control flags for inter-layer signalling
\}
```

Tag structure:

The `Tag` MUST be represented by 64 bits, with the following structure:

- The 3 most significant bits MUST always be set to 0 in the current version (reserved for future use)
- The remaining 61 bits represent a unique identifier for the upper-layer protocol

This design provides 2^61 (approximately 2.3 × 10^18) possible protocol identifiers whilst reserving space for future protocol versioning or extensions.

Protocol tag allocation:

The `Tag` space is divided into ranges for different purposes:

- `0x0000000000000000`: reserved for the probing protocol (path discovery, see RFC-0010)
- `0x0000000000000001`: reserved for the session start protocol (session establishment, see RFC-0009)
- `0x0000000000000002` – `0x000000000000000d`: available for user-defined protocols (12 tags)
- `0x000000000000000e`: catch-all for unknown or experimental protocols
- `0x000000000000000f` – `0x1fffffffffffffff`: reserved for the session protocol (approximately 2^61 - 15 tags, see RFC-0008)

This allocation ensures that core HOPR protocols have well-known identifiers whilst providing space for custom protocols and future extensions.

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 at most the size of the HOPR protocol packet, the upper layer protocol SHALL be responsible for the framing
3. `field`: MUST NOT be serialised, 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 signalling to the HOPR packet protocol. Conversely, the HOPR packet protocol MAY use the 4 least significant bits in `flags` to pass arbitrary signalling to the upper-layer protocol.

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

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.