REsource LOcation And Discovery (RELOAD)
draft-bryan-p2psip-reload-03

Status of this Memo

By submitting this Internet-Draft, each author represents that any applicable patent or other IPR claims of which he or she is aware have been or will be disclosed, and any of which he or she becomes aware will be disclosed, in accordance with Section 6 of BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on August 27, 2008.

Copyright Notice

Copyright (C) The IETF Trust (2008).
Abstract

This document defines REsource LOcation And Discovery (RELOAD), a peer-to-peer (P2P) binary signaling protocol for use on the Internet. A P2P signaling protocol provides its clients with an abstract hash table service between a set of cooperating peers that form the overlay network. RELOAD is designed to support a P2P Session Initiation Protocol (P2PSIP) network, but can be utilized by other applications with similar requirements by defining new usages that specify the data kinds that must be stored for a particular application. RELOAD defines a security model based on a certificate enrollment service that provides unique identities. NAT traversal is a fundamental service of the protocol. RELOAD also allows access from "client" nodes which do not need to route traffic or store data for others.

Table of Contents

1. Introduction .............................................. 6
   1.1. Architecture ....................................... 7
       1.1.1. Usage Layer .................................. 8
       1.1.2. Overlay Routing and Storage Layer ............ 9
       1.1.3. Forwarding Layer .............................. 10
   1.2. Security ............................................... 10
2. Terminology .............................................. 11
3. Overview .................................................. 12
   3.1. Distributed Storage Layer ........................... 13
       3.1.1. DHT Concepts .................................. 13
       3.1.2. DHT Topology .................................. 14
       3.1.3. Routing ....................................... 14
       3.1.4. Storing and Retrieving Structured Data ....... 15
       3.1.5. Joining, Leaving, and Maintenance ............ 16
   3.2. Forwarding Layer .......................... 17
       3.2.1. Forming Direct Connections ..................... 17
       3.2.2. Via Lists ...................................... 18
       3.2.3. Clients ....................................... 19
   3.3. Transport Layer ................................. 20
   3.4. Enrollment ........................................ 20
       3.4.1. Certificate Issuance .......................... 21
       3.4.2. Bootstrap ..................................... 21
   3.5. Security .............................................. 21
       3.5.1. Certificate-Based Security ..................... 21
       3.5.2. Shared-Key Security ........................... 23
   3.6. Migration ............................................ 24
   3.7. Usages Layer ......................................... 24
       3.7.1. SIP Usage ...................................... 25
       3.7.2. Certificate Store Usage ....................... 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.7.3</td>
<td>TURN Usage</td>
<td>27</td>
</tr>
<tr>
<td>3.7.4</td>
<td>Diagnostic Usage</td>
<td>27</td>
</tr>
<tr>
<td>3.7.5</td>
<td>HIP Tunnel Usage</td>
<td>27</td>
</tr>
<tr>
<td>4.</td>
<td>Base Protocol</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Forwarding Header</td>
<td>28</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Changes to Forwarding Header</td>
<td>33</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Message Routing</td>
<td>33</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Fragmentation and Reassembly</td>
<td>35</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Route Logging</td>
<td>36</td>
</tr>
<tr>
<td>4.2</td>
<td>Message Contents Format</td>
<td>38</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Common Header</td>
<td>38</td>
</tr>
<tr>
<td>4.2.2.</td>
<td>Payload</td>
<td>39</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Signature</td>
<td>39</td>
</tr>
<tr>
<td>4.3</td>
<td>Response Codes and Response Errors</td>
<td>42</td>
</tr>
<tr>
<td>5.</td>
<td>End-to-End Timeout and Retransmission</td>
<td>43</td>
</tr>
<tr>
<td>6.</td>
<td>Transports</td>
<td>44</td>
</tr>
<tr>
<td>6.1</td>
<td>TLS</td>
<td>44</td>
</tr>
<tr>
<td>6.2</td>
<td>DTLS</td>
<td>44</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Reliability for Unreliable Transports</td>
<td>44</td>
</tr>
<tr>
<td>6.3</td>
<td>HIP</td>
<td>45</td>
</tr>
<tr>
<td>7.</td>
<td>Method Definitions</td>
<td>46</td>
</tr>
<tr>
<td>7.1</td>
<td>Connection Management</td>
<td>46</td>
</tr>
<tr>
<td>7.1.1</td>
<td>PING</td>
<td>46</td>
</tr>
<tr>
<td>7.1.2</td>
<td>CONNECT</td>
<td>48</td>
</tr>
<tr>
<td>7.1.3</td>
<td>TUNNEL</td>
<td>51</td>
</tr>
<tr>
<td>7.2</td>
<td>Data Storage and Retrieval</td>
<td>53</td>
</tr>
<tr>
<td>7.2.1</td>
<td>STORE</td>
<td>53</td>
</tr>
<tr>
<td>7.2.2</td>
<td>FETCH</td>
<td>60</td>
</tr>
<tr>
<td>7.2.3</td>
<td>REMOVE</td>
<td>64</td>
</tr>
<tr>
<td>7.2.4</td>
<td>FIND</td>
<td>65</td>
</tr>
<tr>
<td>7.3</td>
<td>DHT Maintenance</td>
<td>67</td>
</tr>
<tr>
<td>7.3.1</td>
<td>JOIN</td>
<td>67</td>
</tr>
<tr>
<td>7.3.2</td>
<td>LEAVE</td>
<td>68</td>
</tr>
<tr>
<td>7.3.3.</td>
<td>UPDATE</td>
<td>68</td>
</tr>
<tr>
<td>7.3.4.</td>
<td>ROUTE_QUERY</td>
<td>69</td>
</tr>
<tr>
<td>8.</td>
<td>ICE and Connection Formation</td>
<td>70</td>
</tr>
<tr>
<td>8.1</td>
<td>Overview</td>
<td>71</td>
</tr>
<tr>
<td>8.2</td>
<td>Collecting STUN Servers</td>
<td>72</td>
</tr>
<tr>
<td>8.3</td>
<td>Gathering Candidates</td>
<td>73</td>
</tr>
<tr>
<td>8.4</td>
<td>Encoding the CONNECT Message</td>
<td>73</td>
</tr>
<tr>
<td>8.5</td>
<td>Verifying ICE Support</td>
<td>74</td>
</tr>
<tr>
<td>8.6</td>
<td>Role Determination</td>
<td>74</td>
</tr>
<tr>
<td>8.7</td>
<td>Connectivity Checks</td>
<td>75</td>
</tr>
<tr>
<td>8.8</td>
<td>Concluding ICE</td>
<td>75</td>
</tr>
<tr>
<td>8.9</td>
<td>Subsequent Offers and Answers</td>
<td>75</td>
</tr>
<tr>
<td>8.10</td>
<td>Media Keepalives</td>
<td>75</td>
</tr>
<tr>
<td>8.11</td>
<td>Sending Media</td>
<td>75</td>
</tr>
<tr>
<td>8.12</td>
<td>Receiving Media</td>
<td>76</td>
</tr>
</tbody>
</table>
9. DHT Algorithms ............................................. 76
   9.1. Generic Algorithm Requirements ....................... 76
   9.2. Chord Algorithm ........................................ 76
   9.2.1. Overview ............................................. 76
   9.2.2. Routing ............................................. 77
   9.2.3. Redundancy .......................................... 77
   9.2.4. Joining ............................................. 77
   9.2.5. Routing CONNECTs ................................... 78
   9.2.6. UPDATEs ............................................. 79
   9.2.7. Leaving ............................................. 82
10. Enrollment and Bootstrap .................................... 82
   10.1. Discovery ............................................. 82
   10.2. Overlay Configuration .................................. 83
   10.3. Credentials ........................................... 84
   10.3.1. Credentials for HIP ................................ 85
   10.4. Locating a Peer ....................................... 85
11. Usages ....................................................... 86
   11.1. Generic Usage Requirements ............................ 86
   11.2. SIP Usage ............................................. 87
   11.2.1. SIP-REGISTRATION kind .............................. 87
   11.2.2. GRUU ............................................... 89
   11.2.3. SIP Connect ........................................ 89
   11.2.4. SIP Tunnel ........................................ 89
   11.3. TURN Usage ........................................... 90
   11.4. Certificate Store Usage ................................ 93
   11.5. HIP Tunnel ........................................... 94
   11.6. Diagnostic Usage ..................................... 94
   11.6.1. State Acquisition Mechanisms ...................... 95
   11.6.2. Black-box diagnostics .............................. 96
   11.6.3. Diagnostic Metrics for a P2PSIP Deployment ........ 98
12. Security Considerations ................................... 99
   12.1. Overview ............................................. 99
   12.2. Attacks on P2P Overlays ................................ 99
   12.3. Certificate-based Security ............................ 100
   12.4. Shared-Secret Security ................................ 101
   12.5. Storage Security ...................................... 101
   12.5.1. Authorization ...................................... 102
   12.5.2. Distributed Quota .................................. 102
   12.5.3. Correctness ....................................... 103
   12.5.4. Residual Attacks ................................... 103
   12.6. Routing Security ...................................... 104
   12.6.1. Background ........................................ 104
   12.6.2. Admissions Control ................................ 104
   12.6.3. Peer Identification and Authentication ............. 105
   12.6.4. Protecting the Signaling ........................... 105
   12.6.5. Residual Attacks ................................... 106
   12.7. SIP-Specific Issues ................................... 106
   12.7.1. Fork Explosion ..................................... 106
1. Introduction

This document defines REsource LOcation And Discovery (RELOAD), a peer-to-peer (P2P) signaling protocol for use on the Internet. It provides a Distributed Hash Table (DHT) service, which allows participating nodes to read and write entries into a hash table that is stored collectively among the participants. RELOAD is a lightweight, binary protocol. It provides several functions that are critical for a successful P2P protocol for the Internet. These are:

Security Framework: Security is one of the most challenging problems in a P2P protocol. A P2P network will often be established among a set of peers that do not trust each other. Yet, despite this lack of trust, the network must operate reliably to allow storage and retrieval of data. RELOAD defines an abstract enrollment server, which all entities trust to generate unique identifiers for each user. Using that small amount of trust as an anchor, RELOAD defines a security framework that allows for authorization of P2P protocol functions and authentication of data stored in the overlay. This does not remove all attacks but greatly reduces the possible attack space.

Usage Model: RELOAD is designed to support a variety of applications, including P2P multimedia communications with the Session Initiation Protocol [I-D.ietf-p2psip-concepts]. Consequently, RELOAD has the notion of a usage, one of which is defined to support each application (this document also defines the SIP usage for multimedia communications). Each usage identifies a set of data kind that need to be stored and retrieved from the DHT. Each kind defines a data structure, authorization policies, size quota, and information required for storage and retrieval in the DHT. The usage concept allows RELOAD to be used with new applications through a simple documentation process that supplies the details for each application.

NAT Traversal: Operations for NAT traversal are part of the base design, including establishing new RELOAD connections and tunneling SIP or other application protocols required by P2PSIP. RELOAD makes use of Interactive Connectivity Establishment (ICE) [I-D.ietf-mmusic-ice] to facilitate the creation of the P2P network and the establishment of channels for use by the application protocol (SIP and RTP, for example). RELOAD also defines how peers in the P2P network act as STUN and TURN servers and how those resources can be discovered through the DHT. With these features, RELOAD can run in modes in which nearly all the peers are behind NATs, yet are able to fully participate without imposing any constraints on the actual DHT algorithm.
High Performance Routing: The very nature of DHT algorithms introduces a requirement that peers participating in the P2P network route requests on behalf of other peers in the network. This introduces a load on those other peers, in the form of bandwidth and processing power. RELOAD has been defined to reduce the amount of bandwidth and processing required of peers. It does so by using a very lightweight binary protocol, and furthermore, by defining a packet structure that facilitates low-complexity forwarding, including hardware-based forwarding. In particular, a generic transport header is used for routing the message through the overlay without the contents needing to be parsed by (or even visible to) intermediate peers. The header includes no information about specific IP addresses because none are needed to route along an overlay. The header only includes lists of peers which the message should be routed through/to, as well as some minor options and version flags. Clearly separating the header components necessary for routing from the message contents simplifies processing and increases security.

Transport Flexibility: RELOAD has native support for both DTLS and TLS for the underlying transport protocol, with support for DTLS over UDP as mandatory to implement. TLS over TCP is preferred because it has better bulk data performance and connection stability, but UDP is more likely to provide direct connections between peers in the presence of NATs. Explicit support for fragmentation is provided and required when using UDP. Because there is no single universally available and suitable transport protocol, the peer protocol must be flexible in this regard. New transports can be supported trivially.

Pluggable DHT Algorithms: RELOAD has been designed with an abstract interface to the overlay layer to simplify implementing a variety of structured (DHT) and unstructured overlay algorithms. This specification also defines how RELOAD is used with Chord, which is mandatory to implement. Specifying a default "must implement" DHT will allow interoperability, while the extensibility allows selection of DHTs optimized for a particular application.

These properties were designed specifically to meet the requirements for a P2P protocol to support SIP. However, RELOAD is not limited to usage by SIP and could serve as a tool for supporting other P2P applications with similar needs. RELOAD is also based on the concepts introduced in [I-D.ietf-p2psip-concepts].

1.1. Architecture

Architecturally this specification is divided into several layers, as shown in the following figure.
The three layers defined by RELOAD include:

Usage Layer: Provides an application-specific interface that maps an application’s requirements onto the generic services of the DHT.

Overlay Routing & Storage Layer: Implements the overlay. Chooses what links to establish to form the DHT’s overlay network, manages the storage and migration of data for this peer and on behalf of other peers, and performs searches for requested data across the DHT.

Forwarding Layer: Provides packet forwarding services between nodes. Also handles setting up connections across NATs using ICE.

1.1.1. Usage Layer

The top layer, called the Usage Layer, has application usages, such as the SIP Location Usage, that use the abstract distributed storage API to store and retrieve data from the DHT. The goal of this layer is to implement application-specific usages of the Overlay Routing and Storage Layer below it. The Usage defines how a specific application maps its data into something that can be stored in the DHT, where to store the data, how to secure the data, and finally how applications can retrieve and use the data.

The architecture diagram shows both a SIP usage and an XMPP usage.
single application may require multiple usages. A usage may define multiple kinds of data that are stored in the overlay and may also rely on kinds originally defined by other usages. A usage is not itself encoded on the wire, only the kind-ids and data models are, but is rather a specification of the functionality that is required for a given application. That specification typically specifies semantics, access control rules, and the format and size of the data which may be stored.

One usage may depend on another. For example, the SIP usage depends on a Certificate Store usage (not shown in the diagram) to obtain the certificates required to authenticate messages. Because certificates are stored in standard X.509 form, there is no reason for each usage to specify this service independently.

### 1.1.2. Overlay Routing and Storage Layer

The Overlay Routing and Storage Layer stores and retrieves information, performs maintenance of the DHT as peers join and leave the DHT, and routes messages on the overlay. The DHT implementation is provided by a pluggable component so that each overlay can select an appropriate DHT that relies on the common RELOAD core code.

The Overlay Routing and Replication Logic provides a fairly generic interface that allows the DHT implementation to control the overlay and resource operations and messages. Since each DHT is defined and functions differently, we generically refer to the table of other peers that the DHT maintains and uses to route requests (neighbors) as a Routing Table. The Logic component makes queries to the DHT’s Routing Table to determine the next hop, then encodes and sends the message itself. Similarly, the DHT issues periodic update requests through the logic component to maintain and update its Routing Table.

The DHT shown in the illustration is Chord, but a variety of DHT algorithms are possible through a pluggable interface. A single node could be functioning in multiple overlays simultaneously, each using its own DHT algorithm. Each peer is identified by and its location in the overlay determined by its Peer-ID that is assigned by the enrollment server when the user or peer first enrolls in the overlay. The Peer-ID also determines the set of resources which it will be responsible for storing. The exact mapping between these is determined by the DHT algorithm used by the overlay, therefore the logic component always queries the DHT to determine where a particular resource should be stored.

As peers enter and leave, resources may be stored on different peers, so the information related to them is exchanged as peers enter and leave. Redundancy is used to protect against loss of information in
the event of a peer failure and to protect against compromised or subversive peers. The Logic component notifies the DHT as neighbors join and leave, and the DHT updates its Routing Table and issues resource migration requests as appropriate.

1.1.3. Forwarding Layer

This layer is responsible for getting a packet to the next peer, as determined by the Routing and Storage Layer. The Forwarding Layer establishes and maintains the network connections required by the DHT’s Routing Table. This layer is also responsible for setting up connections to other peers through NATs and firewalls using ICE, and it can elect to forward traffic using relays for NAT and firewall traversal.

1.2. Security

RELOAD provides two security mechanisms, one based on public key certificates and one based on a globally shared key.

RELOAD’s preferred security framework is built upon an enrollment server. The enrollment server issues each new peer a certificate that assigns it a Peer-ID. By generating the Peer-IDs randomly and controlling what peers are issued certificates, the enrollment server protects against many of the attacks on the overlay network. Similarly, all users are issued certificates for their identities by the enrollment server. All resources stored on the overlay must be signed by their creator, thus ensuring that an attacker cannot forge data belonging to another user. The enrollment process is only required to join the overlay (and perhaps to refresh an expired certificate). The peers and users do not need to have real-time access to the enrollment server.

TLS or DTLS are used for communication between peers. In combination with the certificates, this provides both confidentiality and authentication for communication across the overlay. Applications such as P2PSIP can also make use of the users’ certificates to achieve secure end-to-end connections at the application layer.

RELOAD also provides for an authorization framework based on certificates. Each usage defines the access control rules for which users/peers are allowed to read/write each Resource-ID. This protects data belonging to one user from being written by another, as well as allowing for a distributed quota mechanism based on the maximum size of each kind of data.

In addition to the enrollment server model, RELOAD offers a security model using a pre-shared-key. Although this provides significantly
less security than is provided through an enrollment server, it allows ad hoc or ephemeral overlays to be set up with minimal effort on the part of the users.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

We use the terminology and definitions from the Concepts and Terminology for Peer to Peer SIP [I-D.ietf-p2psip-concepts] draft extensively in this document. Other terms used in this document are defined inline when used and are also defined below for reference.

The following important terms from the Concepts document are defined below for reference.

DHT: A distributed hash table. A DHT is an abstract hash table service realized by storing the contents of the hash table across a set of peers.

DHT Algorithm: A DHT algorithm defines the rules for determining which peers in a DHT store a particular piece of data and for determining a topology of interconnections amongst peers in order to find a piece of data.

DHT Instance: A specific hash table and the collection of peers that are collaborating to provide read and write access to it. There can be any number of DHT instances running in an IP network at a time, and each operates in isolation of the others.

P2P Network: Another name for a DHT instance.

P2P Network Name: A string that identifies a unique P2P network. P2P network names are DNS names – for example, "example.org". Lookup of such a name in DNS returns services associated with the DHT, such as enrollment servers, bootstrap peers, or gateways (for example, a SIP gateway between a traditional SIP and a P2P SIP network called "example.com").

Resource-ID: A value that is not human friendly to read and identifies some resources and which is used as a key for storing and retrieving the resource. One way to generate a Resource-ID is by applying a mapping function to some other unique name (e.g., user name or service name) for the resource. The Resource-ID is used by the distributed database algorithm to determine the peer or peers that are responsible for storing the data for the overlay. In structured P2P networks, resource-IDs are generally fixed length and are formed by hashing the resource identifier. In unstructured networks, resource identifiers may be used directly as resource-IDs and may have variable length.
Peer: A host that is participating in the DHT. By virtue of its participation it can store data and is responsible for some portion of the overlay.
Peer-ID: A value that uniquely identifies a peer. Peer-IDs 0 and 2^N - 1 are reserved and are invalid peer-IDs. A value of zero is not used in the wire protocol but can be used to indicate an invalid peer in implementations and APIs. The peer-id of 2^N-1 is used on the wire protocol as a wildcard.
Resource: An object associated with a string identifier. In unstructured P2P networks, the identifier is used directly as a Resource-Id. In structured P2P networks the identifier can be mapped into a Resource-Id by using the string as the input to the hash function. A SIP resource, for example, is identified by its AOR.
User: A human being.

We also introduce the following important new terms.

Connection Table: The set of peers to which a peer is directly connected. This includes peers with which CONNECT handshakes have been done but which have not sent any UPDATES.
Routing Table: The set of peers which a peer can use to route DHT messages. In general, these peers will all be on the connection table but not vice versa, because some peers will have CONNECTed but not sent updates. Peers may send messages directly to peers which are on the connection table but may only route messages to other peers through peers which are on the routing table.
Hashed-ID: The generic term for an identifier in the hash space of the DHT. Examples of Hashed-IDs include Resource-IDs and Peer-IDs. This only applies to structured overlays.
Unhashed-ID: An Unhashed-ID is a string used as an input to a hash function, the result of which is a Hashed-ID. This only applies to structured overlays.
Usage: A usage is an application that wishes to use the DHT for some purpose. Each application wishing to use the DHT defines a set of data kinds that it wishes to use. The SIP usage defines the location, certificate, STUN server and TURN server data kinds.
Destination List: A list of IDs through which a message is to be routed. This allows for request/response source routing. A single ID is a trivial form of destination list.

3. Overview
3.1. Distributed Storage Layer

RELOAD is designed to be extensible to both structured and unstructured overlays. However, this version is only completely worked out for structured overlays such as DHTs. The following text assumes structured overlays; in particular Resource-IDs are assumed to be fixed length for any given overlay, although the protocol allows them to be variable length to allow extension to unstructured overlays.

Each logical address in the DHT where data can be stored is referred to as a Resource-ID. A given peer will be responsible for storing data from many Resource-ID locations. Typically literature on DHTs uses the term "key" to refer to a location in the DHT; however, in this specification the term key is used to refer to public or private keys used for cryptographic operations and the term Resource-ID is used to refer to a location for storage in the DHT.

3.1.1. DHT Concepts

While very early P2P systems used flood based techniques, some newer P2P systems locate resources using a Distributed Hash Table, or DHT to improve efficiency. Peers are organized using a Distributed Hash Table (DHT) structure. In such a system, every resource has a Resource-ID, which is obtained by hashing some keyword or value (an Unhashed-ID) that uniquely identifies the resource. Resources can be thought of as being stored in a hash table at the entry corresponding to their Resource-ID. The peers that make up the overlay network are also assigned an ID, called a Peer-ID, in the same hash space as the Resource-IDs. A peer is responsible for storing all resources that have Resource-IDs near the peer’s Peer-ID. The hash space is divided up so that all of the hash space is always the responsibility of some particular peer, although as peers enter and leave the system a particular peer’s area may change. Messages are exchanged between the peers in the DHT as the peers enter and leave to preserve the structure of the DHT and exchange stored entries. Various DHT implementations may visualize the hash space as a grid, circle, line, or hypercube.

Peers keep information about the location of other peers in the hash space and typically know about many peers nearby in the hash space, and progressively fewer more distant peers. We refer to this table of other peers as a Routing Table. When a peer wishes to operate on a resource it consults the list of peers it is aware of and contacts the peer with the Peer-ID nearest the desired Resource-ID. If that peer does not know how to find the resource, it either returns information about a closer peer it knows about, or forwards the request to a closer peer. In this fashion, the request eventually
reaches the peer responsible for the resource, which then replies to the requester.

3.1.2. DHT Topology

Each DHT will have a somewhat different structure, but many of the concepts are common. The DHT defines a large space of Resource-IDs, which can be thought of as addresses. In many DHTs, the Resource-IDs are simply 128- or 160-bit integers. Each DHT also has a distance metric such that we can say that Resource-ID A is closer to Resource-ID B than to Resource-ID C.

Each peer in the DHT is assigned a Peer-ID and is "responsible" for the nearby space of Resource-IDs. So, for instance, if we have a peer P, then it could also be responsible for storing data associated with Resource-ID P+epsilon as long as no other peer P was closer. The DHT Resource-ID space is divided so that some peer is responsible for each Resource-ID.

3.1.3. Routing

The way routing works in a DHT is specified by the specific DHT algorithm but the basic concepts are common to most systems. Each peer maintains connections to some other set of peers N. There need not be anything special about the peers in N, except that the peer has a direct connection to them: it can reach them without going through any other peer. When it wishes to deliver a message to some peer P, it selects some member of N, N_i that is closer to P than itself (as a degenerate case, P may be in N). The peer sends the message to N_i. At this point two things can happen:

Recursive Routing: N_i repeats the same process as P, sending the message to one of its peers N_j. This same process repeats until the message is delivered to N.

Iterative Routing: N_i consults its table of direct connections and selects a new peer N_j which is closer to N. It responds to the original sending peer with a redirect to N_j. The original peer then sends the message to N_j, where the process repeats until the sending peer is redirected to N.

The advantage of iterative routing is that it consumes less resources for the intermediate peers; they only have to send redirect messages rather than forwarding requests and responses. The advantage of recursive routing is that it does not require the sending or receiving peer to have a rich set of connections to other nodes in the overlay. Thus, iterative routing is problematic in NATed networks because there is no way to guarantee that a peer will be able to form a connection to whatever peer it is redirected to. In
RELOAD, iterative routing is supported using the ROUTE-QUERY request.

In most DHTs, the peers in N are selected in a particular way. One common strategy is to have them arranged exponentially further away from yourself so that any message can be routed in O(\log(N)) steps. The details of the routing structure depend on the DHT algorithm, however, since it defines the distance metric and the structure of the connection table.

In RELOAD, messages may either be REQUESTS or RESPONSES to REQUESTS. Requests are routed as described above. In principle, responses could be routed independently from requests. This is called "Asymmetric" routing because requests and responses will generally follow different paths through the network. Asymmetric routing makes diagnosis of errors difficult because you need to be able to acquire debugging information at multiple locations. In the alternative strategy, called "Symmetric" routing, as requests travel through the network they accumulate a history of the peers they passed through and responses are routed in the opposite direction so that they follow the same path in reverse. RELOAD supports both flavors of routing.

Symmetric routing is easier to debug. Symmetric routing is also required when the overlay topology is changing. For example, when a new peer is joining the overlay, asymmetric routing cannot work because the response would not be able to reach the new peer until it has completed the joining process. Symmetric routing solves this situation because the response is routed from the admitting peer through the bootstrap peer, thus relying on a path that is already known and established. In order to implement symmetric routing, RELOAD provides the Via List (Section 3.2.2) feature. Asymmetric routing, however, requires no state to be stored in the message (as a Via List) or in on-path peers.

[[TODO: again, this is a topic that needs WG discussion. It seems like there are situations where symmetric is very desirable (e.g., startup). It’s less clear that asymmetric will have a performance/state difference that will be significant.]]

[[TODO: it’s not clear that symmetric/asymmetric is really that evocative a terminology. Henning suggested key-basedTRACE-based but EKR doesn’t like that. Other suggestions?]]

3.1.4. Storing and Retrieving Structured Data

The Data Storage Layer provides operations to STORE, FETCH, and REMOVE data. Each location in the DHT is referenced by a single integer Resource-ID. However, each location may contain data
elements corresponding to multiple kinds (e.g., certificate, SIP registration). Similarly, there may be multiple elements of a given kind.

Each kind is identified by a kind-id, which is a code point assigned by IANA. Note that a kind may be employed by multiple usages and new usages are encouraged to use previously defined kinds where possible. As part of the kind definition, protocol designers may define constraints, such as limits on size, on the values which may be stored. For many kinds, the set may be restricted to a single value; some sets may be allowed to contain multiple identical items while others may only have unique items. We define the following data models in this document, though other usages can define their own structures:

- **single value**: There can be at most one item in the set and any value overwrites the previous item.
- **array**: Many values can be stored and addressed by index.
- **dictionary**: The values stored are indexed by a key. Often this key is one of the values from the certificate of the peer sending the STORE request.

### 3.1.5. Joining, Leaving, and Maintenance

When a new peer wishes to join the DHT, it must have a peer-id that it is allowed to use. It uses one of the peer-ids in the certificate it received from the enrollment server. The main steps in joining the DHT are:
First, the peer ("JP," for Joining Peer) uses the bootstrap procedures to find some (any) peer in the DHT. It then typically contacts the peer which would have formerly been responsible for the peer’s Resource-ID (since that is where in the DHT the peer will be joining), the Admitting Peer (AP). It copies the other peer’s state, including the data values it is now responsible for and the identities of the peers with which the other peer has direct connections.

The details of this operation depend mostly on the DHT involved, but a typical case would be:

1. JP sends a JOIN request to AP announcing its intention to join.
2. AP sends an JOIN response.
3. AP does a sequence of STOREs to JP to give it the data it will need.
4. AP does UPDATEs to JP and to other peers to tell it about its own routing table. At this point, both JP and AP consider JP responsible for some section of the DHT.
5. JP makes its own connections to the appropriate peers in the DHT.

After this process is completed, JP is a full member of the DHT and can process STORE/FETCH requests.

3.2. Forwarding Layer

The forwarding layer is responsible for looking at message and doing one of three things:

1. Deciding the message was destined for this peer and passing the message up to the layer above this.
2. Looking at the peer-id that represents the next peer to send the message too and if there is an existing connection, sending the message over the connection.
3. Requesting the DHT Routing logic to tell the forwarding layer which peer the message needs to be forwarded to (based on the target peer-id or resource-id), and then sending the message.

3.2.1. Forming Direct Connections

As described in Section 3.1.3, a peer maintains a set of direct connections to other peers in the DHT. Consider the case of a peer JP just joining the DHT. It communicates with the admitting peer AP
and gets the list of the peers in AP’s routing table. Naively, it could simply connect to the IP address listed for each peer, but this works poorly if some of those peers are behind a NAT or firewall. Instead, we use the CONNECT request to establish a connection.

Say that peer A wishes to form a direct connection to peer B. It gathers ICE candidates and packages them up in a CONNECT request which it sends to B through usual DHT routing procedures. B does its own candidate gathering and sends back a response with its candidates. A and B then do ICE connectivity checks on the candidate pairs. The result is a connection between A and B. At this point, A and B can add each other to their routing tables and send messages directly between themselves without going through other DHT peers.

In general, a peer needs to maintain connections to all of the peers near it in the DHT and to enough other peers to have efficient routing (the details depend on the specific DHT). If a peer cannot form a connection to some other peer, this isn’t necessarily a disaster; DHTs can route correctly even without fully connected links. However, a peer should try to maintain the specified link set and if it detects that it has fewer direct connections, should form more as required.

3.2.2. Via Lists

In a general messaging system, messages need a source and a destination and peers need to be able to send a response to the peer that sent the request. This can be particularly tricky in overlay networks when a new peer is joining, or the overlay network is stabilizing and different peers have different ideas on what the overlay topology is. A simple and reliable way to make sure that a response can reach the node that sent the request in these situations is to have the response traverse the reverse path of the request.

The approach used here is to have each node the request traverses add its peer-id to the "via list" in the request. Then the response is routed by looking at the list and using it as list of peers that the response will be routed thorough. To support this, each message has a destination list of nodes it needs to be routed through as well as a via list of what nodes it has traversed.

When a peer receives a message from the Transport Layer, it adds the peer-id of the node it received the message from to the end of the via list. When a peer goes to transmit a message to the Transport Layer, it looks at the first entry on the destination list. If the entry is this peer, it removes this entry from the list and looks at the next entry and if the entry is not this peer, it sends the message to the first peer on the destination list.
When a peer goes to send a response to a request, it can simply copy the via list in reverse to form the destination list for the response if it wishes to route the response along the reverse path as the request.

Peers that are willing to maintain state may do list compression for privacy reason and to reduce the message size. They do this by taking some number of entries off the via list and replacing them with a unique entry that this peer can later identify. Later, if the peer sees the unique entry in a destination list, it removes the unique entry and replaces it with the all the entries removed from the original via list (and reverses the order of these entries). Note that this technique will generally require storing some per-message state on the intermediate peer, so this is a bandwidth/per-peer state tradeoff. The exception is if the list is not compressed but rather the peer-ids are simply encrypted.

The via list approach provides several features. First it allows a response to follow the same path as the request. This is particularly important for peers that are sending requests while they are joining and before other peers can route to them as well as situations where message are being exchanged to stabilize the overlay network. It also makes it easier to diagnose and manage the system when all peers see the response to any request they forward.

### 3.2.3. Clients

RELOAD also allows for the possibility of client nodes. A client is a node with a peer-id which connects to an admitting peer (or peers) like an ordinary peer but never sends a JOIN or an UPDATE. It is therefore in the AP’s connection table but not its routing table and never is used to store any DHT data. However, because it is reachable through the AP, it can still send and receive messages. The client MUST still have the usual credentials. Also, because it never sends JOINs it is never responsible for storing data.

Because the client may only have a connection to a single AP, which, due to topology shifts may no longer be the responsible peer, clients SHOULD use symmetric routing and should advertise route lists that contain both the AP to which they are connected and themselves. E.g., if the client has peer-id $X$ and the AP has peer-id $Y$, the client should advertise the destination list $(Y, X)$. This guarantees reachability.

Note that clients MAY also contact APs which are not in fact responsible for the client’s peer-id.
3.3. Transport Layer

The transport layer sends and receives messages over TLS and DTLS. For TLS it simply pushes the messages into the stream. For DTLS it takes care of fragmentation issues. The reason for including TLS is the improved performance it can offer for bulk transport of data. The reason for including DTLS is that the percentage of the time that two devices behind NATs can form a direct connection without a relay is much higher for DTLS than for TLS. If all NATs were [I-D.ietf-behave-tcp] compliant, then TLS over TCP would be preferred.

3.4. Enrollment

Before a new user can join the DHT for the first time, they must enroll in the P2P Network for the DHT they want to join. Enrollment will typically be done by contacting a centralized enrollment server. Other approaches (for instance static out of band configuration) are possible but are outside the scope of this specification. During enrollment a new node learns about a particular overlay, sets up a names and credentials, and discovers the bootstrap nodes. This would typically be done when a new peer joined an overlay for the very first time. Bootstrap is the process that happens each time a node boots and is how the peer finds an node that can be used to join the overlay.

Before a node can join an overlay, it needs to be provided with a name for the overlay. Some examples are "example.com", "example", and "example.local". This name is resolved via a DNS SRV lookup for service name p2p_enroll and a protocol of tcp. If the TLD for the name is .local, then this DNS SRV lookup is done using [I-D.cheshire-dnsext-multicastdns] and the service name p2p_menroll. The intention here is to support ad hoc/local overlays. The resulting DNS lookup will provide the address of an enrollment server. Once this server is found, HTTPS is used to retrieve a XML file that contains the parameters for the overlay. These include things such as: what algorithms the overlay uses, overlay parameters, what usages are a peer on this overlay is required to support, the type of credentials required, addresses of credentials servers, the root certificate for the DHT, information about the DHT algorithm that is being used, a P2P-Network-Id that uniquely identifies this ring, and any other parameters it may need to connect to the DHT. The DHT also informs the peers what Usages it is required to support to be a peer on this P2P Network. It also provides an initial list of bootstrap nodes that consists of multiple bootstrap entries that each have the IP address and port for contacting a bootstrap peer. Some of the address may be multicast addresses. In the case of multicast DNS, every peer may also act as an enrollment server.
If shared-key security (Section 3.5.2) is being used, then the peer can proceed directly to bootstrap. If certificate-based security (Section 3.5.1) is being used, the peer MUST contact the credential server to obtain a certificate.

3.4.1. Certificate Issuance

Once the peer has the XML file that identifies if credentials are needed, it can contact the credential server. The user establishes his identity to the server’s satisfaction and provides the server with its public key. The centralized server then returns a certificate binding the user’s user name to his public key. The properties of the certificate are discussed in Section 3.5. The amount of authentication performed here can vary radically depending on the DHT network being joined. Some networks may do no verification at all and some may require extensive identity verification (e.g., checking a driver’s license) before issuing credentials for a given user name. The only invariant that the enrollment server needs to ensure is that no two users may have the same identity.

3.4.2. Bootstrap

The above steps are only done the first time a peer joins a new overlay or when the overlay parameters are close to their expiration time (as listed in the configuration document) and need to be refreshed. The next step is the bootstrap step which is done every time the peer boots.

Bootstrapping consists of looking at the list of cached nodes and bootstraps nodes and sending a RELOAD PING to them to see if they respond. Once a node responds, it can be used to join the overlay. After a node has joined, it keeps track of a small number of peers to which it could directly connect. These are saved as the cached nodes and used next time the peer boots. The point of the cached nodes is to reduce the load on the bootstrap nodes.

3.5. Security

3.5.1. Certificate-Based Security

The certificate-based security model revolves around the enrollment process allocating a unique name to the user and issuing a certificate [RFC3280] for a public/private key pair for the user. All peers in a particular DHT can verify these certificates. A given peer acts on behalf of a user, and that user is responsible for its operation.
The certificate serves two purposes:

- It entitles the user to store data at specific locations in the DHT. Each usage defines the specific rules for determining which certificates can access each Hashed-ID/kind-id pair. For instance, some usages might allow anyone to write at a given location, whereas others might restrict writes to a single certificate.

- It entitles the user to operate a peer that has a peer-id found in the certificate. When the peer is acting as a DTLS or TLS server, it can use this certificate so that a client connecting to it knows it is connected to the correct server.

When a user enrolls, or enrolls a device with no keying material, the user is given a certificate. This certificate contains information that identifies the user and the device they are using. If a user has more than one device, typically they would get one certificate for each device. This allows each device to act as a separate peer.

The contents of the certificate include:

- A public key provided by the user.
- Zero or more user names that the DHT is allowing this user to use. For example, "alice@example.org". Typically a certificate will have one name. In the SIP usage, this name corresponds to the AOR.
- Zero or more peer-ids. Typically there will be one peer-id. Each device will use a different peer-id, even if two devices belong to the same user. Peer-IDs should be chosen randomly by the enrollment server.
- A serial number that is unique to this certificate across all the certificates issued for this DHT.
- An expiration time for the certificate.

Note that because peer-IDs are chosen randomly, they will be randomly distributed with respect to the user name. This has the result that any given peer is highly unlikely to be responsible for storing data corresponding to its own user, which promotes high availability.

3.5.1.1. Storage Permissions

When a peer uses a STORE request to place data at a particular location X, it must sign with the private key that corresponds to a certificate that is suitable for storing at location X. Each data kind in a usage defines the exact rules for determining what certificate is appropriate.

The most natural rule is that a certificate with user name X "owns"
data located at Hash(X) (X is the Unhashed-ID and Hash(X) is the Hashed-ID) and only he can write there. This rule is used for all the kinds defined in this specification. Thus, only a user with a certificate for "alice@example.org" could write to that location in the DHT. However, other usages can define any rules they choose, including publicly writable values.

The digital signature over the data serves two purposes. First, it allows the peer responsible for storing the data to verify that this STORE is authorized. Second, it provides integrity for the data. The signature is saved along with the data value (or values) so that any reader can verify the integrity of the data. Of course, the responsible peer can "lose" the value but it cannot undetectably modify it.

3.5.1.2. Peer Permissions

The second purpose of a certificate is to allow the device to act as a peer with the specified peer-ID. When a peer wishes to connect to peer X, it forms a TLS/DTLS connection to the peer and then performs TLS mutual authentication and verifies that the presented certificate contains peer-ID X.

Note that because the formation of a connection between two nodes generally requires traversing other nodes in the DHT, as specified in Section 3.2.1, those nodes can interfere with connection initiation. However, if they attempt to impersonate the target peer they will be unable to complete the TLS mutual authentication: therefore such attacks can be detected.

3.5.1.3. Expiry and Renewal

At some point before the certificate expires, the user will need to get a new certificate from the enrollment server.

3.5.2. Shared-Key Security

RELOAD also defines a shared-key security model which can be used in closed networks where the peers are not mutually suspicious. In this model, the peers all share a single key which is used to authenticate the peer-to-peer DTLS connections via TLS-PSK/TLS-SRP. If shared-key security mode is in use, a shared-key capable cipher suite such as TLS-PSK or TLS-SRP MUST be used. This is useful for admission control, but is completely unsafe in any setting where peers are not mutually trusted, since it allows any peer to impersonate any other peer.
3.6. Migration

At some point in time, a given P2P Network may want to migrate from one underlying DHT algorithm to another or update to a later extension of the protocol. This can also be used for crypto agility issues. The migration approach is done by having peers initializing algorithm A. When the clients go to periodically renew their credentials, they find out that the P2P Network now requires them to use algorithm A but also to store all the data with algorithm B. At this point there are effectively two DHT rings in use, rings A and B. All data is written to both but queries only go to A. At some point when the clients periodically renew their credentials, they learn that the P2P Network has moved to storing to both A and B but that FETCH requests are done with P2P Network B and that any SEND should first be attempted on P2P Network B and if that fails, retried on P2P Network A. In the final stage when clients renew credentials, they find out that P2P Network A is no longer required and only P2P Network B is in use. Some types of usages and environments may be able to migrate very quickly and do all of these steps in under a week, depending on how quickly software that supports both A and B is deployed and how often credentials are renewed. On the other hand, some very ad-hoc environments involving software from many different providers may take years to migrate.

[[TODO: This needs more filling out]]

3.7. Usages Layer

By itself, the distributed storage layer just provides infrastructure on which applications are built. In order to do anything useful, a usage must be defined. Each Usage needs to specify several things:

- Register kind-id code points for any kinds that the Usage defines.
- Define the data structure for each of the kinds.
- Define access control rules for each kinds.
- Provide a size limit for each kinds.
- Define how the Unhashed-ID is formed that is hashed to form the Resource-ID where each kind is stored.
- Describe how values will be merged after a network partition. Unless otherwise specified, the default merging rule is to act as if all the values that need to be merged were stored and that the order they were stored in corresponds to the stored time values associated with (and carried in) their values. Because the stored time values are those associated with the peer which did the writing, clock skew is generally not an issue. If if two nodes are on different partitions, clocks, this can create merge conflicts. However because RELOAD deliberately segregates storage so that data from different users and peers is stored in different
locations, and a single peer will typically only be in a single network partition, this case will generally not arise.

The kinds defined by a usage may also be applied to other usages. However, a need for different parameters, such as different size limits, would imply the need to create a new kind.

3.7.1. SIP Usage

From the perspective of P2PSIP, the most important usage is the SIP Usage. The basic function of the SIP usage is to allow Alice to start with a SIP URI (e.g., "bob@dht.example.com") and end up with a connection which Bob’s SIP UA can use to pass SIP messages back and forth to Alice’s SIP UA.

This is done using three key operations that are provided by the SIP Usage. They are:

- Mapping SIP URIs that are not GRUUs to other SIP URIs or to the DHT peer responsible for the SIP UA.
- Mapping SIP GRUUs to the DHT peer responsible for the SIP UA.
- Forming a connection directly to a DHT peer that is used to send SIP messages to the SIP UA.

All SIP URIs for a given overlay MUST be constructed so that they terminate in the domain name of the overlay. For instance, if the overlay name is "example.com", then all AORs must be of the form {sip,sips}:username@example.com. Accordingly, to dereference a URI, a P2PSIP implementation MUST check to see if the domain matches an overlay which it is a member of. If so, it uses the following procedures. Otherwise, it MUST follow [RFC3263] procedures. Note that unless the P2PSIP overlay provides some kind of gateway to ordinary SIP (e.g., a publicly accessible SIP server) this is likely to be only partially successful, since, for instance, the callee may not be able to call back.

3.7.1.1. SIP Location

A peer acting as a SIP UA stores their registration information in the DHT by storing either another URI (for retargeting) or a destination lists to reach them at a Resource-ID in the DHT formed from the user’s SIP AOR. When another peer wishes to find a peer that is registered for a SIP URI, the lookup of the user’s name is done by taking the user’s SIP Address or Record (AOR) and using it as the Unhashed-ID that is hashed to get a Resource-ID. When the Unhashed-ID is dereferenced, the result is a set of values. Each value is either another SIP URI or a destination list. If the value is a SIP URI, the calling peer looks up that URI and continues the
process until it gets a destination list.

If the value is a destination list, then it is used to reach a peer that represents a SIP UA registered for that AOR. Typically this destination list will have just one entry but in the case of peers or clients that cannot be directly reached (for instance via a strict NAT or firewall), a destination list with more than one entry may need to be used.

The Unhashed-ID for this usage is a user’s SIP AOR, such as "sip:alice@example.com". This allows the set to store many values in a dictionary structure. The authorization policy is that STORE requests are only allowed if the user name in the signing certificate, when turned into a SIP URL and hashed, matches the Resource-ID. This policy ensures that only a user with the certificate with the user name "alice@example.com" can write to the Resource-ID that will be used to look up calls to "sip:alice@example.com".

[Open Issue: Should the Unhashed-ID be "sip:alice@example.com", "alice@example.com", or a string that includes the code point defined for the kind? The issue here is determining whether different usages that store data at a Unhashed-ID that is primarily formed from "alice@example.com" should hash to the same Resource-ID as the SIP Usage. For example, if a buddy list had a Unhashed-ID that was roughly the same, would we want the buddy list information to end up on the same peers that stored the SIP location data or on different peers?]

3.7.1.2. SIP GRUUs

GRUUs that refer to peers in the P2P network are constructed by simply forming a GRUU, where the value of gr URI parameter contains a base64 encoded version of the destination list that will reach the peer. Typically the destination list is just a single entry with the peer-id of peer.

3.7.1.3. SIP Connect

This usage allows two clients to form a new TLS or DTLS connection between them and then use this connection for sending SIP messages to one another. This does not store any information in the DHT, but it allows the CONNECT request to be used to set up a TLS or DTLS connection between two peers and then use that connection to send SIP messages back and forth.

The CONNECT request will ensure that the connection is formed to a peer that has a certificate which includes the user that the
connection is being formed to.

3.7.1.4. SIP Tunnel

This TUNNEL request allows two peers to exchange SIP messages across
the overlay using the TUNNEL method without first setting up a direct
connection using CONNECT. This allows a SIP message to be sent
immediately, without the delay associated with CONNECT and for a
simple SIP exchange, it may result in fewer messages being sent.

3.7.2. Certificate Store Usage

This usage allows each user to store their certificate in the DHT so
that it can be retrieved to be checked by various peers and
applications. Peers acting on behalf of a particular user store that
user’s certificate in the DHT, and any peer that needs the
certificate can do a FETCH to retrieve the certificate. Typically it
is retrieved to check a signature on a request or the signature on a
chunk of data that the DHT has received.

3.7.3. TURN Usage

This usage defines a new kind for finding STUN-Relay servers. Any
peer that supports this usage saves a pointer to the IP address and
port of the TURN server in the DHT. When a peer wishes to discover a
TURN server, it picks a random Resource-ID and performs a FIND at
that Resource-ID for the appropriate type for the service. If
nothing is found, this can be repeated until an appropriate set of
servers are found.

3.7.4. Diagnostic Usage

This usage defines several new kinds that be queried to find
information about the peer that may be useful for monitoring and
diagnostics. This includes information such as software version,
neighbor information, and performance statistics.

3.7.5. HIP Tunnel Usage

This usage allows two peers running HIP to tunnel HIP messages across
the overlay. This allows the HIP peers to use the overlay as a
rendezvous system to set up a direct path using HIP NAT traversal
mechanisms.
4. Base Protocol

RELOAD is a message-oriented request/response protocol. The messages are encoded using binary fields. All integers are represented in network byte order. The general philosophy behind the design was to use Type, Length, Value fields to allow for extensibility. However, for the parts of a structure that were required in all messages, just define theses in a fixed position as adding a type and length for them is unnecessary and would simply increase bandwidth and introduces new potentials for interoperability issues.

Each message has three parts:

Forwarding Header: Each message has a generic header which is used to forward the message between peers and to its final destination. This header is the only information that an intermediate peer (i.e., one that is not the target of a message) needs to examine.

Message Contents: The message being delivered between the peers. From the perspective of the forwarding layer, the contents is opaque, however, it is interpreted by the higher layers.

Signature: A digital signature over the message contents and parts of the header of the message. Note that this signature can be computed without parsing the message contents.

The following sections describe the format of each part of the message.

4.1. Forwarding Header

The layout of the forwarding header is shown below
The first four bytes identify this message as a RELOAD message. The message is easy to demultiplex from STUN messages by looking at the first bit.

The Overlay field is the 32 bit checksum/hash of the overlay being used. The variable length string representing the overlay name is hashed with SHA-1 and the low order 32 bits are used. The purpose of this field is to allow nodes to participate in multiple overlays and to detect accidental misconfiguration.
TTL (time-to-live) is an 8 bit field indicating the number of iterations, or hops, a message can experience before it is discarded. The TTL value MUST be decremented by one at every hop along the route the message traverses. If the TTL is 0, the message MUST NOT be propagated further and MUST be discarded. The initial value of the TTL should be TBD.

FRAG is a 1 bit field used to specify if this message is a fragment.

- NOT-FRAGMENT : 0x0
- FRAGMENT : 0x1

LFRG is a 1 bit field used to specify whether this is the last fragment in a complete message.

- NOT-LAST-FRAGMENT : 0x0
- LAST-FRAGMENT : 0x1

[[Open Issue: How should the fragment offset and total length be encoded in the header? Right now we have 14 bits reserved with the intention that they be used for fragmenting, though additional bytes in the header might be needed for fragmentation.]]

Version is a 7 bit field that indicates the version of the RELOAD protocol being used.

- Version1.0 : 0x1

The message Length is the count in bytes of the size of the message, including the header.

The Transaction ID is a unique 64 bit number that identifies this transaction and also serves as a salt to randomize the request and the response. Responses use the same Transaction ID as the request they correspond to. Transaction IDs are also used for fragment reassembly.

The Destination List Length and the Via List Length contain the lengths of the route and via lists respectively, in the number of objects.

[[Open Issue: How should we handle peer-id lengths? This basically assumes they’re fixed length per DHT algorithm (but not fixed-length for RELOAD) so that you can unambiguously parse things. Should we have a length byte?]]

The flags word contains control flags. There is one currently defined flag.
ROUTE-LOG : 0x1

The ROUTE-LOG flag indicates that the route log should be included (see Section 4.1.4)

The Destination List contains a sequence of destinations which the message should pass through. The destination list is constructed by the message originator. The first element in the destination list is where the message goes next. The list shrinks as the message traverses each listed peer. Destinations are defined at the end of this section.

The Via List contains the sequence of destinations through which the message has passed. The via list starts out empty and grows as the message traverses each peer.

If a message was being sent thought the sequences of peers A,B,C,D, the message from A to B would have a empty via list and a route of list of B,C,D. The message from B to C would have a via list of A then route of C,D and so on. This means that when the route list is followed exactly, all that is needed to update these lists is to change their lengths. This avoids the need to change or move any of the other list entries. In other cases, some entries may need to be copied or moved.

The destination list and via lists a list of objects of type destination_object:

STRUCTURE: destination_object

0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|                                                               |
|                  Info                                         |
|                                                               |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

STRUCTURE: peer

0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       0x00     |                                               |
004 +-+-+-+-+-+-+-+-+                                               +
|                  Peer                                          |
008 + +-+-+-+-+-+-+-+-+ +

A destination_object can have one of three types (this is extensible):

peer: A peer-id. All peer-ids are of fixed length for a given overlay and therefore peer-ids have no length.

compressed: A compressed list of peer-ids and/or resources. This value is variable length but because it was compressed by one of the peers, it is only meaningful to that peer and cannot be decoded by other peers.

resource: The resource id of the resource which is desired. This type MUST only appear in the final location of a destination list and MUST NOT appear in a via list. It is meaningless to try to route through a resource.
4.1.1. Changes to Forwarding Header

The RELOAD-01 forwarding header was completely fixed, whereas this header includes lists that change en-route. However, this type of operation is easily accomplished in both software and hardware, therefore we still view it as a low-overhead header. The changes include the following.

- Rearranged fields to have a cleaner separation between payload and header.
- Removed DHT, Hash, and Security parameters. These are now in the overlay bootstrap system rather than per-message.
- Source and destination IDs are now destination lists to accommodate source routing and recursion without state on intermediate peers.
- Added route log to header to allow payload/header separation.

4.1.2. Message Routing

4.1.2.1. Request Origination

In order to send a message to a given peer-id or resource-id, a peer must construct an appropriate destination list. The most common such destination list is a single entry containing the peer/resource-id. This simply uses the normal DHT routing mechanisms to forward the message to that destination.

Messages can also be source routed. In order to construct a source route, the originator provides a destination list containing a sequence of resource-ids. The semantics of this destination list are that the message is to traverse in order (potentially with intermediate hops) each entry on the destination list. As each peer is traversed, that entry is removed from the destination list. This makes it possible to address a peer which is potentially behind a NAT or a firewall in such a way that it cannot be connected to directly under any circumstances.

[[TODO: Salman has suggested the originator doing parallel requests/responses. This is an open issue.]]

4.1.2.2. Response Origination

When a peer sends a response to a request, it SHOULD construct the destination list by reversing the order of the entries on the via list. This has the result that the response traverses (at least) the same peers as the request traversed, except in reverse order (symmetric routing). For asymmetric routing, the peer MAY simply use the first entry on the via list.
4.1.2.3. Message Receipt and Forwarding

When a peer receives a message, it first examines the overlay, version, and other header fields to determine whether the message is one it can process. If any of these are incorrect (e.g., the message is for an overlay in which the peer does not participate) it is an error. The peer SHOULD generate an appropriate error but MAY simply drop the message.

Once the peer has determined that the message is correctly formatted, it examines the first entry on the destination list. There are three possible cases here:

- The first entry on the destination list is a private id which is being used for destination list compression.
- The first entry on the destination list is an id for which the peer is responsible.
- The first entry on the destination list is for which another peer is responsible.

These cases are handled separately.

4.1.2.3.1. Private ID

If the first entry on the destination list is a private id, the peer replaces that entry with the store local value that it indexes and then re-examines the destination list to determine which case now applies.

4.1.2.3.2. Responsible ID

If the first entry on the destination list is a Hashed-ID for which the peer is responsible, the peer strips the entry off the route list. If there are remaining entries on the destination list, the peer then re-examines the destination list to determine which case now applies. If the destination list is now empty, then the message was destined for this peer and it MUST pass it to the next layer up.

4.1.2.3.3. Other Hashed-ID

If neither of the other two cases applies, then the peer MUST forward the message towards the first entry on the destination list. This means that it MUST select one of the peers in its route table which is closer to the first entry than to itself and send the message to that peer. If the first entry on the destination list is in the peer's connection table, then it SHOULD forward the message to that peer directly.
When forwarding a message, the peer MUST:

- Decrement the TTL value
- Update the via list.

The natural way to update the via list is simply to add the peer-id of the peer from which the message was received to the end of the list. However, peers may use any algorithm of their choice provided that if the peer received a destination list constructed by reversing the via list it would be able to route the outgoing message correctly, enabling symmetric routing.

For instance, if node D receives a message from node C with via list (A, B), the simple approach is simply to forward to the next node (E) with via list (A, B, C). Now, if E wants to respond to the message, it reverses the via list to produce the destination list, resulting in (D, C, B, A). When D forwards the response to C, the destination list will contain (B, A). However, node D could also list compression and send E the via list (X). E would then use the destination list (D, X). When D processes this destination list, it MUST detect that X is a compressed entry, recover the via list (A, B, C), and reverse that to produce the correct destination list (C, B, A) before sending it to C.

Note that if a peer is using list compression and then exits the overlay, the message cannot be forwarded and will be dropped. The ordinary timeout and retransmission networks provide stability over this type of failure.

4.1.3. Fragmentation and Reassembly

In order to allow transport over datagram protocols, RELOAD messages may be fragmented. If a message is too large for a peer to transmit to the next peer it MUST fragment the message. Note that this implies that intermediate peers may re-fragment messages if the incoming and outgoing paths have different maximum datagram sizes. Intermediate peers SHOULD NOT reassemble fragments.

Upon receipt of a fragmented message by the intended peer, the peer holds the fragments in a holding buffer until the entire message has been received. The message is then reassembled into a single unfragmented message and processed. In order to prevent denial of service attacks, receivers SHOULD timeout incomplete fragments. [[TODO: Describe algorithm]]
4.1.4. Route Logging

The route logging feature provides diagnostic information about the path taken by the request so far and in this manner it is similar in function to SIP’s [RFC3261] Via header field. If the ROUTE-LOG flag is set in the Flags word, at each hop peers MUST append a route log entry to the route log element in the header. The order of the route log entry elements in the message is determined by the order of the peers were traversed along the path. The first route log entry corresponds to the peer at the first hop along the path, and each subsequent entry corresponds to the peer at the next hop along the path. If the ROUTE-LOG flag is set in a request, the route log MUST be copied into the response and the ROUTE-LOG flag set so that the originator receives the ROUTE-LOG data.

If the responder wishes to have a route log in the reverse direction, it MAY set the ROUTE-LOG flag in its response as well. Note, however, that this means that the response will grow on the return path, which may potentially mean that it gets dropped due to becoming too large for some intermediate hop. Thus, this option must be used with care.

STRUCTURE: route_log

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Entries Len          |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Entries                            |
/                                                               /
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The route log is simply a variable length list of route log entries. The first two bytes are the length, followed by a sequence of route log entries, each of which may be individually parsed.

STRUCTURE: route_log_entry

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Version Len          |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Version                            |
/                                                               /
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

| Transport | | +---------------------------+ |
|           | | Id                        |
|           | | +                         |
|           | | +                         |
|           | | +                         |
|           | +---------------------------+ |
|           | Uptime                    |
|           | +---------------------------+ |
|           | Certificate Len           |
|           | +---------------------------+ |
|           | Certificate                |
|           | /                          |
|           | /                          |
|           | /                          |
|           | Address                   |
|           | /                          |
|           | /                          |

STRUCTURE: ip4_address_type

```
0  1  2  3  4  5  6  7  8  9 0 1 2 3 4 5 6 7 8 9 0 1 2
+----------------------------------+
| 0x01 | Addr         |
|      | Port         |
```

STRUCTURE: ip6_address_type

```
0  1  2  3  4  5  6  7  8  9 0 1 2 3 4 5 6 7 8 9 0 1 2
+----------------------------------+
| 0x02 | Addr         |
|      | Port         |
```

Each route log entry consists of the following values:

Version - A textual representation of the software version
Transport - The transport type, 1 for TLS, 2 for DTLS
Id - The peer-id of the peer.
Uptime - The uptime of the peer in seconds.
Certificate - The peer’s certificate. Note that this may be omitted by setting the length to zero.
Address - The address and port of the peer. This can be either an IPv4 or IPv6 address.

4.2. Message Contents Format

Although from the perspective of the forwarding layer the content is opaque, all RELOAD messages share a common content structure consisting of two parts:

Common Header: A common header containing the request method/response code, and a transaction ID.
Payload: The actual body of the request/response. These are dependent on whether this is a request or response and the type of request being carried.

4.2.1. Common Header

The layout of the common header is shown below:

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                 |                             |
|        Message Code             |           Reserved          |
|                                 |                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Message Code is a 16 bit field that indicates which message this is. This field is broken up as follows:

0 Reserved
1..0x7fff Requests and responses. These code points are always paired, with requests being odd and the corresponding response being the request code plus 1. Thus, PING_Q (the PING request) has value 1 and PING_A (the PING response) has value 2
4.2.2. Payload

Payload is a simple string of uninterpreted bytes preceded by a length field indicating the length of the data, not including the length field. The bytes themselves are dependent on the code value.

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Length |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

// Length bytes of data //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

4.2.3. Signature

The Signature element is used to attach signatures to messages and or stored data elements. All signatures are formatted using this element. However, the input structure to the signature computation varies depending on the data element being signed.

```
STRUCTURE: signature
 0 1 2 3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Algorithm | Signature Value Len |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Signature Value |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/ |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Identity |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The signature construct is just a container for the signature. It contains the following values:
Algorithm - The signature algorithm in use. This may have the values RSA-SHA1 (0x01) or RSA-SHA-256 (0x02).
Value - The signature value itself. This is just the string of bytes emitted by the signature algorithm.
Identity - The identity or certificate used to form the signature.

[[TODO: Should we convert all of this to CMS?]]

A number of possible identity formats are permitted, as shown below. The peer may indicate any of:

- Peer-id
- User name
- The certificate itself.

The first byte of the identity field is a type indicating the type of identity in use.

STRUCTURE: signer_object

```
+--------+-+-+--------+-+-+        +--------+-+-+--------+-+-+
|        |   |        |   |        |   |        |   |        |   |
| Signer |   |        |   |        |   |        |   |        |   |
```

STRUCTURE: signer_identity_peer

```
+--------+-+-+        +--------+-+-+
| 0x01    |          | 004    |        +--------+-+-+
| Signer Identity Peer |        |        |        |        +--------+-+-+
```

STRUCTURE: signer_identity_name

```
+--------+-+-+        +--------+-+-+
|        |   |        |   |        |   |        |   |        |   |
|        |   |        |   |        |   |        |   |        |   |
```

Jennings, et al. Expires August 27, 2008 [Page 40]
For signatures over messages the input to the signature function is:

**STRUCTURE: message_signature_input**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-----------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>004 +------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>008 +------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>012 +------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Overlay</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Xid</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Signer Identity</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Message Contents</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**STRUCTURE: signer_identity_certificate**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-----------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x02</td>
<td>Signer Identity Name Len</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-----------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-----------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x03</td>
<td>Signer Identity Certificate Len</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-----------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-----------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signer Identity Name</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-----------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-----------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signer Identity Certificate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-----------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message Contents</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-----------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The contents of this structure are as follows:

Overlay - The overlay identifier from the message.
Xid - The transaction id from the message.
Signer Identity - The identity of the signer (from the signature structure.)
Message Contents - The contents section of the message.

[[TODO: Check the inputs to this carefully.]]

The input to signatures over data values is different, and is described in Section 7.2.1.3.

4.3. Response Codes and Response Errors

A peer processing a request returns its status in the Message Code field of the common header. If the request was a success, then the message code is the response code that matches the request (i.e., the next code up). The response payload is then as defined in the request/response descriptions.

If the request failed, then the message code is set to 0xffff (error) and the payload MUST be an error_response PDU, as shown below.

For any code other than 200, the payload should be as defined below:

STRUCTURE: error_response

0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Error Code          |       Reason Phrase Len       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

The contents of this payload are:
Error Code - A numeric error code indicating the error that occurred
Reason Phrase - A free form text string indicating the reason for the response. The reason phrase SHOULD BE as indicated in the error code list (e.g., "Moved Temporarily).
Error Info - Payload specific error information. This MUST be empty except as specified below.

The following error code values are defined. [[TODO: These are currently semi-aligned with SIP codes. that’s probably bad and we need to fix.]]

302 (Moved Temporarily): The requesting peer SHOULD retry the request at the new address specified in the 302 response message.
401 (Unauthorized): The requesting peer needs to sign and provide a certificate. [[TODO: The semantics here don’t seem quite right.]]
403 (Forbidden): The requesting peer does not have permission to make this request.
404 (Not Found): The resource or peer cannot be found or does not exist.
408 (Request Timeout): A response to the request has not been received in a suitable amount of time. The requesting peer MAY resend the request at a later time.
412 (Precondition Failed): A request can’t be completed because some precondition was incorrect. For instance, the wrong generation counter was provided
498 (Incompatible with Overlay) A peer receiving the request is using a different overlay, DHT algorithm, or hash algorithm. [[Open Issue: What is the best error number and reason phrase to use?]]
499 (UnWilling To Proxy) A peer receiving the request is unwilling to support the Routing mechanism specified in the Routing field of the message header. [[Open Issue: What is the best error number and reason phrase to use?]]

5. End-to-End Timeout and Retransmission

Timeout and retransmission are handled on an end-to-end basis as well as the transports providing a hop by hop reliability mechanism. For end-to-end reliability, the requesting node retransmits a requests every 3 seconds until it receives a response or after it has send the request 5 times. Retransmissions MUST use the same transaction ID.
6. Transports

Currently multiple transport protocols are specified and more may be defined in the future. Implementation MUST implement TLS and DTLS. A given overlay can choose which protocols it uses.

6.1. TLS

TLS runs on top of TCP which offers the best performance from a data transfer point of view and does not require as frequent keep alive messages.

6.2. DTLS

DTLS runs on top of UDP which offers the highest probability of direct connectivity in the face of the current generation of consumer NATs.

6.2.1. Reliability for Unreliable Transports

When RELOAD is carried over DTLS or another unreliable transport, it needs to be used with a reliability and flow control mechanism, which is provided on a hop-by-hop basis, matching the semantics if TCP were used. The basic principle is that each message, regardless of if it carries a request or responses, will get an ACK and be reliably retransmitted. The receiver’s job is very simple, limited to just sending ACKs. All the complexity is at the sender side. This allows the sending implementation to trade off performance versus implementation complexity without affecting the wire protocol.

6.2.1.1. Message Format

Each message being sent is prepended with a header that indicates the 24 bit sequence number.

```
STRUCTURE: header

  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
  ++-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |      0x01     |                      Seq                      |
  ++-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Seq  The sequence number of the message.
```

Each DTLS session has it own sequence number. Initially the value is zero and it increments by exactly one for each message sent over that DTLS session.
6.2.1.2. Acknowledgement Format

| 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 |  |
| 004 | +-----------------------------------------------------+ |
| | Ack Seq |
| 008 | +-----------------------------------------------------+ |
| | Received |

When the receiver receive a message, it SHOULD immediately send an ACK message. The receiver MUST keep track of the 32 most recent sequence numbers received on this DTLS flow. The contents of this packet are:

- **Ack Seq** - The sequence number of the message being acknowledged.
- **Received** - A bitmask indicating whether or not each of the previous 32 packets has been received. The high order bit represents the first packet in the sequence space.

The received field bits in the ACK provide a very high degree of redundancy for the sender to figure out which packets the receiver received and can then estimate packet loss rates. If the sender also keeps track of the time at which recent sequence numbers were sent, the RTT can be estimated.

6.2.1.3. Retransmission and Flow Control

Because the receiver’s role is limited to providing packet acknowledgements, a wide variety of congestion control algorithms can be implemented on the sender side while using the same basic wire protocol. It is RECOMMENDED that senders implement use TFRC-SP [RFC4828] and use the received bitmask to allow the sender to compute packet loss event rates. Senders MUST implement a retransmission and congestion control scheme no more aggressive than TFRC-SP.

6.3. HIP

RELOAD MAY also be used with a HIP transport using the architecture for HIP BONE described in [I-D.camarillo-hip-bone]. From the perspective of the P2P layer, HIP looks very much like normal IP. Either TLS (over TCP) or DTLS (over UDP) is run over top of the HIP. Thus the reliability and congestion control schemes are the same for DTLS section. If an overlay is configured such that HIP is the only transport that it will use, then it may make sense to configure the p2p layer to only offer the ORCHID when gather candidate addresses.
for ICE. This will effectively disable ICE at the p2p layer.

For overlays that use HIP, the enrollment server MUST provide each peer with a unique ORCHID and use that ORCHID to generate the peer-id for the peer (see Section 10.3). Later when the HIP layer wishes to tunnel a message (such as an I1 message) through the overlay, the HIP layer can use the ORCHID to generate the peer-id, and then use the TUNNEL message with the HIP to route the message to that the peer that owns that ORCHID.

7. Method Definitions

In this section, we define the initial set of methods supported by RELOAD. New methods are defined by adding new method codes. Each method defines the contents of the payload element (see Section 4.2.2).

PDUs are named using the following convention. For method type FOO, the request PDU is named FOO_Q and the response PDU is named FOO_A (as are the method codes). When discussing the PDU itself, we use these terms. Throughout the rest of the document we refer to the FOO method or the FOO request/response for easier readability.

7.1. Connection Management

7.1.1. PING

PING is used to test connectivity along a path. A ping can be addressed to a specific peer-id or to the broadcast peer-id (all 1s). In either case, the target peer-ids respond with a simple response containing some status information.

7.1.1.1. Request Definition

The PING_Q message contains a list (potentially empty) of the pieces of status information that the requester would like the responder to provide.

STRUCTURE: ping_q

```
+-------------------------------------------+  +-------------------------------------------+
| Ping Info Len |                                               |
+-------------------------------------------+  +-------------------------------------------+
|                           Ping Info                           |
```

The two currently defined types are:

RESPONSIBLE-SET : 0x01
NUM-RESOURCES : 0x02

RESPONSIBLE-SET indicates that the peer should Respond with the
fraction of the overlay for which the responding peer is responsible
(in parts per billion).

NUM-RESOURCES indicates that the peer should Respond with the number
of resources currently being stored by the peer.

7.1.1.2. Response Definition

A successful PING_A response contains the information elements
requested by the peer.

STRUCTURE: ping_a

A PING_A message contains the following elements:

Response ID - A randomly generated 64-bit response ID. This is used
to distinguish PING responses in cases where the PING request is
multicast.
Infos - A sequence of ping info data structures, as shown below.
The ping info data elements are simple typed elements, with a type identifier as the leading 16 bits and then arbitrary (type-specific) text following. In the case of the two defined types, the responses are 32-bit integers.

The responding peer SHOULD include any values that the requesting peer requested and that it recognizes. They SHOULD be returned in the requested order.

7.1.2. CONNECT

A node sends a CONNECT request when it wishes to establish a direct TCP or UDP connection to another node for the purposes of sending RELOAD messages or application layer protocol messages, such as SIP. Detailed procedures for the CONNECT and its response are described in Section 8.
Note: A CONNECT does not result in updating the routing table of either node. That function is performed by UPDATEs. If node A has CONNECTed to node B, it MAY route messages which are directly addressed to B through that channel but MUST NOT route messages through B to other peers via that channel.

7.1.2.1. Request Definition

A CONNECT_Q message contains the requesting peer’s ICE connection parameters formatted into a binary structure.
STRUCTURE: connect_data

```
| Ufrag Len | +----------------------------------+
| Ufrag     | /                                  |
| Password Len | +----------------------------------+
| Password  | /                                  |
| Application | Fingerprint Len |
| Fingerprint | /                                  |
| Role Len | +----------------------------------+
| Role     | /                                  |
| Candidate List Len | +----------------------------------+
| Candidate List | /                                  |
```

STRUCTURE: candidate

```
| Candidate String Len | +----------------------------------+
| Candidate String    | /                                  |
```
The values contained in connect-request are:

Ufrag - The username fragment (from ICE)
Password - The ICE password.
Application - A 16-bit port number. This port number represents the IANA registered port of the protocol that is going to be sent on this connection. For SIP, this is 5060 or 5061, and for RELOAD is TBD. By using the IANA registered port, we avoid the need for an additional registry and allow RELOAD to be used to set up connections for any existing or future application protocol.
Fingerprint - One fingerprint attribute (from RFC 4572).
Role - An active/passive/actpass attribute from RFC 4145.
Candidate - One or more ICE candidate values. Each candidate has an IP address, IP address family, port, transport protocol, priority, foundation, component ID, STUN type and related address. The candidate_list is a list of string candidate values.

These values should be generated using the procedures of Section 8.

7.1.2.2. Response Definition

If a peer receives a CONNECT request, it SHOULD follow the procedures of Section 8 to process the request and generate its own response (a CONNECT_A) containing a connect_data object. It should then begin ICE checks. When a peer receives a CONNECT response, it SHOULD parse the response and begin its own ICE checks.

7.1.3. TUNNEL

A node sends a TUNNEL request when it wishes to exchange application-layer protocol messages without the expense of establishing a direct connection via CONNECT or when ICE is unable to establish a direct connection via CONNECT and a TURN relay is not available. The application-level protocols that are routed via the TUNNEL request are defined by that application’s usage.

Note: The decision of whether to route application-level traffic across the overlay or to open a direct connection requires careful consideration of the overhead involved in each transaction. Establishing a direct connection requires greater initial setup costs, but after setup, communication is faster and imposes no overhead on the overlay. For example, for the SIP usage, an INVITE request to establish a voice call might be routed over the overlay, a SUBSCRIBE with regular updates would be better used with a CONNECT, and media would both impose too great a load on the overlay and likely receive unacceptable performance. However, there may be a tradeoff between locating TURN servers and relying on TUNNEL for packet routing.
When a usage requires the TUNNEL method, it must specify the specific application protocol(s) that will be TUNNELed and for each protocol, specify:

- An application attribute that indicates the protocol being tunneled. This the IANA-registered port of the application protocol.
- The conditions under which the application will be TUNNELed over the overlay rather than using a direct CONNECT.
- A mechanism for moving future application-level communication from TUNNELing on the overlay to a direct CONNECTion, or an explanation why this is unnecessary.
- A means of associating messages together as required for dialog-oriented or request/response-oriented protocols.
- How the TUNNELed message (and associated responses) will be delivered to the correct application. This is particularly important if there might be multiple instances of the application on or behind a single peer.

7.1.3.1. Request Definition

The TUNNEL_Q message contains the application PDU that the requesting peer wishes to transmit, along with some control information identifying the handling of the PDU.

```
STRUCTURE: tunnel_q
   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
   +---------------------------------------------------------------+
   |         Application          |         Dialog Id Len         |
   +---------------------------------------------------------------+
   |                                                               |
   /                                                               |
   |                                                               |
   /                                                               |
   |                                                               |
   +---------------------------------------------------------------+
   |      Application Pdu Len      |                               |
   +---------------------------------------------------------------+
   |                        Application Pdu                        |
   /                                                               |
   |                                                               |
```

The values contained in the TUNNEL_Q are:
Application - A 16-bit port number. This port number represents the IANA registered port of the protocol that is going to be sent on this connection. For SIP, this is 5060 or 5061, and for RELOAD is TBD. By using the IANA registered port, we avoid the need for an additional registry and allow RELOAD to be used to set up connections for any existing or future application protocol.

Dialog ID - An arbitrary string providing an application-defined way of associating related TUNNELed messages. This attribute may also encode sequence information as required by the application protocol.

Application PDU - An application PDU in the format specified by the application.

7.1.3.2. Response Definition

A TUNNEL_A message serves as confirmation that the message was received by the destination peer. It implies nothing about the processing of the application. If the application protocol specifies an acknowledgement or confirmation, that must be sent with a separate TUNNEL request.

7.2. Data Storage and Retrieval

The STORE, FETCH, and REMOVE methods are used to manipulate information in the DHT. They form an instantiation of the abstract GET and PUT operations described in [I-D.ietf-p2psip-concepts].

7.2.1. STORE

The STORE method is used to store data in the overlay. As described in Section 3.1.4, each location may contain data of multiple kinds. Each kind-id is a code point assigned to a specific application usage by IANA. As part of the Usage definition, protocol designers may define constraints, such as limits on size, on the values which may be stored. For many kinds, the set may be restricted to a single item; some sets may be allowed to contain multiple identical items while others may only have unique items. The protocol currently defines the following data models:

- single value
- array
- dictionary

Each kind MUST specify the appropriate data model for that kind. The format of the STORE request depends on the data model.
7.2.1.1. Request Definition

A STORE_Q message is a sequence of kind-data pairs, each of which represents a sequence of stored values for a given kind. The same kind-id MUST NOT be used twice in a given store request. Each value is then processed in turn. These operations MUST be atomic. If any operation fails, the state MUST be rolled back to before the request was received.

**STRUCTURE: store_q**

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
| +----------------+-------------------+-------------------+
| | Resource Len  | Resource           |
| | +--------------+-+-------------------+-------------------+
| | Store Kind Data Len | Store Data |
| | +----------------+-------------------+-------------------+
| |                |                     |
```

A single STORE request stores data of a number of kinds to a single resource location. The contents of the request are:

- **Resource** - The resource to store at.
- **Store Kind Data** - A series of elements, one for each kind of data to be stored.
Each store kind data element represents the data to be stored for a single kind-id. The contents of the element are:

Kind - The kind-id. Implementations SHOULD reject requests corresponding to unknown kinds unless specifically configured otherwise.

Data Model - The data model of the data.

Generation - The expected current state of the generation counter (approximately the number of times this object has been written, see below for details).

Values - The value or values to be stored. This may contain one or more stored_data values depending on the data model associated with each kind.
Each stored_data element represents a single stored data value. These elements are individually signed. The contents of the element are as follows:

Length – The length of the stored data element.
Storage Time – The time when the data was stored in absolute time, represented in seconds since the Unix epoch. Any attempt to store a data value with a storage time before that of a value known to the receiving peer MUST generate a 412 error. This prevents rollback attacks. Note that this does not require synchronized clocks: the receiving peer uses the storage time in the previous store, not its own clock.
Lifetime – The validity period for the data, in seconds, starting from the time of store.
Signature – A signature over the data value. Section 7.2.1.3 describes the signature computation. The element is formatted as described in Section 4.2.3.
Data Value – The data value itself, as described below.
The responsible peer MUST perform the following checks:

- The kind-id is known.
- The signature over the message is valid or (depending on overlay policy) no signature is required.
The signatures over each individual data element (if any) are valid.
Each element is signed by a credential which is authorized to write this kind at this resource-id
If the generation-counter is non-zero, it must equal the current value of the generation-counter for this kind. This feature allows the generation counter to be used in a way similar to the HTTP Etag feature.
The storage time values are greater than that of any value which would be replaced by this STORE. [[OPEN ISSUE: do peers need to save the storage time of REMOVEs to prevent reinsertion?]]

If all these checks succeed, the peer MUST attempt to store the data values. If the store succeeds and the data is changed, then the peer must increase the generation counter by at least one. If there are multiple stored values in a single store_kind_data, it is permissible for the peer to increase the generation counter by only 1 for the entire kind-id, or by 1 or more than one for each value.

We now discuss each data model:

7.2.1.1.1. Single Value
There may be only one single-value element for each resource-id, kind-id pair. A store of a new single-value element MUST overwrite the current value.

7.2.1.1.2. Array
A store of an array entry replaces (or inserts) the given value at the location specified by the index. Arrays are zero-based. Note that arrays can be sparse. Thus, a store of "X" at index 2 in an empty array produces an array with the values [ NA, NA, "X"]. Future attempts to fetch elements at index 0 or 1 will return empty strings. If the index value is -1, then the value is placed at the end of the array.

7.2.1.1.3. Dictionary
A stored dictionary entry has a dictionary-key used as a lookup key and a dictionary-value containing the data. There may be only one value for any given dictionary-key and therefore a write to a dictionary-key overwrites whatever is there.

7.2.1.2. Response Definition
In response to a successful STORE request the peer MUST return a STORE_A message containing a series of store_kind_response elements...
containing the current value of the generation counter for each
kind-id, as well as a list of the peers where the data was
replicated.

**STRUCTURE:** store_kind_response

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>004</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>008</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>012</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The contents of each element are:

- **Kind** – The kind-id being represented.
- **Generation** – The current value of the generation counter for that
  kind-id.
- **Replicas** – The list of other peers at which the data was/will-be
  replicated. In DHTs and applications where the responsible peer
  is intended to store redundant copies, this allows the storing
  peer to independently verify that the replicas were in fact
  stored.

The response itself is just the store_kind_response values packed
end-to-end.

If the request was rejected because of an invalid generation counter,
then the store-response MUST also be returned, but with a response
code of 412. Otherwise, the response MAY contain a response-error-
reason production or MAY be empty. [[TODO: The generation counter
may need more thinking for uniqueness.]]

### 7.2.1.3. Data Signature Computation

Each stored-data element is individually signed. However, the
signature also must be self-contained and cover the kind-id and
resource-id even though they are not present in the stored value.
The data signed is defined as:
STRUCTURE: stored_data_to_be_signed

0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Resource Len |                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Resource                           |
/+----------------------------------------------------------+
|                            Kind                            |
/+----------------------------------------------------------+
|                          Stored Data                        |
/+----------------------------------------------------------+
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

The contents of this value are as follows:

Resource - The resource ID where this data is stored.
Type - The kind-id for this data.
Stored Data - The contents of the stored data value, as described in
the stored_data PDU of Section 7.2.1.1

[[TODO: Should we include the identity?]]

Once the signature has been computed, the signature is represented
using a signature element, as described in Section 4.2.3.

7.2.2.  FETCH

The FETCH request retrieves one or more data elements stored at a
given resource-id.

7.2.2.1.  Request Definition

A FETCH_Q message consists of a single fetch_request element followed
by a series of fetch_kind_data elements.
The contents of the request are as follows:

**Resource** - The resource ID to fetch from.

**Fetch Data** - A sequence of data specifiers, one for each desired kind-id.

Each fetch_kind_data element is specified as follows.

**STRUCTURE: fetch_kind_data**

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Kind |                                                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Data Model |                                                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Generation |                                      +          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Reference |                                      +          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
Kind – The kind-id of the data being fetched. Implementations SHOULD reject requests corresponding to unknown kinds unless specifically configured otherwise.

Data Model – The data model of the data.

Generation – The last generation counter that the requesting peer saw. This is used to avoid unnecessary fetches.

Reference – A reference to the data value being requested within the data model specified for the kind, as specified below.

STRUCTURE: fetch_array_reference

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
|                             First              |
|                                      +       |
|                                      |       |
|                                      |       |
|                                      |       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
```

STRUCTURE: fetch_dictionary_reference

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
|      Dictionary Keys Len      |                               |
|                        +-------------------------------+
|                        |      Dictionary Keys                   |
|                        |                        /             |
|                        |                        |             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
```

STRUCTURE: dictionary_key

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
|         Key Value Len         |                               |
|                        +-------------------------------+
|                        |      Key Value                     |
|                        |                        /             |
|                        |                        |             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
```

As with STORE, the FETCH_Q contains a list of kind-ids and associated references. The reference encoding depends on the kind of value being stored.

- If the data is of data model single value, the reference is empty.
o If the data is of data model array, the reference contains two integers. The first integer is the beginning of the range and the second is the end of the range. 0 is used to indicate the first element and -1 is used to indicate the final element. The beginning of the range MUST be earlier in the array then the end.

o If the data is of data model dictionary then the reference contains a list of the dictionary keys being requested. If no keys are specified, than this is a wildcard fetch and all key-value pairs are returned. [[TODO: We really need a way to return only the keys. We’ll need to modify this.]]

The generation-counter is used to indicate the requester’s expected state of the storing peer. If the generation-counter in the request matches the stored counter, then the storing peer returns a cache hit indicator rather than the stored data.

Note that because the certificate for a user is typically stored at the same location as any data stored for that user, a requesting peer which does not already have the user’s certificate should request the certificate in the FETCH as an optimization.

7.2.2.2. Response Definition

The response to a successful FETCH request is a FETCH_A message containing the data requested by the requester.

```plaintext
STRUCTURE: fetch_a

0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              Kind                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Generation                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Stored Data Len        |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+                               |
|                          Stored Data                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

There MUST be one fetch_kind_data element for each kind-id in the request. If the generation-counter in the request matches the generation-counter in the stored data, then the count of stored data elements MUST be zero. Otherwise, all relevant data values MUST be
returned. A nonexistent value is represented as a value with an empty data value portion and no signature. In particular, if a dictionary key that does not exist is requested, then there must be a dictionary entry with that key but an empty value.

7.2.3. REMOVE

The REMOVE request is used to remove a stored element or elements from the storing peer. Although each kind defines its own access control requirements, in general only the original signer of the data should be allowed to remove it. Any successful remove of an existing element for a given kind MUST increment the generation counter by at least one.

A remove-request has exactly the same syntax as a FETCH request except that each entry represents a set of values to be removed rather than returned. The same kind-id MUST NOT be used twice in a given remove-request. Each fetch_kind_data is then processed in turn. These operations MUST be atomic. If any operation fails, the state MUST be rolled back to before the request was received.

Before processing the REMOVE request, the peer MUST perform the following checks.

- The kind-id is known.
- The signature over the message is valid or (depending on overlay policy) no signature is required.
- The signer of the message has permissions which permit him to remove this kind of data.
- If the generation-counter is non-zero, it must equal the current value of the generation-counter for this kind. This feature allows the generation counter to be used in a way similar to the HTTP Etag feature.

Assuming that the request is permitted, the operations proceed as follows.

7.2.3.1. Single Value

A REMOVE of a single value element simple causes it not to exist. If no such element exists, then this simply is a silent success.

7.2.3.2. Array

A REMOVE of an array element (or element range) replaces those elements with empty elements. Note that this does not cause the array to be packed. An array which contains ["A", "B", "C"] and then has element 0 removed produces an array containing [NA, "B", "C"].
Note, however, that the removal of the final element of the array shortens the array, so in the above case, the removal of element 2 makes the array ["A", "B"].

7.2.3.3. Dictionary

A REMOVE of a dictionary element (or elements) replaces those elements with empty elements. If no such elements exist, then this is a silent success.

7.2.3.4. Response Definition

The response to a successful REMOVE simply contains a list of the new generation counters for each kind-id, using the same syntax as the response to a STORE request. Note that if the generation counter does not change, that means that the requested items did not exist. However, if the generation counter does change, that does not mean that the items existed.

7.2.4. FIND

The FIND request is used to explore the DHT. A FIND request for a resource-id R and a kind-id T retrieves the resource-id (if any) of the resource of kind T known to the target peer which is closes to R. This method can be used to walk the DHT by interactively fetching R_{n+1}=nearest(1 + R_n).

7.2.4.1. Request Definition

The FIND_Q message contains a series of resource-IDs and kind-ids identifying the resource the peer is interested in.

<table>
<thead>
<tr>
<th>STRUCTURE: find_q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>01234567890123456789012</td>
</tr>
<tr>
<td>+-------------------+</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>++</td>
</tr>
</tbody>
</table>

Jennings, et al. Expires August 27, 2008 [Page 65]
The request contains a list of kind-ids which the FIND is for, as indicated below.

Resource - The desired resource-id
Ids - The desired kind-ids. Each value MUST only appear once.

### 7.2.4.2. Response Definition

A response to a successful FIND request is a FIND_A message containing the closest resource-ID for each kind specified in the request.

**STRUCTURE: find_kind_data**

```
+--------+--------+--------+--------+
|       |       |       |       |
| 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 |
| +-------------------------------------------+ |
| | Kind |                                             |
| +-------------------------------------------+ |
| 004 +-----------------------------------------+ |
| | Closest Len |                                      |
| +-------------------+                               |
| |                  | Closest                                        |
| +-------------------+                               |
| /                   | /                                             |
| +-------------------+                               |
```

If the processing peer is not responsible for the specified resource-id, it SHOULD return a 404 error.

When each kind is defined, it can indicate if the kind is not allowed to be used in a FIND request. This would be done to help achieve some types of security properties for the data stored in that kind.

For each kind-id in the request the response MUST contain a find_response_value indicating the closest resource-id for that kind-id unless the kind is not allowed to be used with FIND in which case a find_kind_data for that kind_id MUST NOT be included in the response. If a kind-id is not known, then the corresponding resource-id MUST be 0. Note that different kind-ids may have different closest resource-ids.

The response is simply a series of find_kind_data elements, one per kind, concatenated end-to-end. The contents of each element are:
Kind - The kind-id.
Closest - The closest resource ID to the specified resource ID.
   This is 0 if no resource ID is known.

Note that the response does not contain the contents of the data
stored at these resource-ids. If the requester wants this, it must
retrieve it using FETCH.

7.3. DHT Maintenance

This section describes methods that are expected to be useful for all
DHTs. These methods have generic semantics (join, leave, update) and
some common fields, but where appropriate allow room for DHT-specific
data.

7.3.1. JOIN

A new peer (but which already has credentials) uses the JOIN_Q
message to join the DHT. The JOIN_Q is sent to the peer which
previously was responsible for the resource-id corresponding to the
peer-id which the new peer has. This notifies the responsible peer
that the new peer is taking over some of the overlay and it needs to
synchronize its state.

STRUCTURE: join_q

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
+-----------------------------------------------+
  004 +-----------------------------------------------+
        | Desired Peer Id|
  008 +-----------------------------------------------+
        |                                                |
  012 +-----------------------------------------------+
        |                                                |
  016 +-----------------------------------------------+
        |                                                |
    /                                           |
    /                                           |
    /                                           |
    /                                           |
    | Dht Specific Data                           |
    |                                             |
    |                                             |
+-----------------------------------------------+
```

The default JOIN_Q contains only the peer-id which the sending peer
wishes to assume. DHTs MAY specific other data to appear in this
request.
The responding peer responds with success or failure. However, if it is success it MUST follow up by executing the right sequence of STOREs and UPDATEs to transfer the appropriate section of the overlay space to the joining peer. In addition, DHTs MAY define data to appear in the response payload.

7.3.2. LEAVE

The LEAVE_Q message is used to indicate that a peer is exiting the overlay. The peer SHOULD send this message to each peer with which it is directly connected prior to exiting the overlay.

STRUCTURE: leave_q

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
| Leaving Peer Id                                              |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
+-------------------------------------------------------------------+
| Dht Specific Data                                                |
|                                                               |
|                                                               |
+-------------------------------------------------------------------+
```

The default LEAVE_Q contains only the peer-id of the leaving peer. DHTs MAY specific other data to appear in this request.

Upon receiving a LEAVE request, a peer MUST update its own routing and routing table, and send the appropriate STORE/UPDATE sequences to re-stabilize the overlay.

7.3.3. UPDATE

Update is the primary DHT-specific maintenance message. It is used by the sender to notify the recipient of the sender’s view of the current state of the overlay and it is up to the recipient to take whatever actions are appropriate to deal with the state change.

The contents of the UPDATE_Q message are completely DHT-specific. The UPDATE_A response is expected to be either success or an error.
7.3.4. ROUTE_QUERY

The ROUTE_QUERY request allows the sender to ask a peer where they would route a message directed to a given destination. In other words, a ROUTE-QUERY for destination X requests the peer-id where the receiving peer would next route to get to X. A ROUTE-QUERY can also request that the receiving peer initiate an UPDATE request to transfer his routing table.

One important use of the ROUTE-QUERY request is to support iterative routing. The way that his works is that the sender selects one of the peers in its neighbor table and sends it a ROUTE-QUERY message with the destination_object set to the peer-id/resource-id it wishes to route to. The neighbor responds with the next peer-id to send to. The sending peer then CONNECTs to that peer and repeats the ROUTE-QUERY. Eventually, the sender gets a response from a peer containing a peer-id that is the same as that peer. At that point, the sender can send whatever request is needed directly to that peer.

Note that this procedure only works well if all the peers are mutually directly reachable--either by all having public IP addresses or at least by all being behind the same NAT. Accordingly, peers MUST only use this method if permitted by the overlay configuration (see Section 10.2).

7.3.4.1. Request Definition

A ROUTE_QUERY_Q message indicates the peer or resource that the requesting peer is interested in. It also contains a "send_update" option allowing the requesting peer to request a full copy of the other peer’s routing table.

STRUCTURE: route_query_q

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Send Update  |                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Destination Object                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The contents of the ROUTE_QUERY_Q message are as follows:
send_update: A single byte. This may be set to 1 to indicate that the requester wishes the responder to initiate an UPDATE request immediately. Otherwise, this value MUST be set to zero.

destination_object: The destination which the requester is interested in. This may be any valid destination object, including a peer-id, compressed ids, or resource-id

7.3.4.2. Response Definition

A response to a successful ROUTE_QUERY request is a ROUTE_QUERY_A message containing the address of the peer to which the responding peer would have routed the request message in recursive routing.

STRUCTURE: route_query_a

0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
004 +                                                               +
|                           Next Peer                           |
008 +                                                               +
|                                                               |
012 +                                                               +
|                                                               |
016 +----------------------------------------------------------------+
|                   /                                               |
|                   /                                               |
|                   /                                               |
|                   /                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

The contents of the ROUTE_QUERY_A are as follows:

next_peer: The peer to which the responding peer would route the message to in order to deliver it to the destination listed in the request.

next_addr: The address of the next peer.

If the requester set the send_update flag, the responder SHOULD initiate an UPDATE immediately after.

8. ICE and Connection Formation

At numerous times during the operation of RELOAD, a node will need to establish a connection to another node. This may be for the purposes
of building finger tables when the node joins the P2P network, or when the node learns of a new neighbor through an UPDATE and needs to establish a connection to that neighbor.

In addition, a node may need to connect to another node for the purposes of an application connection. In the case of SIP, when a node has looked up the target AOR in the DHT, it will obtain a Node-ID that identifies that peer. The next step will be to establish a "direct" connection for the purposes of performing SIP signaling.

In both of these cases, the node starts with a destination Node-ID, and its objective is to create a connection (ideally using TCP, but falling back to UDP when it is not available) to the node with that given Node-ID. The establishment of this connection is done using the CONNECT request in conjunction with ICE. It is assumed that the reader has familiarity with ICE.

RELOAD implementations MUST implement full ICE. Because RELOAD always tries to use TCP and then UDP as a fallback, there will be multiple candidates of the same IP version, which requires full ICE.

8.1. Overview

To utilize ICE, the CONNECT method provides a basic offer/answer operation that exchanges a set of candidates for a single "stream". In this case, the "stream" refers not to RTP or other types of media, but rather to a connection for RELOAD itself or for SIP signaling. The CONNECT request contains the candidates for this stream, and the CONNECT response contains the corresponding answer with candidates for that stream. Though CONNECT provides an offer/answer exchange, it does not actually carry or utilize Session Description Protocol (SDP) messages. Rather, it carries the raw ICE parameters required for ICE operation, and the ICE spec is utilized as if these parameters had actually been used in an SDP offer or answer. In essence, ICE is utilized by mapping the CONNECT parameters into an SDP for the purposes of following the details of ICE itself. That avoids the need for RELOAD to respecify ICE, yet allows it to operate without the baggage that SDP would bring.

In addition, RELOAD only allows for a single offer/answer exchange. Unlike the usage of ICE within SIP, there is never a need to send a subsequent offer to update the default candidates to match the ones selected by ICE.

RELOAD and SIP always run over TLS for TCP connections and DTLS [RFC4347] for UDP "connections". Consequently, once ICE processing has completed, both agents will begin TLS and DTLS procedures to
establish a secure link. It’s important to note that, had a TURN server been utilized for the TCP or UDP stream, the TURN server will transparently relay the TLS messaging and the encrypted TLS content, and thus will not have access to the contents of the connection once it is established. Any attack by the TURN server to insert itself as a man-in-the-middle are thwarted by the usage of the fingerprint mechanism of RFC 4572 [RFC4572], which will reveal that the TLS and DTLS certificates do not match the ones used to sign the RELOAD messages.

An agent follows the ICE specification as described in [I-D.ietf-mmusic-ice] and [I-D.ietf-mmusic-ice-tcp] with the changes and additional procedures described in the subsections below.

8.2. Collecting STUN Servers

ICE relies on the node having one or more STUN servers to use. In conventional ICE, it is assumed that nodes are configured with one or more STUN servers through some out-of-band mechanism. This is still possible in RELOAD but RELOAD also learns STUN servers as it connects to other peers. Because all RELOAD peers implement ICE and use STUN keepalives, every peer is a STUN server [I-D.ietf-behave-rfc3489bis]. Accordingly, any peer you know about will be willing to be a STUN server for you -- though of course it may be behind a NAT.

A peer on a well-provisioned wide-area overlay will be configured with one or more bootstrap peers. These peers make an initial list of STUN servers. However, as the peer forms connections with additional peers, it builds more peers it can use as STUN servers.

Because complicated NAT topologies are possible, a peer may need more than one STUN server. Specifically, a peer that is behind a single NAT will typically observe only two IP addresses in its STUN checks: its local address and its server reflexive address from a STUN server outside its NAT. However, if there are more NATs involved, it may discover that it learns additional server reflexive addresses (which vary based on where in the topology the STUN server is). To maximize the chance of achieving a direct connection, A peer SHOULD group other peers by the peer-reflexive addresses it discovers through them. It SHOULD then select one peer from each group to use as a STUN server for future connections.

Only peers to which the peer currently has connections may be used. If the connection to that host is lost, it MUST be removed from the list of stun servers and a new server from the same group SHOULD be selected.

OPEN ISSUE: should the peer try to keep at least one peer in each
group, even if it has no other reason for the connection? Need to specify when to stop adding new groups if the peer is behind a really bad NAT.

OPEN ISSUE: RELOAD-01 had a Peer-Info structure that allowed peers to exchange information such as a "default" IP-port pair in UPDATEs. This structure could be expanded to include the candidate list for a peer, thus allowing ICE negotiation to begin or even direct communication before a CONNECT request has been received. (The candidate pairs for the P2P port are fixed because the same source port is used for all connections.) However, because this would require significant changes to the ICE algorithm, we have not introduced such an extension at this point.

8.3. Gathering Candidates

When a node wishes to establish a connection for the purposes of RELOAD signaling or SIP signaling (or any other application protocol for that matter), it follows the process of gathering candidates as described in Section 4 of ICE [I-D.ietf-mmusic-ice]. RELOAD utilizes a single component, as does SIP. Consequently, gathering for these "streams" requires a single component.

An agent MUST implement ICE-tcp [I-D.ietf-mmusic-ice], and MUST gather at least one UDP and one TCP host candidate for RELOAD and for SIP.

The ICE specification assumes that an ICE agent is configured with, or somehow knows of, TURN and STUN servers. RELOAD provides a way for an agent to learn these by querying the ring, as described in Section 8.2 and Section 11.3.

The agent SHOULD prioritize its TCP-based candidates over its UDP-based candidates in the prioritization described in Section 4.1.2 of ICE [I-D.ietf-mmusic-ice].

The default candidate selection described in Section 4.1.3 of ICE is ignored; defaults are not signaled or utilized by RELOAD.

8.4. Encoding the CONNECT Message

Section 4.3 of ICE describes procedures for encoding the SDP. Instead of actually encoding an SDP, the candidate information (IP address and port and transport protocol, priority, foundation, component ID, type and related address) is carried within the attributes of the CONNECT request or its response. Similarly, the username fragment and password are carried in the CONNECT message or its response. Section 7.1.2 describes the detailed attribute
encoding for CONNECT. The CONNECT request and its response do not contain any default candidates or the ice-lite attribute, as these features of ICE are not used by RELOAD. The CONNECT request and its response also contain a Next-Protocol attribute, with a value of SIP or RELOAD, which indicates what protocol is to be run over the connection. The RELOAD CONNECT request MUST only be utilized to set up connections for application protocols that can be multiplexed with STUN and RELOAD itself.

Since the CONNECT request contains the candidate information and short term credentials, it is considered as an offer for a single media stream that happens to be encoded in a format different than SDP, but is otherwise considered a valid offer for the purposes of following the ICE specification. Similarly, the CONNECT response is considered a valid answer for the purposes of following the ICE specification.

Since all messages with RELOAD are secured between nodes, the node MUST implement the fingerprint attribute of RFC 4572 [RFC4572], and encode it into the CONNECT request and response as described in Section 7.1.2. This fingerprint will be matched with the certificates utilized to authenticate the RELOAD CONNECT request and its response.

Similarly, the node MUST implement the active, passive, and actpass attributes from RFC 4145 [RFC4145]. However, here they refer strictly to the role of active or passive for the purposes of TLS handshaking. The TCP connection directions are signaled as part of the ICE candidate attribute.

8.5. Verifying ICE Support

An agent MUST skip the verification procedures in Section 5.1 and 6.1 of ICE. Since RELOAD requires full ICE from all agents, this check is not required.

8.6. Role Determination

The roles of controlling and controlled as described in Section 5.2 of ICE are still utilized with RELOAD. However, the offerer (the entity sending the CONNECT request) will always be controlling, and the answerer (the entity sending the CONNECT response) will always be controlled. The connectivity checks MUST still contain the ICE-CONTROLLED and ICE-CONTROLLING attributes, however, even though the role reversal capability for which they are defined will never be needed with RELOAD. This is to allow for a common codebase between ICE for RELOAD and ICE for SDP.
8.7. Connectivity Checks

The processes of forming check lists in Section 5.7 of ICE, scheduling checks in Section 5.8, and checking connectivity checks in Section 7 are used with RELOAD without change.

8.8. Concluding ICE

The controlling agent MUST utilize regular nomination. This is to ensure consistent state on the final selected pairs without the need for an updated offer, as RELOAD does not generate additional offer/answer exchanges.

The procedures in Section 8 of ICE are followed to conclude ICE, with the following exceptions:

- The controlling agent MUST NOT attempt to send an updated offer once the state of its single media stream reaches Completed.
- Once the state of ICE reaches Completed, the agent can immediately free all unused candidates. This is because RELOAD does not have the concept of forking, and thus the three second delay in Section 8.3 of ICE does not apply.

8.9. Subsequent Offers and Answers

An agent MUST NOT send a subsequent offer or answer. Thus, the procedures in Section 9 of ICE MUST be ignored.

8.10. Media Keepalives

STUN MUST be utilized for the keepalives described in Section 10 of ICE.

8.11. Sending Media

The procedures of Section 11 apply to RELOAD as well. However, in this case, the "media" takes the form of application layer protocols (RELOAD or SIP for example) over TLS or DTLS. Consequently, once ICE processing completes, the agent will begin TLS or DTLS procedures to establish a secure connection. The fingerprint from the CONNECT request and its response are used as described in RFC 4572 [RFC4572], to ensure that another node in the P2P network, acting as a TURN server, has not inserted itself as a man-in-the-middle. Once the TLS or DTLS signaling is complete, the application protocol is free to use the connection.

The concept of a previous selected pair for a component does not apply to RELOAD, since ICE restarts are not possible with RELOAD.
8.12. Receiving Media

An agent MUST be prepared to receive packets for the application protocol (TLS or DTLS carrying RELOAD, SIP or anything else) at any time. The jitter and RTP considerations in Section 11 of ICE do not apply to RELOAD or SIP.

9. DHT Algorithms

9.1. Generic Algorithm Requirements

When specifying a new DHT, at least the following need to be described:

- Joining procedures, including the contents of the JOIN message.
- Stabilization procedures, including the contents of the UPDATE message, the frequency of topology probes and keepalives, and the mechanism used to detect when peers have disconnected.
- Exit procedures, including the contents of the LEAVE message.
- The hash algorithm used to go from a Unhashed-ID, such as a user name, to a Resource-ID. This also includes the length of the Resource-IDs and Peer-IDs
- The procedures that peers use to route messages.
- The replication strategy used to ensure data redundancy.

9.2. Chord Algorithm

This algorithm is assigned the name chord-128-2-16+ to indicate it is based on Chord, uses a 128 bit hash function, stores 2 redundant copies of all data, and has finger tables with at least 16 entries.

9.2.1. Overview

The algorithm described here is a modified version of the Chord algorithm. Each peer keeps track of a finger table of 16 entries and a neighborhood table of 6 entries. The neighborhood table contains the 3 peers before this peer and the 3 peers after it in the DHT ring. The first entry in the finger table contains the peer half-way around the ring from this peer; the second entry contains the peer that is 1/4 of the way around; the third entry contains the peer that is 1/8th of the way around, and so on. Fundamentally, the chord data structure can be thought of a doubly-linked list formed by knowing the successors and predecessor peers in the neighborhood table, sorted by the peer-id. As long as the successor peers are correct, the DHT will return the correct result. The pointers to the prior peers are kept to enable inserting of new peers into the list structure. Keeping multiple predecessor and successor pointers makes
it possible to maintain the integrity of the data structure even when consecutive peers simultaneously fail. The finger table forms a skip list, so that entries in the linked list can rapidly be found - it needs to be there so that peers can be found in $O(\log(N))$ time instead of the typical $O(N)$ time that a linked list would provide.

A peer, n, is responsible for a particular Resource-ID k if k is less than or equal to n and k is greater than p, where p is the peer id of the previous peer in the neighborhood table. Care must be taken when computing to note that all math is modulo $2^{128}$.

9.2.2. Routing

If a peer is not responsible for a Resource-ID k, then it routes a request to that location by routing it to the peer in either the neighborhood or finger table that has the largest peer-id that is in the interval between the peer and k.

9.2.3. Redundancy

When a peer receives a STORE request for Resource-ID k, and it is responsible for Resource-ID k, it stores the data and returns a SUCCESS response. [[Open Issue: should it delay sending this SUCCESS until it has successfully stored the redundant copies?]]. It then sends a STORE request to its successor in the neighborhood table and to that peer's successor. Note that these STORE requests are addressed to those specific peers, even though the Resource-ID they are being asked to store is outside the range that they are responsible for. The peers receiving these check they came from an appropriate predecessor in their neighborhood table and that they are in a range that this predecessor is responsible for, and then they store the data.

Note that a malicious node can return a success response but not store the data locally or in the replica set. Requesting peers which wish to ensure that the replication actually occurred SHOULD contact each peer listed in the replicas field of the STORE response and retrieve a copy of the data. [[TODO: Do we want to have some optimization in FETCH where they can retrieve just a digest instead of the data values?]]

9.2.4. Joining

The join process for a joining party (JP) with peer-id n is as follows.
1. JP connects to its chosen bootstrap node.
2. JP uses a series of PINGs to populate its routing table.
3. JP sends CONNECT requests to initiate connections to each of the peers in the connection table as well as to the desired finger table entries. Note that this does not populate their routing tables, but only their connection tables, so JP will not get messages that it is expected to route to other nodes.
4. JP enters all the peers it contacted into its routing table.
5. JP sends a JOIN to its immediate successor, the admitting peer (AP) for peer-id n. The AP sends the response to the JOIN.
6. AP does a series of STORE requests to JP to store the data that JP will be responsible for.
7. AP sends JP an UPDATE explicitly labeling JP as its predecessor. At this point, JP is part of the ring and responsible for a section of the overlay. AP can now forget any data which is assigned to JP and not AP.
8. AP sends an UPDATE to all of its neighbors with the new values of its neighbor set (including JP).
9. JP sends UPDATES to all the peers in its routing table.

In order to populate its routing table, JP sends a PING via the bootstrap node directed at resource-id n+1 (directly after its own resource-id). This allows it to discover its own successor. Call that node p0. It then sends a ping to p0+1 to discover its successor (p1). This process can be repeated to discover as many successors as desired. The values for the two peers before p will be found at a later stage when n receives an UPDATE.

In order to set up its neighbor table entry for peer i, JP simply sends a CONNECT to peer \((n+2^{(\text{numBitsInPeerId}-i)})\). This will be routed to a peer in approximately the right location around the ring.

9.2.5. Routing CONNECTs

When a peer needs to CONNECT with a new peer in its neighborhood table, it MUST source-route the CONNECT request through the peer from which it learned the new peer’s peer-id. Source-routing these requests allows the overlay to recover from instability.

All other CONNECT requests, such as those for new finger table entries, are routed conventionally through the overlay.

If a peer is unable to successfully CONNECT with a peer that should be in its neighborhood, it MUST locate either a TURN server or another peer in the overlay, but not in its neighborhood, through which it can exchange messages with its neighbor peer.
9.2.6. UPDATEs

An UPDATE is defined as

STRUCTURE: chord_update

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Predecessors Len|                                               |
+-+-+-+-+-+-+-+                                               +
|                          Predecessors                         |
/                                                               /
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Successors Len|                                               |
+-+-+-+-+-+-+-+                                               +
|                          Successors                          |
/                                                               /
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The contents of this message are:

Predecessors - The predecessor set of the UPDATEing peer.
Successors - The successor set of the UPDATEing peer.

A peer MUST maintain an association (via CONNECT) to every member of its neighbor set. A peer MUST attempt to maintain at least three predecessors and three successors. However, it MUST send its entire set in any UPDATE message.

9.2.6.1. Sending UPDATEs

Every time a connection to a peer in the neighborhood set is lost (as determined by connectivity pings or failure of some request), the peer should remove the entry from its neighborhood table and replace it with the best match it has from the other peers in its routing table. It then sends an UPDATE to all its remaining neighbors. The update will contain all the peer-ids of the current entries of the table (after the failed one has been removed). Note that when replacing a successor the peer SHOULD delay the creation of new replicas for 30 seconds after removing the failed entry from its neighborhood table in order to allow a triggered update to inform it of a better match for its neighborhood table.

If connectivity is lost to all three of the peers that succeed this peer in the ring, then this peer should behave as if it is joining
the network and use PINGs to find a peer and send it a JOIN. If
connectivity is lost to all the peers in the finger table, this peer
should assume that it has been disconnected from the rest of the
network, and it should periodically try to join the DHT.

9.2.6.2. Receiving UPDATEs

When a peer, N, receives an UPDATE request, it examines the peer-ids
in the UPDATE_Q and at its neighborhood table and decides if this
UPDATE_Q would change its neighborhood table. This is done by taking
the set of peers currently in the neighborhood table and comparing
them to the peers in the update request. There are three major
cases:

- The UPDATE_Q contains peers that would not change the neighbor set
  because they match the neighborhood table.
- The UPDATE_Q contains peers closer to N than those in its
  neighborhood table.
- The UPDATE_Q defines peers that indicate a neighborhood table
  further away from N than some of its neighborhood table. Note
  that merely receiving peers further away does not demonstrate
  this, since the update could be from a node far away from N.
  Rather, the peers would need to bracket N.

In the first case, no change is needed.

In the second case, N MUST attempt to CONNECT to the new peers and if
it is successful it MUST adjust its neighbor set accordingly. Note
that it can maintain the now inferior peers as neighbors, but it MUST
remember the closer ones.

The third case implies that a neighbor has disappeared, most likely
because it has simply been disconnected but perhaps because of
overlay instability. N MUST PING the questionable peers to discover
if they are indeed missing and if so, remove them from its
neighborhood table.

After any PINGs and CONNECTs are done, if the neighborhood table
changes, the peer sends an UPDATE request to each of its neighbors
that was in either the old table or the new table. These UPDATE
requests are what ends up filling in the predecessor/successor tables
of peers that this peer is a neighbor to. A peer MUST NOT enter
itself in its successor or predecessor table and instead should leave
the entries empty.

A peer N which is responsible for a resource-id R discovers that the
replica set for R (the next two nodes in its successor set) has
changed, it MUST send a STORE for any data associated with R to any
new node in the replica set. It SHOULD not delete data from peers which have left the replica set.

When a peer \( N \) detects that it is no longer in the replica set for a resource \( R \) (i.e., there are three predecessors between \( N \) and \( R \)), it SHOULD delete all data associated with \( R \) from its local store.

9.2.6.3. Stabilization

There are four components to stabilization:
1. exchange UPDATES will all peers in its routing table to exchange state
2. search for better peers to place in its finger table
3. search to determine if the current finger table size is sufficiently large
4. search to determine if the overlay has partitioned and needs to recover

A peer MUST periodically send an UPDATE request to every peer in its routing table. The purpose of this is to keep the predecessor and successor lists up to date and to detect connection failures. The default time is about every ten minutes, but the enrollment server SHOULD set this in the configuration document using the "chord-128-2-16+-update-frequency" element (denominated in seconds.) A peer SHOULD randomly offset these UPDATE requests so they do not occur all at once. If an UPDATE request fails or times out, the peer MUST mark that entry in the neighbor table invalid and attempt to reestablish a connection. If no connection can be established, the peer MUST attempt to establish a new peer as its neighbor and do whatever replica set adjustments are required.

Periodically a peer should select a random entry \( i \) from the finger table and do a PING to peer \( (n+2^{(numBitsInPeerId-i)}) \). The purpose of this is to find a more accurate finger table entry if there is one. This is done less frequently than the connectivity checks in the previous section because forming new connections is somewhat expensive and the cost needs to be balanced against the cost of not having the most optimal finger table entries. The default time is about every hour, but the enrollment server SHOULD set this in the configuration document using the "chord-128-2-16+-ping-frequency" element (denominated in seconds). If this returns a different peer than the one currently in this entry of the peer table, then a new connection should be formed to this peer and it should replace the old peer in the finger table.

As an overlay grows, more than 16 entries may be required in the finger table for efficient routing. To determine if its finger table is sufficiently large, one an hour the peer should perform a PING to
determine whether growing its finger table by four entries would result in it learning at least two peers that it does not already have in its neighbor table. If so, then the finger table SHOULD be grown by four entries. Similarly, if the peer observes that its closest finger table entries are also in its neighbor table, it MAY shrink its finger table to the minimum size of 16 entries. [[OPEN ISSUE: there are a variety of algorithms to gauge the population of the overlay and select an appropriate finger table size. Need to consider which is the best combination of effectiveness and simplicity.]]

To detect that a partitioning has occurred and to heal the overlay, a peer P MUST periodically repeat the discovery process used in the initial join for the overlay to locate an appropriate bootstrap peer, B. If an overlay has multiple mechanisms for discovery it should randomly select a method to locate a bootstrap peer. P should then send a PING for its own peer-ID routed through B. If a response is received from a peer S’, which is not P’s successor, then the overlay is partitioned and P should send a CONNECT to S’ routed through B, followed by an UPDATE sent to S’. (Note that S’ may not be in P’s neighborhood table once the overlay is healed, but the connection will allow S’ to discover appropriate neighbor entries for itself via its own stabilization.)

9.2.7. Leaving

Peers SHOULD send a LEAVE request prior to exiting the DHT. Any peer which receives a LEAVE for a peer n in its neighbor set must remove it from the neighbor set, update its replica sets as appropriate (including STOREs of data to new members of the replica set) and send UPDATEDs containing its new predecessor and successor tables.

10. Enrollment and Bootstrap

10.1. Discovery

When a peer first joins a new overlay, it starts with a discovery process to find an enrollment server. Related work to the approach used here is described in [I-D.garcia-p2psip-dns-sd-bootstrapping] and [I-D.matthews-p2psip-bootstrap-mechanisms]. The peer first determines the overlay name. This value is provided by the user or some other out of band provisioning mechanism. If the name is an IP address, that is directly used otherwise the peer MUST do a DNS SRV query using a Service name of "p2p_enroll" and a protocol of tcp to find an enrollment server.

If the overlay name ends in .local, then the DNS SRV lookup is done
using implement [I-D.cheshire-dnsext-dns-sd] with a Service name of "p2p_menroll" can also be tried to find an enrollment server. If they implement this, the user name can be used as the Instance Identifier label.

Once an address for the enrollment servers is determined, the peer forms an HTTPS connection to that IP address. The certificate MUST match the overlay name as described in [RFC2818]. The peer then performs a GET to the URL formed by appending a path of "/p2psip/enroll" to the overlay name. For example, if the overlay name was example.com, the URL would be "https://example.com/p2psip/enroll".

The result is an XML configuration file with the syntax described in the following section.

10.2. Overlay Configuration

This specification defines a new content type "application/p2p-overlay+xml" for an MIME entity that contains overlay information. This information is fetched from the enrollment server, as described above. An example document is shown below. An example document is shown below.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<overlay name="chord.example.com" expiration="86400">
  <dht name="chord-128-2-8"/>
  <root-cert>[DER certificate here]</root-cert>
  <required-usage name="SIP"/>
  <credential-server url="https://www.example.com/csr"/>
  <bootstrap-peer address="192.0.2.2" port="5678"/>
  <bootstrap-peer address="192.0.2.3" port="5678"/>
  <bootstrap-peer address="192.0.2.4" port="5678"/>
  <multicast-bootstrap="192.0.2.99" port="5678"/>
</overlay>
```

The file MUST be a well formed XML document and it SHOULD contain an encoding declaration in the XML declaration. If the charset parameter of the MIME content type declaration is present and it is different from the encoding declaration, the charset parameter takes precedence. Every application conformance to this specification MUST accept the UTF-8 character encoding to ensure minimal interoperability. The namespace for the elements defined in this specification is urn:ietf:params:xml:ns:p2p:overlay.

The file can contain multiple "overlay" elements where each one contains the configuration information for a different overlay. Each "overlay" has the following attributes:
name: name of the overlay
expiration: time in future at which this overlay configuration is
not longer valid and need to be retrieved again. This is
expressed in seconds from the current time.

Inside each overlay element, the following elements can occur:

dht - This element has an attribute called name that describes which
DHT algorithm is being used.
root-cert - This element contains a DER encoded X.509v3 certificate
that is the root trust store used to sign all certificates in this
overlay. There can be more than one of these.
required-usage - This element has an attribute called "name" that
describes a usage that peers in this overlay are required to
support. More than one required-usage element may be present.
credential-server - This element contains the URL at which the
credential server can be reached in a "url" element. This URL
MUST be of type "https:". More than one credential-server element
may be present.
bootstrap-peer - This elements represents the address of one of the
bootstrap peers. It has an attribute called "address" that
represents the IP address (either IPv4 or IPv6, since they can be
distinguished) and an attribute called "port" that represents the
port. More than one bootstrap-peer element may be present.
multicast-bootstrap - This element represents the address of a
multicast address and port that may be used for bootstrap and that
peers SHOULD listen on to enable bootstrap. It has an attributed
called "address" that represents the IP address and an attribute
called "port" that represents the port. More than one "multicast-
bootstrap" element may be present.
iterative-permitted - This element indicates that iterative routing
(see Section 7.3.4) MAY be used. If iterative routing is
permitted, then this value MUST be set to "TRUE". Otherwise, it
SHOULD be absent, but MAY be set to "FALSE".

[[TODO: Do a RelaxNG grammar.]]

10.3. Credentials

If the configuration document contains a credential-server element,
credentials are required to use the DHT. A peer which does not yet
have credentials MUST contact the credential server to acquire them.

In order to acquire credentials, the peer generates an asymmetric key
pair and then generates a "Simple Enrollment Request" (as defined in
[I-D.ietf-pkix-2797-bis]) and sends this over HTTPS as defined in
[I-D.ietf-pkix-cmc-trans] to the URL in the credential-server
element. The subjectAltName in the request MUST contain the required
user name(s).

The credential server MUST authenticate the request using HTTP digest [RFC2617]. If the authentication succeeds and the requested user name(s) is acceptable, the server and returns a certificate. The SubjectAltName field in the certificate contains the following values:

- One or more Peer-IDs which MUST be cryptographically random [RFC4086]. These MUST be chosen by the credential server in such a way that they are unpredictable to the requesting user.
- The names this user is allowed to use in the overlay

The certificate is returned in a "Simple Enrollment Response".

The client MUST check that the certificate returned was signed by one of the certificates received in the "root-cert" list of the overlay configuration data. The peer then reads the certificate to find the Peer-IDs it can use.

10.3.1. Credentials for HIP

When RELOAD is used with HIP, the certificates MUST be generated so that:

- Each node is assigned a unique ORCHID.
- The peer-id can be uniquely determined from the ORCHID.

Because in general, ORCHIDs are shorter than peer-ids, this means that the ORCHIDS MUST be generated first and MUST be cryptographically random in order to make the peer-ids cryptographically random. The mapping function used to produce the peer-id from the ORCHID MUST be the same as that used by the DHT to produce resource-ids from Unhashed-IDs.

In addition to the usual attributes, when HIP is in use certificates MUST contain a subjectAltName with an iPAddress value containing the HIP ORCHID. This allows these certificates to be used by the HIP peers during the HIP base exchange.

10.4. Locating a Peer

In order to join the overlay, the peer MUST contact a peer. Typically this means contacting the bootstrap peers, since they are guaranteed to have public IP addresses (the system should not advertise them as bootstrap peers otherwise). If the peer has cached peers it SHOULD contact them first by sending a PING request to the known peer address with the destination peer-id set to that peer’s peer-id.
If no cached peers are available, then the peer SHOULD send a PING request to the address and port found in the broadcast-peers element in the configuration document. This MAY be a multicast or anycast address. The PING should use the wildcard peer-id as the destination peer-id.

The responder peer that receives the PING request SHOULD check that the overlay name is correct and that the requester peer sending the request has appropriate credentials for the overlay before responding to the PING request even if the response is only an error.

When the requester peer finally does receive a response from some responding peer, it can note the peer-id in the response and use this peer-id to start sending requests to join the DHT as described in Section 3.1.5 and Section 7.3.

After a peer has successfully joined the overlay network, it SHOULD periodically look at any peers to which it has managed to form direct connections. Some of these peers MAY be added to the cached-peers list and used in future boots. Peers that are not directly connected MUST NOT be cached. The RECOMMENDED number of peers to cache is 10.

11. Usages

11.1. Generic Usage Requirements

A new usage MUST specify the following information:

- The kind-ids which the usage defines and what each kind means.
- The data model for the data being stored (single value, array, dictionary, etc.) for each kind
- Access control rules for each kind, indicating what credentials are allowed to read and write that kind-id at a given location.
- The minimum amounts of data of each kind that a conformant implementation MUST store.

While each kind MUST define what data model is used for its data, that does not mean that it must define new data models. Where practical, kind SHOULD use the built-in data models. However, they MAY define any new required data models. The intention is that the basic data model set be sufficient for most applications/usages.

Note: New usages MAY reuse existing kind-ids. New kind-ids only need to be defined where different data is stored or different behavior is required.
11.2. SIP Usage

The SIP usage allows a RELOAD overlay to be used as a distributed SIP registrar/proxy network. The basic function of the SIP usage is to allow Alice to start with a SIP URI (e.g., "bob@dht.example.com") and end up with a connection which Bob’s SIP UA can use to pass SIP messages back and forth to Alice’s SIP UA. Provides the following three functions:

- Mapping SIP URIs that are not GRUUs to the overlay peer responsible for the SIP UA.
- Mapping SIP GRUUs to the DHT peer responsible for the SIP UA.
- Forming a connection directly to a DHT peer that is used to send SIP messages to the SIP UA.

Section 3.7.1 provides an overview of how these fit together.

11.2.1. SIP-REGISTRATION kind

The first mapping is provided using the SIP-REGISTRATION kind-id:

Kind IDs The Unhashed-ID for the SIP-REGISTRATION kind-id is a URI, typically the AOR for the user. The data stored is a sip-registration-data, which can contain either another URI or a destination list to the peer which is acting for the user. [[TODO: we want to somehow put caller-pref in here along with the route list, but I’m not sure how to do it yet.]]

Data Model The data model for the SIP-REGISTRATION kind-id is dictionary. The dictionary key is the peer-id of the storing peer. This allows each peer (presumably corresponding to a single device) to store a single route mapping.

Access Control If certificate-based access control is being used, stored data of kind-id SIP-REGISTRATION must be signed by a certificate which (1) contains user name matching the storing URI used as the Unhashed-ID for the resource-id and (2) contains a peer-id matching the storing dictionary key.

Data Sizes Peers MUST be prepared to store SIP-REGISTRATION values of up to 10 kilobytes and must be prepared to store up to 10 values for each user name.

The contents of the SIP-REGISTRATION kind are
A registration may contain either a URI (type code 0x01) or a contact preferences structure and a destination list (type code 0x01). The leading byte indicates the type.
11.2.2. GRUUs

GRUUs do not require storing data in the DHT. Rather, they are constructed by embedding a base64-encoded destination list in the gr URI parameter of the GRUU. The base64 encoding is done with the alphabet specified in table 1 of RFC 4648 with the exception that ~ is used in place of =. An example GRUU is "sip:alice@example.com;gr=MDEyMzQ1Njc4OTAxMjM0NTY3ODk~". When a peer needs to route a message to a GRUU in the same P2P network, it simply uses the destination list and connects to that peer.

Anonymous GRUUs are done in roughly the same way but require either that the enrollment server issue a different peer-id for each anonymous GRUU required or that a destination list be used that includes a peer that compresses the destination list to stop the peer-id from being revealed.

11.2.3. SIP Connect

Once the destination list for a user has been identified, the calling peer uses the CONNECT request to form a connection to the peer identified by the destination list. The CONNECT request MUST contain the connect-application value of 5160 (SIP). If certificate-based authentication is in use, the responding peer MUST present a certificate with a peer-id matching the terminal entry in the route list.

[[TODO: Note that this constrains destination lists from hiding the last peer-id when used here. I think that’s OK, but we should take a look]]

Once the association has been formed, the calling peer sends generic SIP messages down the new association and ordinary SIP procedures are followed.

11.2.4. SIP Tunnel

This usage allows two peers to exchange SIP messages across the overlay using the TUNNEL method. TUNNEL is provided as an alternative to using CONNECT because it allows a SIP message to be sent immediately, without the delay associated with CONNECT. For a simple SIP exchange, it may result in fewer messages being sent.

An implementation SHOULD use CONNECT for a dialog that is expected to endure for sufficient time and exchange significant numbers of messages. An implementation MAY establish an initial dialog using TUNNElIng and then migrate it to a direct dialog opened with CONNECT once that negotiation is complete.
As an application of TUNNEL, this usage defines the following items:

- For SIP, the application attribute is 5060.
- The application MAY establish any dialog using TUNNEL if it expects to replace it once a CONNECT request completes. The application SHOULD NOT exchange messages with another SIP UA repeatedly using a TUNNEL unless it is unable to complete a CONNECT.
- The Replaces header should be used to migrate dialogs established via TUNNEL to a direct connection.
- The dialogid is the GRUU of the destination of the request.
- By using the GRUU of the destination as the dialogid, the receiving peer is able to deliver the message to the appropriate process without parsing the SIP message.

In constructing the message, the SIP UA forms the message as if it were being routed directly to the GRUU of the destination. The SIP stack hands the message to RELOAD for delivery. Although the message is passed through a sequence of untrusted peers, it is not subject to modification by those peers because of the message’s signature.

OPEN ISSUE: should specify how to request encryption of the message end-to-end.

Note: The easiest implementation of TUNNEL is likely to default to sending all messages across a TUNNEL when the first message is sent to a new destination GRUU and simultaneously issuing a CONNECT. Messages then continue through the TUNNEL until the CONNECT completes, at which point they are delivered via the new connection.

OPEN ISSUE: If the tunneling vs direct decision can be made equivalently to a link-layer decision, it may not be necessary to modify the dialog or inform the SIP UA in any way that it has now obtained a direct route.

11.3. TURN Usage

When a node starts up, it joins the overlay network and forms several connection in the process. If the ICE stage in any of these connection return a reflexive address that is not the same as the peers perceived address, then the peers is behind a NAT and not an candidate for a TURN server. Additionally, if the peers IP address is in the private address space range, then it is not a candidate for a TURN server. Otherwise, the peer SHOULD assume it is a potential TURN server and follow the procedures below.

If the node is a candidate for a TURN server it will insert some
pointers in the overlay so that other peers can find it. The overlay configuration file specifies a `turnDensity` parameter that indicates how many times each TURN server should record itself in the overlay. Typically this should be set to the reciprocal of the estimate of what percentage of peers will act as TURN servers. For each value, called `d`, between 1 and `turnDensity`, the peer forms a Unhashed-ID by concatenating its peer-ID and the value `d`. This Unhashed-ID is hashed to form a Resource-ID. The address of the peer is stored at that Resource-ID using type TURN-SERVICE and the turn-server production:

Note: Correct functioning of this algorithm depends critically on having `turnDensity` be an accurate estimate of the true density of TURN servers. If `turnDensity` is too high, then the process of finding TURN servers becomes extremely expensive as multiple candidate resource-ids must be probed.

Peers that provide the STUN-Relay server type need to support the TURN extensions to STUN for media relay of both UDP and TCP traffic as defined in [I-D.ietf-behave-turn] and [I-D.ietf-behave-tcp].
STRUCTURE: turn_server

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

STRUCTURE: ip4_address_type

<table>
<thead>
<tr>
<th>ip4_addr_type</th>
<th>Addr</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

STRUCTURE: ip6_address_type

<table>
<thead>
<tr>
<th>ip6_addr_type</th>
<th>Addr</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x02</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[[OPEN ISSUE: This structure only works for TURN servers that have public addresses. It may be possible to use TURN servers that are behind well-behaved NATs by first ICE connecting to them. If we decide we want to enable that, this structure will need to change to either be a peer-id or include that as an option.]]

Kind IDs This usage defines the TURN-SERVICE kind-id to indicate that a peer is willing to act as a TURN server. The FIND command MUST return results for the TURN-SERVICE kind-id.
Data Model  The TURN-SERVICE stores a single value for each resource-id.

Access Control  If certificate-based access control is being used, stored data of kind TURN-SERVICE MUST be authenticated by a certificate which contains a peer-id which when hashed with the iteration counter produces the resource-id being stored at.

Data Sizes  TURN-SERVICE values are of fixed size. Peers MUST be prepared to store values with iteration counter of up to 100.

The data is stored in a data structure with the IP address of the server and an indication whether the address is an IPv4 or IPv6 address. The Unhashed-ID used to form the storage Resource-ID is simply the peer-id. The access control rule is that the certificate used to sign the request must contain a peer-id that when hashed would match the Resource-ID where the data is being stored.

Peers can find other servers by selecting a random Resource-ID and then doing a FIND request for the appropriate server type with that Resource-ID. The FIND request gets routed to a random peer based on the Resource-ID. If that peer knows of any servers, they will be returned. The returned response may be empty if the peer does not know of any servers, in which case the process gets repeated with some other random Resource-ID. As long as the ratio of servers relative to peers is not too low, this approach will result in finding a server relatively quickly.

Open issues: Should there be low and high bandwidth version of STUN-Relay one can find? Low would be usable for signaling type things and high would be usable for audio, video, and others.

11.4.  Certificate Store Usage

The Certificate Store usage allows a peer to store its certificate in the overlay, thus avoiding the need to send a certificate in each message - a reference may be sent instead.

A user/peer MUST store its certificate at resource-ids derived from two Unhashed-IDs:

- The user names in the certificate.
- The peer-ids in the certificate.

Note that in the second case the certificate is not stored at the peer’s peer-id but rather at a hash of the peer’s peer-id. The intention here (as is common throughout RELOAD) is to avoid making a peer responsible for its own data.

A peer MUST ensure that the user’s certificates are stored in the DHT.
when joining and redo the check about every 24 hours after that. Certificate data should be stored with an expiry time of 60 days. When a client is checking the existence of data, if the expiry is less than 30 days, it should be refreshed to have an expiry of 60 days. The certificate information is frequently used for many operations, and peers should cache it for 8 hours.

Kind IDs This usage defines the CERTIFICATE kind-id to store a peer or user’s certificate.
Data Model The data model for CERTIFICATE data is array.
Access Control The CERTIFICATE MUST contain a peer-id or user name which, when hashed, maps the resource-id at which the value is being stored.
Data Sizes Peers MUST be prepared to store at least 10 certificates of sizes up to 1K each.

11.5. HIP Tunnel

This usage allows two peers to exchange HIP messages across the overlay using the TUNNEL method. This is meant to be used as part of the HIP BONE architecture described in [I-D.camarillo-hip-bone].

As an application of TUNNEL, this usage defines the following items:

- For HIP, the application attribute is TBD (IANA port number).
- The dialogid is empty and set to zero length.

In order to route HIP messages correctly, there needs to be an unambiguous mapping between the ORCHID assigned to each HIP node and the peer-id assigned to that node. The ORCHID MUST be used as the Unhashed-ID to generate the peer-id. [TODO: We need a general scheme for mapping Unhashed-IDs to IDs so they don’t collide. This isn’t the place to define it.] Messages to a given ORCHID are then routed to the mapped peer-id. Section 10.3 describes more considerations for the generation of ORCHIDs for use with RELOAD.

TODO - should discuss interaction of HIP and P2P retransmission timers

11.6. Diagnostic Usage

[[TODO: reduce text of motivation description in the next version]]

The development and deployment of a peer-to-peer system is a continuous process. The developers write code which is tested on a scale that may be smaller than the actual deployment size. After this local testing, the code is deployed in a real environment. Bugs arise during development and deployment phases. The designers of the
peer-to-peer system need mechanisms which can help identify problems and bugs in a peer-to-peer system during development and deployment phases. Peer-to-peer systems are an example of a distributed system and it is not a trivial task to provide protocol mechanisms, tools and techniques to identify problems that may arise in such systems.

The diagnostic mechanisms can broadly be classified into online and offline mechanisms. The online mechanisms attempt to identify faults in a running system where as offline mechanisms try to infer faults by gathering the log files of machines participating in a distributed system.

In a peer-to-peer system, a peer maintains routing state to forward messages according to the overlay protocol being used. In addition, a peer stores information published by other peers. The routing and storage of resources consume network, space (memory), and CPU resources. A peer also needs to keep track of how long the P2PSIP application has been running and the last time peers in the routing table were last contacted. During development and deployment phase, an overlay designer needs mechanisms to query some or all of the above mentioned information.

The overlay designer may also treat overlay as a black box and determine if the routing mechanisms are working correctly under various levels of churn.

Thus, there are at least two types of online diagnostic mechanisms: 1) state acquisition 2) black-box diagnostics

11.6.1. State Acquisition Mechanisms

The protocol provides a DIAGNOSTIC method [TODO] which queries the peer for its routing state, average bandwidth, CPU utilization, and storage state. The DIAGNOSTIC request should typically be sent over a reliable transport protocol as the response will likely exceed UDP MTU size. The state acquisition mechanism can be used to construct a local view of the connectivity state of the system. It can also be used to construct a geographical map of the system.

Below, we identify potential issues with the state acquisition mechanisms.

Security: If any peer can query the routing or storage state of any other peer, then clearly privacy and security concerns arise. To address this, the state acquisition mechanisms need an access list like mechanism so that only the overlay implementer can query the state of all the nodes. Alternatively, the state acquisition mechanisms are only enabled during the development phase or are only
enabled for 'admin' users.

Scalability: It is possible to query the state of few hundred or a few thousand nodes (as it is currently done in our live system on Planet lab); however, a serial state acquisition of a million node is a non starter. In large scale networks, one option is to query the state of few hundred nodes and to construct an high level connectivity map. CAIDA [ref] collects data at a few vantage points to construct BGP maps.

Instantaneous vs. long term state: Another issue with these state acquisitions mechanisms is whether they acquire the instantaneous state snapshot or an exponential moving average or a list of snapshots over a period of time. For diagnostic metrics such as CPU utilization, an exponential moving average metric is also helpful in addition to the instantaneous snapshot.

Pull vs. push: The state acquisition mechanisms can either be pull-based or push-based or a combination of both. In pull-based mechanisms, peer explicitly request state of another peer. This may not be sufficient because pull-based mechanisms require a to periodically poll a peer for any change state. In a push-based mechanism, peers advertise any change in certain metrics to their routing or neighbor peers. As an example of push-based mechanism, a peer which starts to relay a call may indicate a change in its bandwidth to its routing or neighbor peers in a PING message.

Development vs. deployment: A hard problem is to decide which diagnostics are absolutely necessary during deployment and which are needed during development.

Clearly, complete state acquisition has security concerns in a deployed system. The other option an overlay implementer can use is to run a few peers and have complete control over the functionality of these peers. These peers are same as other peers with the difference that an overlay implementer can explicitly query the state of these peers. It can then use this information to ‘crawl’ the overlay network and construct a local map of the network.

11.6.2. Black-box diagnostics

[[TODO: a better name for this section]]

Black-box diagnostics: DHTs are examples of structured peer-to-peer networks and they allow nodes to store key/value pairs in the overlay. A simple diagnostic mechanism is to treat the overlay as a black-box: publish several key/value pairs at one peer and then look them up from another peer. For this kind of diagnostic mechanism,
clients are more suitable as they do not provide any routing or storage services to the overlay and can connect to an arbitrary peer.

The Diagnostic Usage allow a peer to report various statistics about itself that may be useful for diagnostics or performance management. It can be used to discover information such as the software version, uptime, and performance statistics of a peer. The usage defines several new kinds which can be retrieved to get the statistics. The peer-id is directly used when retrieving data so no Unhashed-ID is defined. The access control model for all of these is local policy defined by the peer. The peer MAY have a list of users (such as "admin") that it is willing to return the information for and restrict access to users with that name. The access control can be determined on a per kind basis - for example, a node may be willing to return the software version to any users while specific information about performance may not be returned.

The following kinds are defined:

SOFTWARE_VERSION A single value element containing a US-ASCII string that identifies the manufacture, model, and version of the software.

UPTIME A single value element containing an unsigned 64-bit integer specifying the time the nodes has been up in seconds.

AS_NUMBER A single value element containing the Autonomous System [TODO REF] number as an unsigned 32-bit integer. Zero is returned if the AS number is unknown.

(OPEN ISSUES: How to determine a AS number? This metric is primarily used for advertising and locating STUN/TURN servers. A TURN server is inserted and looked up under H(AS). What if there are no TURN servers in the same AS? )

CPU_UTILIZATION A single value element containing an unsigned 8-bit integer representing the percentage CPU load from 1 to 100.

(OPEN ISSUE: It is not a very precise metric.)

DATA_STORED A single value element containing an unsigned 64-bit integer representing the number of bytes of data being stored by this node.

MESSAGES_SENT An array element containing the number of messages sent and received. The array is indexed by method code. Each entry in the array is a pair of unsigned 64-bit integers (packed end to end) representing sent and received.

INSTANCES_STORED An array element containing the number of instances of each kind stored. The array is index by kind-id. Each entry is an unsigned 64-bit integer.
11.6.3. Diagnostic Metrics for a P2PSIP Deployment

Clearly, all diagnostic metrics are useful during development and testing. The hard question is which metrics are absolutely necessary for a deployed P2PSIP system. We attempt to identify these metrics and classify them under ‘resource’ and ‘peer’ metrics.

For ‘resource’ metric, we identify CPU_UTILIZATION, EWMA_BYTES_SENT, EWMA_BYTES_RCVD, and MEMORY_FOOTPRINT as the key metrics and for ‘peer’ metric we identify UPTIME, LAST_CONTACT, and RTT as the metrics that are crucial for a deployed P2PSIP system.

(OPEN QUESTION: any other metrics?)

(OPEN: Below, we sketch how these metrics can be used. A peer can use EWMA_BYTES_SENT and EWMA_BYTES_RCVD of another peer to infer whether it is acting as a media relay. It may then choose not to forward any requests for media relay to this peer. Similarly, among the various candidates for filling up routing table, a peer may prefer a peer with a large UPTIME value, small RTT, and small LAST_CONTACT value.)
12. Security Considerations

12.1. Overview

RELOAD provides a generic storage service, albeit one designed to be useful for P2P SIP. In this section we discuss security issues that are likely to be relevant to any usage of RELOAD. In Section 12.7 we describe issues that are specific to SIP.

In any DHT, any given user depends on a number of peers with which they have no well-defined relationship except that they are fellow members of the DHT. In practice, these other nodes may be friendly, lazy, curious, or outright malicious. No security system can provide complete protection in an environment where most nodes are malicious. The goal of security in RELOAD is to provide strong security guarantees of some properties even in the face of a large number of malicious nodes and to allow the DHT to function correctly in the face of a modest number of malicious nodes.

P2PSIP deployments require the ability to authenticate both peers and resources (users) without the active presence of a trusted entity in the system. We describe two mechanisms. The first mechanism is based on public key certificates and is suitable for general deployments. The second is based on an overlay-wide shared symmetric key and is suitable only for limited deployments in which the relationship between admitted peers is not adversarial.

12.2. Attacks on P2P Overlays

The two basic functions provided by DHT nodes are storage and routing: some node is responsible for storing a peer’s data and for allowing a peer to fetch other peer’s data. Some other set of nodes are responsible for routing messages to and from the storing nodes. Each of these issues is covered in the following sections.

P2P overlays are subject to attacks by subversive nodes that may attempt to disrupt routing, corrupt or remove user registrations, or eavesdrop on signaling. The certificate-based security algorithms we describe in this draft are intended to protect DHT routing and user registration information in RELOAD messages.

To protect the signaling from attackers pretending to be valid peers (or peers other than themselves), the first requirement is to ensure that all messages are received from authorized members of the overlay. For this reason, RELOAD transports all messages over DTLS or TLS, which provides message integrity and authentication of the directly communicating peer. In addition, when the certificate-based security system is used, messages and data are digitally signed with
the sender’s private key, providing end-to-end security for communications.

12.3. Certificate-based Security

This specification stores users’ registrations and possibly other data in a Distributed Hash table (DHT). This requires a solution to securing this data as well as securing, as well as possible, the routing in the DHT. Both types of security are based on requiring that every entity in the system (whether user or peer) authenticate cryptographically using an asymmetric key pair tied to a certificate.

When a user enrolls in the DHT, they request or are assigned a unique name, such as "alice@dht.example.net". These names are unique and are meant to be chosen and used by humans much like a SIP Address of Record (AOR) or an email address. The user is also assigned one or more peer-IDs by the central enrollment authority. Both the name and the peer ID are placed in the certificate, along with the user’s public key.

Each certificate enables an entity to act in two sorts of roles:

- As a user, storing data at specific Resource-IDs in the DHT corresponding to the user name.
- As a DHT peer with the peer ID(s) listed in the certificate.

Note that since only users of this DHT need to validate a certificate, this usage does not require a global PKI. It does, however, require a central enrollment authority which acts as the certificate authority for the DHT. This authority signs each peer’s certificate. Because each peer possesses the CA’s certificate (which they receive on enrollment) they can verify the certificates of the other entities in the overlay without further communication. Because the certificates contain the user/peer’s public key, communications from the user/peer can be verified in turn.

In order to protect data storage, in the certificate-based security scheme, all stored data is signed by the owner of the data. This allows the storing peer to verify that the storer is authorized to perform a store at that resource-id and also allows any consumer of the data to verify the provenance and integrity of the data when it retrieves it.

All implementations MUST implement certificate-based security.
12.4. Shared-Secret Security

For small environments where deployment of the PKI necessary to use a certificate-based model is impractical, RELOAD supports a shared secret security that relies on a single key that is shared among all members of the overlay. It is appropriate for small groups that wish to form a private network without complexity. In shared secret mode, all the peers share a single symmetric key which is used to key TLS-PSK [RFC4279] or TLS-SRP [I-D.ietf-tls-srp] mode. A peer which does not know the key cannot form TLS connections with any other peer and therefore cannot join the overlay.

The shared-secret scheme prohibits unauthorized peers from joining the overlay, but it provides no protection from a compromised peer inserting arbitrary resource registrations, performing a Sybil attack [Sybil], or performing other attacks on the resources or routing. Thus, it is only safe to use in limited settings in which peers are not adversarial. In addition, because the messages and data are not authenticated, each intermediate peer MUST take care to use TLS and check the other peer’s knowledge of the shared secret, or message insertion is possible.

If the shared secret key for the shared-key security scheme is discovered by an attacker, then most of the security of the scheme is lost: an attacker can impersonate any peer to any other peer. Thus, the shared-secret scheme is only appropriate for small deployments, such as a small office or ad hoc overlay set up among participants in a meeting.

One natural approach to a shared-secret scheme is to use a user-entered password as the key. The difficulty with this is that in TLS-PSK mode, such keys are very susceptible to dictionary attacks. If passwords are used as the source of shared-keys, then TLS-SRP is a superior choice because it is not subject to dictionary attacks.

12.5. Storage Security

When certificate-based security is used in RELOAD, any given Resource-ID/kind-id pair (a slot) is bound to some small set of certificates. In order to write data in a slot, the writer must prove possession of the private key for one of those certificates. Moreover, all data is stored signed by the certificate which authorized its storage. This set of rules makes questions of authorization and data integrity — which have historically been thorny for DHTs — relatively simple.

When shared-secret security is used, then all peers trust all other peers, provided that they have demonstrated that they have the
credentials to join the overlay at all. The following text therefore applies only to certificate-based security.

12.5.1. Authorization

When a client wants to store some value in a slot, it first digitally signs the value with its own private key. It then sends a STORE request that contains both the value and the signature towards the storing peer (which is defined by the Unhashed-ID construction algorithm for that particular kind of value).

When the storing peer receives the request, it must determine whether the storing client is authorized to store in this slot. In order to do so, it executes the Unhashed-ID construction algorithm for the specified kind based on the user’s certificate information. It then computes the Resource-ID from the Unhashed-ID and verifies that it matches the slot which the user is requesting to write to. If it does, the user is authorized to write to this slot, pending quota checks as described in the next section.

For example, consider the certificate with the following properties:

User name: alice@dht.example.com
Peer-Id: 013456789abcdef
Serial: 1234

If Alice wishes to STORE a value of the "SIP Location" kind, the Unhashed-ID will be the SIP AOR "sip:alice@dht.example.com". The Resource-ID will be determined by hashing the Unhashed-ID. When a peer receives a request to store a record at Resource-ID X, it takes the signing certificate and recomputes the Unhashed-ID, in this case "alice@dht.example.com". If H("alice@dht.example.com")=X then the STORE is authorized. Otherwise it is not. Note that the Unhashed-ID construction algorithm may be different for other kinds.

12.5.2. Distributed Quota

Being a peer in a DHT carries with it the responsibility to store data for a given region of the DHT. However, if clients were allowed to store unlimited amounts of data, this would create unacceptable burdens on peers, as well as enabling trivial denial of service attacks. RELOAD addresses this issue by requiring each usage to define maximum sizes for each kind of stored data. Attempts to store values exceeding this size MUST be rejected (if peers are inconsistent about this, then strange artifacts will happen when the zone of responsibility shifts and a different peer becomes responsible for overlarge data). Because each slot is bound to a small set of certificates, these size restrictions also create a
distributed quota mechanism, with the quotas administered by the central enrollment server.

Allowing different kinds of data to have different size restrictions allows new usages the flexibility to define limits that fit their needs without requiring all usages to have expansive limits.

### 12.5.3. Correctness

Because each stored value is signed, it is trivial for any retrieving peer to verify the integrity of the stored value. Some more care needs to be taken to prevent version rollback attacks. Rollback attacks on storage are prevented by the use of store times and lifetime values in each store. A lifetime represents the latest time at which the data is valid and thus limits (though does not completely prevent) the ability of the storing node to perform a rollback attack on retrievers. In order to prevent a rollback attack at the time of the STORE request, we require that storage times be monotonically increasing. Storing peers MUST reject STORE requests with storage times smaller than or equal to those they are currently storing. In addition, a fetching node which receives a data value with a storage time older than the result of the previous fetch knows a rollback has occurred.

### 12.5.4. Residual Attacks

The mechanisms described here provide a high degree of security, but some attacks remain possible. Most simply, it is possible for storing nodes to refuse to store a value (i.e., reject any request). In addition, a storing node can deny knowledge of values which it previously accepted. To some extent these attacks can be ameliorated by attempting to store to/retrieve from replicas, but a retrieving client does not know whether it should try this or not, since there is a cost to doing so.

Although the certificate-based authentication scheme prevents a single peer from being able to forge data owned by other peers. Furthermore, although a subversive peer can refuse to return data resources for which it is responsible it cannot return forged data because it cannot provide authentication for such registrations. Therefore parallel searches for redundant registrations can mitigate most of the affects of a compromised peer. The ultimate reliability of such an overlay is a statistical question based on the replication factor and the percentage of compromised peers.

In addition, when a kind is multivalued (e.g., an array data model), the storing node can return only some subset of the values, thus biasing its responses. This can be countered by using single
values rather than sets, but that makes coordination between multiple storing agents much more difficult. This is a tradeoff that must be made when designing any usage.

12.6. Routing Security

Because the storage security system guarantees (within limits) the integrity of the stored data, routing security focuses on stopping the attacker from performing a DOS attack on the system by misrouting requests in the DHT. There are a few obvious observations to make about this. First, it is easy to ensure that an attacker is at least a valid peer in the DHT. Second, this is a DOS attack only. Third, if a large percentage of the peers on the DHT are controlled by the attacker, it is probably impossible to perfectly secure against this.

12.6.1. Background

In general, attacks on DHT routing are mounted by the attacker arranging to route traffic through or two nodes it controls. In the Eclipse attack [Eclipse] the attacker tampers with messages to and from nodes for which it is on-path with respect to a given victim node. This allows it to pretend to be all the nodes that are reachable through it. In the Sybil attack [Sybil], the attacker registers a large number of nodes and is therefore able to capture a large amount of the traffic through the DHT.

Both the Eclipse and Sybil attacks require the attacker to be able to exercise control over her peer IDs. The Sybil attack requires the creation of a large number of peers. The Eclipse attack requires that the attacker be able to impersonate specific peers. In both cases, these attacks are limited by the use of centralized, certificate-based admission control.

12.6.2. Admissions Control

Admission to an RELOAD DHT is controlled by requiring that each peer have a certificate containing its peer ID. The requirement to have a certificate is enforced by using TLS mutual authentication on each connection. Thus, whenever a peer connects to another peer, each side automatically checks that the other has a suitable certificate. These peer IDs are randomly assigned by the central enrollment server. This has two benefits:

- It allows the enrollment server to limit the number of peer IDs issued to any individual user.
- It prevents the attacker from choosing specific peer IDs.

The first property allows protection against Sybil attacks (provided
the enrollment server uses strict rate limiting policies). The second property deters but does not completely prevent Eclipse attacks. Because an Eclipse attacker must impersonate peers on the other side of the attacker, he must have a certificate for suitable peer IDs, which requires him to repeatedly query the enrollment server for new certificates which only will match by chance. From the attacker’s perspective, the difficulty is that if he only has a small number of certificates the region of the DHT he is impersonating appears to be very sparsely populated by comparison to the victim’s local region.

12.6.3. Peer Identification and Authentication

In general, whenever a peer engages in DHT activity that might affect the routing table it must establish its identity. This happens in two ways. First, whenever a peer establishes a direct connection to another peer it authenticates via TLS mutual authentication. All messages between peers are sent over this protected channel and therefore the peers can verify the data origin of the last hop peer for requests and responses without further cryptography.

In some situations, however, it is desirable to be able to establish the identity of a peer with whom one is not directly connected. The most natural case is when a peer UPDATEs its state. At this point, other peers may need to update their view of the DHT structure, but they need to verify that the UPDATE message came from the actual peer rather than from an attacker. To prevent this, all DHT routing messages are signed by the peer that generated them.

[TODO: this allows for replay attacks on requests. There are two basic defenses here. The first is global clocks and loose anti-replay. The second is to refuse to take any action unless you verify the data with the relevant node. This issue is undecided.]

[TODO: I think we are probably going to end up with generic signatures or at least optional signatures on all DHT messages.]

12.6.4. Protecting the Signaling

The goal here is to stop an attacker from knowing who is signaling what to whom. An attacker being able to observe the activities of a specific individual is unlikely given the randomization of IDs and routing based on the present peers discussed above. Furthermore, because messages can be routed using only the header information, the actual body of the RELOAD message can be encrypted during transmission.

There are two lines of defense here. The first is the use of TLS or
DTLS for each communications link between peers. This provides protection against attackers who are not members of the overlay. The second line of defense, if certificate-based security is used, is to digitally sign each message. This prevents adversarial peers from modifying messages in flight, even if they are on the routing path.

### 12.6.5. Residual Attacks

The routing security mechanisms in RELOAD are designed to contain rather than eliminate attacks on routing. It is still possible for an attacker to mount a variety of attacks. In particular, if an attacker is able to take up a position on the DHT routing between A and B it can make it appear as if B does not exist or is disconnected. It can also advertise false network metrics in attempt to reroute traffic. However, these are primarily DoS attacks.

The certificate-based security scheme secures the namespace, but if an individual peer is compromised or if an attacker obtains a certificate from the CA, then a number of subversive peers can still appear in the overlay. While these peers cannot falsify responses to resource queries, they can respond with error messages, effecting a DoS attack on the resource registration. They can also subvert routing to other compromised peers. To defend against such attacks, a resource search must still consist of parallel searches for replicated registrations.

### 12.7. SIP-Specific Issues

#### 12.7.1. Fork Explosion

Because SIP includes a forking capability (the ability to retarget to multiple recipients), fork bombs are a potential DoS concern. However, in the SIP usage of RELOAD, fork bombs are a much lower concern because the calling party is involved in each retargeting event and can therefore directly measure the number of forks and throttle at some reasonable number.

#### 12.7.2. Malicious Retargeting

Another potential DoS attack is for the owner of an attractive number to retarget all calls to some victim. This attack is difficult to ameliorate without requiring the target of a SIP registration to authorize all stores. The overhead of that requirement would be excessive and in addition there are good use cases for retargeting to a peer without there explicit cooperation.
12.7.3. Privacy Issues

All RELOAD SIP registration data is public. Methods of providing location and identity privacy are still being studied.

13. IANA Considerations

This section contains the new code points registered by this document. The IANA policies are TBD.

13.1. Overlay Algorithm Types

IANA SHALL create/(has created) a "RELOAD Overlay Algorithm Type" Registry. Entries in this registry are strings denoting the names of DHT algorithms. The registration policy for this registry is TBD.

The initial contents of this registry are:

chord-128-2-8 - The algorithm defined in Section 9.2 of this document.

13.2. Data Kind-Id

IANA SHALL create/(has created) a "RELOAD Data Kind-Id" Registry. Entries in this registry are 32-bit integers denoting data kinds, as described in Section 11.1. The registration policy for this registry is TBD.

The initial contents of this registry are:

<table>
<thead>
<tr>
<th>Kind</th>
<th>Kind-Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIP-REGISTRATION</td>
<td>TBD</td>
</tr>
<tr>
<td>TURN_SERVICE</td>
<td>TBD</td>
</tr>
<tr>
<td>CERTIFICATE</td>
<td>TBD</td>
</tr>
<tr>
<td>SOFTWARE_VERSION</td>
<td>TBD</td>
</tr>
<tr>
<td>UPTIME</td>
<td>TBD</td>
</tr>
<tr>
<td>AS_NUMBER</td>
<td>TBD</td>
</tr>
<tr>
<td>CPU_UTILIZATION</td>
<td>TBD</td>
</tr>
<tr>
<td>DATA_STORED</td>
<td>TBD</td>
</tr>
<tr>
<td>MESSAGES_SENT</td>
<td>TBD</td>
</tr>
<tr>
<td>INSTANCES_STORED</td>
<td>TBD</td>
</tr>
<tr>
<td>ROUTING_TABLE_SIZE</td>
<td>TBD</td>
</tr>
<tr>
<td>NEIGHBOR_TABLE_SIZE</td>
<td>TBD</td>
</tr>
</tbody>
</table>
13.3. Data Model

IANA SHALL create/(has created) a "RELOAD Data Model" Registry. Entries in this registry are 8-bit integers denoting data models, as described in Section 3.1.4. The registration policy for this registry is TBD.

+-----------------+-------+
| Data Model      | Identifier |
+-----------------+-------+
| SINGLE_VALUE    | TBD   |
| ARRAY           | TBD   |
| DICTIONARY      | TBD   |

13.4. Message Codes

IANA SHALL create/(has created) a "RELOAD Message Code" Registry. Entries in this registry are 16-bit integers denoting method codes as described in Section 4.2.1 The registration policy for this registry is TBD.

The initial contents of this registry are:
### 14. Error Codes

IANA SHALL create/(has created) a "RELOAD Error Code" Registry.

Entries in this registry are 16-bit integers denoting error codes.

[[TODO: Complete this once we decide on error code strategy.]]

### 15. Examples

See draft [TODO add ref] for message flow examples.

### 16. Acknowledgments

This draft is a merge of the "REsource LOcation And Discovery"
(RELOAD)" draft by David A. Bryan, Marcia Zangrilli and Bruce B. Lowekamp, the "Address Settlement by Peer to Peer" draft by Cullen Jennings, Jonathan Rosenberg, and Eric Rescorla, the "Security Extensions for RELOAD" draft by Bruce B. Lowekamp and James Deverick, the "A Chord-based DHT for Resource Lookup in P2PSIP" by Marcia Zangrilli and David A. Bryan, and the Peer-to-Peer Protocol (P2PP) draft by Salman A. Baset, Henning Schulzrinne, and Marcin Matuszewski.

Thanks to the many people who contributed including: Michael Chen, TODO - fill in.

17. Appendix: Operation with SIP clients outside the DHT domain

18. Appendix: Notes on DHT Algorithm Selection

An important point: if you assume NATs are doing ICE to set up connections, you want a lot fewer connections than you might have on a very open network - this might push towards something like Chord with fewer connections than, say, bamboo.

TODO - ref draft-irtf-p2prg-survey-search

19. References

19.1. Normative References


Informative References

[I-D.ietf-behave-tcp]
Guha, S., "NAT Behavioral Requirements for TCP",

[I-D.ietf-p2psip-concepts]
Bryan, D., "Concepts and Terminology for Peer to Peer SIP",
draft-ietf-p2psip-concepts-00 (work in progress),


progress), October 2007.

[I-D.camarillo-hip-bone]

Authors’ Addresses

Cullen Jennings
Cisco
170 West Tasman Drive
MS: SJC-21/2
San Jose, CA  95134
USA

Phone:  +1 408 421-9990
Email:  fluffy@cisco.com

Bruce B. Lowekamp
SIPeerior; William & Mary
3000 Easter Circle
Williamsburg, VA  23188
USA

Phone:  +1 757 565 0101
Email:  lowekamp@sipeerior.com

Eric Rescorla
Network Resonance
2064 Edgewood Drive
Palo Alto, CA  94303
USA

Phone:  +1 650 320-8549
Email:  ekr@networkresonance.com
Jonathan Rosenberg  
Cisco  
Edison, NJ  
USA  
Email: jdrosen@cisco.com  

Salman A. Baset  
Columbia University  
1214 Amsterdam Avenue  
New York, NY  
USA  
Email: salman@cs.columbia.edu  

Henning Schulzrinne  
Columbia University  
1214 Amsterdam Avenue  
New York, NY  
USA  
Email: hgs@cs.columbia.edu
Full Copyright Statement

Copyright (C) The IETF Trust (2008).

This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY, THE IETF TRUST AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Intellectual Property

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at http://www.ietf.org/ipr.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Acknowledgment

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).