Independent Stream
Internet-Draft
Intended status: Informational
Expires: 18 September 2025

M. Schanzenbach
Fraunhofer AISEC
C. Grothoff
Berner Fachhochschule
B. Fix
GNUnet e.V.

17 March 2025

The R5N Distributed Hash Table
draft-schanzen-r5n-07

Abstract

This document contains the R^5N DHT technical specification. R^5N is a secure distributed hash table (DHT) routing algorithm and data structure for decentralized applications. It features an open peer-to-peer overlay routing mechanism which supports ad-hoc permissionless participation and support for topologies in restricted-route environments. Optionally, the paths data takes through the overlay can be recorded, allowing decentralized applications to use the DHT to discover routes.

This document defines the normative wire format of protocol messages, routing algorithms, cryptographic routines and security considerations for use by implementers.

This specification was developed outside the IETF and does not have IETF consensus. It is published here to guide implementation of R^5N and to ensure interoperability among implementations including the pre-existing GNUnet implementation.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 18 September 2025.

Table of Contents

1. Introduction 3
1.1. Requirements Notation 4
1.2. System Model 4
1.3. Security Model 4
2. Terminology 5
3. Motivation 7
3.1. Restricted-route topologies 7
3.2. Key differences to RELOAD 8
4. Overview 9
5. Underlay 10
6. Routing 12
6.1. Routing Table 13
6.2. Peer Discovery 14
6.3. Peer Bloom Filter 15
6.4. Routing Functions 16
6.5. Pending Table 18
7. Message Processing 19
7.1. Message components 19
7.1.1. Flags 19
7.1.2. Path 20
7.1.3. Path Element 21
7.2. HelloMessage 27
7.2.1. Wire Format 27
7.2.2. Processing 28
7.3. PutMessage 28
7.3.1. Wire Format 29
7.3.2. Processing 31
7.4. GetMessage 33
7.4.1. Wire Format 33
7.4.2. Result Filter 35
7.4.3. Processing 35
7.5. ResultMessage 37

7.5.1. Wire Format 37
7.5.2. Processing 39
8. Blocks 41
8.1. Block Operations 41
8.2. HELLO Blocks 43
8.3. Persistence 47
8.3.1. Approximate Search Considerations 48
8.3.2. Caching Strategy Considerations 48
9. Security Considerations 49
9.1. Disjoint Underlay or Application Protocol Support 49
9.2. Approximate Result Filtering 49
9.3. Access Control 50
9.4. Block-level confidentiality and privacy 50
9.5. Protocol extensions and cryptographic agility 50
9.6. Availability versus security tradeoffs in routing table evictions 50
10. IANA Considerations 51
11. GANA Considerations 51
11.1. Block Type Registry 51
11.2. GNUnet URI Schema Subregistry 52
11.3. GNUnet Signature Purpose Registry 53
11.4. GNUnet Message Type Registry 53
12. Implementation and deployment status 53
13. Normative References 53
14. Informative References 55
Appendix A. Bloom filters in R^5N 56
Appendix B. Overlay Operations 57
B.1. GET operation 57
B.2. PUT operation 59
Appendix C. HELLO URLs 59
Authors' Addresses 60

1. Introduction

This specification describes the protocol of R^5N. R^5N is a Distributed Hash Table (DHT). The name is an acronym for "randomized recursive routing for restricted-route networks" and its first academic description can be found in [R5N].

DHTs are a key data structure for the construction of decentralized applications and generally provide a robust and efficient means to distribute the storage and retrieval of key-value pairs.

The core idea behind R^5N is to combine a randomized routing algorithm with an efficient, deterministic closest-peer algorithm. This allows us to construct an algorithm that is able to escape and circumvent restricted route environments while at the same time allow for a logarithmically bounded routing complexity.

R^5N also includes advanced features like recording the path a key-value pair took through the network, response filters and on-path application-specific data validation.

This document defines the normative wire format of peer-to-peer messages, routing algorithms, cryptographic routines and security considerations for use by implementors.

1.1. Requirements Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. System Model

DHTs usually operate as overlay networks consisting of peers communicating over the existing Internet. Hence canonical DHT designs often assume that the IP protocol provides the peers of the overlay with unrestricted end-to-end pairwise connectivity. However, in practice firewalls and network address translation (NAT) [RFC2663] make it difficult for peers operating on consumer end-devices to directly communicate, especially in the absence of core network infrastructure enabling NAT traversal via protocols such as interactive connectivity establishment (ICE) [RFC5245].

Furthermore, not all peer-to-peer networks consistently operate over the Internet, such as mobile ad-hoc networks (MANETs). While routing protocols have been designed for such networks ([RFC3561]) these generally have issues with security in the presence of malicious participants, as they vulnerable to impersonation attacks. The usual solution to these issues is to assert that the entire MANET is a closed network and to require authentication on all control messages. In contrast, the system model for R^5N is that of an open network without any kind of authorities that could restrict access only to trusted participants.

1.3. Security Model

We assume that the network is open and thus a fraction of the participating peers is malicious. Malicious peers may create, alter, delay or drop messages. We also assume that an adversary can control (or fake) many peers [Sybil], thus any kind of voting or punishment of malicious peers would be rather pointless.

Honest peers are expected to establish and maintain many connections. We assume that as a result the adversary is generally unable to prevent honest peers from maintaining a sufficient number of direct connections with other honest peers to achieve acceptable performance. As the number of malicious peers and their connections increases, performance of the system should gracefully degrade, and only collapse for peers that an adversary has fully isolated from the benign network.

The main security objectives are to provide honest nodes correct results and to limit the propagation of invalid data. Invalid data includes both invalid key-value pairs as well as invalid routing path data if such routing meta-data is present. While malicious nodes may make up arbitrary key-value pairs and paths within the adversary's domain, invalid key-value pairs are ideally discarded at the first honest node, and path data honestly state entry- and exit-points from the honest network into the subset of malicious nodes.

Malicious nodes may attempt to exhaust the storage capacity of honest nodes by distributing well-formed (but possibly otherwise useless) application data. We assume that storage space is relatively cheap compared to bandwidth and that honest nodes also frequently re-publish the useful data that they publish. As a result, an adversary may reduce the effectiveness and longevity of data cached in the DHT, but is assumed to not be able to effectively prevent publication and retrieval of application data by honest nodes.

2. Terminology

Address An _address_ is a UTF-8 [RFC3629] string which can be used to address a _peer_ through the Underlay (Section 5). The format of an address is not enforced by this specification, but it is expected that in most cases the address is a URI [RFC3986].

Applications _Applications_ are higher-layer components which directly use the _Core Operations_. Possible _applications_ include the GNU Name System [RFC9498] and the GNUnet Confidential Ad-hoc Decentralized End-to-End Transport (CADET) [cadet].

Core Operations The _Core Operations_ provide an interface to the core operations of the DHT overlay to _applications_. This includes storing _blocks_ in the DHT and retrieving _blocks_ from the DHT.

Block Variable-size unit of payload stored in the DHT under a _key_. In the context of "key-value stores" this refers to "value" stored under a _key_.

Block Storage The _block storage_ component is used to persist and manage _blocks_ stored by _peers_. It includes logic for enforcing storage quotas, caching strategies and block validation.

Block Type A unique 32-bit value identifying the data format of a _block_. _Block types_ are public and applications that require application-specific block payloads are expected to register one or more block types in the GANA Block-Type registry (Section 11.1) and provide a specification of the associated block operations (Section 8.1) to implementors of R^5N.

Bootstrapping _Bootstrapping_ is the process of establishing a connection to the peer-to-peer network. It requires an initial, non-empty set of reachable _peers_ and corresponding _addresses_ supported by the implementation to connect to.

Initiator The _peer_ that initially creates and sends a DHT protocol message (Section 7.2, Section 7.3, Section 7.4, Section 7.5).

HELLO block A HELLO block is a type of _block_ that is used to store and retrieve _addresses_ of a _peer_. It is used by the peer discovery mechanism in Section 6.2.

HELLO URL HELLO URLs are HELLO blocks represented as URLs. They are used for out-of-band exchanges of _peer_ _addresses_ and for signalling address updates to _neighbours_. Implementation details of HELLO URLs and examples are found in Appendix C.

Key 512-bit identifier of a location in the DHT. Multiple blocks can be stored under the same _key_. A _peer identity_ is also a key. In the context of "key-value stores" this refers to "key" under which _blocks_ are stored.

Message Processing The _message processing_ component of the DHT implementation processes requests from and generates responses to _applications_ and the _underlay interface_.

Neighbor A neighbor is a _peer_ which is directly able to communicate with our _peer_ via the _underlay interface_.

Peer A host that is participating in the overlay by running an implementation of the DHT protocol. Each participating host is responsible for holding some portion of the data that has been stored in the overlay, and they are responsible for routing messages on behalf of other _peers_ as needed by the _routing algorithm_.

Peer Identity The _peer identity_ is the identifier used on the

overlay to identify a _peer_. It is a SHA-512 hash of the _peer public key_.

Peer Public Key The _peer public key_ is the key used to authenticate a _peer_ in the underlay.

Routing The _routing_ component includes the routing table as well as routing and _peer_ selection logic. It facilitates the R^5N routing algorithm with required data structures and algorithms.

Underlay Interface The _underlay interface_ is an abstraction layer on top of the supported links of a _peer_. Peers may be linked by a variety of different transports, including "classical" protocols such as TCP, UDP and TLS or higher-layer protocols such as GNUnet, I2P or Tor.

3. Motivation

3.1. Restricted-route topologies

Restricted-route topologies emerge when a connected underlay topology prevents (or restricts) direct connections between some of the nodes. This commonly occurs through the use of NAT ([RFC2663]). Nodes operated behind a NAT cause common DHT routing algorithms such as Kademlia [Kademlia] to exhibit degraded performance or even to fail. While excluding such nodes is an option, this limits load distribution and is ineffective for some networks, such as MANETs.

In general, nodes may not be mutually reachable (for example due to a firewall or NAT) despite being "neighbours" according to the routing table construction algorithm of a particular DHT. For example, Kademlia uses the XOR metric and would generally connect nodes that have peer identities with a small XOR distance. However, the XOR distance between (basically randomly assigned) peer identities is completely divorced from the ability of the nodes to directly communicate. DHTs usually use greedy routing to store data at the peer(s) closest to the key. In cases where a DHT cannot connect peers according to the construction rules of its routing algorithm, the topology may end up with multiple (local) minima for a given key. Using canonical greedy routing from a particular fixed location in the network, a node may then only be able to publish and retrieve data in the proximity of its local minima.

R^5N addresses this problem by prepending a random walk before a classical, deterministic XOR-based routing algorithm is employed. The optimal number of random hops taken is equal to the mixing time of the graph. The mixing time for various graphs is well known; for small-world networks [Smallworld], the mixing time has been shown to be around O(log n) where n is the number of nodes in the network [Smallworldmix].

Thus, if the network exhibits the properties of a small world topology [Smallworld], a random walk of length O(log n) will cause the algorithm to land on a random node in the network. Consequently, the deterministic part of the algorithm will encounter a random local minimum. It is then possible to repeat this process in order to store or retrieve data in the context of all or at least multiple local minima. The ideal length of the random walk and the number of repetitions expected to cover all local minima depends on the network topology. Our design assumes that the benign subset of the network forms a small-world topology [Smallworld] and then obtains an estimate of the current number of nodes n in the network and then uses log n for the actual length of the random walk.

3.2. Key differences to RELOAD

[RFC6940] specifies the RELOAD DHT. The R^5N DHT described in this document differs from RELOAD in its objectives and thus in its design. The authors of RELOAD make the case that P2P networks are often established among a set of peers that do not trust each other. It addresses this issue by requiring that node identifiers are either assigned by a central authority, or self-issued in the case of closed networks. In other words, by enforcing the P2P network to be established among a set of _trusted_ peers. This misses the point that this openness is a core requirement of efficient and useful DHTs as they serve a fundamental part in a decentralized network infrastructure. R^5N, by contrast, is intended for open overlay networks, and thus does not include a central enrollment server to certify participants and does not limit participation in another way. As participants could be malicious, R^5N includes on-path customizable key-value validation to delete malformed data and path randomiziation to help evade malicious peers. R^5N also expects to perform over a network where not every peer can communicate with every other peer, and where thus its route discovery feature provides utility to higher-level applications. As a result, both the features and the security properties of RELOAD and R^5N are different, except in that both allow storing and retrieving key-value pairs.

4. Overview

In R^5N peers provide to their applications the two fundamental core operations of any DHT:

* PUT: This operation stores a block under a key on one or more peers with the goal of making the block availiable for queries using the GET operation. In the classical definition of a dictionary interface, this operation would be called "insert".

* GET: This operation queries the network of peers for any number of blocks previously stored under or near a key. In the classical definition of a dictionary interface, this operation would be called "find".

An example for possible semantics of the above operations provided as an API to applications by an implementation are outlined in Appendix B.

A peer does not necessarily need to expose the above operations to applications, but it commonly will. A peer that does not expose the above operations could be a host purely used for bootstrapping, routing or supporting the overlay network with resources.

Similarly, there could be hosts on the network that participate in the DHT but do not route traffic or store data. Examples for such hosts would be mobile devices with limited bandwidth, battery and storage capacity. Such hosts may be used to run applications that use the DHT. However, we will not refer to such hosts as peers.

In a trivial scenario where there is only one peer (on the local host), R^5N operates similarly to a dictionary data structure. However, the default use case is one where nodes communicate directly and indirectly in order to realize a distributed storage mechanism. This communication requires a lower-level peer addressing and message transport mechanism such as TCP/IP. R^5N is agnostic to the underlying transport protocol which is why this document defines a common addressing and messaging interface in Section 5. The interface provided by this underlay is used across the specification of the R^5N protocol. It also serves as a set of requirements of possible transport mechanisms that can be used to implement R^5N with. That being said, common transport protocols such as TCP/IP or UDP/IP and their interfaces are suitable R^5N underlays and are used as such by existing implementations.

Specifics about the protocols of the underlays implementing the underlay interface or the applications using the DHT are out of the scope of this document.

To establish an initial connection to a network of R^5N peers, at least one initial, addressable peer is required as part of the bootstrapping process. Further peers, including neighbors, are then learned via a peer discovery process as defined in Section 6.2.

Across this document, the functional components of an R^5N implementation are divided into routing (Section 6), message processing (Section 7) and block storage (Section 8). Figure 1 illustrates the architectural overview of R^5N.

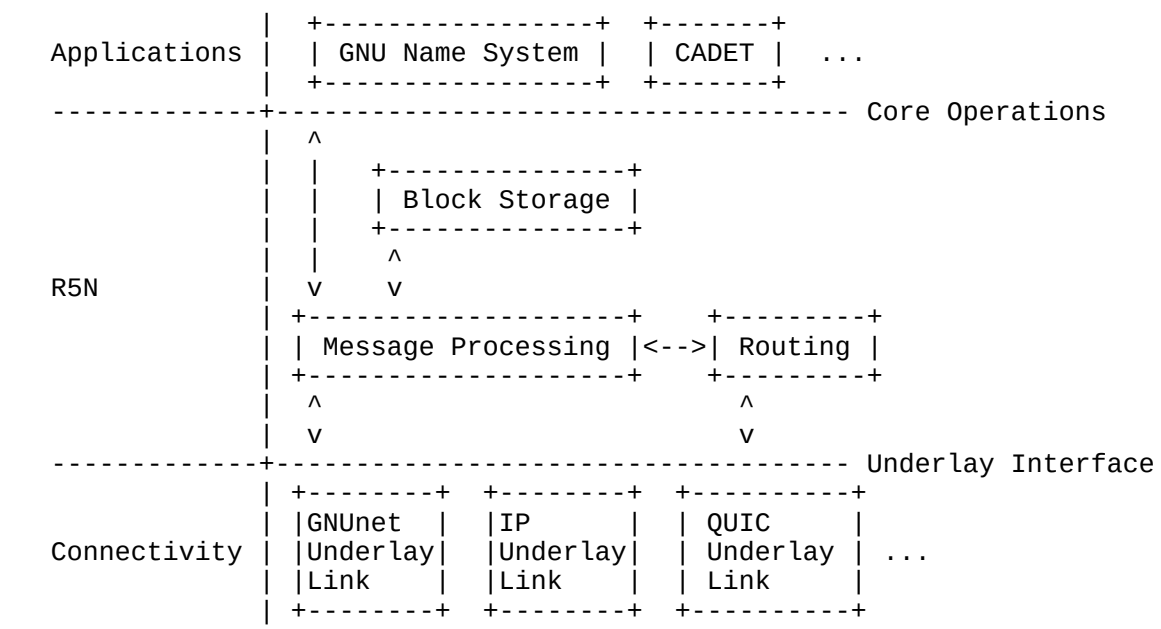


Figure 1: The R5N architecture.

5. Underlay

A peer MUST support one or more underlay protocols. Peers supporting multiple underlays effectively create a bridge between different networks. How peers are addressed in a specific underlay is out of scope of this document. For example, a peer may have a TCP/IP address, or expose a QUIC endpoint, or both. While the specific addressing options and mechanisms are out of scope for this document, it is necessary to define a universal addressing format in order to facilitate the distribution of address information to other peers in the DHT overlay. This standardized format is the HELLO block (described in Section 8.2), which contains sets of addresses. If the address is a URI, it may indicate which underlay understands the

respective address.

It is expected that the underlay provides basic mechanisms to manage peer connectivity and addressing. The essence of the underlay interface is captured by the following set of API calls:

TRY_CONNECT(P, A) This call allows the DHT implementation to signal to the underlay that the DHT wants to establish a connection to the target peer P using the given address A. If the connection attempt is successful, information on the new peer connection will be offered through the PEER_CONNECTED signal.

HOLD(P) This call tells the underlay to hold on to a connection to a peer P. Underlays are usually limited in the number of active connections. With this function the DHT can indicate to the underlay which connections should preferably be preserved.

DROP(P) This call tells the underlay to drop the connection to a peer P. This call is only there for symmetry and used during the peer's shutdown to release all of the remaining HOLDs. As R^5N always prefers the longest-lived connections, it would never drop an active connection that it has called HOLD() on before. Nevertheless, underlay implementations should not rely on this always being true. A call to DROP() also does not imply that the underlay must close the connection: it merely removes the preference to preserve the connection that was established by HOLD().

SEND(P, M) This call allows the local peer to send a protocol message M to a peer P. Sending messages is expected to be done on a best-effort basis, thus the underlay does not have to guarantee delivery or message ordering. If the underlay implements flow- or congestion-control, it may discard messages to limit its queue size.

ESTIMATE_NETWORK_SIZE() -> L2NSE This call must return an estimate of the network size. The resulting L2NSE value must be the base-2 logarithm of the _estimated_ number of peers in the network. This estimate is used by the routing algorithm. If the underlay does not support a protocol for network size estimation (such as [NSE]) the value is assumed to be provided as a configuration parameter to the underlay implementation.

The above calls are meant to be actively executed by the implementation as part of the peer-to-peer protocol. In addition, the underlay creates _signals_ to drive updates of the routing table, local storage and message processing (Section 7). Specifically, the underlay is expected to emit the following signals (usually implemented as callbacks) based on network events observed by the underlay implementation:

PEER_CONNECTED -> P This signal allows the DHT to react to a newly connected peer P. Such an event triggers, for example, updates in the routing table and gossiping of HELLOs to that peer. Underlays may include meta-data about the connection, for example to indicate that the connection is from a resource-constrained host that does not intend to function as a full peer and thus should not be considered for routing.

PEER_DISCONNECTED -> P This signal allows the DHT to react to a recently disconnected peer. Such an event primarily triggers updates in the routing table.

ADDRESS_ADDED -> A The underlay signals indicates that an address A was added for our local peer and that henceforth the peer may be reachable under this address. This information is used to advertise connectivity information about the local peer to other peers. A is an address suitable for inclusion in a HELLO payload Section 8.2.

ADDRESS_DELETED -> A This underlay signal indicates that an address A was removed from the set of addresses the local peer is possibly reachable under. The signal is used to stop advertising this address to other peers.

RECEIVE -> (P, M) This signal informs the local peer that a protocol message M was received from a peer P.

6. Routing

To enable routing, any R^5N implementation must keep information about its current set of neighbors. Upon receiving a connection notification from the underlay interface through a PEER_CONNECTED signal, information on the new neighbor MUST be added to the routing table, except if the respective k-bucket in the routing table is full or if meta-data is present that indicates that the peer does not wish to participate in routing. Peers added to the routing table SHOULD be signalled to the underlay as important connections using a HOLD() call. Similarly when a disconnect is indicated by the underlay through a PEER_DISCONNECTED signal, the peer MUST be removed from the routing table.

To achieve logarithmically bounded routing performance, the data structure for managing neighbors and their metadata MUST be implemented using the k-buckets concept of [Kademlia] as defined in Section 6.1. Maintenance of the routing table (after bootstrapping) is described in Section 6.2.

Unlike [Kademlia], routing decisions in R^5N are also influenced by a Bloom filter in the message that prevents routing loops. This data structure is discussed in Section 6.3.

In order to select peers which are suitable destinations for routing messages, R^5N uses a hybrid approach: Given an estimated network size L2NSE retrieved using ESTIMATE_NETWORK_SIZE(), the peer selection for the first L2NSE hops is random. After the initial L2NSE hops, peer selection follows an XOR-based peer distance calculation. Section 6.4 describes the corresponding routing functions.

Finally, each ResultMessage is routed back along the path that the corresponding GetMessage took previously. This is enabled by tracking state per GetMessage in the pending table described in Section 6.5.

6.1. Routing Table

Whenever a PEER_CONNECTED signal is received from the underlay, the respective peer is considered for insertion into the routing table. The routing table consists of an array of k-buckets. Each k-bucket contains a list of neighbors. The i-th k-bucket stores neighbors whose peer public keys are between XOR-distance 2^i and 2^(i+1) from the local peer; i can be directly computed from the two peer identities using the GetDistance() function. System constraints will typically force an implementation to impose some upper limit on the number of neighbors kept per k-bucket. Upon insertion, the implementation MUST call HOLD() on the respective neighor.

Implementations SHOULD try to keep at least 5 entries per k-bucket. Embedded systems that cannot manage this number of connections MAY use connection-level signalling to indicate that they are merely a client utilizing a DHT and not able to participate in routing. DHT peers receiving such connections MUST NOT include connections to such restricted systems in their k-buckets, thereby effectively excluding them when making routing decisions.

If a system hits constraints with respect to the number of active connections, an implementation MUST evict neighbours from those k-buckets with the largest number of neighbors. The eviction strategy MUST be to drop the shortest-lived connection per k-bucket first.

Implementations MAY cache valid addresses of disconnected peers outside of the routing table and sporadically or periodically try to (re-)establish connection to the peer by making TRY_CONNECT() calls to the underlay interface if the respective k-bucket has empty slots.

6.2. Peer Discovery

Initially, implementations require at least one initial connection to a neighbor (signalled through PEER_CONNECTED). The first connection SHOULD be established by an out-of-band exchange of the information from a HELLO block. This is commonly achieved through the configuration of hardcoded bootstrap peers or bootstrap servers either for the underlay or the R^5N implementation.

Implementations MAY have other means to achieve this initial connection. For example, implementations could allow the application or even end-user to provide a working HELLO which is then in turn used to call TRY_CONNECT() on the underlay in order to trigger a subsequent PEER_CONNECTED signal from the underlay interface. Appendix C specifies a URL format for encoding HELLO blocks as text strings. The URL format thus provides a portable, human-readable, text-based serialization format that can, for example, be encoded into a QR code for dissemination. HELLO URLs SHOULD be supported by implementations for both import and export of HELLOs.

To discover additional peers for its routing table, a peer MUST initiate GetMessage requests (see Section 7.4) asking for blocks of type HELLO using its own peer identity in the QUERY_HASH field of the message. The PEER_BF field of the GetMessage MUST be initialized to filter the peer's own peer identity as well as the peer identities of all currently connected neighbors. These requests MUST use the FindApproximate and DemultiplexEverywhere flags. FindApproximate will ensure that other peers will reply with results where the keys are merely considered close-enough, while DemultiplexEverywhere will cause each peer on the path to respond if it has relevant information. The combination of these flags is thus likely to yield HELLOs of peers that are useful somewhere in the initiator's routing table. The RECOMMENDED replication level to be set in the REPL_LVL field is 4. The size and format of the result filter is specified in Section 8.2. The XQUERY MUST be empty.

In order to facilitate peers answering requests for HELLOs, the underlay is expected to provide the implementation with addresses signalled through ADDRESS_ADDED. It is possible that no addresses are provided if a peer can only establish outgoing connections and is otherwise unreachable. An implementation MUST advertise its addresses periodically to its neighbors through HelloMessages. The advertisement interval and expiration SHOULD be configurable. If the values are chosen at the discretion of the implementation, it is RECOMMENDED to choose external factors such as expiration of DHCP leases to determine the values. The specific frequency of advertisements SHOULD be smaller than the expiration period. It MAY additionally depend on available bandwidth, the set of already connected neighbors, the workload of the system and other factors which are at the discretion of the developer. If HelloMessages are not updated before they expire, peers might be unable to discover and connect to the respective peer, and thus miss out on quality routing table entries. This would degrade the performance of the DHT and SHOULD thus be avoided by advertising updated HELLOs before the previous one expires. When using unreliable underlays, an implementation MAY use higher frequencies and transmit more HelloMessages within an expiration interval to ensure that neighbours almost always have non-expired HelloMessages at their disposal even if some messages are lost.

Whenever a peer receives such a HelloMessage from another peer that is already in the routing table, it must cache it as long as that peer remains in its routing table (or until the HELLO expires) and serve it in response to GET requests for HELLO blocks (see Section 7.4.3). This behaviour makes it unnecessary for peers to initiate dedicated PutMessages containing HELLO blocks.

6.3. Peer Bloom Filter

As DHT GetMessages and PutMessages traverse a random path through the network for the first L2NSE hops, a key design objective of R^5N is to avoid routing loops. The peer Bloom filter is part of the routing metadata in messages to prevent circular routes. It is updated at each hop where the hop’s peer identity derived from the peer’s public key is added to it. The peer Bloom filter follows the definition in Appendix A. It MUST be L=1024 bits (128 bytes) in size and MUST set k=16 bits per element. The set of elements E consists of of all possible 256-bit peer public keys and the mapping function M is defined as follows:

M(e) -> SHA-512 (e) as uint32[]

The element e is the peer public key which is hashed using SHA-512. The resulting 512-bit peer identity is interpreted as an array of k=16 32-bit integers in network byte order which are used to set and check the bits in B using BF-SET() and BF-TEST().

At this size, the Bloom filter reaches a false-positive rate of approximately fifty percent at about 200 entries. For peer discovery where the Bloom filter is initially populated with peer identities from the local routing table, the 200 entries would still be enough for 40 buckets assuming 5 peers per bucket, which corresponds to an overlay network size of approximately 1 trillion peers. Thus, L=1024 bits should suffice for all conceivable use-cases.

For the next hop selection in both the random and the deterministic case, any peer which is in the peer Bloom filter for the respective message is excluded from the peer selection process. Any peer which is forwarding GetMessages or PutMessages (Section 7) thus adds its own peer public key to the peer Bloom filter. This allows other peers to (probabilistically) exclude already traversed peers when searching for the next hops in the routing table.

We note that the peer Bloom filter may exclude peers due to false-postive matches. This is acceptable as routing should nevertheless terminate (with high probability) in close vicinity of the key. Furthermore, due to the randomization of the first L2NSE hops, it is possible that false-positives will be different when a request is repeated.

6.4. Routing Functions

Using the data structures described so far, the R^5N routing component provides the following functions for message processing (Section 7):

GetDistance(A, B) -> Distance This function calculates the binary XOR between A and B. The resulting distance is interpreted as an integer where the leftmost bit is the most significant bit.

SelectClosestPeer(K, B) -> N This function selects the neighbor N from our routing table with the shortest XOR-distance to the key K. This means that for all other peers N' in the routing table GetDistance(N, K) < GetDistance(N',K). Peers with a positive test against the peer Bloom filter B are not considered.

SelectRandomPeer(B) -> N This function selects a random peer N from all neighbors. Peers with a positive test in the peer Bloom filter B are not considered.

SelectPeer(K, H, B) -> N This function selects a neighbor N depending on the number of hops H parameter. If H < NETWORK_SIZE_ESTIMATE returns SelectRandomPeer(B), and otherwise returns SelectClosestPeer(K, B).

IsClosestPeer(N, K, B) -> true | false This function checks if N is the closest peer for K (cf. SelectClosestPeer(K, B)). Peers with a positive test in the Bloom filter B are not considered.

ComputeOutDegree(REPL_LVL, HOPCOUNT, L2NSE) -> Number This function computes the number of neighbors that a message should be forwarded to. The arguments are the desired replication level (REPL_LVL), the HOPCOUNT of the message so far and and the current network size estimate (L2NSE) as provided by the underlay. The result is the non-negative number of next hops to select. The following figure gives the pseudocode for computing the number of neighbors the peer should attempt to forward the message to.

```
ComputeOutDegree(REPL_LVL, HOPCOUNT, L2NSE):
  if (HOPCOUNT > L2NSE * 4)
    return 0;
  if (HOPCOUNT > L2NSE * 2)
    return 1;
  if (0 = REPL_LEVL)
    REPL_LEVL = 1
  if (REPL_LEVEL > 16)
    REPL_LEVEL = 16
    RM1 = REPL_LEVEL - 1
    FRAC = 1 + (RM1 / (L2NSE + RM1 * HOPCOUNT))
  return PROUND(FRAC)
```

Figure 2: Computing the number of next hops.

The above calculation of FRAC may yield values that are not discrete. The result is FRAC rounded probabilistically (PROUND) to the nearest discrete value, using the fraction as the probability for rounding up. For example, a value of 3.14 is rounded up to 4 with a probability of 14% and rounded down to 3 with a probability of 86%. This probabilistic rounding is necessary to achieve the statistically expected value of the replication level and average number of peers a message is forwarded to.

6.5. Pending Table

R^5N performs stateful routing where the messages only carry the query hash and do not encode the ultimate source or destination of the request. Routing a request towards the key is done hop-by-hop using the routing table and the query hash. The pending table is used to route responses back to the originator. In the pending table each peer primarily maps a query hash to the associated originator of the request. The pending table MUST store entries for the last MAX_RECENT requests the peer has encountered. To ensure that the peer does not run out of memory, information about older requests MAY be discarded. The value of MAX_RECENT MAY be configurable at the host level to use available memory resources without conflicting with other system requirements and limitations. MAX_RECENT SHOULD be at least 128 * 10^3. If the pending table is smaller, the likelihood grows that a peer receives a response to a query but is unable to forward it to the initiator because it forgot the predecessor. Eventually, the initiator would likely succeed by repeating the query. However, this would be much more expensive than peers having an adequately sized pending table.

For each entry in the pending table, the DHT MUST track the query key, the peer identity of the previous hop, the extended query, requested block type, flags, and the result filter. If the query did not provide a result filter, a fresh result filter MUST still be created to filter duplicate replies. Details of how a result filter works depend on the type, as described in Section 8.1.

When a second query from the same origin for the same query hash is received, the DHT MUST attempt to merge the new request with the state for the old request. If this is not possible (say because the MUTATOR differs), the existing result filter MUST be discarded and replaced with the result filter of the incoming message.

We note that for local applications, a fixed limit on the number of concurrent requests may be problematic. Hence, it is RECOMMENDED that implementations track requests from local applications separately and preserve the information about requests from local applications until the local application explicitly stops the request.

7. Message Processing

An implementation will process messages either because it needs to transmit messages as part of routing table maintenance, or due to requests from local applications, or because it received a message from a neighbor. If instructed through an application-facing API such as the one outlined in Appendix B, a peer acts as an initiator of GetMessages or PutMessages. The status of initiator is relevant for peers when processing ResultMessages due to the required handover of results to the application that requested the respective result.

The implementation MUST listen for RECEIVE(P, M) signals from the underlay and react to the respective messages sent by the peer P.

Whether initiated locally or received from a neighbor, an implementation processes messages according to the wire formats and the required validations detailed in the following sections. Where required, the local peer public key is referred to as SELF.

7.1. Message components

This section describes some data structures and fields shared by various types of messages.

7.1.1. Flags

Flags is an 8-bit vector representing binary options. Each flag is represented by a bit in the field starting from 0 as the rightmost bit to 7 as the leftmost bit.

0: DemultiplexEverywhere This bit indicates that each peer along the way should process the request. If the bit is not set, intermediate peers only route the message and only peers which consider themselves closest to the key (based on their routing table) look for answers in their local storage for GetMessages, or respectively cache the block in their local storage for PutMessages and ResultMessages.

1: RecordRoute This bit indicates to keep track of the path that the message takes in the P2P network.

2: FindApproximate This bit asks peers to return results even if the key does not exactly match the query hash.

3: Truncated This is a special flag which is set if a peer truncated the recorded path. This results in the first hop on the path to be given without a signature to enable checking of the next signature. This flag MUST NOT be set in a GetMessage.

4-7: Reserved The remaining bits are reserved for future use and MUST be set to zero when initiating an operation. If non-zero bits are received, implementations MUST preserve these bits when forwarding messages.

7.1.2. Path

If the RecordRoute flag is set, the route of a PutMessage or a ResultMessage through the overlay network is recorded in the PATH field of the message. PATH is a list of path elements. A new path element (Section 7.1.3) is appended to the existing PATH before a peer sends the message to the next peer.

A path element contains a signature and the public key of the peer that created the element. The signature is computed over the public keys of the previous peer (from which the message was received) and next peer (the peer the message is send to). A new message has no previous peer and uses all ZEROs (32 NULL-bytes) in the public key field when creating the signature.

Assuming peer A sends a new PUT message to peer B, which forwards the message to peer C, which forwards to peer D which finally stores the data. The PATH field of the message received at peer D contains three path elements (build from top to bottom):

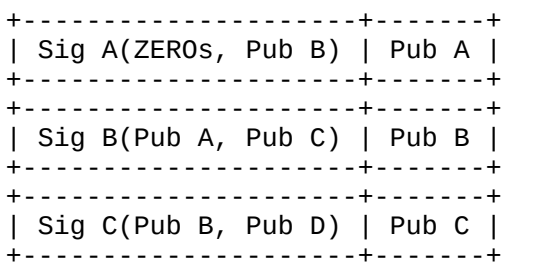


Figure 3: Example PATH

Note that the wire format of PATH (Section 7.1.3) will not include the last public key (Pub C in our example) as this will be redundant; the receiver of a message can use the public key of the sender as the public key to verify the last signature.

The PATH is stored along with the payload data from the PUT message at the final peer. Note that the same payload stored at different peers will have a different PATH associated with it.

When the storing peer delivers the data based on a GET request, it initializes the PATH field with the stored path value and appends a new path element. The first part of PATH in a GET response message is called the PutPath, followed by the GetPath. This way the combined PATH will record the whole route of the payload from the originating peer (initial PutMessage) to the requesting peer (initial GetMessage).

When receiving a message with flag RecordRoute and PATH, a peer is encouraged to verify the integrity of PATH (if the available resources of the peer allow this) by checking the signatures of the path elements.

If an invalid signature is detected, the path is truncated keeping only element fields following the faulty signature and setting the Truncated flag. Assume that peer C detects a faulty signature from peer B, the trunacted path has two entries:

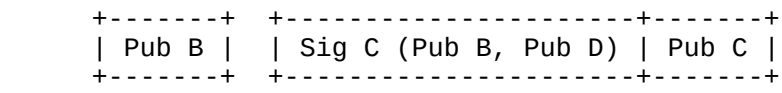


Figure 4: Example truncated PATH

The Truncated flag indicates that the first path element does not contain a signature but only the public key of the peer where the signature fails.

7.1.3. Path Element

A path element represents a hop in the path a message has taken through the overlay network. The wire format of a path element is illustrated in Figure 5.

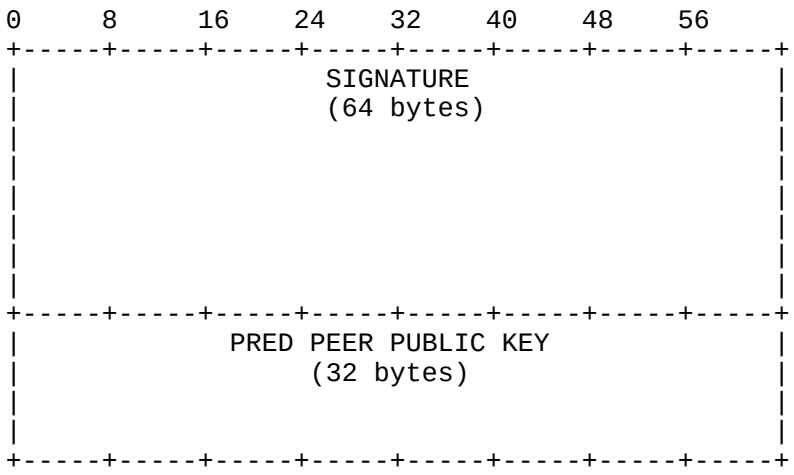


Figure 5: The Wire Format of a path element.

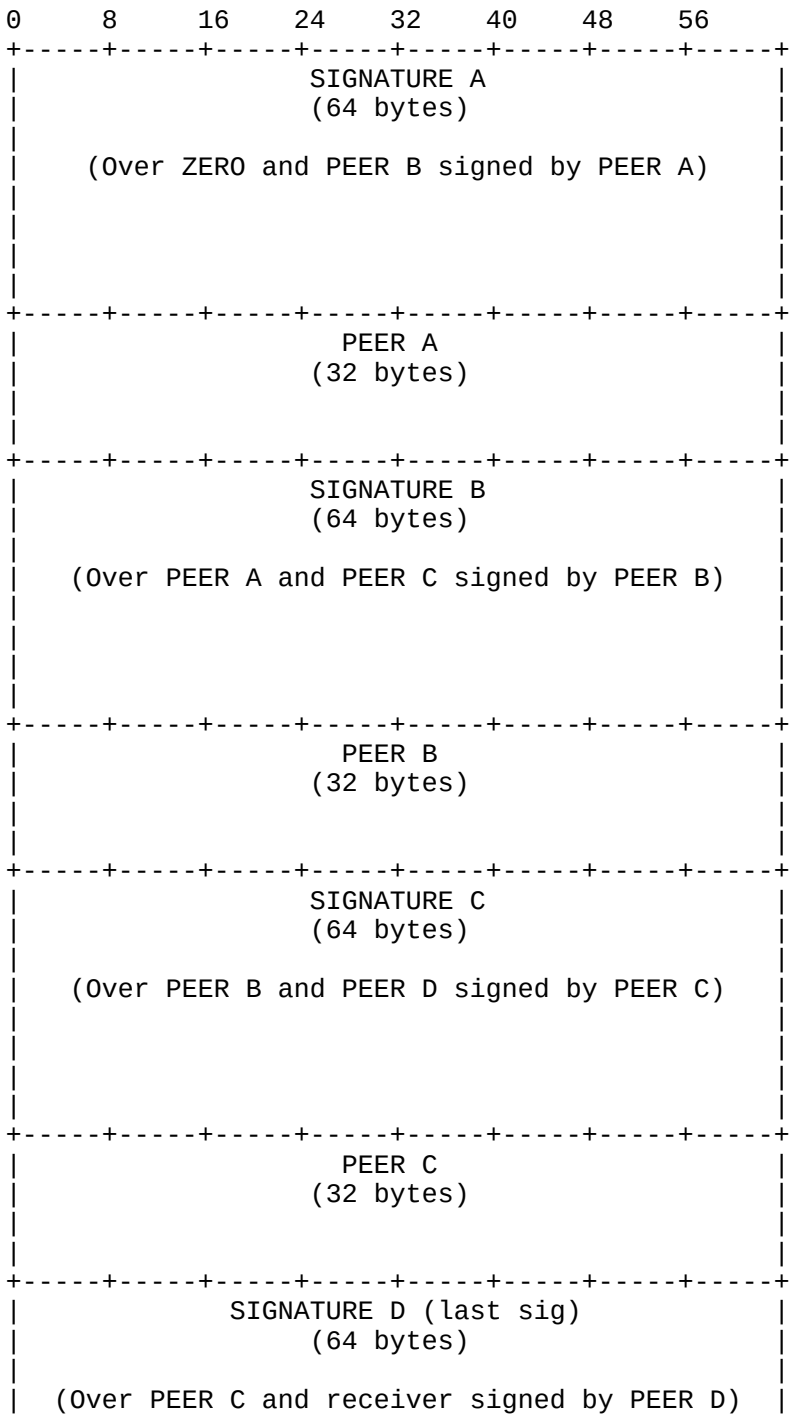
where:

SIGNATURE is a 64 byte EdDSA signature [ed25519] created using the current hop's private key which affirms the public keys of the peers from the previous and next hops.

PRED PEER PUBLIC KEY is the EdDSA public key [ed25519] of the previous peer on the path.

An ordered list of path elements may be appended to any routed PutMessages or ResultMessages. The last signature (after which the peer public key is omitted) is created by the current hop only after the peer made its routing decision identifiying the successor peer. The peer public key is not included after the last signature as it must be that of the sender of the message and including it would thus be redundant. Similarly, the predecessor of the first element of an untruncated path is not stated explicitly, as it must be ZERO (32 NULL-bytes).

Figure 6 shows the wire format of an example path from peer A over peers B and C and D as it would be received by peer E in the PUTPATH of a PutMessage, or as the combined PUTPATH and GETPATH of a ResultMessage. The wire format of the path elements allows a natural extension of the PUTPATH along the route of the ResultMessage to the destination forming the GETPATH. The PutMessage would indicate in the PATH_LEN field a length of 3. The ResultMessage would indicate a path length of 3 as the sum of the field values in PUTPATH_L and GETPATH_L. Basically, the last signature does not count for the path length.


0 8 16 24 32 40 48 56
SIGNATURE A
(64 bytes)
(Over ZERO and PEER B signed by PEER A)
PEER A
(32 bytes)
SIGNATURE B
(64 bytes)
(Over PEER A and PEER C signed by PEER B)
PEER B
(32 bytes)
SIGNATURE C
(64 bytes)
(Over PEER B and PEER D signed by PEER C)
PEER C
(32 bytes)
SIGNATURE D (last sig)
(64 bytes)
(Over PEER C and receiver signed by PEER D)

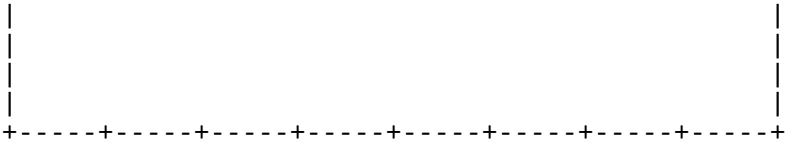


Figure 6: Example of a path as found in PutMessages or ResultMessages from Peer A to Peer D as transmitted by Peer D.

A path may be truncated in which case the signature of the truncated path element is omitted leaving only the public key of the peer preceeding the truncation which is required for the verification of the subsequent path element signature. Such a truncated path is indicated with the respective truncated flag (Section 7.1.1). For truncated paths, the peer public key of the signer of the last path element is again omitted as it must be that of the sender of the PutMesssage or ResultMessage. Similarly, the public key of the receiving peer used in the last path element is omitted as it must be SELF. The wire format of a truncated example path from peers B over C and D to E (possibly still originating at A, but the origin is unknowable to E due to truncation) is illustrated in Figure 7. Here, a ResultMessage would indicate in the PATH_LEN field a length of 1 while a PutMessage would indicate a length of 1 as the sum of PUTPATH_L and GETPATH_L fields. Basically, the truncated peer and the last signature do not count for the path length.

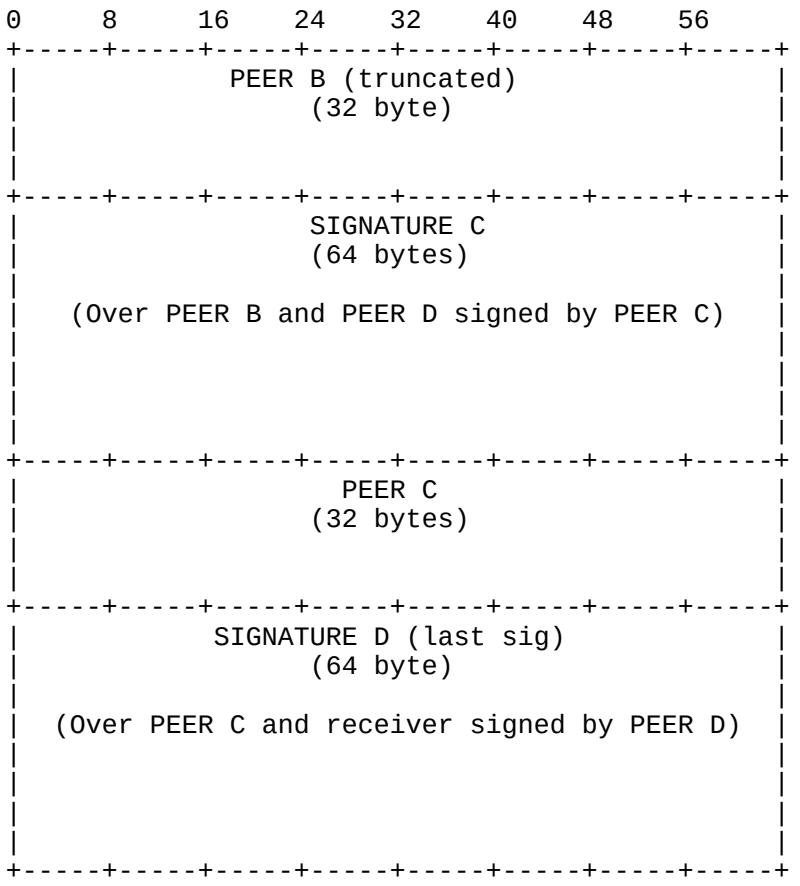


Figure 7: Example of a truncated path from Peer B to Peer D as transmitted by Peer D.

The SIGNATURE field in a path element covers a 64-bit contextualization header, the the block expiration, a hash of the block payload, as well as the predecessor peer public key and the peer public key of the successor that the peer making the signature is routing the message to. Thus, the signature made by SELF basically says that SELF received the block payload from PEER PREDECESSOR and has forwarded it to PEER SUCCESSOR. The wire format is illustrated in Figure 8.

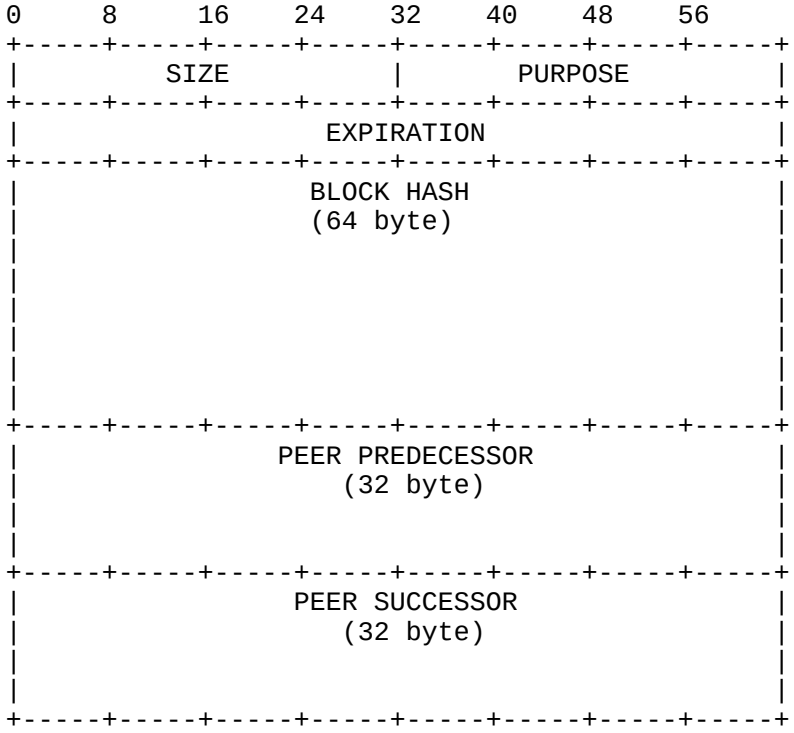


Figure 8: The Wire Format of the path element for Signing.

SIZE A 32-bit value containing the length of the signed data in bytes in network byte order. The length of the signed data MUST be 144 bytes.

PURPOSE A 32-bit signature purpose flag. This field MUST be 6 (in network byte order).

EXPIRATION denotes the absolute 64-bit expiration date of the block in microseconds since midnight (0 hour), January 1, 1970 UTC in network byte order.

BLOCK HASH a SHA-512 hash over the block payload.

PEER PREDECESSOR the peer public key of the previous hop. If the signing peer initiated the PUT, this field is set to all zeroes.

PEER SUCCESSOR the peer public key of the next hop (not of the signer).

7.2. HelloMessage

When the underlay signals the implementation of added or removed addresses through ADDRESS_ADDED and ADDRESS_DELETED an implementation MUST disseminate those changes to neighbors using HelloMessages (as already discussed in section Section 6.2). HelloMessages are used to inform neighbors of a peer about the sender's available addresses. The recipients use these messages to inform their respective underlays about ways to sustain the connections and to generate HELLO blocks (see Section 8.2) to answer peer discovery queries from other peers.

7.2.1. Wire Format

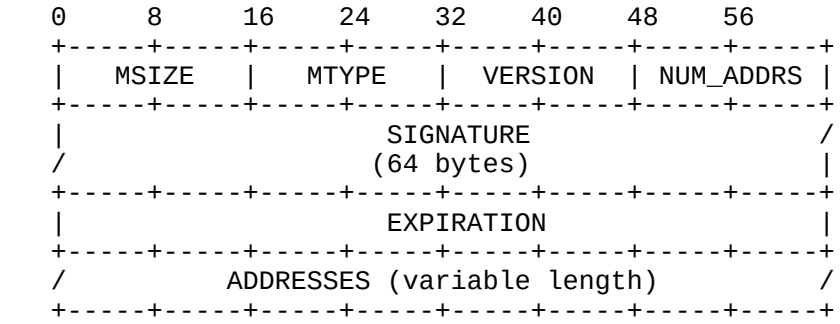


Figure 9: The HelloMessage Wire Format.

where:

MSIZE denotes the size of this message in network byte order.

MTYPE is the 16-bit message type. It must be set to the value 157 in network byte order as defined in the GANA "GNUnet Message Type" registry (see Section 11.4).

VERSION is a 16-bit field that indicates the version of the HelloMessage. Must be zero. In the future, this may be used to extend or update the HelloMessage format.

NUM_ADDRS is a 16-bit number in network byte order that gives the total number of addresses encoded in the ADDRESSES field.

SIGNATURE is a 64 byte EdDSA signature [ed25519] using the sender's private key affirming the information contained in the message. The signature is signing exactly the same data that is being signed in a HELLO block as described in Section 8.2.

EXPIRATION denotes the absolute 64-bit expiration date of the content. The value specified is microseconds since midnight (0 hour), January 1, 1970, but must be a multiple of one million (so that it can be represented in seconds in a HELLO URL). Stored in network byte order.

ADDRESSES A sequence of exactly NUM_ADDRS addresses which can be used to contact the peer. Each address MUST be 0-terminated. If NUM_ADDRS = 0 then this field is omitted (0 bytes).

7.2.2. Processing

If the initiator of a HelloMessage is SELF, the message is simply sent to all neighbors P currently in the routing table using the SEND() function of the underlay as defined in Section 5.

Upon receiving a HelloMessage from a peer P an implementation MUST process it step by step as follows:

1. If P is not in its routing table, the message is discarded.

2. The signature is verified, including a check that the expiration time is in the future. If the signature is invalid, the message is discarded.

3. The information contained in the HelloMessage is used to synthesize a block of type HELLO (Section 8.2). The block is cached in the routing table until it expires, or the peer is removed from the routing table, or the information is replaced by another message from the peer. The implementation SHOULD instruct the underlay to connect to all now available addresses using TRY_CONNECT() in order to make the underlay aware of alternative addresses for this connection and to maintain optimal connectivity.

4. Received HelloMessages MUST NOT be forwarded.

7.3. PutMessage

PutMessages are used to store information at other peers in the DHT. Any application-facing API which allows applications to initiate PutMessages to store data in the DHT needs to receive sufficient, possibly implementation-specific information to construct the initial PutMessage. In general, application-facing APIs supporting multiple applications and block types need to be given the block type (BTYPE) and message FLAGS in addition to the actual BLOCK payload. The BLOCK_KEY could be provided explicitly, or in some cases might be derived using the DeriveBlockKey() function from the block type

specific operations defined in Section 8.1.

7.3.1. Wire Format

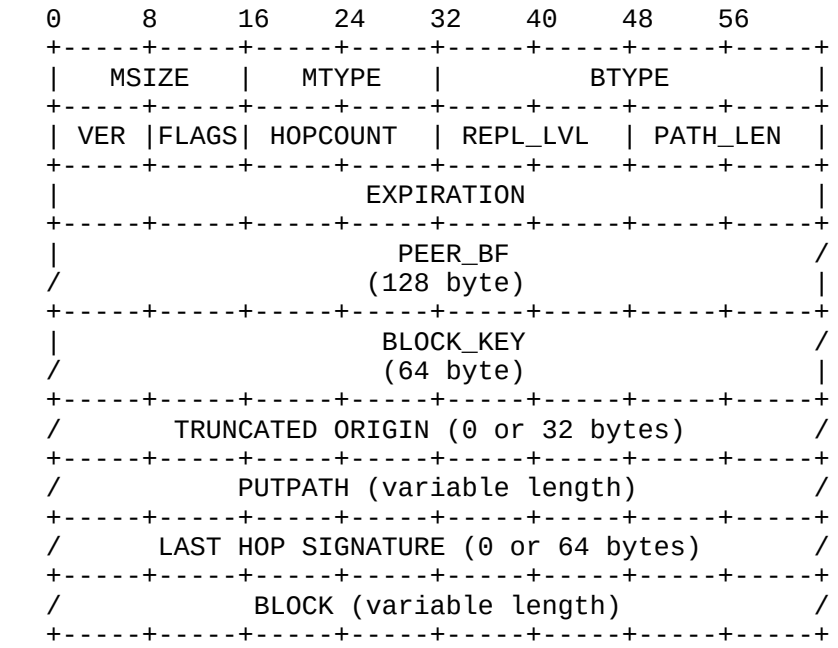


Figure 10: The PutMessage Wire Format.

where:

MSIZE denotes the size of this message in network byte order.

MTYPE is the 16-bit message type. Read-only. It must be set to the value 146 in network byte order as defined in the GANA "GNUnet Message Type" registry Section 11.4.

BTYPE is a 32-bit block type in network byte order. The block type indicates the content type of the payload. Set by the initiator. Read-only.

VER is a 8-bit protocol version. Set to zero. May be used in future protocol versions.

FLAGS is a 8-bit vector with binary options (see Section 7.1.1). Set by the initiator. Read-only.

HOPCOUNT is a 16-bit number in network byte order indicating how

many hops this message has traversed to far. Set by the initiator to zero. MUST be incremented by one by each peer before forwarding the request.

REPL_LVL is a 16-bit number in network byte order indicating the desired replication level of the data. Set by the initiator. Read-only.

PATH_LEN is a 16-bit number in network byte order indicating the number of path elements recorded in PUTPATH. As PUTPATH is optional, this value may be zero. If the PUTPATH is enabled, set initially to zero by the initiator. Updated by processing peers to match the path length in the message.

EXPIRATION denotes the absolute 64-bit expiration date of the content in microseconds since midnight (0 hour), January 1, 1970 in network byte order. Set by the initiator. Read-only.

PEER_BF A peer Bloom filter to stop circular routes (see Section 6.3). Set by the initiator to contain the local peer and all neighbors it is forwarded to. Modified by processing peers to include their own peer public key using BF-SET().

BLOCK_KEY The key under which the PutMessage wants to store content under. Set by the initiator. Read-only.

TRUNCATED ORIGIN is only provided if the Truncated flag is set in FLAGS. If present, this is the public key of the peer just before the first entry on the PUTPATH and the first peer on the PUTPATH is not the actual origin of the message. Thus, to verify the first signature on the PUTPATH, this public key must be used. Note that due to the truncation, this last hop cannot be verified to exist. Value is modified by processing peers.

PUTPATH the variable-length PUT path. The path consists of a list of PATH_LEN path elements. Set by the initiator to zero. Incremented by processing peers.

LAST HOP SIGNATURE is only provided if the RecordRoute flag is set in FLAGS. If present, this is an EdDSA signature [ed25519] by the sender of this message (using the same format as the signatures in PUTPATH) affirming that the sender forwarded the message from the predecessor (all zeros if PATH_LEN is zero, otherwise the last peer in PUTPATH) to the target peer. Modified by processing peers (if flag is set).

BLOCK the variable-length block payload. The contents are

determined by the BTYPE field. The length is determined by MSIZE minus the size of all of the other fields. Set by the initiator. Read-only.

7.3.2. Processing

Upon receiving a PutMessage from a peer P, or created through initiation by an overlay API, an implementation MUST process it step by step as follows:

1. The EXPIRATION field is evaluated. If the message is expired, it MUST be discarded.

2. If the BTYPE is ANY, then the message MUST be discarded. If the BTYPE is not supported by the implementation, no validation of the block payload is performed and processing continues at (5). Else, the block MUST be validated as defined in (3) and (4).

3. The message is evaluated using the block validation functions matching the BTYPE. First, the client attempts to derive the key using the respective DeriveBlockKey procedure as described in Section 8.1. If a key can be derived and does not match, the message MUST be discarded.

4. Next, the ValidateBlockStoreRequest procedure for the BTYPE as described in Section 8.1 is used to validate the block payload. If the block payload is invalid, the message MUST be discarded.

5. The peer identity of the sender peer P SHOULD be in PEER_BF. If not, the implementation MAY log an error, but MUST continue.

6. If the RecordRoute flag is not set, the PATH_LEN MUST be set to zero. If the flag is set and PATH_LEN is non-zero, the local peer SHOULD verify the signatures from the PUTPATH. Verification MAY involve checking all signatures or any random subset of the signatures. It is RECOMMENDED that peers adapt their behavior to available computational resources so as to not make signature verification a bottleneck. If an invalid signature is found, the PUTPATH MUST be truncated to only include the elements following the invalid signature.

7. If the local peer is the closest peer (cf. IsClosestPeer(SELF, BLOCK_KEY, PeerFilter)) or the DemultiplexEverywhere flag ist set, the message SHOULD be stored locally in the block storage if possible. The implementation MAY choose not store the block if external factors or configurations prevent this, such as limited (allotted) disk space.

8. If the BTYPE of the message indicates a HELLO block, the peer MUST be considered for the local routing table by using the peer identity in BLOCK_KEY. If neither the peer is already connected nor the respective k-bucket is already full, then the peer MUST try to establish a connection to the peer indicated in the HELLO block using the address information from the HELLO block and the underlay's TRY_CONNECT() function. The implementation MUST instruct the underlay to try to connect to all provided addresses using TRY_CONNECT() in order to make the underlay aware of multiple addresses for this connection. When a connection can be established, the underlay's PEER_CONNECTED signal will cause the peer to be added to the respective k-bucket of the routing table (Section 6).

9. Given the value in REPL_LVL, HOPCOUNT and FALSE = IsClosestPeer(SELF, BLOCK_KEY, PeerFilter) the number of peers to forward to MUST be calculated using ComputeOutDegree(). The implementation SHOULD select this number of peers to forward the message to using the function SelectPeer() (Section 6.4) using the BLOCK_KEY, HOPCOUNT, and utilizing PEER_BF as peer Bloom filter. For each selected peer PEER_BF is updated with that peer in between calls to SelectPeer(). The implementation MAY forward to fewer or no peers in order to handle resource constraints such as limited bandwidth or simply because there are not enough suitable connected peers. For each selected peer with peer identity P a dedicated PutMessage_P is created containing the original (and where applicable already updated) fields of the received PutMessage. In each message _all_ selected peer identities and the local peer identity MUST be added to the PEER_BF and the HOPCOUNT MUST be incremented by one. If the RecordRoute flag is set, a new path element is created using the predecessor peer public key and the signature of the current peer. The path element is added to the PUTPATH fields and the PATH_LEN field is incremented by one. When creating the path element signature, the successor must be set to the recipient peer P of the PutMessage_P. The successor in the new path element is the recipient peer P. If the path becomes too long for the resulting message to be transmitted by the underlay, it MUST be truncated. Finally, the messages are sent using SEND(P, PutMessage_P) to each recipient.

7.4. GetMessage

GetMessages are used to request information from other peers in the DHT. An application-level API which allows applications to initiate GetMessages needs to provide sufficient, implementation-specific information needed to construct the initial GetMessage. For example, implementations supporting multiple applications and blocks will need to be given the block type, message FLAG parameters and possibly an XQUERY in addition to just the QUERY_HASH. In some cases, it might also be useful to enable the application to assist in the construction of the RESULT_FILTER such that it can filter already known results. Note that the RESULT_FILTER may need to be re-constructed every time the query is retransmitted by the initiator (details depending on the BTYPE) and thus a RESULT_FILTER can often not be passed directly as an argument by the application to an application API. Instead, applications would typically provide the set of results to be filtered, allowing the DHT to construct the RESULT_FILTER whenever it retransmits a GetMessage request as initiator.

7.4.1. Wire Format

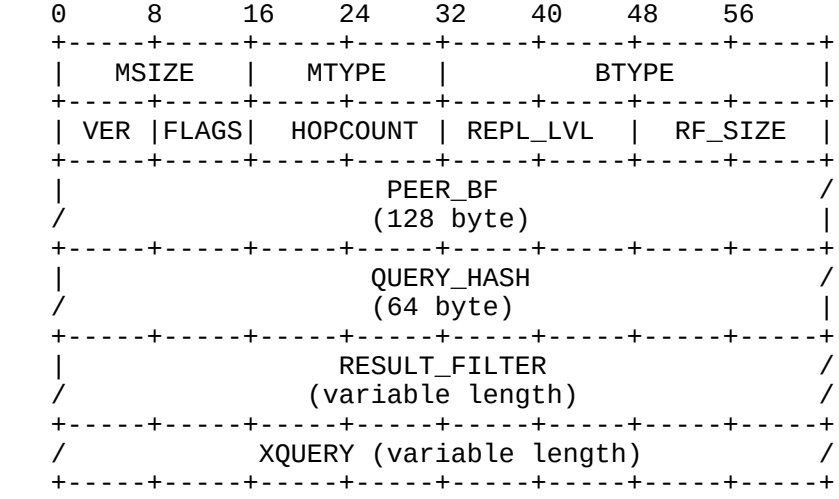


Figure 11: The GetMessage Wire Format.

where:

MSIZE denotes the size of this message in network byte order.

MTYPE is the 16-bit message type. Read-only. It must be set to the

value 147 in network byte order as defined in the GANA "GNUnet Message Type" registry Section 11.4.

BTYPE is a 32-bit block type field in network byte order. The block type indicates the content type of the payload. Set by the initiator. Read-only.

VER is a 8-bit protocol version. Set to zero. May be used in future protocol versions.

FLAGS is a 8-bit vector with binary options (see Section 7.1.1). Set by the initiator. Read-only.

HOPCOUNT is a 16-bit number in network byte order indicating how many hops this message has traversed to far. Set by the initiator to zero. Incremented by each peer by one per hop.

REPL_LVL is a 16-bit number in network byte order indicating the desired replication level of the data. Set by the initiator. Read-only.

RF_SIZE is a 16-bit number in network byte order indicating the length of the RESULT_FILTER. Set by the initiator. Read-only.

PEER_BF A peer Bloom filter to stop circular routes (see Section 6.3). Set by the initiator to include itself and all connected neighbors in the routing table. Modified by processing peers to include their own peer identity.

QUERY_HASH The query used to indicate what the key is under which the initiator is looking for blocks with this request. The block type may use a different evaluation logic to determine applicable result blocks. Set by the initiator. Read-only.

RESULT_FILTER the variable-length result filter with RF_SIZE bytes as described in Section 7.4.2. Set by the initiator. Modified by processing peers based on results returned.

XQUERY the variable-length extended query. Optional. Set by the initiator. Read-only. The length is determined by subtracting the length of all other fields from MSIZE.

7.4.2. Result Filter

A result filter is used to indicate to other peers which results are not of interest when processing a GetMessage (Section 7.4). Any peer which is processing GetMessages and has a result which matches the query key MUST check the result filter and only send a reply message if the result does not test positive under the result filter. Before forwarding the GetMessage, the result filter MUST be updated using the result of the BTYPE-specific FilterResult (see Section 8.1) function to filter out all results already returned by the local peer.

How a result filter is implemented depends on the block type as described in Section 8.1. Result filters may be probabilistic data structures. Thus, it is possible that a desireable result is filtered by a result filter because of a false-positive test.

How exactly a block result is added to a result filter is specified as part of the definition of a block type (cf. Section 8.2).

7.4.3. Processing

Upon receiving a GetMessage from a peer P, or created through initiation by the overlay API, an implementation MUST process it step by step as follows:

1. If the BTYPE is supported, the QUERY_HASH and XQUERY fields are validated as defined by the respective ValidateBlockQuery() procedure for this type. If the result yields REQUEST_INVALID, the message MUST be discarded and processing ends. If the BTYPE is not supported, the message MUST be forwarded (Skip to step 4). If the BTYPE is ANY, the message is processed further without validation.

2. The peer identity of the sender peer P SHOULD be in the PEER_BF peer Bloom filter. If not, the implementation MAY log an error, but MUST continue.

3. The local peer SHOULD try to produce a reply in any of the following cases: (1) If the local peer is the closest peer (cf. IsClosestPeer(SELF, QueryHash, PeerFilter), or (2) if the DemultiplexEverywhere flag is set, or (3) if the local peer is not the closest and a previously cached ResultMessage also matches this request (Section 7.5.2).

   The reply is produced (if one is available) using the following steps:

a) If the BTYPE is HELLO, the implementation MUST only consider synthesizing its own addresses and the addresses it has cached for the peers in its routing table as HELLO block replies. Otherwise, if the BTYPE does not indicate a request for a HELLO block or ANY, the implementation MUST only consider blocks in the local block storage and previously cached ResultMessages.

b) If the FLAGS field includes the flag FindApproximate, the peer SHOULD respond with the closest block (smallest value of GetDistance(QUERY_HASH, BLOCK_KEY)) it can find that is not filtered by the RESULT_BF. Otherwise, the peer MUST respond with the block with a BLOCK_KEY that matches the QUERY_HASH exactly and that is not filtered by the RESULT_BF.

c) Any resulting (synthesized) block is encapsulated in a ResultMessage. The ResultMessage SHOULD be transmitted to the neighbor from which the request was received.

Implementations MAY not reply if they are resource-constrained. However, ResultMessages MUST be given the highest priority among competing transmissions.

If the BTYPE is supported and ValidateBlockReply for the given query has yielded a status of FILTER_LAST, processing MUST end and not continue with forwarding of the request to other peers.

4. The implementation MUST create (or merge) an entry in the pending table (see Section 6.5) for the query represented by this GetMessage. The pending table MUST store the last MAX_RECENT requests, and peers thus MUST discard the oldest existing request if memory constraints on the pending table are encountered. Note that peers MUST clean up state for queries that had response with a status of FILTER_LAST even if they are not the oldest query in the pending table.

5. Using the value in REPL_LVL, the number of peers to forward to MUST be calculated using ComputeOutDegree(). If there is at least one peer to forward to, the implementation SHOULD select up to this number of peers to forward the message to. Furthermore, the implementation SHOULD select up to this number of peers to using the function SelectPeer() (from Section 6.4) using the QUERY_HASH, HOPCOUNT, and the PEER_BF. The implementation MAY forward to fewer or no peers in order to handle resource constraints such as bandwidth. Before forwarding, the peer Bloom filter PEER_BF MUST be updated to filter all selected peers and the local peer identity SELF. For all peers with peer identity P chosen to forward the message to, SEND(P, GetMessage_P) is

called. Here, GetMessage_P is the original message with the updated fields for HOPCOUNT (incremented by one), updated PEER_BF and updated RESULT_FILTER (based on results already returned).

7.5. ResultMessage

ResultMessages are used to return information to other peers in the DHT or to applications using the overlay API that previously initiated a GetMessage. The initiator of a ResultMessage is a peer triggered through the processing of a GetMessage.

7.5.1. Wire Format

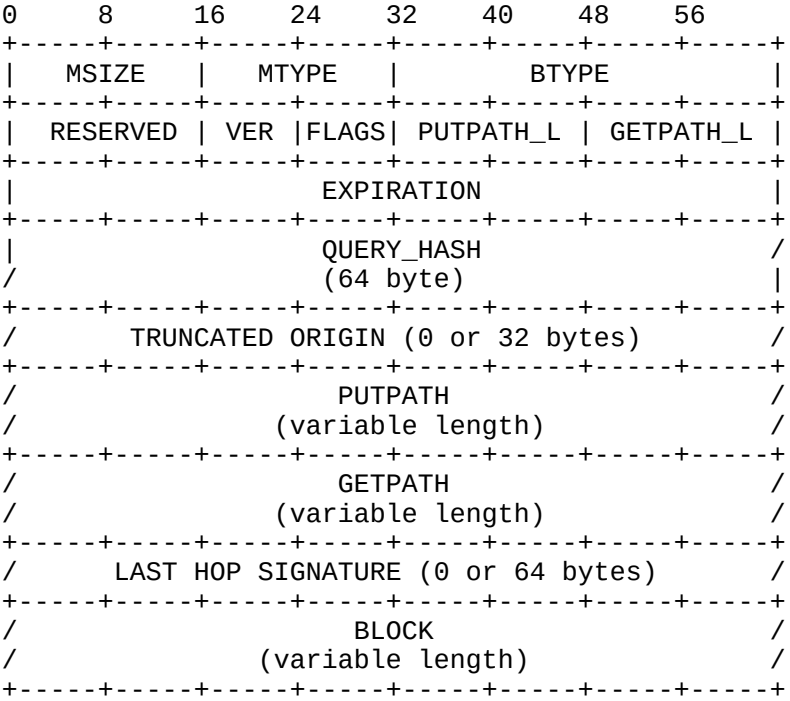


Figure 12: The ResultMessage Wire Format

where:

MSIZE denotes the size of this message in network byte order.

MTYPE is the 16-bit message type. Set by the initiator. Read-only. It must be set to the value 148 in network byte order as defined in the GANA "GNUnet Message Type" registry (see Section 11.4).

BTYPE is a 32-bit block type field in network byte order. The block type indicates the content type of the payload. Set by the initiator. Read-only.

RESERVED is a 16-bit value. Implementations MUST set this value to zero when originating a result message. Implementations MUST forward this value unchanged even if it is non-zero.

VER is a 8-bit protocol version in network byte order. Set to zero. May be used in future protocol versions.

FLAGS is a 8-bit vector with binary options (see Section 7.1.1). Set by the initiator of the response based on the flags retained from the original PutMessage, possibly setting the Truncated bit if the initiator is forced to truncate the path. For HELLO blocks, the FLAGS should simply be cleared.

PUTPATH_L is a 16-bit number in network byte order indicating the number of path elements recorded in PUTPATH. As PUTPATH is optional, this value may be zero even if the message has traversed several peers. Set by the initiator to the PATH_LEN of the PutMessage from which the block originated. Modified by processing peers in case of path truncation.

GETPATH_L is a 16-bit number in network byte order indicating the number of path elements recorded in GETPATH. As GETPATH is optional, this value may be zero even if the message has traversed several peers. MUST be set to zero by the initiator. Modified by processing peers to match the path length in the message.

EXPIRATION denotes the absolute 64-bit expiration date of the content in microseconds since midnight (0 hour), January 1, 1970 in network byte order. Set by the initiator to the expiration value as recorded from the PutMessage from which the block originated. Read-only.

QUERY_HASH the query hash corresponding to the GetMessage which caused this reply message to be sent. Set by the initiator using the value of the GetMessage. Read-only.

TRUNCATED ORIGIN is only provided if the Truncated flag is set in FLAGS. If present, this is the public key of the peer just before the first entry on the PUTPATH and the first peer on the PUTPATH is not the actual origin of the message. Thus, to verify the first signature on the PUTPATH, this public key must be used. Note that due to the truncation, this last hop cannot be verified to exist. Set by processing peers.

PUTPATH the variable-length PUT path. The path consists of a list of PUTPATH_L path elements. Set by the initiator to the the PUTPATH of the PutMessage from which the block originated. Modified by processing peers in case of path truncation.

GETPATH the variable-length PUT path. The path consists of a list of GETPATH_L path elements. Set by processing peers.

LAST HOP SIGNATURE is only provided if the RecordRoute flag is set in FLAGS. If present, this is an EdDSA signature [ed25519] by the sender of this message (using the same format as the signatures in PUTPATH) affirming that the sender forwarded the message from the predecessor (all zeros if PATH_LEN is zero, otherwise the last peer in PUTPATH) to the target peer.

BLOCK the variable-length resource record data payload. The contents are defined by the respective type of the resource record. Set by the initiator. Read-only.

7.5.2. Processing

Upon receiving a ResultMessage from a connected peer or triggered by the processing of a GetMessage, an implementation MUST process it step by step as follows:

1. First, the EXPIRATION field is evaluated. If the message is expired, it MUST be discarded.

2. If the BTYPE is supported, then the BLOCK MUST be validated against the requested BTYPE. To do this, the peer checks that the block is valid using ValidateBlockStoreRequest. If the result is BLOCK_INVALID, the message MUST be discarded.

3. If the PUTPATH_L or the GETPATH_L are non-zero, the local peer SHOULD verify the signatures from the PUTPATH and the GETPATH. Verification MAY involve checking all signatures or any random subset of the signatures. It is RECOMMENDED that peers adapt their behavior to available computational resources so as to not make signature verification a bottleneck. If an invalid signature is found, the path MUST be truncated to only include the elements following the invalid signature. In particular, any invalid signature on the GETPATH will cause PUTPATH_L to be set to zero.

4. The peer also attempts to compute the key using DeriveBlockKey. This may result in NONE. The result is used later. Note that even if a key was computed, it does not have to match the QUERY_HASH.

5. If the BTYPE of the message indicates a HELLO block, the peer SHOULD be considered for the local routing table by using the peer identity computed from the block using DeriveBlockKey. An implementation MAY choose to ignore the HELLO, for example because the routing table or the respective k-bucket is already full. If the peer is a suitable candidate for insertion, the local peer MUST try to establish a connection to the peer indicated in the HELLO block using the address information from the HELLO block and the underlay's TRY_CONNECT() function. The implementation MUST instruct the underlay to connect to all provided addresses using TRY_CONNECT() in order to make the underlay aware of multiple addresses for this connection. When a connection is established, the signal PEER_CONNECTED will cause the peer to be added to the respective k-bucket of the routing table (see Section 6).

6. If the QUERY_HASH of this ResultMessage does not match an entry in the pending table (Section 6.5), then the message is discarded and processing ends. Otherwise, processing continues for each entry in the table as follows.

   a) If the FindApproximate flag was not set in the query and the BTYPE enabled the implementation to compute the key from the block, the computed key must exactly match the QUERY_HASH, otherwise the result does not match the pending query and processing continues with the next pending table entry.

   b) If the BTYPE is supported, result block MUST be validated against the specific query using the respective FilterBlockResult function. This function MUST update the result filter if a result is returned to the originator of the query.

   c) If the BTYPE is not supported, filtering of exact duplicate replies MUST still be performed before forwarding the reply. Such duplicate filtering MAY be implemented probabilistically, for example using a Bloom filter. The result of this duplicate filtering is always either FILTER_MORE or FILTER_DUPLICATE.

   d) If the RecordRoute flag is set in FLAGS, the local peer identity MUST be appended to the GETPATH of the message and the respective signature MUST be set using the query origin as the PEER SUCCESSOR and the response origin as the PEER PREDECESSOR. If the flag is not set, the GETPATH_L and PUTPATH_L MUST be set to zero when forwarding the result.

e) If the result filter result is either FILTER_MORE or FILTER_LAST, the message is forwarded to the origin of the query as defined in the entry which may either be the local peer or a remote peer. In case this is a query of the local peer the result may have to be provided to applications through the overlay API. Otherwise, the result is forwarded using SEND(P, ResultMessage') where ResultMessage' is the now modified message. If the result was FILTER_LAST, the query MUST be removed from the pending table.

8. Finally, the implementation SHOULD cache ResultMessages in order to provide already seen replies to future GetMessages. The implementation MAY choose not no cache any or a limited number of ResultMessages for reasons such as resource limitations.

8. Blocks

This section describes various considerations R^5N implementations must consider with respect to blocks. Specifically, implementations SHOULD be validate and persist blocks. Implementations MAY not support validation for all types of blocks. For example, on some devices, storing blocks is impossible due to lack of storage capacity. Block storage improves lookup performance for local applications and also other peers. Not storing blocks results in degraded performance.

The block type determines the format and handling of the block payload by peers in PutMessages and ResultMessages. Applications can and should define their own block types. Block types MUST be registered with GANA (see Section 11.1). Especially when new block types are introduced, some peers MAY lack support for the respective block operations.

8.1. Block Operations

Block validation operations are used as part of message processing (see Section 7) for all types of DHT messages. To enable these validation operations, any block type specification MUST define the following functions:

ValidateBlockQuery(Key, XQuery) -> RequestEvaluationResult is used to evaluate the request for a block as part of GetMessage processing. Here, the block payload is unkown, but if possible the XQuery and Key SHOULD be verified. Possible values for the RequestEvaluationResult are:

REQUEST_VALID The query is valid.

REQUEST_INVALID The query format does not match the block type. For example, a mandatory XQuery was not provided, or of the size of the XQuery is not appropriate for the block type.

DeriveBlockKey(Block) -> Key | NONE is used to synthesize the block key from the block payload as part of PutMessage and ResultMessage processing. The special return value of NONE implies that this block type does not permit deriving the key from the block. A Key may be returned for a block that is ill-formed.

ValidateBlockStoreRequest(Block) -> BlockEvaluationResult is used to evaluate a block payload as part of PutMessage and ResultMessage processing. Possible values for the BlockEvaluationResult are:

BLOCK_VALID The block is valid.

BLOCK_INVALID The block payload does not match the block type.

SetupResultFilter(FilterSize, Mutator) -> RF is used to setup an empty result filter. The arguments are typically the size of the set of results that must be filtered at the initiator, and a Mutator value which MAY be used to deterministically re-randomize probabilistic data structures. RF MUST be a byte sequence suitable for transmission over the network.

FilterResult(Block, Key, RF, XQuery) -> (FilterEvaluationResult, RF') is used to filter results against specific queries. This function does not check the validity of Block itself or that it matches the given key, as this must have been checked earlier. Locally stored blocks from previously observed ResultMessages and PutMessages use this function to perform filtering based on the request parameters of a particular GET operation. Possible values for the FilterEvaluationResult are:

FILTER_MORE Block is a valid result, and there may be more.

FILTER_LAST The given Block is the last possible valid result.

FILTER_DUPLICATE Block is a valid result, but considered to be a duplicate (was filtered by the RF) and SHOULD NOT be returned to the previous hop. Peers that do not understand the block type MAY return such duplicate results anyway and implementations must take this into account.

FILTER_IRRELEVANT Block does not satisfy the constraints imposed

by the XQuery. The result SHOULD NOT be returned to the previous hop. Peers that do not understand the block type MAY return such irrelevant results anyway and implementations must take this into account.

If the main evaluation result is FILTER_MORE, the function also returns an updated result filter where the Block is added to the set of filtered replies. An implementation is not expected to actually differenciate between the FILTER_DUPLICATE and FILTER_IRRELEVANT return values: in both cases the Block is ignored for this query.

8.2. HELLO Blocks

For bootstrapping and peer discovery, the DHT implementation uses its own block type called "HELLO". HELLO blocks are the only type of block that MUST be supported by every R^5N implementation. A block with this block type contains the peer public key of the peer that published the HELLO together with a set of addresses of this peer. The key of a HELLO block is the SHA-512 hash [RFC4634] of the peer public key and thus the peer’s identity in the DHT.

The HELLO block type wire format is illustrated in Figure 13. A query for block of type HELLO MUST NOT include extended query data (XQuery). Any implementation encountering a request for a HELLO with non-empty XQuery data MUST consider the request invalid and ignore it.

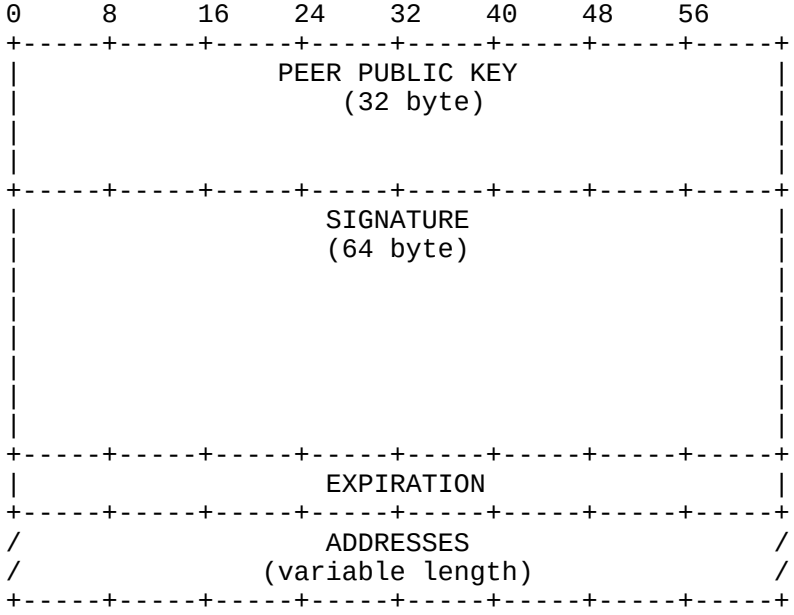


Figure 13: The HELLO Block Format.

PEER PUBLIC KEY is the public key of the peer to which the ADDRESSES belong. This is also the public key needed to verify the SIGNATURE.

EXPIRATION denotes the absolute 64-bit expiration date of the content. The value specified is microseconds since midnight (0 hour), January 1, 1970 in network byte order, but must be a multiple of one million (so that it can be represented in seconds in a HELLO URL).

ADDRESSES is a list of UTF-8 addresses (Section 2) which can be used to contact the peer. Each address MUST be 0-terminated. The set of addresses MAY be empty, for example if the peer knows that it cannot be reached from the outside (i.e. NAT).

SIGNATURE is the EdDSA signature [ed25519] of the HELLO block. The signature covers various information derived from the actual data in the HELLO block. The data signed over includes the block expiration time, a constant that uniquely identifies the purpose of the signature, and a hash of the addresses with 0-terminators in the same order as they are present in the HELLO block. The format is illustrated in Figure 14.

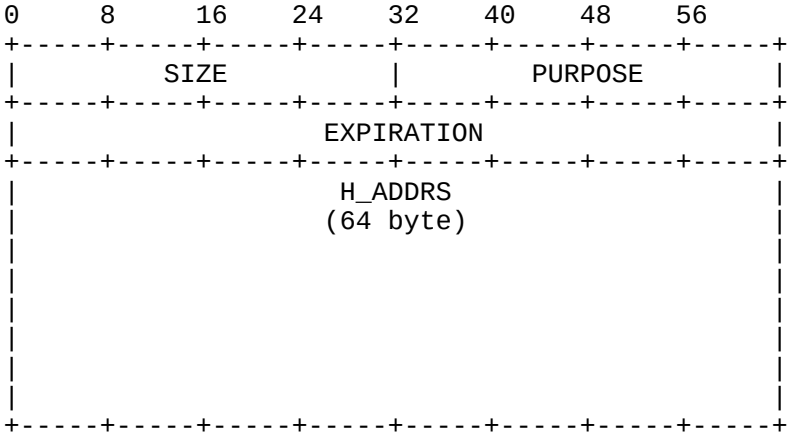


Figure 14: The Wire Format of the HELLO for Signing.

SIZE A 32-bit value containing the length of the signed data in bytes in network byte order. The length of the signed data MUST be 80 bytes.

PURPOSE A 32-bit signature purpose flag. This field MUST be 7 in network byte order.

EXPIRATION denotes the absolute 64-bit expiration date of the HELLO in microseconds since midnight (0 hour), January 1, 1970 in network byte order.

H_ADDRS a SHA-512 hash over the addresses in the HELLO. H_ADDRS is generated over the ADDRESSES field as provided in the HELLO block using SHA-512 [RFC4634].

The HELLO block operations MUST be implemented as follows:

ValidateBlockQuery(Key, XQuery) -> RequestEvaluationResult To validate a block query for a HELLO is to simply check that the XQuery is empty. If it is empty, REQUEST_VALID ist returned. Otherwise, REQUEST_INVALID is returned.

DeriveBlockKey(Block) -> Key | NONE To derive a block key for a HELLO is to simply hash the PEER PUBLIC KEY from the HELLO. The result of this function is thus always the SHA-512 hash over the PEER PUBLIC KEY.

ValidateBlockStoreRequest(Block) -> BlockEvaluationResult To

validate a block store request is to verify the EdDSA SIGNATURE over the hashed ADDRESSES against the public key from the PEER PUBLIC KEY field. If the signature is valid BLOCK_VALID is returned. Otherwise BLOCK_INVALID is returned.

SetupResultFilter(FilterSize, Mutator) -> RF The RF for HELLO blocks is implemented using a Bloom filter following the definition from Appendix A and consists of a variable number of bits L. L depends on the FilterSize which will be the number of connected peers |E| known to the peer creating a HELLO block from its own addresses: L is set to the minimum of 2^18 bits (2^15 bytes) and the lowest power of 2 that is strictly larger than 2*K*|E| bits (K*|E|/4 bytes).

The k-value for the Bloom filter MUST be 16. The elements used in the Bloom filter consist of an XOR between the H_ADDRS field (as computed using SHA-512 over the ADDRESSES) and the SHA-512 hash of the MUTATOR field from a given HELLO block. The mapping function M(H_ADDRS XOR MUTATOR) is defined as follows:

M(e = H_ADDR XOR MUTATOR) -> e as uint32[]

M is an identity function and returns the 512-bit XOR result unmodified. This resulting byte string is interpreted as k=16 32-bit integers in network byte order which are used to set and check the bits in B using BF-SET() and BF-TEST(). The 32-bit MUTATOR is prepended to the L-bit Bloom filter field HELLO_BF containing B to create the result filter for a HELLO block:

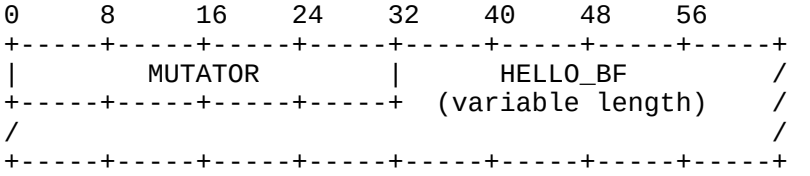


Figure 15: The HELLO Block Result Filter.

where:

MUTATOR The 32-bit mutator for the result filter.

HELLO_BF The L-bit Bloom filter array.

The MUTATOR value is used to additionally "randomize" the computation of the Bloom filter while remaining deterministic across peers. It is only ever set by the peer initiating the GET request, and changed every time the GET request is repeated.

Peers forwarding GET requests MUST not change the mutator value included in the RESULT_FILTER as they might not be able to recalculate the result filter with a different MUTATOR value.

Consequently, repeated requests have statistically independent probabilities of creating false-positives in a result filter. Thus, even if for one request a result filter may exclude a result as a false-positive match, subsequent requests are likely to not have the same false-positives.

HELLO result filters can be merged if the Bloom filters have the same size and MUTATOR by setting all bits to 1 that are set in either Bloom filter. This is done whenever a peer receives a query with the same MUTATOR, predecessor and Bloom filter size.

FilterResult(Block, Key, RF, XQuery) -> (FilterEvaluationResult, RF') The H_ADDRS field is XORed with the SHA-512 hash of the MUTATOR field from the HELLO block and the resulting value is checked against the Bloom filter in RF. Consequently, HELLOs with completely identical sets of addresses will be filtered and FILTER_DUPLICATE is returned. Any small variation in the set of addresses will cause the block to no longer be filtered (with high probability) and FILTER_MORE is returned.

8.3. Persistence

An implementation SHOULD provide a local persistence mechanism for blocks. Embedded systems that lack storage capability MAY use connection-level signalling to indicate that they are merely a client utilizing a DHT and are not able to participate with storage. The local storage MUST provide the following functionality:

Store(Key, Block) Stores a block under the specified key. If an block with identical payload exists already under the same key, the meta data should be set to the maximum expiration time of both blocks and use the corresponding PUTPATH (and if applicable TRUNCATED ORIGIN) of that version of the block.

Lookup(Key) -> Set of Blocks Retrieves blocks stored under the specified key.

LookupApproximate(Key) -> List of Blocks Retrieves the blocks stored under the specified key and any blocks under keys close to the specified key, in order of decreasing proximity.

8.3.1. Approximate Search Considerations

Over time a peer may accumulate a significant number of blocks which are stored locally in the persistence layer. Due to the expected high number of blocks, the method to retrieve blocks close to the specified lookup key in the LookupApproximate API must be implemented with care with respect to efficiency.

It is RECOMMENDED to limit the number of results from the LookupApproximate procedure to a result size which is easily manageable by the local system. The RECOMMENDED default is to return blocks with the four closest keys. Note that filtering by the RF will be done by the DHT afterwards and it is NOT RECOMMENDED to fetch additional records even if all four closest keys are filtered by the RF. The main reason for this is to ensure peers do not spend extensive resources to process approximate lookups. In particular, implementations MUST limit the worst-case effort they spent on approximate lookups.

In order to efficiently find a suitable result set, the implementation SHOULD follow the following procedure:

1. Sort all blocks by the block key in ascending (decending) order. The block keys are interpreted as integers.

2. Alternatingly select a block with a key larger and smaller from the sortings. The resulting set is then sorted by the XOR distance to the query. The selection process continues until the upper bound for the result set is reached and both sortings do not yield any closer blocks.

An implementation MAY decide to use a custom algorithm in order to find the closest blocks in the local storage. But, especially for more primitive approaches (such as only comparing XOR distances for all blocks in the storage), more simplistic procedures may become ineffective for large data sets and care MUST be taken to strictly bound the maximum effort expended per query.

8.3.2. Caching Strategy Considerations

An implementation MUST implement an eviction strategy for blocks stored in the block storage layer.

In order to ensure the freshness of blocks, an implementation MUST evict expired blocks in favor of new blocks.

An implementation MAY preserve blocks which are often requested. This approach can be expensive as it requires the implementation to keep track of how often a block is requested.

An implementation MAY preserve blocks which are close to the local peer public key.

An implementation MAY provide configurable storage quotas and adapt its eviction strategy based on the current storage size or other constrained resources.

9. Security Considerations

If an upper bound to the maximum number of neighbors in a k-bucket is reached, the implementation MUST prefer to preserve the oldest working connections instead of new connections. This makes Sybil attacks [Sybil] less effective as an adversary would have to invest more resources over time to mount an effective attack.

The ComputeOutDegree function limits the REPL_LVL to a maximum of 16. This imposes an upper limit on bandwidth amplification an attacker may achieve for a given network size and topology.

9.1. Disjoint Underlay or Application Protocol Support

We note that peers implementing disjoint sets of underlay protocols may experience difficulties communicating (unless other peers bridge the respective underlays). Similarly, peers that do not support a particular application will not be able to validate application-specific payloads and may thus be tricked into storing or forwarding corrupt blocks.

9.2. Approximate Result Filtering

When a FindApproximate flag is encountered in a query, a peer will try to respond with the closest block it has that is not filtered by the result Bloom filter (RF). Implementations MUST ensure that the cost of evaluating any such query is reasonably small. For example, implementations SHOULD consider ways to avoid an exhaustive search of their database. Not doing so can lead to denial of service attacks as there could be cases where too many local results are filtered by the result filter.

9.3. Access Control

By design R^5N does not rely on strict admission control through the use of either centralized enrollment servers or pre-shared keys. This is a key distintion over protocols that do rely on this kind of access control such as [RFC6940] which, like R^5N, provides a peer-to-peer (P2P) signaling protocol with extensible routing and topology mechanisms. Some decentralized applications, such as the GNU Name System ([RFC9498]), require an open system that enables ad-hoc participation.

9.4. Block-level confidentiality and privacy

Applications using the DHT APIs to store application-specific block types may have varying security and privacy requirements. R^5N does NOT provide any kind confidentiality or privacy for blocks, for example through the use of cryptography. This must be provided by the application as part of the block type design. One example where confidentiality and privacy are required are GNS records and their record blocks as defined in [RFC9498]. Other possibilities to protect the block objects may be implemented using ideas from other efforts such as Oblivious HTTP and its encapsulation of HTTP requests and responses [RFC9458].

9.5. Protocol extensions and cryptographic agility

R^5N makes heavy use of the Ed25519 cryptosystem. It cannot be ruled out that the relevant primitives are broken at any point in the future. In this case, the R^5N design can be reused by modifying the messages and related artifacts defined in Section 7.1. In order to extend and modify the R^5N protocol in general and to replace cryptographic primitives in particular, new message types (MTYPE fields) can be registered in [GANA] and the message formats updated accordingly. Peers processing messages MUST NOT modify the MTYPE field in order to prevent possible security downgrades.

9.6. Availability versus security tradeoffs in routing table evictions

R^5N does not implement locality-sensitive as it does not preferentially evict distant nodes (where distance is a metric based on closeness in the physical network). Locality-sensitive routing table eviction may offer performance improvements especially if the local network and its resources can be leveraged more efficiently. Similarly, if requests (and responses) can be contained to the local network, this can offer better privacy. But, this an important security trade-off when choosing network locality over R^5N’s eviction strategy (Section 6.1): "Flash mob"-style attackers that quickly spin up a large number of nodes a target node’s proximity are

displacing legitimate, benign neighbours. In case of the R^5N eviction strategy these will likely not degrade the routing table to the same degree because long-lived connections are preferred. This in turn forces an attacker to run their nodes for a long time to run a successful attack.

It is important to highlight that in order to address the R^5N threat and security model (Section 1.3), the routing starts with a random walk. Should all nodes implement a locality sensitive eviction strategy, the theoretical effectiveness of this measure would drastically decrease. R^5N puts security and availability under its threat model, over performance and privacy.

It should be noted that any reasonable locality metric will choose nodes that implicitly provide more stable network connections than distant nodes as the probability for network failures grows with physical distance. As a consequence it can be assumed that a locality-sensitive metric and R^5N's eviction strategy eventually converge into a similar situation where a node primarily maintains a routing table consisting of long-lived and somewhat local connections.

10. IANA Considerations

IANA maintains a registry called the "Uniform Resource Identifier (URI) Schemes" registry. The registry should be updated to include an entry for the 'gnunet' URI scheme. IANA is requested to update that entry to reference this document when published as an RFC.

11. GANA Considerations

11.1. Block Type Registry

GANA [GANA] is requested to create a "DHT Block Types" registry. The registry shall record for each entry:

* Name: The name of the block type (case-insensitive ASCII string, restricted to alphanumeric characters
* Number: 32-bit
* Comment: Optionally, a brief English text describing the purpose of the block type (in UTF-8)
* Contact: Optionally, the contact information of a person to contact for further information

* References: Required, references (such as an RFC) specifying the block type and its block functions

The registration policy for this sub-registry is "First Come First Served", as described in [RFC8126]. GANA created the registry as follows:

Number	Name	References	Description
0	ANY	[This.I-D]	Reserved
13	DHT_HELLO	[This.I-D]	Address data for a peer

Contact: r5n-registry@gnunet.org

Figure 16: The Block Type Registry.

11.2. GNUnet URI Schema Subregistry

GANA [GANA] is requested to create a "gnunet://" sub-registry. The registry shall record for each entry:

* Name: The name of the subsystem (case-insensitive ASCII string, restricted to alphanumeric characters)

* Comment: Optionally, a brief English text describing the purpose of the subsystem (in UTF-8)

* Contact: Optionally, the contact information of a person to contact for further information

* References: Optionally, references describing the syntax of the URL (such as an RFC or LSD)

The registration policy for this sub-registry is "First Come First Served", as described in [RFC8126]. GANA created this registry as follows:

Name	References	Description
HELLO	[This.I-D]	How to contact a peer.
ADDRESS	N/A	Network address.

Contact: gnunet-registry@gnunet.org

Figure 17: GNUnet scheme Subregistry.

11.3. GNUnet Signature Purpose Registry

GANA amended the "GNUnet Signature Purpose" registry as follows:

Purpose	Name	References	Description
6	DHT PATH ELEMENT	[This.I-D]	DHT message routing data
7	HELLO PAYLOAD	[This.I-D]	Peer contact information

Figure 18: The Signature Purpose Registry Entries.

11.4. GNUnet Message Type Registry

GANA is requested to amend the "GNUnet Message Type" registry as follows:

Type	Name	References	Description
146	DHT PUT	[This.I-D]	Store information in DHT
147	DHT GET	[This.I-D]	Request information from DHT
148	DHT RESULT	[This.I-D]	Return information from DHT
157	HELLO Message	[This.I-D]	Peer contact information

Figure 19: The Message Type Registry Entries.

12. Implementation and deployment status

There are two implementations conforming to this specification written in C and Go, respectively. The C implementation as part of GNUnet [GNUnetGNS] represents the original and reference implementation. The Go implementation [GoGNS] demonstrates how two implementations of GNS are interoperable if they are built on top of the same underlying DHT storage.

Currently, the GNUnet peer-to-peer network [GNUnet] is an active deployment of R^5N. It’s primary but not only purpose is to provide the storage underlay for the GNU Name System [RFC9498]. The [GoGNS] implementation shows how R^5N implementations can interoperate.

13. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC2663] Srisuresh, P. and M. Holdrege, "IP Network Address Translator (NAT) Terminology and Considerations", RFC 2663, DOI 10.17487/RFC2663, August 1999, <https://www.rfc-editor.org/info/rfc2663>.

[RFC3561] Perkins, C., Belding-Royer, E., and S. Das, "Ad hoc On-Demand Distance Vector (AODV) Routing", RFC 3561, DOI 10.17487/RFC3561, July 2003, <https://www.rfc-editor.org/info/rfc3561>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC4634] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and HMAC-SHA)", RFC 4634, DOI 10.17487/RFC4634, July 2006, <https://www.rfc-editor.org/info/rfc4634>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC5245] Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", RFC 5245, DOI 10.17487/RFC5245, April 2010, <https://www.rfc-editor.org/info/rfc5245>.

[RFC6940] Jennings, C., Lowekamp, B., Ed., Rescorla, E., Baset, S., and H. Schulzrinne, "REsource LOcation And Discovery (RELOAD) Base Protocol", RFC 6940, DOI 10.17487/RFC6940, January 2014, <https://www.rfc-editor.org/info/rfc6940>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC9458] Thomson, M. and C. A. Wood, "Oblivious HTTP", RFC 9458, DOI 10.17487/RFC9458, January 2024, <https://www.rfc-editor.org/info/rfc9458>.

[RFC9498] Schanzenbach, M., Grothoff, C., and B. Fix, "The GNU Name System", RFC 9498, DOI 10.17487/RFC9498, November 2023, <https://www.rfc-editor.org/info/rfc9498>.

[ed25519] Bernstein, D., Duif, N., Lange, T., Schwabe, P., and B. Yang, "High-Speed High-Security Signatures", 2011, <http://link.springer.com/chapter/10.1007/978-3-642-23951-9_9>.

[GANA] GNUnet e.V., "GNUnet Assigned Numbers Authority (GANA)", April 2020, <https://gana.gnunet.org/>.

14. Informative References

[R5N] Evans, N. S. and C. Grothoff, "R5N: Randomized recursive routing for restricted-route networks", 2011, <https://doi.org/10.1109/ICNSS.2011.6060022>.

[NSE] Evans, N. S., Polot, B., and C. Grothoff, "Efficient and Secure Decentralized Network Size Estimation", 2012, <https://doi.org/10.1007/978-3-642-30045-5_23>.

[Kademlia] Maymounkov, P. and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric.", 2002, <http://css.csail.mit.edu/6.824/2014/papers/kademlia.pdf>.

[Sybil] Douceur, J., "The Sybil Attack", 2002, <https://link.springer.com/chapter/10.1007/3-540-45748-8_24>.

[Smallworldmix] Dell'Amico, M., "A Measurement of Mixing Time in Social Networks", 2009, <https://api.semanticscholar.org/CorpusID:44240877>.

[Smallworld] Kleinberg, J., "Navigation in a small world", 2000, <https://www.nature.com/articles/35022643>.

[cadet] Polot, B. and C. Grothoff, "CADET: Confidential ad-hoc decentralized end-to-end transport", 2014, <https://doi.org/10.1109/MedHocNet.2014.6849107>.

[GNUnet] GNUnet e.V., "The GNUnet Project", <https://gnunet.org>.

[GNUnetGNS] GNUnet e.V., "The GNUnet GNS Implementation", <https://git.gnunet.org/gnunet.git/tree/src/gns>.

[GoGNS] Fix, B., "The Go GNS Implementation", <https://github.com/bfix/gnunet-go/tree/master/src/gnunet/service/gns>.

Appendix A. Bloom filters in R^5N

R^5N uses Bloom filters in several places. This section gives some general background on Bloom filters and defines functions on this data structure shared by the various use-cases in R^5N.

A Bloom filter (BF) is a space-efficient probabilistic datastructure to test if an element is part of a set of elements. Elements are identified by an element ID. Since a BF is a probabilistic datastructure, it is possible to have false-positives: when asked if an element is in the set, the answer from a BF is either "no" or "maybe".

Bloom filters are defined as a string of L bits. The bits are initially always empty, meaning that the bits are set to zero. There are two functions which can be invoked on the Bloom filter "bf": BF-SET(bf, e) and BF-TEST(bf, e) where "e" is an element that is to be added to the Bloom filter or queried against the set.

A mapping function M is used to map each ID of each element from the set to a subset of k bits. In the original proposal by Bloom, M is non-injective and can thus map the same element multiple times to the same bit. The type of the mapping function can thus be described by the following mathematical notation:

```
-------------------------------------
# M: E->B^k
-------------------------------------
# L = Number of bits
# B = 0,1,2,3,4,...L-1 (the bits)
# k = Number of bits per element
# E = Set of elements
-------------------------------------
Example: L=256, k=3
M('element-data') = {4,6,255}
```

Figure 20: Bloom filter mapping function.

When adding an element to the Bloom filter bf using BF-SET(bf, e), each integer n of the mapping M(e) is interpreted as a bit offset n mod L within bf and set to 1.

When testing if an element may be in the Bloom filter bf using BF-TEST(bf,e), each bit offset n mod L within bf MUST have been set to 1. Otherwise, the element is not considered to be in the Bloom filter.

Appendix B. Overlay Operations

An implementation of this specification commonly exposes the two overlay operations "GET" and "PUT". The following are non-normative examples of APIs for those operations. Their behaviour is described prosaically in order to give implementers a fuller picture of the protocol.

B.1. GET operation

A basic GET operation interface may be exposed as:

GET(Query-Key, Block-Type) -> Results as List

The procedure typically takes at least two arguments to initiate a lookup:

QueryKey: is the 512-bit key to look for in the DHT.

Block-Type: is the type of block to look for, possibly "any".

The GET procedure may allow additional optional parameters in order to control or modify the query:

Replication-Level: is an integer which controls how many nearest peers the request should reach.

Flags: is a 16-bit vector which indicates certain processing requirements for messages. Any combination of flags as defined in Section 7.1.1 may be specified.

eXtended-Query (XQuery): is medatadata which may be required

depending on the respective Block-Type. A Block-Type must define if the XQuery can or must be used and what the specific format of its contents should be. Extended queries are in general used to implement domain-specific filters. These might be particularly useful in combination with FindApproximate to add a well-defined filter by an application-specific distance. Regardless, the DHT does not define any particular semantics for an XQuery. See also Section 8.

Result-Filter: is data for a Block-type-specific filter which allows applications to indicate results which are not relevant anymore to the caller (see Section 7.4.2).

The GET procedure should be implemented as an asynchronous operation that returns individual results as they are found in the DHT. It should terminate only once the application explicitly cancels the operation. A single result commonly consists of:

Block-Type: is the desired type of block in the result.

Block-Data: is the application-specific block payload. Contents are specific to the Block-Type.

Block-Expiration: is the expiration time of the block. After this time, the result should no longer be used.

Key: is the key under which the block was stored. This may be different from the key that was queried if the flag FindApproximate was set.

GET-Path: is a signed path of the public keys of peers which the query traversed through the network. The DHT will try to make the path available if the RecordRoute flag was set by the application calling the PUT procedure. The reported path may have been truncated from the beginning. The API SHOULD signal truncation by exposing the Truncated flag.

PUT-Path: is a signed path of the public keys of peers which the result message traversed. The DHT will try to make the path available if the RecordRoute flag was set for the GET procedure. The reported path may have been truncated from the beginning. The API SHOULD signal truncation by exposing the Truncated flag. As the block was cached by the node at the end of this path, this path is more likely to be stale compared to the GET-Path.

Truncated: is true if the GET-Path or PUT-Path was truncated, otherwise false.

B.2. PUT operation

A PUT operation interface may be exposed as:

PUT(Key, Block-Type, Block-Expiration, Block-Data)

The procedure typically takes at least four parameters:

Key: is the key under which to store the block.

Block-Type: is the type of the block to store.

Block-Expiration: specifies when the block should expire.

Block-Data: is the application-specific payload of the block to store.

The PUT procedure may accept additional optional parameters that control or modify the operation:

Replication-Level: is an integer which controls how many nearest peers the request should reach.

Flags: is a bit-vector which indicates certain processing requirements for messages. Any combination of flags as defined in Section 7.1.1 may be specified.

The PUT procedure does not necessarily yield any information.

Appendix C. HELLO URLs

The general format of a HELLO URL uses "gnunet://" as the scheme, followed by "hello/" for the name of the GNUnet subsystem, followed by "/"-separated values with the GNS Base32 encoding [RFC9498] of the peer public key, a Base32-encoded EdDSA signature [ed25519], and an expiration time in seconds since the UNIX Epoch in decimal format. After this a "?" begins a list of key-value pairs where the key is the URI scheme of one of the peer’s addresses and the value is the URL-escaped payload of the address URI without the "://".

The general syntax of HELLO URLs specified using Augmented Backus-Naur Form (ABNF) of [RFC5234] is:

```
hello-URL = "gnunet://hello[:version]/" meta [ "?" addrs ]
version = *(DIGIT)
meta = pid "/" sig "/" exp
pid = *bchar
sig = *bchar
exp = *DIGIT
addrs = addr *( "&" addr )
addr = addr-name "=" addr-value
addr-name = scheme
addr-value = *pchar
bchar = *(ALPHA / DIGIT)
```

Figure 21

’scheme’ is defined in [RFC3986] in Section 3.1. ’pchar’ is defined in [RFC3986], Appendix A.

For example, consider the following URL:

```
gnunet://hello/1MVZC83SFHXMADVJ5F4
S7BSM7CCGFNVJ1SMQPGW9Z7ZQBZ689ECG/
CFJD9SY1NY5VM9X8RC5G2X2TAA7BCVCE16
726H4JEGTAEB26JNCZKDHBPSN5JD3D60J5
GJMHFJ5YGRGY4EYBP0E2FJJ3KFEYN6HYM0G/
1708333757?foo=example.com&bar+baz=1.2.3.4%3A5678%2Ffoo
```

Figure 22

It specifies that the peer with the pid "1MVZ..." is reachable via "foo" at "example.com" and "bar+baz" at "1.2.3.4" on port 5678 until 1708333757 seconds after the Epoch. Note that "foo" and "bar+baz" here are underspecified and just used as a simple example. In practice, the addr-name refers to a scheme supported by a DHT underlay.

Authors’ Addresses

Martin Schanzenbach
Fraunhofer AISEC
Lichtenbergstrasse 11
85748 Garching
Germany
Email: martin.schanzenbach@aisec.fraunhofer.de

Christian Grothoff
Berner Fachhochschule
Hoeheweg 80
CH-2501 Biel/Bienne
Switzerland
Email: grothoff@gnunet.org

Bernd Fix
GNUnet e.V.
Boltzmannstrasse 3
85748 Garching
Germany
Email: fix@gnunet.org