P2PSIP   C. Jennings
Internet-Draft   Cisco
Intended status: Standards Track   B. Lowekamp, Ed.
Expires: June 15, 2009   unaffiliated
                                                             E. Rescorla
                                                       Network Resonance
                                                                S. Baset
                                                          H. Schulzrinne
                                                     Columbia University
                                                       December 12, 2008

REsource LOcation And Discovery (RELOAD) Base Protocol
draft-ietf-p2psip-base-01

Status of this Memo

Abstract

   This document defines REsource LOcation And Discovery (RELOAD), a
   peer-to-peer (P2P) signaling protocol for use on the Internet. A P2P
   signaling protocol provides its clients with an abstract storage and
   messaging 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 kinds of data 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 that do not need to route traffic or store data
   for others.

Table of Contents

[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 generic, self-organizing overlay network service, allowing
nodes to efficiently route messages to other nodes and to efficiently
store and retrieve data in the overlay. RELOAD provides several
features that are critical for a successful P2P protocol for the
Internet:

Security Framework: A P2P network will often be established among a
   set of peers that do not trust each other. RELOAD leverages a
   central enrollment server to provide credentials for each peer
   which can then be used to authenticate each operation. This
   greatly reduces the possible attack surface.

Usage Model: RELOAD is designed to support a variety of
   applications, including P2P multimedia communications with the
   Session Initiation Protocol [I-D.ietf-p2psip-sip]. RELOAD allows
   the definition of new application usages, each of which can define
   its own data types, along with the rules for their use. This
   allows RELOAD to be used with new applications through a simple
   documentation process that supplies the details for each
   application.

NAT Traversal: RELOAD is designed to function in environments where
   many if not most of the nodes are behind NATs or firewalls.
   Operations for NAT traversal are part of the base design,
   including using ICE to establish new RELOAD or application
   protocol connections.

High Performance Routing: The very nature of overlay 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 with a
   simple, lightweight forwarding header, thus minimizing the amount
   of effort required by intermediate peers.

Pluggable Overlay 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" overlay algorithm will allow interoperability, while
   the extensibility allows selection of overlay algorithms optimized
   for a particular application.

These properties were designed specifically to meet the requirements for a P2P protocol to support SIP. This document defines the base protocol for the distributed storage and location service, as well as critical usages for NAT traversal and security. The SIP Usage itself is described separately in [I-D.ietf-p2psip-sip]. 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. Basic Setting

In this section, we provide a brief overview of the operational setting for RELOAD. See the concepts document for more details. A RELOAD Overlay Instance consists of a set of nodes arranged in a partly connected graph. Each node in the overlay is assigned a numeric Node-ID which, together with the specific overlay algorithm in use, determines its position in the graph and the set of nodes it connects to. The figure below shows a trivial example which isn't drawn from any particular overlay algorithm, but was chosen for convenience of representation.

```
            +--------+              +--------+              +--------+
            | Node 10|--------------| Node 20|--------------| Node 30|
            +--------+              +--------+              +--------+
                |                       |                       |
                |                       |                       |
            +--------+              +--------+              +--------+
            | Node 40|--------------| Node 50|--------------| Node 60|
            +--------+              +--------+              +--------+
                |                       |                       |
                |                       |                       |
            +--------+              +--------+              +--------+
            | Node 70|--------------| Node 80|--------------| Node 90|
            +--------+              +--------+              +--------+
                                        |
                                        |
                                    +--------+
                                    | Node 85|
                                    |(Client)|
                                    +--------+
```
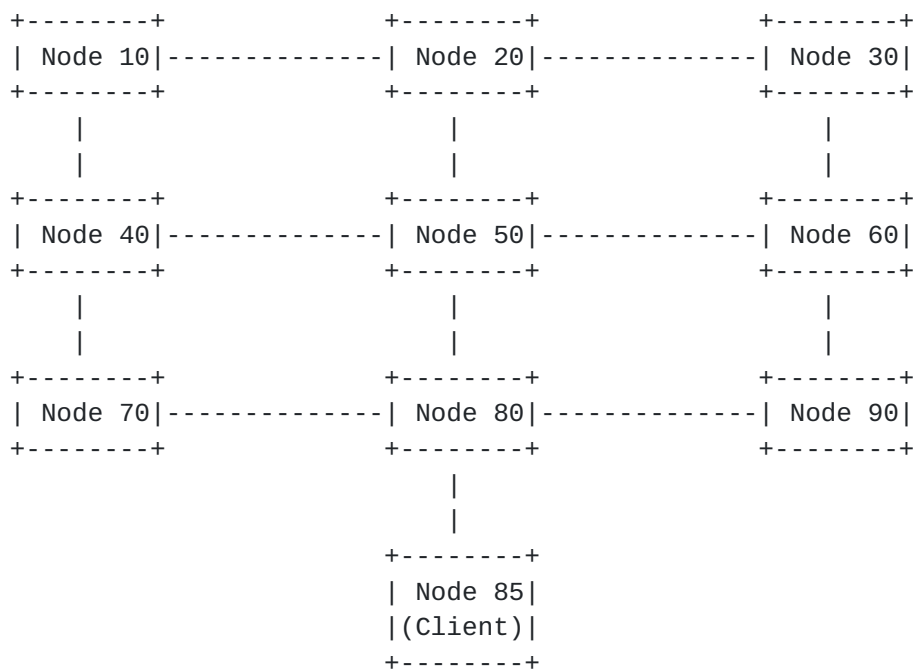
Because the graph is not fully connected, when a node wants to send a message to another node, it may need to route it through the network. For instance, Node 10 can talk directly to nodes 20 and 40, but not to Node 70. In order to send a message to Node 70, it would first send it to Node 40 with instructions to pass it along to Node 70. Different overlay algorithms will have different connectivity graphs, but the general idea behind all of them is to allow any node in the

graph to efficiently reach every other node within a small number of hops.

The RELOAD network is not only a messaging network. It is also a storage network. Records are stored under numeric addresses which occupy the same space as node identifiers. Nodes are responsible for storing the data associated with some set of addresses as determined by their Node-ID. For instance, we might say that every node is responsible for storing any data value which has an address less than or equal to its own Node-ID, but greater than the next lowest Node-ID. Thus, Node-20 would be responsible for storing values 11-20.

RELOAD also supports clients. These are nodes which have Node-IDs but do not participate in routing or storage. For instance, in the figure above Node 85 is a client. It can route to the rest of the RELOAD network via Node 80, but no other node will route through it and Node 90 is still responsible for all addresses between 81-90. We refer to non-client nodes as peers.

Other applications (for instance, SIP) can be defined on top of RELOAD and use these two basic RELOAD services to provide their own services.

1.2. Architecture

RELOAD is fundamentally an overlay network. Therefore, it can be divided into components that mimic the layering of the Internet model[RFC1122].

```
             Application

        +-------+  +-------+
        | SIP   |  | XMPP  |  ...
        | Usage |  | Usage |
        +-------+  +-------+
      ------------------------------------ Messaging API
    +------------------+     +---------+
    |     Message      |<--->| Storage |
    |    Transport     |     +---------+
    +------------------+          ^
         ^        ^               |
         |        v               v
         |     +------------------+
         |     |     Topology     |
         |     |      Plugin      |
         |     +------------------+
         |          ^
         v          v
      +------------------+
      |  Forwarding &    |
      | Link Management  |
      +------------------+
      ------------------------------------ Overlay Link API
        +-------+  +------+
        |TLS    |  |DTLS  |  ...
        +-------+  +------+
```
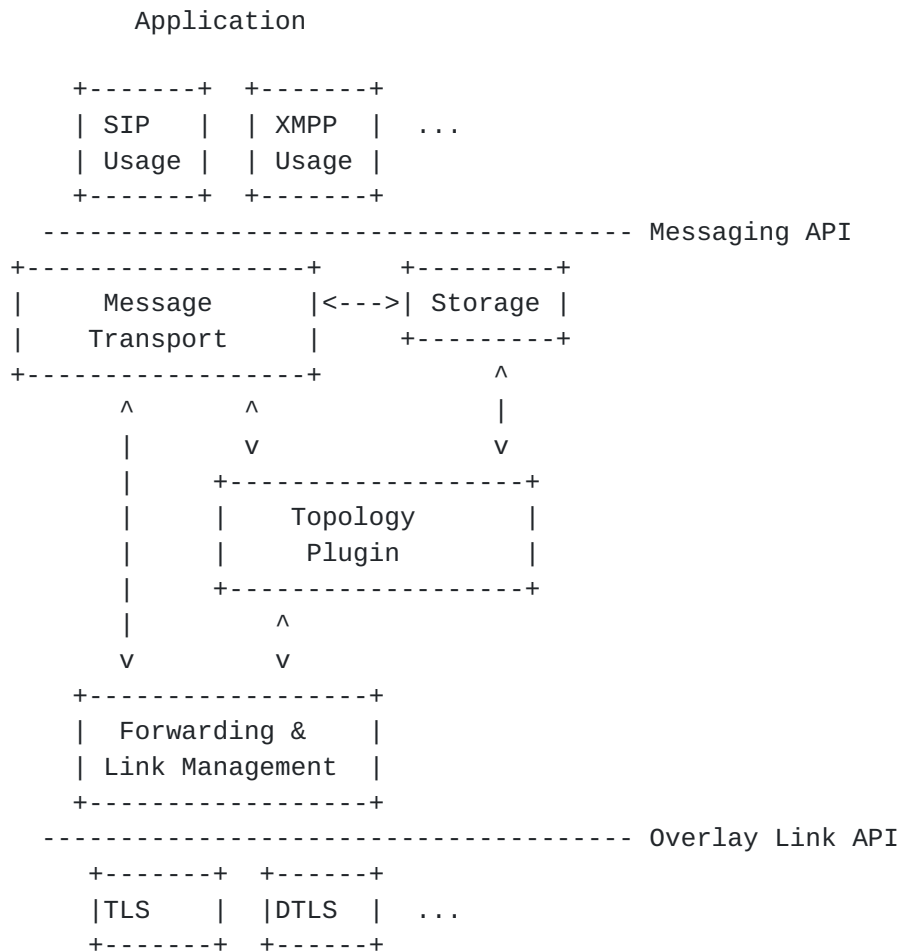
The major components of RELOAD are:

Usage Layer: Each application defines a RELOAD usage; a set of data
   kinds and behaviors which describe how to use the services
   provided by RELOAD. These usages all talk to RELOAD through a
   common Message Transport API.

Message Transport: Handles the end-to-end reliability, manages
   request state for the usages, and forwards Store and Fetch
   operations to the Storage component. Delivers message responses
   to the component initiating the request.

Storage: The Storage component is responsible for processing
   messages relating to the storage and retrieval of data. It talks
   directly to the Topology Plugin to manage data replication and
   migration, and it talks to the Message Transport to send and
   receive messages.

   Topology Plugin: The Topology Plugin is responsible for implementing
      the specific overlay algorithm being used. It uses the Message
      Transport component to send and receive overlay management
      messages, to the Storage component to manage data replication, and
      directly to the Forwarding Layer to control hop-by-hop message
      forwarding. This component closely parallels conventional routing
      algorithms, but is more tightly coupled to the Forwarding Layer
      because there is no single "routing table" equivalent used by all
      overlay algorithms.

   Forwarding and Link Management Layer: Stores and implements the
      routing table by providing packet forwarding services between
      nodes. It also handles establishing new links between nodes,
      including setting up connections across NATs using ICE.

   Overlay Link Layer: TLS and DTLS are the "link layer" protocols used
      by RELOAD for hop-by-hop communication. Each such protocol
      includes the appropriate provisions for per-hop framing or hop-by-
      hop ACKs required by unreliable transports.

   To further clarify the roles of the various layer, this figure
   parallels the architecture with each layer's role from an overlay
   perspective and implementation layer in the internet:

```
                     | Internet Model  |
         Real        |   Equivalent    |            Reload
       Internet      |   in Overlay    |         Architecture
    ---------------+-----------------+------------------------------------
                     |                 |   +-------+  +-------+
                     |   Application   |   | SIP   |  | XMPP  |  ...
                     |                 |   | Usage |  | Usage |
                     |                 |   +-------+  +-------+
                     |                 |  ----------------------------------
                     |                 |+-----------------+    +---------+
                     |   Transport     ||     Message     |<--->| Storage |
                     |                 ||     Transport   |     +---------+
                     |                 |+-----------------+          ^
                     |                 |    ^         ^              |
                     |                 |    |         v              v
       Application   |                 |    |     +-------------------+
                     |   (Routing)     |    |     |     Topology      |
                     |                 |    |     |      Plugin       |
                     |                 |    |     +-------------------+
                     |                 |    v         ^
                     |                 |              v
                     |   Network       |    +-------------------+
                     |                 |    |   Forwarding &    |
                     |                 |    | Link Management   |
                     |                 |    +-----------------+
                     |                 |  ----------------------------------
       Transport     |   Link          |    +-------+  +------+
                     |                 |    |TLS    |  |DTLS  |  ...
                     |                 |    +-------+  +------+
    ---------------+-----------------+------------------------------------
       Network       |
                     |
         Link        |
```
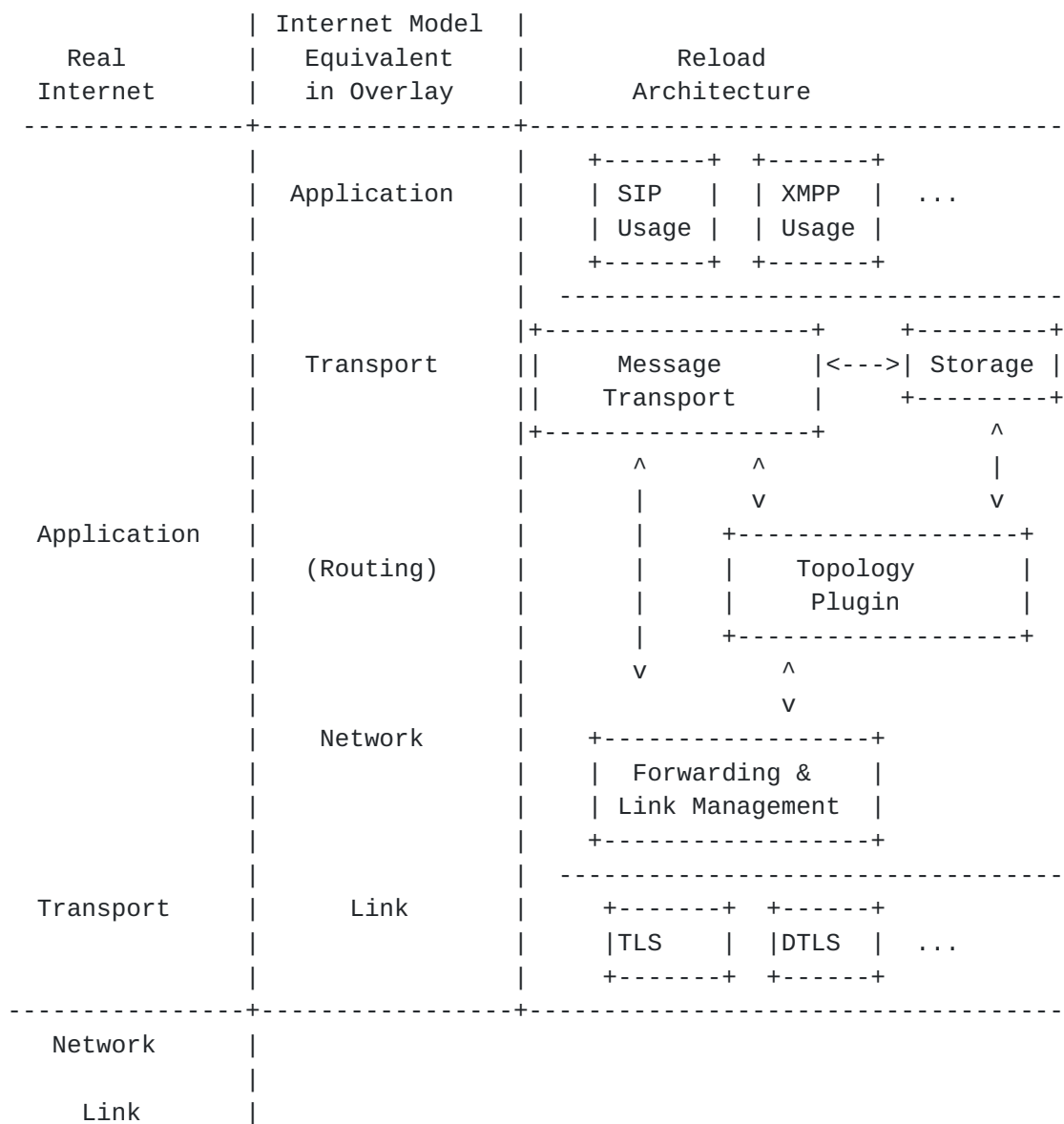
1.2.1. Usage Layer

   The top layer, called the Usage Layer, has application usages, such
   as the SIP Location Usage, that use the abstract Message Transport
   API provided by RELOAD. The goal of this layer is to implement
   application-specific usages of the generic overlay services provided
   by RELOAD. The usage defines how a specific application maps its
   data into something that can be stored in the overlay, 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. A
   single application may require multiple usages, for example a SIP
   application may also require a voicemail usage. 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.

Because the security and storage policies for each kind are dictated
by the usage defining the kind, the usages may be coupled with the
Storage component to provide security policy enforcement and to
implement appropriate storage strategies according to the needs of
the usage. The exact implementation of such an interface is outside
the scope of this draft.

This draft also defines a Diagnostics Usage, which can be used to
obtain diagnostic information about a peer in the overlay. The
Diagnostics Usage is interesting both to administrators monitoring
the overlay as well as to some overlay algorithms that base their
decisions on capabilities and current load of nodes in the overlay.

1.2.2. Message Transport

The Message Transport provides a generic message routing service for
the overlay. The Message Transport layer is responsible for end-to-
end message transactions, including retransmissions. Each peer is
identified by its location in the overlay as determined by its
Node-ID. A component that is a client of the Message Transport can
perform two basic functions:

o Send a message to a given peer specified by Node-ID or to the peer
   responsible for a particular Resource-ID.
o Receive messages that other peers sent to a Node-ID or Resource-ID
   for which this peer is responsible.

All usages rely on the Message Transport component to send and
receive messages from peers. For instance, when a usage wants to
store data, it does so by sending Store requests. Note that the
Storage component and the Topology Plugin are themselves clients of
the Message Transport, because they need to send and receive messages
from other peers.

The Message Transport API is similar to those described as providing
"Key based routing" (KBR), although as RELOAD supports different
overlay algorithms (including non-DHT overlay algorithms) that
calculate keys in different ways, the actual interface must accept
Resource Names rather than actual keys.

1.2.3. Storage

One of the major functions of RELOAD is to allow nodes to store data
in the overlay and to retrieve data stored by other nodes or by
themselves. The Storage component is responsible for processing data

storage and retrieval messages. For instance, the Storage component
might receive a Store request for a given resource from the Message
Transport. It would then query the appropriate usage before storing
the data value(s) in its local data store and sends a response to the
Message Transport for delivery to the requesting peer. Typically,
these messages will come for other nodes, but depending on the
overlay topology, a node might be responsible for storing data for
itself as well, especially if the overlay is small.

A peer's Node-ID determines the set of resources that it will be
responsible for storing. However, the exact mapping between these is
determined by the overlay algorithm used by the overlay. The Storage
component will only receive a Store request from the Message
Transport if this peer is responsible for that Resource-ID. The
Storage component is notified by the Topology Plugin when the
Resource-IDs for which it is responsible change, and the Storage
component is then responsible for migrating resources to other peers,
as required.

1.2.4. Topology Plugin

RELOAD is explicitly designed to work with a variety of overlay
algorithms. In order to facilitate this, the overlay algorithm
implementation is provided by a Topology Plugin so that each overlay
can select an appropriate overlay algorithm that relies on the common
RELOAD core protocols and code.

The Topology Plugin is responsible for maintaining the overlay
algorithm Routing Table, which is consulted by the Forwarding and
Link Management Layer before routing a message. When connections are
made or broken, the Forwarding and Link Management Layer notifies the
Topology Plugin, which adjusts the routing table as appropriate. The
Topology Plugin will also instruct the Forwarding and Link Management
Layer to form new connections as dictated by the requirements of the
overlay algorithm Topology. The Topology Plugin issues periodic
update requests through Message Transport to maintain and update its
Routing Table.

As peers enter and leave, resources may be stored on different peers,
so the Topology Plugin also keeps track of which peers are
responsible for which resources. As peers join and leave, the
Topology Plugin instructs the Storage component to issue resource
migration requests as appropriate, in order to ensure that other
peers have whatever resources they are now responsible for. The
Topology Plugin is also responsible for providing redundant data
storage to protect against loss of information in the event of a peer
failure and to protect against compromised or subversive peers.

1.2.5. **Forwarding and Link Management Layer**

   The Forwarding and Link Management Layer is responsible for getting a
   packet to the next peer, as determined by the Topology Plugin. This
   Layer establishes and maintains the network connections as required
   by the Topology Plugin. 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.

   This layer provides a fairly generic interface that allows the
   topology plugin control the overlay and resource operations and
   messages. Since each overlay algorithm is defined and functions
   differently, we generically refer to the table of other peers that
   the overlay algorithm maintains and uses to route requests
   (neighbors) as a Routing Table. The Topology Plugin actually owns
   the Routing Table, and forwarding decisions are made by querying the
   Topology Plugin for the next hop for a particular Node-ID or
   Resource-ID. If this node is the destination of the message, the
   message is delivered to the Message Transport.

   The Forwarding and Link Management Layer sits on top of the Overlay
   Link Layer protocols that carry the actual traffic. This
   specification defines how to use DTLS and TLS protocols to carry
   RELOAD messages.

1.3. **Security**

   RELOAD's security model is based on each node having one or more
   public key certificates. In general, these certificates will be
   assigned by a central server which also assigns Node-IDs, although
   self-signed certificates can be used in closed networks. These
   credentials can be leveraged to provide communications security for
   RELOAD messages. RELOAD provides communications security at three
   levels:

   Connection Level: Connections between peers are secured with TLS
      or DTLS.
   Message Level: Each RELOAD message must be signed.
   Object Level: Stored objects must be signed by the storing peer.

   These three levels of security work together to allow peers to verify
   the origin and correctness of data they receive from other peers,
   even in the face of malicious activity by other peers in the overlay.
   RELOAD also provides access control built on top of these
   communications security features. Because the peer responsible for
   storing a piece of data can validate the signature on the data being
   stored, the responsible peer can determine whether a given operation

is permitted or not.

RELOAD also provides a shared secret based admission control feature
using shared secrets and TLS-PSK. In order to form a TLS connection
to any node in the overlay, a new node needs to know the shared
overlay key, thus restricting access to authorized users.

1.4. Structure of This Document

The remainder of this document is structured as follows.

o Section 2 provides definitions of terms used in this document.
o Section 3 provides an overview of the mechanisms used to establish
   and maintain the overlay.
o Section 4 provides an overview of the mechanism RELOAD provides to
   support other applications.
o Section 5 defines the protocol messages that RELOAD uses to
   establish and maintain the overlay.
o Section 6 defines the protocol messages that are used to store and
   retrieve data using RELOAD.
o Section 7 defines the Certificate Store Usage that is fundamental
   to RELOAD security.
o Section 8 defines the TURN Server Usage needed to locate TURN
   servers for NAT traversal.
o Section 9 defines a diagnostic usage for obtaining information
   about node performance.
o Section 10 defines a specific Topology Plugin using Chord.
o Section 11 defines the mechanisms that new RELOAD nodes use to
   join the overlay for the first time.
o Section 12 provides an extended example.

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.
Terms which are new to this document (and perhaps should be added to
the concepts document) are marked with a (*).

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.

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

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

Peer: A host that is participating in the overlay. Peers are
   responsible for holding some portion of the data that has been
   stored in the overlay and also route messages on behalf of other
   hosts as required by the Overlay Algorithm.

Client: A host that is able to store data in and retrieve data from
   the overlay but which is not participating in routing or data
   storage for the overlay.

Node: We use the term "Node" to refer to a host that may be either a
   Peer or a Client. Because RELOAD uses the same protocol for both
   clients and peers, much of the text applies equally to both.
   Therefore we use "Node" when the text applies to both Clients and
   Peers and the more specific term when the text applies only to
   Clients or only to Peers.

Node-ID: A 128-bit value that uniquely identifies a node. Node-IDs
   0 and $2^{128} - 1$ are reserved and are invalid Node-IDs. A value of
   zero is not used in the wire protocol but can be used to indicate
   an invalid node in implementations and APIs. The Node-ID of
   $2^{128}-1$ is used on the wire protocol as a wildcard. (*)

Resource: An object or group of objects associated with a string
   identifier see "Resource Name" below.

Resource Name: The potentially human readable name by which a
   resource is identified. In unstructured P2P networks, the
   resource name is sometimes used directly as a Resource-ID. In
   structured P2P networks the resource name is typically mapped into
   a Resource-ID by using the string as the input to hash function.
   A SIP resource, for example, is often identified by its AOR which
   is an example of a Resource Name.(*)

Resource-ID: A value that identifies some resources and which is
   used as a key for storing and retrieving the resource. Often this
   is not human friendly/readable. 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 name. In
   unstructured networks, resource names may be used directly as
   Resource-IDs and may have variable length.

Connection Table: The set of peers to which a node is directly
   connected. This includes nodes with which Attach handshakes have
   been done but which have not sent any Updates.

Routing Table: The set of peers which a node can use to route
   overlay messages. In general, these peers will all be on the
   connection table but not vice versa, because some peers will have
   Attached 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. (*)

Destination List: A list of IDs through which a message is to be
   routed. A single Node-ID is a trivial form of destination list.
   (*)

Usage: A usage is an application that wishes to use the overlay for
   some purpose. Each application wishing to use the overlay 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.
   (*)

[3](#). Overlay Management Overview

The most basic function of RELOAD is as a generic overlay network.
Nodes need to be able to join the overlay, form connections to other
nodes, and route messages through the overlay to nodes to which they
are not directly connected. This section provides an overview of the
mechanisms that perform these functions.

[3.1](#). Security and Identification

Every node in the RELOAD overlay is identified by a Node-ID. The
Node-ID is used for three major purposes:

o To address the node itself.
o To determine its position in the overlay topology when the overlay
   is structured.
o To determine the set of resources for which the node is
   responsible.

Each node has a certificate [RFC3280] containing a Node-ID, which is
globally unique.

The certificate serves multiple purposes:

o It entitles the user to store data at specific locations in the
   Overlay Instance. Each data kind defines the specific rules for
   determining which certificates can access each Resource-ID/Kind-ID
   pair. For instance, some kinds might allow anyone to write at a
   given location, whereas others might restrict writes to the owner
   of a single certificate.
o It entitles the user to operate a node that has a Node-ID found in
   the certificate. When the node forms a connection to another
   peer, it can use this certificate so that a node connecting to it
   knows it is connected to the correct node. In addition, the node
   can sign messages, thus providing integrity and authentication for
   messages which are sent from the node.
o It entitles the user to use the user name found in the
   certificate.

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.

RELOAD supports two certificate issuance models. The first is based
on a central enrollment process which allocates a unique name and
Node-ID to the node a certificate for a public/private key pair for
the user. All peers in a particular Overlay Instance have the
enrollment server as a trust anchor and so can verify any other
peer's certificate.

In some settings, a group of users want to set up an overlay network
but are not concerned about attack by other users in the network.
For instance, users on a LAN might want to set up a short term ad hoc
network without going to the trouble of setting up an enrollment
server. RELOAD supports the use of self-generated and self-signed
certificates. When self-signed certificates are used, the node also
generates its own Node-ID and username. The Node-ID is computed as a
digest of the public key, to prevent Node-ID theft, however this
model is still subject to a number of known attacks (most notably
Sybil attacks [Sybil]) and can only be safely used in closed networks
where users are mutually trusting.

3.1.1. Shared-Key Security

   RELOAD also provides an admission control system based on shared
   keys. In this model, the peers all share a single key which is used
   to authenticate the peer-to-peer connections via TLS-PSK/TLS-SRP.

3.2. Clients

   RELOAD defines a single protocol that is used both as the peer
   protocol and the client protocol for the overlay. This simplifies
   implementation, particularly for devices that may act in either role,
   and allows clients to inject messages directly into the overlay.

   We use the term "peer" to identify a node in the overlay that routes
   messages for nodes other than those to which it is directly
   connected. Peers typically also have storage responsibilities. We
   use the term "client" to refer to nodes that do not have routing or
   storage responsibilities. When text applies to both peers and
   clients, we will simply refer to such a device as a "node."

   RELOAD's client support allows nodes that are not participating in
   the overlay as peers to utilize the same implementation and to
   benefit from the same security mechanisms as the peers. Clients
   possess and use certificates that authorize the user to store data at
   its locations in the overlay. The Node-ID in the certificate is used
   to identify the particular client as a member of the overlay and to
   authenticate its messages.

   For more discussion of the motivation for RELOAD's client support,
   see Appendix C.

3.2.1. Client Routing

   There are two routing options by which a client may be located in an
   overlay.

   o Establish a connection to the peer responsible for the client's
      Node-ID in the overlay. Then requests may be sent from/to the
      client using its Node-ID in the same manner as if it were a peer,
      because the responsible peer in the overlay will handle the final
      step of routing to the client. This will not work in overlays
      where NAT or firewall do not allow all clients to form connections
      with any other peer.
   o Establish a connection with an arbitrary peer in the overlay
      (perhaps based on network proximity or an inability to establish a
      direct connection with the responsible peer). In this case, the
      client will rely on RELOAD's Destination List feature to ensure
      reachability. The client can initiate requests, and any node in

the overlay that knows the Destination List to its current
location can reach it, but the client is not directly reachable
directly using only its Node-ID. The Destination List required to
reach it must be learnable via other mechanisms, such as being
stored in the overlay by a usage, if the client is to receive
incoming requests from other members of the overlay.

3.2.2. Minimum Functionality Requirements for Clients

A node may act as a client simply because it does not have the
resources or even an implementation of the topology plugin required
to acts as a peer in the overlay. In order to exchange RELOAD
messages with a peer, a client must meet a minimum level of
functionality. Such a client must:

o Implement RELOAD's connection-management connections that are used
   to establish the connection with the peer.
o Implement RELOAD's data retrieval methods (with client
   functionality).
o Be able to calculate Resource-IDs used by the overlay.
o Possess security credentials required by the overlay it is
   implementing.

A client speaks the same protocol as the peers, knows how to
calculate Resource-IDs, and signs its requests in the same manner as
peers. While a client does not necessarily require a full
implementation of the overlay algorithm, calculating the Resource-ID
requires an implementation of the appropriate algorithm for the
overlay.

RELOAD does not support a separate protocol for clients that do not
meet these functionality requirements. Any such extension would
either entail compromises on the features of RELOAD or require an
entirely new protocol to reimplement the core features of RELOAD.
Furthermore, for SIP and many other applications, a native
application-level protocol already exists that is sufficient for such
a client to interact with a member of the RELOAD overlay.

3.3. Routing

This section will discuss the requirements RELOAD's routing
capabilities must meet, then describe the routing features in the
protocol, and provide a brief overview of how they are used.
Appendix B discusses some alternative designs and the tradeoffs that
would be necessary to support them.

RELOAD's routing capabilities must meet the following requirements:

   NAT Traversal: RELOAD must support establishing and using
      connections between nodes separated by one or more NATs, including
      locating peers behind NATs for those overlays allowing/requiring
      it.
   Clients: RELOAD must support requests from and to clients that do
      not participate in overlay routing.
   Client promotion: RELOAD must support clients that become peers at a
      later point as determined by the overlay algorithm and deployment.
   Low state: RELOAD's routing algorithms must not require
      significant state to be stored on intermediate peers.
   Return routability in unstable topologies: At some points in
      times, different nodes may have inconsistent information about the
      connectivity of the routing graph. In all cases, the response to
      a request needs to delivered to the node that sent the request and
      not to some other node.

   To meet these requirements, RELOAD's routing relies on two basic
   mechanisms:

   Via Lists: The forwarding header used by all RELOAD messages
      contains both a Via List (built hop-by-hop as the message is
      routed through the overlay) and a Destination List (providing
      source-routing capabilities for requests and return-path routing
      for responses).
   Route_Query: The Route_Query method allows a node to query a peer
      for the next hop it will use to route a message. This method is
      useful for diagnostics and for iterative routing.

   The basic routing mechanism used by RELOAD is Symmetric Recursive.
   We will first describe symmetric routing and then discuss its
   advantages in terms of the requirements discussed above.

   Symmetric recursive routing requires a message follow the path
   through the overlay to the destination without returning to the
   originating node: each peer forwards the message closer to its
   destination. The return path of the response is then the same path
   followed in reverse. For example, a message following a route from A
   to Z through B and X:

```
    A          B          X          Z
    ------------------------------

    ---------->
    Dest=Z
               ---------->
               Via=A
               Dest=Z
                          ---------->
                          Via=A, B
                          Dest=Z


                          <----------
                          Dest=X, B, A
               <----------
                  Dest=B, A
    <----------
          Dest=A
```

Note that the preceding Figure does not indicate whether A is a
client or peer, A forwards its request to B and the response is
returned to A in the same manner regardless of A's role in the
overlay.

This figure shows use of full via-lists by intermediate peers B and
X. However, if B and/or X are willing to store state, then they may
elect to truncate the lists, save that information internally (keyed
by the transaction id), and return the response message along the
path from which it was received when the response is received. This
option requires greater state on intermediate peers but saves a small
amount of bandwidth and reduces the need for modifying the message in
route. Selection of this mode of operation is a choice for the
individual peer, the techniques are mutually interoperable even on a
single message. The figure below shows B using full via lists but X
truncating them and saving the state internally.

```
    A         B         X         Z
    -------------------------------

    ---------->
    Dest=Z
              ---------->
              Via=A
              Dest=Z
                        ---------->
                        Dest=Z

                        <----------
                              Dest=X
                  <----------
                  Dest=B, A
    <----------
         Dest=A
```

For debugging purposes, a Route Log attribute is available that
stores information about each peer as the message is forwarded.

RELOAD also supports a basic Iterative routing mode (where the
intermediate peers merely return a response indicating the next hop,
but do not actually forward the message to that next hop themselves).
Iterative routing is implemented using the Route_Query method, which
requests this behavior. Note that iterative routing is selected only
by the initiating node. RELOAD does not support an intermediate peer
returning a response that it will not recursively route a normal
request. The willingness to perform that operation is implicit in
its role as a peer in the overlay.

3.4. Connectivity Management

In order to provide efficient routing, a peer needs to maintain a set
of direct connections to other peers in the Overlay Instance. Due to
the presence of NATs, these connections often cannot be formed
directly. Instead, we use the Attach request to establish a
connection. Attach uses ICE [I-D.ietf-mmusic-ice-tcp] to establish
the connection. It is assumed that the reader is familiar with ICE.

Say that peer A wishes to form a direct connection to peer B. It
gathers ICE candidates and packages them up in an Attach request
which it sends to B through usual overlay 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 overlay

peers.

There is one special case in which Attach cannot be used: when a
peer is joining the overlay and is not connected to any peers. In
order to support this case, some small number of "bootstrap nodes"
need to be publicly accessible so that new peers can directly connect
to them. Section 11 contains more detail on this.

In general, a peer needs to maintain connections to all of the peers
near it in the Overlay Instance and to enough other peers to have
efficient routing (the details depend on the specific overlay). If a
peer cannot form a connection to some other peer, this isn't
necessarily a disaster; overlays 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. This also implies that
peers need to periodically verify that the connected peers are still
alive and if not try to reform the connection or form an alternate
one.

3.5. Overlay Algorithm Support

The Topology Plugin allows RELOAD to support a variety of overlay
algorithms. This draft defines a DHT based on Chord [Chord], which
is mandatory to implement, but the base RELOAD protocol is designed
to support a variety of overlay algorithms.

3.5.1. Support for Pluggable Overlay Algorithms

RELOAD defines three methods for overlay maintenance: Join, Update,
and Leave. However, the contents of those messages, when they are
sent, and their precise semantics are specified by the actual overlay
algorithm; RELOAD merely provides a framework of commonly-needed
methods that provides uniformity of notation (and ease of debugging)
for a variety of overlay algorithms.

3.5.2. Joining, Leaving, and Maintenance Overview

When a new peer wishes to join the Overlay Instance, it must have a
Node-ID that it is allowed to use. It uses the Node-ID in the
certificate it received from the enrollment server. The details of
the joining procedure are defined by the overlay algorithm, but the
general steps for joining an Overlay Instance are:

o Forming connections to some other peers.
o Acquiring the data values this peer is responsible for storing.

o Informing the other peers which were previously responsible for
   that data that this peer has taken over responsibility.

The first thing the peer needs to do is form a connection to some
"bootstrap node". Because this is the first connection the peer
makes, these nodes must have public IP addresses and therefore can be
connected to directly. Once a peer has connected to one or more
bootstrap nodes, it can form connections in the usual way by routing
Attach messages through the overlay to other nodes. Once a peer has
connected to the overlay for the first time, it can cache the set of
nodes it has connected to with public IP addresses for use as future
bootstrap nodes.

Once the peer has connected to a bootstrap node, it then needs to
take up its appropriate place in the overlay. This requires two
major operations:

o Forming connections to other peers in the overlay to populate its
   Routing Table.
o Getting a copy of the data it is now responsible for storing and
   assuming responsibility for that data.

The second operation is performed by contacting the Admitting Peer
(AP), the node which is currently responsible for that section of the
overlay.

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

1. JP (Joining Peer) sends a Join request to AP (Admitting Peer)
    announcing its intention to join.
2. AP sends a 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 Overlay Instance.
5. JP makes its own connections to the appropriate peers in the
    Overlay Instance.

After this process is completed, JP is a full member of the Overlay
Instance and can process Store/Fetch requests.

Note that the first node is a special case. When ordinary nodes
cannot form connections to the bootstrap nodes, then they are not
part of the overlay. However, the first node in the overlay can
obviously not connect to others nodes. In order to support this
case, potential first nodes (which must also serve as bootstrap nodes

initially) must somehow be instructed (perhaps by configuration
settings) that they are the entire overlay, rather than not part of
it.

3.6. First-Time Setup

Previous sections addressed how RELOAD works once a node has
connected. This section provides an overview of how users get
connected to the overlay for the first time. RELOAD is designed so
that users can start with the name of the overlay they wish to join
and perhaps a username and password, and leverage that into having a
working peer with minimal user intervention. This helps avoid the
problems that have been experienced with conventional SIP clients
where users are required to manually configure a large number of
settings.

3.6.1. Initial Configuration

In the first phase of the process, the user starts out with the name
of the overlay and uses this to download an initial set of overlay
configuration parameters. The user does a DNS SRV lookup on the
overlay name to get the address of a configuration server. It can
then connect to this server with HTTPS to download a configuration
document which contains the basic overlay configuration parameters as
well as a set of bootstrap nodes which can be used to join the
overlay.

3.6.2. Enrollment

If the overlay is using centralized enrollment, then a user needs to
acquire a certificate before joining the overlay. The certificate
attests both to the user's name within the overlay and to the Node-
IDs which they are permitted to operate. In that case, the
configuration document will contain the address of an enrollment
server which can be used to obtain such a certificate. The
enrollment server may (and probably will) require some sort of
username and password before issuing the certificate. The enrollment
server's ability to restrict attackers' access to certificates in the
overlay is one of the cornerstones of RELOAD's security.

4. Application Support Overview

RELOAD is not intended to be used alone, but rather as a substrate
for other applications. These applications can use RELOAD for a
variety of purposes:

   o To store data in the overlay and retrieve data stored by other
      nodes.
   o As a discovery mechanism for services such as TURN.
   o To form direct connections which can be used to transmit
      application-level messages.

   This section provides an overview of these services.

4.1. Data Storage

   RELOAD provides operations to Store, Fetch, and Remove data. Each
   location in the Overlay Instance is referenced by a 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, as shown below:

```
                  +--------------------------------+
                  |            Resource-ID         |
                  |                                |
                  | +------------+  +------------+ |
                  | |   Kind 1   |  |   Kind 2   | |
                  | |            |  |            | |
                  | | +--------+ |  | +--------+ | |
                  | | | Value  | |  | | Value  | | |
                  | | +--------+ |  | +--------+ | |
                  | |            |  |            | |
                  | | +--------+ |  | +--------+ | |
                  | | | Value  | |  | | Value  | | |
                  | | +--------+ |  | +--------+ | |
                  | |            |  +------------+ |
                  | | +--------+ |                 |
                  | | | Value  | |                 |
                  | | +--------+ |                 |
                  | +------------+                 |
                  +--------------------------------+
```

   Each kind is identified by a Kind-ID, which is a code point assigned
   by IANA. 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. Note that a kind may be
   employed by multiple usages and new usages are encouraged to use
   previously defined kinds where possible. 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 a numeric 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.

In order to protect stored data from tampering, by other nodes, each
stored value is digitally signed by the node which created it. When
a value is retrieved, the digital signature can be verified to detect
tampering.

4.1.1. Storage Permissions

A major issue in peer-to-peer storage networks is minimizing the
burden of becoming a peer, and in particular minimizing the amount of
data which any peer is required to store for other nodes. RELOAD
addresses this issue by only allowing any given node to store data at
a small number of locations in the overlay, with those locations
being determined by the node's certificate. When a peer uses a Store
request to place data at a location authorized by its certificate, it
signs that data with the private key that corresponds to its
certificate. Then the peer responsible for storing the data is able
to verify that the peer issuing the request is authorized to make
that request. Each data kind defines the exact rules for determining
what certificate is appropriate.

The most natural rule is that a certificate authorizes a user to
store data keyed with their user name X. This rules 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 overlay. 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 undetectable
modify it.

The size requirements of the data being stored in the overlay are
variable. For instance, a SIP AoR and voicemail differ widely in the
storage size. RELOAD leaves it to the Usage and overlay

configuration to address the size imbalance of various kinds.

4.1.2. Usages

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 specifies several things:

o Registers Kind-ID code points for any kinds that the Usage
   defines.
o Defines the data structure for each of the kinds.
o Defines access control rules for each kinds.
o Defines how the Resource Name is formed that is hashed to form the
   Resource-ID where each kind is stored.
o Describes 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 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.

4.1.3. Replication

Replication in P2P overlays can be used to provide:

persistence: if the responsible peer crashes and/or if the storing
   peer leaves the overlay
security: to guard against DoS attacks by the responsible peer or
   routing attacks to that responsible peer
load balancing: to balance the load of queries for popular
   resources.

A variety of schemes are used in P2P overlays to achieve some of
these goals. Common techniques include replicating on neighbors of
the responsible peer, randomly locating replicas around the overlay,
or replicating along the path to the responsible peer.

The core RELOAD specification does not specify a particular

replication strategy. Instead, the first level of replication
strategies are determined by the overlay algorithm, which can base
the replication strategy on the its particular topology. For
example, Chord places replicas on successor peers, which will take
over responsibility should the responsible peer fail [Chord].

If additional replication is needed, for example if data persistence
is particularly important for a particular usage, then that usage may
specify additional replication, such as implementing random
replications by inserting a different well known constant into the
Resource Name used to store each replicated copy of the resource.
Such replication strategies can be added independent of the
underlying algorithm, and their usage can be determined based on the
needs of the particular usage.

4.2. Service Discovery

RELOAD does not currently define a generic service discovery
algorithm as part of the base protocol; although a TURN-specific
discovery mechanism is provided. A variety of service discovery
algorithm can be implemented as extensions to the base protocol, such
as ReDIR [opendht-sigcomm05].

4.3. Application Connectivity

There is no requirement that a RELOAD usage must use RELOAD's
primitives for establishing its own communication if it already
possesses its own means of establishing connections. For example,
one could design a RELOAD-based resource discovery protocol which
used HTTP to retrieve the actual data.

For more common situations, however, the overlay itself is used to
establish a connection rather than an external authority such as DNS,
RELOAD provides connectivity to applications using the same Attach
method as is used for the overlay maintenance. For example, if a
P2PSIP node wishes to establish a SIP dialog with another P2PSIP
node, it will use Attach to establish a direct connection with the
other node. This new connection is separate from the peer protocol
connection, it is a dedicated UDP or TCP flow used only for the SIP
dialog. Each usage specifies which types of connections can be
initiated using Attach.

5. Overlay Management Protocol

This section defines the basic protocols used to create, maintain,
and use the RELOAD overlay network. We start by defining how
messages are transmitted, received, and routed in an existing

overlay, then define the message structure, and then finally define
the messages used to join and maintain the overlay.

5.1. Message Routing

This section describes procedures used by nodes to route messages
through the overlay.

5.1.1. Request Origination

In order to originate a message to a given Node-ID or Resource-ID, a
node constructs an appropriate destination list. The simplest such
destination list is a single entry containing the peer or
Resource-ID. The resulting message will use the normal overlay
routing mechanisms to forward the message to that destination. The
node can also construct a more complicated destination list for
source routing.

Once the message is constructed, the node sends the message to some
adjacent peer. If the first entry on the destination list is
directly connected, then the message MUST be routed down that
connection. Otherwise, the topology plugin MUST be consulted to
determine the appropriate next hop.

Parallel searches for the resource are a common solution to improve
reliability in the face of churn or of subversive peers. Parallel
searches for usage-specified replicas are managed by the usage layer.
However, a single request can also be routed through multiple
adjacent peers, even when known to be sub-optimal, to improve
reliability [vulnerabilities-acsac04]. Such parallel searches MAY BE
specified by the topology plugin.

Because messages may be lost in transit through the overlay, RELOAD
incorporates an end-to-end reliability mechanism. When an
originating node transmits a request it MUST set a 3 second timer.
If a response has not been received when the timer fires, the request
is retransmitted with the same transaction identifier. The request
MAY be retransmitted up to 4 times (for a total of 5 messages).
After the timer for the fifth transmission fires, the message SHALL
be considered to have failed. Note that this retransmission
procedure is not followed by intermediate nodes. They follow the
hop-by-hop reliability procedure described in Section 5.5.2.

The above algorithm can result in multiple requests being delivered
to a node. Receiving nodes MUST generate semantically equivalent
responses to retransmissions of the same request (this can be
determined by transaction id) if the request is received within the
maximum request lifetime (15 seconds). For some requests (e.g.,

FETCH) this can be accomplished merely by processing the request
again. For other requests, (e.g., STORE) it may be necessary to
maintain state for the duration of the request lifetime.

5.1.2. 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 if local
policy can override this in which case the messages is silently
dropped.

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:

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

These cases are handled as discussed below.

5.1.2.1. Responsible ID

If the first entry on the destination list is a ID for which the node
is responsible, there are several sub-cases.
o If the entry is a Resource-ID, then it MUST be the only entry on
   the destination list. If there are other entries, the message
   MUST be silently dropped. Otherwise, the message is destined for
   this node and it passes it up to the upper layers.
o If the entry is a Node-ID which belongs to this node, then the
   message is destined for this node. If this is the only entry on
   the destination list, the message is destined for this node and is
   passed up to the upper layers. Otherwise the entry is removed
   from the destination list and the message is passed it to the
   Message Transport. If the message is a response and there is
   state for the transaction ID, the state is reinserted into the
   destination list first.
o If the entry is a Node-ID which is not equal to this node, then
   the node MUST drop the message silently unless the Node-ID
   corresponds to a node which is directly connected to this node
   (i.e., a client). In that case, it MUST forward the message to
   the destination node as described in the next section.

Note that this implies that in order to address a message to "the
peer that controls region X", a sender sends to Resource-ID X, not
Node-ID X.

5.1.2.2. Other 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 to which it is connected
and which is likely to be responsible for the first entry on the
destination list. 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. Otherwise, it consult the routing table to
forward the message.

Any intermediate peer which forwards a RELOAD message MUST arrange
that if it receives a response to that message the response can be
routed back through the set of nodes through which the request
passed. This may be arranged in one of two ways:

o The peer MAY add an entry to the via list in the forwarding header
   that will enable it to determine the correct node.
o The peer MAY keep per-transaction state which will allow it to
   determine the correct node.

As an example of the first strategy, if node D receives a message
from node C with via list (A, B), then D would 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 (C, B, A).

As an example of the second strategy, if node D receives a message
from node C with transaction ID X and via list (A, B), it could store
(X, C) in its state database and forward the message with the via
list unchanged. When D receives the response, it consults its state
database for transaction id X, determines that the request came from
C, and forwards the response to C.

Intermediate peer which modify the via list are not required to
simply add entries. The only requirement is that the peer be able to
reconstruct the correct destination list on the return route. RELOAD
provides explicit support for this functionality in the form of
private IDs, which can replace any number of via list entries. For
instance, in the above example, Node D might send E a via list
containing only the private ID (I). E would then use the destination
list (D, I) to send its return message. When D processes this
destination list, it would detect that I is a private ID, recover the

   via list (A, B, C), and reverse that to produce the correct
   destination list (C, B, A) before sending it to C. This feature is
   called List Compression. I MAY either be a compressed version of the
   original via list or an index into a state database containing the
   original via list.

   Note that if an intermediate peer exits the overlay, then on the
   return trip the message cannot be forwarded and will be dropped. The
   ordinary timeout and retransmission mechanisms provide stability over
   this type of failure.

5.1.2.3. Private ID

   If the first entry on the destination list is a private id (e.g., a
   compressed via list), the peer MUST that entry with the original via
   list that it replaced indexes and then re-examine the destination
   list to determine which case now applies.

5.1.3. Response Origination

   When a peer sends a response to a request, it MUST construct the
   destination list by reversing the order of the entries on the via
   list. This has the result that the response traverses the same peers
   as the request traversed, except in reverse order (symmetric
   routing). Note that this rule will need to be relaxed if other
   routing algorithms are supported.

5.2. Message Structure

   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, we
   just define these in a fixed position as adding a type and length for
   them is unnecessary and would simply increase bandwidth and
   introduces new potential for interoperability issues.

   Each message has three parts, concatenated as shown below:

```
       +-------------------------+
       |     Forwarding Header   |
       +-------------------------+
       |     Message Contents    |
       +-------------------------+
       |        Signature        |
       +-------------------------+
```

The contents of these parts are as follows:

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.

5.2.1. Presentation Language

The structures defined in this document are defined using a C-like
syntax based on the presentation language used to define TLS.
Advantages of this style include:

o It is easy to write and familiar enough looking that most readers
   can grasp it quickly.
o The ability to define nested structures allows a separation
   between high-level and low level message structures.
o It has a straightforward wire encoding that allows quick
   implementation, but the structures can be comprehended without
   knowing the encoding.
o The ability to mechanically (compile) encoders and decoders.

This presentation is to some extent a placeholder. We consider it an
open question what the final protocol definition method and encodings
use. We expect this to be a question for the WG to decide.

Several idiosyncrasies of this language are worth noting.

o All lengths are denoted in bytes, not objects.
o Variable length values are denoted like arrays with angle
   brackets.
o "select" is used to indicate variant structures.

For instance, "uint16 array<0..2^8-2>;" represents up to 254 bytes
but only up to 127 values of two bytes (16 bits) each..

5.2.1.1. **Common Definitions**

   The following definitions are used throughout RELOAD and so are
   defined here. They also provide a convenient introduction to how to
   read the presentation language.

   An enum represents an enumerated type. The values associated with
   each possibility are represented in parentheses and the maximum value
   is represented as a nameless value, for purposes of describing the
   width of the containing integral type. For instance, Boolean
   represents a true or false:

          enum { false (0), true(1), (255)} Boolean;

   A boolean value is either a 1 or a 0 and is represented as a single
   byte on the wire.

   The NodeId, shown below, represents a single Node-ID.

            typedef opaque   NodeId[16];

   A NodeId is a fixed-length 128-bit structure represented as a series
   of bytes, most significant byte first. Note: the use of "typedef"
   here is an extension to the TLS language, but its meaning should be
   relatively obvious.

   A ResourceId, shown below, represents a single Resource-ID.

            typedef opaque   ResourceId<0..2^8-1>;

   Like a NodeId, a Resource-ID is an opaque string of bytes, but unlike
   Node-IDs, Resource-IDs are variable length, up to 255 bytes (2048
   bits) in length. On the wire, each ResourceId is preceded by a
   single length byte (allowing lengths up to 255). Thus, the 3-byte
   value "Foo" would be encoded as: 03 46 4f 4f.

   A more complicated example is IpAddressPort, which represents a
   network address and can be used to carry either an IPv6 or IPv4
   address:

```
        enum {reserved_addr(0), ipv4_address (1), ipv6_address (2),
            (255)} AddressType;

        struct  {
          uint32                  addr;
          uint16                  port;
        } IPv4AddrPort;

        struct  {
          uint128                 addr;
          uint16                  port;
        } IPv6AddrPort;


        struct {
          AddressType             type;
          uint8                   length;

          select (type) {
            case ipv4_address:
                IPv4AddrPort        v4addr_port;

            case ipv6_address:
                IPv6AddrPort        v6addr_port;

            /* This structure can be extended */

         } IpAddressPort;
```

The first two fields in the structure are the same no matter what
kind of address is being represented:

type
   the type of address (v4 or v6).

length
   the length of the rest of the structure.

By having the type and the length appear at the beginning of the
structure regardless of the kind of address being represented, an
implementation which does not understand new address type X can still
parse the IpAddressPort field and then discard it if it is not
needed.

The rest of the IpAddressPort structure is either an IPv4AddrPort or
an IPv6AddrPort. Both of these simply consist of an address

represented as an integer and a 16-bit port. As an example, here is
the wire representation of the IPv4 address "192.0.2.1" with port
"6100".

```
            01              ; type    = IPv4
            06              ; length  = 6
            c0 00 02 01  ; address = 192.0.2.1
            17 d4           ; port     = 6100
```

5.2.2. Forwarding Header

The forwarding header is defined as a ForwardingHeader structure, as
shown below.

```
        struct {
          uint32            relo_token;
          uint32            overlay;
          uint8             ttl;
          uint8             reserved;
          uint16            fragment;
          uint8             version;
          uint24            length;
          uint64            transaction_id;
          uint16            flags;

          uint16            via_list_length;
          uint16            destination_list_length;
          uint16            route_log_length;
          uint16            options_length;
          Destination       via_list[via_list_length];
          Destination       destination_list
                               [destination_list_length];
          RouteLogEntry     route_log[route_log_length];
          ForwardingOptions options[options_length];
        } ForwardingHeader;
```

The contents of the structure are:

relo_token
   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. This field MUST contain the value 0xc2454c4f
   (the string 'RELO' with the high bit of the first byte set.).

overlay
   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. This is not a security critical
   function.

ttl
   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.

fragment
   This field is used to handle fragmentation. The high order two
   bits are used to indicate the fragmentation status: If the high
   bit (0x8000) is set, it indicates that the message is a fragment.
   If the next bit (0x4000) is set, it indicates that this is the
   last fragment.
   The remainder of the field is used to indicate the fragment
   offset. [[Open Issue: This is conceptually clear, but the
   details are still lacking. Need to define 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
   The version of the RELOAD protocol being used. This document
   describes version 0.1, with a value of 0x01.

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

transaction_id
   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.

flags
   The flags word contains control flags. Which are ORed together.
   There is two currently defined flags: ROUTE-LOG (0x1) and
   RESPONSE-ROUTE-LOG (0x2). These flags indicate that the route log
   should be included (see Section 5.2.2.2.).

via_list_length
   The length of the via list in bytes. Note that in this field and
   the following two length fields we depart from the usual variable-
   length convention of having the length immediately precede the
   value in order to make it easier for hardware decoding engines to
   quickly determine the length of the header.

destination_list_length
   The length of the destination list in bytes.

route_log_length
   The length of the route log in bytes.

options_length
   The length of the header options in bytes.

via_list
   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.

destination_list
   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.

route_log
   Contains a series of route log entries. See Section 5.2.2.2.

options
   Contains a series of ForwardingOptions entries. See
   Section 5.2.2.3.

5.2.2.1. Destination and Via Lists

   The destination list and via lists are sequences of Destination
   values:

```
     enum {reserved(0), peer(1), resource(2), compressed(3), (255) }
         DestinationType;


     select (destination_type) {
       case peer:
          NodeId              node_id;

       case resource:
          ResourceId          resource_id;

       case compressed:
          opaque              compressed_id<0..2^8-1>;

       /* This structure may be extended with new types */

     } DestinationData;


     struct {
       DestinationType       type;
       uint8                 length;
       DestinationData       destination_data;
     } Destination;
```

This is a TLV structure with the following contents:

type
   The type of the DestinationData PDU. This may be one of "peer",
   "resource", or "compressed".

length
   The length of the destination_data.

destination_value
   The destination value itself, which is an encoded DestinationData
   structure, depending on the value of "type".

Note: This structure encodes a type, length, value. The length
   field specifies the length of the DestinationData values, which
   allows the addition of new DestinationTypes. This allows an
   implementation which does not understand a given DestinationType
   to skip over it.

A DestinationData can be one of three types:

peer
   A Node-ID.

compressed
   A compressed list of Node-IDs and/or resources. Because this
   value was compressed by one of the peers, it is only meaningful to
   that peer and cannot be decoded by other peers. Thus, it is
   represented as an opaque string.

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.

5.2.2.2. Route Logging

   The route logging feature provides diagnostic information about the
   path taken by the message 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 or reject the request.
   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, the route
   log entries in the request MUST be copied to the response or the
   request rejected. If, and only if, the ROUTE-LOG-RESPONSE flag is
   set in a request, the ROUTE-LOG flag MUST be set in the response.

   Note that use of the ROUTE-LOG-RESPONSE flag 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.

   The route log is defined as follows:

```
      enum { (255) } RouteLogExtensionType;

      struct {
        RouteLogExtensionType     type;
        uint16                    length;

        select (type){
          /* Extension values go here */
        } extension;
      } RouteLogExtension;

      enum {
         reserved(0),
         tcp_tls(1),
         udp_dtls(2),
         (255)
      } OverlayLink;

      struct {
        opaque                version<0..2^8-1>;    /* A string */
        OverlayLink           linkProtocol;             /* TCP or UDP */
        NodeId                id;
        uint32                uptime;
        IpAddressPort         address;
        opaque                certificate<0..2^16-1>;
        RouteLogExtension     extensions<0..2^16-1>;
      } RouteLogEntry;

      struct {
         RouteLogEntry        entries<0..2^16-1>;
      } RouteLog;
```

The route log consists of an arbitrary number of RouteLogEntry
values, each representing one node through which the message has
passed.

Each RouteLogEntry consists of the following values:

version
    A textual representation of the software version

linkProtocol

      The Overlay Link Layer protocol, currently either "tcp_tls" or
      "udp_dtls".

   id
      The Node-ID of the peer.

   uptime
      The uptime of the peer in seconds.

   address
      The address and port of the peer.

   certificate
      The peer's certificate. Note that this may be omitted by setting
      the length to zero.

   extensions
      Extensions, if any.

   Extensions are defined using a RouteLogExtension structure. New
   extensions are defined by defining a new code point for
   RouteLogExtensionType and adding a new arm to the RouteLogExtension
   structure. The contents of that structure are:

   type
      The type of the extension.

   length
      The length of the rest of the structure.

   extension
      The extension value.

5.2.2.3. Forwarding Options

   The Forwarding header can be extended with forwarding header options,
   which are a series of ForwardingOptions structures:

```
    enum { (255) } ForwardingOptionsType;

    struct {
      ForwardingOptionsType      type;
      uint8                      flags;
      uint16                     length;
      select (type) {
        /* Option values go here */
      } option;
```

```
      } ForwardingOption;
```

Each ForwardingOption consists of the following values:

type
    The type of the option.

length
    The length of the rest of the structure.

flags
    Three flags are defined FORWARD_CRITICAL(0x01),
    DESTINATION_CRITICAL(0x02), and RESPONSE_COPY(0x04). These flags
    MUST NOT be set in a response. If the FORWARD_CRITICAL flag is
    set, any node that would forward the message but does not
    understand this options MUST reject the request with an 757 error
    response. If the DESTINATION_CRITICAL flag is set, any node
    generates a response to the message but does not understand the
    forwarding option MUST reject the request with an 757 error
    response. If the RESPONSE_COPY flag is set, any node generating a
    response MUST copy the option from the request to the response and
    clear the RESPONSE_COPY, FORWARD_CRITICAL and DESTINATION_CRITICAL
    flags.

option
    The option value.

5.2.3. Message Contents Format

The second major part of a RELOAD message is the contents part, which
is defined by MessageContents:

```
        struct {
          MessageCode           message_code;
          opaque                payload<0..2^24-1>;
        } MessageContents;
```

The contents of this structure are as follows:

message_code
    This indicates the message that is being sent. The code space 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, "probe_request" (the
   Probe request) has value 1 and "probe_answer" (the Probe
   response) has value 2

0xffff Error

message_body
   The message body itself, represented as a variable-length string
   of bytes. The bytes themselves are dependent on the code value.
   See the sections describing the various RELOAD methods (Join,
   Update, Attach, Store, Fetch, etc.) for the definitions of the
   payload contents.

5.2.3.1. Response Codes and Response Errors

A peer processing a request returns its status in the message_code
field. 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.

When the message code is 0xffff, the payload MUST be an
ErrorResponse.

```
      public struct {
        uint16             error_code;
        opaque             error_info<0..2^16-1>;
      } ErrorResponse;
```

The contents of this structure are as follows:

error_code
   A numeric error code indicating the error that occurred.

error_info
   A free form text string indicating the reason for the response for
   diagnostic purposes.

The following error code values are defined. The numeric values for
these are defined in Section 14.5.

Error_Unauthorized: The requesting peer needs to sign and provide a
   certificate. [[TODO: The semantics here don't seem quite
   right.]]

Error_Forbidden: The requesting peer does not have permission to
   make this request.

Error_Not_Found: The resource or peer cannot be found or does not
   exist.

Error_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.

Error_Precondition_Failed: A request can't be completed because some
   precondition was incorrect. For instance, the wrong generation
   counter was provided

Error_Incompatible_with_Overlay: A peer receiving the request is
   using a different overlay, overlay algorithm, or hash algorithm.

Error_Unsupported_Forwarding_Option: A peer receiving the request
   with a forwarding options flagged as critical but the peer does
   not support this option. See section Section 5.2.2.3.

5.2.4. Signature

The third part of a RELOAD message is the signature, represented by a
Signature structure. The message signature is computed over the
payload and parts of forwarding header. The payload, in case of a
Store, may contain an additional signature computed over a StoreReq
structure. All signatures are formatted using the Signature element.
This element is also used in other contexts where signatures are
needed. The input structure to the signature computation varies
depending on the data element being signed.

```
      enum {reserved(0), signer_identity_peer (1),
            signer_identity_name (2), signer_identity_certificate (3),
            (255)} SignerIdentityType;

      select (identity_type) {
        case signer_identity_peer:
          NodeId              id;

        case signer_identity_name:
          opaque              name<0..2^16-1>;

        case signer_identity_certificate:
          opaque              certificate<0..2^16-1>;

        /* This structure may be extended with new types */
      } SignerIdentityValue;


      struct {
        SignerIdentityType    identity_type;
        uint16                length;
        SignerIdentityValue   identity[SignerIdentity.length];
      } SignerIdentity;


      struct  {
         SignatureAndHashAlgorithm    algorithm;
         SignerIdentity               identity;
         opaque                       signature_value<0..2^16-1>;
      } Signature;
```

The signature construct contains the following values:

algorithm
   The signature algorithm in use. The algorithm definitions are
   found in the IANA TLS SignatureAlgorithm Registry.

identity
   The identity or certificate used to form the signature

signature_value

The value of the signature

A number of possible identity formats are permitted. The current
possibilities are: a Node-ID, a user name, and a certificate.

For signatures over messages the input to the signature is computed
over:

overlay + transaction_id + MessageContents + SignerIdentity

Where overlay and transaction_id come from the forwarding header and
+ indicates concatenation.

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

The input to signatures over data values is different, and is
described in Section 6.1.

5.3. Overlay Topology

As discussed in previous sections, RELOAD does not itself implement
any overlay topology. Rather, it relies on Topology Plugins, which
allow a variety of overlay algorithms to be used while maintaining
the same RELOAD core. This section describes the requirements for
new topology plugins and the methods that RELOAD provides for overlay
topology maintenance.

5.3.1. Topology Plugin Requirements

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

o Joining procedures, including the contents of the Join message.
o 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.
o Exit procedures, including the contents of the Leave message.
o The length of the Resource-IDs and Node-IDs. For DHTs, the hash
   algorithm to compute the hash of an identifier.
o The procedures that peers use to route messages.
o The replication strategy used to ensure data redundancy.

5.3.2. Methods and types for use by topology plugins

This section describes the methods that topology plugins use to join,
leave, and maintain the overlay.

5.3.2.1. **Join**

   A new peer (but which already has credentials) uses the JoinReq
   message to join the overlay. The JoinReq is sent to the responsible
   peer depending on the routing mechanism described in the topology
   plugin. This notifies the responsible peer that the new peer is
   taking over some of the overlay and it needs to synchronize its
   state.

```
        struct {
          NodeId                    joining_peer_id;
          opaque                    overlay_specific_data<0..2^16-1>;
        } JoinReq;
```

   The minimal JoinReq contains only the Node-ID which the sending peer
   wishes to assume. Overlay algorithms MAY specify other data to
   appear in this request.

   If the request succeeds, the responding peer responds with a JoinAns
   message, as defined below:

```
        struct {
          opaque                    overlay_specific_data<0..2^16-1>;
        } JoinAns;
```

   If the request succeeds, the responding peer 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, overlay algorithms MAY define data to appear in the
   response payload that provides additional info.

   In general, nodes which cannot form connections SHOULD report an
   error. However, implementations MUST provide some mechanism whereby
   nodes can determine they are potentially the first node and take
   responsibility for the overlay. This specification does not mandate
   any particular mechanism, but a configuration flag or setting seems
   appropriate.

5.3.2.2. **Leave**

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

```
        public struct {
          NodeId                    leaving_peer_id;
```

```
        opaque                  overlay_specific_data<0..2^16-1>;
   } LeaveReq;
```

   LeaveReq contains only the Node-ID of the leaving peer. Overlay
   algorithms MAY specify other data to appear in this request.

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

5.3.2.3. Update

   Update is the primary overlay-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 (its routing state) and it is up to
   the recipient to take whatever actions are appropriate to deal with
   the state change.

   The contents of the UpdateReq message are completely overlay-
   specific. The UpdateAns response is expected to be either success or
   an error.

5.3.2.4. 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 RouteQuery for a destination X requests the Node-ID where
   the receiving peer would next route to get to X. A RouteQuery can
   also request that the receiving peer initiate an Update request to
   transfer his routing table.

   One important use of the RouteQuery request is to support iterative
   routing. The sender selects one of the peers in its routing table
   and sends it a RouteQuery message with the destination_object set to
   the Node-ID or Resource-ID it wishes to route to. The receiving peer
   responds with information about the peers to which the request would
   be routed. The sending peer MAY then Attaches to that peer(s), and
   repeats the RouteQuery. Eventually, the sender gets a response from
   a peer that is closest to the identifier in the destination_object as
   determined by the topology plugin. At that point, the sender can
   send messages directly to that peer.

5.3.2.4.1. Request Definition

   A RouteQueryReq 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.

```
      struct {
        Boolean                 send_update;
        Destination             destination;
        opaque                  overlay_specific_data<0..2^16-1>;
      } RouteQueryReq;
```

The contents of the RouteQueryReq message are as follows:

send_update
   A single byte. This may be set to "true" to indicate that the
   requester wishes the responder to initiate an Update request
   immediately. Otherwise, this value MUST be set to "false".

destination
   The destination which the requester is interested in. This may be
   any valid destination object, including a Node-ID, compressed ids,
   or Resource-ID.

overlay_specific_data
   Other data as appropriate for the overlay.

5.3.2.4.2. Response Definition

A response to a successful RouteQueryReq request is a RouteQueryAns
message. This is completely overlay specific.

5.3.2.5. Probe

Probe provides a number of primitive "exploration" services: (1) it
allows node to determine which resources another node is responsible
for (2) it allows some discovery services in multicast settings. A
probe can be addressed to a specific Node-ID, or the peer controlling
a given location (by using a resource ID). In either case, the
target Node-IDs respond with a simple response containing some status
information.

5.3.2.5.1. Request Definition

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

```
        enum { responsible_set(1), num_resources(2), (255)}
            ProbeInformationType;

        struct {
          ProbeInformationType     requested_info<0..2^8-1>;
        } ProbeReq
```

The two currently defined values for ProbeInformation are:

responsible_set
    indicates that the peer should Respond with the fraction of the
    overlay for which the responding peer is responsible.

num_resources
    indicates that the peer should Respond with the number of
    resources currently being stored by the peer.

5.3.2.5.2. Response Definition

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


```
        struct {
          ProbeInformationType     type;

          select (type) {
            case responsible_set:
              uint32              responsible_ppb;

            case num_resources:
              uint32              num_resources;

            /* This type may be extended */

          };
        } ProbeInformation;

        struct {
          ProbeInformation         probe_info<0..2^16-1>;
        } ProbeAns;
```


A ProbeAns message contains the following elements:

   probe_info
      A sequence of ProbeInformation structures, as shown below.

   Each of the current possible Probe information types is a 32-bit
   unsigned integer. For type "responsible_ppb", it is the fraction of
   the overlay for which the peer is responsible in parts per billion.
   For type "num_resources", it is the number of resources the peer is
   storing.

   The responding peer SHOULD include any values that the requesting
   peer requested and that it recognizes. They SHOULD be returned in
   the requested order. Any other values MUST NOT be returned.

5.4. Forwarding and Link Management Layer

   Each node maintains connections to a set of other nodes defined by
   the topology plugin. This section defines the methods RELOAD uses to
   form and maintain connections between nodes in the overlay. Three
   methods are defined:

   Attach: used to form connections between nodes. When node A wants
      to connect to node B, it sends an Attach message to node B through
      the overlay. The Attach contains A's ICE parameters. B responds
      with its ICE parameters and the two nodes perform ICE to form
      connection.
   AttachLite: like attach, it is used to form connections between
      nodes but instead of using full ICE, it only uses a subset known
      as ICE-Lite.
   Ping: is a simple request/response which is used to verify
      connectivity of the target peer.

5.4.1. Attach

   A node sends an Attach 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 Attach and its response are described in
   Section 5.4.1.

   An Attach in and of itself does not result in updating the routing
   table of either node. That function is performed by Updates. If
   node A has Attached to node B, but not received any Updates from 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. The process of Attaching is separate from the process of
   becoming a peer (using Update) to prevent half-open states where a
   node has started to form connections but is not really ready to act

   as a peer.

5.4.1.1. Request Definition

   An AttachReq message contains the requesting peer's ICE connection
   parameters formatted into a binary structure.

```
        typedef opaque          IceCandidate<0..2^16-1>;

        struct  {
          opaque                ufrag<0..2^8-1>;
          opaque                password<0..2^8-1>;
          uint16                application;
          opaque                role<0..2^8-1>;
          IceCandidate          candidates<0..2^16-1>;
        } AttachReqAns;
```

   The values contained in AttachReq and AttachAns 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.

   role
      An active/passive/actpass attribute from RFC 4145 [RFC4145].

   candidates
      One or more ICE candidate values in the string representation used
      in ordinary ICE. [[OPEN ISSUE: This is convenient for stacks,
      but unaesthetic.]] 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 from ICE.

   These values should be generated using the procedures described in
   Section 5.4.1.3.

5.4.1.2. Response Definition

   If a peer receives an Attach request, it SHOULD follow the process
   the request and generate its own response with a AttachReqAns. It
   should then begin ICE checks. When a peer receives an Attach
   response, it SHOULD parse the response and begin its own ICE checks.

5.4.1.3. Using ICE With RELOAD

   This section describes the profile of ICE that is used with RELOAD.
   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.

   In ICE as defined by [I-D.ietf-mmusic-ice], SDP is used to carry the
   ICE parameters. In RELOAD, this function is performed by a binary
   encoding in the Attach method. This encoding is more restricted than
   the SDP encoding because the RELOAD environment is simpler:

   o Only a single media stream is supported.
   o 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.
   o 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.

   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.

5.4.1.4. 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 [RFC5389]. Accordingly, any
   peer a node knows will be willing to be a STUN server -- 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.

5.4.1.5. 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 overlay, as described in
Section 5.4.1.4 and Section 8.

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.

5.4.1.6. Encoding the Attach Message

Section 4.3 of ICE describes procedures for encoding the SDP for
conveying RELOAD or SIP ICE candidates. 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 Attach request or
its response. Similarly, the username fragment and password are
carried in the Attach message or its response. Section 5.4.1
describes the detailed attribute encoding for Attach. The Attach
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 Attach request and its response also contain a application
attribute, with a value of SIP or RELOAD, which indicates what
protocol is to be run over the connection. The RELOAD Attach request
MUST only be utilized to set up connections for application protocols
that can be multiplexed with STUN.

Since the Attach 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 Attach response is considered
a valid answer for the purposes of following the ICE specification.

5.4.1.7. **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.

5.4.1.8. **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 Attach request) will always be controlling, and
the answerer (the entity sending the Attach 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.

5.4.1.9. **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.

5.4.1.10. **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:

   o The controlling agent MUST NOT attempt to send an updated offer
      once the state of its single media stream reaches Completed.
   o 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.

5.4.1.11. 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.

5.4.1.12. Media Keepalives

   STUN MUST be utilized for the keepalives described in Section 10 of
   ICE. [[TODO - this does not define what happens for TCP]]

5.4.1.13. 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 node which sent the Attach
   request MUST be the TLS server. The other node MUST be the TLS
   client. The nodes MUST verify that the certificate presented in the
   handshake matches the identity of the other peer as found in the
   Attach message. 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.

5.4.1.14. 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.

5.4.2. **AttachLite**

   An alternative to using the full ICE supported by the Attach request
   is to use ICE-Lite with the AttachLite request. This will not work
   in all of the scenarios where ICE would work, but in some cases,
   particularly those with no NATs or firewalls, it will work.
   Configuration for the overlay indicates if this can be used or not.

   OPEN ISSUE: We originally envisioned adding support for ICE-Lite
   directly to the regular Attach method. However, we found that both
   the parameters and processing were completely different, resulting in
   almost no overlap between the two methods. Therefore we chose to
   separate this out for overlays where the complexities of ICE are not
   needed. Note that it is still possible for a node with a public
   unfiltered address intending to interoperate to implement Attach
   without the candidate gathering phases of ICE and achieve essentially
   the same result. If simpler behavior or a better encoding of ICE-
   Lite in Attach is developed, such an approach would be preferable.

5.4.2.1. **Request Definition**

   An AttachLiteReq message contains the requesting peer's ICE-Lite
   connection parameters formatted into a binary structure. When using
   the AttachLite request, both sides act as ICE-Lite hosts.

```
      struct {
           IpAddressPort addr_port;
           Transport         transport;
           uint32             priority;
      } IceLiteCandidate;

      struct  {
        uint16                        application;
        IceLiteCandidate      candidates<0..2^16-1>;
      } AttachLiteReqs;
```

   The values contained in AttachLiteReq are:

   application
      A 16-bit port number used in the same was as in the Attach
      request. This port number represents the IANA registered port of
      the protocol that is going to be sent on this connection.

candidates
   One or more ICE candidate values. Each one contains an IP address
   and family, transport protocol, and port to connect to as well as
   a priority.

These values should be generated using the procedures described in
Section 5.4.1.3.

5.4.2.2. Attach-Lite Connectivity Checks

STUN is not used for connectivity checks when doing ICE-Lite, instead
the DTLS or TLS handshake forms the connectivity check. The host
that received the AttachLiteReq MUST initiate TLS or DTLS connections
to candidates provided in the request. When a connection forms, the
node MUST check the certificate is for the node that send
AttachLiteReq and if is not, MUST close the connection.

Since TLS provides the connectivity check, there is no need for the
RFC 4571 [RFC4571] style framing shim for STUN when using TLS and
this is not used for this protocol.

5.4.2.3. Implementation Notes for Attach-Lite

This is a non normative section to help implementors.

At times ICE can seem a bit daunting to gets one head around. For a
simple IPv4 only peer, a simple implementation of Attach-Lite could
be done be doing the following:
o When sending an AttachLiteReq, form one with a candidate with a
   priority value of $(2^{24})*(126)+(2^8)*(65535)+(2^0)*(256-1)$ that
   specifies the UDP port being listened to and another one with the
   TCP port.
o When receiving an AttachLiteReq, try to form a connection to each
   candidate in the request. Check the certificate receive in the
   TLS handshake has the correct Node-ID as the node that send the
   AttchLiteReq. If multiple connection succeed, close all but the
   one with highest priority.
o Do normal TLS and DTLS with no need for any special framing or
   STUN processing.

5.4.3. Ping

Ping is used to test connectivity along a path. A ping can be
addressed to a specific Node-ID, the peer controlling a given
location (by using a resource ID), or to the broadcast Node-ID (all
1s).

5.4.3.1. Request Definition

```
        struct {
        } PingReq
```

5.4.3.2. Response Definition

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

```
        struct {
          uint64                 response_id;
        } PingAns;
```

A PingAns 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.

5.5. Overlay Link Layer

RELOAD can use multiple Overlay Link protocols to send its messages.
Because ICE is used to establish connections (see Section 5.4.1.3),
RELOAD nodes are able to detect which Overlay Link protocols are
offered by other nodes and establish connections between each other.
Any link protocol needs to be able to establish a secure,
authenticated connection, and provide data origin authentication and
message integrity for individual data elements. RELOAD currently
supports two Overlay Link protocols:

o   TLS [TODO REF] over TCP
o   DTLS [RFC4347] over UDP

Note that although UDP does not properly have "connections", both TLS
and DTLS have a handshake which establishes a stateful association, a
similar stateful construct, and we simply refer to these as
"connections" for the purposes of this document.

5.5.1. Future Support for HIP

The P2PSIP Working Group has expressed interest in supporting a HIP-
based link protocol. Such support would require specifying such

   details as:

   o How to issue certificates which provided identities meaningful to
      the HIP base exchange. We anticipate that this would require a
      mapping between ORCHIDs and NodeIds.
   o How to carry the HIP I1 and I2 messages. We anticipate that this
      would require defining a HIP Tunnel usage.
   o How to carry RELOAD messages over HIP.

   We leave this work as a topic for another draft.

5.5.2. Reliability for Unreliable Links

   When RELOAD is carried over DTLS or another unreliable link protocol,
   it needs to be used with a reliability and congestion 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.

   In order to support unreliable links, each message is wrapped in a
   very simple framing layer (FramedMessage) which is only used for each
   hop. This layer contains a sequence number which can then be used
   for ACKs.

5.5.2.1. Framed Message Format

   The definition of FramedMessage is:

        enum {data (128), ack (129), (255)} FramedMessageType;

        struct {
          FramedMessageType   type;

          select (type) {
            case data:
              uint24   sequence;
              opaque   message<0..2^24-1>;

            case ack:
              uint24   ack_sequence;
              uint32   received;
        };

```
      } FramedMessage;
```

The type field of the PDU is set to indicate whether the message is
data or an acknowledgement. Note that these values have been set to
force the first bit to be high, thus allowing easy demultiplexing
with STUN. All FramedMessageType values must be > 128.

If the message is of type "data", then the remainder of the PDU is as
follows:

sequence
    the sequence number

message
    the original message that is being transmitted.

Each connection has it own sequence number. Initially the value is
zero and it increments by exactly one for each message sent over that
connection.

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 association in order to generate
the appropriate ack.

If the PDU is of type "ack", the contents are as follows:

ack_sequence
    The sequence number of the message being acknowledged.

received
    A bitmask indicating whether or not each of the previous 32
    packets has been received before the sequence number in
    ack_sequence. 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.

5.5.2.2. 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. Senders MUST implement a retransmission and congestion
control scheme no more aggressive then TFRC[RFC5348]. One way to do
that is for senders to implement TFRC-SP [RFC4828] and use the
received bitmask to allow the sender to compute packet loss event
rates.

5.5.3. Fragmentation and Reassembly

   In order to allow transmission 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 mitigate denial of
   service attacks, receivers SHOULD time out incomplete fragments.
   [[TODO: Describe algorithm]]

6. Data Storage Protocol

   RELOAD provides a set of generic mechanisms for storing and
   retrieving data in the Overlay Instance. These mechanisms can be
   used for new applications simply by defining new code points and a
   small set of rules. No new protocol mechanisms are required.

   The basic unit of stored data is a single StoredData structure:

```
        struct {
          uint32                length;
          uint64                storage_time;
          uint32                lifetime;
          StoredDataValue       value;
          Signature             signature;
        } StoredData;
```

   The contents of this structure are as follows:

length
    The length of the rest of the structure in octets.

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 already stored at this
    location MUST generate a Error_Data_Too_Old 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.

value
    The data value itself, as described in Section 6.2.

signature
    A signature over the data value. Section 6.1 describes the
    signature computation. The element is formatted as described in
    Section 5.2.4

    Each Resource-ID specifies a single location in the Overlay Instance.
    However, each location may contain multiple StoredData values
    distinguished by Kind-ID. The definition of a kind describes both
    the data values which may be stored and the data model of the data.
    Some data models allow multiple values to be stored under the same
    Kind-ID. Section Section 6.2 describes the available data models.
    Thus, for instance, a given Resource-ID might contain a single-value
    element stored under Kind-ID X and an array containing multiple
    values stored under Kind-ID Y.

6.1. Data Signature Computation

    Each StoredData 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 StoredData
    structure. The input to the signature algorithm is:

        resource_id + kind + StoredData

    Where these values are:

resource
    The resource ID where this data is stored.

kind
    The Kind-ID for this data.

StoredData
    The contents of the stored data value, as described in the
    previous sections.

[OPEN ISSUE: Should we include the identity in the string that forms
the input to the signature algorithm?.]

Once the signature has been computed, the signature is represented
using a signature element, as described in Section 5.2.4.

6.2. Data Models

The protocol currently defines the following data models:

o single value
o array
o dictionary

These are represented with the StoredDataValue structure:

```
      enum { reserved(0), single_value(1), array(2),
             dictionary(3), (255)} DataModel;

      struct {
        Boolean                   exists;
        opaque                    value<0..2^32-1>;
      } DataValue;


      struct {
        DataModel                 model;

        select (model) {
          case single_value:
            DataValue             single_value_entry;

          case array:
            ArrayEntry            array_entry;

          case dictionary:
            DictionaryEntry       dictionary_entry;


          /* This structure may be extended */
        } ;
      } StoredDataValue;
```

   We now discuss the properties of each data model in turn:

6.2.1. Single Value

   A single-value element is a simple, opaque sequence of bytes. There
   may be only one single-value element for each Resource-ID, Kind-ID
   pair.

   A single value element is represented as a DataValue, which contains
   the following two elements:

   exists
      This value indicates whether the value exists at all. If it is
      set to False, it means that no value is present. If it is True,
      that means that a value is present. This gives the protocol a
      mechanism for indicating nonexistence as opposed to emptiness.

   value
      The stored data.

6.2.2. Array

   An array is a set of opaque values addressed by an integer index.
   Arrays are zero based. Note that arrays can be sparse. For
   instance, 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 values with "exists" set to
   False.

   A array element is represented as an ArrayEntry:

           struct {
             uint32   index;
             DataValue   value;
           } ArrayEntry;

   The contents of this structure are:

   index
      The index of the data element in the array.

   value
      The stored data.

6.2.3. Dictionary

   A dictionary is a set of opaque values indexed by an opaque key with
   one value for each key. A single dictionary entry is represented as
   follows

   A dictionary element is represented as a DictionaryEntry:

           typedef opaque   DictionaryKey<0..2^16-1>;

           struct {
             DictionaryKey   key;
             DataValue   value;
           } DictionaryEntry;

The contents of this structure are:

key
    The dictionary key for this value.

value
    The stored data.

6.3. Data Storage Methods

RELOAD provides several methods for storing and retrieving data:

o Store values in the overlay
o Fetch values from the overlay
o Remove values from the overlay
o Find the values stored at an individual peer

These methods are each described in the following sections.

6.3.1. Store

The Store method is used to store data in the overlay. The format of
the Store request depends on the data model which is determined by
the kind.

6.3.1.1. Request Definition

A StoreReq message is a sequence of StoreKindData values, 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.

The store request is defined by the StoreReq structure:

```
    struct {
        KindId                 kind;
        DataModel              data_model;
        uint64                 generation_counter;
        StoredData             values<0..2^32-1>;
    } StoreKindData;

    struct {
        ResourceId             resource;
        uint8                  replica_number;
        StoreKindData          kind_data<0..2^32-1>;
    } StoreReq;
```

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

resource
   The resource to store at.

replica_number
   The number of this replica. When a storing peer saves replicas to
   other peers each peer is assigned a replica number starting from 1
   and sent in the Store message. This field is set to 0 when a node
   is storing its own data. This allows peers to distinguish replica
   writes from original writes.

kind_data
   A series of elements, one for each kind of data to be stored.

If the replica number is zero, then the peer MUST check that it is
responsible for the resource and if not reject the request. If the
replica number is nonzero, then the peer MUST check that it expects
to be a replica for the resource and if not reject the request.

Each StoreKindData 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. The kind defines what this has to be
   so this is redundant in the case where the software interpreting
   the messages understands the kind.

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.

The peer MUST perform the following checks:

o The kind_id is known and supported.

o The data_model matches the kind_id.

o The signatures over each individual data element (if any) are
   valid.

o Each element is signed by a credential which is authorized to
   write this kind at this Resource-ID

o For original (non-replica) stores, the peer MUST check that if the
   generation-counter is non-zero, it equals 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.

o 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. For non-replica stores, 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
StoreKindData, 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. Accordingly, all stored data values must
have a generation counter of 1 or greater. 0 is used by other nodes
to indicate that they are indifferent to the generation counter's
current value. For replica Stores, the peer MUST set the generation
counter to match the generation_counter in the message. Replica
Stores MUST NOT use a generation counter of 0.

The properties of stores for each data model are as follows:

Single-value:
   A store of a new single-value element creates the element if it
   does not exist and overwrites any existing value with the new
   value.

Array:
   A store of an array entry replaces (or inserts) the given value at
   the location specified by the index. Because arrays are sparse, a
   store past the end of the array extends it with nonexistent values
   (exists=False) as required. A store at index 0xffffffff places
   the new value at the end of the array regardless of the length of
   the array. The resulting StoredData has the correct index value
   when it is subsequently fetched.

Dictionary:

A store of a dictionary entry replaces (or inserts) the given
value at the location specified by the dictionary key.

The following figure shows the relationship between these structures
for an example store which stores the following values at resource
"1234"

o The value "abc" in the single value slot for kind X
o The value "foo" at index 0 in the array for kind Y
o The value "bar" at index 1 in the array for kind Y

```
                            Store
                        resource=1234
                          /        \
                         /          \
               StoreKindData        StoreKindData
                   kind=X               kind=Y
              model=Single-Value     model=Array
                     |                     /\
                     |                    /  \
                 StoredData              /    \
                     |                  /      \
                     |             StoredData  StoredData
              StoredDataValue          |           |
                value="abc"            |           |
                                       |           |
                              StoredDataValue  StoredDataValue
                                  index=0         index=1
                                value="foo"     value="bar"
```
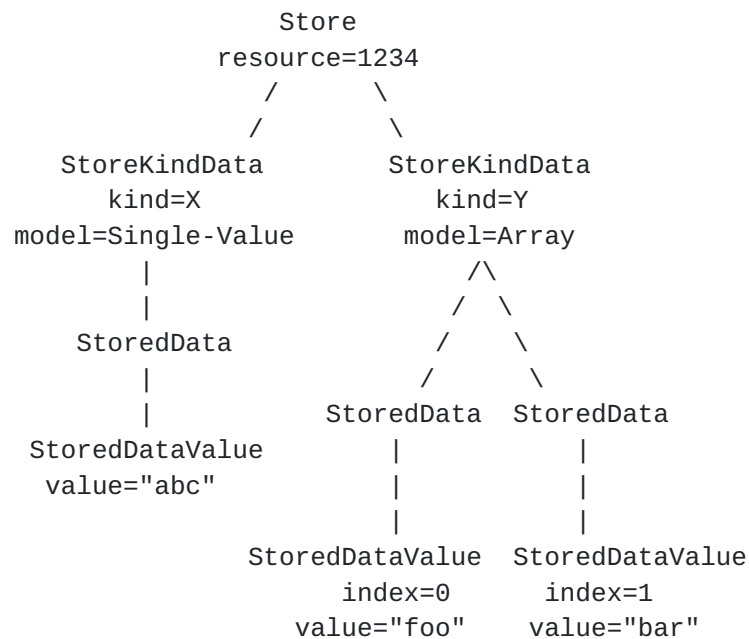
6.3.1.2. **Response Definition**

In response to a successful Store request the peer MUST return a
StoreAns message containing a series of StoreKindResponse 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.

```
      struct {
        KindId                  kind;
        uint64                  generation_counter;
        NodeId                  replicas<0..2^16-1>;
      } StoreKindResponse;


      struct {
        StoreKindResponse       kind_responses<0..2^16-1>;
```

```
     } StoreAns;
```

The contents of each StoreKindResponse 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 overlays 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 by
   doing its own Fetch.

The response itself is just StoreKindResponse values packed end-to-
end.

If any of the generation counters in the request precede the
corresponding stored generation counter, then the peer MUST fail the
entire request and respond with a Error_Data_Too_Old error. The
error_info in the ErrorResponse MUST be a StoreAns response
containing the correct generation counter for each kind and empty
replicas lists. [[TODO need to fix this up in better way]]

If the data being stored is too large for the allowed limit by the
given usage, then the peer MUST fail the request and generate an
Error_Data_Too_Larg error.

6.3.2. Fetch

The Fetch request retrieves one or more data elements stored at a
given Resource-ID. A single Fetch request can retrieve multiple
different kinds.

[6.3.2.1](#). Request Definition

```
      struct {
        int32            first;
        int32            last;
      } ArrayRange;

      struct {
        KindId                  kind;
        DataModel               model;
        uint64                  generation;
        uint16                  length;

        select (model) {
          case single_value: ;     /* Empty */

          case array:
              ArrayRange        indices<0..2^16-1>;

          case dictionary:
              DictionaryKey     keys<0..2^16-1>;

          /* This structure may be extended */

        } model_specifier;
      } StoredDataSpecifier;

      struct {
        ResourceId              resource;
        StoredDataSpecifier     specifiers<0..2^16-1>;
      } FetchReq;
```

The contents of the Fetch requests are as follows:

resource
   The resource ID to fetch from.

specifiers
   A sequence of StoredDataSpecifier values, each specifying some of
   the data values to retrieve.

Each StoredDataSpecifier specifies a single kind of data to retrieve
and (if appropriate) the subset of values that are to be retrieved.
The contents of the StoredDataSpecifier structure are as follows:

kind
   The Kind-ID of the data being fetched. Implementations SHOULD
   reject requests corresponding to unknown kinds unless specifically
   configured otherwise.

model
   The data model of the data. This must be checked against the
   Kind-ID.

generation
   The last generation counter that the requesting peer saw. This
   may be used to avoid unnecessary fetches or it may be set to zero.

length
   The length of the rest of the structure, thus allowing
   extensibility.

model_specifier
   A reference to the data value being requested within the data
   model specified for the kind. For instance, if the data model is
   "array", it might specify some subset of the values.

The model_specifier is as follows:

o If the data is of data model single value, the specifier is empty.
o If the data is of data model array, the specifier contains of a
   list of ArrayRange elements, each of which 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
   0xffffffff is used to indicate the final element. The beginning
   of the range MUST be earlier in the array then the end. The
   ranges MUST be non-overlapping.
o If the data is of data model dictionary then the specifier
   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 response
with no StoredData values.

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.

6.3.2.2. **Response Definition**

   The response to a successful Fetch request is a FetchAns message
   containing the data requested by the requester.

```
        struct {
          KindId                kind;
          uint64                generation;
          StoredData            values<0..2^32-1>;
        } FetchKindResponse;

        struct {
          FetchKindResponse     kind_responses<0..2^32-1>;
        } FetchAns;
```

   The FetchAns structure contains a series of FetchKindResponse
   structures. There MUST be one FetchKindResponse element for each
   Kind-ID in the request.

   The contents of the FetchKindResponse structure are as follows:

   kind
       the kind that this structure is for.

   generation
       the generation counter for this kind.

   values
       the relevant values. If the generation counter in the request
       matches the generation-counter in the stored data, then no
       StoredData values are returned. Otherwise, all relevant data
       values MUST be returned. A nonexistent value is represented with
       "exists" set to False.

   There is one subtle point about signature computation on arrays. If
   the storing node uses the append feature (where the
   index=0xffffffff), then the index in the StoredData that is returned
   will not match that used by the storing node, which would break the
   signature. In order to avoid this issue, the index value in array is
   set to zero before the signature is computed. This implies that
   malicious storing nodes can reorder array entries without being
   detected. [[OPEN ISSUE: We've considered a number of alternate
   designs here that would preserve security against this attack if the
   storing node did not use the append feature. However, they are more
   complicated for one or both sides. If this attack is considered
   serious, we can introduce one of them.]]

6.3.3. Stat

The Stat request is used to get metadata (length, generation counter, digest, etc.) for a stored element without retrieving the element itself. The name is from the UNIX stat(2) system call which performs a similar function for files in a filesystem. It also allows the requesting node to get a list of matching elements without requesting the entire element.

6.3.3.1. Request Definition

The Stat request is identical to the Fetch request. It simply specifies the elements to get metadata about.

```
struct {
  ResourceId              resource;
  StoredDataSpecifier     specifiers<0..2^16-1>;
} StatReq;
```

6.3.3.2. Response Definition

The Stat response contains the same sort of entries that a Fetch response would contain, however instead of containing the element data it contains metadata.

```
struct {
  Boolean              exists;
  uint32               value_length;
  HashAlgorithm        hash_algorithm;
  opaque               hash_value<0..255>;
} MetaData;


struct {
  uint32               index;
  MetaData             value;
} ArrayEntryMeta;

struct {
  DictionaryKey        key;
  MetaData             value;
} DictionaryEntryMeta;

struct {
  DataModel               model;
```

```
          select (model) {
            case single_value:
              MetaData              single_value_entry;

            case array:
              ArrayEntryMeta        array_entry;

            case dictionary:
              DictionaryEntryMeta   dictionary_entry;


            /* This structure may be extended */
          } ;
        } MetaDataValue;

        struct {
          uint32                  length;
          uint64                  storage_time;
          uint32                  lifetime;
          MetaDataValue           metadata;
        } StoredMetaData;

        struct {
          KindId                  kind;
          uint64                  generation;
          StoredMetaData          values<0..2^32-1>;
        } StatKindResponse;

        struct {
          StatKindResponse        kind_responses<0..2^32-1>;
        } StatAns;
```

The structures used in StatAns parallel those used in FetchAns: a
response consists of multiple StatKindResponse values, one for each
kind that was in the request. The contents of the StatKindResponse
are the same as those in the FetchKindResponse, except that the
values list contains StoredMetaData entries instead of StoredData
entries.

The contents of the StoredMetaData structure are the same as the
corresponding fields in StoredData except that there is no signature
field and the value is a MetaDataValue rather than a StoredDataValue.

A MetaDataValue is a variant structure, like a StoredDataValue,
except for the types of each arm, which replace DataValue with
MetaData.

The only really new structure is MetaData, which has the following

   contents:

   exists
      Same as in DataValue

   value_length
      The length of the stored value.

   hash_algorithm
      The hash algorithm used to perform the digest of the value.

   hash_value
      A digest of the value using hash_algorithm.

6.3.4. Remove

   The Remove request is used to remove a stored element or elements
   from the storing peer. Any successful remove of an existing element
   for a given kind MUST increment the generation counter by at least 1.

```
     struct {
       ResourceId              resource;
       StoredDataSpecifier     specifiers<0..2^16-1>;
     } RemoveReq;
```

   A RemoveReq 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
   RemoveReq. Each specifier 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.

   o The Kind-ID is known.
   o The signature over the message is valid or (depending on overlay
      policy) no signature is required.
   o The signer of the message has permissions which permit him to
      remove this kind of data. Although each kind defines its own
      access control requirements, in general only the original signer
      of the data should be allowed to remove it.
   o 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.

6.3.4.1. Single Value

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

6.3.4.2. Array

A Remove of an array element (or element range) replaces those
elements with null 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"].

6.3.4.3. Dictionary

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

6.3.4.4. 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.

```
     struct {
       StoreKindResponse          kind_responses<0..2^16-1>;
     } RemoveAns;
```

6.3.5. Find

The Find request can be used to explore the Overlay Instance. 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 Overlay Instance by
interactively fetching R_n+1=nearest(1 + R_n).

6.3.5.1. Request Definition

   The FindReq message contains a series of Resource-IDs and Kind-IDs
   identifying the resource the peer is interested in.

```
      struct {
        ResourceID                   resource;
        KindId                       kinds<0..2^8-1>;
      } FindReq;
```

   The request contains a list of Kind-IDs which the Find is for, as
   indicated below:

   resource
      The desired Resource-ID

   kinds
      The desired Kind-IDs. Each value MUST only appear once.

6.3.5.2. Response Definition

   A response to a successful Find request is a FindAns message
   containing the closest Resource-ID for each kind specified in the
   request.

```
      struct {
        KindId                   kind;
        ResourceID               closest;
      } FindKindData;

      struct {
        FindKindData             results<0..2^16-1>;
      } FindAns;
```

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

   For each Kind-ID in the request the response MUST contain a
   FindKindData indicating the closest Resource-ID for that Kind-ID
   unless the kind is not allowed to be used with Find in which case a
   FindKindData 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 FindKindData 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.

6.3.6. Defining New Kinds

TODO: is this the right place in the document for this?

A new kind MUST define:

o The meaning of the data to be stored.
o The Kind-ID.
o The data model (single value, array, dictionary, etc.)
o Access control rules for indicating what credentials are allowed
   to read and write that Kind-ID at a given location.

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, kinds 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.

7. 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 Resource Names:

o The user name in the certificate.
o The Node-ID in the certificate.

Note that in the second case the certificate is not stored at the
peer's Node-ID but rather at a hash of the peer's Node-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
Overlay Instance. New certificates are stored at the end of the
list. This structure allows users to store and old and new
certificate the both have the same Node-ID which allows for migration
of certificates when they are renewed.

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 Node-ID or user name
   which, when hashed, maps to the Resource-ID at which the value is
   being stored.

8. TURN Server Usage

The TURN server usage allows a RELOAD peer to advertise that it is
prepared to be a TURN server as defined in [I-D.ietf-behave-turn].
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 Resource Name
by concatenating its Peer-ID and the value d. This Resource Name is
hashed to form a Resource-ID. The address of the peer is stored at
that Resource-ID using type TURN-SERVICE and the TurnServer object:

```
     struct {
       uint8                   iteration;
       IpAddressAndPort        server_address;
     } TurnServer;
```

The contents of this structure are as follows:

iteration
   the d value

server_address
   the address at which the TURN server can be contacted.

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 peers that provide this service 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 [RFC5382].

[[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.

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.

9. Diagnostic Usage

   The Diagnostic Usage allows a node 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, routing table, stored resource-objects, and performance
statistics of a peer. The usage defines several new kinds which can
be retrieved to get the statistics and also allows to retrieve other
kinds that a node stores. In essence, the usage allows querying a
node's state such as storage and network to obtain the relevant
information. Additional diagnostic capabilities have been proposed
in [I-D.zheng-p2psip-diagnose].

The access control model for all kinds is a local policy defined by
the peer or the overlay policy. The peer may be configured with a
list of users that it is willing to return the information for and
restrict access to users with that name. Unless specific policy
overrides it, data SHOULD NOT be returned for users not on the list.
The access control can also 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.

TODO - need to explain how this is addressed to node-id. [TODO: Do
we need a DIAGNOSTIC method? Access control mechanisms for
DIAGNOSTIC may be different from a Fetch.]

The following kinds are defined:

ROUTING_TABLE_SIZE A single value element containing an unsigned 32-
   bit integer representing the number of peers in the peer's routing
   table.

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

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

APP_UPTIME A single value element containing an unsigned 64-bit
   integer specifying the time the p2p application has been up in
   seconds.

MEMORY_FOOTPRINT A single value element containing an unsigned 32-
   bit integer representing the memory footprint of the peer program
   in kilo bytes.

Note: What's a kilo byte? 1000 or 1024? -- Cullen
Note: Good question. 1000 seems like not quite enough room but
    1024 is too much? -- EKR

DATASIZE_STORED An unsigned 64-bit integer representing the number
   of bytes of data being stored by this node.

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.

MESSAGES_SENT_RCVD 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.

EWMA_BYTES_SENT A single value element containing an unsigned 32-bit
   integer representing an exponential weighted average of bytes sent
   per second by this peer.
   sent = alpha x sent_present + (1 - alpha) x sent
   where sent_present represents the bytes sent per second since the
   last calculation and sent represents the last calculation of bytes
   sent per second. A suitable value for alpha is 0.8. This value
   is calculated every five seconds.

EWMA_BYTES_RCVD A single value element containing an unsigned 32-bit
   integer representing an exponential weighted average of bytes
   received per second by this peer. Same calculation as above.

[[TODO: We would like some sort of bandwidth measurement, but we're
kind of unclear on the units and representation.]]

9.1. Diagnostic Metrics for a P2PSIP Deployment

(OPEN QUESTION: any other metrics?)

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.

10. Chord Algorithm

This algorithm is assigned the name chord-128-2-16+ to indicate it is

based on Chord, uses SHA-1 then truncates that to 128 bit for the
hash function, stores 2 redundant copies of all data, and has finger
tables with at least 16 entries.

10.1. 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 Node-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 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.

10.2. Reactive vs Periodic Recovery

Open Issue: The algorithm currently presented in this section uses
reactive recovery---when a neighbor is lost, that information is
immediately propagated. Research in DHT performance by Rhea et al.
indicates that this is not optimal in large-scale networks with churn
[handling-churn-usenix04]. Addressing this issue, however, needs to
take into account the requirements placed on this algorithm. Because
it is the mandatory DHT for RELOAD, the algorithm described here is
designed to meet two primary challenges:
o Scale from small (ten or fewer) overlays on a LAN to global
   overlays with millions of nodes
o Simple to implement

One of the challenges these requirements entail is achieving
reasonable performance as the overlay scales without undue
complexity. We have two possibly conflicting concerns:

o A small-scale overlay may not be stable without reactive recovery,
   because a single peer represents a large portion of the overlay.
o A large-scale overlay with significant churn may perform poorly,
   both in terms of traffic volume and success rates, when using
   reactive recovery.
As a result, multiple solutions have been proposed:
o Identify one set of behaviors that achieves adequate functionality
   as the overlay scales.
o Add a parameter dictating the type of recovery used by peers in
   the overlay, configuring the peers appropriately as they join the
   overlay.
o Make the algorithm adaptive, according to the size of the overlay
   or the churn rates observed.

At IETF 72, the WG elected to defer a decision on the final choice
until data could be collected on the effectiveness of the strategies.
This section, therefore, retains the reactive recovery model until
evidence supporting a decision is available.

10.3. Routing

If a peer is not responsible for a Resource-ID k, but is directly
connected to a node with Node-ID k, then it routes the message to
that node. Otherwise, it routes the request to the peer in the
routing table that has the largest Node-ID that is in the interval
between the peer and k.

10.4. 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 peers 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. They do not themselves perform further Stores
because they can determine that they are not responsible for the
Resource-ID.

Note that a malicious node can return a success response but not
store the data locally or in the replica set. Requesting peers that
wish to ensure that the replication actually occurred SHOULD [[Open
Issue: SHOULD or MAY?]] 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?]]

10.5. Joining

The join process for a joining party (JP) with Node-ID n is as
follows.

1. JP connects to its chosen bootstrap node.
2. JP uses a series of Probes to populate its routing table.
3. JP sends Attach 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 Node-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 Probe 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 probe 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 an Attach to peer (n+2^(numBitsInNodeId-i). This will be
routed to a peer in approximately the right location around the ring.

10.6. Routing Attaches

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

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

   If a peer is unable to successfully Attach 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

10.7. Updates

   A chord Update is defined as

```
         enum { reserved (0),
               peer_ready(1), neighbors(2), full(3), (255) }
             ChordUpdateType;


         struct {
           ChordUpdateType         type;

           select(type){
             case peer_ready:                    /* Empty */
               ;

             case neighbors:
               NodeId              predecessors<0..2^16-1>;
               NodeId              successors<0..2^16-1>;

             case full:
               NodeId              predecessors<0..2^16-1>;
               NodeId              successors<0..2^16-1>;
               NodeId              fingers<0..2^16-1>;
           };
         } ChordUpdate;
```

   The "type" field contains the type of the update, which depends on
   the reason the update was sent.

   peer_ready: this peer is ready to receive messages. This message
      is used to indicate that a node which has Attached is a peer and
      can be routed through. It is also used as a connectivity check to
      non-neighbor pers.
   neighbors: this version is sent to members of the Chord neighbor
      table.

full: this version is sent to peers which request an Update with a
   RouteQueryReq.

If the message is of type "neighbors", then the contents of the
message will be:

predecessors
   The predecessor set of the Updating peer.

successors
   The successor set of the Updating peer.

If the message is of type "full", then the contents of the message
will be:

predecessors
   The predecessor set of the Updating peer.

successors
   The successor set of the Updating peer.

fingers
   The finger table if the Updating peer, in numerically ascending
   order.

A peer MUST maintain an association (via Attach) 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 sent to neighbors.

10.7.1. Sending Updates

Every time a connection to a peer in the neighborhood set is lost (as
determined by connectivity probes 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 Node-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 Probes 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.

10.7.2. Receiving Updates

When a peer, N, receives an Update request, it examines the Node-IDs
in the UpdateReq and at its neighborhood table and decides if this
UpdateReq 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:

o The UpdateReq contains peers that would not change the neighbor
   set because they match the neighborhood table.
o The UpdateReq contains peers closer to N than those in its
   neighborhood table.
o The UpdateReq 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 Attach 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 Probe the questionable peers to discover
if they are indeed missing and if so, remove them from its
neighborhood table.

After any Probes and Attaches 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.

If 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.

10.7.3. Stabilization

There are four components to stabilization:
1. exchange Updates with 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. If a finger table entry is
found to have failed, the peer MUST search for a replacement as
directed below.

A peer MUST periodically select a random entry i from the finger
table and evaluate whether that entry should be replaced. The
default time interval is about every hour, but the enrollment server
SHOULD set this in the configuration document using the "chord-128-2-
16+-probe-frequency" element (denominated in seconds).

To evaluate whether the i'th finger table entry needs to be replaced,
if the Node-ID of the entry is not valid for that finger table entry,
the peer SHOULD search for a better entry. A peer searches for a
better entry using a Probe request. If the Probe returns a different
peer than the one currently in this entry of the finger table, then a
new connection should be formed to replace the old entry in the
finger table.

A peer SHOULD consider the finger table entry valid if it is in the

range [n+2^(numBitsInNodeId-i), n+2^(numBitsInNodeId-(i-1))-
2^(numBitsInNodeId-(i+1))]. When searching for a better entry, the
peer SHOULD send the Probe to a Node-ID selected randomly from that
range. Random selection is preferred over a search for strictly
spaced entries to minimize the effect of churn on overlay routing
[minimizing-churn-sigcomm06]. An implementation or subsequent
specification MAY choose a method for selecting finger table entries
other than choosing randomly within the range. It is RECOMMENDED
that any such alternate methods be employed only on finger table
stabilization and not for the selection of initial finger table
entries unless the alternative method is faster and imposes less
overhead on the overlay.

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 Probe 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 Probe for its own Node-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 Attach 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.)

10.8. Route Query

For this topology plugin, the RouteQueryReq contains no additional
information. The RouteQueryAns contains the single node ID of the
next peer to which the responding peer would have routed the request
message in recursive routing:

```
      struct {
         NodeId                  next_id;
      } ChordRouteQueryAns;
```

   The contents of this structure 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.

   If the requester set the send_update flag, the responder SHOULD
   initiate an Update immediately after sending the RouteQueryAns.

10.9. Leaving

   Peers SHOULD send a Leave request prior to exiting the Overlay
   Instance. 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 Updates containing its new predecessor and
   successor tables.

11. Enrollment and Bootstrap

11.1. Overlay Configuration

   This specification defines a new content type "application/
   p2p-overlay+xml" for an MIME entity that contains overlay
   information. An example document is shown below.

```
      <?xml version="1.0" encoding="UTF-8"?>
        <overlay instance-name="chord.example.com" expiration="86400">
          <toplogy-plugin algorithm-name="chord-128-2-16+"/>
          <root-cert>[PEM encoded certificate here]</root-cert>
          <required-kind name="SIP-REGISTRATION" max-values="10"
           max-size="1000"/>
          <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 conforment 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:

instance-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:

topology-plugin This element has an attribute called algorithm-name
   that describes the overlay-algorithm being used.
root-cert   This element contains a PEM 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-kinds   This element indicates the kinds that members must
   support. It has three attributes:
   * name: a string representing the kind.
   * max-count: the maximum number of values which members of the
      overlay must support.
   * max-size: the maximum size of individual values.
   For instance, the example above indicates that members must
   support SIP-REGISTRATION with a maximum of 10 values of up to 1000
   bytes each. Multiple required-kinds elements 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.
self-signed-permitted   This element indicates whether self-signed
   certificates are permitted. If it is set to "TRUE", then self-
   signed certificates are allowed, in which case the credential-
   server and root-cert elements may be absent. Otherwise, it SHOULD
   be absent, but MAY be set "FALSE". This element also contains an
   attribute "digest" which indicates the digest to be used to
   compute the Node-ID. Valid values for this parameter are "SHA-1"
   and "SHA-256".

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.
clients-permitted   This element represents whether clients are
   permitted or whether all nodes must be peers. If it is set to
   "TRUE" or absent, this indicates that clients are permitted. If
   it is set to "FALSE" then nodes MUST join as peers.
attach-lite-permitted   This element represents whether nodes are
   allowed to use the AttachLite request in this overlay. If it is
   absent, it is treated as if it was set to "FALSE".
chord-128-2-16+-update-frequency   The update frequency for the
   Chord-128-2-16+ topology plugin (see Section 10).
chord-128-2-16+-probe-frequency   The probe frequency for the Chord-
   128-2-16+ topology plugin (see Section 10).
credential-server Base URL for credential server.
shared-secret If shared secret mode is used, this contains the
   shared secret.

   [[TODO: Do a RelaxNG grammar.]]

11.2. Discovery Through Enrollment Server

   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]. Another scheme for
   referencing overlays is described in
   [I-D.hardie-p2poverlay-pointers]. 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.

   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].

   Whenever a peer contacts the enrollment server, it MUST fetch a new
   copy of the configuration file. To do this, the peer 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 described above, which replaces any previously
learned configuration file for this overlay.

[[OPEN ISSUE: for unsecured overlays or overlays not specified by
domain name, need to specify another way to obtain/validate certs and
to update configuration info]]

11.3. Credentials

If the configuration document contains a credential-server element,
credentials are required to join the Overlay Instance. 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
[RFC5272]) and sends this over HTTPS as defined in [RFC5273] to the
URL in the credential-server element. The subjectAltName in the
request MUST contain the required user name.

The credential server MUST authenticate the request using the
provided user name and password. If the authentication succeeds and
the requested user name is acceptable, the server and returns a
certificate. The SubjectAltName field in the certificate contains
the following values:

o One or more Node-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. These
   are of type URI and MUST contain RELOAD URIs as described in
   Section 14.10 and MUST contain a Destination list with a single
   entry of type "node_id".
o The names this user is allowed to use in the overlay, using type
   rfc822Name.

The certificate is returned in a "Simple Enrollment Response".
[[TODO: REF]]

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
Node-IDs it can use.

11.3.1. Self-Generated Credentials

   If the "self-signed-permitted" element is present and set to "TRUE",
   then a node MUST generate its own self-signed certificate to join the
   overlay. The self-signed certificate MAY contain any user name of
   the users choice. Users SHOULD make some attempt to make it unique
   but this document does not specify any mechanisms for that.

   The Node-ID MUST be computed by applying the digest specified in the
   self-signed-permitted element to the DER representation of the user's
   public key. When accepting a self-signed certificate, nodes MUST
   check that the Node-ID and public keys match. This prevents Node-ID
   theft.

   Once the node has constructed a self-signed certificate, it MAY join
   the overlay. Before storing its certificate in the overlay
   (Section 7) it SHOULD look to see if the user name is already taken
   and if so choose another user name. Note that this only provides
   protection against accidental name collisions. Name theft is still
   possible. If protection against name theft is desired, then the
   enrollment service must be used.

11.4. Joining the Overlay 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 Probe request to the
   known peer address with the destination Node-ID set to that peer's
   Node-ID.

   If no cached peers are available, then the peer SHOULD send a Probe
   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 Probe should use the wildcard Node-ID as the
   destination Node-ID.

   The responder peer that receives the Probe 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 Probe 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 Node-ID in the response and use this
   Node-ID to start sending requests to join the Overlay Instance as
   described in Section 5.3.

   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.

12. Message Flow Example

   In the following example, we assume that JP has formed a connection
   to one of the bootstrap peers. JP then sends an Attach through that
   peer to the admitting peer (AP) to initiate a connection. When AP
   responds, JP and AP use ICE to set up a connection and then set up
   TLS.

```
          JP           PPP          PP           AP           NP           NNP          BP
           |            |            |            |            |            |            |
           |            |            |            |            |            |            |
           |            |            |            |            |            |            |
           |Attach Dest=JP           |            |            |            |            |
           |-------------------------------------------------------------------------->|
           |            |            |            |            |            |            |
           |            |            |            |            |            |            |
           |            |            |Attach Dest=JP           |            |            |
           |            |            |<--------------------------------------|            |
           |            |            |            |            |            |            |
           |            |            |            |            |            |            |
           |            |            |Attach Dest=JP           |            |            |
           |            |            |-------->|            |            |            |
           |            |            |            |            |            |            |
           |            |            |            |            |            |            |
           |            |            |AttachAns   |            |            |            |
           |            |            |<--------|            |            |            |
           |            |            |            |            |            |            |
           |            |            |            |            |            |            |
           |            |            |AttachAns   |            |            |            |
           |            |            |-------------------------------------->|            |
           |            |            |            |            |            |            |
           |            |            |            |            |            |            |
           |AttachAns    |            |            |            |            |            |
           |<--------------------------------------------------------------------------|
           |            |            |            |            |            |            |
           |            |            |            |            |            |            |
           |TLS         |            |            |            |            |            |
           |.............................|            |            |            |            |
           |            |            |            |            |            |            |
           |            |            |            |            |            |            |
           |            |            |            |            |            |            |
```

```
    |         |         |         |         |         |         |
```

Once JP has connected to AP, it needs to populate its Routing Table.
In Chord, this means that it needs to populate its neighbor table and
its finger table. To populate its neighbor table, it needs the
successor of AP, NP. It sends an Attach to the Resource-IP AP+1,
which gets routed to NP. When NP responds, JP and NP use ICE and TLS
to set up a connection.

```
         JP         PPP        PP         AP         NP         NNP        BP
          |          |          |          |          |          |          |
          |          |          |          |          |          |          |
          |          |          |          |          |          |          |
          |Attach AP+1          |          |          |          |          |
          |---------------------------->|          |          |          |
          |          |          |          |          |          |          |
          |          |          |          |          |          |          |
          |          |          |          |Attach AP+1          |          |
          |          |          |          |-------->|          |          |
          |          |          |          |          |          |          |
          |          |          |          |          |          |          |
          |          |          |          |AttachAns |          |          |
          |          |          |          |<--------|          |          |
          |          |          |          |          |          |          |
          |          |          |          |          |          |          |
          |AttachAns |          |          |          |          |          |
          |<----------------------------|          |          |          |
          |          |          |          |          |          |          |
          |          |          |          |          |          |          |
          |Attach    |          |          |          |          |          |
          |-------------------------------------->|          |          |
          |          |          |          |          |          |          |
          |          |          |          |          |          |          |
          |TLS       |          |          |          |          |          |
          |.......................................|          |          |
          |          |          |          |          |          |          |
          |          |          |          |          |          |          |
          |          |          |          |          |          |          |
          |          |          |          |          |          |          |
```
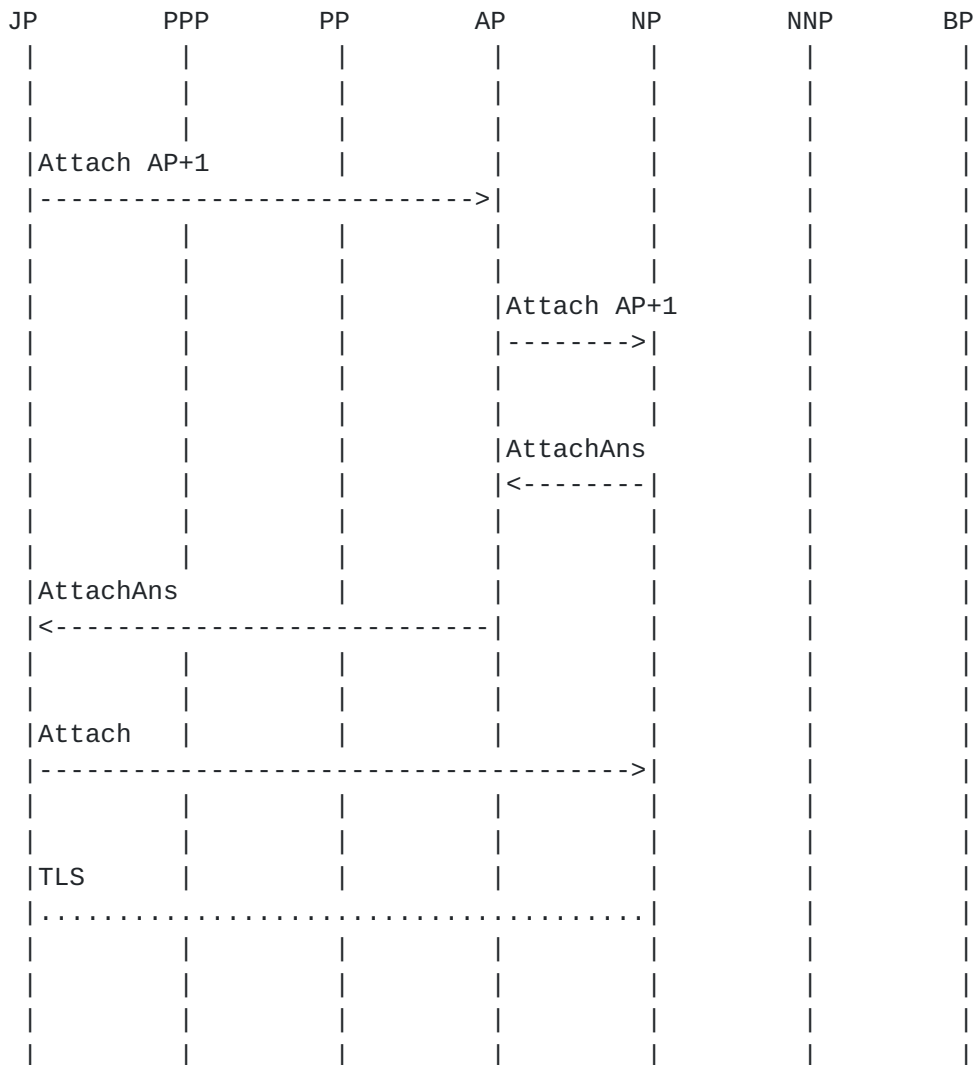
JP also needs to populate its finger table (for Chord). It issues a
Attach to a variety of locations around the overlay. The diagram
below shows it sending an Attach halfway around the Chord ring the JP
+ 2^127.

```
            JP          NP          XX          TP
             |           |           |           |
             |           |           |           |
             |           |           |           |
             |Attach JP+2<<126       |           |
             |-------->|           |           |
             |           |           |           |
             |           |           |           |
             |           |Attach JP+2<<126       |
             |           |-------->|           |
             |           |           |           |
             |           |           |           |
             |           |           |Attach JP+2<<126
             |           |           |-------->|
             |           |           |           |
             |           |           |           |
             |           |           |AttachAns|
             |           |           |<--------|
             |           |           |           |
             |           |           |           |
             |           |AttachAns|           |
             |           |<--------|           |
             |           |           |           |
             |           |           |           |
             |AttachAns|           |           |
             |<--------|           |           |
             |           |           |           |
             |           |           |           |
             |TLS        |           |           |
             |...........................|
             |           |           |           |
             |           |           |           |
             |           |           |           |
             |           |           |           |
```
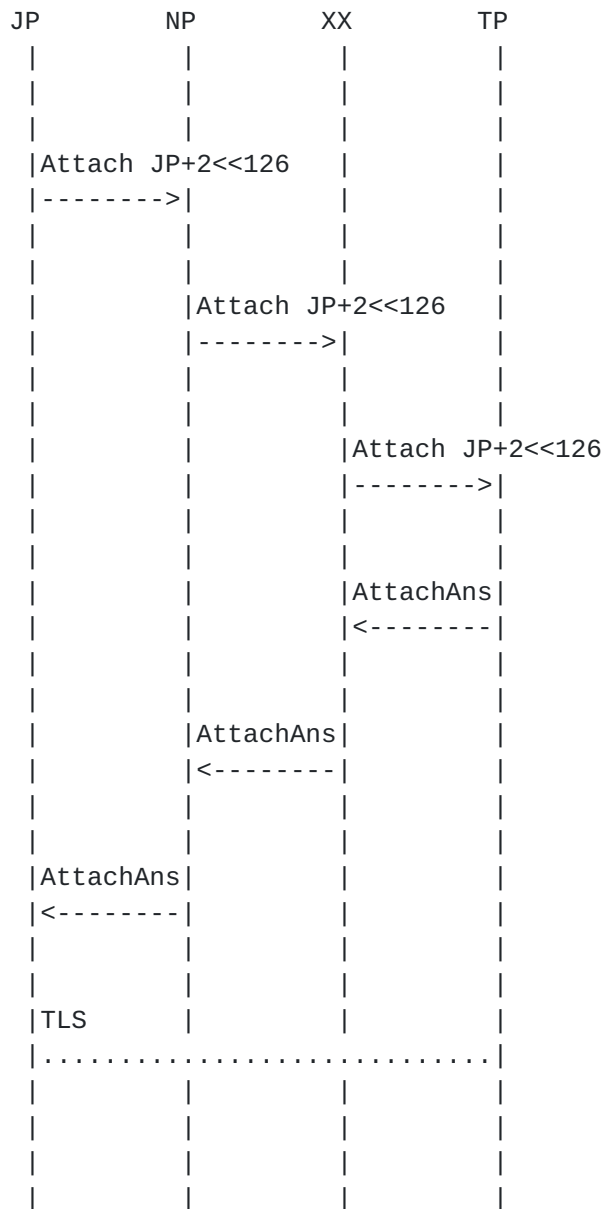
   Once JP has a reasonable set of connections he is ready to take his
   place in the DHT. He does this by sending a Join to AP. AP does a
   series of Store requests to JP to store the data that JP will be
   responsible for. AP then sends JP an Update explicitly labeling JP
   as its predecessor. At 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.

```
          JP         PPP        PP         AP         NP         NNP        BP
           |          |          |          |          |          |          |
           |          |          |          |          |          |          |
           |          |          |          |          |          |          |
           |JoinReq   |          |          |          |          |          |
           |------------------------------->|          |          |          |
           |          |          |          |          |          |          |
           |          |          |          |          |          |          |
           |JoinAns   |          |          |          |          |          |
           |<-------------------------------|          |          |          |
           |          |          |          |          |          |          |
           |          |          |          |          |          |          |
           |StoreReq Data A      |          |          |          |          |
           |<-------------------------------|          |          |          |
           |          |          |          |          |          |          |
           |          |          |          |          |          |          |
           |StoreAns  |          |          |          |          |          |
           |------------------------------->|          |          |          |
           |          |          |          |          |          |          |
           |          |          |          |          |          |          |
           |StoreReq Data B      |          |          |          |          |
           |<-------------------------------|          |          |          |
           |          |          |          |          |          |          |
           |          |          |          |          |          |          |
           |StoreAns  |          |          |          |          |          |
           |------------------------------->|          |          |          |
           |          |          |          |          |          |          |
           |          |          |          |          |          |          |
           |UpdateReq |          |          |          |          |          |
           |<-------------------------------|          |          |          |
           |          |          |          |          |          |          |
           |          |          |          |          |          |          |
           |UpdateAns |          |          |          |          |          |
           |------------------------------->|          |          |          |
           |          |          |          |          |          |          |
           |          |          |          |          |          |          |
           |          |          |          |          |          |          |
           |          |          |          |          |          |          |
```
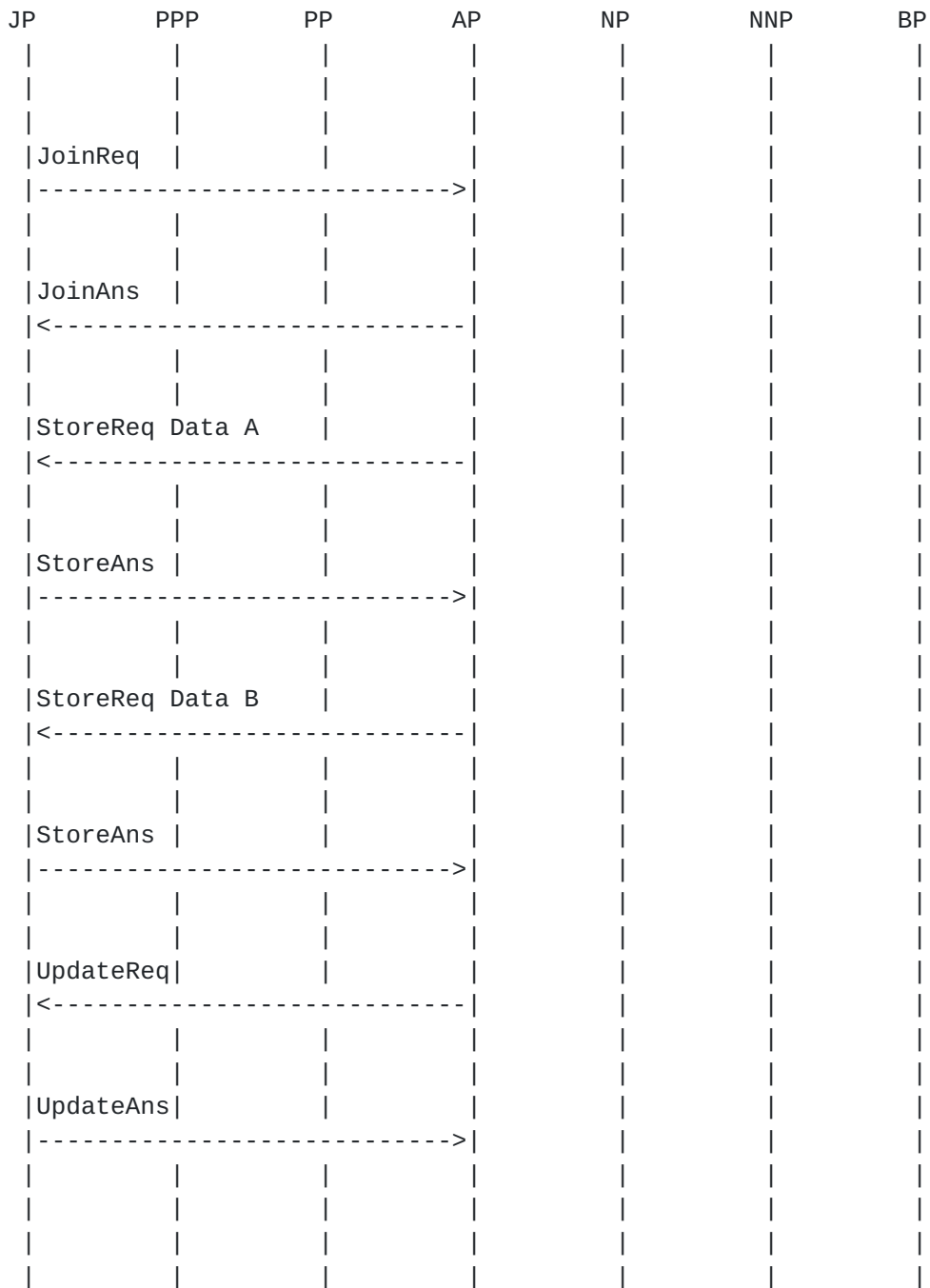
In Chord, JP's neighbor table needs to contain its own predecessors.
It couldn't connect to them previously because Chord has no way to
route immediately to your predecessors. However, now that it has
received an Update from AP, it has APs predecessors, which are also
its own, so it sends Attaches to them. Below it is shown connecting
to its closest predecessor, PP.

```
          JP        PPP       PP        AP        NP        NNP       BP
          |         |         |         |         |         |         |
          |         |         |         |         |         |         |
          |         |         |         |         |         |         |
          |Attach Dest=PP     |         |         |         |         |
          |--------------------------->|         |         |         |
          |         |         |         |         |         |         |
          |         |         |         |         |         |         |
          |         |         |Attach Dest=PP     |         |         |
          |         |         |<--------|         |         |         |
          |         |         |         |         |         |         |
          |         |         |         |         |         |         |
          |         |         |AttachAns|         |         |         |
          |         |         |-------->|         |         |         |
          |         |         |         |         |         |         |
          |         |         |         |         |         |         |
          |AttachAns|         |         |         |         |         |
          |<---------------------------|         |         |         |
          |         |         |         |         |         |         |
          |         |         |         |         |         |         |
          |TLS      |         |         |         |         |         |
          |..................|         |         |         |         |
          |         |         |         |         |         |         |
          |         |         |         |         |         |         |
          |UpdateReq|         |         |         |         |         |
          |----------------->|         |         |         |         |
          |         |         |         |         |         |         |
          |         |         |         |         |         |         |
          |UpdateAns|         |         |         |         |         |
          |<-----------------|         |         |         |         |
          |         |         |         |         |         |         |
          |         |         |         |         |         |         |
          |UpdateReq|         |         |         |         |         |
          |--------------------------->|         |         |         |
          |         |         |         |         |         |         |
          |         |         |         |         |         |         |
          |UpdateAns|         |         |         |         |         |
          |<---------------------------|         |         |         |
          |         |         |         |         |         |         |
          |         |         |         |         |         |         |
          |UpdateReq|         |         |         |         |         |
          |------------------------------------->|         |         |
          |         |         |         |         |         |         |
          |         |         |         |         |         |         |
          |UpdateAns|         |         |         |         |         |
          |<-------------------------------------|         |         |
          |         |         |         |         |         |         |
          |         |         |         |         |         |         |
          |         |         |         |         |         |         |
```
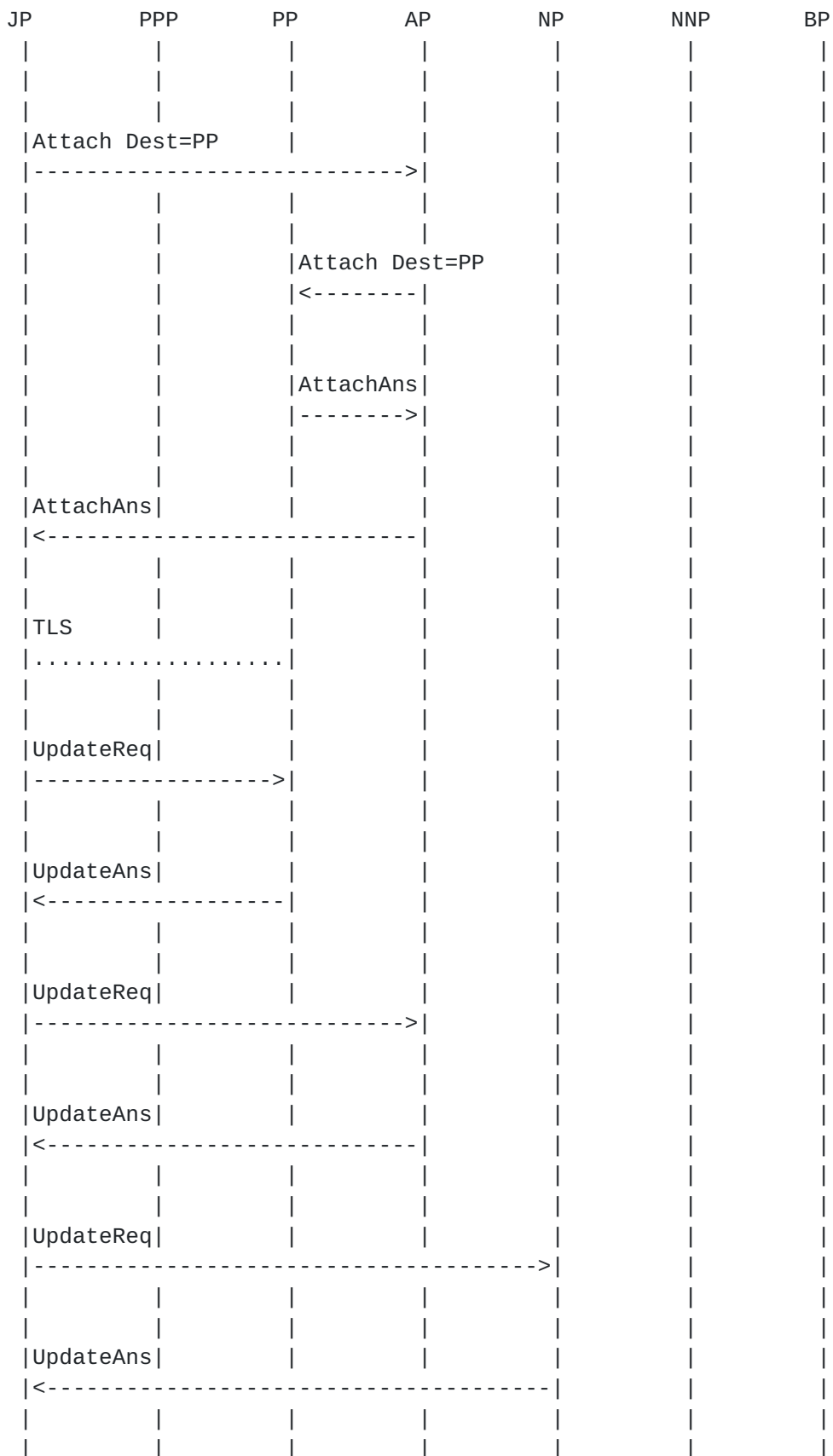
Finally, now that JP has a copy of all the data and is ready to route
messages and receive requests, it sends Updates to everyone in its
Routing Table to tell them it is ready to go. Below, it is shown
sending such an update to TP.

```
         JP          NP          XX          TP
          |           |           |           |
          |           |           |           |
          |           |           |           |
          |Update     |           |           |
          |--------------------------------->|
          |           |           |           |
          |           |           |           |
          |UpdateAns|           |           |
          |<---------------------------------|
          |           |           |           |
          |           |           |           |
          |           |           |           |
          |           |           |           |
```
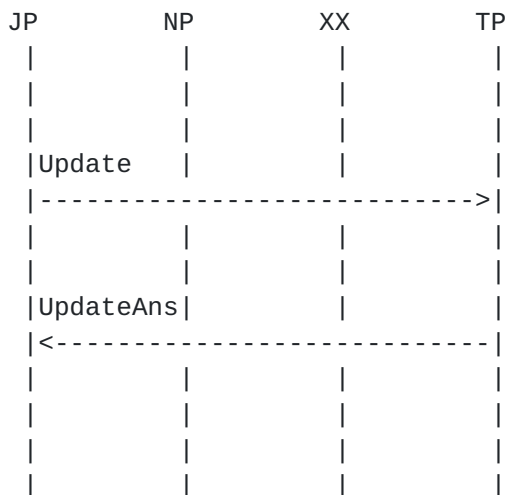
13. Security Considerations

13.1. Overview

RELOAD provides a generic storage service, albeit one designed to be
useful for P2PSIP. In this section we discuss security issues that
are likely to be relevant to any usage of RELOAD.

In any Overlay Instance, 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 Overlay Instance. 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 overlay 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 an admission control mechanism based on
an overlay-wide shared symmetric key.

13.2. Attacks on P2P Overlays

   The two basic functions provided by overlay 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 overlay 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 a
   secure channel (TLS and DTLS are defined in this document) which
   provides message integrity and authentication of the directly
   communicating peer. In addition, messages and data are digitally
   signed with the sender's private key, providing end-to-end security
   for communications.

13.3. Certificate-based Security

   This specification stores users' registrations and possibly other
   data in an overlay network. This requires a solution to securing
   this data as well as securing, as well as possible, the routing in
   the overlay. 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 Overlay Instance, 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 Node-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:

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

   Note that since only users of this Overlay Instance need to validate

a certificate, this usage does not require a global PKI. Instead,
certificates are signed by require a central enrollment authority
which acts as the certificate authority for the Overlay Instance.
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.

If self-signed certificates are used, then the security provided is
significantly decreased, since attackers can mount Sybil attacks. In
addition, attackers cannot trust the user names in certificates
(though they can trust the Node-IDs because they are
cryptographically verifiable). This scheme is only appropriate for
small deployments, such as a small office or ad hoc overlay set up
among participants in a meeting. Some additional security can be
provided by using the shared secret admission control scheme as well.

Because all stored data is signed by the owner of the data the
storing peer can 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.

13.4. **Shared-Secret Security**

RELOAD also supports a shared secret admission control scheme 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 [RFC5054] mode. A peer which does not know the
key cannot form TLS connections with any other peer and therefore
cannot join the overlay.

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.

13.5. **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 overlays - relatively simple.

13.5.1. 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 Resource Name 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 Resource Name construction algorithm for the
specified kind based on the user's certificate information. It then
computes the Resource-ID from the Resource Name 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
          Node-ID: 013456789abcdef
          Serial: 1234

If Alice wishes to Store a value of the "SIP Location" kind, the
Resource Name will be the SIP AOR "sip:alice@dht.example.com". The
Resource-ID will be determined by hashing the Resource Name. When a
peer receives a request to store a record at Resource-ID X, it takes
the signing certificate and recomputes the Resource Name, 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
Resource Name construction algorithm may be different for other
kinds.

13.5.2. Distributed Quota

Being a peer in a Overlay Instance carries with it the responsibility
to store data for a given region of the Overlay Instance. 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 configurations 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.

13.5.3. 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.

13.5.4. 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.

13.6. 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 overlay. 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 Overlay Instance. Second, this is a DOS
attack only. Third, if a large percentage of the peers on the
Overlay Instance are controlled by the attacker, it is probably
impossible to perfectly secure against this.

13.6.1. 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.

13.6.2. Admissions Control

Admission to an RELOAD Overlay Instance is controlled by requiring
that each peer have a certificate containing its peer ID. The
requirement to have a certificate is enforced by using certificate-
based 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:

   o It allows the enrollment server to limit the number of peer IDs
      issued to any individual user.
   o 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 Overlay Instance he is
   impersonating appears to be very sparsely populated by comparison to
   the victim's local region.

[13.6.3](#). Peer Identification and Authentication

   In general, whenever a peer engages in overlay 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 certificate-based
   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 overlay structure,
   but they need to verify that the Update message came from the actual
   peer rather than from an attacker. To prevent this, all overlay
   routing messages are signed by the peer that generated them.

   [OPEN ISSUE: 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 overlay messages.]

[13.6.4](#). 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.

13.6.5. 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 overlay 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.

14. IANA Considerations

This section contains the new code points registered by this
document. [NOTE TO IANA/RFC-EDITOR: Please replace RFC-AAAA with
the RFC number for this specification in the following list.]

[[TODO - add IANA registration for p2p_enroll SRV and p2p_menroll]]

14.1. Overlay Algorithm Types

IANA SHALL create a "RELOAD Overlay Algorithm Type" Registry.
Entries in this registry are strings denoting the names of overlay
algorithms. The registration policy for this registry is RFC 5226
IETF Review. The initial contents of this registry are:

```
+-----------------+----------+
| Algorithm Name  |      RFC |
+-----------------+----------+
| chord-128-2-16+ | RFC-AAAA |
+-----------------+----------+
```

14.2. Data Kind-ID

   IANA SHALL create a "RELOAD Data Kind-ID" Registry. Entries in this
   registry are 32-bit integers denoting data kinds, as described in
   Section 4.1.2. Code points in the range 0x00000001 to 0x7fffffff
   SHALL be registered via RFC 5226 Standards Action. Code points in
   the range 0x8000000 to 0xfffffffe SHALL be registered via RFC 5226
   Expert Review. The initial contents of this registry are:

```
+--------------------+------------+----------+
| Kind               |   Kind-ID  |      RFC |
+--------------------+------------+----------+
| INVALID            |          0 | RFC-AAAA |
| SIP-REGISTRATION   |          1 | RFC-AAAA |
| TURN_SERVICE       |          2 | RFC-AAAA |
| CERTIFICATE        |          3 | RFC-AAAA |
| ROUTING_TABLE_SIZE |          4 | RFC-AAAA |
| SOFTWARE_VERSION   |          5 | RFC-AAAA |
| MACHINE_UPTIME     |          6 | RFC-AAAA |
| APP_UPTIME         |          7 | RFC-AAAA |
| MEMORY_FOOTPRINT   |          8 | RFC-AAAA |
| DATASIZE_StoreD    |          9 | RFC-AAAA |
| INSTANCES_StoreD   |         10 | RFC-AAAA |
| MESSAGES_SENT_RCVD |         11 | RFC-AAAA |
| EWMA_BYTES_SENT    |         12 | RFC-AAAA |
| EWMA_BYTES_RCVD    |         13 | RFC-AAAA |
| LAST_CONTACT       |         14 | RFC-AAAA |
| RTT                |         15 | RFC-AAAA |
| Reserved           | 0x7fffffff | RFC-AAAA |
| Reserved           | 0xffffffff | RFC-AAAA |
+--------------------+------------+----------+
```

14.3. Data Model

   IANA SHALL create a "RELOAD Data Model" Registry. Entries in this
   registry are 8-bit integers denoting data models, as described in
   Section 6.2. Code points in this registry SHALL be registered via
   RFC 5226 IETF Review. The initial contents of this registry are:

```
              +--------------+------+----------+
              | Data Model   | Code |      RFC |
              +--------------+------+----------+
              | INVALID      |    0 | RFC-AAAA |
              | SINGLE_VALUE |    1 | RFC-AAAA |
              | ARRAY        |    2 | RFC-AAAA |
              | DICTIONARY   |    3 | RFC-AAAA |
              | RESERVED     |  255 | RFC-AAAA |
              +--------------+------+----------+
```

14.4. Message Codes

IANA SHALL create a "RELOAD Message Code" Registry. Entries in this
registry are 16-bit integers denoting method codes as described in
Section 5.2.3. These codes SHALL be registered via RFC 5226
Standards Action. The initial contents of this registry are:

```
          +-------------------+----------------+----------+
          | Message Code Name |   Code Value   |      RFC |
          +-------------------+----------------+----------+
          | invalid           |             0  | RFC-AAAA |
          | probe_req         |             1  | RFC-AAAA |
          | probe_ans         |             2  | RFC-AAAA |
          | attach_req        |             3  | RFC-AAAA |
          | attach_ans        |             4  | RFC-AAAA |
          | unused            |             5  |          |
          | unused            |             6  |          |
          | store_req         |             7  | RFC-AAAA |
          | store_ans         |             8  | RFC-AAAA |
          | fetch_req         |             9  | RFC-AAAA |
          | fetch_ans         |            10  | RFC-AAAA |
          | remove_req        |            11  | RFC-AAAA |
          | remove_ans        |            12  | RFC-AAAA |
          | find_req          |            13  | RFC-AAAA |
          | find_ans          |            14  | RFC-AAAA |
          | join_req          |            15  | RFC-AAAA |
          | join_ans          |            16  | RFC-AAAA |
          | leave_req         |            17  | RFC-AAAA |
          | leave_ans         |            18  | RFC-AAAA |
          | update_req        |            19  | RFC-AAAA |
          | update_ans        |            20  | RFC-AAAA |
          | route_query_req   |            21  | RFC-AAAA |
          | route_query_ans   |            22  | RFC-AAAA |
          | ping_req          |            23  | RFC-AAAA |
          | ping_ans          |            24  | RFC-AAAA |
          | stat_req          |            25  | RFC-AAAA |
          | stat_ans          |            26  | RFC-AAAA |
          | attachlite_req    |            27  | RFC-AAAA |
          | attachlite_ans    |            28  | RFC-AAAA |
          | reserved          | 0x8000..0xfffe | RFC-AAAA |
          | error             |         0xffff | RFC-AAAA |
          +-------------------+----------------+----------+
```

14.5. Error Codes

   IANA SHALL create a "RELOAD Error Code" Registry. Entries in this
   registry are 16-bit integers denoting error codes. New entries SHALL
   be defined via RFC 5226 Standards Action. The initial contents of
   this registry are:

```
+------------------------------------+---------------+----------+
| Error Code Name                    |  Code Value   |     RFC  |
+------------------------------------+---------------+----------+
| invalid                            |            0  | RFC-AAAA |
| Error_Unauthorized                 |            1  | RFC-AAAA |
| Error_Forbidden                    |            2  | RFC-AAAA |
| Error_Not_Found                    |            3  | RFC-AAAA |
| Error_Request_Timeout              |            4  | RFC-AAAA |
| Error_Precondition_Failed          |            5  | RFC-AAAA |
| Error_Incompatible_with_Overlay    |            6  | RFC-AAAA |
| Error_Unsupported_Forwarding_Option|            7  | RFC-AAAA |
| Error_Data_Too_Large               |            8  | RFC-AAAA |
| Error_Data_Too_Old                 |            9  | RFC-AAAA |
| reserved                           | 0x8000..0xfffe| RFC-AAAA |
+------------------------------------+---------------+----------+
```

14.6. Route Log Extension Types

IANA SHALL create a "RELOAD Route Log Extension Type Registry." New
entries SHALL be defined via RFC 5226 Specification Required. The
initial contents of this registry are:

```
+--------------------------+------+---------------+
| Route Log Extension Name | Code | Specification |
+--------------------------+------+---------------+
| invalid                  |   0  |      RFC-AAAA |
| reserved                 |  255 |      RFC-AAAA |
+--------------------------+------+---------------+
```

14.7. Overlay Link Types

IANA shall create a "RELOAD Overlay Link Type Registry." New entries
SHALL be defined via RFC 5226 Standards Action. This registry SHALL
be initially populated with the following values:

```
+----------+------+---------------+
| Protocol | Code | Specification |
+----------+------+---------------+
| invalid  |   0  |      RFC-AAAA |
| tcp_tls  |   1  |      RFC-AAAA |
| udp_dtls |   2  |      RFC-AAAA |
| reserved |  255 |      RFC-AAAA |
+----------+------+---------------+
```

14.8. Forwarding Options

   IANA shall create a "Forwarding Option Registry". Entries in this
   registry between 1 and 127 SHALL be defined via RFC 5226 Standards
   Action. Entries in this registry between 128 and 254 SHALL be
   defined via RFC 5226 Specification Required. This registry SHALL be
   initially populated with the following values:

```
         +-------------------+------+---------------+
         | Forwarding Option | Code | Specification |
         +-------------------+------+---------------+
         | invalid           |    0 |     RFC-AAAA  |
         | reserved          |  255 |     RFC-AAAA  |
         +-------------------+------+---------------+
```

14.9. Probe Information Types

   IANA shall create a "RELOAD Probe Information Type Registry".
   Entries in this registry SHALL be defined via RFC 5226 Standards
   Action. This registry SHALL be initially populated with the
   following values:

```
         +-----------------+------+---------------+
         | Probe Option    | Code | Specification |
         +-----------------+------+---------------+
         | invalid         |    0 |     RFC-AAAA  |
         | responsible_set |    1 |     RFC-AAAA  |
         | requested_info  |    2 |     RFC-AAAA  |
         | reserved        |  255 |     RFC-AAAA  |
         +-----------------+------+---------------+
```

14.10. reload: URI Scheme

   This section describes the scheme for a reload: URI, which can be
   used to refer to either:

   o A peer.
   o A resource inside a peer.

   The reload: URI is defined using a subset of the URI schema
   specified in Appendix A. of RFC 3986 [REF] and the associated URI
   Guidelines [REF: RFC4395] per the following ABNF syntax:

```
     RELOAD-URI = "reload://" destination "@" overlay "/"
             [specifier]

           destination = 1 * HEXDIG
     overlay = reg-name
```

```
   specifier = 1*HEXDIG
```

The definitions of these productions are as follows:

destination: a hex-encoded Destination List object.

overlay: the name of the overlay.

specifier : a hex-encoded StoredDataSpecifier indicating the data
   element.

If no specifier is present than this URI addresses the peer which can
be reached via the indicated destination list at the indicated
overlay name. If a specifier is present, then the URI addresses the
data value.

14.10.1. URI Registration

The following summarizes the information necessary to register the
reload: URI.

URI Scheme Name: reload
Status: permanent
URI Scheme Syntax: see Section 14.10.
URI Scheme Semantics: The reload: URI is intended to be used as a
   reference to a RELOAD peer or resource.
Encoding Considerations: The reload: URI is not intended to be
   human-readable text, therefore they are encoded entirely in US-
   ASCII.
Applications/protocols that use this URI scheme: The RELOAD
   protocol described in RFC-AAAA.
   TBD for the rest of this template.

15. 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.

16. References

16.1. Normative References

   [RFC2119]   Bradner, S., "Key words for use in RFCs to Indicate
               Requirement Levels", BCP 14, RFC 2119, March 1997.

   [I-D.ietf-mmusic-ice]
               Rosenberg, J., "Interactive Connectivity Establishment
               (ICE): A Protocol for Network Address Translator (NAT)
               Traversal for Offer/Answer Protocols",
               draft-ietf-mmusic-ice-19 (work in progress), October 2007.

   [RFC5389]   Rosenberg, J., Mahy, R., Matthews, P., and D. Wing,
               "Session Traversal Utilities for NAT (STUN)", RFC 5389,
               October 2008.

   [I-D.ietf-behave-turn]
               Rosenberg, J., Mahy, R., and P. Matthews, "Traversal Using
               Relays around NAT (TURN): Relay Extensions to Session
               Traversal Utilities for NAT (STUN)",
               draft-ietf-behave-turn-12 (work in progress),
               November 2008.

   [RFC5273]   Schaad, J. and M. Myers, "Certificate Management over CMS
               (CMC): Transport Protocols", RFC 5273, June 2008.

   [RFC5272]   Schaad, J. and M. Myers, "Certificate Management over CMS
               (CMC)", RFC 5272, June 2008.

   [RFC4279]   Eronen, P. and H. Tschofenig, "Pre-Shared Key Ciphersuites
               for Transport Layer Security (TLS)", RFC 4279,
               December 2005.

   [I-D.ietf-mmusic-ice-tcp]
               Rosenberg, J., "TCP Candidates with Interactive
               Connectivity Establishment (ICE)",
               draft-ietf-mmusic-ice-tcp-07 (work in progress),
               July 2008.

   [RFC4347]   Rescorla, E. and N. Modadugu, "Datagram Transport Layer
               Security", RFC 4347, April 2006.

   [RFC5348]   Floyd, S., Handley, M., Padhye, J., and J. Widmer, "TCP
               Friendly Rate Control (TFRC): Protocol Specification",

              RFC 5348, September 2008.

16.2. Informative References

   [RFC4828] Floyd, S. and E. Kohler, "TCP Friendly Rate Control
              (TFRC): The Small-Packet (SP) Variant", RFC 4828,
              April 2007.

   [I-D.ietf-p2psip-concepts]
              Bryan, D., Matthews, P., Shim, E., Willis, D., and S.
              Dawkins, "Concepts and Terminology for Peer to Peer SIP",
              draft-ietf-p2psip-concepts-02 (work in progress),
              July 2008.

   [RFC1122] Braden, R., "Requirements for Internet Hosts -
              Communication Layers", STD 3, RFC 1122, October 1989.

   [RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston,
              A., Peterson, J., Sparks, R., Handley, M., and E.
              Schooler, "SIP: Session Initiation Protocol", RFC 3261,
              June 2002.

   [RFC5382] Guha, S., Biswas, K., Ford, B., Sivakumar, S., and P.
              Srisuresh, "NAT Behavioral Requirements for TCP", BCP 142,
              RFC 5382, October 2008.

   [RFC4145] Yon, D. and G. Camarillo, "TCP-Based Media Transport in
              the Session Description Protocol (SDP)", RFC 4145,
              September 2005.

   [RFC4571] Lazzaro, J., "Framing Real-time Transport Protocol (RTP)
              and RTP Control Protocol (RTCP) Packets over Connection-
              Oriented Transport", RFC 4571, July 2006.

   [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

   [RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness
              Requirements for Security", BCP 106, RFC 4086, June 2005.

   [RFC5054] Taylor, D., Wu, T., Mavrogiannopoulos, N., and T. Perrin,
              "Using the Secure Remote Password (SRP) Protocol for TLS
              Authentication", RFC 5054, November 2007.

   [RFC3280] Housley, R., Polk, W., Ford, W., and D. Solo, "Internet
              X.509 Public Key Infrastructure Certificate and
              Certificate Revocation List (CRL) Profile", RFC 3280,
              April 2002.

   [I-D.matthews-p2psip-bootstrap-mechanisms]
              Cooper, E., "Bootstrap Mechanisms for P2PSIP",
              draft-matthews-p2psip-bootstrap-mechanisms-00 (work in
              progress), February 2007.

   [I-D.garcia-p2psip-dns-sd-bootstrapping]
              Garcia, G., "P2PSIP bootstrapping using DNS-SD",
              draft-garcia-p2psip-dns-sd-bootstrapping-00 (work in
              progress), October 2007.

   [I-D.pascual-p2psip-clients]
              Pascual, V., Matuszewski, M., Shim, E., Zhang, H., and S.
              Yongchao, "P2PSIP Clients",
              draft-pascual-p2psip-clients-01 (work in progress),
              February 2008.

   [RFC4787] Audet, F. and C. Jennings, "Network Address Translation
              (NAT) Behavioral Requirements for Unicast UDP", BCP 127,
              RFC 4787, January 2007.

   [I-D.jiang-p2psip-sep]
              Jiang, X. and H. Zhang, "Service Extensible P2P Peer
              Protocol", draft-jiang-p2psip-sep-01 (work in progress),
              February 2008.

   [I-D.zheng-p2psip-diagnose]
              Yongchao, S., Zhang, H., and X. Jiang, "Diagnose P2PSIP
              Overlay Network Failures", draft-zheng-p2psip-diagnose-03
              (work in progress), November 2008.

   [I-D.hardie-p2poverlay-pointers]
              Hardie, T., "Mechanisms for use in pointing to overlay
              networks, nodes, or resources",
              draft-hardie-p2poverlay-pointers-00 (work in progress),
              January 2008.

   [I-D.ietf-p2psip-sip]
              Jennings, C., Lowekamp, B., Rescorla, E., Baset, S., and
              H. Schulzrinne, "A SIP Usage for RELOAD",
              draft-ietf-p2psip-sip-00 (work in progress), October 2008.

   [Sybil]   Douceur, J., "The Sybil Attack", IPTPS 02, March 2002.

   [Eclipse] Singh, A., Ngan, T., Druschel, T., and D. Wallach,
              "Eclipse Attacks on Overlay Networks: Threats and
              Defenses", INFOCOM 2006, April 2006.

   [non-transitive-dhts-worlds05]

            Freedman, M., Lakshminarayanan, K., Rhea, S., and I.
            Stoica, "Non-Transitive Connectivity and DHTs",
             WORLDS'05.

   [lookups-churn-p2p06]
            Wu, D., Tian, Y., and K. Ng, "Analytical Study on
            Improving DHT Lookup Performance under Churn", IEEE
            P2P'06.

   [bryan-design-hotp2p08]
            Bryan, D., Lowekamp, B., and M. Zangrilli, "The Design of
            a Versatile, Secure P2PSIP Communications Architecture for
            the Public Internet", Hot-P2P'08.

   [opendht-sigcomm05]
            Rhea, S., Godfrey, B., Karp, B., Kubiatowicz, J.,
            Ratnasamy, S., Shenker, S., Stoica, I., and H. Yu,
            "OpenDHT: A Public DHT and its Uses", SIGCOMM'05.

   [Chord]   Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.,
            Kaashoek, M., Dabek, F., and H. Balakrishnan, "Chord: A
            Scalable Peer-to-peer Lookup Service for Internet
            Applications", IEEE/ACM Transactions on Networking Volume
            11, Issue 1, 17-32, Feb 2003.

   [vulnerabilities-acsac04]
            Srivatsa, M. and L. Liu, "Vulnerabilities and Security
            Threats in Structured Peer-to-Peer Systems: A Quantitative
            Analysis", ACSAC 2004.

   [handling-churn-usenix04]
            Rhea, S., Geels, D., Roscoe, T., and J. Kubiatowicz,
            "Handling Churn in a DHT", USENIX 2004.

   [minimizing-churn-sigcomm06]
            Godfrey, P., Shenker, S., and I. Stoica, "Minimizing Churn
            in Distributed Systems", SIGCOMM 2006.

Appendix A. Change Log

A.1. Changes since draft-ietf-p2psip-reload-00

   o Split base protocol from combined draft into new draft.
   o Update architecture discussion to address concerns raised about
      clarity of roles.

   o Moved extensive discussion of routing and client behaviors to
      appendix.
   o Split Ping into Ping and Probe
   o Added AttachLite to provide way to implement ICE-Lite
   o added Stat call for retrieving meta-data
   o added discussion of periodic vs reactive recovery issue
   o changed finger table stabilization to prefer long-lived over best-
      match
   o removed mDNS discovery method
   o updated IANA considerations to be more complete
   o changed error codes from http-based

A.2. Changes since draft-ietf-p2psip-base-00

   o removed TUNNEL method
   o allow implementations more flexibility in picking finger table
      entry and revise random range
   o decouple overlay configuration from enrollment server
   o add error for data too large
   o change architecture to overlay perspective from previous revision
      and update terminology in document to match

Appendix B. Routing Alternatives

   Significant discussion has been focused on the selection of a routing
   algorithm for P2PSIP. This section discusses the motivations for
   selection of symmetric recursive routing for RELOAD and describes the
   extensions that would be required to support additional routing
   algorithms.

B.1. Iterative vs Recursive

   Iterative routing has a number of advantages. It is easier to debug,
   consumes fewer resources on intermediate peers, and allows the
   querying peer to identify and route around misbehaving peers
   [non-transitive-dhts-worlds05]. However, in the presence of NATs
   iterative routing is intolerably expensive because a new connection
   must be established for each hop (using ICE) [bryan-design-hotp2p08].

   Iterative routing is supported through the Route_Query mechanism and
   is primarily intended for debugging. It is also allows the querying
   peer to evaluate the routing decisions made by the peers at each hop,
   consider alternatives, and perhaps detect at what point the
   forwarding path fails.

B.2. **Symmetric vs Forward response**

   An alternative to the symmetric recursive routing method used by
   RELOAD is Forward-Only routing, where the response is routed to the
   requester as if it is a new message initiating by the responder (in
   the previous example, Z sends the response to A as if it were sending
   a request). Forward-only routing requires no state in either the
   message or intermediate peers.

   The drawback of forward-only routing is that it does not work when
   the overlay is unstable. For example, if A is in the process of
   joining the overlay and is sending a Join request to Z, it is not yet
   reachable via forward routing. Even if it is established in the
   overlay, if network failures produce temporary instability, A may not
   be reachable (and may be trying to stabilize its network connectivity
   via Attach messages).

   Furthermore, forward-only responses are less likely to reach the
   querying peer than symmetric recursive because the forward path is
   more likely to have a failed peer than the request path (which was
   just tested to route the request) [non-transitive-dhts-worlds05].

   An extension to RELOAD that supports forward-only routing but relies
   on symmetric responses as a fallback would be possible, but due to
   the complexities of determining when to use forward-only and when to
   fallback to symmetric, we have chosen not to include it as an option
   at this point.

B.3. **Direct Response**

   Another routing option is Direct Response routing, in which the
   response is returned directly to the querying node. In the previous
   example, if A encodes its IP address in the request, then Z can
   simply deliver the response directly to A. In the absence of NATs or
   other connectivity issues, this is the optimal routing technique.

   The challenge of implementing direct response is the presence of
   NATs. There are a number of complexities that must be addressed. In
   this discussion, we will continue our assumption that A issued the
   request and Z is generating the response.

   o The IP address listed by A may be unreachable, either due to NAT
      or firewall rules. Therefore, a direct response technique must
      fallback to symmetric response [non-transitive-dhts-worlds05].
      The hop-by-hop ACKs used by RELOAD allow Z to determine when A has
      received the message (and the TLS negotiation will provide earlier
      confirmation that A is reachable), but this fallback requires a
      timeout that will increase the response latency whenever A is not

       reachable from Z.
   o Whenever A is behind a NAT it will have multiple candidate IP
      addresses, each of which must be advertised to ensure
      connectivity, therefore Z will need to attempt multiple
      connections to deliver the response.
   o One (or all) of A's candidate addresses may route from Z to a
      different device on the Internet. In the worst case these nodes
      may actually be running RELOAD on the same port. Therefore,
      establishing a secure connection to authenticate A before
      delivering the response is absolutely necessary. This step
      diminishes the efficiency of direct response because multiple
      roundtrips are required before the message can be delivered.
   o If A is behind a NAT and does not have a connection already
      established with Z, there are only two ways the direct response
      will work. The first is that A and Z are both behind the same
      NAT, in which case the NAT is not involved. In the more common
      case, when Z is outside A's NAT, the response will only be
      received if A's NAT implements endpoint-independent filtering. As
      the choice of filtering mode conflates application transparency
      with security [RFC4787], and no clear recommendation is available,
      the prevalence of this feature in future devices remains unclear.

   An extension to RELOAD that supports direct response routing but
   relies on symmetric responses as a fallback would be possible, but
   due to the complexities of determining when to use direct response
   and when to fallback to symmetric, and the reduced performance for
   responses to peers behind restrictive NATs, we have chosen not to
   include it as an option at this point.

B.4. Relay Peers

   SEP [I-D.jiang-p2psip-sep] has proposed implementing a form of direct
   response by having A identify a peer, Q, that will be directly
   reachable by any other peer. A uses Attach to establish a connection
   with Q and advertises Q's IP address in the request sent to Z. Z
   sends the response to Q, which relays it to A. This then reduces the
   latency to two hops, plus Z negotiating a secure connection to Q.

   This technique relies on the relative population of nodes such as A
   that require relay peers and peers such as Q that are capable of
   serving as a relay peer. It also requires nodes to be able to
   identify which category they are in. This identification problem has
   turned out to be hard to solve and is still an open area of
   exploration.

   An extension to RELOAD that supports relay peers is possible, but due
   to the complexities of implementing such an alternative, we have not
   added such a feature to RELOAD at this point.

A concept similar to relay peers, essentially choosing a relay peer
at random, has previously been suggested to solve problems of
pairwise non-transitivity [non-transitive-dhts-worlds05], but
deterministic filtering provided by NATs make random relay peers no
more likely to work than the responding peer.

B.5. Symmetric Route Stability

A common concern about symmetric recursive routing has been that one
or more peers along the request path may fail before the response is
received. The significance of this problem essentially depends on
the response latency of the overlay. An overlay that produces slow
responses will be vulnerable to churn, whereas responses that are
delivered very quickly are vulnerable only to failures that occur
over that small interval.

The other aspect of this issue is whether the request itself can be
successfully delivered. Assuming typical connection maintenance
intervals, the time period between the last maintenance and the
request being sent will be orders of magnitude greater than the delay
between the request being forwarded and the response being received.
Therefore, if the path was stable enough to be available to route the
request, it is almost certainly going to remain available to route
the response.

An overlay that is unstable enough to suffer this type of failure
frequently is unlikely to be able to support reliable functionality
regardless of the routing mechanism. However, regardless of the
stability of the return path, studies show that in the event of high
churn, iterative routing is a better solution to ensure request
completion [lookups-churn-p2p06] [non-transitive-dhts-worlds05]

Finally, because RELOAD retries the end-to-end request, that retry
will address the issues of churn that remain.

Appendix C. Why Clients?

There are a wide variety of reasons a node may act as a client rather
than as a peer [I-D.pascual-p2psip-clients]. This section outlines
some of those scenarios and how the client's behavior changes based
on its capabilities.

C.1. Why Not Only Peers?

For a number of reasons, a particular node may be forced to act as a
client even though it is willing to act as a peer. These include:

   o The node does not have appropriate network connectivity, typically
      because it has a low-bandwidth network connection.
   o The node may not have sufficient resources, such as computing
      power, storage space, or battery power.
   o The overlay algorithm may dictate specific requirements for peer
      selection. These may include participation in the overlay to
      determine trustworthiness, control the number of peers in the
      overlay to reduce overly-long routing paths, or ensure minimum
      application uptime before a node can join as a peer.

   The ultimate criteria for a node to become a peer are determined by
   the overlay algorithm and specific deployment. A node acting as a
   client that has a full implementation of RELOAD and the appropriate
   overlay algorithm is capable of locating its responsible peer in the
   overlay and using CONNECT to establish a direct connection to that
   peer. In that way, it may elect to be reachable under either of the
   routing approaches listed above. Particularly for overlay algorithms
   that elect nodes to serve as peers based on trustworthiness or
   population, the overlay algorithm may require such a client to locate
   itself at a particular place in the overlay.

C.2. Clients as Application-Level Agents

   SIP defines an extensive protocol for registration and security
   between a client and its registrar/proxy server(s). Any SIP device
   can act as a client of a RELOAD-based P2PSIP overlay if it contacts a
   peer that implements the server-side functionality required by the
   SIP protocol. In this case, the peer would be acting as if it were
   the user's peer, and would need the appropriate credentials for that
   user.

   Application-level support for clients is defined by a usage. A usage
   offering support for application-level clients should specify how the
   security of the system is maintained when the data is moved between
   the application and RELOAD layers.

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 (editor)
    unaffiliated
    2790 Linden Ln
    Williamsburg, VA 23185
    USA

    Email: bbl@lowekamp.net

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

    Phone: +1 650 320-8549
    Email: ekr@networkresonance.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