P2PSIP
Internet-Draft
Intended status: Standards Track
Expires: January 8, 2012

C. Jennings
Cisco
B. Lowekamp, Ed.
Skype
E. Rescorla
RTFM, Inc.
S. Baset
H. Schulzrinne
Columbia University
July 7, 2011

REsource LOcation And Discovery (RELOAD) Base Protocol
draft-ietf-p2psip-base-16

Abstract

This specification 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.

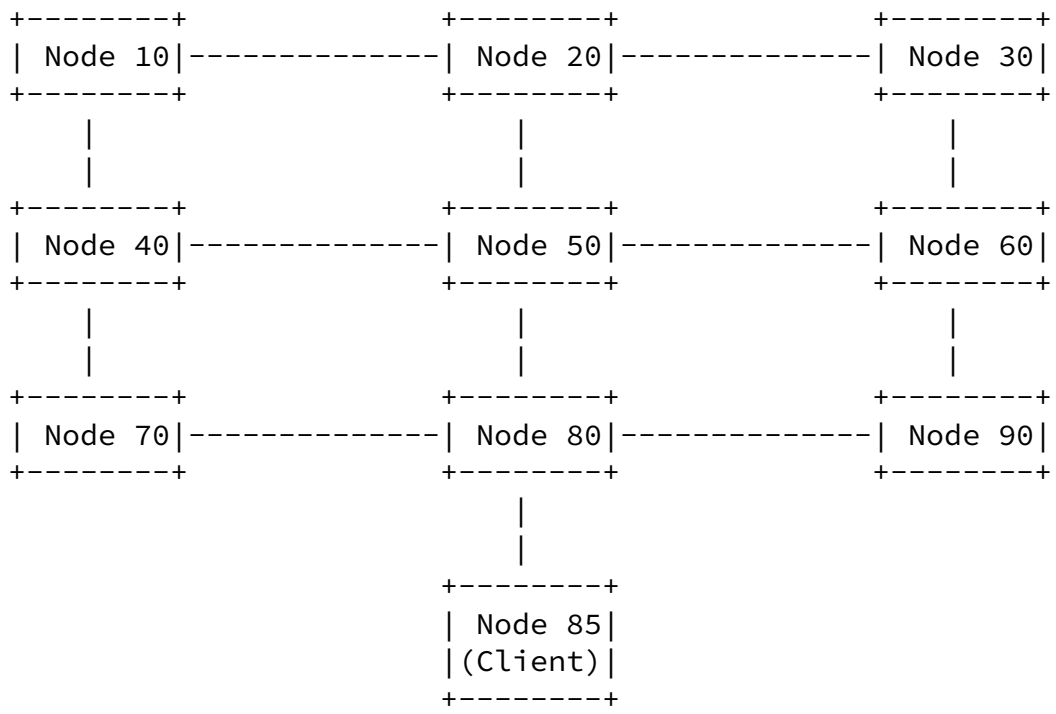
Pluggable Overlay Algorithms: RELOAD has been designed with an abstract interface to the overlay layer to simplify implementing a variety of structured (e.g., distributed hash tables) and unstructured overlay algorithms. This specification also defines how RELOAD is used with the Chord based DHT algorithm, which is mandatory to implement. Specifying a default "must implement" overlay algorithm promotes interoperability, while 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[I-D.ietf-p2psip-concepts] for more details. A RELOAD Overlay Instance consists of a set of nodes arranged in a 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.



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.

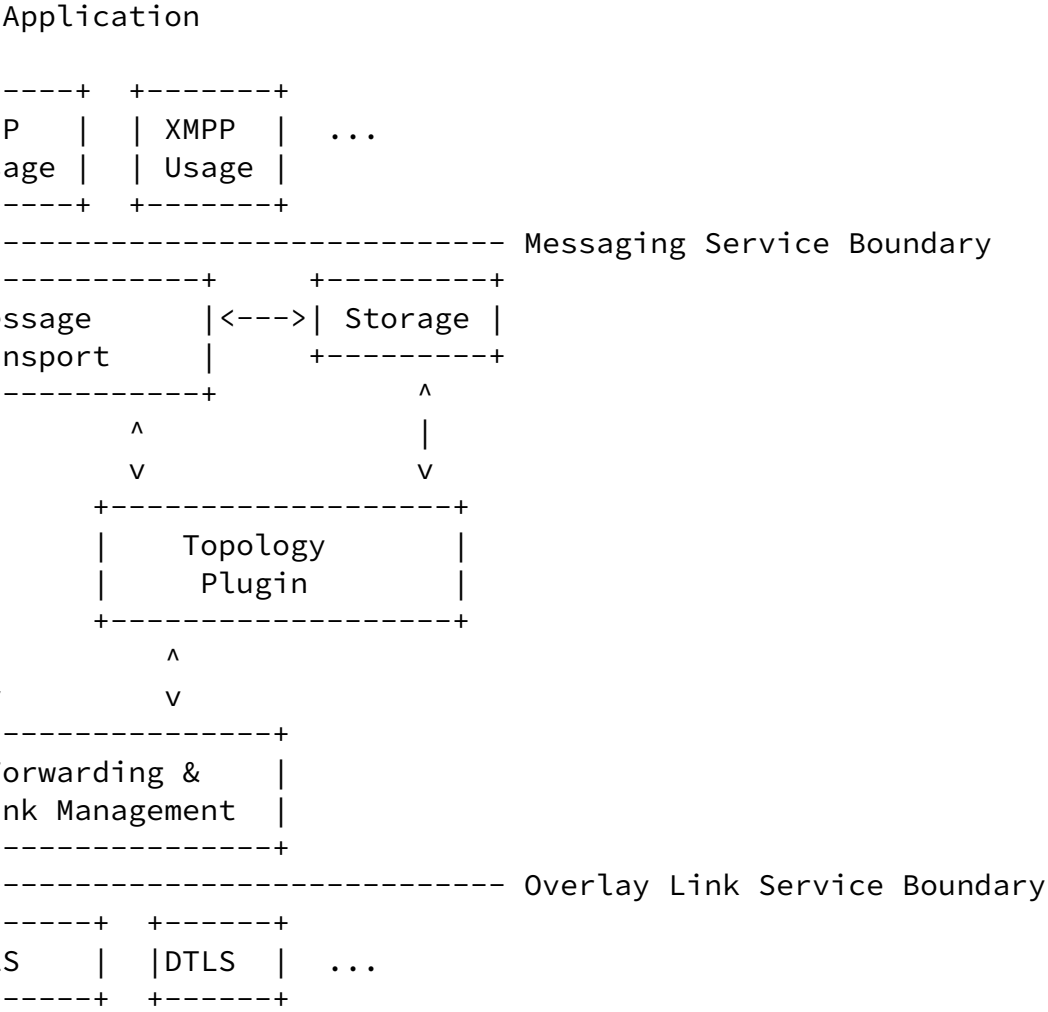
The RELOAD network is not only a messaging network. It is also a storage network. Records are stored under numeric addresses which occupy the same space as node identifiers. Peers are responsible for storing the data associated with some set of addresses as determined by their Node-ID. For instance, we might say that every peer 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. The following figure shows the layered RELOAD architecture.



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 Service.

Message Transport: Handles 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 component to send and receive messages.

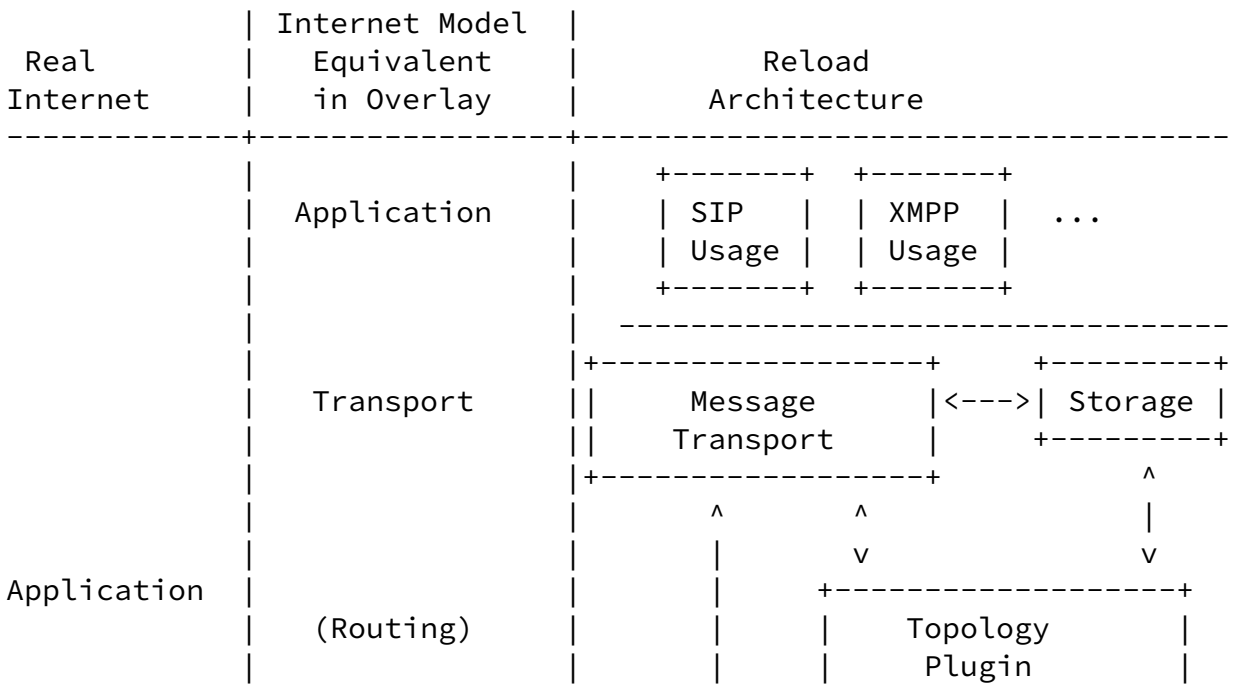
Topology Plugin: The Topology Plugin is responsible for implementing the specific overlay algorithm being used. It uses the Message Transport component to send and receive overlay management messages, to the Storage component to manage data replication, and directly to the Forwarding Layer to control 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.

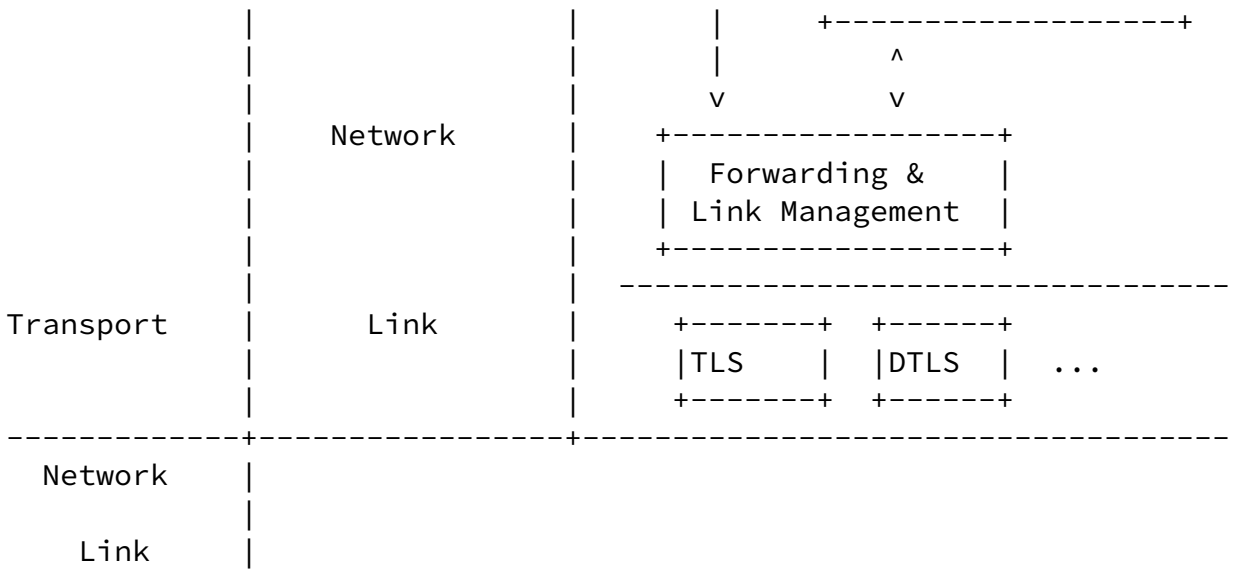
Forwarding and Link Management Layer: Stores and implements the routing table by providing packet forwarding services between nodes. It also handles establishing new links between nodes, including setting up connections across NATs using ICE.

Overlay Link Layer: Responsible for actually transporting traffic directly between nodes. Each such protocol includes the appropriate provisions for per-hop framing or hop-by-hop ACKs required by unreliable transports. TLS [RFC5246] and DTLS [RFC4347] are the currently defined "link layer" protocols used by RELOAD for hop-by-hop communication. New protocols can be defined, as described in Section 5.6.1 and Section 10.1. As this document defines only TLS and DTLS, we use those terms throughout the remainder of the document with the understanding that some

future specification may add new overlay link layers.

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





1.2.1. Usage Layer

The top layer, called the Usage Layer, has application usages, such as the SIP Registration Usage [I-D.ietf-p2psip-sip], that use the abstract Message Transport Service 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 softphone 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 specification.

1.2.2. Message Transport

The Message Transport component 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:

- Send a message to a given peer specified by Node-ID or to the peer responsible for a particular Resource-ID.
- Receive messages that other peers sent to a Node-ID or Resource-ID for which the receiving 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 Service 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 sending a response to the Message Transport for delivery to the requesting node. Typically, these messages will come from 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 in use. 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 for 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 message 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 generic interface that allows the topology plugin to 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.

This layer also utilizes a framing header to encapsulate messages as they are forwarding along each hop. This header aids reliability congestion control, flow control, etc. It has meaning only in the context of that individual link.

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, DTLS, or potentially some to be defined future protocol.
Message Level: Each RELOAD message must be signed.
Object Level: Stored objects must be signed by the creating 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 an optional 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 only. This feature is used together with certificate-based access control, not as a replacement for it. It is typically used when self-signed certificates are being used but would generally not be used when the certificates were all signed by an enrollment server.

1.4. Structure of This Document

The remainder of this document is structured as follows.

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

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.

Kind: A kind defines a particular type of data that can be stored in the overlay. Applications define new Kinds to store the data they use. Each Kind is identified with a unique integer called a Kind-ID.

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 (i.e. client or peer) when the text applies only to Clients or only to Peers.

Node-ID: A fixed-length value that uniquely identifies a node. Node-IDs of all 0s and all 1s 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 all 1s 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. Structured and unstructured P2P networks are described in [RFC5694]. 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 be variable lengths.

Connection Table: The set of nodes 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 that are in the connection table but may only route messages to other peers through peers that are in 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 data kind.

The term "maximum request lifetime" is the maximum time a request will wait for a response; it defaults to 15 seconds. The term "successor replacement hold-down time" is the amount of time to wait before starting replication when a new successor is found; it defaults to 30 seconds.

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:

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

Each node has a certificate [RFC5280] containing a Node-ID, which is unique within an overlay instance.

The certificate serves multiple purposes:

- 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.
- 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 uses this certificate so that a node connecting to it knows it is connected to the correct node (technically: a (D)TLS association with client authentication is formed.) In addition, the node can sign messages, thus providing integrity and authentication for messages which are sent from the node.
- 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 multiple certificate issuance models. The first is based on a central enrollment process which allocates a unique name and Node-ID and puts them in a certificate 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, 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.

The general principle here is that the security mechanisms (TLS and message signatures) are always used, even if the certificates are self-signed. This allows for a single set of code paths in the systems with the only difference being whether certificate verification is required to chain to a single root of trust.

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 as 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 such devices as "nodes."

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 certain 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.

In RELOAD, unlike some other designs, clients are not a first-class concept. From the perspective of a peer, a client is simply a node which has not yet sent any Updates or Joins. It might never do so (if it's a client) or it might eventually do so (if it's just a node that's taking a long time to join). The routing and storage rules

for RELOAD provide for correct behavior by peers regardless of whether other nodes attached to them are clients or peers. Of course, a client implementation must know that it intends to be a client, but this localizes complexity only to that node.

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

3.2.1. Client Routing

Clients may insert themselves in the overlay in two ways:

- 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 may require a TURN relay in cases where NATs or firewalls prevent a client from forming a direct connections with its responsible peer. Note that clients that choose this option need to process Update messages from the peer. Those updates can indicate that the peer no longer is responsible for the Client's Node-ID. The client would then need to form a connection to the appropriate peer. Failure to do so will result in the client no longer receiving messages.
- 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 using only its Node-ID. If the client is to receive incoming requests from other members of the overlay, the Destination List required to reach it must be learnable via other mechanisms, such as being stored in the overlay by a usage. A client connected this way using a certificate with only a single Node-ID MAY proceed to use the connection without performing an Attach. A client wishing to connect using this mechanism with a certificate with multiple Node-IDs MUST perform an Attach after using Ping to probe the Node-ID of the node to which it is connected.

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 act 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:

- Implement RELOAD's connection-management operations that are used to establish the connection with the peer.
- Implement RELOAD's data retrieval methods (with client functionality).
- Be able to calculate Resource-IDs used by the overlay.
- 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.

3.3. Routing

This section will discuss the requirements RELOAD's routing capabilities were designed to meet, then describe the routing features in the protocol, and then provide a brief overview of how they are used. Appendix A 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.

RELOAD's routing provides three mechanisms designed to assist in meeting these needs:

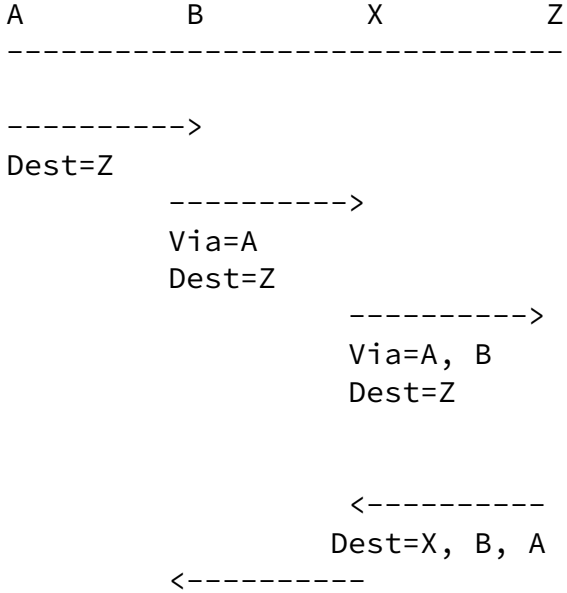
Destination Lists: While in principle it is possible to just inject a message into the overlay with a bare Node-ID as the destination, RELOAD provides a source routing capability in the form of "Destination Lists". A "Destination List provides a list of the nodes through which a message must flow.

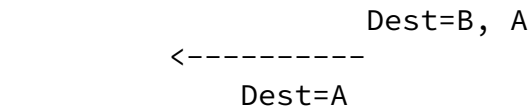
Via Lists: In order to allow responses to follow the same path as requests, each message also contains a "Via List", which is added to by each node a message traverses. This via list can then be inverted and used as a destination list for the response.

RouteQuery: The RouteQuery 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 recursive routing and then discuss its advantages in terms of the requirements discussed above.

Symmetric recursive routing requires that a request message follow a path through the overlay to the destination: 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:

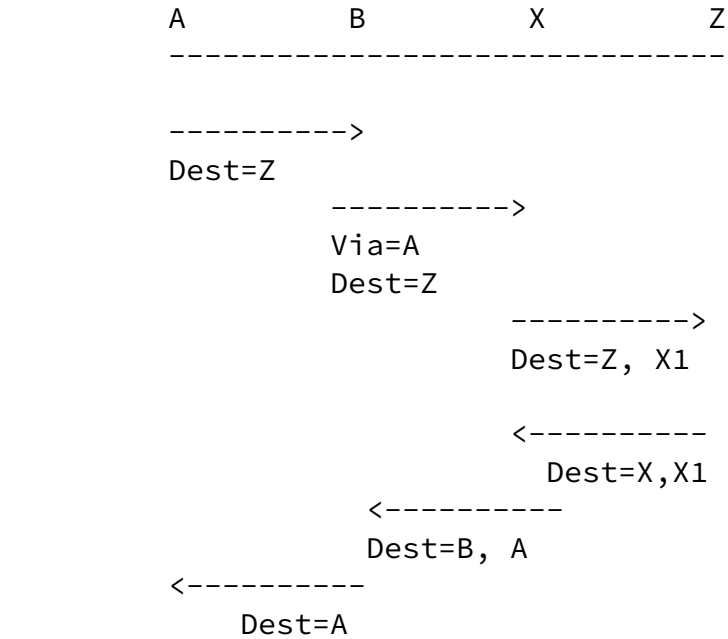




Note that the preceding Figure does not indicate whether A is a client or peer: A forwards its request to B and the response is

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 to be stored on intermediate peers but saves a small amount of bandwidth and reduces the need for modifying the message en route. Selection of this mode of operation is a choice for the individual peer; the techniques are interoperable even on a single message. The figure below shows B using full via lists but X truncating them to X1 and saving the state internally.



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 RouteQuery method, which requests this behavior. Note that iterative routing is selected only by the initiating node.

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 [RFC5245] 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 are two cases where Attach is not used. The first is 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" typically need to be publicly accessible so that new peers can directly connect to them. Section 10 contains more detail on this. The second case is when a client node connects to a node at an arbitrary IP address, rather than to its responsible peer, as described in the second bullet point of Section 3.2.1.

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 specification 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 and a set of credentials which match that Node-ID. When an enrollment server is used that Node-ID will be in the certificate the node 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:

- Forming connections to some other peers.
- Acquiring the data values this peer is responsible for storing.
- 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 to form a connection to some "bootstrap node". Because this is the first connection the peer makes, these nodes must have public IP addresses so that they 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 a peer has connected to a bootstrap node, it then needs to take up its appropriate place in the overlay. This requires two major operations:

- Forming connections to other peers in the overlay to populate its Routing Table.
- 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 other 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.

Note that clients do not perform either of these operations.

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 node 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.

If a node already has the valid configuration document that it received by some out of band method, this step can be skipped.

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

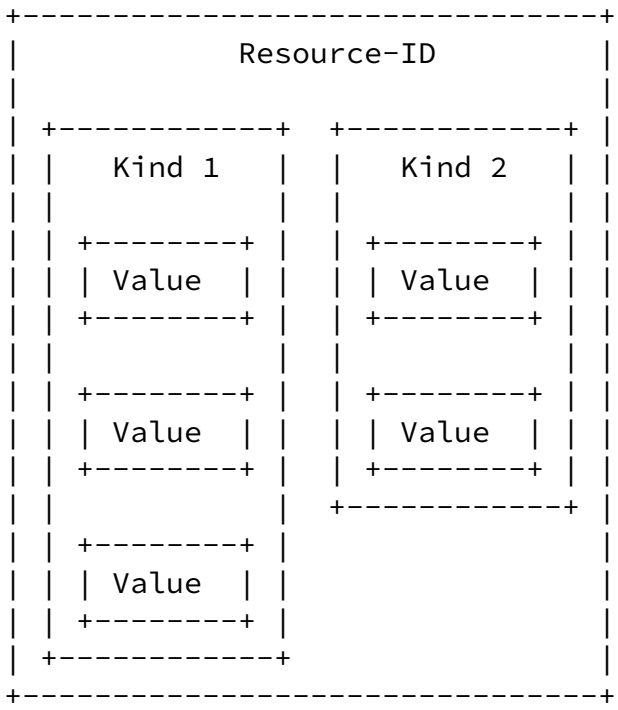
RELOAD is not intended to be used alone, but rather as a substrate for other applications. These applications can use RELOAD for a variety of purposes:

- To store data in the overlay and retrieve data stored by other nodes.
- As a discovery mechanism for services such as TURN.
- To form direct connections which can be used to transmit application-level messages without using the overlay.

This section provides an overview of these services.

4.1. Data Storage

RELOAD provides operations to Store and Fetch 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:



Each kind is identified by a Kind-ID, which is a code point either assigned by IANA or allocated out of a private range. 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 rule is used for all the kinds defined in this specification. Thus, only a user with a certificate for "alice@example.org" could write to that location in the 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 undetectably 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 limit size imbalance of various kinds.

4.1.2. 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 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. 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 needs to specify several things:

- Registers Kind-ID code points for any kinds that the Usage defines.
- Defines the data structure for each of the kinds.
- Defines access control rules for each of the kinds.
- Defines how the Resource Name is formed that is hashed to form the Resource-ID where each kind is stored.
- 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 as if 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, write to the same location, and have clock skew, 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.

4.3. Service Discovery

RELOAD does not currently define a generic service discovery algorithm as part of the base protocol, although a simplistic TURN-specific discovery mechanism is provided. A variety of service discovery algorithms can be implemented as extensions to the base protocol, such as the service discovery algorithm ReDIR [opendht-sigcomm05] or [I-D.ietf-p2psip-service-discovery].

4.4. 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, it is the overlay itself - rather than an external authority such as DNS - which is used to establish a connection. RELOAD provides connectivity to applications using the AppAttach method. For example, if a P2PSIP node wishes to establish a SIP dialog with another P2PSIP node, it will use AppAttach 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.

5. Overlay Management Protocol

This section defines the basic protocols used to create, maintain, and use the RELOAD overlay network. We start by defining the basic concept of how message destinations are interpreted when routing messages. We then describe the symmetric recursive routing model, which is RELOAD's default routing algorithm. We then define the message structure and then finally define the messages used to join and maintain the overlay.

5.1. 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 local policy can override this and cause the messages to be 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:

- The first entry on the destination list is an ID for which the peer is responsible. A peer is always responsible for the wildcard Node-ID. Handling of this case is described in Section 5.1.1.
- The first entry on the destination list is an ID for which another peer is responsible. Handling of this case is described in Section 5.1.2.
- The first entry on the destination list is a private ID that is being used for destination list compression. Handling of this case is described in Section 5.1.3. Note that private IDs can be distinguished from Node-IDs and Resource-IDs on the wire as described in Section 5.3.2.2).

These cases are handled as discussed below.

5.1.1. Responsible ID

If the first entry on the destination list is an ID for which the peer is responsible, there are several sub-cases to consider.

- 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 equals this node's Node-ID, 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 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 before the message is further processed.
- o If the entry is the wildcard Node-ID, 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 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. Other ID

If neither of the other three 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, the peer consults the routing table to forward the message.

Any intermediate peer which forwards a RELOAD request 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:

- The peer MAY add an entry to the via list in the forwarding header that will enable it to determine the correct node.
- 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 peers 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. It MAY either be a compressed version of the original via list or an index into a state database containing

the original via list.

No matter what mechanism for storing via list state is used, 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.

Note that if an intermediate peer retains per-transaction state instead of modifying the via list, it needs some mechanism for timing out that state, otherwise its state database will grow without bound. Whatever algorithm is used, unless a FORWARD_CRITICAL forwarding option or overlay configuration option explicitly indicates this state is not needed, the state MUST be maintained for at least the value of the overlay reliability timer (3 seconds) and MAY be kept longer. Future extension, such as [I-D.jiang-p2psip-relay], may define mechanisms for determining when this state does not need to be retained.

None of the above mechanisms are required for responses, since there is no need to ensure that subsequent requests follow the same path.

To be precise on the responsibility of the intermediate node, suppose that an intermediate node, A, receives a message from node B with via list X-Y-Z. Node A MUST implement an algorithm that ensures that A returns a response to this request to node B with the destination list B-Z-Y-X, provided that the node to which A forwards the request follows the same contract. Node A normally learns the Node-ID B is using via an Attach, but a node using a certificate with a single Node-ID MAY elect to not send an Attach (see Section 3.2.1 bullet 2). If a node with a certificate with multiple Node-IDs attempts to route a message other than a Ping or Attach through a node without performing an Attach, the receiving node MUST reject the request with an Error_Forbidden error. The node MUST implement support for returning responses to a Ping or Attach request made by a joining node Attaching to its responsible peer.

5.1.3. Private ID

If the first entry in the destination list is a private id (e.g., a compressed via list), the peer MUST replace that entry with the original via list that it replaced and then re-examine the destination list to determine which of the three cases in Section 5.1 now applies.

5.2. Symmetric Recursive Routing

This Section defines RELOAD's symmetric recursive routing algorithm, which is the default algorithm used by nodes to route messages through the overlay. All implementations MUST implement this routing algorithm. An overlay may be configured to use alternative routing algorithms, and alternative routing algorithms may be selected on a per-message basis.

5.2.1. 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 Node-ID 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 5.6.3.

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.

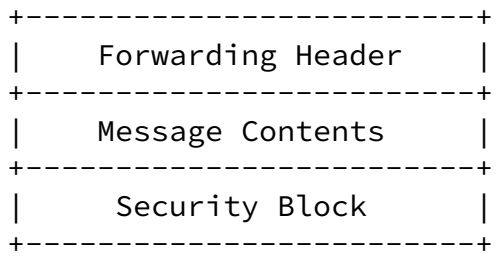
5.2.2. Response Origination

When a peer sends a response to a request using this routing algorithm, 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).

5.3. 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:



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 are opaque, however, they are interpreted by the higher layers.

Security Block: A security block containing certificates and a digital signature over the "Message Contents" section. Note that this signature can be computed without parsing the message contents. All messages MUST be signed by their originator.

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

5.3.1. Presentation Language

The structures defined in this document are defined using a C-like syntax based on the presentation language used to define TLS[RFC5246]. Advantages of this style include:

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

Several idiosyncrasies of this language are worth noting.

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

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

5.3.1.1. 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. The max value of 255 indicates this is represented as a single byte on the wire.

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

```
typedef opaque NodeId[NodeIdLength];
```

A NodeId is a fixed-length structure represented as a series of bytes, with the most significant byte first. The length is set on a per-overlay basis within the range of 16-20 bytes (128 to 160 bits). (See Section 10.1 for how NodeIdLength is set.) Note: the use of "typedef" here is an extension to the TLS language, but its meaning should be relatively obvious. Note the [size] syntax defines a fixed length element that does not include the length of the element in the on the wire encoding.

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

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

Like a NodeId, a ResourceId is an opaque string of bytes, but unlike NodeIds, ResourceIds are variable length, up to 254 bytes (2040 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. Note the < range > syntax defines a variable length element that does include the length of the element in the on the wire encoding. The number of bytes to encode the length on the wire is derived by range; i.e., it is the minimum

number of bytes which can encode the largest range value.

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


```
enum {reservedAddr(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
```

Unless a given structure that uses a select explicitly allows for unknown types in the select, any unknown type SHOULD be treated as an parsing error and the whole message discarded with no response.

5.3.2. Forwarding Header

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

```
    struct {
      uint32            relo_token;
      uint32            overlay;
```

```
        uint16              configuration_sequence;
        uint8               version;
        uint8               ttl;
        uint32              fragment;
        uint32              length;
        uint64              transaction_id;
        uint32              max_response_length;
        uint16              via_list_length;
        uint16              destination_list_length;
        uint16              options_length;
        Destination         via_list[via_list_length];
        Destination         destination_list
                              [destination_list_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. This field MUST contain the value 0xd2454c4f (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 [RFC3174] 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.

configuration_sequence: The sequence number of the configuration file.

version: The version of the RELOAD protocol being used. This is a fixed point integer between 0.1 and 25.4. This document describes version 0.1, with a value of 0x01. [[Note to RFC Editor: Please update this to version 1.0 with value of 0x0a and remove this note.]]

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, and a "Error_TTL_Exceeded" error should be generated. The initial value of the TTL SHOULD be 100 unless defined otherwise by the overlay configuration.

fragment: This field is used to handle fragmentation. The high order two bits are used to indicate the fragmentation status: If the high bit (0x80000000) is set, it indicates that the message is a fragment. If the next bit (0x40000000) is set, it indicates that this is the last fragment. The next six bits (0x20000000 to 0x01000000) are reserved and SHOULD be set to zero. The remainder of the field is used to indicate the fragment offset; see Section 5.7

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 allows receivers to disambiguate transactions which are otherwise identical. In order to provide a high probability that transaction IDs are unique, they MUST be randomly generated. Responses use the same Transaction ID as the request they correspond to. Transaction IDs are also used for fragment reassembly.

max_response_length: The maximum size in bytes of a response. Used by requesting nodes to avoid receiving (unexpected) very large responses. If this value is non-zero, responding peers MUST check that any response would not exceed it and if so generate an Error_Response_Too_Large value. This value SHOULD be set to zero for responses.

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.

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.

options: Contains a series of ForwardingOptions entries. See Section 5.3.2.3.

5.3.2.1. Processing Configuration Sequence Numbers

In order to be part of the overlay, a node MUST have a copy of the overlay configuration document. In order to allow for configuration document changes, each version of the configuration document has a sequence number which is monotonically increasing mod 65536. Because the sequence number may in principle wrap, greater than or less than are interpreted by modulo arithmetic as in TCP.

When a destination node receives a request, it MUST check that the configuration_sequence field is equal to its own configuration sequence number. If they do not match, it MUST generate an error, either Error_Config_Too_Old or Error_Config_Too_New. In addition, if the configuration file in the request is too old, it MUST generate a ConfigUpdate message to update the requesting node. This allows new configuration documents to propagate quickly throughout the system. The one exception to this rule is that if the configuration_sequence field is equal to 0xffff, and the message type is ConfigUpdate, then the message MUST be accepted regardless of the receiving node's configuration sequence number. Since 65535 is a special value, peers sending a new configuration when the configuration sequence is currently 65534 MUST set the configuration sequence number to 0 when they send out a new configuration.

5.3.2.2. Destination and Via Lists

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

```
  enum {reserved(0), node(1), resource(2), compressed(3),
            /* 128-255 not allowed */ (255) }
            DestinationType;

  select (destination_type) {
   case node:
          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;

  struct {
     uint16                compressed_id; /* top bit MUST be 1 */
  } Destination;
```

If a destination structure has its first bit set to 1, then it is a 16 bit integer. If the first bit is not set, then it is a structure starting with DestinationType. If it is a 16 bit integer, it is treated as if it were a full structure with a DestinationType of compressed and a compressed_id that was 2 bytes long with the value of the 16 bit integer. When the destination structure is not a 16 bit integer, it is the TLV structure with the following contents:

type
: The type of the DestinationData Payload Data Unit (PDU). This may be one of "node", "resource", or "compressed".

length
: The length of the destination_data.

destination_data
: 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:

node
: 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.

One possible encoding of the 16 bit integer version as an opaque identifier is to encode an index into a connection table. To avoid misrouting responses in the event a response is delayed and the connection table entry has changed, the identifier SHOULD be split between an index and a generation counter for that index. At startup, the generation counters should be initialized to random values. An implementation could use 12 bits for the connection table index and 3 bits for the generation counter. (Note that this does not suggest a 4096 entry connection table for every node, only the

ability to encode for a larger connection table.) When a connection table slot is used for a new connection, the generation counter is incremented (with wrapping). Connection table slots are used on a rotating basis to maximize the time interval between uses of the same slot for different connections. When routing a message to an entry in the destination list encoding a connection table entry, the node confirms that the generation counter matches the current generation counter of that index before forwarding the message. If it does not

match, the message is silently dropped.

5.3.2.3. Forwarding Options

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

```
 enum { reservedForwarding(0), (255) }
   ForwardingOptionsType;

 struct {
   ForwardingOptionsType     type;
   uint8                     flags;
   uint16                    length;
   select (type) {
         /* This type may be extended */
   } option;
 } ForwardingOption;
```

Each ForwardingOption consists of the following values:

type
: The type of the option. This structure allows for unknown options types.

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 Error_Unsupported_Forwarding_Option error response. If the DESTINATION_CRITICAL flag is set, any node that generates a response to the message but does not understand the forwarding option MUST reject the request with an Error_Unsupported_Forwarding_Option error response. If the RESPONSE_COPY flag is set, any node generating a response MUST copy the option from the request to the response except that the RESPONSE_COPY, FORWARD_CRITICAL and DESTINATION_CRITICAL flags must be cleared.

option
The option value.

5.3.3. Message Contents Format

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

```
enum { reservedMessagesExtension(0), (2^16-1) } MessageExtensionType;

struct {
  MessageExtensionType  type;
  Boolean               critical;
  opaque                extension_contents<0..2^32-1>;
} MessageExtension;

struct {
  uint16                 message_code;
  opaque                 message_body<0..2^32-1>;
  MessageExtensions      extensions<0..2^32-1>;
} 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

  The message codes are defined in Section 13.8

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.

extensions
: Extensions to the message. Currently no extensions are defined, but new extensions can be defined by the process described in Section 13.14.

All extensions have the following form:

type
: The extension type.

critical
: Whether this extension must be understood in order to process the message. If critical = True and the recipient does not understand the message, it MUST generate an Error_Unknown_Extension error. If critical = False, the recipient MAY choose to process the message even if it does not understand the extension.

extension_contents

The contents of the extension (extension-dependent).

5.3.3.1. 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 has 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
: An optional arbitrary byte string. Unless otherwise specified, this will be a UTF-8 text string providing further information about what went wrong.

The following error code values are defined. The numeric values for these are defined in Section 13.9.

Error_Forbidden: The requesting node 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 node MAY resend the request at a later time.

Error_Data_Too_Old: A store cannot be completed because the storage_time precedes the existing value.

Error_Data_Too_Old: A store cannot be completed because the storage_time precedes the existing value.

Error_Data_Too_Large: A store cannot be completed because the requested object exceeds the size limits for that kind.

Error_Generation_Counter_Too_Low: A store cannot be completed because the generation counter precedes the existing value.

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 5.3.2.3.

Error_TTL_Exceeded: A peer receiving the request where the TTL got decremented to zero. See section Section 5.3.2.

Error_Message_Too_Large: A peer receiving the request that was too large. See section Section 5.6.

Error_Response_Too_Large: A peer would have generated a response

that is too large per the max_response_length field.

Error_Config_Too_Old: A destination peer received a request with a configuration sequence that's too old. See Section 5.3.2.1.

Error_Config_Too_New: A destination node received a request with a configuration sequence that's too new. See Section 5.3.2.1.

Error_Unknown_Kind: A destination node received a request with an unknown kind-id. See Section 6.4.1.2.

Error_In_Progress: An Attach is already in progress to this peer. See Section 5.5.1.2.

Error_Unknown_Extension: A destination node received a request with an unknown extension.

5.3.4. Security Block

The third part of a RELOAD message is the security block. The security block is represented by a SecurityBlock structure:

```
struct {
   CertificateType     type;
   opaque              certificate<0..2^16-1>;
} GenericCertificate;

struct {
   GenericCertificate certificates<0..2^16-1>;
   Signature          signature;
} SecurityBlock;
```

The contents of this structure are:

certificates
: A bucket of certificates.

signature

A signature over the message contents.

The certificates bucket SHOULD contain all the certificates necessary to verify every signature in both the message and the internal message objects. This is the only location in the message which contains certificates, thus allowing for only a single copy of each certificate to be sent. In systems which have some alternate certificate distribution mechanism, some certificates MAY be omitted. However, implementors should note that this creates the possibility that messages may not be immediately verifiable because certificates must first be retrieved.

Each certificate is represented by a GenericCertificate structure, which has the following contents:

type
: The type of the certificate, as defined in [RFC6091]. Only the use of X.509 certificates is defined in this draft.

certificate
: The encoded version of the certificate. For X.509 certificates, it is the DER form.

The signature is computed over the payload and parts of the 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 { reservedSignerIdentity(0),
            cert_hash(1), cert_hash_node_id(2),
            none(3)
            (255)} SignerIdentityType;

     struct {
       select (identity_type) {

         case cert_hash;
           HashAlgorithm      hash_alg;              // From TLS
           opaque             certificate_hash<0..2^8-1>;

         case cert_hash_node_id:
           HashAlgorithm      hash_alg;              // From TLS
           opaque             certificate_node_id_hash<0..2^8-1>;

         case none:
           /* empty */
         /* This structure may be extended with new types if necessary*/
       };
     } SignerIdentityValue;

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

     struct  {
        SignatureAndHashAlgorithm     algorithm;   // From TLS
        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 and HashAlgorithm registries. All implementations MUST support RSASSA-PKCS1-v1_5 [RFC3447] signatures with SHA-256 hashes.

identity

The identity used to form the signature.

signature_value
The value of the signature.

There are two permitted identity formats, one for a certificate with only one node-id and one for a certificate with multiple node-ids. In the first case, the cert_hash type SHOULD be used. The hash_alg field is used to indicate the algorithm used to produce the hash. The certificate_hash contains the hash of the certificate object (i.e., the DER-encoded certificate).

In the second case, the cert_hash_node_id type MUST be used. The hash_alg is as in cert_hash but the cert_hash_node_id is computed over the NodeId used to sign concatenated with the certificate. I.e., H(NodeID || certificate). The NodeId is represented without any framing or length fields, as simple raw bytes. This is safe because NodeIds are fixed-length for a given overlay.

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.

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

All RELOAD messages MUST be signed. Upon receipt, the receiving node MUST verify the signature and the authorizing certificate. This check provides a minimal level of assurance that the sending node is a valid part of the overlay as well as cryptographic authentication of the sending node. In addition, responses MUST be checked as

follows:

1. The response to a message sent to a specific Node-ID MUST have been sent by that Node-ID.
2. The response to a message sent to a Resource-Id MUST have been sent by a Node-ID which is as close to or closer to the target Resource-Id than any node in the requesting node's neighbor table.

The second condition serves as a primitive check for responses from wildly wrong nodes but is not a complete check. Note that in periods of churn, it is possible for the requesting node to obtain a closer neighbor while the request is outstanding. This will cause the

response to be rejected and the request to be retransmitted.

In addition, some methods (especially Store) have additional authentication requirements, which are described in the sections covering those methods.

5.4. 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.

5.4.1. Topology Plugin Requirements

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

- Joining procedures, including the contents of the Join message.
- Stabilization procedures, including the contents of the Update message, the frequency of topology probes and keepalives, and the mechanism used to detect when peers have disconnected.
- Exit procedures, including the contents of the Leave message.
- The length of the Resource-IDs. For DHTs, the hash algorithm to compute the hash of an identifier.
- The procedures that peers use to route messages.

o The replication strategy used to ensure data redundancy.

All overlay algorithms MUST specify maintenance procedures that send Updates to clients and peers that have established connections to the peer responsible for a particular ID when the responsibility for that ID changes. Because tracking this information is difficult, overlay algorithms MAY simply specify that an Update is sent to all members of the Connection Table whenever the range of IDs for which the peer is responsible changes.

5.4.2. Methods and types for use by topology plugins

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

5.4.2.1. Join

A new peer (but one that 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. Receivers of the JoinReq MUST verify that the joining_peer_id field matches the Node-ID used to sign the message and if not MUST reject the message with an Error_Forbidden error.

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 that 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.

5.4.2.2. 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.

```
   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. Receivers of the LeaveReq MUST verify that the leaving_peer_id field matches the Node-ID used to sign the message and if not MUST reject the message with an Error_Forbidden error.

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.

5.4.2.3. 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. In general, peers send Update messages to all their adjacencies whenever they detect a topology shift.

When a peer receives an Attach request with the send_update flag set to "true" (Section 5.4.2.4.1, it MUST send an Update message back to the sender of the Attach request after the completion of the corresponding ICE check and TLS connection. Note that the sender of a such Attach request may not have joined the overlay yet.

When a peer detects through an Update that it is no longer responsible for any data value it is storing, it MUST attempt to Store a copy to the correct node unless it knows the newly responsible node already has a copy of the data. This prevents data loss during large-scale topology shifts such as the merging of partitioned overlays.

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

5.4.2.4. RouteQuery

The RouteQuery 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 for the node that the receiving peer would next route to in order to get to X. A RouteQuery can also request that the receiving peer initiate an Update request to transfer the receiving peer's 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 use the Attach method to attach to that peer(s), and repeat 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.

5.4.2.4.1. Request Definition

A RouteQueryReq message indicates the peer or resource that the requesting node is interested in. It also contains a "send_update" option allowing the requesting node 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.

5.4.2.4.2. Response Definition

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

5.4.2.5. Probe

Probe provides primitive "exploration" services: it allows node to determine which resources another node is responsible for; and it allows some discovery services using multicast, anycast, or

broadcast. 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.

5.4.2.5.1. 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 { reservedProbeInformation(0), responsible_set(1),
       num_resources(2), uptime(3),  (255)}
     ProbeInformationType;

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

The 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.

uptime
: indicates that the peer should Respond with how long the peer has been up in seconds.

5.4.2.5.2. Response Definition

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

```
        struct {
          select (type) {
            case responsible_set:
              uint32             responsible_ppb;

            case num_resources:
              uint32             num_resources;

            case uptime:
              uint32              uptime;
            /* This type may be extended */

          };
        } ProbeInformationData;

        struct {
          ProbeInformationType    type;
          uint8                   length;
          ProbeInformationData    value;
        } ProbeInformation;

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

A ProbeAns message contains a sequence of ProbeInformation structures. Each has a "length" indicating the length of the following value field. This structure allows for unknown option types.

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. For the type "uptime" it is the number of seconds the peer has been up.

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

5.5. 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 RELOAD connections between nodes using ICE for NAT traversal. 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. Attach also allows two nodes to connect via No-ICE instead of full ICE.

AppAttach: used to form application layer connections between nodes.

Ping: is a simple request/response which is used to verify connectivity of the target peer.

5.5.1. Attach

A node sends an Attach request when it wishes to establish a direct TCP or UDP connection to another node for the purpose of sending RELOAD messages. A client that can establish a connection directly need not send an attach as described in the second bullet of Section 3.2.1

As described in Section 5.1, an Attach may be routed to either a Node-ID or to a Resource-ID. An Attach routed to a specific Node-ID will fail if that node is not reached. An Attach routed to a Resource-ID will establish a connection with the peer currently responsible for that Resource-ID, which may be useful in establishing a direct connection to the responsible peer for use with frequent or large resource updates.

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 Join and Update), to prevent half-open states

where a node has started to form connections but is not really ready to act as a peer. Thus, clients (unlike peers) can simply Attach without sending Join or Update.

5.5.1.1. Request Definition

An Attach request message contains the requesting node ICE connection parameters formatted into a binary structure.


```
enum { reservedOverlayLink(0), DTLS-UDP-SR(1),
        DTLS-UDP-SR-NO-ICE(3), TLS-TCP-FH-NO-ICE(4),
        (255) } OverlayLinkType;

enum { reservedCand(0), host(1), srflx(2), prflx(3), relay(4),
     (255) } CandType;

struct {
  opaque                name<0..2^16-1>;
  opaque                value<0..2^16-1>;
} IceExtension;

struct {
  IpAddressPort         addr_port;
  OverlayLinkType       overlay_link;
  opaque                foundation<0..255>;
  uint32                priority;
  CandType              type;
  select (type){
    case host:
      ;           /* Nothing */
    case srflx:
    case prflx:
    case relay:
      IpAddressPort     rel_addr_port;
  };
  IceExtension          extensions<0..2^16-1>;
} IceCandidate;
```

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

The values contained in AttachReqAns are:

ufrag
: The username fragment (from ICE).

password

: The ICE password.

role
: An active/passive/actpass attribute from RFC 4145 [RFC4145]. This value MUST be 'passive' for the offerer (the peer sending the Attach request) and 'active' for the answerer (the peer sending the Attach response).

candidates
: One or more ICE candidate values, as described below.

send_update
: Has the same meaning as the send_update field in RouteQueryReq.

Each ICE candidate is represented as an IceCandidate structure, which is a direct translation of the information from the ICE string structures, with the exception of the component ID. Since there is only one component, it is always 1, and thus left out of the PDU. The remaining values are specified as follows:

addr_port
: corresponds to the connection-address and port productions.

overlay_link
corresponds to the OverlayLinkType production, Overlay Link protocols used with No-ICE MUST specify "No-ICE" in their description. Future overlay link values can be added be defining new OverlayLinkType values in the IANA registry in Section 13.10. Future extensions to the encapsulation or framing that provide for backward compatibility with that specified by a previously defined OverlayLinkType values MUST use that previous value. OverlayLinkType protocols are defined in Section 5.6
A single AttachReqAns MUST NOT include both candidates whose OverlayLinkType protocols use ICE (the default) and candidates that specify "No-ICE".

foundation
corresponds to the foundation production.

priority
corresponds to the priority production.

type
corresponds to the cand-type production.

rel_addr_port
corresponds to the rel-addr and rel-port productions. Only present for type "relay".

extensions
ICE extensions. The name and value fields correspond to binary translations of the equivalent fields in the ICE extensions.

These values should be generated using the procedures described in Section 5.5.1.3.

5.5.1.2. Response Definition

If a peer receives an Attach request, it MUST determine how to process the request as follows:

- o If it has not initiated an Attach request to the originating peer of this Attach request, it MUST process this request and SHOULD generate its own response with an AttachReqAns. It should then begin ICE checks.
- o If it has already sent an Attach request to and received the response from the originating peer of this Attach request, and as a as a result, an ICE check and TLS connection is in progress, then it SHOULD generate an Error_In_Progress error instead of an AttachReqAns.
- o If it has already sent an Attach request to but not yet received the response from the originating peer of this Attach request, it SHOULD apply the following tie-breaker heuristic to determine how to handle this Attach request and the incomplete Attach request it has sent out:
  - * If the peer's own Node-ID is smaller when compared as big-endian unsigned integers, it MUST cancel its own incomplete Attach request. It MUST then process this Attach request, generate an AttachReqAns response, and proceed with the corresponding ICE check.
  - * If the peer's own Node-ID is larger when compared as big-endien unsigned integers, it MUST generate an Error_In_Progress error to this Attach request, then proceed to wait for and complete the Attach and the corresponding ICE check it has originated.
- o If the peer is overloaded or detects some other kind of error, it MAY generate an error instead of an AttachReqAns.

When a peer receives an Attach response, it SHOULD parse the response and begin its own ICE checks.

5.5.1.3. Using ICE With RELOAD

This section describes the profile of ICE that is used with RELOAD. RELOAD implementations MUST implement full ICE.

In ICE as defined by [RFC5245], 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:

- Only a single media stream is supported.
- In this case, the "stream" refers not to RTP or other types of media, but rather to a connection for RELOAD itself or other application-layer protocols such as SIP.
- 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 [RFC5245] with the changes and additional procedures described in the subsections below.

5.5.1.4. 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 capable of responding to STUN Binding requests [RFC5389]. Accordingly, any peer that a node knows about can be used like 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 nodes. These nodes make an initial list of STUN servers. However, as the peer forms connections with additional peers, it builds more peers it can use like 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 learn 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 MUST be selected unless there are no others servers in the group in which case some other peer MAY be used.

5.5.1.5. Gathering Candidates

When a node wishes to establish a connection for the purposes of RELOAD signaling or application signaling, it follows the process of gathering candidates as described in Section 4 of ICE [RFC5245]. RELOAD utilizes a single component. Consequently, gathering for these "streams" requires a single component. In the case where a node has not yet found a TURN server, the agent would not include a relayed candidate.

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 5.5.1.4 and Section 8.

The default candidate selection described in Section 4.1.4 of ICE is ignored; defaults are not signaled or utilized by RELOAD.

An alternative to using the full ICE supported by the Attach request is to use No-ICE mechanism by providing candidates with "No-ICE" Overlay Link protocols. Configuration for the overlay indicates whether or not these Overlay Link protocols can be used. An overlay MUST be either all ICE or all No-ICE.

No-ICE 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.

5.5.1.6. Prioritizing Candidates

However, standardization of additional protocols for use with ICE is expected, including TCP[I-D.ietf-mmusic-ice-tcp] and protocols such as SCTP and DCCP. UDP encapsulations for SCTP and DCCP would expand the available Overlay Link protocols available for RELOAD. When additional protocols are available, the following prioritization is RECOMMENDED:

- Highest priority is assigned to protocols that offer well-understood congestion and flow control without head of line blocking. For example, SCTP without message ordering, DCCP, or those protocols encapsulated using UDP.
- Second highest priority is assigned to protocols that offer well-understood congestion and flow control but have head of line blocking such as TCP.
- Lowest priority is assigned to protocols encapsulated over UDP that do not implement well-established congestion control algorithms. The DTLS/UDP with SR overlay link protocol is an example of such a protocol.

5.5.1.7. Encoding the Attach Message

Section 4.3 of ICE describes procedures for encoding the SDP for conveying RELOAD candidates. Instead of actually encoding an SDP, the candidate information (IP address and port and transport protocol, priority, foundation, 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 5.5.1 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.

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.

5.5.1.8. 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.

5.5.1.9. 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.

5.5.1.10. Full ICE

When the overlay uses ICE , connectivity checks and nominations are used as in regular ICE.

5.5.1.10.1. 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.

5.5.1.10.2. Concluding ICE

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

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

5.5.1.10.3. Media Keepalives

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

5.5.1.11. No-ICE

No-ICE is selected when either side has provided "no ICE" Overlay Link candidates. STUN is not used for connectivity checks when doing No-ICE; instead the DTLS or TLS handshake (or similar security layer of future overlay link protocols) forms the connectivity check. The certificate exchanged during the (D)TLS handshake must match the node that sent the AttachReqAns and if it does not, the connection MUST be closed.

5.5.1.12. 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.

5.5.1.13. Sending Media

The procedures of Section 11 of ICE apply to RELOAD as well. However, in this case, the "media" takes the form of application layer protocols (e.g. RELOAD) 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 server MUST request TLS client authentication. 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.

5.5.1.14. 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.

5.5.2. AppAttach

A node sends an AppAttach request when it wishes to establish a direct connection to another node for the purposes of sending application layer messages. AppAttach is nearly identical to Attach, except for the purpose of the connection: it is used to transport non-RELOAD "media". A separate request is used to avoid implementor confusion between the two methods (this was found to be a real

problem with initial implementations). The AppAttach request and its response contain an application attribute, which indicates what protocol is to be run over the connection.

5.5.2.1. Request Definition

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

```
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>;
} AppAttachReq;
```


The values contained in AppAttachReq and AppAttachAns are:

ufrag
: The username fragment (from ICE)

password
: The ICE password.

application
: A 16-bit application-id as defined in the Section 13.5. This number represents the IANA registered application that is going to send data on this connection. For SIP, this is 5060 or 5061.

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

candidates
: One or more ICE candidate values

The application using connection set up with this request is responsible for providing sufficiently frequent keep traffic for NAT and Firewall keep alive and for deciding when to close the connection.

5.5.2.2. Response Definition

If a peer receives an AppAttach request, it SHOULD process the request and generate its own response with a AppAttachAns. It should then begin ICE checks. When a peer receives an AppAttach response, it SHOULD parse the response and begin its own ICE checks. If the application ID is not supported, the peer MUST reply with an Error_Not_Found error.

```
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>;
} AppAttachAns;
```

The meaning of the fields is the same as in the AppAttachReq.

5.5.3. Ping

Ping is used to test connectivity along a path. A ping can be addressed to a specific Node-ID, to the peer controlling a given

location (by using a resource ID), or to the broadcast Node-ID (2^128-1).

5.5.3.1. Request Definition

```
struct {
  opaque<0..2^16-1> padding;
} PingReq
```

The Ping request is empty of meaningful contents. However, it may contain up to 65535 bytes of padding to facilitate the discovery of overlay maximum packet sizes.

5.5.3.2. Response Definition

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

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

A PingAns message contains the following elements:

response_id
: A randomly generated 64-bit response ID. This is used to distinguish Ping responses.

time
: The time when the Ping response was created represented in the same way as storage_time defined in Section 6.

5.5.4. ConfigUpdate

The ConfigUpdate method is used to push updated configuration data across the overlay. Whenever a node detects that another node has old configuration data, it MUST generate a ConfigUpdate request. The ConfigUpdate request allows updating of two kinds of data: the configuration data (Section 5.3.2.1) and kind information (Section 6.4.1.1).

5.5.4.1. Request Definition

```
enum { reservedConfigUpdate(0), config(1), kind(2), (255) }
     ConfigUpdateType;

typedef uint32           KindId;
typedef opaque           KindDescription<0..2^16-1>;

struct {
  ConfigUpdateType       type;
  uint32                 length;
```

```
        select (type) {
          case config:
                      opaque             config_data<0..2^24-1>;

          case kind:
                      KindDescription    kinds<0..2^24-1>;

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

The ConfigUpdateReq message contains the following elements:

type
: The type of the contents of the message. This structure allows for unknown content types.

length
: The length of the remainder of the message. This is included to preserve backward compatibility and is 32 bits instead of 24 to facilitate easy conversion between network and host byte order.

config_data (type==config)
: The contents of the configuration document.

kinds (type==kind)
: One or more XML kind-block productions (see Section 10.1). These MUST be encoded with UTF-8 and assume a default namespace of "urn:ietf:params:xml:ns:p2p:config-base".

5.5.4.2. Response Definition

```
      struct {
      } ConfigUpdateAns
```

If the ConfigUpdateReq is of type "config" it MUST only be processed if all the following are true:

- The sequence number in the document is greater than the current configuration sequence number.
- The configuration document is correctly digitally signed (see Section 10 for details on signatures.

Otherwise appropriate errors MUST be generated.

If the ConfigUpdateReq is of type "kind" it MUST only be processed if it is correctly digitally signed by an acceptable kind signer as specified in the configuration file. Details on kind-signer field in the configuration file is described in Section 10.1. In addition, if the kind update conflicts with an existing known kind (i.e., it is signed by a different signer), then it should be rejected with "Error_Forbidden". This should not happen in correctly functioning overlays.

If the update is acceptable, then the node MUST reconfigure itself to match the new information. This may include adding permissions for new kinds, deleting old kinds, or even, in extreme circumstances, exiting and reentering the overlay, if, for instance, the DHT algorithm has changed.

The response for ConfigUpdate is empty.

5.6. Overlay Link Layer

RELOAD can use multiple Overlay Link protocols to send its messages. Because ICE is used to establish connections (see Section 5.5.1.3), RELOAD nodes are able to detect which Overlay Link protocols are offered by other nodes and establish connections between them. Any link protocol needs to be able to establish a secure, authenticated connection and to provide data origin authentication and message integrity for individual data elements. RELOAD currently supports three Overlay Link protocols:

- DTLS [RFC4347] over UDP with Simple Reliability (SR)
- TLS [RFC5246] over TCP with Framing Header, No-ICE
- DTLS [RFC4347] over UDP with SR, No-ICE

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

If a peer receives a message that is larger than value of max-message-size defined in the overlay configuration, the peer SHOULD send an Error_Message_Too_Large error and then close the TLS or DTLS session from which the message was received. Note that this error can be sent and the session closed before receiving the complete

message. If the forwarding header is larger than the max-message-size, the receiver SHOULD close the TLS or DTLS session without sending an error.

The Framing Header (FH) is used to frame messages and provide timing when used on a reliable stream-based transport protocol. Simple Reliability (SR) makes use of the FH to provide congestion control and semi-reliability when using unreliable message-oriented transport protocols. We will first define each of these algorithms, then define overlay link protocols that use them.

Note: We expect future Overlay Link protocols to define replacements for all components of these protocols, including the framing header. These protocols have been chosen for simplicity of implementation and reasonable performance.

Note to implementers: There are inherent tradeoffs in utilizing short timeouts to determine when a link has failed. To balance the tradeoffs, an implementation should be able to quickly act to remove entries from the routing table when there is reason to suspect the link has failed. For example, in a Chord derived overlay algorithm, a closer finger table entry could be substituted for an entry in the finger table that has experienced a timeout. That entry can be restored if it proves to resume functioning, or replaced at some point in the future if necessary. End-to-end retransmissions will handle any lost messages, but only if the failing entries do not remain in the finger table for subsequent retransmissions.

5.6.1. Future Overlay Link Protocols

It is possible to define new link-layer protocols and apply them to a new overlay using the "overlay-link-protocol" configuration directive (see Section 10.1.). However, any new protocols MUST meet the following requirements.

Endpoint authentication When a node forms an association with another endpoint, it MUST be possible to cryptographically verify that the endpoint has a given Node-Id.

Traffic origin authentication and integrity When a node receives traffic from another endpoint, it MUST be possible to cryptographically verify that the traffic came from a given association and that it has not been modified in transit from the other endpoint in the association. The overlay link protocol MUST also provide replay prevention/detection.

Traffic confidentiality When a node sends traffic to another endpoint, it MUST NOT be possible for a third party not involved in the association to determine the contents of that traffic.

Any new overlay protocol MUST be defined via RFC 5226 Standards Action; see Section 13.11.

5.6.1.1. HIP

In a Host Identity Protocol Based Overlay Networking Environment (HIP BONE) [RFC6079] HIP [RFC5201] provides connection management (e.g., NAT traversal and mobility) and security for the overlay network. The P2PSIP Working Group has expressed interest in supporting a HIP-based link protocol. Such support would require specifying such details as:

- 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.
- How to carry the HIP I1 and I2 messages.
- How to carry RELOAD messages over HIP.

[I-D.ietf-hip-reload-instance] documents work in progress on using RELOAD with the HIP BONE.

5.6.1.2. ICE-TCP

The ICE-TCP draft [I-D.ietf-mmusic-ice-tcp] allows TCP to be supported as an Overlay Link protocol that can be added using ICE.

5.6.1.3. Message-oriented Transports

Modern message-oriented transports offer high performance, good congestion control, and avoid head of line blocking in case of lost data. These characteristics make them preferable as underlying transport protocols for RELOAD links. SCTP without message ordering and DCCP are two examples of such protocols. However, currently they are not well-supported by commonly available NATs, and specifications for ICE session establishment are not available.

5.6.1.4. Tunneled Transports

As of the time of this writing, there is significant interest in the IETF community in tunneling other transports over UDP, motivated by the situation that UDP is well-supported by modern NAT hardware, and similar performance can be achieved to native implementation. Currently SCTP, DCCP, and a generic tunneling extension are being

proposed for message-oriented protocols. Once ICE traversal has been specified for these tunneled protocols, they should be straightforward to support as overlay link protocols.

5.6.2. Framing Header

In order to support unreliable links and to allow for quick detection of link failures when using reliable end-to-end transports, 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. The same header is used for both reliable and unreliable transports for simplicity of implementation.

The definition of FramedMessage is:

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

struct {
  FramedMessageType       type;

  select (type) {
    case data:
      uint32              sequence;
      opaque              message<0..2^24-1>;

    case ack:
      uint32              ack_sequence;
      uint32              received;
  };
} FramedMessage;
```

The type field of the PDU is set to indicate whether the message is data or an acknowledgement.

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

sequence
: the sequence number. This increments by 1 for each framed message sent over this transport session.

message
: the message that is being transmitted.

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

When the receiver receives 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 if each of the previous 32 sequence numbers before this packet has been among the 32 packets most recently received on this connection. When a packet is received with a sequence number N, the receiver looks at the sequence number of the previously 32 packets received on this connection. Call the previously received packet number M. For each of the previous 32 packets, if the sequence number M is less than N but greater than

N-32, the N-M bit of the received bitmask is set to one; otherwise it is zero. Note that a bit being set to one indicates positively that a particular packet was received, but a bit being set to zero means only that it is unknown whether or not the packet has been received, because it might have been received before the 32 most recently received packets.

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

5.6.3. Simple Reliability

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. The basic principle is that each message, regardless of whether or not it carries a request or response, 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.

5.6.3.1. 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. In general, senders MAY implement any rate control scheme of their choice, provided that it is REQUIRED to be no more aggressive then TFRC[RFC5348].

The following section describes a simple, inefficient scheme that complies with this requirement. Another alternative would be TFRC-SP [RFC4828] and use the received bitmask to allow the sender to compute packet loss event rates.

5.6.3.1.1. Trivial Retransmission

A node SHOULD retransmit a message if it has not received an ACK after an interval of RTO ("Retransmission TimeOut"). The node MUST double the time to wait after each retransmission. In each retransmission, the sequence number is incremented.

The RTO is an estimate of the round-trip time (RTT). Implementations can use a static value for RTO or a dynamic estimate which will result in better performance. For implementations that use a static value, the default value for RTO is 500 ms. Nodes MAY use smaller values of RTO if it is known that all nodes are within the local network. The default RTO MAY be chosen larger, and this is RECOMMENDED if it is known in advance (such as on high latency access links) that the round-trip time is larger.

Implementations that use a dynamic estimate to compute the RTO MUST use the algorithm described in RFC 6298[RFC6298], with the exception that the value of RTO SHOULD NOT be rounded up to the nearest second but instead rounded up to the nearest millisecond. The RTT of a successful STUN transaction from the ICE stage is used as the initial measurement for formula 2.2 of RFC 6298. The sender keeps track of the time each message was sent for all recently sent messages. Any time an ACK is received, the sender can compute the RTT for that message by looking at the time the ACK was received and the time when the message was sent. This is used as a subsequent RTT measurement for formula 2.3 of RFC 6298 to update the RTO estimate. (Note that because retransmissions receive new sequence numbers, all received ACKs are used.)

The value for RTO is calculated separately for each DTLS session.

Retransmissions continue until a response is received, or until a total of 5 requests have been sent or there has been a hard ICMP error [RFC1122] or a TLS alert. The sender knows a response was received when it receives an ACK with a sequence number that indicates it is a response to one of the transmissions of this messages. For example, assuming an RTO of 500 ms, requests would be sent at times 0 ms, 500 ms, 1500 ms, 3500 ms, and 7500 ms. If all retransmissions for a message fail, then the sending node SHOULD close the connection routing the message.

To determine when a link may be failing without waiting for the final timeout, observe when no ACKs have been received for an entire RTO

interval, and then wait for three retransmissions to occur beyond that point. If no ACKs have been received by the time the third retransmission occurs, it is RECOMMENDED that the link be removed from the routing table. The link MAY be restored to the routing table if ACKs resume before the connection is closed, as described above.

Once an ACK has been received for a message, the next message can be sent, but the peer SHOULD ensure that there is at least 10 ms between sending any two messages. The only time a value less than 10 ms can be used is when it is known that all nodes are on a network that can support retransmissions faster than 10 ms with no congestion issues.

5.6.4. DTLS/UDP with SR

This overlay link protocol consists of DTLS over UDP while implementing the Simple Reliability protocol. STUN Connectivity checks and keepalives are used.

5.6.5. TLS/TCP with FH, No-ICE

This overlay link protocol consists of TLS over TCP with the framing header. Because ICE is not used, STUN connectivity checks are not used upon establishing the TCP connection, nor are they used for keepalives.

Because the TCP layer's application-level timeout is too slow to be useful for overlay routing, the Overlay Link implementation MUST use the framing header to measure the RTT of the connection and calculate an RTO as specified in Section 2 of [RFC6298]. The resulting RTO is not used for retransmissions, but as a timeout to indicate when the link SHOULD be removed from the routing table. It is RECOMMENDED that such a connection be retained for 30s to determine if the failure was transient before concluding the link has failed permanently.

When sending candidates for TLS/TCP with FH, No-ICE, a passive candidate MUST be provided.

5.6.6. DTLS/UDP with SR, No-ICE

This overlay link protocol consists of DTLS over UDP while implementing the Simple Reliability protocol. Because ICE is not used, no STUN connectivity checks or keepalives are used.

5.7. Fragmentation and Reassembly

In order to allow transmission over datagram protocols such as DTLS, RELOAD messages may be fragmented.

Any node along the path can fragment the message but only the final destination reassembles the fragments. When a node takes a packet and fragments it, each fragment has a full copy of the Forwarding Header but the data after the Forwarding Header is broken up in appropriate sized chunks. The size of the payload chunks needs to take into account space to allow the via and destination lists to grow. Each fragment MUST contain a full copy of the via and destination list and MUST contain at least 256 bytes of the message body. If the via and destination list are so large that this is not possible, RELOAD fragmentation is not performed and IP-layer fragmentation is allowed to occur. When a message must be fragmented, it SHOULD be split into equal-sized fragments that are no larger than the PMTU of the next overlay link minus 32 bytes. This is to allow the via list to grow before further fragmentation is required.

Note that this fragmentation is not optimal for the end-to-end path - a message may be refragmented multiple times as it traverses the overlay but is only assembled at the final destination. This option has been chosen as it is far easier to implement than e2e PMTU discovery across an ever-changing overlay, and it effectively addresses the reliability issues of relying on IP-layer fragmentation. However, PING can be used to allow e2e PMTU to be implemented if desired.

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 message and processed. In order to mitigate denial of service attacks, receivers SHOULD time out incomplete fragments after maximum request lifetime (15 seconds). Note this time was derived from looking at the end to end retransmission time and saving fragments long enough for the full end to end retransmissions to take place. Ideally the receiver would have enough buffer space to deal with as many

fragments as can arrive in the maximum request lifetime. However, if the receiver runs out of buffer space to reassemble the messages it MUST drop the message.

When a message is fragmented, the fragment offset value is stored in the lower 24 bits of the fragment field of the forwarding header. The offset is the number of bytes between the end of the forwarding header and the start of the data. The first fragment therefore has an offset of 0. The first and last bit indicators MUST be appropriately set. If the message is not fragmented, then both the first and last fragment bits are set to 1 and the offset is 0 resulting in a fragment value of 0xC0000000. Note that this means that the first fragment bit is always 1, so isn't actually that useful.

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 size of the StoredData structure in octets excluding the size of length itself.

storage_time
: The time when the data was stored represented as the number of milliseconds elapsed since midnight Jan 1, 1970 UTC not counting leap seconds. This will have the same values for seconds as standard UNIX time or POSIX time. More information can be found at [UnixTime]. 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. The node SHOULD make a best-effort attempt to use a correct clock to determine this number, however, the protocol does not require synchronized clocks: the receiving peer uses the storage time in the previous store, not its own clock. Clock values are used so that when clocks are generally synchronized, data may be stored in a single transaction, rather than querying for the value of a counter before the actual store.
If a node attempting to store new data in response to a user request (rather than as an overlay maintenance operation such as occurs during unpartitioning) is rejected with an Error_Data_Too_Old error, the node MAY elect to perform its store using a storage_time that increments the value used with the previous store. This situation may occur when the clocks of nodes storing to this location are not properly synchronized.

lifetime
The validity period for the data, in seconds, starting from the time the peer receives the StoreReq.

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

signature
A signature as defined in Section 6.1.

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 || storage_time || StoredDataValue ||
SignerIdentity

Where || indicates concatenation.

Where these values are:

resource_id
The resource ID where this data is stored.

kind
The Kind-ID for this data.

storage_time

The contents of the storage_time data value.

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

SignerIdentity
The signer identity as defined in Section 5.3.4.

Once the signature has been computed, the signature is represented using a signature element, as described in Section 5.3.4.

6.2. Data Models

The protocol currently defines the following data models:

- single value
- array
- dictionary

These are represented with the StoredDataValue structure. The actual dataModel is known from the kind being stored.

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

        struct {
          select (dataModel) {
            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 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. Access Control Policies

Every kind which is storable in an overlay MUST be associated with an access control policy. This policy defines whether a request from a

given node to operate on a given value should succeed or fail. It is anticipated that only a small number of generic access control policies are required. To that end, this section describes a small set of such policies and Section 13.4 establishes a registry for new policies if required. Each policy has a short string identifier which is used to reference it in the configuration document.

In the following policies, the term "signer" refers to the signer of the StoredValue object and, in the case of non-replica stores, to the signer of the StoreReq message. I.e., in a non-replica store, both the signer of the StoredValue and the signer of the StoreReq MUST conform to the policy. In the case of a replica store, the signer of the StoredValue MUST conform to the policy and the StoreReq itself MUST be checked as described in Section 6.4.1.1.

6.3.1. USER-MATCH

In the USER-MATCH policy, a given value MUST be written (or overwritten) if and only if the signer's certificate has a user name which hashes (using the hash function for the overlay) to the Resource-ID for the resource. Recall that the certificate may, depending on the overlay configuration, be self-signed.

6.3.2. NODE-MATCH

In the NODE-MATCH policy, a given value MUST be written (or overwritten) if and only if the signer's certificate has a specified Node-ID which hashes (using the hash function for the overlay) to the Resource-ID for the resource and that Node-ID is the one indicated in the SignerIdentity value cert_hash.

6.3.3. USER-NODE-MATCH

The USER-NODE-MATCH policy may only be used with dictionary types. In the USER-NODE-MATCH policy, a given value MUST be written (or overwritten) if and only if the signer's certificate has a user name which hashes (using the hash function for the overlay) to the Resource-ID for the resource. In addition, the dictionary key MUST be equal to the Node-ID in the certificate and that Node-ID MUST be the one indicated in the SignerIdentity value cert_hash.

6.3.4. NODE-MULTIPLE

In the NODE-MULTIPLE policy, a given value MUST be written (or overwritten) if and only if signer's certificate contains a Node-ID such that H(Node-ID || i) is equal to the Resource-ID for some small integer value of i and that Node-ID is the one indicated in the SignerIdentity value cert_hash. When this policy is in use, the

maximum value of i MUST be specified in the kind definition.

Note that as i is not carried on the wire, the verifier MUST iterate through potential i values up to the maximum value in order to determine whether a store is acceptable.

6.4. Data Storage Methods

RELOAD provides several methods for storing and retrieving data:

- Store values in the overlay
- Fetch values from the overlay
- Stat: get metadata about values in the overlay
- Find the values stored at an individual peer

These methods are each described in the following sections.

6.4.1. 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.

6.4.1.1. 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;
    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 that the request sender is consistent with being the responsible node (i.e., that the receiving peer does not know of a better node) 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 MUST reject requests corresponding to unknown kinds.

generation_counter
   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:

- The Kind-ID is known and supported.
- The signatures over each individual data element (if any) are valid. If this check fails, the request MUST be rejected with an Error_Forbidden error.
- Each element is signed by a credential which is authorized to write this kind at this Resource-ID. If this check fails, the request MUST be rejected with an Error_Forbidden error.

- For original (non-replica) stores, the StoreReq is signed by a credential which is authorized to write this kind at this Resource-Id. If this check fails, the request MUST be rejected with an Error_Forbidden error.
- For replica stores, the StoreReq is signed by a Node-Id which is a plausible node to either have originally stored the value or in the replica set. What this means is overlay specific, but in the case of the Chord based DHT defined in this specification, replica StoreReqs MUST come from nodes which are either in the known replica set for a given resource or which are closer than some node in the replica set. If this check fails, the request MUST be rejected with an Error_Forbidden error.
- 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.
- For replica Stores, the peer MUST set the generation counter to match the generation counter in the message, and MUST NOT check the generation counter against the current value. Replica Stores MUST NOT use a generation counter of 0.
- The storage time values are greater than that of any value which would be replaced by this Store.
- The size and number of the stored values is consistent with the limits specified in the overlay configuration.
- If the data is signed with identity_type set to "none" and/or SignatureAndHashAlgorithm values set to {0, 0} ("anonymous" and "none"), the StoreReq MUST be rejected with an Error_forbidden error. Only synthesized data returned by the storage can use these values

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 in the Store request to indicate that the generation counter should be ignored for processing this request; however the responsible peer should increase the stored generation counter and should return the correct generation counter in the response.

When a peer stores data previously stored by another node (e.g., for replicas or topology shifts) it MUST adjust the lifetime value downward to reflect the amount of time the value was stored at the peer. The adjustment SHOULD be implemented by an algorithm

equivalent to the following: at the time the peer initially receives the StoreReq it notes the local time T. When it then attempts to do a StoreReq to another node it should decrement the lifetime value by the difference between the current local time and T.

Unless otherwise specified by the usage, if a peer attempts to store data previously stored by another node (e.g., for replicas or topology shifts) and that store fails with either an Error_Generation_Counter_Too_Low or an Error_Data_Too old error, the peer MUST fetch the newer data from the peer generating the error and use that to replace its own copy. This rule allows resynchronization after partitions heal.

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

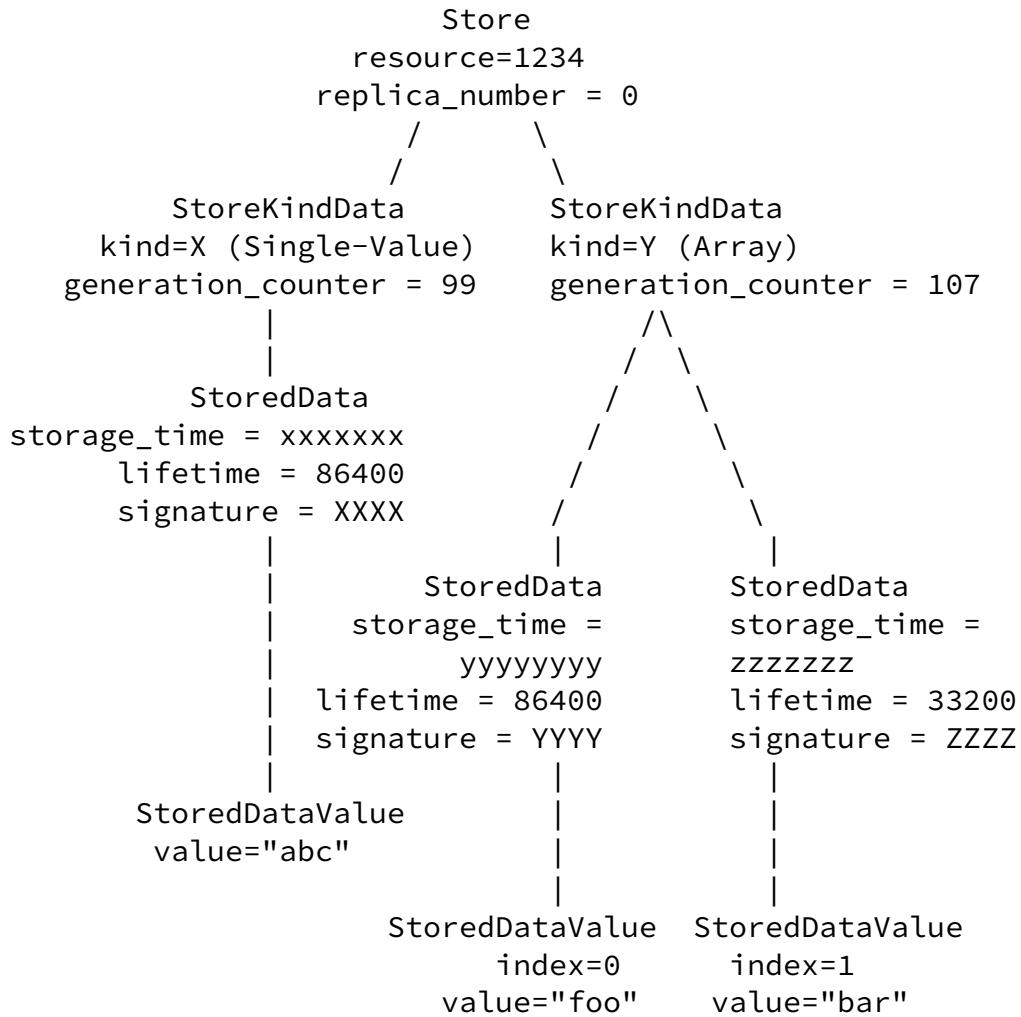
Single-value:
: A store of a new single-value element creates the element if it does not exist and overwrites any existing value with the new value.

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"

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



6.4.1.2. 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 will be replicated by the node processing the request.

```
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_counter
: 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 have in fact been stored. It does this verification by using the Stat method (see Section 6.4.3). Note that the storing peer is not required to perform this verification.

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 an Error_Generation_Counter_Too_Low error. The error_info in the ErrorResponse MUST be a StoreAns response containing the correct generation counter for each kind and the replica list, which will be empty. For original (non-replica) stores, a node which receives such an error SHOULD attempt to fetch the data and, if the storage_time value is newer, replace its own data with that newer data. This rule improves data consistency in the case of partitions and merges.

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_Large error.

If any type of request tries to access a data kind that the node does not know about, an Error_Unknown_Kind MUST be generated. The error_info in the Error_Response is:

```
KindId        unknown_kinds<0..2^8-1>;
```

which lists all the kinds that were unrecognized. A node which receives this error MUST generate a ConfigUpdate message which contains the appropriate kind definition (assuming that in fact a kind was used which was defined in the configuration document).

6.4.1.3. Removing Values

This version of RELOAD (unlike previous versions) does not have an explicit Remove operation. Rather, values are Removed by storing "nonexistent" values in their place. Each DataValue contains a boolean value called "exists" which indicates whether a value is present at that location. In order to effectively remove a value, the owner stores a new DataValue with "exists" set to "false":

```
exists = false
value = {} (0 length)
```

The owner SHOULD use a lifetime for the nonexistent value at least as long as the remainder of the lifetime of the value it is replacing; otherwise it is possible for the original value to be accidentally or

maliciously re-stored after the storing node has expired it. Note that there is still a window of vulnerability for replay attack after the original lifetime has expired (as with any store). This attack can be mitigated by doing a nonexistent store with a very long lifetime.

Storing nodes MUST treat these nonexistent values the same way they treat any other stored value, including overwriting the existing value, replicating them, and aging them out as necessary when lifetime expires. When a stored nonexistent value's lifetime expires, it is simply removed from the storing node like any other stored value expiration.

Note that in the case of arrays and dictionaries, expiration may create an implicit, unsigned "nonexistent" value to represent a gap in the data structure, as might happen when any value is aged out. However, this value isn't persistent nor is it replicated. It is simply synthesized by the storing node.

6.4.2. 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.

6.4.2.1. Request Definition

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

struct {
```

```
      KindId                  kind;
      uint64                  generation;
      uint16                  length;

      select (dataModel) {
        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.

dataModel
: The data model of the data. This is not transmitted on the wire but comes from the definition of the kind.

generation
: The last generation counter that the requesting node 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:

- If the data model is single value, the specifier is empty.
- If the data model is array, the specifier contains 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 first integer must be less than the second. While multiple ranges MAY be specified, they MUST NOT overlap.
- If the data model is 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.

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 node that does not already have the user's certificate should request the certificate in the Fetch as an optimization.

6.4.2.2. 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 (i.e., one which the node has no knowledge of) is represented by a synthetic value with "exists" set to False and has an empty signature. Specifically, the identity_type is set to "none", the SignatureAndHashAlgorithm values are set to {0, 0} ("anonymous" and "none" respectively), and the signature value is of zero length. This removes the need for the responding node to do signatures for values which do not exist. These signatures are unnecessary as the entire response is signed by that node. Note that entries which have been removed by the procedure of Section 6.4.1.3 and have not yet expired also have exists = false but have valid signatures from the node which did the store.

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 the array is set to zero before the signature is computed. This implies that malicious storing nodes can reorder array entries without being detected.

6.4.3. 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 file system. It also allows the requesting node to get a list of matching elements without requesting the entire element.

6.4.3.1. 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;
```

6.4.3.2. 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 {
          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                  value_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.

6.4.4. 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 closest to R. This method can be used to walk the Overlay Instance by iteratively fetching R_n+1=nearest(1 + R_n).

6.4.4.1. Request Definition

The FindReq message contains a Resource-ID and a series of 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, and if not the request MUST be rejected with an error.

6.4.4.2. Response Definition

A response to a successful Find request is a FindAns message containing the closest Resource-ID on the peer 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 an Error_Not_Found error code.

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.

6.4.5. Defining New Kinds

There are two ways to define a new kind. The first is by writing a

document and registering the kind-id with IANA. This is the preferred method for kinds which may be widely used and reused. The second method is to simply define the kind and its parameters in the configuration document using the section of kind-id space set aside for private use. This method MAY be used to define ad hoc kinds in new overlays.

However a kind is defined, the definition must include:

- The meaning of the data to be stored (in some textual form).
- The Kind-ID.
- The data model (single value, array, dictionary, etc).
- The access control model.

In addition, when kinds are registered with IANA, each kind is assigned a short string name which is used to refer to it in configuration documents.

While each kind needs to 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 existing 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:

- The user name in the certificate.

- 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 an old and a new certificate that both have the same Node-ID, which allows for migration of certificates when they are renewed.

This usage defines the following kinds:

Name: CERTIFICATE_BY_NODE

Data Model: The data model for CERTIFICATE_BY_NODE data is array.

Access Control: NODE-MATCH.

Name: CERTIFICATE_BY_USER

Data Model: The data model for CERTIFICATE_BY_USER data is array.

Access Control: USER-MATCH.

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 [RFC5766]. When a node starts up, it joins the overlay network and forms several connections in the process. If the ICE stage in any of these connections returns a reflexive address that is not the same as the peer's perceived address, then the peer is behind a NAT and not a candidate for a TURN server. Additionally, if the peer's IP address is in the private address space range, then it is also 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 turn-density 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. If the turn-density is not set to zero, for each value, called d, between 1 and turn-density, the peer forms a Resource Name by concatenating its Node-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 on having turn-density be an reasonable estimate of the reciprocal of the proportion of nodes in the overlay that can act as TURN servers. If the turn-density value in the configuration file is too low, then the process of finding TURN servers becomes more expensive as multiple candidate Resource-IDs must be probed to find a TURN server.

Peers that provide this service need to support the TURN extensions to STUN for media relay as defined in [RFC5766].

This usage defines the following kind to indicate that a peer is willing to act as a TURN server:

Name TURN-SERVICE
Data Model The TURN-SERVICE kind stores a single value for each Resource-ID.
Access Control NODE-MULTIPLE, with maximum iteration counter 20.

Peers can find other servers by selecting a random Resource-ID and then doing a Find request for the appropriate Kind-ID 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. Chord Algorithm

This algorithm is assigned the name chord-reload to indicate it is an adaptation of the basic Chord based DHT algorithm.

This algorithm differs from the originally presented Chord algorithm [Chord]. It has been updated based on more recent research results and implementation experiences, and to adapt it to the RELOAD protocol. A short list of differences:

- The original Chord algorithm specified that a single predecessor and a successor list be stored. The chord-reload algorithm attempts to have more than one predecessor and successor. The predecessor sets help other neighbors learn their successor list.
- The original Chord specification and analysis called for iterative routing. RELOAD specifies recursive routing. In addition to the performance implications, the cost of NAT traversal dictates recursive routing.
- Finger table entries are indexed in opposite order. Original Chord specifies finger[0] as the immediate successor of the peer. chord-reload specifies finger[0] as the peer 180 degrees around the ring from the peer. This change was made to simplify discussion and implementation of variable sized finger tables. However, with either approach no more than O(log N) entries should typically be stored in a finger table.
- The stabilize() and fix_fingers() algorithms in the original Chord algorithm are merged into a single periodic process. Stabilization is implemented slightly differently because of the larger neighborhood, and fix_fingers is not as aggressive to reduce load, nor does it search for optimal matches of the finger table entries.
- RELOAD uses a 128 bit hash instead of a 160 bit hash, as RELOAD is not designed to be used in networks with close to or more than 2^128 nodes (and it is hard to see how one would assemble such a network).
- RELOAD uses randomized finger entries as described in Section 9.7.4.2.

- This algorithm allows the use of either reactive or periodic recovery. The original Chord paper used periodic recovery. Reactive recovery provides better performance in small overlays, but is believed to be unstable in large (>1000) overlays with high levels of churn [handling-churn-usenix04]. The overlay configuration file specifies a "chord-reactive" element that indicates whether reactive recovery should be used.

9.1. Overview

The algorithm described here is a modified version of the Chord algorithm. Each peer keeps track of a finger table and a neighbor table. The neighbor table contains at least the three peers before and after this peer in the DHT ring. There may not be three entries in all cases such as small rings or while the ring topology is changing. 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 neighbor 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 the insertion 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 Node-ID of the previous peer in the neighbor table. Care must be taken when computing to note that all math is modulo 2^128.

9.2. Hash Function

For this Chord based topology plugin, the size of the Resource-ID is 128 bits. The hash of a Resource-ID is computed using SHA-1 [RFC3174]then truncating the SHA-1 result to the most significant 128

bits.

9.3. Routing

The routing table is the union of the neighbor table and the finger table.

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. If no such node is found, it finds the smallest Node-Id that is greater than k and routes the message to that node.

9.4. 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. It then sends a Store request to its successor in the neighbor table and to that peer's successor. Note that these Store requests are addressed to those specific peers, even though the Resource-ID they are being asked to store is outside the range that they are responsible for. The peers receiving these check they came from an appropriate predecessor in their neighbor 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.

Managing replicas as the overlay changes is described in Section 9.7.3.

The sequential replicas used in this overlay algorithm protect against peer failure but not against malicious peers. Additional replication from the Usage is required to protect resources from such attacks, as discussed in Section 12.5.4.

9.5. Joining

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

follows.

1. JP MUST connect to its chosen bootstrap node.
2. JP SHOULD send an Attach request to the admitting peer (AP) for Node-ID n. The "send_update" flag should be used to acquire the routing table for AP.
3. JP SHOULD send Attach requests to initiate connections to each of the peers in the neighbor 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 MUST enter all the peers it has contacted into its routing table.
5. JP MUST send a Join to AP. The AP sends the response to the Join.
6. AP MUST do a series of Store requests to JP to store the data that JP will be responsible for.
7. AP MUST send 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. The AP MUST send an Update to all of its neighbors with the new values of its neighbor set (including JP).
9. The JP MUST send Updates to all the peers in its neighbor table.

If JP sends an Attach to AP with send_update, it immediately knows most of its expected neighbors from AP's routing table update and can directly connect to them. This is the RECOMMENDED procedure.

If for some reason JP does not get AP's routing table, it can still populate its neighbor table incrementally. It sends a Ping directed at Resource-ID n+1 (directly after its own Resource-ID). This allows it to discover its own successor. Call that node p0. It then sends a ping to p0+1 to discover its successor (p1). This process can be repeated to discover as many successors as desired. The values for the two peers before p will be found at a later stage when n receives an Update. An alternate procedure is to send Attaches to those nodes rather than pings, which forms the connections immediately but may be slower if the nodes need to collect ICE candidates, thus reducing parallelism.

In order to set up its finger table entry for peer i, JP simply sends an Attach to peer (n+2^(128-i). This will be routed to a peer in approximately the right location around the ring.

The joining peer MUST NOT send any Update message placing itself in the overlay until it has successfully completed an Attach with each peer that should be in its neighbor table.

9.6. Routing Attaches

When a peer needs to Attach to a new peer in its neighbor 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.

9.7. Updates

An Update for this DHT is defined as


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


struct {
   uint32                uptime;
   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 "uptime" field contains the time this peer has been up in seconds.

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 peers.

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 of 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, even though this will not be possible if the ring is very small. It is RECOMMENDED that O(log(N)) predecessors and successors be maintained in the neighbor set.

9.7.1. Handling Neighbor Failures

Every time a connection to a peer in the neighbor table is lost (as determined by connectivity pings or the failure of some request), the peer MUST remove the entry from its neighbor table and replace it with the best match it has from the other peers in its routing table. If using reactive recovery, it then sends an immediate Update to all nodes in its Neighbor Table. 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 successor replacement hold-down time (30 seconds) after removing the failed entry from its neighbor table in order to allow a triggered update to inform it of a better match for its neighbor table.

If the neighbor failure effects the peer's range of responsible IDs, then the Update MUST be sent to all nodes in its Connection Table.

A peer MAY attempt to reestablish connectivity with a lost neighbor either by waiting additional time to see if connectivity returns or by actively routing a new Attach to the lost peer. Details for these

procedures are beyond the scope of this document. In no event does an attempt to reestablish connectivity with a lost neighbor allow the peer to remain in the neighbor table. Such a peer is returned to the neighbor table once connectivity is reestablished.

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

9.7.2. Handling Finger Table Entry Failure

If a finger table entry is found to have failed, all references to the failed peer are removed from the finger table and replaced with the closest preceding peer from the finger table or neighbor table.

If using reactive recovery, the peer initiates a search for a new finger table entry as described below.

9.7.3. Receiving Updates

When a peer, N, receives an Update request, it examines the Node-IDs in the UpdateReq and at its neighbor table and decides if this UpdateReq would change its neighbor table. This is done by taking the set of peers currently in the neighbor table and comparing them to the peers in the update request. There are two major cases:

- The UpdateReq contains peers that match N's neighbor table, so no change is needed to the neighbor set.
- The UpdateReq contains peers N does not know about that should be in N's neighbor table, i.e. they are closer than entries in the neighbor table.

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.

After any Pings and Attaches are done, if the neighbor table changes and the peer is using reactive recovery, the peer sends an Update request to each member of its Connection Table. These Update requests are what end 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 is responsible for a Resource-ID R, and N 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.

When a peer discovers that its range of responsible IDs have changed, it MUST send an Update to all entries in its connection table.

9.7.4. Stabilization

There are four components to stabilization:

1. exchange Updates with all peers in its neighbor 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.

9.7.4.1. Updating neighbor table

A peer MUST periodically send an Update request to every peer in its Connection Table. The purpose of this is to keep the predecessor and successor lists up to date and to detect failed peers. The default time is about every ten minutes, but the configuration server SHOULD set this in the configuration document using the "chord-update-interval" element (denominated in seconds.) A peer SHOULD randomly offset these Update requests so they do not occur all at once.

9.7.4.2. Refreshing finger table

A peer MUST periodically search for new peers to replace invalid entries in the finger table. A finger table entry i is valid if it is in the range [n+2^(128-i) , n+2^(128-(i-1))-1]. Invalid entries occur in the finger table when a previous finger table entry

has failed or when no peer has been found in that range.

A peer SHOULD NOT send Ping requests looking for new finger table entries more often than the configuration element "chord-ping-interval", which defaults to 3600 seconds (one per hour).

Two possible methods for searching for new peers for the finger table entries are presented:

Alternative 1: A peer selects one entry in the finger table from among the invalid entries. It pings for a new peer for that finger table entry. The selection SHOULD be exponentially weighted to attempt to replace earlier (lower i) entries in the finger table. A simple way to implement this selection is to search through the finger table entries from i=0 and each time an invalid entry is encountered, send a Ping to replace that entry with probability 0.5.

Alternative 2: A peer monitors the Update messages received from its connections to observe when an Update indicates a peer that would be used to replace in invalid finger table entry, i, and flags that entry in the finger table. Every "chord-ping-interval" seconds, the peer selects from among those flagged candidates using an exponentially weighted probability as above.

When searching for a better entry, the peer SHOULD send the Ping 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. Any such alternate methods SHOULD 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.

A peer MAY choose to keep connections to multiple peers that can act for a given finger table entry.

9.7.4.3. Adjusting finger table size

If the finger table has less than 16 entries, the node SHOULD attempt

to discover more fingers to grow the size of the table to 16. The value 16 was chosen to ensure high odds of a node maintaining connectivity to the overlay even with strange network partitions.

For many overlays, 16 finger table entries will be enough, but as an overlay grows very large, more than 16 entries may be required in the finger table for efficient routing. An implementation SHOULD be capable of increasing the number of entries in the finger table to 128 entries.

Note to implementers: Although log(N) entries are all that are required for optimal performance, careful implementation of stabilization will result in no additional traffic being generated

when maintaining a finger table larger than log(N) entries. Implementers are encouraged to make use of RouteQuery and algorithms for determining where new finger table entries may be found. Complete details of possible implementations are outside the scope of this specification.

A simple approach to sizing the finger table is to ensure the finger table is large enough to contain at least the final successor in the peer's neighbor table.

9.7.4.4. Detecting partitioning

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 node, B. P should then send a Ping 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 an Attach to S' routed through B, followed by an Update sent to S'. (Note that S' may not be in P's neighbor table once the overlay is healed, but the connection will allow S' to discover appropriate neighbor entries for itself via its own stabilization.)

Future specifications may describe alternative mechanisms for determining when to repeat the discovery process.

9.8. 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_peer;
} ChordRouteQueryAns;
```

The contents of this structure are as follows:

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

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

9.9. Leaving

To support extensions, such as [I-D.ietf-p2psip-self-tuning], Peers SHOULD send a Leave request to all members of their neighbor table prior to exiting the Overlay Instance. The overlay_specific_data field MUST contain the ChordLeaveData structure defined below:

```
          enum { reserved (0),
                  from_succ(1), from_pred(2), (255) }
                ChordLeaveType;

           struct {
             ChordLeaveType         type;

              select(type) {
                case from_succ:
                  NodeId               successors<0..2^16-1>;
                case from_pred:
                   NodeId               predecessors<0..2^16-1>;
              };
           } ChordLeaveData;
```

The 'type' field indicates whether the Leave request was sent by a predecessor or a successor of the recipient:

from_succ
: The Leave request was sent by a successor.

from_pred
: The Leave request was sent by a predecessor.

If the type of the request is 'from_succ', the contents will be:

successors
: The sender's successor list.

If the type of the request is 'from_pred', the contents will be:

predecessors
: The sender's predecessor list.

Any peer which receives a Leave for a peer n in its neighbor set follows procedures as if it had detected a peer failure as described in Section 9.7.1.

10. Enrollment and Bootstrap

The section defines the format of the configuration data as well the process to join a new overlay.

10.1. 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 xmlns="urn:ietf:params:xml:ns:p2p:config-base"
   xmlns:ext="urn:ietf:params:xml:ns:p2p:config-ext1"
```

```
      xmlns:chord="urn:ietf:params:xml:ns:p2p:config-chord">
      <configuration instance-name="overlay.example.org" sequence="22"
          expiration="2002-10-10T07:00:00Z" ext:ext-example="stuff" >
          <topology-plugin> CHORD-RELOAD </topology-plugin>
          <node-id-length>16</node-id-length>
          <root-cert>
   MIIDJDCCAo2gAwIBAgIBADANBgkqhkiG9w0BAQUFADBwMQswCQYDVQQGEwJVUzET
   MBEGA1UECBMKQ2FsaWZvcm5pYTERMA8GA1UEBxMIU2FuIEpvc2UxDjAMBgNVBAoT
   BXNpcGl0MSkwJwYDVQQLEyBTaXBpdCBUZXN0IENlcnRpZmljYXRlIEF1dGhvcml0
   eTAeFw0wMzA3MTgxMjIxNTJaFw0xMzA3MTUxMjIxNTJaMHAxCzAJBgNVBAYTAlVT
   MRMwEQYDVQQIEwpDYWxpZm9ybmlhMREwDwYDVQQHEwhTYW4gSm9zZTEOMAwGA1UE
   ChMFc2lwaXQxKTAnBgNVBAsTIFNpcGl0IFRlc3QgQ2VydGlmaWNhdGUgQXV0aG9y
   aXR5MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDDIh6DkcUDLDyK9BEUxkud
   +nJ4xrCVGKfgjHm6XaSuHiEtnfELHM+9WymzkBNzZpJu30yzsxwfKoIKugdNUrD4
   N3viCicwcN35LgP/KnbN34cavXHr4ZlqxH+OdKB3hQTpQa38A7YXdaoz6goW2ft5
   Mi74z03GNKP/G9BoKOGd5QIDAQABo4HNMIHKMB0GA1UdDgQWBBRrRhcU6pR2JYBU
   bhNU2qHjVBShtjCBmgYDVR0jBIGSMIGPgBRrRhcU6pR2JYBUbhNU2qHjVBShtqF0
   pHIwcDELMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExETAPBgNVBAcT
   CFNhbiBKb3NlMQ4wDAYDVQQKEwVzaXBpdDEpMCcGA1UECxMgU2lwaXQgVGVzdCBD
   ZXJ0aWZpY2F0ZSBBdXRob3JpdHmCAQAwDAYDVR0TBAUwAwEB/zANBgkqhkiG9w0B
   AQUFAAOBgQCWbRvv1ZGTRXxbH8/EqkdSCzSoUPrs+rQqR0xdQac9wNY/nlZbkR3O
   qAezG6Sfmklvf+DOg5RxQq/+Y6I03LRepc7KeVDpaplMFGnpfKsibETMipwzayNQ
   QgUf4cKBiF+65Ue7hZuDJa2EMv8qW4twEhGDYclpFU9YozyS1OhvUg==
          </root-cert>
          <root-cert> YmFkIGNlcnQK </root-cert>
          <enrollment-server>https://example.org</enrollment-server>
          <enrollment-server>https://example.net</enrollment-server>
          <self-signed-permitted
                    digest="sha1">false</self-signed-permitted>
          <bootstrap-node address="192.0.0.1" port="6084" />
          <bootstrap-node address="192.0.2.2" port="6084" />
          <bootstrap-node address="2001:DB8::1" port="6084" />
          <turn-density> 20 </turn-density>
          <multicast-bootstrap address="192.0.0.1" />
```



```
          <multicast-bootstrap address="233.252.0.1" port="6084" />
          <clients-permitted> false </clients-permitted>
          <no-ice> false </no-ice>
          <chord:chord-update-interval>
              400</chord:chord-update-interval>
          <chord:chord-ping-interval>30</chord:chord-ping-interval>
          <chord:chord-reactive> true </chord:chord-reactive>
```

```
<shared-secret> password </shared-secret>
<max-message-size>4000</max-message-size>
<initial-ttl> 30 </initial-ttl>
<overlay-link-protocol>TLS</overlay-link-protocol>
<configuration-signer>47112162e84c69ba</configuration-signer>
<kind-signer> 47112162e84c69ba </kind-signer>
<kind-signer> 6eba45d31a900c06 </kind-signer>
<bad-node> 6ebc45d31a900c06 </bad-node>
<bad-node> 6ebc45d31a900ca6 </bad-node>

<ext:example-extension> foo </ext:example-extension>

<mandatory-extension>
    urn:ietf:params:xml:ns:p2p:config-ext1
</mandatory-extension>

<required-kinds>
  <kind-block>
    <kind name="SIP-REGISTRATION">
        <data-model>SINGLE</data-model>
        <access-control>USER-MATCH</access-control>
        <max-count>1</max-count>
        <max-size>100</max-size>
    </kind>
    <kind-signature>
         VGhpcyBpcyBub3QgcmlnaHQhCg==
    </kind-signature>
  </kind-block>
  <kind-block>
    <kind id="2000">
        <data-model>ARRAY</data-model>
        <access-control>NODE-MULTIPLE</access-control>
        <max-node-multiple>3</max-node-multiple>
        <max-count>22</max-count>
        <max-size>4</max-size>
        <ext:example-kind-extension> 1
                </ext:example-kind-extension>
    </kind>
    <kind-signature>
       VGhpcyBpcyBub3QgcmlnaHQhCg==
    </kind-signature>
```

```
          </kind-block>
        </required-kinds>
    </configuration>
    <signature> VGhpcyBpcyBub3QgcmlnaHQhCg== </signature>

    <configuration instance-name="other.example.net">
    </configuration>
    <signature> VGhpcyBpcyBub3QgcmlnaHQhCg== </signature>

 </overlay>
```

The file MUST be a well formed XML document and it SHOULD contain an encoding declaration in the XML declaration. The file MUST use the UTF-8 character encoding. The namespace for the elements defined in this specification is urn:ietf:params:xml:ns:p2p:config-base and urn:ietf:params:xml:ns:p2p:config-chord".

The file can contain multiple "configuration" elements where each one contains the configuration information for a different overlay. Each configuration element may be followed by signature elements that provides a signature over the preceding configuration element. Each configuration element has the following attributes:

instance-name: name of the overlay
expiration: time in the future at which this overlay configuration is no longer valid. The node SHOULD retrieve a new copy of the configuration at a randomly selected time that is before the expiration time. Note that if the certificates expire before a new configuration is retried, the node will not be able to validate the configuration file.
sequence: a monotonically increasing sequence number between 1 and 2^16-2

Inside each overlay element, the following elements can occur:

topology-plugin This element defines the overlay algorithm being used. If missing the default is "CHORD-RELOAD".
node-id-length This element contains the length of a NodeId (NodeIdLength) in bytes. This value MUST be between 16 (128 bits) and 20 (160 bits). If this element is not present, the default of 16 is used.
root-cert This element contains a base-64 encoded X.509v3 certificate that is a root trust anchor used to sign all certificates in this overlay. There can be more than one root-cert element.

enrollment-server This element contains the URL at which the enrollment server can be reached in a "url" element. This URL MUST be of type "https:". More than one enrollment-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 enrollment-server and root-cert elements may be absent. Otherwise, it SHOULD be absent, but MAY be set to "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 "sha1" and "sha256" representing SHA-1 [RFC3174] and SHA-256 [RFC6234] respectively. Implementations MUST support both of these algorithms.

bootstrap-node This element represents the address of one of the bootstrap nodes. It has an attribute called "address" that represents the IP address (either IPv4 or IPv6, since they can be distinguished) and an optional attribute called "port" that represents the port and defaults to 6084. The IP address is in typical hexadecimal form using standard period and colon separators as specified in [RFC5952]. More than one bootstrap-peer element may be present.

turn-density This element is a positive integer that represents the approximate reciprocal of density of nodes that can act as TURN servers. For example, if 5% of the nodes can act as TURN servers, this would be set to 20. If it is not present, the default value is 1. If there are no TURN servers in the overlay, it is set to zero.

multicast-bootstrap This element represents the address of a multicast, broadcast, or anycast address and port that may be used for bootstrap. Nodes SHOULD listen on the address. It has an attributed called "address" that represents the IP address and an optional attribute called "port" that represents the port and defaults to 6084. 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 are not allowed to remain clients after the initial join. There is currently no way for the overlay to enforce this.

no-ice This element represents whether nodes are required to use the "No-ICE" Overlay Link protocols in this overlay. If it is

absent, it is treated as if it were set to "false".

chord-update-interval The update frequency for the Chord-reload topology plugin (see Section 9).

chord-ping-interval The ping frequency for the Chord-reload topology plugin (see Section 9).

chord-reactive Whether reactive recovery should be used for this overlay. Set to "true" or "false". Default if missing is "true". (see Section 9).

shared-secret If shared secret mode is used, this contains the shared secret.

max-message-size Maximum size in bytes of any message in the overlay. If this value is not present, the default is 5000.

initial-ttl Initial default TTL (time to live, see Section 5.3.2) for messages. If this value is not present, the default is 100.

overlay-link-protocol Indicates a permissible overlay link protocol (see Section 5.6.1 for requirements for such protocols). An arbitrary number of these elements may appear. If none appear, then this implies the default value, "TLS", which refers to the use of TLS and DTLS. If one or more elements appear, then no default value applies.

kind-signer This contains a single Node-ID in hexadecimal and indicates that the certificate with this Node-ID is allowed to sign kinds. Identifying kind-signer by Node-ID instead of certificate allows the use of short lived certificates without constantly having to provide an updated configuration file.

configuration-signer This contains a single Node-ID in hexadecimal and indicates that the certificate with this Node-ID is allowed to sign configurations for this instance-name. Identifying the signer by Node-ID instead of certificate allows the use of short lived certificates without constantly having to provide an updated configuration file.

bad-node This contains a single Node-ID in hexadecimal and indicates that the certificate with this Node-ID MUST NOT be considered valid. This allows certificate revocation. An arbitrary number of these elements can be provided. Note that because certificates may expire, bad-node entries need only be present for the lifetime of the certificate. Technically

speaking, bad node-ids may be reused once their certificates have expired, the requirement for node-ids to be pseudo randomly generated gives this event a vanishing probability.

mandatory-extension This element contains the name of an XML namespace that a node joining the overlay MUST support. The presence of a mandatory-extension element does not require the extension to be used in the current configuration file, but can indicate that it may be used in the future. Note that the namespace is case-sensitive, as specified in [w3c-xml-namespaces] Section 2.3. More than one mandatory-extension element may be present.

Inside each overlay element, the required-kinds elements can also occur. This element indicates the kinds that members must support and contains multiple kind-block elements that each define a single kind that MUST be supported by nodes in the overlay. Each kind-block consists of a single kind element and a kind-signature. The kind element defines the kind. The kind-signature is the signature computed over the kind element.

Each kind has either an id attribute or a name attribute. The name attribute is a string representing the kind (the name registered to IANA) while the id is an integer kind-id allocated out of private space.

In addition, the kind element contains the following elements:

max-count: the maximum number of values which members of the overlay must support.

data-model: the data model to be used.

max-size: the maximum size of individual values.

access-control: the access control model to be used.

max-node-multiple: This is optional and only used when the access control is NODE-MULTIPLE. This indicates the maximum value for the i counter. This is an integer greater than 0.

All of the non optional values MUST be provided. If the kind is registered with IANA, the data-model and access-control elements MUST match those in the kind registration, and clients MUST ignore them in favor of the IANA versions. Multiple required-kinds elements MAY be present.

The kind-block element also MUST contain a "kind-signature" element. This signature is computed across the kind from the beginning of the first < of the kind to the end of the last > of the kind in the same way as the signature element described later in this section.

The configuration file is a binary file and cannot be changed - including whitespace changes - or the signature will break. The signature is computed by taking each configuration element and starting from, and including, the first < at the start of <configuration> up to and including the > in </configuration> and treating this as a binary blob that is signed using the standard SecurityBlock defined in Section 5.3.4. The SecurityBlock is base 64 encoded using the base64 alphabet from RFC[RFC4648] and put in the signature element following the configuration object in the configuration file.

When a node receives a new configuration file, it MUST change its configuration to meet the new requirements. This may require the node to exit the DHT and re-join. If a node is not capable of

supporting the new requirements, it MUST exit the overlay. If some information about a particular kind changes from what the node previously knew about the kind (for example the max size), the new information in the configuration files overrides any previously learned information. If any kind data was signed by a node that is no longer allowed to sign kinds, that kind MUST be discarded along with any stored information of that kind. Note that forcing an avalanche restart of the overlay with a configuration change that requires re-joining the overlay may result in serious performance problems, including total collapse of the network if configuration parameters are not properly considered. Such an event may be necessary in case of a compromised CA or similar problem, but for large overlays should be avoided in almost all circumstances.

10.1.1. Relax NG Grammar

The grammar for the configuration data is:

```
namespace chord = "urn:ietf:params:xml:ns:p2p:config-chord"
namespace local = ""
default namespace p2pcf = "urn:ietf:params:xml:ns:p2p:config-base"
namespace rng = "http://relaxng.org/ns/structure/1.0"
```

```
anything =
    (element * { anything }
     | attribute * { text }
     | text)*

foreign-elements = element * - (p2pcf:* | local:* | chord:*)
                   { anything }*
foreign-attributes = attribute * - (p2pcf:*|local:*|chord:*)
                     { text }*
foreign-nodes = (foreign-attributes | foreign-elements)*

start =  element p2pcf:overlay {
      overlay-element
}

overlay-element &=  element configuration {
            attribute instance-name { xsd:string },
            attribute expiration { xsd:dateTime }?,
            attribute sequence { xsd:long }?,
            foreign-attributes*,
            parameter
        }+
overlay-element &= element signature {
            attribute algorithm { signature-algorithm-type }?,
            xsd:base64Binary
```



```
        }*

signature-algorithm-type |= "rsa-sha1"
signature-algorithm-type |=  xsd:string # signature alg extensions

parameter &= element topology-plugin { topology-plugin-type }?
topology-plugin-type |= xsd:string # topo plugin extensions
parameter &= element max-message-size { xsd:unsignedInt }?
parameter &= element initial-ttl { xsd:int }?
parameter &= element root-cert { xsd:base64Binary }*
parameter &= element required-kinds { kind-block* }?
parameter &= element enrollment-server { xsd:anyURI }*
parameter &= element kind-signer {  xsd:string }*
parameter &= element configuration-signer {  xsd:string }*
parameter &= element bad-node {  xsd:string }*
```

```
   parameter &= element no-ice { xsd:boolean }?
   parameter &= element shared-secret { xsd:string }?
   parameter &= element overlay-link-protocol { xsd:string }*
   parameter &= element clients-permitted { xsd:boolean }?
   parameter &= element turn-density { xsd:unsignedByte }?
   parameter &= element node-id-length { xsd:int }?
   parameter &= element mandatory-extension { xsd:string }*
   parameter &= foreign-elements*

   parameter &=
       element self-signed-permitted {
           attribute digest { self-signed-digest-type },
           xsd:boolean
       }?
   self-signed-digest-type |= "sha1"
   self-signed-digest-type |=  xsd:string # signature digest extensions

   parameter &= element bootstrap-node {
                   attribute address { xsd:string },
                   attribute port { xsd:int }?
                }*

   parameter &= element multicast-bootstrap {
                   attribute address {  xsd:string },
                   attribute port { xsd:int }?
                }*

   kind-block = element kind-block {
       element kind {
           (  attribute name { kind-names }
              | attribute id { xsd:unsignedInt } ),
           kind-parameter
       } &
```



```
       element kind-signature  {
           attribute algorithm { signature-algorithm-type }?,
           xsd:base64Binary
       }?
   }

   kind-parameter &= element max-count { xsd:int }
   kind-parameter &= element max-size { xsd:int }
```

```
kind-parameter &= element max-node-multiple { xsd:int }?

kind-parameter &= element data-model { data-model-type }
data-model-type |= "SINGLE"
data-model-type |= "ARRAY"
data-model-type |= "DICTIONARY"
data-model-type |=  xsd:string # data model extensions

kind-parameter &= element access-control { access-control-type }
access-control-type |= "USER-MATCH"
access-control-type |= "NODE-MATCH"
access-control-type |= "USER-NODE-MATCH"
access-control-type |= "NODE-MULTIPLE"
access-control-type |= xsd:string # access control extensions

kind-parameter &= foreign-elements*

kind-names |= "TURN-SERVICE"
kind-names |= "CERTIFICATE_BY_NODE"
kind-names |= "CERTIFICATE_BY_USER"
kind-names |= xsd:string # kind extensions

# Chord specific parameters
topology-plugin-type |= "CHORD-RELOAD"
parameter &= element chord:chord-ping-interval { xsd:int }?
parameter &= element chord:chord-update-interval { xsd:int }?
parameter &= element chord:chord-reactive { xsd:boolean }?
```

10.2. Discovery Through Configuration Server

When a node first enrolls in a new overlay, it starts with a discovery process to find a configuration server.

The node MAY start by determines the overlay name. This value is provided by the user or some other out of band provisioning mechanism. The out of band mechanisms MAY also provide an optional URL for the configuration server. If a URL for the configuration server is not provided, the node MUST do a DNS SRV query using a Service name of "p2psip-enroll" and a protocol of TCP to find a

configuration server and form the URL by appending a path of "/.well-

known/p2psip-enroll" to the overlay name. This uses the "well known URI" framework defined in [RFC5785]. For example, if the overlay name was example.com, the URL would be "https://example.com//.well-known/p2psip-enroll".

Once an address and URL for the configuration server is determined, the peer forms an HTTPS connection to that IP address. The certificate MUST match the overlay name as described in [RFC2818]. Then the node MUST fetch a new copy of the configuration file. To do this, the peer performs a GET to the URL. The result of the HTTP GET is an XML configuration file described above, which replaces any previously learned configuration file for this overlay.

For overlays that do not use a configuration server, nodes obtain the configuration information needed to join the overlay through some out of band approach such an XML configuration file sent over email.

10.3. Credentials

If the configuration document contains a enrollment-server element, credentials are required to join the Overlay Instance. A peer which does not yet have credentials MUST contact the enrollment server to acquire them.

RELOAD defines its own trivial certificate request protocol. We would have liked to have used an existing protocol but were concerned about the implementation burden of even the simplest of those protocols, such as [RFC5272] and [RFC5273]. Our objective was to have a protocol which could be easily implemented in a Web server which the operator did not control (e.g., in a hosted service) and was compatible with the existing certificate handling tooling as used with the Web certificate infrastructure. This means accepting bare PKCS#10 requests and returning a single bare X.509 certificate. Although the MIME types for these objects are defined, none of the existing protocols support exactly this model.

The certificate request protocol is performed over HTTPS. The request is an HTTP POST with the following properties:

- If authentication is required, there is an URL parameter of "password" and "username" containing the user's name and password in the clear (hence the need for HTTPS)
- If more than one Node-ID is required, there is an URL parameter of "nodeids" containing the number of Node-IDs required.
- The body is of content type "application/pkcs10", as defined in [RFC2311].

- The Accept header contains the type "application/pkix-cert", indicating the type that is expected in the response.

The enrollment 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 generates and returns a certificate. The SubjectAltName field in the certificate contains the following values:

- One or more Node-IDs which MUST be cryptographically random [RFC4086]. Each MUST be chosen by the enrollment server in such a way that they are unpredictable to the requesting user. E.g., the user MUST NOT be informed of potential (random) Node-IDs prior to authenticating. Each is placed in the subjectAltName using the uniformResourceIdentifier type and MUST contain RELOAD URIs as described in Section 13.15 and MUST contain a Destination list with a single entry of type "node_id".
- A single name this user is allowed to use in the overlay, using type rfc822Name.

The certificate is returned as type "application/pkix-cert" as defined in [RFC2585], with an HTTP status code of 200 OK. Certificate processing errors should be treated as HTTP errors and have appropriate HTTP status codes.

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 node then reads the certificate to find the Node-IDs it can use.

10.3.1. Self-Generated Credentials

If the "self-signed-permitted" element is present in the configuration 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.

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 (more specifically the subjectPublicKeyInfo) and taking the high order bits. 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.

10.4. Searching for a Bootstrap Node

If no cached bootstrap nodes are available and the configuration file has an multicast-bootstrap element, then the node SHOULD send a Ping request over UDP to the address and port found to each multicast-bootstrap element found in the configuration document. This MAY be a multicast, broadcast, or anycast address. The Ping should use the wildcard Node-ID as the destination Node-ID.

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

10.5. Contacting a Bootstrap Node

In order to join the overlay, the joining node MUST contact a node in the overlay. Typically this means contacting the bootstrap nodes, since they are reachable by the local peer or have public IP addresses. If the joining node has cached a list of peers it has previously been connected with in this overlay, as an optimization it MAY attempt to use one or more of them as bootstrap nodes before falling back to the bootstrap nodes listed in the configuration file.

When contacting a bootstrap node, the joining node first forms the DTLS or TLS connection to the bootstrap node and then sends an Attach request over this connection with the destination Node-ID set to the joining node's Node-ID.

When the requester node finally does receive a response from some responding node, 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 5.4.

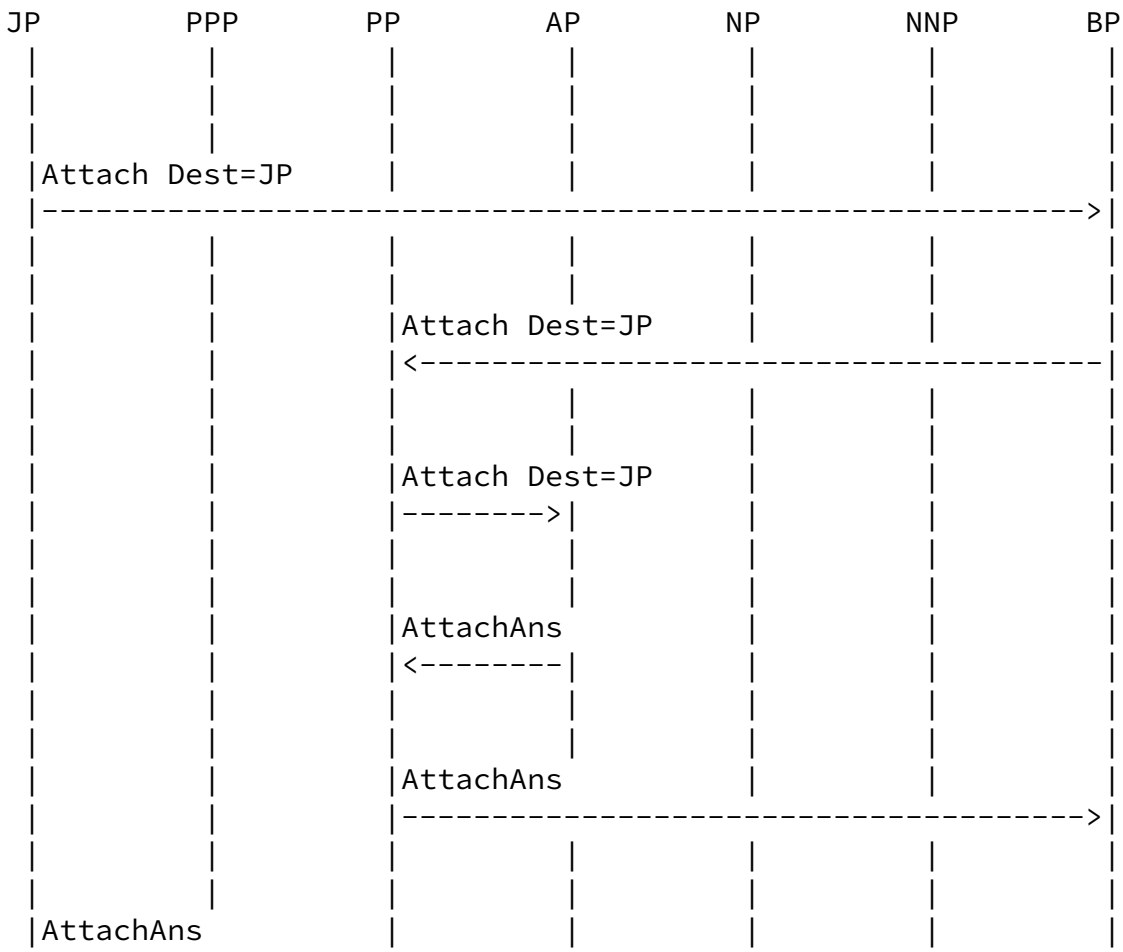
After a node has successfully joined the overlay network, it will

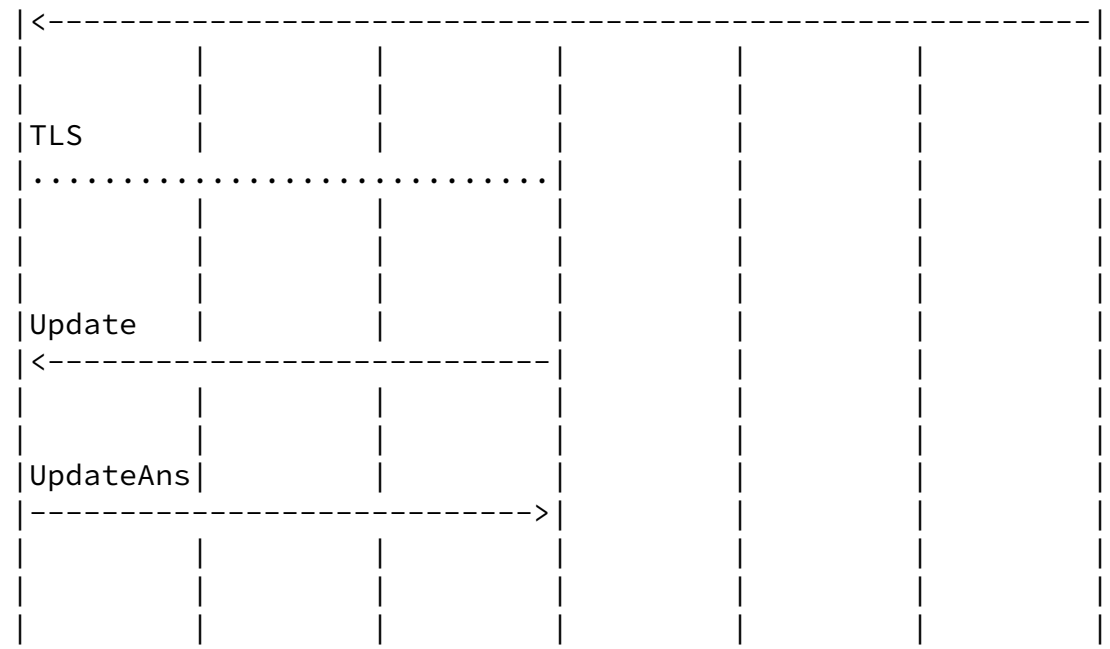
have direct connections to several peers. Some MAY be added to the cached bootstrap nodes list and used in future boots. Peers that are not directly connected MUST NOT be cached. The suggested number of peers to cache is 10. Algorithms for determining which peers to cache are beyond the scope of this specification.

11. Message Flow Example

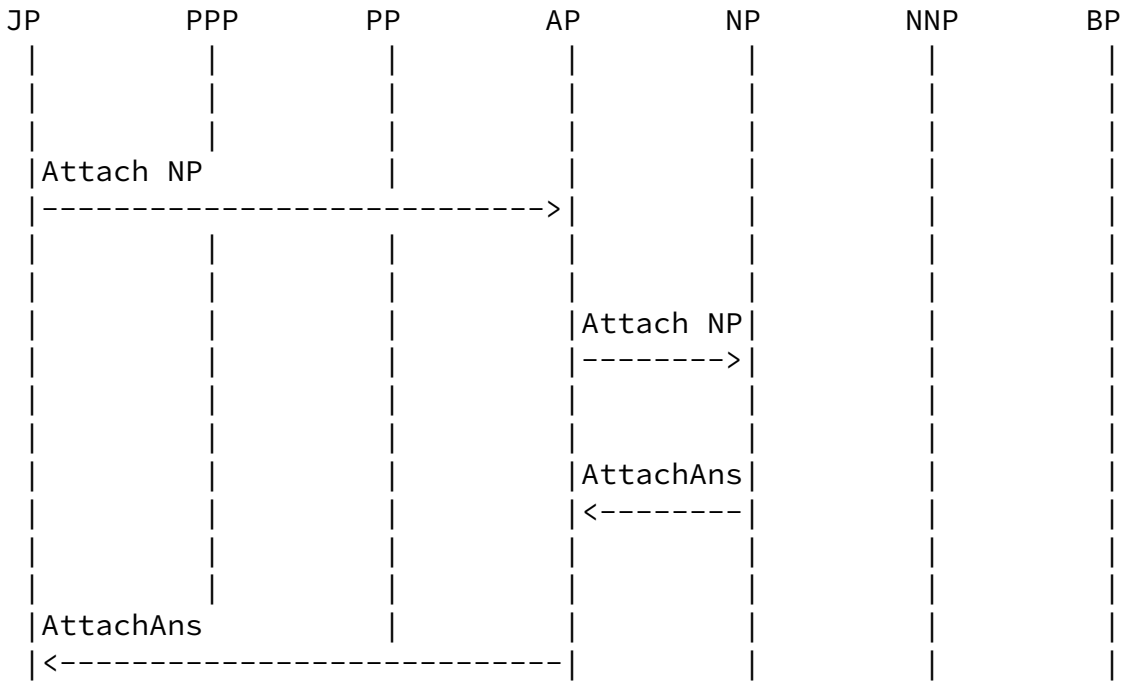
The following abbreviation are used in the message flow diagrams: JP = joining peer, AP = admitting peer, NP = next peer after the AP, NNP = next next peer which is the peer after NP, PP = previous peer before the AP, PPP = previous previous peer which is the peer before the PP, BP = bootstrap peer.

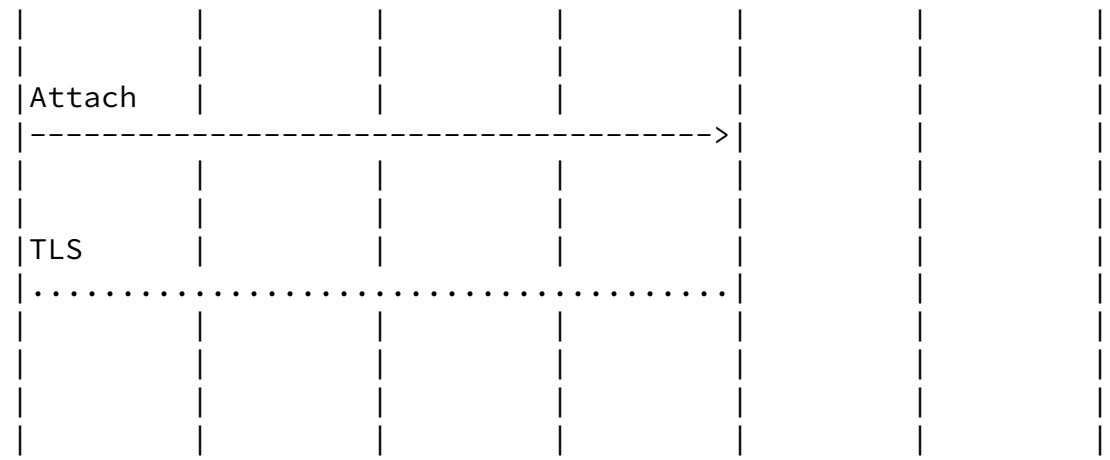
In the following example, we assume that JP has formed a connection to one of the bootstrap nodes. JP then sends an Attach through that peer to a resource ID of itself (JP). It gets routed to the admitting peer (AP) because JP is not yet part of the overlay. When AP responds, JP and AP use ICE to set up a connection and then set up TLS. Once AP has connected to JP, AP sends to JP an Update to populate its Routing Table. The following example shows the Update happening after the TLS connection is formed but it could also happen before in which case the Update would often be routed through other nodes.


JP
PPP
PP
AP
NP
NNP
BP
Attach Dest=JP
Attach Dest=JP
Attach Dest=JP
AttachAns
AttachAns
AttachAns

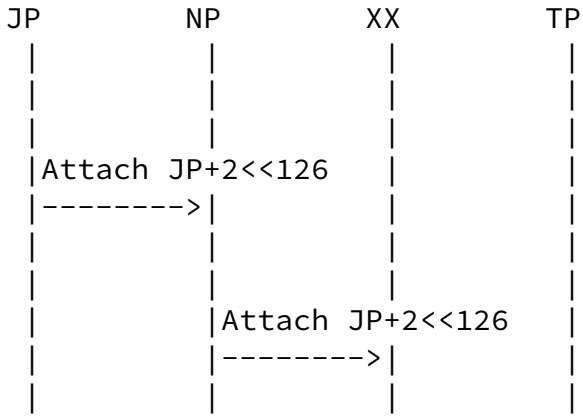


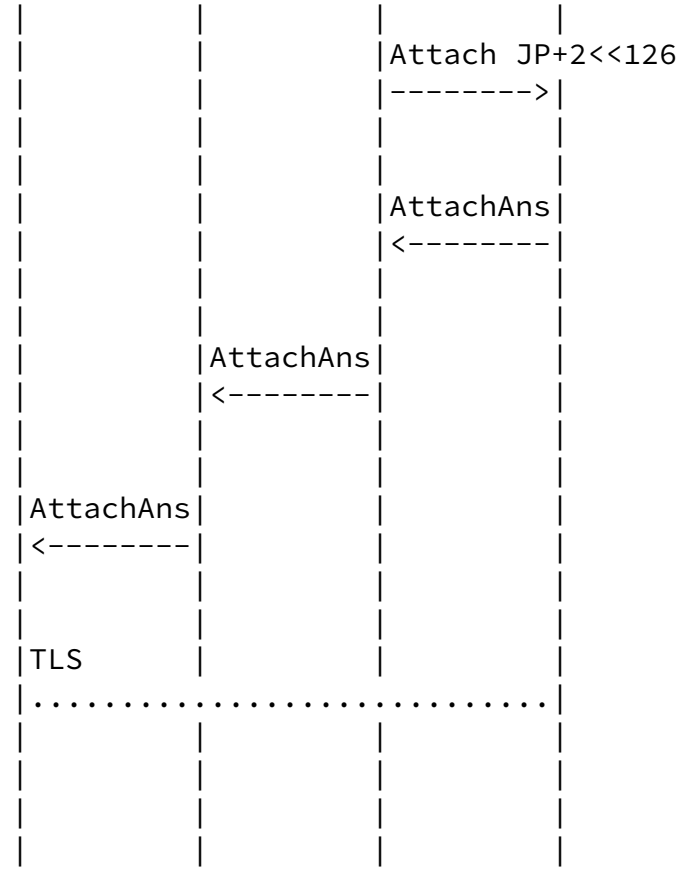
The JP then forms connections to the appropriate neighbors, such as NP, by sending an Attach which gets routed via other nodes. When NP responds, JP and NP use ICE and TLS to set up a connection.



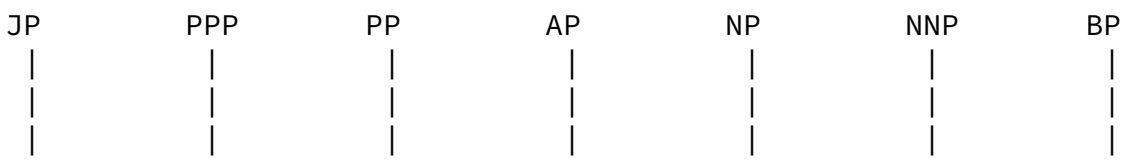


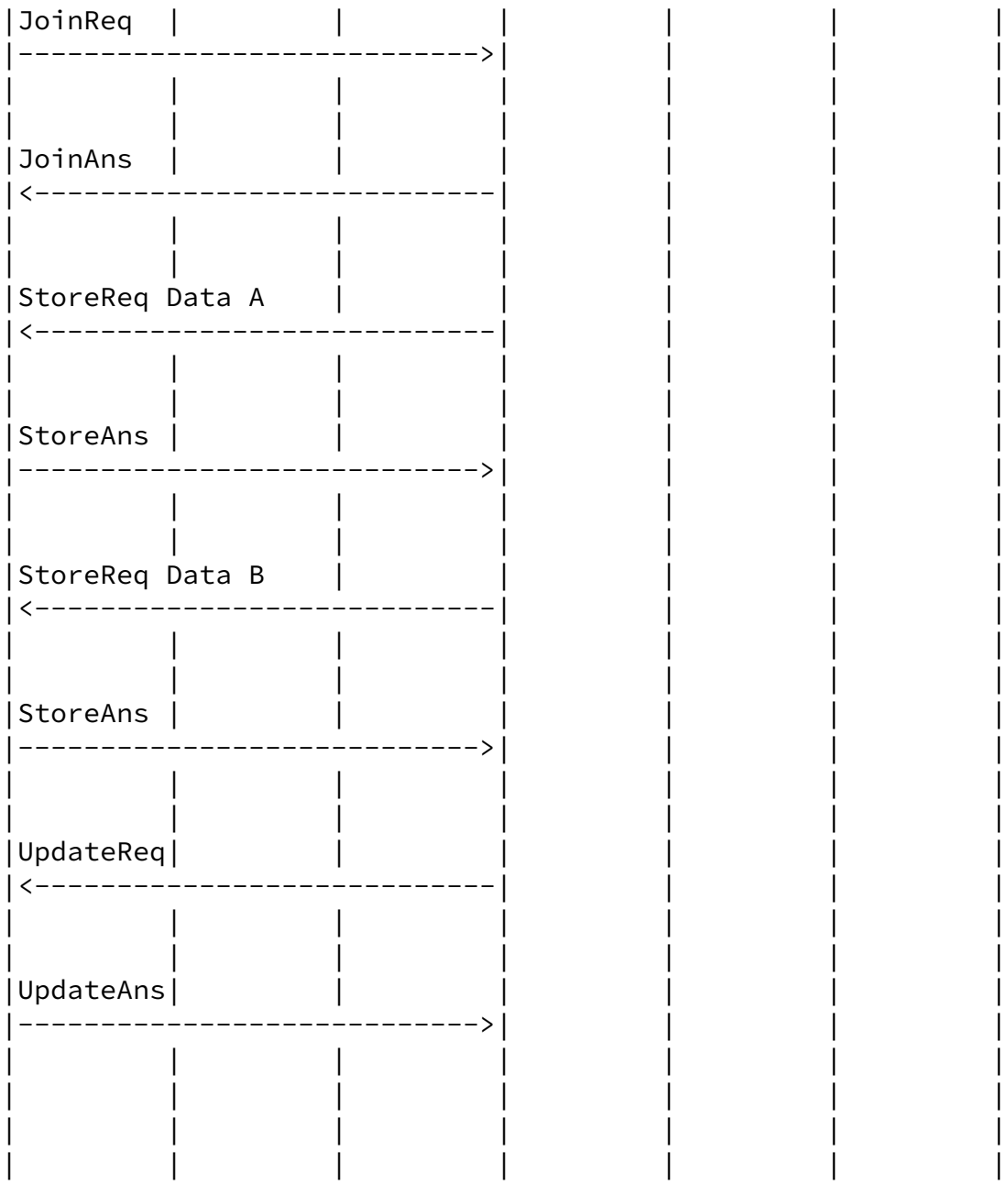
JP also needs to populate its finger table (for the Chord based DHT). It issues an Attach to a variety of locations around the overlay. The diagram below shows it sending an Attach halfway around the Chord ring to the JP + 2^127.



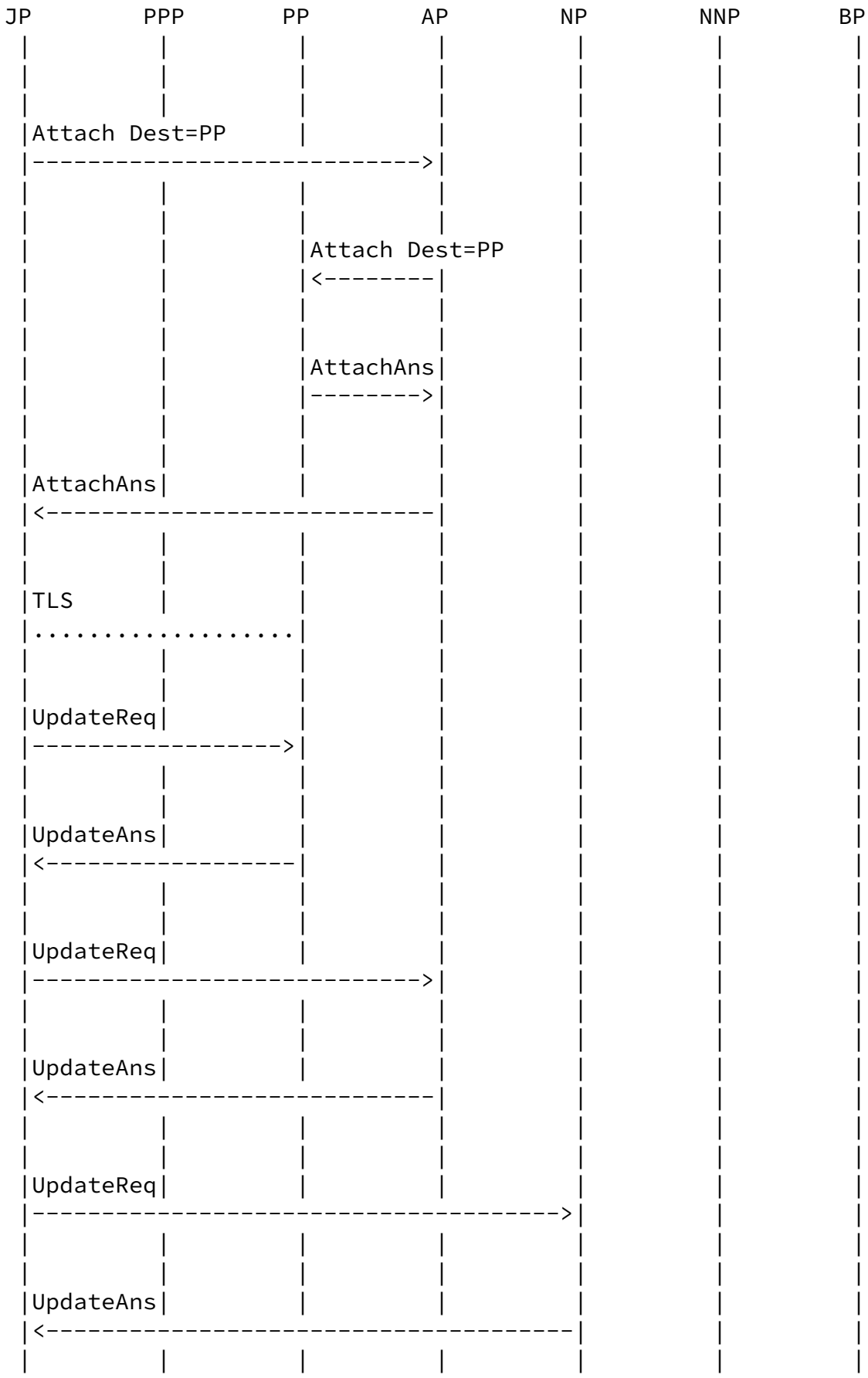


Once JP has a reasonable set of connections, it is ready to take its place in the DHT. It does this by sending a Join to AP. AP does a series of Store requests to JP to store the data that JP will be responsible for. AP then sends JP an Update explicitly labeling JP as its predecessor. At this point, 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.

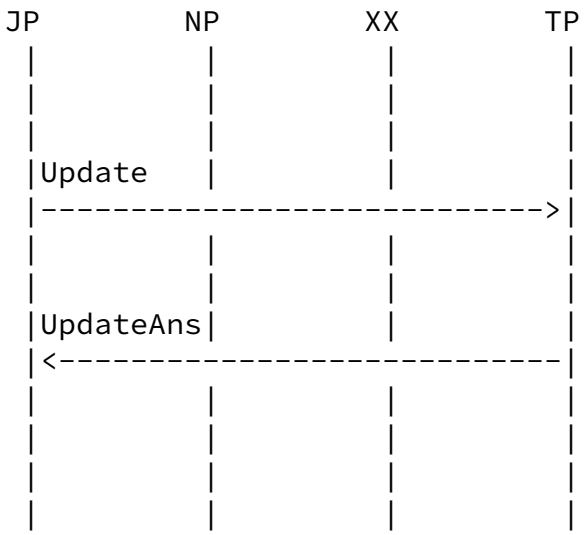




In Chord, JP's neighbor table needs to contain its own predecessors. It couldn't connect to them previously because it did not yet know their addresses. However, now that it has received an Update from AP, it has AP's predecessors, which are also its own, so it sends Attaches to them. Below it is shown connecting to AP's 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.



12. Security Considerations

12.1. 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. More background information can be found in [RFC5765].

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.

12.2. 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 third peer to fetch this stored data. Other 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 specification 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.

12.3. 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 Node-ID are placed in the certificate, along with the user's public key.

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

- As a user, storing data at specific Resource-IDs in the Overlay Instance corresponding to the user name.
- As a overlay peer with the Node-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 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 may be appropriate for some small deployments, such as a small office or an ad hoc overlay set up among participants in a meeting where all hosts on the network are trusted. 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 allow any consumer of the data to verify the provenance and integrity of the data when it retrieves it.

Note that RELOAD does not itself provide a revocation/status

mechanism (though certificates may of course include OCSP responder information). Thus, certificate lifetimes should be chosen to balance the compromise window versus the cost of certificate renewal. Because RELOAD is already designed to operate in the face of some fraction of malicious peers, this form of compromise is not fatal.

All implementations MUST implement certificate-based security.

12.4. 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.

12.5. Storage Security

When certificate-based security is used in RELOAD, any given Resource-ID/Kind-ID pair is bound to some small set of certificates. In order to write data, the writer must prove possession of the private key for one of those certificates. Moreover, all data is stored, signed with the same private key that was used to authorize the storage. This set of rules makes questions of authorization and data integrity - which have historically been thorny for overlays - relatively simple.

12.5.1. Authorization

When a client wants to store some value, 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 at this Resource-ID/Kind-ID pair. Determining this requires comparing the user's identity to the requirements of the access control model (see Section 6.3). If it satisfies those requirements the user is authorized to write, 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. Because SIP Location uses the USER-NODE-MATCH policy, it first verifies that the user name in the certificate hashes to the requested Resource-ID. It then verifies that the Node-Id in the certificate matches the dictionary key being used for the store. If both of these checks succeed, the Store is authorized. Note that because the access control model is different for different kinds, the exact set of checks will vary.

12.5.2. Distributed Quota

Being a peer in an Overlay Instance carries with it the responsibility to store data for a given region of the Overlay Instance. However, allowing clients to store unlimited amounts of data would create unacceptable burdens on peers and would also enable 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

Resource-ID/Kind-ID pair is bound to a small set of certificates, these size restrictions also create a distributed quota mechanism, with the quotas administered by the central configuration server.

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

12.5.3. Correctness

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

12.5.4. Residual Attacks

The mechanisms described here provides 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 has 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.

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 effects 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 trade off that must be made when designing any usage.

12.6. Routing Security

Because the storage security system guarantees (within limits) the integrity of the stored data, routing security focuses on stopping the attacker from performing a DOS attack that misroutes 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.

12.6.1. Background

In general, attacks on DHT routing are mounted by the attacker arranging to route traffic through one 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 Node-IDs. The Sybil attack requires the creation of a large number of peers. The Eclipse attack requires that the attacker be able to impersonate specific peers. In both cases, these attacks are limited by the use of centralized, certificate-based admission control.

12.6.2. Admissions Control

Admission to a RELOAD Overlay Instance is controlled by requiring that each peer have a certificate containing its Node-Id. The requirement to have a certificate is enforced by using certificate-based mutual authentication on each connection. (Note: the following only applies when self-signed certificates are not used.) Whenever a peer connects to another peer, each side automatically checks that the other has a suitable certificate. These Node-Ids are randomly assigned by the central enrollment server. This has two benefits:

- It allows the enrollment server to limit the number of Node-IDs issued to any individual user.
- It prevents the attacker from choosing specific Node-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 Node-Ids, which requires him to repeatedly query the enrollment server for new certificates, which will match only 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.

12.6.3. 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.

Replay is typically prevented for messages that impact the topology

of the overlay by having the information come directly, or be verified by, the nodes that claimed to have generated the update. Data storage replay detection is done by signing time of the node that generated the signature on the store request thus providing a time based replay protection but the time synchronization is only needed between peers that can write to the same location.

12.6.4. Protecting the Signaling

The goal here is to stop an attacker from knowing who is signaling what to whom. An attacker is unlikely to be able to observe the activities of a specific individual 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 is to digitally sign each message. This prevents adversarial peers from modifying messages in flight, even if they are on the routing path.

12.6.5. Residual Attacks

The routing security mechanisms in RELOAD are designed to contain rather than eliminate attacks on routing. It is still possible for an attacker to mount a variety of attacks. In particular, if an attacker is able to take up a position on the 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 an 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.

13. 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.]

13.1. Well-Known URI Registration

IANA will make the following "Well Known URI" registration as described in [RFC5785]:

[[Note to RFC Editor - this paragraph can be removed before publication.]] A review request was sent to wellknown-uri-review@ietf.org on October 12, 2010.

URI suffix:	p2psip-enroll
Change controller:	IETF <iesg@ietf.org>
Specification document(s):	[RFC-AAAA]
Related information:	None

13.2. Port Registrations

[[Note to RFC Editor - this paragraph can be removed before publication.]] IANA has already allocated a TCP port for the main peer to peer protocol. This port has the name p2p-sip and the port number of 6084. IANA needs to update this registration to be defined for UDP as well as TCP.

IANA will make the following port registration:

Registration Technical Contact	Cullen Jennings <fluffy@cisco.com>

Registration Owner	IETF <iesg@ietf.org>
Transport Protocol	TCP & UDP
Port Number	6084
Service Name	p2psip-enroll
Description	Peer to Peer Infrastructure Enrollment
Reference	[RFC-AAAA]

13.3. 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-RELOAD	RFC-AAAA

13.4. Access Control Policies

IANA SHALL create a "RELOAD Access Control Policy" Registry. Entries in this registry are strings denoting access control policies, as described in Section 6.3. New entries in this registry SHALL be registered via RFC 5226 Standards Action. The initial contents of this registry are:

Access Policy	RFC
USER-MATCH	RFC-AAAA
NODE-MATCH	RFC-AAAA
USER-NODE-MATCH	RFC-AAAA
NODE-MULTIPLE	RFC-AAAA

+-----------------+----------+

13.5. Application-ID

IANA SHALL create a "RELOAD Application-ID" Registry. Entries in this registry are 16-bit integers denoting application kinds. Code points in the range 0x0001 to 0x7fff SHALL be registered via RFC 5226 Standards Action. Code points in the range 0x8000 to 0xf000 SHALL be registered via RFC 5226 Expert Review. Code points in the range 0xf001 to 0xfffe are reserved for private use. The initial contents of this registry are:

Application	Application-ID	Specification
INVALID	0	RFC-AAAA
SIP	5060	Reserved for use by SIP Usage
SIP	5061	Reserved for use by SIP Usage
Reserved	0xffff	RFC-AAAA

13.6. 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 4.2. Code points in the range 0x00000001 to 0x7fffffff SHALL be registered via RFC 5226 Standards Action. Code points in the range 0x8000000 to 0xf0000000 SHALL be registered via RFC 5226 Expert Review. Code points in the range 0xf0000001 to 0xfffffffe are reserved for private use via the kind description mechanism described in Section 10. The initial contents of this registry are:

Kind	Kind-ID	RFC
INVALID	0	RFC-AAAA
TURN_SERVICE	2	RFC-AAAA
CERTIFICATE_BY_NODE	3	RFC-AAAA
CERTIFICATE_BY_USER	16	RFC-AAAA
Reserved	0x7fffffff	RFC-AAAA
Reserved	0xfffffffe	RFC-AAAA

13.7. Data Model

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

Data Model	RFC
INVALID	RFC-AAAA
SINGLE	RFC-AAAA
ARRAY	RFC-AAAA
DICTIONARY	RFC-AAAA
RESERVED	RFC-AAAA

13.8. 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 5.3.3. 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

unused (was remove_req)	11	RFC-AAAA
unused (was 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
unused (was attachlite_req)	27	RFC-AAAA
unused (was attachlite_ans)	28	RFC-AAAA
app_attach_req	29	RFC-AAAA
app_attach_ans	30	RFC-AAAA
unused (was app_attachlite_req)	31	RFC-AAAA
unused (was app_attachlite_ans)	32	RFC-AAAA
config_update_req	33	RFC-AAAA
config_update_ans	34	RFC-AAAA
reserved	0x8000..0xfffe	RFC-AAAA
error	0xffff	RFC-AAAA

13.9. 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
Unused	1	RFC-AAAA
Error_Forbidden	2	RFC-AAAA
Error_Not_Found	3	RFC-AAAA

Error_Request_Timeout	4	RFC-AAAA
Error_Generation_Counter_Too_Low	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
Error_TTL_Exceeded	10	RFC-AAAA
Error_Message_Too_Large	11	RFC-AAAA
Error_Unknown_Kind	12	RFC-AAAA
Error_Unknown_Extension	13	RFC-AAAA
Error_Response_Too_Large	14	RFC-AAAA
Error_Config_Too_Old	15	RFC-AAAA
Error_Config_Too_New	16	RFC-AAAA
Error_In_Progress	17	RFC-AAAA
reserved	0x8000..0xfffe	RFC-AAAA

13.10. Overlay Link Types

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

Protocol	Code	Specification
reserved	0	RFC-AAAA
DTLS-UDP-SR	1	RFC-AAAA
DTLS-UDP-SR-NO-ICE	3	RFC-AAAA
TLS-TCP-FH-NO-ICE	4	RFC-AAAA
reserved	255	RFC-AAAA

13.11. Overlay Link Protocols

IANA shall create an "Overlay Link Protocol Registry". Entries in this registry SHALL be defined via RFC 5226 Standards Action. This registry SHALL be initially populated with the following value: "TLS".

13.12. 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

13.13. 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
num_resources	2	RFC-AAAA
uptime	3	RFC-AAAA
reserved	255	RFC-AAAA

13.14. Message Extensions

IANA shall create a "RELOAD Extensions Registry". Entries in this registry SHALL be defined via RFC 5226 Specification Required. This registry SHALL be initially populated with the following values:

Extensions Name	Code	Specification
invalid	0	RFC-AAAA
reserved	0xFFFF	RFC-AAAA

13.15. reload URI Scheme

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

- A peer.
- A resource inside a peer.

The reload URI is defined using a subset of the URI schema specified in Appendix A of RFC 3986 [RFC3986] and the associated URI Guidelines [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 (i.e., multiple concatenated Destination objects with no length prefix prior to the object as a whole.)

overlay: the name of the overlay.

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

If no specifier is present then 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.

13.15.1. URI Registration

[[Note to RFC Editor - please remove this paragraph before publication.]] A review request was sent to uri-review@ietf.org on Oct 7, 2010.

The following summarizes the information necessary to register the

reload URI.

URI Scheme Name: reload
Status: permanent
URI Scheme Syntax: see Section 13.15 of RFC-AAAA
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, so it is encoded entirely in US-ASCII.
Applications/protocols that use this URI scheme: The RELOAD protocol described in RFC-AAAA.
Interoperability considerations: See RFC-AAAA.
Security considerations: See RFC-AAAA
Contact: Cullen Jennings <fluffy@cisco.com>
Author/Change controller: IESG
References: RFC-AAAA

13.16. Media Type Registration

[[Note to RFC Editor - please remove this paragraph before publication.]] A review request was sent to ietf-types@iana.org on May 27, 2011.

Type name: application

Subtype name: p2p-overlay+xml

Required parameters: none

Optional parameters: none

Encoding considerations: Must be binary encoded.

Security considerations: This media type is typically not used to transport information that typically needs to be kept confidential however there are cases where it is integrity of the information is important. For these cases using a digital signature is RECOMMENDED. One way of doing this is specified in RFC-AAAA. In the case when the

media includes a "shared-secret" element, then the contents of the file need to be kept confidential or else anyone that can see the shared-secret and effect the RELOAD overlay network.

Interoperability considerations: No knows interoperability consideration beyond those identified for application/xml in [RFC3023].

Published specification: RFC-AAAA

Applications that use this media type: The type is used to configure the peer to peer overlay networks defined in RFC-AAAA.

Additional information: The syntax for this media type is specified in Section 10.1 of RFC-AAAA. The contents MUST be valid XML compliant with the relax NG grammar specified in RFC-AAAA and use the UTF-8[RFC3629] character encoding.

Magic number(s): none

File extension(s): relo

Macintosh file type code(s): none

Person & email address to contact for further information: Cullen Jennings <c.jennings@ieee.org>

Intended usage: COMMON

Restrictions on usage: None

Author: Cullen Jennings <c.jennings@ieee.org>

Change controller: IESG

14. Acknowledgments

This specification 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 authors of RFC 5389 for text included from that. Vidya Narayanan provided many comments and improvements.

The ideas and text for the Chord specific extension data to the Leave mechanisms was provided by Jouni Maenpaa, Gonzalo Camarillo, and Jani Hautakorpi.

Thanks to the many people who contributed including Ted Hardie, Michael Chen, Dan York, Das Saumitra, Lyndsay Campbell, Brian Rosen, David Bryan, Dave Craig, and Julian Cain. Extensive working last call comments were provided by: Jouni Maenpaa, Roni Even, Gonzalo Camarillo, Ari Keranen, John Buford, Michael Chen, Frederic-Philippe Met, and David Bryan. Special thanks to Marc Petit-Huguenin who provied an amazing amount to detailed review.

15. References

15.1. Normative References

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

[RFC2585] Housley, R. and P. Hoffman, "Internet X.509 Public Key Infrastructure Operational Protocols: FTP and HTTP", RFC 2585, May 1999.

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

[RFC3023] Murata, M., St. Laurent, S., and D. Kohn, "XML Media Types", RFC 3023, January 2001.

[RFC3174] Eastlake, D. and P. Jones, "US Secure Hash Algorithm 1 (SHA1)", RFC 3174, September 2001.

[RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

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

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

[RFC4395] Hansen, T., Hardie, T., and L. Masinter, "Guidelines and Registration Procedures for New URI Schemes", BCP 35, RFC 4395, February 2006.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[RFC5245] Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", RFC 5245, April 2010.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

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

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

[RFC5348] Floyd, S., Handley, M., Padhye, J., and J. Widmer, "TCP Friendly Rate Control (TFRC): Protocol Specification", RFC 5348, September 2008.

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

October 2008.

[RFC5766] Mahy, R., Matthews, P., and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)", RFC 5766, April 2010.

[RFC5952] Kawamura, S. and M. Kawashima, "A Recommendation for IPv6 Address Text Representation", RFC 5952, August 2010.

[RFC6091] Mavrogiannopoulos, N. and D. Gillmor, "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication", RFC 6091, February 2011.

[RFC6234] Eastlake, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, May 2011.

[RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, June 2011.

[w3c-xml-namespaces]
Bray, T., Hollander, D., Layman, A., Tobin, R., and Henry S. , "Namespaces in XML 1.0 (Third Edition)".

15.2. Informative References

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

[Eclipse] Singh, A., Ngan, T., Druschel, T., and D. Wallach,

"Eclipse Attacks on Overlay Networks: Threats and Defenses", INFOCOM 2006, April 2006.

[I-D.ietf-hip-reload-instance]
Keranen, A., Camarillo, G., and J. Maenpaa, "Host Identity Protocol-Based Overlay Networking Environment (HIP BONE) Instance Specification for REsource LOcation And Discovery (RELOAD)", draft-ietf-hip-reload-instance-03 (work in

progress), January 2011.

[I-D.ietf-mmusic-ice-tcp]
Rosenberg, J., Keranen, A., Lowekamp, B., and A. Roach, "TCP Candidates with Interactive Connectivity Establishment (ICE)", draft-ietf-mmusic-ice-tcp-13 (work in progress), April 2011.

[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-03 (work in progress), October 2010.

[I-D.ietf-p2psip-self-tuning]
Maenpaa, J., Camarillo, G., and J. Hautakorpi, "A Self-tuning Distributed Hash Table (DHT) for REsource LOcation And Discovery (RELOAD)", draft-ietf-p2psip-self-tuning-04 (work in progress), July 2011.

[I-D.ietf-p2psip-service-discovery]
Maenpaa, J. and G. Camarillo, "Service Discovery Usage for REsource LOcation And Discovery (RELOAD)", draft-ietf-p2psip-service-discovery-03 (work in progress), July 2011.

[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-05 (work in progress), July 2010.

[I-D.jiang-p2psip-relay]
Jiang, X., Zong, N., Even, R., and Y. Zhang, "An extension to RELOAD to support Direct Response and Relay Peer routing", draft-jiang-p2psip-relay-05 (work in progress), March 2011.

[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.

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

[RFC2311] Dusse, S., Hoffman, P., Ramsdell, B., Lundblade, L., and L. Repka, "S/MIME Version 2 Message Specification", RFC 2311, March 1998.

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

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

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

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

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

[RFC5201] Moskowitz, R., Nikander, P., Jokela, P., and T. Henderson, "Host Identity Protocol", RFC 5201, April 2008.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.

[RFC5694] Camarillo, G. and IAB, "Peer-to-Peer (P2P) Architecture: Definition, Taxonomies, Examples, and Applicability", RFC 5694, November 2009.

[RFC5765] Schulzrinne, H., Marocco, E., and E. Ivov, "Security Issues and Solutions in Peer-to-Peer Systems for Realtime Communications", RFC 5765, February 2010.

[RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, April 2010.

[RFC6079] Camarillo, G., Nikander, P., Hautakorpi, J., Keranen, A., and A. Johnston, "HIP BONE: Host Identity Protocol (HIP) Based Overlay Networking Environment (BONE)", RFC 6079, January 2011.

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

[UnixTime] Wikipedia, "Unix Time", <http:/wikipedia.org/wiki/Unix_time>.

[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.

[handling-churn-usenix04] Rhea, S., Geels, D., Roscoe, T., and J. Kubiatowicz, "Handling Churn in a DHT", In Proc. of the USENIX Annual Technical Conference June 2004 USENIX 2004.

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

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

[non-transitive-dhts-worlds05] Freedman, M., Lakshminarayanan, K., Rhea, S., and I. Stoica, "Non-Transitive Connectivity and DHTs", WORLDS'05.

[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.

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

Appendix A. Routing Alternatives

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

A.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 RouteQuery mechanism and is primarily intended for debugging. It 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.

A.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 were a new message initiated 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 ones are, because the forward path is more likely to have a failed peer than is 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.

A.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.

- 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.
- 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.
- 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, it is

absolutely necessary to establish a secure connection to authenticate A before delivering the response. This step diminishes the efficiency of direct response because multiple roundtrips are required before the message can be delivered.

- 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 both be 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.

A.4. Relay Peers

[I-D.jiang-p2psip-relay] 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 makes random relay peers no more likely to work than the responding peer.

A.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 B. 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.

B.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:

- The node does not have appropriate network connectivity, typically because it has a low-bandwidth network connection.
- The node may not have sufficient resources, such as computing power, storage space, or battery power.
- The overlay algorithm may dictate specific requirements for peer selection. These may include participating in the overlay to determine trustworthiness; controlling the number of peers in the overlay to reduce overly-long routing paths; or ensuring 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 Attach 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.

B.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.

Appendix C. Change Log

C.1. Changes since draft-ietf-p2psip-reload-13

Note to RFC Editor: Please remove this section.

- Fixed finger formula.
- Support for getting a certificate with multiple Node-IDs.
- Added configuration signer.
- Added way to specify mandatory extensions in configuration file.

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)
Skype
Palo Alto, CA
USA

Email: bbl@lowekamp.net

Eric Rescorla
RTFM, Inc.
2064 Edgewood Drive
Palo Alto, CA 94303

USA

Phone: +1 650 678 2350
Email: ekr@rtfm.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