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.

Status of this Memo

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

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

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

This Internet-Draft will expire on July 20, 2012.
# Table of Contents

1. Introduction ................................................. 8
   1.1. Basic Setting ........................................ 9
   1.2. Architecture ......................................... 10
      1.2.1. Usage Layer ...................................... 13
      1.2.2. Message Transport .............................. 14
      1.2.3. Storage ......................................... 15
      1.2.4. Topology Plugin ............................... 15
      1.2.5. Forwarding and Link Management Layer ........ 16
   1.3. Security ............................................. 17
   1.4. Structure of This Document ........................... 18
2. Terminology .................................................. 18
3. Overlay Management Overview ................................. 21
   3.1. Security and Identification .......................... 21
       3.1.1. Shared-Key Security .......................... 23
   3.2. Clients ............................................... 23
       3.2.1. Client Routing .................................. 24
       3.2.2. Minimum Functionality Requirements for Clients . . 24
   3.3. Routing ................................................ 25
   3.4. Connectivity Management ............................... 28
   3.5. Overlay Algorithm Support ............................. 28
       3.5.1. Support for Pluggable Overlay Algorithms ....... 29
       3.5.2. Joining, Leaving, and Maintenance Overview . . 29
   3.6. First-Time Setup ...................................... 30
       3.6.1. Initial Configuration ............................ 31
       3.6.2. Enrollment ....................................... 31
4. Application Support Overview ................................. 31
   4.1. Data Storage ........................................... 32
       4.1.1. Storage Permissions .............................. 33
       4.1.2. Replication ...................................... 34
   4.2. Usages .................................................. 34
   4.3. Service Discovery ...................................... 35
   4.4. Application Connectivity ............................... 35
5. Overlay Management Protocol ................................. 35
   5.1. Message Receipt and Forwarding ......................... 36
       5.1.1. Responsible ID ................................... 36
       5.1.2. Other ID ......................................... 37
       5.1.3. Opaque ID ........................................ 39
   5.2. Symmetric Recursive Routing ............................ 39
       5.2.1. Request Origination .............................. 39
       5.2.2. Response Origination ............................ 40
   5.3. Message Structure ...................................... 40
       5.3.1. Presentation Language ........................... 41
       5.3.1.1. Common Definitions ............................ 42
       5.3.2. Forwarding Header ............................... 44
       5.3.2.1. Processing Configuration Sequence Numbers ... 46
       5.3.2.2. Destination and Via Lists .................... 47
5.3.2.3. Forwarding Options .......................... 49
5.3.3. Message Contents Format ........................ 50
5.3.3.1. Response Codes and Response Errors .......... 51
5.3.4. Security Block ................................. 53
5.4. Overlay Topology .................................. 57
5.4.1. Topology Plugin Requirements .................... 57
5.4.2. Methods and types for use by topology plugins ... 58
  5.4.2.1. Join ..................................... 58
  5.4.2.2. Leave .................................... 59
  5.4.2.3. Update .................................... 59
  5.4.2.4. RouteQuery ................................. 60
  5.4.2.5. Probe ..................................... 61
5.5. Forwarding and Link Management Layer .............. 63
  5.5.1. Attach ....................................... 63
    5.5.1.1. Request Definition ......................... 64
    5.5.1.2. Response Definition ........................ 67
    5.5.1.3. Using ICE With RELOAD ..................... 68
    5.5.1.4. Collecting STUN Servers .................... 68
    5.5.1.5. Gathering Candidates ....................... 69
    5.5.1.6. Prioritizing Candidates .................... 69
    5.5.1.7. Encoding the Attach Message ................ 70
    5.5.1.8. Verifying ICE Support ...................... 70
    5.5.1.9. Role Determination .......................... 71
    5.5.1.10. Full ICE ................................ 71
    5.5.1.11. No-ICE .................................. 71
    5.5.1.12. Subsequent Offers and Answers ............ 72
    5.5.1.13. Sending Media ............................ 72
    5.5.1.14. Receiving Media ........................... 72
  5.5.2. AppAttach ................................... 72
    5.5.2.1. Request Definition ......................... 72
    5.5.2.2. Response Definition ......................... 73
  5.5.3. Ping ......................................... 74
    5.5.3.1. Request Definition ......................... 74
    5.5.3.2. Response Definition ........................ 74
  5.5.4. ConfigUpdate .................................. 75
    5.5.4.1. Request Definition ......................... 75
    5.5.4.2. Response Definition ........................ 76
5.6. Overlay Link Layer ................................ 76
  5.6.1. Future Overlay Link Protocols ................. 78
    5.6.1.1. HIP ..................................... 78
    5.6.1.2. ICE-TCP .................................. 79
    5.6.1.3. Message-oriented Transports ................ 79
    5.6.1.4. Tunneled Transports ........................ 79
  5.6.2. Framing Header ................................ 79
  5.6.3. Simple Reliability ................................ 81
    5.6.3.1. Stop and Wait Sender Algorithm ................ 82
  5.6.4. DTLS/UDP with SR .............................. 83
  5.6.5. TLS/TCP with FH, No-ICE ....................... 83
5.6.6. DTLS/UDP with SR, No-ICE .......................... 83
5.7. Fragmentation and Reassembly .......................... 84
6. Data Storage Protocol .................................. 85
   6.1. Data Signature Computation ......................... 86
   6.2. Data Models ........................................ 87
      6.2.1. Single Value ................................... 88
      6.2.2. Array ........................................... 88
      6.2.3. Dictionary ..................................... 89
   6.3. Access Control Policies ............................ 89
      6.3.1. USER-MATCH ................................... 90
      6.3.2. NODE-MATCH ................................... 90
      6.3.3. USER-NODE-MATCH ................................ 90
      6.3.4. NODE-MULTIPLE .................................. 90
   6.4. Data Storage Methods ............................... 91
      6.4.1. Store ........................................... 91
         6.4.1.1. Request Definition ......................... 91
         6.4.1.2. Response Definition ......................... 95
         6.4.1.3. Removing Values ............................. 97
      6.4.2. Fetch ............................................ 97
         6.4.2.1. Request Definition ......................... 98
         6.4.2.2. Response Definition ......................... 100
      6.4.3. Stat ............................................. 101
         6.4.3.1. Request Definition ......................... 101
         6.4.3.2. Response Definition ......................... 101
      6.4.4. Find ............................................. 103
         6.4.4.1. Request Definition ......................... 103
         6.4.4.2. Response Definition ......................... 104
      6.4.5. Defining New Kinds .............................. 105
   7. Certificate Store Usage ............................... 105
8. TURN Server Usage ....................................... 106
9. Chord Algorithm ......................................... 108
   9.1. Overview ........................................... 109
   9.2. Hash Function ...................................... 109
   9.3. Routing ............................................. 109
   9.4. Redundancy ......................................... 110
   9.5. Joining ............................................. 110
   9.6. Routing Attaches .................................... 111
   9.7. Updates ............................................. 111
      9.7.1. Handling Neighbor Failures ....................... 113
      9.7.2. Handling Finger Table Entry Failure ............. 114
      9.7.3. Receiving Updates ................................ 114
      9.7.4. Stabilization .................................... 115
         9.7.4.1. Updating neighbor table ..................... 115
         9.7.4.2. Refreshing finger table ...................... 115
         9.7.4.3. Adjusting finger table size ................. 116
         9.7.4.4. Detecting partitioning ....................... 117
      9.8. Route query ........................................ 117
      9.9. Leaving ............................................. 118
10. Enrollment and Bootstrap ............................... 119
   10.1. Overlay Configuration .............................. 119
      10.1.1. Relax NG Grammar ............................. 125
   10.2. Discovery Through Configuration Server .............. 128
   10.3. Credentials .................................. 128
      10.3.1. Self-Generated Credentials .................. 130
   10.4. Searching for a Bootstrap Node ..................... 130
   10.5. Contacting a Bootstrap Node ....................... 130
11. Message Flow Example ................................ 131
12. Security Considerations ............................... 137
   12.1. Overview .................................. 137
   12.2. Attacks on P2P Overlays ........................... 138
   12.4. Shared-Secret Security ............................. 139
   12.5. Storage Security ................................ 140
      12.5.1. Authorization ................................ 140
      12.5.2. Distributed Quota ............................ 141
      12.5.3. Correctness ................................ 141
      12.5.4. Residual Attacks .............................. 141
   12.6. Routing Security ................................ 142
      12.6.1. Background ................................ 142
      12.6.2. Admissions Control ............................ 143
      12.6.3. Peer Identification and Authentication ........ 143
      12.6.4. Protecting the Signaling ....................... 144
      12.6.5. Routing Loops and Dos Attacks ................ 144
      12.6.6. Residual Attacks ............................. 144
13. IANA Considerations ................................ 145
   13.1. Well-Known URI Registration ....................... 145
   13.2. Port Registrations ................................ 145
   13.3. Overlay Algorithm Types ............................ 146
   13.4. Access Control Policies ............................ 146
   13.5. Application-ID .................................. 147
   13.6. Data Kind-ID .................................. 147
   13.7. Data Model .................................... 148
   13.8. Message Codes .................................. 148
   13.9. Error Codes ................................... 150
   13.10. Overlay Link Types ............................... 150
   13.11. Overlay Link Protocols ............................ 151
   13.12. Forwarding Options ............................... 151
   13.13. Probe Information Types .......................... 152
   13.15. reload URI Scheme ................................ 152
      13.15.1. URI Registration ............................ 153
   13.16. Media Type Registration ........................... 154
   13.17. XML Name Space Registration ....................... 155
      13.17.1. Config URL ................................ 155
      13.17.2. Config Chord URL ............................ 155
14. Acknowledgments ..................................... 155
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. The idea here is that RELOAD provides a generic structure that should fit most types of overlay topologies (ring, hyperspace, etc.). To instantiate an actual network, you combine RELOAD with a specific overlay algorithm, which defines how to construct the overlay topology and route...
messages efficiently within it. 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.

1.1. Basic Setting

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

```
+--------+              +--------+              +--------+
| Node 10|--------------| Node 20|--------------| Node 30|
+--------+              +--------+              +--------+
|                       |                       |                       |
|                       |                       |                       |
+--------+              +--------+              +--------+
| Node 40|--------------| Node 50|--------------| Node 60|
+--------+              +--------+              +--------+
|                       |                       |                       |
|                       |                       |                       |
+--------+              +--------+              +--------+
| Node 70|--------------| Node 80|--------------| Node 90|
+--------+              +--------+              +--------+
|                       |                       |                       |
|                       |                       |                       |
+--------+              +--------+              +--------+
| Node 85|--------------| Node 90|--------------| Node 95|
+--------+              +--------+              +--------+
```

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, albeit one designed for small-scale storage rather than for bulk storage of large objects. 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. 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 responsible for end-to-end reliability, accomplished by timer-based retransmissions. Unlike the Internet transport layer, however, this layer does not provide congestion control. RELOAD is a request-response protocol, with no more than two pairs of request-response messages used in typical transactions between pairs of nodes, therefore there are no opportunities to observe and react to end-to-end congestion. As with all Internet applications, implementers are strongly discouraged from writing applications that react to loss by immediately retrying the transaction.

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.
Stability of the underlying network supporting the overlay (the Internet) and congestion control between overlay neighbors, which exchange routing updates and data replicas in addition to forwarding end-to-end messages, is handled by the Forwarding and Link Management layer described below.

Real-world experience has shown that a fixed timeout for the end-to-end retransmission timer is sufficient for practical overlay networks. This timer is adjustable via the overlay configuration. As the overlay configuration can be rapidly updated, this value could be dynamically adjusted at coarse time scales, although algorithms for determining how to accomplish this are beyond the scope of this specification. In many cases, however, more appropriate means of improving network performance, such as the Topology Plugin removing lossy links from use in overlay routing or reducing the overall hop-count of end-to-end paths will be more effective than simply increasing the retransmission timer.

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.

Congestion control is implemented at this layer to protect the Internet paths used to form the link in the overlay. Additionally, retransmission is performed to improve the reliability of end-to-end transactions. The relationship between this layer and the Message Transport Layer is similar to the relationship between link-level congestion control and retransmission in modern wireless networks is to Internet transport protocols.

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 is signed.

Object Level: Stored objects is 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].

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.

Joining Peer: A node that is attempting to become a Peer in a particular Overlay.

Admitting Peer: A Peer in the Overlay which helps the Joining Peer join the Overlay.

Bootstrap Node: A network node used by Joining Peers to help locate the Admitting Peer.

Peer Admission: The act of admitting a peer (the "Joining Peer") into an Overlay. After the admission process is over, the joining peer is a fully-functional peer of the overlay. During the admission process, the joining peer may need to present credentials to prove that it has sufficient authority to join the overlay.
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, in strict order. A single Node-ID or a Resource-ID is a trivial form of destination list. When multiple Node-IDs are specified (no more than one Resource-ID is permitted, and it MUST be the last entry) a Destination List is a loose source route.

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.
Transaction ID: A randomly chosen identifier selected by the originator of a request and used to correlate requests and responses.

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

The overlay parameters are specified in a configuration document. Because the parameters include security critical information such as the certificate signing trust anchors, the configuration document must be retrieved securely. The initial configuration document is either initially fetched over HTTPS or manually provisioned; subsequent configuration document updates are received either by periodically refreshing from the configuration server, or, more commonly, by being flood filled through the overlay, which allows for fast propagation once an update is pushed. In the latter case, updates are via digital signatures tracing back to the initial configuration document.

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. Note that the relevant cryptographic property for the digest is preimage resistance. Collision-resistance is not required since an attacker who can create two nodes with the same Node-ID but different public key obtains no advantage. 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. Another drawback of this approach is that user’s data is then tied to their keys, so if a key is changed any data stored under their Node-ID must then be re-stored. This is not an issue for centrally-issued Node-IDs provided that the CA re-issues the same Node-ID when a new certificate is generated.

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 also have storage responsibilities. We use the term "client" to refer to nodes that do not have routing or storage responsibilities. When text applies to both peers and clients, we will simply refer to such 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 entity. 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 connection 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 can use a Ping to probe the Node-ID of the node to which it is connected before doing the Attach.

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.
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 capabilities of RELOAD’s routing layer, the protocol features used to implement them, and 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 provides the following capabilities:

Resource-based routing: RELOAD supports routing messages based solely on the name of the resource. Such messages are delivered to a node that is responsible for that resource. Both structured and unstructured overlays are supported, so the route may not be deterministic for all Topology Plugins.

Node-based routing: RELOAD supports routing messages to a specific node in the overlay.

Clients: RELOAD supports requests from and to clients that do not participate in overlay routing, located via either of the mechanisms described above.

Bridging overlays: Similar to how a Destination List is used to reach a client attached via an arbitrary peer, RELOAD can route messages between two different overlays by building a destination list that includes a peer (or client) with connectivity to both networks.

NAT Traversal: RELOAD supports establishing and using connections between nodes separated by one or more NATs, including locating peers behind NATs for those overlays allowing/requiring it.

Low state: RELOAD’s routing algorithms do not require significant state (i.e., state linear or greater in the number of outstanding messages that have passed through it) to be stored on intermediate peers.

Routability in unstable topologies: Overlay topology changes constantly in an overlay of moderate size due to the failure of individual nodes and links in the system. RELOAD’s routing allows peers to re-route messages when a failure is detected, and replies can be returned to the requesting node as long as the peers that originally forwarded the successful request do not fail before the response is returned.
RELOAD’s routing utilizes three basic mechanisms:

Destination Lists: While in principle it is possible to just inject a message into the overlay with a single 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 in order (i.e., it is loose source routed). The minimal destination list contains just a single value.

Via Lists: In order to allow responses to follow the same path as requests, each message also contains a "Via List", which is appended 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:

A         B         X         Z
-------------------------------
-------->                  
Dest=Z

--------->
Via=A

-------->
Dest=Z

-------->
Via=A,B

-------->
Dest=Z

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

<----------
Dest=B,A

<----------
Dest=A

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

This figure shows use of full via-lists by intermediate peers B and X. However, if B and/or X are willing to store state, then they may elect to truncate the lists, save that information internally (keyed by the transaction id), and return the response message along the path from which it was received when the response is received. This option requires greater state 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.

```
A         B         X         Z
-------------------------------
--------->
Dest=Z
        --------->
Via=A
Dest=Z
             --------->
Via=X1
Dest=Z

<----------
Dest=X,X1
<--------
Dest=B,A
<--------
Dest=A
```

As before, when B receives the message, he creates via list consisting of [A]. However, instead of sending [A,B], X creates an opaque ID X1 which maps internally to [A, B] (perhaps by being an encryption of [A, B] and forwards to Z with only X1 as the via list. When the response arrives at X, it maps X1 back to [A, B] and then inverts it to produce the new destination list [B, A] and routes it to B.

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, which is mandatory to implement, but the base RELOAD protocol is designed to support a variety of overlay algorithms. The information needed
to implement this DHT is fully contained in this specification but it is easier to understand if you are familiar with Chord [Chord] based DHTs. A nice tutorial can be found at [wikiChord].

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, which is specified by configuration for all nodes in the overlay, and thus known to nodes prior to their attempting to join the overlay. 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 past adjacencies which have public IP address and attempt to use them as future bootstrap nodes. Note that this requires some notion of which addresses are likely to be public as discussed in Section 8.

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 [RFC2818] 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. The expected domain name for HTTPS is the name of 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. Note that that out of band method MUST provide authentication and integrity, because the configuration document contains the trust anchors for the system.

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:

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

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 individually 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 node 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 and the message MUST be discarded. 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 (note that this does not include signature checking on intermediate nodes as the message may be fragmented) 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 an opaque ID that is being used for destination list compression. Handling of this case is described in Section 5.1.3. Note that opaque 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 (mutually exclusive) sub-cases to consider.

- 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 verify the signature and pass it up to the upper layers.
- 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 so the node passes it up to the upper layers. Otherwise the node
removes the entry from the destination list and repeats the routing process with the next entry on the destination list. If the message is a response and list compression was used, then the node first modifies the destination list to reinsert the saved state, e.g., by unpacking any opaque ids.

- If the entry is the wildcard Node-ID, the message is destined for this node and it passes it up to the upper layers.

- 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 the later 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 ensure 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. There are two major ways of accomplishing this:

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

As an example of the first strategy, consider an example with nodes A, B, C, D and E. 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 MUST be able to reconstruct the correct destination list on the return route. RELOAD provides explicit support for this functionality in the form of opaque 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 opaque 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 opaque 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. Possibilities for a opaque id include a compressed version of the original via list or an index into a state database containing the original via list, but the details are a local matter.

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 configuration parameter 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. Opaque ID

If the first entry in the destination list is an opaque 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. I.e., a node in an overlay which supports SRR and routing algorithm XXX might use SRR some of the time and XXX some of the time.

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 uses 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 requests for a 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, for instance by having the usage store data at multiple Resource-IDs with the requesting node sending requests to each of those Resource-
IDs. However, a single request MAY also be routed through multiple adjacent peers, even when known to be sub-optimal, to improve reliability [vulnerabilities-acscac04]. Such parallel searches MAY be specified by the topology plugin, in which case it would return multiple next hops and the request would be routed to all of them.

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 timer to the current overlay-reliability-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:

```c
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.

```c
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. This MUST be formed by taking the lower 32 bits of the SHA-1 [RFC3174] hash of the overlay name. 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. The overlay name MUST consist of a sequence of charters what would be allowable as a DNS name.

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 just before transmission. If a received message has a TTL of 0, and the message is not destined for the receiving node, then the message MUST NOT be propagated further and an "Error_TTL_Exceeded" error should be generated. The initial value of the TTL SHOULD be 100 and MUST NOT exceed 100 unless defined otherwise by the overlay configuration. Implementations which receive message with a TTL greater than the current value of initial-ttl (or the 100 default) MUST discard the message and send an "Error_TTL_Exceeded" error.

fragment: This field is used to handle fragmentation. The high bit (0x80000000) MUST be set for historical reasons. If the next bit (0x40000000) is set to 1, it indicates that this is the last (or only) 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_Incompatible_with_Overlay" 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), opaque_id_type(3),
      /* 128-255 not allowed */ (255) } DestinationType;

select (destination_type) {
  case node:
    NodeId               node_id;

  case resource:
    ResourceId           resource_id;

  case opaque_id_type:
    opaque               opaque_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               opaque_id; /* top bit MUST be 1 */
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 opaque_id_type and a opaque_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 "opaque_id_type".

- **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.

- **opaque**: 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:

```c
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 these 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:

eenum { 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
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. Developers are encouraged to put enough diagnostic information to be useful in error_info.

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

Error_Forgotten: 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_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, or some other parameter that is inconsistent with the overlay configuration.

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.

Error_Invalid_Message: Something about this message is invalid but it doesn’t fit the other error codes. When this message is sent, implementations SHOULD provide some meaningful description in error_info to aid in debugging.

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, except for those certificates in a root-cert element of the current configuration file. 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 that have an alternative certificate distribution mechanism, some certificates MAY be omitted. However, unless an alternative mechanism for immediately generating certificates, such as shared secret security (Section 12.4) is used, it is strongly RECOMMENDED that implementors include all referenced certificates, otherwise there is the possibility that messages may not be immediately verifiable because certificates must first be retrieved.

NOTE TO IMPLEMENTERS: This requirement implies that a peer storing data is obligated to retain certificates for the data it holds regardless of whether it is responsible for or actually holding the certificates for the Certificate Store usage.

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.
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.

```c
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 MUST 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 (and fragment reassembly if needed) the destination node MUST verify the signature and the authorizing certificate. If the signature fails, the implementation SHOULD simply drop the message and MUST not process it. 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 by the requesting node:
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.
- 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.

```c
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.

Because joins may only be executed between nodes which are directly adjacent, receiving peers MUST verify that any JoinReq they receive arrives from a transport channel that is bound to the Node-Id to be assumed by the joining peer.) This also prevents replay attacks provided that DTLS anti-replay is used.

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

```c
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.

Joining nodes MUST verify that the signature on the JoinAns message matches the expected target (i.e., the adjacency over which they are
joining.)  If not, they MUST discard the message.

In general, nodes which cannot form connections SHOULD report an error to the user. However, implementations MUST provide some mechanism whereby nodes can determine that they are potentially the first node and take responsibility for the overlay (the idea is to avoid having ordinary nodes try to become responsible for the entire overlay during a partition.) 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.

```c
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.

Because leaves may only be executed between nodes which are directly adjacent, receiving peers MUST verify that any LeaveReq they receive arrives from a transport channel that is bound to the Node-ID to be assumed by the leaving peer.) This also prevents replay attacks provided that DTLS anti-replay is used.

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.

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

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, opaque 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.

```plaintext
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;

denum { 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 by 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:

- 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.
- If it has already sent an Attach request to and received the
  response from the originating peer of this Attach request, and 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.
- 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-endian
    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.
- 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 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
For each group of 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.

Head of line blocking is undesirable in an Overlay Link protocol because the messages carried on a RELOAD link are independent, rather than stream-oriented. Therefore, if message N on a link is lost, delaying message N+1 on that same link until N is successfully retransmitted does nothing other than increase the latency for the transaction of message N+1 as they are unrelated to each other. Therefore, while the high quality, performance, and availability of modern TCP implementations makes them very attractive, their performance as an Overlay Link protocol is not optimal.

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 message, 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
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.

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>;
}
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

```c
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.

```c
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 the Kind information (Section 6.4.1.1).

5.5.4.1. Request Definition

```c
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

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:
o  The sequence number in the document is greater than the current
    configuration sequence number.
o  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 (i.e.,
one listed in the current 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.

If an implementation receives repeated ConfigUpdates which it cannot
verify with sequence numbers substantially in advance of its own
configuration document, it SHOULD contact the configuration server to
get the latest configuration file in order to avoid permanent
breakage. The details of this are left up to the implementation.

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
A 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) (OverlayLinkType=DTLS-UDP-SR)
- TLS [RFC5246] over TCP with Framing Header, No-ICE (OverlayLinkType=TLS-TCP-FH-NO-ICE)
- DTLS [RFC4347] over UDP with SR, No-ICE (OverlayLinkType=DTLS-UDP-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 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.
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 its 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.

Note that because retransmissions receive new sequence numbers, multiple ACKs may be received for the same message. This approach provides more information than traditional TCP sequence numbers, but care must be taken when applying algorithms designed based on TCP’s stream-oriented sequence number.

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.

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. The sender algorithm used MUST meet the requirements of [RFC5405].
5.6.3.1. Stop and Wait Sender Algorithm

This section describes one possible implementation of a sender algorithm for Simple Reliability. It is adequate for overlays running on underlying networks with low latency and loss (LANs) or low-traffic overlays on the Internet.

A node MUST NOT have more than one unacknowledged message on the DTLS connection at a time. Note that because retransmissions of the same message are given new sequence numbers, there may be multiple unacknowledged sequence numbers in use.

The RTO ("Retransmission TimeOut") is based on an estimate of the round-trip time (RTT). The value for RTO is calculated separately for each DTLS session. 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.)

An initiating node SHOULD retransmit a message if it has not received an ACK after an interval of RTO (transit nodes do not retransmit at this layer). The node MUST double the time to wait after each retransmission. For each retransmission, the sequence number MUST be incremented.

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.

A sender MUST wait 10ms between receipt of an ACK and transmission of the next message.

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. Any compliant sender algorithm may be 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 list, destination list, and ForwardingOptions and MUST contain at least 256 bytes of the message body. If these elements cannot fit within the MTU of the underlying datagram protocol, 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 refrragmented 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 discovery 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.

The fragment field of the forwarding header is used to encode fragmentation information. 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 last fragment indicator MUST be appropriately set. If the message is not fragmented, it is simply treated as if it is the only fragment: the last fragment bit is set and the offset is 0 resulting in a fragment value of 0xC0000000.

Note: the reason for this definition of the fragment field is that originally the high bit was defined in part of the specification as "is fragmented" and so there was some specification ambiguity about how to encode messages with only one fragment. This ambiguity was resolved in favor of always encoding as the "last" fragment with offset 0, thus simplifying the receiver code path, but resulting in the high bit being redundant. Because messages MUST be set with the high bit set to 1, implementations SHOULD discard any message with it set to 0. Implementations (presumably legacy ones) which choose to accept such messages MUST either ignore the remaining bits or ensure that they are 0. They MUST NOT try to interpret as fragmented messages with the high bit set low.

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:

```c
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:

\[
\text{resource_id} \ || \ \text{kind} \ || \ \text{storage_time} \ || \ \text{StoredDataValue} \ || \ \text{SignerIdentity}
\]

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.

Note that there is no necessarily relationship between the validity window of a certificate and the expiry of the data it is authenticating. When signatures are verified, the current time MUST be compared to the certificate validity period. However, it is permitted to have a value signed which expires after a certificate’s validity period (though this will likely cause verification failure at some future time.)

### 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:

```c
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:

```c
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:

```c
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
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.
o 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.

o 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.

o For original (non-replica) stores, the peer MUST check that if the generation counter is non-zero, it equals the current value of the generation counter for this Kind. This feature allows the generation counter to be used in a way similar to the HTTP Etag feature.

o 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.

o The storage time values are greater than that of any value which would be replaced by this Store.

o The size and number of the stored values is consistent with the limits specified in the overlay configuration.

o 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
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

RELOAD 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

```c
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.

modelSpecifier
   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 modelSpecifier 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.

```c
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.

Upon receipt of a FetchAns message, nodes MUST verify the signatures on all the received values. Any values with invalid signatures (including expired certificates) MUST be discarded. Note that this
implies that implementations which wish to store data for long periods of time must have certificates with appropriate expiry dates or re-store periodically. Implementations MAY return the subset of values with valid signatures, but in that case SHOULD somehow signal to the application that a partial response was received.

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.

```c
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 \( R_{n+1} = \text{nearest}(1 + R_n) \).

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.

```c
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 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 SHOULD NOT be a candidate for a TURN server. Additionally, if the peer's IP address is in the private address space range as defined by [RFC1918], then it is also SHOULD NOT be 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 MAY 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

---

NOTE TO IMPLEMENTERS: As the access control for this usage is not CERTIFICATE_BY_NODE or CERTIFICATE_BY_USER, the certificates used by TurnServer entries need to be retained as described in Section 5.3.4.

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 MUST be 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 MUST route the message to that node. Otherwise, it MUST route 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 MUST route 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 MUST store the data and returns a success response. It MUST 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 SHOULD 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 MUST 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 detects 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:

```c
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:

```c
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:

```c
type = from_succ
    The Leave request was sent by a successor.

type = from_pred
    The Leave request was sent by a predecessor.
```

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

```c
successors
    The sender’s successor list.
```

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

```c
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
<?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"
  <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>
      MIIDJDCCAo2gAwIBAgIBADANBgkqhkiG9w0BAQUFADBwMQswCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWVzc3p5YTEMA8GA1UEBxMII2U3IhAkBgNVBAoT
BXNpcg10MSkwJwYDVQQLEw1JZGx0MREwDQYDVQQHEwVhdXRvcmVzMSsGCCsGAQUFBzAChHRITUwEBzCBMAoGCC5q0MA8GA1UdDgQWBBRrRhcU6pR2JYBUbhN
bhNU2gjyjVBShjCeBmgyDVKoR0jB1GIGfGmGRhG6pR2JYBUbhNU2qHZV8tC9EF
pHTwCPELMaAGA1UEBxMMeCVMX2EARBgNWBqTcKhhbGlbmb3JuaWE9TAPBgNV
CFNhNbIBKF31N04wdaDYQFQKEwVzaxBnpSEOgCggA1UEC4HgU21waXQgVGzdCBD
ZXJP0aW9ypZ2F0SBdURb3JpdHMvcQAaWDAYDVR0TAUwAwEB/zANBgkqhkiG9w0
AQUFwAABoQCFvvl1ZGTRXbXH8/EqkdsczSoQf/rQ0rQac9wNY/n2Zbkr3O
qAzaG5Sfkmk1w7f+doG5qRqXg/+y61031Rep7KeVdpapl1MFnpfKsiETM1pzwayNQ
QgUf4cKB1f+65Qe7uZuOrJa2EMv8qW4twEhGyC1pFU9yoz510hVg==
    </root-cert>
  </configuration>
</overlay>
```

<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-reliability-timer> 3000 </overlay-reliability-timer>
<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-block>
    <kind-signature>
        VGhpcyBpcyBub3QgcmlnaHQhCg==
    </kind-signature>
</required-kinds>

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 provide 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. All times MUST be in UTC.
- **sequence:** a monotonically increasing sequence number between 0 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. Note that there is no necessary relationship
between the overlay name/configuration server name and the
enrollment server name.

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. The security guarantee here is that any agent which is able to access the configuration document (presumably protected by some sort of HTTP access control or network topology) is able to recover the shared secret and hence join the overlay.
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-reliability-timer   Default value for the end-to-end retransmission timer for messages, in milliseconds. If not present, the default value is 3000.
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.
kindsigner   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 needs to be treated as a binary blob that cannot be changed - including any 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. Any configuration file through the overlay (as opposed to directly from the configuration server) MUST be signed by one of the configure-signers from the previous extant configuration. Recipients MUST verify the signature prior to accepting 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 = ""
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 }?

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 &= element 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 determining 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 MUST form 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 MUST replace any previously learned configuration file for this overlay.

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

10.3. Credentials

If the configuration document contains an 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]. The 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 parameter encodes as described in [RFC2388] and the following properties:

- If authentication is required, there is an form 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 form parameter of "nodeids" containing the number of Node-IDs required.
- There MUST be a form parameter of "csr" with a content type of "application/pkcs10", as defined in [RFC2311].
- The Accept header MUST contain 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. The reason for using the RFC 2388 "multipart/form-data" encoding is so that the password parameter will not be encoded in the URL to reduce the chance of accidental leakage of the password. If the authentication succeeds and the requested user name is acceptable, the server generates and returns a certificate for the certificate signing request in the "csr" parameter of the request. 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". The enrollment server SHOULD maintain a mapping of users to node-ids and if the same user returns (e.g., to have their certificate re-issued) return the same Node-ID, thus avoiding the need for implementations to re-store all their data when their certificates expire.
- A single name this user is allowed to use in the overlay, using type rfc822Name. Enrollment servers SHOULD take care to only allow legal characters in the name (e.g., no embedded NULs), rather than simply accepting any name provided by the user.

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. In particular, password errors SHOULD be returned as 401 Unauthorized. [[ OPEN ISSUE: We know this isn’t right and have a question out to the apps AD. ]]
The client MUST check that the certificate returned chains back to one of the certificates received in the "root-cert" list of the overlay configuration data (including PKIX BasicConstraints checks.) 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 MUST first form 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 abbreviations 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.
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.
Finally, now that JP has a copy of all the data and is ready to route messages and receive requests, it sends Updates to everyone in its Routing Table to tell them it is ready to go. Below, it is shown sending such an update to TP.

```
JP  NP  XX  TP
  |   |   |   |
  |   |   |   |
  |   |   |   |
  |   |   |   |
Update
  --------------
  <-----
UpdateAns
  <-----------
```

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. Routing Loops and Dos Attacks

Source routing mechanisms are known to create the possibility for DoS amplification, especially by the induction of routing loops [RFC5095]. In order to limit amplification, the initial-ttl value in the configuration file SHOULD be set to a value slightly larger than the longest expected path through the network. For Chord, experience has shown that log(2) of the number of nodes in the network + 5 is a safe bound. Because nodes are required to enforce the initial-ttl as the maximum value, an attacker cannot achieve an amplification factor greater than initial-ttl, thus limiting the additional capabilities provided by source routing.

In order to prevent the use of loops for targeted implementation attacks, implementations SHOULD check the destination list for duplicate entries and discard such records with an "Error_Invalid_Message" error. This does not completely prevent loops but does require that at least one attacker node be part of the loop.

12.6.6. 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 SHALL 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 SHALL 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        |
| Reference                                        | Enrollment [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:

<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHORD-RELOAD</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>EXP-OVERLAY</td>
<td>RFC-AAAA</td>
</tr>
</tbody>
</table>

The value EXP-OVERLAY has been made available for the purposes of experimentation. This value is not meant for vendor specific use of any sort and it MUST NOT be used for operational deployments.

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:

<table>
<thead>
<tr>
<th>Access Policy</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER-MATCH</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>NODE-MATCH</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>USER-NODE-MATCH</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>NODE-MULTIPLE</td>
<td>RFC-AAAA</td>
</tr>
</tbody>
</table>
The value EXP-MATCH has been made available for the purposes of experimentation. This value is not meant for vendor specific use of any sort and it MUST NOT be used for operational deployments.

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 0xfff0 SHALL be registered via RFC 5226 Expert Review. Code points in the range 0xfff1 to 0xfffffe are reserved for private use. The initial contents of this registry are:

<table>
<thead>
<tr>
<th>Application</th>
<th>Application-ID</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVALID</td>
<td>0</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>SIP</td>
<td>5060</td>
<td>Reserved for use by SIP Usage</td>
</tr>
<tr>
<td>SIP</td>
<td>5061</td>
<td>Reserved for use by SIP Usage</td>
</tr>
<tr>
<td>Reserved</td>
<td>0xffff</td>
<td>RFC-AAAA</td>
</tr>
</tbody>
</table>

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 0x80000000 to 0xfff000000 SHALL be registered via RFC 5226 Expert Review. Code points in the range 0xff000001 to 0xffffffff are reserved for private use via the Kind description mechanism described in Section 10. The initial contents of this registry are:

<table>
<thead>
<tr>
<th>Kind</th>
<th>Kind-ID</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVALID</td>
<td>0</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>TURN-SERVICE</td>
<td>2</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>CERTIFICATE_BY_NODE</td>
<td>3</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>CERTIFICATE_BY_USER</td>
<td>16</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Reserved</td>
<td>0xffffffff</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Reserved</td>
<td>0xffffffff</td>
<td>RFC-AAAA</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Data Model</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVALID</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>SINGLE</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>ARRAY</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>DICTIONARY</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>EXP-DATA</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>RESERVED</td>
<td>RFC-AAAA</td>
</tr>
</tbody>
</table>

The value EXP-DATA has been made available for the purposes of experimentation. This value is not meant for vendor specific use of any sort and it MUST NOT be used for operational deployments.

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:
<table>
<thead>
<tr>
<th>Message Code Name</th>
<th>Code Value</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>invalid</td>
<td>0</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>probe_req</td>
<td>1</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>probe_ans</td>
<td>2</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>attach_req</td>
<td>3</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>attach_ans</td>
<td>4</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>unused</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>unused</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>store_req</td>
<td>7</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>store_ans</td>
<td>8</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>fetch_req</td>
<td>9</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>fetch_ans</td>
<td>10</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>unused (was remove_req)</td>
<td>11</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>unused (was remove_ans)</td>
<td>12</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>find_req</td>
<td>13</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>find_ans</td>
<td>14</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>join_req</td>
<td>15</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>join_ans</td>
<td>16</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>leave_req</td>
<td>17</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>leave_ans</td>
<td>18</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>update_req</td>
<td>19</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>update_ans</td>
<td>20</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>route_query_req</td>
<td>21</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>route_query_ans</td>
<td>22</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>ping_req</td>
<td>23</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>ping_ans</td>
<td>24</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>stat_req</td>
<td>25</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>stat_ans</td>
<td>26</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>unused (was attachlite_req)</td>
<td>27</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>unused (was attachlite_ans)</td>
<td>28</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>app_attach_req</td>
<td>29</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>app_attach_ans</td>
<td>30</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>unused (was app_attachlite_req)</td>
<td>31</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>unused (was app_attachlite_ans)</td>
<td>32</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>config_update_req</td>
<td>33</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>config_update_ans</td>
<td>34</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>exp_a_req</td>
<td>35</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>exp_a_ans</td>
<td>36</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>exp_b_req</td>
<td>37</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>exp_b_ans</td>
<td>38</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>reserved</td>
<td>0x8000..0xffff</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>error</td>
<td>0xffff</td>
<td>RFC-AAAA</td>
</tr>
</tbody>
</table>

The values exp_a_req, exp_a_ans, exp_b_req, and exp_b_ans have been made available for the purposes of experimentation. These values are
not meant for vendor specific use of any sort and MUST NOT be used for operational deployments.

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:

<table>
<thead>
<tr>
<th>Error Code Name</th>
<th>Code Value</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>invalid</td>
<td>0</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Unused</td>
<td>1</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Error_Forbidden</td>
<td>2</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Error_Not_Found</td>
<td>3</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Error_Request_Timeout</td>
<td>4</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Error_Generation_Counter_Too_Low</td>
<td>5</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Error_Incompatible_with_Overlay</td>
<td>6</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Error_Unsupported_Forwarding_Option</td>
<td>7</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Error_Data_Too_Large</td>
<td>8</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Error_Data_Too_Old</td>
<td>9</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Error_TTL_Exceeded</td>
<td>10</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Error_Message_Too_Large</td>
<td>11</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Error_Unknown_Kind</td>
<td>12</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Error_Unknown_Extension</td>
<td>13</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Error_Response_Too_Large</td>
<td>14</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Error_Config_Too_Old</td>
<td>15</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Error_Config_Too_New</td>
<td>16</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Error_In_Progress</td>
<td>17</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Error_Exp_A</td>
<td>18</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Error_Exp_B</td>
<td>19</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>Error_Invalid_Message</td>
<td>20</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>reserved</td>
<td>0x8000..0xfffe</td>
<td>RFC-AAAA</td>
</tr>
</tbody>
</table>

The values Error_Exp_A and Error_Exp_B have been made available for the purposes of experimentation. These values are not meant for vendor specific use of any sort and MUST NOT be used for operational deployments.

13.10. Overlay Link Types

IANA SHALL create a "RELOAD Overlay Link Registry". For more information on the link types defined here, see Section 5.6. New entries SHALL be defined via RFC 5226 Standards Action. This registry SHALL be initially populated with the following values:
The value EXP-LINK has been made available for the purposes of experimentation. This value is not meant for vendor specific use of any sort and it MUST NOT be used for operational deployments.

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 values:

+---------------+---------------+
| Link Protocol | Specification |
+---------------+---------------+
| TLS           | RFC-AAAA      |
| EXP-PROTOCOL  | RFC-AAAA      |
+---------------+---------------+

The value EXP-PROTOCOL has been made available for the purposes of experimentation. This value is not meant for vendor specific use of any sort and it MUST NOT be used for operational deployments.

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      |
| exp-forward       | 1    | RFC-AAAA      |
| reserved          | 255  | RFC-AAAA      |
+-------------------+------+---------------+
The value exp-forward has been made available for the purposes of experimentation. This value is not meant for vendor specific use of any sort and it MUST NOT be used for operational deployments.

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:

<table>
<thead>
<tr>
<th>Probe Option</th>
<th>Code</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>invalid</td>
<td>0</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>responsible_set</td>
<td>1</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>num_resources</td>
<td>2</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>uptime</td>
<td>3</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>exp-probe</td>
<td>4</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>reserved</td>
<td>255</td>
<td>RFC-AAAA</td>
</tr>
</tbody>
</table>

The value exp-probe has been made available for the purposes of experimentation. This value is not meant for vendor specific use of any sort and it MUST NOT be used for operational deployments.


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:

<table>
<thead>
<tr>
<th>Extensions Name</th>
<th>Code</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>invalid</td>
<td>0</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>exp-ext</td>
<td>1</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>reserved</td>
<td>0xFFFF</td>
<td>RFC-AAAA</td>
</tr>
</tbody>
</table>

The value exp-ext has been made available for the purposes of experimentation. This value is not meant for vendor specific use of any sort and it MUST NOT be used for operational deployments.

13.15. reload URI Scheme

This section describes the scheme for a reload URI, which can be used to refer to either:
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 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 MUST be kept confidential or else anyone that can see the shared-secret and effect the RELOAD overlay network.

Interoperability considerations: No known 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

13.17. XML Name Space Registration

This document registers two URIs for the config and config-chord XML namespaces in the IETF XML registry defined in [RFC3688].

13.17.1. Config URL


Registrant Contact: The IESG.

XML: N/A, the requested URIs are XML namespaces

13.17.2. Config Chord URL


Registrant Contact: The IESG.

XML: N/A, the requested URIs are XML namespaces

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 last call comments were provided by: Jouni Maenpaa, Roni Even, Gonzalo Camarillo, Ari Keranen, John Buford, Michael Chen, Frederic-Philippe Met, Mary Barnes, and David Bryan. Special thanks to Marc Petit-Huguenin who provided an amazing amount of detailed review.

15. References

15.1. Normative References


[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO


15.2. Informative References


REsource LOccation 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-06 (work in progress), July 2011.

[I-D.jiang-p2psip-relay]
to RELOAD to support Direct Response and Relay Peer
routing", draft-jiang-p2psip-relay-05 (work in progress),
March 2011.

[RFC1122] Braden, R., "Requirements for Internet Hosts -

[RFC2311] Dusse, S., Hoffman, P., Ramsdell, B., Lundblade, L., and

[RFC3688] Mealling, M., "The IETF XML Registry", BCP 81, RFC 3688,

[RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness

[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,

"Using the Secure Remote Password (SRP) Protocol for TLS

of Type 0 Routing Headers in IPv6", RFC 5095,
December 2007.


[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S.,


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. 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.

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
Abstract

This document proposes an optional extension to RELOAD to support direct response routing mode. RELOAD recommends symmetric recursive routing for routing messages. The new optional extension provides a shorter route for responses reducing the overhead on intermediary peers and describes the potential cases where this extension can be used.

Status of this Memo

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

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

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

This Internet-Draft will expire on June 2, 2012.

Copyright Notice

Copyright (c) 2011 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.
Table of Contents

1. Introduction .............................................. 4
   1.1. Backgrounds ........................................ 4
2. Terminology ............................................... 4
3. Problem Statement ......................................... 5
   3.1. Overview ............................................ 5
      3.1.1. Symmetric Recursive Routing (SRR) ................ 5
      3.1.2. Direct Response Routing (DRR) ................... 6
   3.2. Scenarios Where DRR can be Used ..................... 7
      3.2.1. Managed or Closed P2P System .................... 7
      3.2.2. Wireless Scenarios .............................. 7
4. Relationship Between SRR and DRR ......................... 8
   4.1. How DRR Works ...................................... 8
   4.2. How SRR and DRR Work Together ..................... 8
5. Comparison on cost of SRR and DRR ......................... 8
   5.1. Closed or managed networks .......................... 9
   5.2. Open networks ..................................... 10
6. Extensions to RELOAD ...................................... 10
   6.1. Basic Requirements ................................. 10
   6.2. Modification To RELOAD Message Structure ........... 11
      6.2.1. State-keeping Flag ............................... 11
      6.2.2. Extensive Routing Mode ......................... 11
   6.3. Creating a Request ................................ 12
      6.3.1. Creating a Request for DRR ..................... 12
   6.4. Request And Response Processing .................... 12
      6.4.1. Destination Peer: Receiving a Request And Sending a Response ...................... 13
      6.4.2. Sending Peer: Receiving a Response ............. 13
7. Optional Methods to Investigate Node Connectivity ....... 13
   7.1. Getting Addresses To Be Used As Candidates for DRR .... 14
   7.2. Public Reacheability Test ........................... 15
8. Security Considerations .................................. 16
9. IANA Considerations ...................................... 16
   9.1. A new RELOAD Forwarding Option ..................... 16
10. Acknowledgements ........................................ 16
11. References .............................................. 16
    11.1. Normative References .............................. 16
    11.2. Informative References ............................ 17
Authors' Addresses ........................................... 17
1. Introduction

1.1. Backgrounds

RELOAD [I-D.ietf-p2psip-base] recommends symmetric recursive routing (SRR) for routing messages and describes the extensions that would be required to support additional routing algorithms. Other than SRR, two other routing options: direct response routing (DRR) and relay peer routing (RPR) are also discussed in Appendix D in [I-D.ietf-p2psip-base]. As we show in section 3, DRR is advantageous over SRR in some scenarios by reducing load (CPU and link BW) on intermediary peers. For example, in a closed network where every node is in the same address realm, DRR performs better than SRR. In other scenarios, using a combination of DRR and SRR together is more likely to bring benefits than if SRR is used alone. Some discussion on connectivity is in Non-Transitive Connectivity and DHTs [http://srhea.net/papers/ntr-worlds05.pdf].

Note that in this draft, we focus on DRR routing mode and its extensions to RELOAD to produce a standalone solution. Please refer to RPR draft [I-D.ietf-p2psip-rpr] for RPR routing mode.

We first discuss the problem statement in Section 3, then how to combine DRR and SRR is presented in Section 4. In Section 5, we give comparison on the cost of SRR and DRR in both managed and open networks. An extension to RELOAD to support DRR is proposed in Section 6. Some optional methods to check node connectivity is introduced in Section 7, as informational text.

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. We also use terms defined in NAT behavior discovery [I-D.ietf-behave-nat-behavior-discovery]. Other terms used in this document are defined inline when used and are also defined below for reference.

There are two types of roles in the RELOAD architecture: peer and client. Node is used when describing both peer and client. In discussions specific to behavior of a peer or client, the term peer or client is used instead.
Publicly Reachable: A node is publicly reachable if it can receive unsolicited messages from any other node in the same overlay. Note: "publicly" does not mean that the nodes must be on the public Internet, because the RELOAD protocol may be used in a closed system.

Direct Response Routing (DRR): refers to a routing mode in which responses to P2PSIP requests are returned to the sending peer directly from the destination peer based on the sending peer’s own local transport address(es). For simplicity, the abbreviation DRR is used instead in the following text.

Symmetric Recursive Routing (SRR): refers to a routing mode in which responses follow the request path in the reverse order to get back to the sending peer. For simplicity, the abbreviation SRR is used instead in the following text.

3. Problem Statement

RELOAD is expected to work under a great number of application scenarios. The situations where RELOAD is to be deployed differ greatly. For instance, some deployments are global, such as a Skype-like system intended to provide public service. Some run in closed networks of small scale. SRR works in any situation, but DRR may work better in some specific scenarios.

3.1. Overview

RELOAD is a simple request-response protocol. After sending a request, a node waits for a response from a destination node. There are several ways for the destination node to send a response back to the source node. In this section, we will provide detailed information on two routing modes: SRR and DRR.

Some assumptions are made in the following illustrations.

1) Peer A sends a request destined to a peer who is the responsible peer for Resource-ID k;
2) Peer X is the root peer being responsible for resource k;
3) The intermediate peers for the path from A to X are peer B, C, D.

3.1.1. Symmetric Recursive Routing (SRR)

For SRR, when the request sent by peer A is received by an intermediate peer B, C or D, each intermediate peer will insert information on the peer from whom they got the request in the via-
list as described in RELOAD. As a result, the destination peer X will know the exact path which the request has traversed. Peer X will then send back the response in the reverse path by constructing a destination list based on the via-list in the request.

A            B            C             D           X
|  Request   |            |            |            |
|----------->|            |            |            |
|            | Request    |            |            |
|----------->|            |            |            |
|            |            | Request    |            |
|----------->|            |            |            |
|            |            |            | Request    |
|----------->|            |            |<-----------|
|            |            |            |            |
|            |  Response  |            |            |<-----------|
|            |<-----------|            |            |
|            |            |            |            |
|            |            |            |  Response  |
|            |            |<-----------|            |
|            |            |            |            |
|            |            |            |            |
|            |            |            |            |
|            |            |            |            |

SRR works in any situation, especially when there are NATs or firewalls. A downside of this solution is that the message takes several hops to return to the client, increasing the bandwidth usage and CPU/battery load of multiple nodes.

3.1.2. Direct Response Routing (DRR)

In DRR, peer X receives the request sent by peer A through intermediate peer B, C and D, as in SRR. However, peer X sends the response back directly to peer A based on peer A’s local transport address. In this case, the response won’t be routed through intermediate peers. Shorter route means less overhead on intermediary peers, especially in the case of wireless network where the CPU and uplink BW is limited. In the absence of NATs or other connectivity issues, this is the optimal routing technique. Note that establishing a secure connection requires multiple round trips. Please refer to Section 5 for cost comparison between SRR and DRR.
3.2. Scenarios Where DRR can be Used

This section lists several scenarios where using DRR would work, and when the increased efficiency would be advantageous.

3.2.1. Managed or Closed P2P System

The properties that make P2P technology attractive, such as the lack of need for centralized servers, self-organization, etc. are attractive for managed systems as well as unmanaged systems. Many of these systems are deployed on private network where nodes are in the same address realm and/or can directly route to each other. In such a scenario, the network administrator can indicate preference for DRR in the peer’s configuration file. Peers in such a system would always try DRR first, but peers must also support SRR in case DRR fails. If during the process of establishing a direct connection with the sending peer, the responding peer receives a response with SRR as the preferred routing mode (or it fails to establish the direct connection), the responding peer should not use DRR but switch to SRR. A node can keep a list of unreachable nodes based on trying DRR and use only SRR for these nodes. The advantage in using DRR is on the network stability since it puts less overhead on the intermediary peers that will not route the responses. The intermediary peers will need to route less messages and save CPU resources as well as the link bandwidth usage.

3.2.2. Wireless Scenarios

While some mobile deployments may use clients, in mobile networks with full peers, there is an advantage to using DRR in order to reduce the load on intermediary nodes. Using DRR helps with reducing radio battery usage and bandwidth by the intermediary peers. The service provider may recommend in the configuration using DRR based on his knowledge of the topology.
4. Relationship Between SRR and DRR

4.1. How DRR Works

DRR is very simple. The only requirement is for the source peers to provide their (publically reachable) transport address to the destination peers, so that the destination peer knows where to send the response. Responses are sent directly to the requesting peer.

4.2. How SRR and DRR Work Together

DRR is not intended to replace SRR. As seen from Section 3, DRR has better performance in some scenarios, but have limitations as well, see for example section 4.3 in Non-Transitive Connectivity and DHTs [http://srhea.net/papers/ntr-worlds05.pdf]. As a result, it is better to use these two modes together to adapt to each peer’s specific situation. In this section, we give some informative suggestions on how to transition between the routing modes in RELOAD.

A peer can collect statistical data on the success of the different routing modes based on previous transactions and keep a list of non-reachable addresses. Based on the data, the peer will have a clearer view about the success rate of different routing modes. Other than the success rate, the peer can also get data of fine granularity, for example, the number of retransmission the peer needs to achieve a desirable success rate.

A typical strategy for the node is as follows. A node chooses to start with DRR. Based on the success rate as seen from the lost message statistics or responses that used DRR, the node can either continue to offer DRR first or switch to SRR.

The node can decide whether to try DRR based on other information such as configuration file information. If an overlay runs within a private network and all nodes in the system can reach each other directly, nodes may send most of the transactions with DRR.

5. Comparison on cost of SRR and DRR

The major advantages in using DRR are in going through less intermediary peers on the response. By doing that it reduces the load on those peers’ resources like processing and communication bandwidth.
5.1. Closed or managed networks

As described in Section 3, many P2P systems run in a closed or managed environment (e.g. carrier networks) so that network administrators would know that they could safely use DRR.

SRR brings out more routing hops than DRR. Assuming that there are N nodes in the P2P system and Chord is applied for routing, the number of hops for a response in SRR and DRR are listed in the following table. Establishing a secure connection between sending peer and responding peer with (D)TLS requires multiple messages. Note that establishing (D)TLS secure connections for P2P overlay is not optimal in some cases, e.g. direct response routing where (D)TLS is heavy for temporary connections. Instead, some alternate security techniques, e.g. using public keys of the destination to encrypt the messages, signing timestamps to prevent reply attacks can be adopted.

Therefore, in the following table, we show the cases of: 1) no (D)TLS in DRR; 2) still using DTLS in DRR as sub-optimal and, as the worst-cost case, 7 messages are used during the DTLS handshaking [DTLS]. (TLS Handshake is two round-trip negotiation protocol while DTLS handshake is three round-trip negotiation protocol.)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Success</th>
<th>No. of Hops</th>
<th>No. of Msgs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR</td>
<td>Yes</td>
<td>logN</td>
<td>logN</td>
</tr>
<tr>
<td>DRR</td>
<td>Yes</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DRR(DTLS)</td>
<td>Yes</td>
<td>1</td>
<td>7+1</td>
</tr>
</tbody>
</table>

From the above comparison, it is clear that:

1) In most cases of N > 2 (2^1), DRR has fewer hops than SRR. Shorter route means less overhead and resource usage on intermediary peers, which is an important consideration for adopting DRR in the cases where the resource such as CPU and BW is limited, e.g. the case of mobile, wireless network.

2) In the cases of N > 256 (2^8), DRR also has fewer messages than SRR.

3) In the cases where N < 256, DRR has more messages than SRR (but still has fewer hops than SRR). So the consideration to use DRR or SRR depends on other factors like using less resources (bandwidth and processing) from the intermediaries peers. Section 4 provides use cases where DRR has better chance to work or where the intermediary resources considerations are important.
5.2. Open networks

In open network where DRR is not guaranteed, DRR can fall back to SRR if it fails after trial, as described in Section 4. Based on the same settings in Section 5.1, the number of hops, number of messages for a response in SRR and DRR are listed in the following table.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Success</th>
<th>No. of Hops</th>
<th>No. of Msgs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR</td>
<td>Yes</td>
<td>logN</td>
<td>logN</td>
</tr>
<tr>
<td>DRR</td>
<td>Yes</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Fail&amp;Fall back to SRR</td>
<td>1+logN</td>
<td>1+logN</td>
</tr>
<tr>
<td>DRR(DTLS)</td>
<td>Yes</td>
<td>1</td>
<td>7+1</td>
</tr>
<tr>
<td></td>
<td>Fail&amp;Fall back to SRR</td>
<td>1+logN</td>
<td>8+logN</td>
</tr>
</tbody>
</table>

From the above comparison, it can be observed that:

1) Trying DRR would still have a good chance of fewer hops than SRR. Suppose that P peers are publicly reachable, the number of hops in DRR and SRR is $P(1+(N-P)\log N)$ and $N\log N$, respectively. The condition for fewer hops in DRR is $P(1+(N-P)\log N) < N\log N$, which is $P/N > 1/\log N$. This means that when the number of peers N grows, the required ratio of publicly reachable peers P/N for fewer hops in DRR decreases. Therefore, the chance of trying DRR with fewer hops than SRR becomes better as the scale of the network increases.

2) In the cases of large network and the success rate of DRR is good, it is still possible that DRR has fewer messages than SRR. Otherwise, the consideration to use DRR or SRR depends on other factors like using less resources from the intermediaries peers.

6. Extensions to RELOAD

Adding support for DRR requires extensions to the current RELOAD protocol. In this section, we define the changes required to the protocol, including changes to message structure and to message processing.

6.1. Basic Requirements

All peers implementing DRR MUST support SRR.

All peers MUST be able to process requests for routing in SRR, and MAY support DRR routing requests.
6.2. Modification To RELOAD Message Structure

RELOAD provides an extensible framework to accommodate future extensions. In this section, we define a ForwardingOption structure to support DRR mode. Additionally we present a state-keeping flag to inform intermediate peers if they are allowed to not maintain state for a transaction.

6.2.1. State-keeping Flag

RELOAD allows intermediate peers to maintain state in order to implement SRR, for example for implementing hop-by-hop retransmission. If DRR is used, the response will not follow the reverse path, and the state in the intermediate peers won’t be cleared until such state expires. In order to address this issue, we propose a new flag, state-keeping flag, in the message header to indicate whether the state keeping is not required in the intermediate peers.

flag : 0x8 IGNORE-STATE-KEEPING

If IGNORE-STATE-KEEPING is set, any peer receiving this message and which is not the destination of the message SHOULD forward the message with the full VIA list and SHOULD not maintain any internal state.

6.2.2. Extensive Routing Mode

This draft introduces a new forwarding option for an extensive routing mode. This option conforms to the description in section 5.3.2.3 in [I-D.ietf-p2psip-base].

We first define a new type to define the new option, extensive_routing_mode:

The option value will be illustrated in the following figure, defining the ExtensiveRoutingModeOption structure:

```
enum {(0), DRR(1), RPR(2), (255)} RouteMode;
struct {
    RouteMode routemode;
    OverlayLinkType transport;
    IpAddressPort ipaddressport;
    Destination destination<1..2^8-1>;
} ExtensiveRoutingModeOption;
```

The above structure reuses: OverlayLinkType, Destination and IpAddressPort structure defined in section 5.5.1.1, 5.3.2.2 and
5.3.1.1 in [I-D.ietf-p2psip-base].

RouteMode: refers to which type of routing mode is indicated to the destination peer. Currently, only DRR and RPR (specified in RPR draft [I-D.ietf-p2psip-rpr]) are defined.

OverlayLinkType: refers to the transport type which is used to deliver responses from the destination peer to the sending peer.

IpAddressPort: refers to the transport address that the destination peer should use to send the response to. This will be a sending node address for DRR.

Destination: refers to the sending node itself. If the routing mode is DRR, then the destination only contains the sending node’s node-id.

6.3.  Creating a Request

6.3.1.  Creating a Request for DRR

When using DRR for a transaction, the sending peer MUST set the IGNORE-STATE-KEEPING flag in the ForwardingHeader. Additionally, the peer MUST construct and include a ForwardingOptions structure in the ForwardingHeader. When constructing the ForwardingOption structure, the fields MUST be set as follows:

1) The type MUST be set to extensive_routing_mode.

2) The ExtensiveRoutingModeOption structure MUST be used for the option field within the ForwardingOptions structure. The fields MUST be defined as follows:

2.1) routemode set to 0x01 (DRR).

2.2) transport set as appropriate for the sender.

2.3) ipaddressport set to the peer’s associated transport address.

2.4) The destination structure MUST contain one value, defined as type peer and set with the sending peer’s own values.

6.4.  Request And Response Processing

This section gives normative text for message processing after DRR is introduced. Here, we only describe the additional procedures for supporting DRR. Please refer to [I-D.ietf-p2psip-base] for RELOAD base procedures.
6.4.1. Destination Peer: Receiving a Request And Sending a Response

When the destination peer receives a request, it will check the options in the forwarding header. If the destination peer can not understand extensive_routing_mode option in the request, it MUST attempt to use SRR to return an "Error_Unknown_Extension" response (defined in Section 5.3.3.1 and Section 13.9 in [I-D.ietf-p2psip-base]) to the sending peer.

If the routing mode is DRR, the peer MUST construct the Destination list for the response with only one entry, using the sending peer’s node-id from the option in the request as the value.

In the event that the routing mode is set to DRR and there is not exactly one destination, the destination peer MUST try to return an "Error_Unknown_Extension" response (defined in Section 5.3.3.1 and Section 13.9 in [I-D.ietf-p2psip-base]) to the sending peer using SRR.

After the peer constructs the destination list for the response, it sends the response to the transport address which is indicated in the ipaddressport field in the option using the specific transport mode in the Forwardingoption. If the destination peer receives a retransmit with SRR preference on the message it is trying to respond to now, the responding peer should abort the DRR response and use SRR.

6.4.2. Sending Peer: Receiving a Response

Upon receiving a response, the peer follows the rules in [I-D.ietf-p2psip-base]. The peer should note if DRR worked in order to decide if to offer DRR again. If the peer does not receive a response until the timeout it SHOULD resend the request using SRR.

7. Optional Methods to Investigate Node Connectivity

This section is for informational purposes only for providing some mechanisms that can be used when the configuration information does not specify if DRR can be used. It summarizes some methods which can be used for a node to determine its own network location compared with NAT. These methods may help a node to decide which routing mode it may wish to try. Note that there is no foolproof way to determine if a node is publically reachable, other than via out-of-band mechanisms. As such, peers using these mechanisms may be able to optimize traffic, but must be able to fall back to SRR routing if the other routing mechanisms fail.
For DRR to function correctly, a node may attempt to determine whether it is publicly reachable. If it is not, the peers should fall back to SRR. If the peer believes it is publically reachable, DRR may be attempted. NATs and firewalls are two major contributors preventing DRR from functioning properly. There are a number of techniques by which a node can get its reflexive address on the public side of the NAT. After obtaining the reflexive address, a peer can perform further tests to learn whether the reflexive address is publicly reachable. If the address appears to be publically reachable, the nodes to which the address belongs can use DRR for responses.

Some conditions are unique in P2PSIP architecture which could be leveraged to facilitate the tests. In P2P overlay network, each node only has partial a view of the whole network, and knows of a few nodes in the overlay. P2P routing algorithms can easily deliver a request from a sending node to a peer with whom the sending node has no direct connection. This makes it easy for a node to ask other nodes to send unsolicited messages back to the requester.

In the following sections, we first introduce several ways for a node to get the addresses needed for the further tests. Then a test for learning whether a peer may be publicly reachable is proposed.

7.1. Getting Addresses To Be Used As Candidates for DRR

In order to test whether a peer may be publicly reachable, the node should first get one or more addresses which will be used by other nodes to send him messages directly. This address is either a local address of a node or a translated address which is assigned by a NAT to the node.

STUN is used to get a reflexive address on the public side of a NAT with the help of STUN servers. There is also a STUN usage [I-D.ietf-behave-nat-behavior-discovery] to discover NAT behavior. Under RELOAD architecture, a few infrastructure servers can be leveraged for this usage, such as enrollment servers, diagnostic servers, bootstrap servers, etc.

The node can use a STUN Binding request to one of STUN servers to trigger a STUN Binding response which returns the reflexive address from the server’s perspective. If the reflexive transport address is the same as the source address of the Binding request, the node can determine that there likely is no NAT between him and the chosen infrastructure server. (Certainly, in some rare cases, the allocated address happens to be the same as the source address. Further tests will detect this case and rule it out in the end.). Usually, these infrastructure severs are publicly reachable in the overlay, so the
node can be considered publicly reachable. On the other hand, with the techniques in [I-D.ietf-behave-nat-behavior-discovery], a node can also decide whether it is behind NAT with endpoint-independent mapping behavior. If the node is behind a NAT with endpoint-independent mapping behavior, the reflexive address should also be a candidate for further tests.

UPnP-IGD is a mechanism that a node can use to get the assigned address from its residential gateway and after obtaining this address to communicate it with other nodes, the node can receive unsolicited messages from outside, even though it is behind a NAT. So the address obtained through the UPnP mechanism should also be used for further tests.

Another way that a node behind NAT can use to learn its assigned address by NAT is NAT-PMP. Like in UPnP-IGD, the address obtained using this mechanism should also be tested further.

The above techniques are not exhaustive. These techniques can be used to get candidate transport addresses for further tests.

7.2. Public Reacheability Test

Using the transport addresses obtained by the above techniques, a node can start a test to learn whether the candidate transport address is publicly reachable. The basic idea for the test is for a node to send a request and expect another node in the overlay to send back a response. If the response is received by the sending node successfully and also the node giving the response has no direct connection with the sending node, the sending node can determine that the address is probably publicly reachable and hence the node may be publicly reachable at the tested transport address.

In P2P overlay, a request is routed through the overlay and finally a destination peer will terminate the request and give the response. In a large system, there is a high probability that the destination peer has no direct connection with the sending node. Especially in RELOAD architecture, every node maintains a connection table. So it is easier for a node to check whether it has direct connection with another node.

Note: Currently, no existing message in base RELOAD can achieve the test. In our opinion, this kind of test is within diagnostic scope, so authors hope WG can define a new diagnostic message to do that. We don’t plan to define the message in this document, for the objective of this draft is to propose an extension to support DRR. The following text is informative.
If a node wants to test whether its transport address is publicly reachable, it can send a request to the overlay. The routing for the test message would be different from other kinds of requests because it is not for storing/fetching something to/from the overlay or locating a specific node, instead it is to get a peer who can deliver the sending node an unsolicited response and which has no direct connection with him. Each intermediate peer receiving the request first checks whether it has a direct connections with the sending peer. If there is a direct connection, the request is routed to the next peer. If there is no direct connection, the intermediate peer terminates the request and sends the response back directly to the sending node with the transport address under test.

After performing the test, if the peer determines that it may be publicly reachable, it can try DRR in subsequent transaction.

8. Security Considerations

As a routing alternative, the security part of DRR conforms to section 12.6 in based draft[I-D.ietf-p2psip-base] which describes routing security.

9. IANA Considerations

9.1. A new RELOAD Forwarding Option

A new RELOAD Forwarding Option type is add to the Registry.

Type: 0x2 - extensive_routing_mode

10. Acknowledgements

David Bryan has helped extensively with this document, and helped provide some of the text, analysis, and ideas contained here. The authors would like to thank Ted Hardie, Narayanan Vidya, Dondeti Lakshminath, Bruce Lowekamp, Stephane Bryant and Marc Petit-Huguenin for their constructive comments.

11. References

11.1. Normative References

11.2. Informative References


Authors’ Addresses

Ning Zong (editor)
Huawei Technologies

Email: zongning@huawei.com

Xingfeng Jiang
Huawei Technologies

Email: jiang.x.f@huawei.com

Roni Even
Huawei Technologies

Email: even.roni@huawei.com

Yunfei Zhang
China Mobile

Email: zhangyunfei@chinamobile.com
Abstract

This document proposes an optional extension to RELOAD to support relay peer routing mode. RELOAD recommends symmetric recursive routing for routing messages. The new optional extension provides a shorter route for responses reducing the overhead on intermediary peers and describes the potential cases where this extension can be used.

Status of this Memo

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

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

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

This Internet-Draft will expire on June 2, 2012.

Copyright Notice

Copyright (c) 2011 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect
Table of Contents

1. Introduction ........................................ 4
   1.1. Backgrounds ...................................... 4
2. Terminology ........................................... 4
3. Problem Statement ...................................... 5
   3.1. Overview ......................................... 5
   3.1.1. Relay Peer Routing (RPR) ..................... 6
   3.2. Scenarios Where RPR Benefits ................... 6
   3.2.1. Managed or Closed P2P System ................. 6
   3.2.2. Using Bootstrap Peers as Relay Peers .......... 7
   3.2.3. Wireless Scenarios .......................... 7
4. Relationship Between SRR and RPR ........................ 7
   4.1. How RPR Works ................................... 7
   4.2. How SRR and RPR Work Together .................... 8
5. Comparison on cost of SRR and RPR ........................ 8
   5.1. Closed or managed networks ...................... 8
   5.2. Open networks .................................... 9
6. Extensions to RELOAD .................................. 9
   6.1. Basic Requirements ............................... 9
   6.2. Modification To RELOAD Message Structure ........ 10
   6.2.1. State-keeping Flag ............................ 10
   6.2.2. Extensive Routing Mode ........................ 10
   6.3. Creating a Request ............................... 10
   6.3.1. Creating a request for RPR ................... 10
   6.4. Request And Response Processing ................. 11
   6.4.1. Destination Peer: Receiving a Request And Sending
          a Response ....................................... 11
   6.4.2. Sending Peer: Receiving a Response ............ 12
   6.4.3. Relay Peer Processing .......................... 12
7. Discovery Of Relay Peer ............................... 12
8. Optional Methods to Investigate Node Connectivity ........ 12
9. Security Considerations ................................ 13
10. IANA Considerations .................................. 13
11. Acknowledgements .................................... 13
12. References .......................................... 14
   12.1. Normative References ......................... 14
   12.2. Informative References ....................... 14
Authors’ Addresses ...................................... 15
1. Introduction

1.1. Backgrounds

RELOAD [I-D.ietf-p2psip-base] recommends symmetric recursive routing (SRR) for routing messages and describes the extensions that would be required to support additional routing algorithms. Other than SRR, two other routing options: direct response routing (DRR) and relay peer routing (RPR) are also discussed in Appendix D in [I-D.ietf-p2psip-base]. DRR is specified in [I-D.ietf-p2psip-drr]. As we show in section 3, RPR is advantageous over SRR in some scenarios reducing load (CPU and link BW) on intermediary peers. RPR works better in a network where relay peers are provisioned in advance so that relay peers are publicly reachable in the P2P system. In other scenarios, using a combination of RPR and SRR together is more likely to bring benefits than if SRR is used alone. Some discussion on connectivity is in Non-Transitive Connectivity and DHTs [http://srhea.net/papers/ntr-worlds05.pdf].

Note that in this draft, we focus on RPR routing mode and its extensions to RELOAD. Some text such as modification to RELOAD message structure, optional methods to investigate node connectivity described in DRR draft [I-D.ietf-p2psip-drr] are also relevant to RPR.

We first discuss the problem statement in Section 3, then how to combine RPR and SRR is presented in Section 4. In Section 5, we give comparison on the cost of SRR and RPR in both managed and open networks. An extension to RELOAD to support RPR is proposed in Section 6. Discovery of relay peers is introduced in Section 7. Some optional methods to check node connectivity is introduced in Section 8.

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. We also use terms defined in NAT behavior discovery [I-D.ietf-behave-nat-behavior-discovery]. Other terms used in this document are defined inline when used and are also defined below for reference.

There are two types of roles in the RELOAD architecture: peer and
client. Node is used when describing both peer and client. In discussions specific to behavior of a peer or client, the term peer or client is used instead.

Publicly Reachable: A node is publicly reachable if it can receive unsolicited messages from any other node in the same overlay. Note: "publicly" does not mean that the nodes must be on the public Internet, because the RELOAD protocol may be used in a closed system.

Relay Peer: A type of publicly reachable peer that can receive unsolicited messages from all other nodes in the overlay and forward the responses from destination peers towards the request sender.

Relay Peer Routing (RPR): refers to a routing mode in which responses to P2PSIP requests are sent by the destination peer to a relay peer transport address who will forward the responses towards the sending peer. For simplicity, the abbreviation RPR is used instead in the following text.

Symmetric Recursive Routing (SRR): refers to a routing mode in which responses follow the request path in the reverse order to get back to the sending peer. For simplicity, the abbreviation SRR is used instead in the following text.

3. Problem Statement

RELOAD is expected to work under a great number of application scenarios. The situations where RELOAD is to be deployed differ greatly. For instance, some deployments are global, such as a Skype-like system intended to provide public service. Some run in closed networks of small scale. SRR works in any situation, but RPR may work better in some specific scenarios.

3.1. Overview

RELOAD is a simple request-response protocol. After sending a request, a node waits for a response from a destination node. There are several ways for the destination node to send a response back to the source node. In this section, we will provide detailed information on RPR.

Note that the same illustrative settings can be found in DRR draft [I-D.ietf-p2psip-drr].
3.1.1. Relay Peer Routing (RPR)

If peer A knows it is behind a NAT or NATs, and knows one or more
relay peers with whom they have a prior connections, peer A can try
RPR. Assume A is associated with relay peer R. When sending the
request, peer A includes information describing peer R transport
address in the request. When peer X receives the request, peer X
sends the response to peer R, which forwards it directly to Peer A on
the existing connection. Note that RPR also allows a shorter route
for responses compared to SRR, which means less overhead on
intermediary peers. Establishing a connection to the relay with TLS
requires multiple round trips. Please refer to Section 5 for cost
comparison between SRR and RPR.

This technique relies on the relative population of nodes such as A
that require relay peers and peers such as R that are capable of
serving as a relay peers. It also requires mechanism to enable peers
to know which nodes can be used as their relays. This mechanism may
be based on configuration, for example as part of the overlay
configuration an initial list of relay peers can be supplied.
Another option is in a response to ATTACH request the peer can signal
that it can be used as a relay peer.

A            B            C             D          X           R
|  Request  |        |            |            |           |
|----------->|        |            |            |           |

|  Request  |            |  Request  |            |            |
|----------->|            |----------->|            |            |

<----------------|---------------------|------------------------|

3.2. Scenarios Where RPR Benefits

In this section, we will list several scenarios where using RPR would
provide improved performance.

3.2.1. Managed or Closed P2P System

As described in Section 3.2.1, many P2P systems run in a closed or
managed environment so that network administrators can better manage
their system. For example, the network administrator can deploy
several relay peers which are publicly reachable in the system and indicate their presence in the configuration file. After learning where these relay peers are, peers behind NATs can use RPR with the help from these relay peers. Peers must also support SRR in case RPR fails.

Another usage is to install relay peers on the managed network boundary allowing external peers to send responses to peers inside the managed network.

3.2.2. Using Bootstrap Peers as Relay Peers

Bootstrap peers must be publicly reachable in a RELOAD architecture. As a result, one possible architecture would be to use the bootstrap peers as relay peers for use with RPR. The requirements for being a relay peer are publicly accessible and maintaining a direct connection with its client. As such, bootstrap peers are well suited to play the role of relay peers.

3.2.3. Wireless Scenarios

While some mobile deployments may use clients, in mobile networks using peers, RPR may reduce radio battery usage and bandwidth usage by the intermediary peers. The service provider may recommend in the configuration using RPR based on his knowledge of the topology. Such relay peers may also help connectivity to external networks.

4. Relationship Between SRR and RPR

4.1. How RPR Works

Peers using RPR must maintain a connection with their relay peer(s). This can be done in the same way as establishing a neighbor connection between peers by using the Attach method.

A requirement for RPR is for the source peer to convey their relay peer (or peers) transport address in the request, so the destination peer knows where the relay peer are and send the response to a relay peer first. The request should include also the requesting peer information enabling the relay peer to route the response back to the right peer.

Note that being a relay peer does not require that the relay peer have more functionality than an ordinary peer. As discussed later, relay peers comply with the same procedure as an ordinary peer to forward messages. The only difference is that there may be a larger traffic burden on relay peers. Relay peers can decide whether to
accept a new connection based on their current burden.

4.2. How SRR and RPR Work Together

RPR is not intended to replace SRR. As seen from Section 3, RPR has better performance in some scenarios, but have limitations as well, see for example section 4.3 in Non-Transitive Connectivity and DHTs [http://srhea.net/papers/ntr-worlds05.pdf]. As a result, it is better to use these two modes together to adapt to each peer’s specific situation. Note that the informative suggestions on how to transition between SRR and RPR (e.g. compute success rate of RPR, fall back to SRR, etc) are same with that on DRR and RPR. Please refer to DRR draft [I-D.ietf-p2psip-drr] for more details. Similarly, the node can decide whether to try RPR based on other information such as configuration file information. If a relay peer is provided by the service provider, nodes may prefer RPR over SRR.

5. Comparison on cost of SRR and RPR

The major advantages in using RPR are in going through less intermediary peers on the response. By doing that it reduces the load on those peers’ resources like processing and communication bandwidth.

5.1. Closed or managed networks

As described in Section 3, many P2P systems run in a closed or managed environment (e.g. carrier networks) so that network administrators would know that they could safely use RPR.

The number of hops for a response in SRR and RPR are listed in the following table. Note that the same illustrative settings can be found in DRR draft [I-D.ietf-p2psip-drr].

<table>
<thead>
<tr>
<th>Mode</th>
<th>Success</th>
<th>No. of Hops</th>
<th>No. of Msgs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR</td>
<td>Yes</td>
<td>logN</td>
<td>logN</td>
</tr>
<tr>
<td>RPR</td>
<td>Yes</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>RPR(DTLS)</td>
<td>Yes</td>
<td>2</td>
<td>7+2</td>
</tr>
</tbody>
</table>

From the above comparison, it is clear that:

1) In most cases of N > 4 (2^2), RPR has fewer hops than SRR. Shorter route means less overhead and resource usage on intermediary peers, which is an important consideration for adopting RPR in the cases where the resource such as CPU and BW is limited, e.g. the case of mobile, wireless network.
2) In the cases of N > 512 (2^9), RPR also has fewer messages than SRR.

3) In the cases where N < 512, RPR has more messages than SRR (but still has fewer hops than SRR). So the consideration to use RPR or SRR depends on other factors like using less resources (bandwidth and processing) from the intermediaries peers. Section 4 provides use cases where RPR has better chance to work or where the intermediary resources considerations are important.

5.2. Open networks

In open network where RPR is not guaranteed, RPR can fall back to SRR if it fails after trial, as described in Section 4. Based on the same settings in Section 5.1, the number of hops, number of messages for a response in SRR and RPR are listed in the following table.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Success</th>
<th>No. of Hops</th>
<th>No. of Msgs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR</td>
<td>Yes</td>
<td>logN</td>
<td>logN</td>
</tr>
<tr>
<td>RPR</td>
<td>Yes</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>RPR (DTLS)</td>
<td>Fail &amp; Fall back to SRR</td>
<td>2+logN</td>
<td>2+logN</td>
</tr>
</tbody>
</table>

From the above comparison, it can be observed that:

1) Trying RPR would still have a good chance of fewer hops than SRR. The detailed analysis is same as DRR case and can be found in DRR draft [I-D.ietf-p2psip-drr].

2) In the cases of large network and the success rate of RPR is good, it is still possible that RPR has fewer messages than SRR. Otherwise, the consideration to use RPR or SRR depends on other factors like using less resources from the intermediaries peers.

6. Extensions to RELOAD

Adding support for RPR requires extensions to the current RELOAD protocol. In this section and in DRR [I-D.ietf-p2psip-drr], we define the changes required to the protocol, including changes to message structure and to message processing.

6.1. Basic Requirements

The basic requirements to peers for supporting RPR are same as DRR case. Please refer to DRR draft [I-D.ietf-p2psip-drr].
6.2. Modification To RELOAD Message Structure

RELOAD provides an extensible framework to accommodate future extensions. In this section and in DRR[I-D.ietf-p2psip-drr], we define a ForwardingOption structure to support RPR mode.

6.2.1. State-keeping Flag

The state-keeping flag to support RPR is same as DRR case. Please refer to DRR draft [I-D.ietf-p2psip-drr].

6.2.2. Extensive Routing Mode

The ForwardingOption structure to support RPR is same as DRR case. Please refer to DRR draft [I-D.ietf-p2psip-drr]. The definition of the fields is as follow:

RouteMode: refers to which type of routing mode is indicated to the destination peer. Currently, only DRR (specified in DRR draft [I-D.ietf-p2psip-drr]) and RPR are defined.

OverlayLinkType: refers to the transport type which is used to deliver responses from the destination peer to the relay peer.

IpAddressPort: refers to the transport address that the destination peer should use to send the response to. This will be a relay peer address for RPR.

Destination: refers to the relay peer itself. If the routing mode is RPR, then the destination contains two destinations, which are the relay peer’s node-id and the sending node’s node-id.

6.3. Creating a Request

6.3.1. Creating a request for RPR

When using RPR for a transaction, the sending peer MUST set the IGNORE-STATE-KEEPING flag in the ForwardingHeader. Additionally, the peer MUST construct and include a ForwardingOptions structure in the ForwardingHeader. When constructing the ForwardingOption structure, the fields MUST be set as follows:

1) The type MUST be set to extensive_routing_mode.

2) The ExtensiveRoutingModeOption structure MUST be used for the option field within the ForwardingOptions structure. The fields MUST be defined as follows:
2.1) routemode set to 0x02 (RPR).

2.2) transport set as appropriate for the relay peer.

2.3) ipaddressport set to the transport address of the relay peer that the sender wishes the message to be relayed through.

2.4) destination structure MUST contain two values. The first MUST be defined as type peer and set with the values for the relay peer. The second MUST be defined as type peer and set with the sending peer’s own values.

6.4. Request And Response Processing

This section gives normative text for message processing after RPR is introduced. Here, we only describe the additional procedures for supporting RPR. Please refer to [I-D.ietf-p2psip-base] for RELOAD base procedures.

6.4.1. Destination Peer: Receiving a Request And Sending a Response

When the destination peer receives a request, it will check the options in the forwarding header. If the destination peer can not understand extensive_routing_mode option in the request, it MUST attempt to use SRR to return an "Error_Unknown_Extension" response (defined in Section 5.3.3.1 and Section 13.9 in [I-D.ietf-p2psip-base]) to the sending peer.

If the routing mode is RPR, the destination peer MUST construct a Destination list for the response with two entries. The first MUST be set to the relay peer node-id from the option in the request and the second MUST be the sending node node-id from the option of the request.

In the event that the routing mode is set to RPR and there are not exactly two destinations the destination peer MUST try to send an "Error_Unknown_Extension" response (defined in Section 5.3.3.1 and Section 13.9 in [I-D.ietf-p2psip-base]) to the sending peer using SRR.

After the peer constructs the destination list for the response, it sends the response to the transport address which is indicated in the ipaddressport field in the option using the specific transport mode in the Forwardingoption. If the destination peer receives a retransmit with SRR preference on the message it is trying to response to now, the responding peer should abort the RPR response and use SRR.
6.4.2. Sending Peer: Receiving a Response

Upon receiving a response, the peer follows the rules in [I-D.ietf-p2psip-base]. If the sender used RPR and does not get a response until the timeout, it MAY either resend the message using RPR but with a different relay peer (if available), or resend the message using SRR.

6.4.3. Relay Peer Processing

Relay peers are designed to forward responses to nodes who are not publicly reachable. For the routing of the response, this draft still uses the destination list. The only difference from SRR is that the destination list is not the reverse of the via-list, instead it is constructed from the forwarding option as described below.

When a relay peer receives a response, it MUST follow the rules in [I-D.ietf-p2psip-base]. It receives the response, validates the message, re-adjust the destination-list and forward the response to the next hop in the destination list based on the connection table. There is no added requirement for relay peer.

7. Discovery Of Relay Peer

There are several ways to distribute the information about relay peers throughout the overlay. P2P network providers can deploy some relay peers and advertise them in the configuration file. With the configuration file at hand, peers can get relay peers to try RPR. Another way is to consider relay peer as a service and then some service advertisement and discovery mechanism can also be used for discovering relay peers, for example, using the same mechanism as used in TURN server discovery in base RELOAD [I-D.ietf-p2psip-base]. Another option is to let a peer advertise his capability to be a relay in the response to ATTACH or JOIN.

8. Optional Methods to Investigate Node Connectivity

This section is for informational purposes only for providing some mechanisms that can be used when the configuration information does not specify if RPR can be used. It summarizes some methods which can be used for a node to determine its own network location compared with NAT. These methods may help a node to decide which routing mode it may wish to try. Note that there is no foolproof way to determine if a node is publically reachable, other than via out-of-band mechanisms. As such, peers using these mechanisms may be able to optimize traffic, but must be able to fall back to SRR routing if the
other routing mechanisms fail.

For RPR to function correctly, a node may attempt to determine whether it is publicly reachable. If it is not, RPR may be chosen to route the response with the help from relay peers, or the peers should fall back to SRR. NATs and firewalls are two major contributors preventing RPR from functioning properly. There are a number of techniques by which a node can get its reflexive address on the public side of the NAT. After obtaining the reflexive address, a peer can perform further tests to learn whether the reflexive address is publicly reachable. If the address appears to be publicly reachable, the nodes to which the address belongs can be a candidate to serve as a relay peer. Nodes which are not publicly reachable may still use RPR to shorten the response path with the help from relay peers.

Some conditions are unique in P2PSIP architecture which could be leveraged to facilitate the tests. In P2P overlay network, each node only has partial a view of the whole network, and knows of a few nodes in the overlay. P2P routing algorithms can easily deliver a request from a sending node to a peer with whom the sending node has no direct connection. This makes it easy for a node to ask other nodes to send unsolicited messages back to the requester.

The approaches for a node to get the addresses needed for the further tests, as well as the test for learning whether a peer may be publicly reachable is same as the DRR case. Please refer to DRR draft [I-D.ietf-p2psip-drr] for more details.

9. Security Considerations

As a routing alternative, the security part of RPR conforms to section 12.6 in based draft[I-D.ietf-p2psip-base] which describes routing security.

10. IANA Considerations

No IANA action is needed.

11. Acknowledgements

David Bryan has helped extensively with this document, and helped provide some of the text, analysis, and ideas contained here. The authors would like to thank Ted Hardie, Narayanan Vidya, Dondeti Lakshminath, Bruce Lowekamp, Stephane Bryant and Marc Petit-Huguenin
for their constructive comments.

12. References

12.1. Normative References


12.2. Informative References


Authors’ Addresses

Ning Zong (editor)
Huawei Technologies
Email: zongning@huawei.com

Xingfeng Jiang
Huawei Technologies
Email: jiang.x.f@huawei.com

Roni Even
Huawei Technologies
Email: even.roni@huawei.com

Yunfei Zhang
China Mobile
Email: zhangyunfei@chinamobile.com
Abstract

REsource LOcation and Discovery (RELOAD) does not define a generic service discovery mechanism as part of the base protocol. This document defines how the Recursive Distributed Rendezvous (ReDiR) service discovery mechanism used in OpenDHT can be applied to RELOAD overlays to provide a generic service discovery mechanism.

Status of this Memo

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

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

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

This Internet-Draft will expire on July 9, 2012.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.
Table of Contents

1.  Introduction ................................................. 3
2.  Terminology .................................................. 4
3.  Introduction to ReDiR ........................................ 4
4.  Using ReDiR in a RELOAD Overlay Instance ................. 6
   4.1.  Data Structure ....................................... 6
   4.2.  Selecting the Starting Level ......................... 7
   4.3.  Service Provider Registration ....................... 7
   4.4.  Refreshing Registrations ............................ 8
   4.5.  Service Lookups ..................................... 8
   4.6.  Removing Registrations ............................... 9
5.  Access Control Rules ....................................... 9
6.  REDIR Kind Definition ...................................... 10
7.  Example ....................................................... 10
8.  Overlay Configuration Document Extension ................. 12
9.  Security Considerations .................................... 12
10. IANA Considerations ........................................ 13
   10.1. Access Control Policies ............................ 13
   10.2. Data Kind-ID ......................................... 13
   10.3. ReDiR Namespaces .................................... 13
11. Acknowledgments ............................................. 13
12. References .................................................... 14
   12.1. Normative References ............................... 14
   12.2. Informative References ............................ 14
Authors’ Addresses ............................................... 14
1. Introduction

REsource LOcation And Discovery (RELOAD) [I-D.ietf-p2psip-base] is a peer-to-peer signaling protocol that can be used to maintain an overlay network, and to store data in and retrieve data from the overlay. Although RELOAD defines a Traversal Using Relays around Network Address Translation (TURN) specific service discovery mechanism, it does not define a generic service discovery mechanism as part of the base protocol. This document defines how the Recursive Distributed Rendezvous (ReDiR) service discovery mechanism [Redir] used in OpenDHT can be applied to RELOAD overlays.

In a Peer-to-Peer (P2P) overlay network such as a RELOAD Overlay Instance, the peers forming the overlay share their resources in order to provide the service the system has been designed to provide. The peers in the overlay both provide services to other peers and request services from other peers. Examples of possible services peers in a RELOAD Overlay Instance can offer to each other include a TURN relay service, a voice mail service, a gateway location service, and a transcoding service. Typically, only a small subset of the peers participating in the system are providers of a given service. A peer that wishes to use a particular service faces the problem of finding peers that are providing that service from the Overlay Instance.

A naive way to perform service discovery is to store the Node-IDs of all nodes providing a particular service under a well-known key k. The limitation of this approach is that it scales linearly in the number of nodes that provide the service. The problem is two-fold: the node n that is responsible for service s identified by key k may end up storing a large number of Node-IDs and most importantly, may also become overloaded since all service lookup requests for service s will need to be answered by node n. An efficient service discovery mechanism does not overload the nodes storing pointers to service providers. In addition, the mechanism must ensure that the load of providing a given service is distributed evenly among the nodes providing the service.

ReDiR implements service discovery by building a tree structure of the nodes that provide a particular service and embedding it into the RELOAD Overlay Instance using RELOAD Store and Fetch requests. Each service provided in the Overlay Instance has its own tree. The nodes in a ReDiR tree contain pointers to service providers. During service discovery, a peer wishing to use a given service fetches ReDiR tree nodes one-by-one from the RELOAD Overlay Instance until it finds a service provider responsible for its Node-ID. It has been proved that ReDiR can find a service provider using only a constant number of Fetch operations [Redir].
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].

This document uses the terminology and definitions from the Concepts and Terminology for Peer to Peer SIP [I-D.ietf-p2psip-concepts] draft.

DHT: Distributed Hash Tables (DHTs) are a class of decentralized distributed systems that provide a lookup service similar to a hash table. Given a key, any participating peer can retrieve the value associated with that key. The responsibility for maintaining the mapping from keys to values is distributed among the peers.

H(x): Hash calculated over x.

I(l,k): The unique interval at level l in the ReDiR tree that encloses key k.

n.id: Node-ID of node n.

Namespace: An arbitrary identifier that identifies a service provided in the RELOAD Overlay Instance. An example of a namespace is "voice-mail". The namespace is an UTF-8 text string.

numBitsInNodeId: Number of bits in a Node-ID.

ReDiR tree: A tree structure of the nodes that provide a particular service. The nodes embed the ReDiR tree into the RELOAD Overlay Instance using RELOAD Store and Fetch requests.

Successor: The successor of identifier k in namespace ns is the node that has joined ns whose identifier most immediately follows k.

3. Introduction to ReDiR

Recursive Distributed Rendezvous (ReDiR) [Redir] does not require new functionality from the RELOAD base protocol. This is possible since ReDiR interacts with the RELOAD overlay through a put/get API using RELOAD Store and Fetch requests. ReDiR builds a tree structure of the nodes that provide a particular service and embeds it into the RELOAD Overlay Instance using the Store and Fetch requests. ReDiR
performs lookup in a logarithmic number of Fetch operations with high probability. Further, if the tree's height is estimated based on past lookups, the average lookup can be reduced to a constant number of Fetch operations assuming that Node-IDs are distributed uniformly at random.

In ReDiR, each service provided in the overlay is identified by an arbitrary identifier, called its namespace. All service providers join a namespace and peers wishing to use a service perform lookups within the namespace of the service. A ReDiR lookup for identifier k in namespace ns returns a node that has joined ns whose identifier is the closest successor of k.

Each tree node in the ReDiR tree contains a list of Node-IDs of peers providing a particular service. Each node in the tree has a level. The root is at level 0, the immediate children of the root are at level 1, and so forth. The ReDiR tree has a branching factor, whose value is determined by a new element in the RELOAD overlay configuration document, called branching-factor. At every level i in the tree, there are at most branching-factor^i nodes. The nodes at any level are labeled from left to right, such that a pair (i,j) uniquely identifies the jth node from the left at level i. This tree is embedded into the RELOAD Overlay Instance node by node, by storing the values of node (i,j) at key H(namespace,i,j). As an example, the root of the tree for a voice mail service is stored at H("voice-mail",0,0). Each node (i,j) in the tree is associated with branching-factor intervals of the DHT keyspace as follows:

\[
[2^{\text{numBitsInNodeID}}b^i(-i)*(j+(b'/b)),
2^{\text{numBitsInNodeID}}b^i(-i)*(j+((b'+1)/b))], \text{ for } 0\leq b' < b,
\]

where b is the branching-factor.

Figure 1 shows an example of a ReDiR tree with a branching factor of 2. Each node is shown as two horizontal lines separated by a vertical bar. The lines represent the two intervals each node is responsible for. At level 0, there is only one node, (0,0) responsible for two intervals that together cover the entire identifier space of the RELOAD Overlay Instance. At level 1, there are two nodes, (1,0) and (1,1), each of which is responsible for half of the identifier space. At level 2, there are four nodes. Each of them owns one fourth of the identifier space. At level 3, there are eight nodes each of which is responsible for one eighth of the identifier space.
4. Using ReDiR in a RELOAD Overlay Instance

4.1. Data Structure

ReDiR tree nodes are stored using the dictionary data model defined in RELOAD base [I-D.ietf-p2psip-base]. The data stored is a RedirServiceProvider Resource Record:

```c
enum { none(0), (255) }
    RedirServiceProviderExtType;

struct {
    RedirServiceProviderExtType   type;
   NodeId                        serviceProvider;
    opaque                        namespace<0..2^16-1>;
    uint16                        level;
    uint16                        node;
    uint16                        length;

    select (type) {
        /* This type may be extended */
    } extension;

} RedirServiceProvider;
```

The contents of the RedirServiceProvider Resource Record are as follows:

- type
The type of an extension to the RedirServiceProvider Resource Record. Unknown types are allowed.

**serviceProvider**
The Node-ID of a service provider.

**namespace**
An opaque string containing the namespace.

**level**
The level in the ReDiR tree.

**node**
The position of the node storing this RedirServiceProvider record at the current level in the ReDiR tree.

**length**
The length of the rest of the Resource Record.

**extension**
An extension value. The RedirServiceProvider Resource Record can be extended to include for instance service or service provider specific information.

### 4.2. Selecting the Starting Level

Before registering as a service provider or performing a service lookup, a peer needs to determine the starting level \( L_{\text{start}} \) for the registration or lookup operation in the ReDiR tree. It is RECOMMENDED that \( L_{\text{start}} \) is initially set to 2. In subsequent registrations, \( L_{\text{start}} \) MUST be set to the lowest level at which registration last completed.

In the case of service lookups, nodes MUST record the levels at which the last 16 service lookups completed and take \( L_{\text{start}} \) to be the mode of those depths.

### 4.3. Service Provider Registration

A node MUST use the following procedure to register as a service provider in the RELOAD Overlay Instance:

1. A node \( n \) with Node-ID \( n\.id \) wishing to register as a service provider starts from level \( L_{\text{start}} \) (see Section 4.2 for the details on selecting the starting level). Therefore, node \( n \) sets level=\( L_{\text{start}} \).
2. Node n MUST send a RELOAD Fetch request to fetch the contents of the tree node associated with I(level,n.id). An interval I(l,k) is defined as the unique interval at level l in the ReDiR tree that encloses key k. The fetch MUST be a wildcard fetch.

3. Node n MUST send a RELOAD Store request to add its RedirServiceProvider entry to the dictionary stored in the tree node associated with I(level,n.id)

4. If node n’s Node-ID is the numerically lowest or highest among the Node-IDs stored in the tree node associated with I(Lstart,n.id), node n MUST continue up the tree towards the root (level 0), repeating steps 2 and 3. Node n MUST continue this until it reaches either the root or a level at which n.id is not the lowest or highest Node-ID in the interval I(level,n.id).

5. Node n MUST also walk down the tree through tree nodes associated with the intervals I(Lstart,n.id),I(Lstart+1,n.id),..., at each step fetching the current contents of the tree node, and storing its RedirServiceProvider record if n.id is the lowest or highest Node-ID in the interval. Node n MUST end this downward walk when it reaches a level at which it is the only service provider in the interval.

Note that above, when we refer to ‘the tree node associated with I(l,k)’, we mean the entire tree node (that is, all the intervals within the tree node) responsible for interval I(l,k). In contrast, I(l,k) refers to a specific interval within a tree node.

4.4. Refreshing Registrations

All state in the ReDiR tree is soft. Therefore, a service provider needs to periodically repeat the registration process to refresh its RedirServiceProvider Resource Record. If a record expires, it MUST be dropped from the dictionary by the peer storing the tree node. Deciding an appropriate lifetime for the RedirServiceProvider Resource Records is up to each service provider. Every service provider MUST repeat the entire registration process periodically until it leaves the RELOAD Overlay Instance.

Note that no new mechanisms are needed to keep track of the remaining lifetime of RedirServiceProvider records. The ‘storage_time’ and ‘lifetime’ fields of RELOAD’s StoredData structure can be used for this purpose in the usual way.

4.5. Service Lookups

The purpose of a service lookup on identifier k in namespace ns is to find the node that has joined ns whose identifier most immediately follows identifier k.
A service lookup is similar to the registration operation. The service lookup starts from some level level=Lstart (see Section 4.2 for the details on selecting the starting level). At each step, the node n wishing to discover a service provider MUST fetch the tree node associated with the current interval I(level,n.id) using a RELOAD Fetch request and MUST determine where to look next as follows:

1. If the successor of node n is not present in the tree node associated with I(level,n.id), then its successor must occur in a larger range of the keyspace. Node n MUST set level=level-1 and try again.
2. If n.id is sandwiched between two Node-IDs in I(level,n.id), the successor must lie somewhere in I(level,n.id). Node n MUST set level=level+1 and repeat.
3. Otherwise, there is a Node-ID s stored in the node associated with I(level,n.id) whose identifier succeeds n.id, and there are no Node-IDs between n.id and s. Thus, s must be the closest successor of n.id, and the lookup is done.

Note that above, when we refer to ‘the tree node associated with I(l,k)’, we mean the entire tree node (that is, all the intervals within the tree node) responsible for interval I(l,k). In contrast, I(l,k) refers to a specific interval within a tree node.

4.6. Removing Registrations

Before leaving the RELOAD Overlay Instance, a service provider MUST remove the RedirServiceProvider records it has stored by storing exists=False values in their place, as described in [I-D.ietf-p2psip-base].

5. Access Control Rules

As specified in RELOAD base [I-D.ietf-p2psip-base], 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. Usages can define any access control rules they choose, including publicly writable values.

ReDiR requires an access control policy that allows multiple nodes in the overlay read and write access to the ReDiR tree nodes stored in the overlay. Therefore, none of the access control policies specified in RELOAD base [I-D.ietf-p2psip-base] is sufficient.

This document defines a new access control policy, called NODE-ID-
MATCH. In this policy, a given value MUST be written and overwritten only if the request is signed with a key associated with a certificate whose Node-ID is equal to the dictionary key. In addition, provided that exists=TRUE, the Node-ID MUST belong to one of the intervals associated with the tree node (the number of intervals each tree node has is determined by the branching-factor parameter). Finally, provided that exists=TRUE, H(namespace, level, node), where namespace, level, and node are taken from the RedirServiceProvider structure being stored, MUST be equal to the Resource-ID for the resource. The NODE-ID-MATCH policy may only be used with dictionary types.

6. REDIR Kind Definition

This section defines the REDIR kind.

Name
REDIR

Kind IDs
The Resource Name for the REDIR Kind-ID is created by concatenating three pieces of information: namespace, level, and node number. Namespace is a string identifying a service, such as "voice-mail". Level is an integer specifying a level in the ReDiR tree. Node number is an integer identifying a ReDiR tree node at a specific level. The data stored is a RedirServiceProvider structure that was defined in Section 4.1.

Data Model
The data model for the REDIR Kind-ID is dictionary. The dictionary key is the Node-ID of the service provider.

Access Control
The access control policy for the REDIR kind is the NODE-ID-MATCH policy that was defined in Section 5.

7. Example

Figure 2 shows an example of a ReDiR tree containing information about four different service providers whose Node-IDs are 2, 3, 4, and 7. In the example, numBitsInNodeID=4. Initially, the ReDiR tree is empty.
First, peer 2 whose Node-ID is 2 joins the namespace. Since this is the first registration peer 2 performs, peer 2 sets the starting level $L_{\text{start}}$ to 2, as was described in Section 4.2. Also all other peers in this example will start from level 2. First, peer 2 fetches the contents of the tree node associated with interval $I(2,2)$ from the overlay. Peer 2 also stores its RedirServiceProvider record in that tree node. Since peer 2’s Node-ID is the only Node-ID stored in the tree node (i.e., peer 2’s Node-ID fulfills the condition that it is the numerically lowest or highest among the keys stored in the node), peer 2 continues up the tree. In fact, peer 2 continues up in the tree all the way to the root inserting its own Node-ID in all levels since the tree is empty (which means that peer 2’s Node-ID always fulfills the condition that it is the numerically lowest or highest Node-ID in the interval $I(\text{level, 2})$ during the upward walk). As described in Section 4.3, peer 2 also walks down the tree. The downward walk peer 2 does ends at level 2 since peer 2 is the only node in its interval at that level.

The next peer to join the namespace is peer 3, whose Node-ID is 3. Peer 3 starts from level 2. At that level, peer 3 stores its RedirServiceProvider record in the same interval $I(2,3)$ that already contains the Node-ID of peer 2. Since peer 3 has the numerically highest Node-ID in the tree node associated with $I(2,3)$, peer 3 continues up the tree. Peer 3 stores its RedirServiceProvider record also at levels 1 and 0 since its Node-ID is numerically highest among the Node-IDs stored in the intervals to which its own Node-ID belongs. Peer 3 also does a downward walk which starts from level 2 (i.e., the starting level). Since peer 3 is not the only node in interval $I(2,3)$, it continues down the tree to level 3. The downward walk ends at this level since peer 3 is the only service provider in the interval $I(3,3)$.

The third peer to join the namespace is peer 7, whose Node-ID is 7. Like the two earlier peers, also peer 7 starts from level 2 because this is the first registration it performs. Peer 7 stores its RedirServiceProvider record at level 2. At level 1, peer 7 has the numerically highest (and lowest) Node-ID in its interval $I(1,7)$.
(because it is the only node in interval I(1,7); peers 2 and 3 are
stored in the same tree node but in a different interval) and
therefore it stores its Node-ID in the tree node associated with that
interval. Peer 7 also has the numerically highest Node-ID in the
interval I(0,7) associated with its Node-ID at level 0. Finally,
peer 7 performs a downward walk, which ends at level 2 because peer 7
is the only node in its interval at that level.

The final peer to join the ReDiR tree is peer 4, whose Node-ID is 4.
Peer 4 starts by storing its RedirServiceProvider record at level 2.
Since it has the numerically lowest Node-ID in the tree node
associated with interval I(2,4), it continues up in the tree to level
1. At level 1, peer 4 stores its record in the tree node associated
with interval I(1,4) because it has the numerically lowest Node-ID in
that interval. Next, peer 4 continues to the root level, at which it
stores its RedirServiceProvider record and finishes the upward walk
since the root level was reached. Peer 4 also does a downward walk
starting from level 2. The downward walk stops at level 2 because
peer 4 is the only peer in the interval I(2,4).

8. Overlay Configuration Document Extension

This document extends the RELOAD overlay configuration document by
adding a new element "branching-factor" inside the new "REDIR" kind
element:

redir:branching-factor:  The branching factor of the ReDiR tree. The
default value is 10.

This new element is formally defined as follows:

namespace redir = "urn:ietf:params:xml:ns:p2p:service-discovery"
parameter & element redir:branching-factor { xsd:unsignedInt }

The 'redir' namespace is added into the <mandatory-extension> element
in the overlay configuration file.

9. Security Considerations

There are no new security considerations introduced in this document.
The security considerations of RELOAD [I-D.ietf-p2psip-base] apply.
10. IANA Considerations

10.1. Access Control Policies

This document introduces one additional access control policy to the "RELOAD Access Control Policy" Registry:

NODE-ID-MATCH

This access control policy was described in Section 5.

10.2. Data Kind-ID

This document introduces one additional data Kind-ID to the "RELOAD Data Kind-ID" Registry:

<table>
<thead>
<tr>
<th>Kind</th>
<th>Kind-ID</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>REDIR</td>
<td>104</td>
<td>RFC-A AAA</td>
</tr>
</tbody>
</table>

This Kind-ID was defined in Section 6.

10.3. ReDiR Namespaces

IANA SHALL create a "ReDiR Namespaces" Registry. Entries in this registry are strings denoting ReDiR namespace values. The initial contents of this registry are:

<table>
<thead>
<tr>
<th>Namespace</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>turn-server</td>
<td>RFC-A AAA</td>
</tr>
<tr>
<td>voice-mail</td>
<td>RFC-A AAA</td>
</tr>
</tbody>
</table>

11. Acknowledgments

The authors would like to thank Marc Petit-Huguenin for his comments on the draft.
12. References

12.1. Normative References

[I-D.ietf-p2psip-base]  


12.2. Informative References

[I-D.ietf-p2psip-concepts]  


Authors’ Addresses

Jouni Maenpaa  
Ericsson  
Hirsalantie 11  
Jorvas  02420  
Finland

Email: Jouni.Maenpaa@ericsson.com

Gonzalo Camarillo  
Ericsson  
Hirsalantie 11  
Jorvas  02420  
Finland

Email: Gonzalo.Camarillo@ericsson.com
A Constrained Application Protocol (CoAP) Usage for REsource LOcation
And Discovery (RELOAD)
draft-jimenez-p2psip-coap-reload-01

Abstract

This document defines a Constrained Application Protocol (CoAP) Usage for REsource LOcation And Discovery (RELOAD). The CoAP Usage provides the functionality to federate Wireless Sensor Networks (WSN) in a peer-to-peer fashion. The CoAP Usage also provides a rendezvous service for CoAP Nodes and caching of sensor information. The RELOAD AppAttach method is used to establish a direct connection between nodes through which CoAP messages are exchanged.

Status of this Memo

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

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

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

This Internet-Draft will expire on September 1, 2012.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents
Table of Contents

1. Introduction .............................................. 3
2. Terminology ............................................. 4
3. Architecture ............................................. 4
4. Registering CoAP URIs ................................. 6
5. Rendezvous .............................................. 6
6. Forming a direct connection and reading data .......... 7
7. Caching Mechanisms ....................................... 10
    7.1. ProxyCache ........................................ 10
    7.2. SensorCache ........................................ 11
8. CoAP Usage Kinds Definition ............................ 13
    8.1. CoAP-REGISTRATION Kind ........................... 13
    8.2. CoAP-CACHING Kind ................................ 13
9. Access Control Rules .................................... 14
10. Security Considerations ................................. 14
11. IANA Considerations ..................................... 15
    11.1. RELOAD Sensor Type Registry ..................... 15
    11.2. CoAP-REGISTRATION Kind-ID ....................... 15
    11.3. CoAP-CACHING Kind-ID ............................ 15
    11.4. Access Control Policies ......................... 16
12. References ............................................ 16
    12.1. Normative References ............................ 16
    12.2. Informative References .......................... 17
Authors’ Addresses ........................................ 17
1. Introduction

The Constrained Application Protocol (CoAP) is a specialized web transfer protocol. It realizes the Representational State Transfer (REST) architecture for the most constrained nodes, such as sensors and actuators. CoAP can be used not only between nodes on the same constrained network but also between constrained nodes and nodes on the Internet. The latter is possible since CoAP can be translated to Hypertext Transfer Protocol (HTTP) for integration with the web. Application areas of CoAP include different forms of M2M communication, such as home automation, construction, health care or transportation. Areas with heavy use of sensor and actuator devices that monitor and interact with the surrounding environment.

The CoAP Usage for RELOAD allows CoAP nodes to store resources in a RELOAD peer-to-peer overlay, provides a rendezvous service, and enables the use of RELOAD overlay as a cache for sensor data. This functionality is implemented in the RELOAD overlay itself, without the use of centralized servers. The CoAP Usage involves three basic functions:

1. Registration: CoAP nodes can use the RELOAD data storage functionality to store a mapping from their CoAP URI to their Node-ID in the overlay, and to retrieve the Node-IDs of other nodes.
2. Rendezvous: Once a CoAP node has identified the Node-ID for an URI it wishes to retrieve, it can use the RELOAD message routing system to set up a direct connection which can be used to exchange CoAP messages.
3. Caching: Nodes can use the RELOAD overlay as a caching mechanism for their sensor information. This is specially useful for battery constrained nodes that can make their data available in the cache provided by the overlay while in sleep mode.

For instance, a CoAP proxy (See Section 3) could register its Node-ID (e.g. "9996172") and a list of sensors (e.g. "/temperature-1; ./temperature-2; ./temperature-3") under its URI (e.g. "coap://overlay-1.com/proxy-1/").

When a node wants to discover the values associated with that URI, it queries the overlay for "coap://overlay-1.com/proxy-1/" and gets back the Node-ID of the proxy and the list of its associated sensors. The requesting node can then use the RELOAD overlay to establish a direct connection with the proxy and to read sensor values.

Moreover, the CoAP proxy can store the sensor information in the overlay. In this way information can be retrieved directly from the overlay without performing a direct connection to the storing proxy.
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 Concepts and Terminology for Peer to Peer SIP [I-D.ietf-p2psip-concepts] and the RELOAD Base Protocol [I-D.ietf-p2psip-base] extensively in this document.

3. Architecture

In our architecture we extend the different nodes present in RELOAD (Peer, Client) and add support for sensor devices or other constrained devices. Figure 1 illustrates our architecture. The different nodes, according to their functionality are:

Client
Devices that are capable of participating in a RELOAD overlay as client nodes, that is they do not route messages in the overlay.

Router
Devices that are members of (i.e., peers in) a RELOAD overlay and capable of forwarding RELOAD messages following a path through the overlay to the destination.

Sensor
Devices capable of measuring a physical quantity. Sensors usually acquire quantifiable information about their surrounding environment such as: temperature, humidity, electric current, moisture, radiation, and so on.

Actuator
Devices capable of interacting and affecting their environment such as: electrical motors, pneumatic actuators, electric switches, and so on.

Proxy
Devices having sufficient resources to run RELOAD either as client or peer. These devices are located at the edge of the sensor network and, in case of Wireless Sensor Networks (WSN), act as coordinators of the network.

Physical devices can have one or several of the previous functional roles. According to the functionalities that are present in each of the nodes, they can be:
Constrained Node
A Constrained Node (CN) is a node with limited computational capabilities. If it is wireless then it will be part of a Low-Rate Wireless Personal Area Network (LR-WPAN), it might be movable and often offer unreliable connectivity. Also, devices will usually be in sleep mode in order to prevent battery drain, and will not communicate during those periods. A CN is NOT part of the RELOAD overlay, therefore it can not act as a client, router nor proxy. A CN is always either a either a Sensor or an Actuator. In the latter case the node is often connected to a continuous energy power supply.

Reload Node
A Reload Node (RN) MUST implement the client functionality in the Overlay. Additionally the node will often be a full RELOAD peer with Proxy functionality. A RN may also be sensor or actuator since it can have those devices connected to it.
4. Registering CoAP URIs

CoAP URIs are typically resolved using a DNS. When CoAP is needed in a RELOAD environment, URI resolution is provided by the overlay as a whole. Instead of registering register a URI, a peer stores a CoAPRegistration structure under a hash of its own URI. This uses the CoAP REGISTRATION Kind-ID, which is formally defined in Section 6, and that uses a DICTIONARY data model.

As an example, if a CoAP proxy that is located in an overlay overlay-1.com using a Node-ID "9996172" wants to register three different temperature sensors to the URI "coap://overlay-1.com/proxy-1/.well-known/", it might store the following mapping in the overlay:

\[
\text{Resource-ID} = h(\text{coap://overlay-1.com/proxy-1/.well-known/}) \\
\text{KEY} = 9996172, \\
\text{VALUE} = \{./\text{temperature-1}; \\
\quad ./\text{temperature-2}; \\
\quad ./\text{temperature-3}\}
\]

Note that the Resource-ID stored in the overlay is calculated as hash over the URI (i.e. \(h(\text{URI})\)), for instance SHA-1 in RELOAD.

This would inform any other node performing a lookup for the previous URI "coap://overlay-1.com/proxy-1/.well-known/" that the Node-ID value for proxy-1 is "9996172". In addition, this mapping provides relevant information as to the number of sensors (CNs) and the URI path to connect to them using CoAP.

5. Rendezvous

The RELOAD overlay supports rendezvous by fetching mapping information between CoAP URIs and Node-IDs.

As an example, if a node RN located in the overlay overlay-1.com wishes to read which resources are served at a RN with URI coap://overlay-1.com/proxy-1/, it performs a fetch in the overlay. The Resource-ID used in this fetch is a SHA-1 hash over the URI "coap://overlay-1.com/proxy-1/.well-known/".

After this fetch request, the overlay will return the following result:
Resource-ID = h(coap://overlay-1.com/proxy-1/.well-known/)
KEY = 9996172,
VALUE = { ./temperature-1;
         ./temperature-2;
         ./temperature-3}

The obtained KEY is the Node-ID of the RN responsible of this KEY/VALUE pair. The VALUE is the set of URIs necessary to read data from the CNs associated with the RN.

Using the RELOAD DICTIONARY model allows for multiple nodes to perform a store to the same Resource-ID. This feature allows for performing rendezvous with multiple RNs that host CNs of the same class.

As an example, a fetch to the URI "coap://overlay-1.com/temperature/.well-known/" could return the following results:

Resource-ID = h(coap://overlay-1.com/temperature/.well-known/)
KEY = 9992323,
VALUE = { ./temperature}

KEY = 9996172,
VALUE = { ./temperature-1;
         ./temperature-2;
         ./temperature-3}

KEY = 9996173,
VALUE = { ./temp-a;
         ./temp-b}

6. Forming a direct connection and reading data

Once a RN (e.g., node-A) has obtained the rendezvous information for a node in the overlay (e.g., proxy-1), it can open a direct connection to that node. This is performed by sending an AppAttach request to the Node-ID obtained during the rendezvous process.

After the AppAttach negotiation, node-A can access to the values of the CNs at proxy-1 using the URIs obtained during the rendezvous. Following the example in Section 5, the URIs for accessing to the CNs at proxy-1 would be:
Note that the ".well-known" string has been removed from the URIs, as this is only used during CNs discovery. Figure 1 shows a sample of a node reading humidity data.
Figure 2: An Example of a Message Sequence
7. Caching Mechanisms

The CoAP protocol itself supports the caching of sensor information in order to reduce the response time and network bandwidth consumption of future, equivalent requests. This storage is done in CoAP proxies.

This CoAP usage proposes an additional caching mechanism for storing sensor information directly in the overlay. This caching mechanism is primarily intended for CNs with sensor capabilities, not for RN sensors. This is due to the battery constraints of CNs, forcing them to stay in sleep mode for long periods of time.

Whenever a CN wakes up, it sends the most recent data from its sensors to its proxy (RN), which stores the data in the overlay using a RELOAD StoredData structure defined in Section 6 of the RELOAD base draft [I-D.ietf-p2psip-base]. We use the StoredDataValue structure defined in Section 6.2 of the RELOAD base draft, in particular we use the SingleValue format type to store the cached values in the overlay. From that structure length, storage_time, lifetime and Signature are used in the same way. The only difference is DataValue which in our case can be either a ProxyCache or a SensorCache:

```c
enum { reserved (0), proxy_cache(1), sensor_cache(2), (255) } CoAPCachingType;
struct {
    CoAPCachingType coap_caching_type;
    select(coap_caching_type) {
        case proxy_cache:       ProxyCache proxy_cache_entry;
        case sensor_cache:      SensorCache sensor_cache_entry;
    }
    /* extensions */
} CoAPCaching;
```

7.1. ProxyCache

ProxyCache is meant to store values and sensor information (e.g. inactivity time) for all the sensors associated with a certain proxy, as well as their CoAP URIs. On the other hand, SensorCache is used for storing the information and cached value of only one sensor (CoAP URI is not necessary, as is the same as the one used for generating the Resource-ID associated to that SensorCache entry).
ProxyCache contains the fields Node-ID and series of SensorEntry types.

struct {
    Node-ID         Node_ID;
    uint32          length;
    SensorEntry     sensors[length];
} ProxyCache;

Node-ID
    The Node-ID of the Proxy Node (PN) responsible for different sensor devices;
length
    The length of the rest of the structure;
SensorEntry
    List of sensors in the form of SensorEntry types;

SensorEntry contains the coap_uri, sensor_info and a series of SensorValue types.

struct {
    opaque          coap_uri;
    SensorInfo      sensor_info;
    uint32          length;
    SensorValue     sensor_value[length];
} SensorEntry;

coop_uri
    CoAP name of the sensor device in question;
sensor_info
    contains relevant sensor information;
length
    The length of the rest of the structure;
sensor_value
    contains a list of values stored by the sensor;

7.2.  SensorCache

SensorCache: contains the information related to one sensor.

struct {
    Node-ID         Node_ID;
    SensorInfo      sensor_info;
    uint32          length;
    SensorValue     sensor_value[length];
} SensorCache;
Node_ID
   identifies the Node-ID of the Proxy Node responsible for the sensor;
sensor_info
   contains relevant sensor information;
length
   The length of the rest of the structure;
sensor_value
   contains a list of values stored by the sensor;

SensorInfo contains relevant sensor information, such as sensor_type, duration_of_inactivity and c fields.

```c
struct {
   integer         sensor_type;
   uint32          duration_of_inactivity;
   uint32          last_awake;

   /* extensions */
} SensorInfo;
```

sensor_type
   is an integer identifying the type of the sensor. See Figure 3;
duration_of_inactivity
   contains the sleep pattern (if any) that the sensor device follows, specified in seconds;
last_awake
   indicates the last time that the sensor was awake 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;

SensorValue contains the measurement_time, lifetime and value.

```c
struct {
   uint32          measurement_time;
   uint32          lifetime;
   opaque          value;

   /* extensions */
} SensorValue;
```
measurement_time
indicates the moment in which the measure was taken represented as
the number of milliseconds elapsed since midnight Jan 1, 1970 UTC
not counting leap seconds;
lifetime
indicates the validity time of that measured value in milliseconds
since measurement_time;
value
indicates the actual value measured. It can be of different types
(integer, long, string) therefore opaque has been used;

8. CoAP Usage Kinds Definition

This section defines the CoAP-REGISTRATION and CoAP-CACHING kinds.

8.1. CoAP-REGISTRATION Kind
Kind IDs
The Resource Name for the CoAP-REGISTRATION Kind-ID is the CoAP
URI. The data stored is a CoAPRegistration, which contains a set
of CoAP URIs.
Data Model
The data model for the CoAP-REGISTRATION Kind-ID is dictionary.
The dictionary key is the Node-ID of the storing RN. This allows
each RN to store a single mapping.
Access Control
URI-NODE-MATCH. The "coap:" prefix needs to be removed from the
COAP URI before matching. See Section 9.

Data stored under the COAP-REGISTRATION kind is of type
CoAPRegistration, defined below.

struct {
    Node-ID Node_ID;
    uint16 coap_uris_length;
    opaque coap_uris (0..2^16-1);
} CoAPRegistration;

8.2. CoAP-CACHING Kind
KindIDs
The Resource Name for the CoAP-CACHING Kind-ID is the CoAP URI.
The data stored is a CoAPCaching, which contains a cached value.
Data Model
The data model for the CoAP-CACHING Kind-ID is single value.
Access Control

URI-MATCH. The "coap:" prefix needs to be removed from the COAP URI before matching. See Section 9.

Data stored under the CoAP-CACHING kind is of type CoAPCaching, defined in Section 7.

9. Access Control Rules

As specified in RELOAD base [I-D.ietf-p2psip-base], 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. Usages can define any access control rules they choose, including publicly writable values.

CoAP Usage for RELOAD requires an access control policy that allows multiple nodes in the overlay read and write access. This access is for registering and caching information using CoAP URIs as identifiers. Therefore, none of the access control policies specified in RELOAD base are sufficient [I-D.ietf-p2psip-base].

This document defines two access control policies, called URI-MATCH and URI-NODE-MATCH. In URI-MATCH policy, a given value MUST be written and overwritten if and only if the signer’s certificate has an associated URI which canonicalized form hashes (using the hash function for the overlay) to the Resource-ID for the resource.

In URI-NODE-MATCH policy, a given value MUST be written and overwritten if and only if the signer’s certificate has an associated URI which canonicalized form 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.

These Access Control Policies are specified for IANA in Section 11.4.

10. Security Considerations

TBD.
11. IANA Considerations

11.1. RELOAD Sensor Type Registry

IANA SHALL create a "RELOAD sensor type" Registry. Entries in this registry are 16-bit integers denoting method codes as described in Section 7. The initial contents of this registry are:

<table>
<thead>
<tr>
<th>Code Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>temperature</td>
<td>0</td>
</tr>
<tr>
<td>humidity</td>
<td>1</td>
</tr>
<tr>
<td>acceleration</td>
<td>2</td>
</tr>
<tr>
<td>pressure</td>
<td>3</td>
</tr>
<tr>
<td>altitude</td>
<td>4</td>
</tr>
<tr>
<td>luminance</td>
<td>5</td>
</tr>
<tr>
<td>velocity</td>
<td>6</td>
</tr>
<tr>
<td>signal_strength</td>
<td>7</td>
</tr>
<tr>
<td>battery</td>
<td>8</td>
</tr>
<tr>
<td>heart_rate</td>
<td>9</td>
</tr>
</tbody>
</table>

Figure 3

11.2. CoAP-REGISTRATION Kind-ID

This document introduces one additional data Kind-ID to the "RELOAD Data Kind-ID" Registry:

<table>
<thead>
<tr>
<th>Kind</th>
<th>Kind-ID</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoAP-REGISTRATION</td>
<td>105</td>
<td>RFC-AAAA</td>
</tr>
</tbody>
</table>

This Kind-ID was defined in Section 4.

11.3. CoAP-CACHING Kind-ID

This document introduces one additional data Kind-ID to the "RELOAD Data Kind-ID" Registry:
This Kind-ID was defined in Section 4.

11.4. Access Control Policies

IANA SHALL create a "CoAP Usage for RELOAD Access Control Policy" Registry. Entries in this registry are strings denoting access control policies, as described in Section 8.1. New entries in this registry SHALL be registered via RFC 5226 [RFC5226]. Standards Action. The initial contents of this registry are:

<table>
<thead>
<tr>
<th>Access Policy</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI-NODE-MATCH</td>
<td>RFC-AAAA</td>
</tr>
<tr>
<td>URI-MATCH</td>
<td>RFC-AAAA</td>
</tr>
</tbody>
</table>

This access control policy was described in Section 9.

12. References

12.1. Normative References


12.2. Informative References


Authors’ Addresses

Jaime Jimenez
Ericsson
Hirsalantie 11
Jorvas  02420
Finland

Email: jaime.j.jimenez@ericsson.com

Jose M. Lopez-Vega
University of Granada
CITIC-UGR Periodista Rafael Gomez Montero 2
Granada  18071
Spain

Email: jmlvega@ugr.es

Jouni Maenpaa
Ericsson
Hirsalantie 11
Jorvas  02420
Finland

Email: jouni.maenpaa@ericsson.com
A RELOAD Usage for Distributed Conference Control (DisCo)
draft-knauf-p2psip-disco-04

Abstract

This document defines a RELOAD Usage for Distributed Conference Control (DisCo) with SIP. DisCo preserves conference addressing through a single SIP URI by splitting its semantic of identifier and locator using a new Kind data structure. Conference members are enabled to select conference controllers based on proximity awareness and to recover from failures of individual resource instances. DisCo proposes call delegation to balance the load at focus peers.

Status of this Memo

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

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

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

This Internet-Draft will expire on May 3, 2012.

Copyright Notice

Copyright (c) 2011 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents.
Internet-Draft  DisCo  October 2011

carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents
1. Introduction ............................................. 4
2. Terminology ............................................. 5
3. Overview of DisCo ....................................... 6
   3.1. Reference Scenario ................................ 6
   3.2. Initiating a Distributed Conference ............... 7
   3.3. Joining a Conference ............................... 8
   3.4. Conference State Synchronization .................. 9
   3.5. Call delegation ................................... 10
   3.6. Resilience ....................................... 10
   3.7. Topology Awareness ................................ 10
4. RELOAD Usage for Distributed Conference Control ......... 11
   4.1. Shared Resource DisCo-Registration ................. 11
   4.2. Kind Data Structure ................................ 11
   4.3. Variable Conference Identifier .................... 12
   4.4. Conference Creation ............................... 12
   4.5. Advertising Focus Ability .......................... 13
   4.6. Determining Coordinates ............................ 14
   4.7. Proximity-aware Conference Participation ........... 14
   4.8. Configuration Document Extension ................... 16
5. Conference State Synchronization .......................... 18
   5.1. Event Package Overview ............................. 18
   5.2. <distributed-conference> ......................... 20
   5.3. <version-vector>/<version> ......................... 20
   5.4. <conference-description> ........................... 21
   5.5. <focus> ........................................ 22
      5.5.1. <focus-state> ................................. 23
      5.5.2. <users>/<user> .............................. 23
      5.5.3. <relations>/<relation> ....................... 24
   5.6. Distribution of Change Events ....................... 24
   5.7. Translation to Conference-Info Event Package ....... 25
      5.7.1. <conference-info> ............................ 26
      5.7.2. <conference-description> ...................... 26
      5.7.3. <host-info> ................................. 26
      5.7.4. <conference-state> ........................... 26
      5.7.5. <users>/<user> .............................. 27
      5.7.6. <sidebars-by-ref>/<sidebars-by-value> ....... 27
6. Distributed Conference Control with SIP .................... 28
   6.1. Call delegation .................................... 28
   6.2. Conference Access ................................. 29
1. Introduction

This document describes a RELOAD Usage for distributed conference control (DisCo) in a tightly coupled model with SIP [RFC3261]. The Usage provides self-organizing and scalable signaling that allows RELOAD peers, clients and plain SIP user agents to participate in a managed P2P conference. DisCo defines the following functions:

- A SIP protocol scheme for distributed conference control
- RELOAD Usage and definition of conferencing Kind
- Mechanisms for conference synchronization and call delegation
- Mechanism for proximity-aware routing within a conference
- An XML event package for distributed conferences

In this document, the term distributed conferencing refers to a multiparty conversation in a tightly coupled model in which the point of control (i.e., the focus) is identified by a unique URI, but the focus service is located at many independent entities. Multiple SIP [RFC3261] user agents uniformly control and manage a multiparty session. This document defines a new Usage for RELOAD, including an additional Kind code point with a corresponding data structure that complies with the demands for distributed conferences. The ‘DisCo’ data structure stores the mapping of a single conference URI to multiple conference controllers and thereby separates the conference identifier from focus instantiations.

Authorized controllers of a conference are permitted to register their mapping in the DisCo data structure autonomously. Thus, the DisCo data structure represents a co-managed Resource in RELOAD. To provide trusted and secure access to a co-managed Resource, this document uses the definitions for Shared Resources (ShaRe) [I-D.knauf-p2psip-share].

Delay and jitter are critical issues in multimedia communications. The proposed conferencing scheme supports mechanisms to build an optimized interconnecting graph between conference participants and their responsible conference controllers. Conference members will be enabled to select the closest focus with respect to delay or jitter.

DisCo extends conference control mechanisms to provide a consistent and reliable conferencing environment. Controlling peers maintain a consistent view of the entire conference state. The multiparty system can be re-structured based on call delegation operations.
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 [RFC2119].

We use the terminology and definitions from the RELOAD base draft [I-D.ietf-p2psip-base], the peer-to-peer SIP concepts draft [I-D.ietf-p2psip-concepts], the usage for shared resources draft [I-D.knauf-p2psip-share], and the terminology formed by the framework for conferencing with SIP [RFC4353]. Additionally, the following terms are used:

Coordinate Value: An opaque string that describes a host’s relative position in the network topology.

Focus peer: A RELOAD peer that provides SIP conferencing functions and implements the Usage for distributed conferencing. It is called ‘active’ if already involved in signaling relation to conference participants. Otherwise, if only registered in a distributed conference data structure, it is referred to as a ‘potential’ focus peer.
3. Overview of DisCo

3.1. Reference Scenario

The reference scenario for the Distributed Conference Control (DisCo) is shown in Figure 1. Peers are connected via a RELOAD [I-D.ietf-p2psip-base] instance, in which peers A and B are managing a single multiparty conference. The conference is identified by a unique conference URI, but located at peers A and B fulfilling the role of the focus. The mapping of the conference URI to one or more responsible focus peers is stored in a new RELOAD Resource for distributed conferencing within a data structure denoted as DisCo-Registration. The storing peer SP of the distributed conference resource holds this data.

The focus peers A and B maintain SIP signaling relations to conference participants, which may have different conference protocol capabilities. In this example, peer A is the focus for the RELOAD peer C and the plain SIP user agent E whereas peer B serves as a focus for RELOAD peer D and the RELOAD client F.

RELOAD peers and clients obtain the contact information for the conference from the storing peer. In contrast to this, the user agent E receives the conference URI not by RELOAD mechanisms, but resolves the ID and joins the conference by plain SIP negotiation.

Focus peers maintain a SIP signaling relation among each other used for notification messages that synchronize the conference focus peers’ knowledge about the entire conference state. Additionally, focus peers can transfer calls to each other by a call delegation mechanism.
3.2. Initiating a Distributed Conference

To create a conference the initiating user agent announces itself as a focus for the conference. It stores its own contact information (Node-ID) as a DisCo-Registration Kind (cf. Figure 2) in the RELOAD overlay. The hashed conference URI is used as the Resource-ID. This Resource will later contain the contact IDs of all potential focus peers including optional topological descriptors.
3.3. Joining a Conference

A RELOAD-aware node (cf. Bob in Figure 2) intending to join an existing conference requests the list of potential focus peers stored in the DisCo-Registration under the conference’s Resource-ID. The node selects any of the focus peers (e.g., Alice) and establishes a connection using AppAttach/ICE [RFC5245]. This transport is then used to send an INVITE to the conference applying the chosen focus as the contact. The selection of the focus peer can optionally be based on proximity information if available.

A conference member proposes itself as a focus for subsequent participants by adding its Node-ID to the DisCo-Registration stored under the conference URI in the RELOAD overlay. The decision whether a peer announces as a focus incorporates bandwidth, power, and other constraints, but details are beyond the scope of this document.
3.4. Conference State Synchronization

Each focus of a conference maintains signaling connections to its related participants independently from other conference controllers. This distributed conference design effects that the entire SIP conference state is jointly held by all focus peers. In DisCo, state synchronization is based on a SIP specific event notifications mechanism [RFC3265].

Each focus peer maintains its view of the entire conference state by subscribing to the other focus peers’ XML event package for distributed conferences. This is based on the event package for conference state (cf. [RFC4575]). Details are defined in this document in Section 5. Receivers of event notifications update their
The event notification package for distributed conferences enables focus peers to synchronize the entire conference state. The event package defines additional XML elements and complex types (see Section 8 for more details), which describe the responsibilities of any focus peer in the conference. By providing a global view each focus peer is enabled to perform additional load balancing operations and enhances the robustness against departures of focus peers.

3.5. Call delegation

Call delegation (see Section 6.1) is a feature used to transfer an incoming participation request to another focus peer. It can be applied to prevent overloading of focus peers. Call delegation is realized through SIP REFER requests, which carry signaling and session description information of the caller to be transferred. This feature is achieved transparently for the transferred user agent by using a source routing mechanism at SIP dialog establishment. Descriptions of overload detection are beyond the scope of this document.

3.6. Resilience

A focus peer can decide to leave the conference or may ungracefully fail. In a traditional conferencing scenario, loss of the conference controller or the media distributor would cause a complete failure of the multiparty conversation. Distributed conferencing uses the redundancy provided by multiple focus peers to reconfigure a current multiparty conversation. Participants that lose their entry point to the conference re-invite themselves via the remaining focus peers or will be re-invited by the latter. This option is based on the conference state and call delegation functions.

3.7. Topology Awareness

DisCo supports the optimization of the conference topology in respect of the underlying network using topological descriptors. An extension for the RELOAD XML configuration document is defined in Section 4.8 to support landmarking approaches. Each peer intending to create or participate in a distributed conference MAY determine a topological descriptor that describes its relative position in the network. Focus peers store these coordinate values in an additional data field in the DisCo-Registration data structure. This enables peers joining the conference to select the closest focus with respect to its coordinate values.
4. RELOAD Usage for Distributed Conference Control

4.1. Shared Resource DisCo-Registration

A distributed conference is a cooperative service of several individual devices that use a common identifier. To enable a mapping from one conference identifier to multiple focus peers, this document defines the new Kind data structure DisCo-Registration. The DisCo Kind uses the definitions for Shared Resources [I-D.knauf-p2psip-share] to allow a jointly maintenance by multiple focus peers. Hence, write permission to a DisCo-registration is shared by the conference creator with all authorized focus peers.

DisCo complies with the following requirements for Shared Resources:

Isolated Data Storage: DisCo uses the dictionary data model. Each dictionary key is the Node-ID of the certificate that will be used to sign the stored data

ResourceNameExtension field: A DisCo-Registration can contain the ResourceNameExtension structure an initial field in the Kind data structure. It contains the conference URI of the registered DisCo-record.

4.2. Kind Data Structure

Each DisCo-Registration data structure stores a mapping of a conference identifier to one or multiple focus peers that cooperatively control the conference. Additionally, each DisCo-Registration provides the coordinate value, which indicates the relative network position of the focus peers.

The data structure uses the RELOAD dictionary type. The dictionary key MUST be the Node-ID of the focus peer that is associated with the dictionary entry. This allows a focus peer to update only its own mapping. The DisCo data structure of type DisCoRegistration is constructed as follows:

```c
struct {
    /* This field is optional, see documentation */
    ResourceNameExtension res_name_ext;
    opaque coordinate<0..2^16-1>;
    NodeId node_id;
} DisCoRegistration;
```

The DisCoRegistration structure is composed of the following values:
res_name_ext: This field can contain the conference URI. It meets the requirement for the USER-CHAIN-ACL access policy defined in [I-D.knauf-p2psip-share] to enable variable resource names.

coordinate: This field contains a topological descriptor that indicates the relative position of the peer in the network. To support different algorithms the coordinate field is represented as an opaque string.

node_id: This field contains the Node-ID of the peer storing the DisCoRegistration and is used to contact the peer when utilizing its service as a focus.

4.3. Variable Conference Identifier

DisCo-Registrations can be stored based on a variable Resource Name. However, a conference identifier set by a user MUST follow the requirements for Kinds using variable Resource Names as defined in the ShaRe Usage [I-D.knauf-p2psip-share].

4.4. Conference Creation

The registration of a distributed conference includes the storage of the following two Kinds (see Figure 3).

DisCo-Registration Kind: contains the conference identifier and the optional coordinate value. If used, the coordinate value MAY be determined previously to the conference registration. The Resource Name and hence the Resource-ID is defined by the hash over the desired conference identifier (using the hash algorithm of the overlay).

Access Control List Kind: contains the conference participants that are allowed to register as focus peers for a conference (see [I-D.knauf-p2psip-share]). Its Resource Name/ID is equal to those of the corresponding DisCo-Registration.

Preliminary to storage of DisCo-Registration and Access Control List (ACL) Kinds the conference creator SHOULD perform a RELOAD StatReq to verify that no former conference is present. If neither a DisCo-Registration nor an associated ACL exist, the conference creator stores a DisCo-Registration with a valid conference identifier (see Section 4.3) and an ACL referring to the DisCo-Registration Kind-ID.

If DisCo registrations and ACL Kinds from previous conferences are still existing there are two options. First, if conference creator is aware of the indexes from previous ACL Kinds, it refreshes the root item of this ACL and stores its registration as focus peer as
DisCo-Registration Kind. Second, If the creator is unaware of indexes, it fetches all Access List Kinds to determine the index of the root item.

<table>
<thead>
<tr>
<th>Alice</th>
<th>Peer1</th>
<th>Overlay</th>
<th>PeerN</th>
<th>Storing Peer</th>
</tr>
</thead>
<tbody>
<tr>
<td>StatReq Res:Conf-URI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;-------</td>
<td>&lt;-------</td>
<td>&lt;-------</td>
<td>&lt;-------</td>
<td></td>
</tr>
<tr>
<td>StoreReq</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;-------</td>
<td>&lt;-------</td>
<td>&lt;-------</td>
<td>&lt;-------</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: Initial creation of a Distributed Conference

Optionally the conference initiator (or any active focus) MAY store an additional RELOAD SIP-Registration in the overlay [I-D.ietf-p2psip-sip] or even at a standard SIP registrar [RFC3261] under a URI for which it has write permission. This allows DisCo-unaware or even legacy SIP user agents to participate in the conference. Those registrations SHOULD always point to a currently active focus, who is prepared to accept legacy user agents. The user agent who initially performed the registrations is responsible for updating them appropriately. How authorization has been obtained to perform those registration is out of scope of this document.

The lifetime of a distributed conference is not necessarily limited by the participation time of its creator. As long as the root item of an Access Control List to a DisCo-Registration is not expired, Authorized Peers are allowed to further provide a conferencing service and even store new focus registrations.

4.5. Advertising Focus Ability

All participants of a distributed conference MAY potentially become a focus peer for their conference. This depends on its capacities such as sufficient processing power (CPU, Memory) for the desired media type, the quality of the network connectivity, and the conference policy. Information about network connectivity with respect to NAT or restricted firewalls can be obtained via ICE [RFC5245] connectivity checks. If a peer is behind a NAT, it SHOULD allow for incoming connections via AppAttach/ICE. Peers that can only accept connections with the support of TURN should not act as a focus.
Nevertheless, under special circumstances it might be reasonable to locate a focus peer behind such a NAT (e.g., within a companies network infrastructure).

If a participant is a candidate to become a focus for the conference, it stores its Node-ID and optional coordinate value into the DisCo data structure. An entry in the corresponding ACL [I-D.knauf-p2psip-share] must be present to allow this peer to write the DisCo-Registration resource. By storing the mapping into the data structure a participant becomes a potential focus.

4.6. Determining Coordinates

Each RELOAD peer within the context of a distributed conference MAY be aware of its relative position in the network topology. To construct a topology-aware conference, the DisCo Usage provides the coordinate value within the DisCo-Registration data structure. Focus peers store their relative network position together with the announcement as conference focus. Joining peers that are able to determine their coordinates may then select a focus peer whose relative position is in its vicinity (see Section 4.7).

Some algorithms determine topology information by measuring Round-Trip Times (RTT) towards a set of hosts serving as landmarks (e.g., [landmarks-infocomm02]). To support such algorithms this document describes an extension to the RELOAD XML configuration document that allows to configure the set of landmark hosts peer must use for position estimation (see Section 4.8). Once a focus peer has registered its mapping in the DisCo data structure, it also stores the according coordinates in the same mapping. These <Node-ID,coordinates> vectors are used by peers joining the conference to select the focus peer that is relatively closest to itself.

Because topology-awareness can be obtained by many different approaches a concrete algorithms is out of scope of this document.

4.7. Proximity-aware Conference Participation

The participation procedure to a distributed conference is composed of three operation.

1. Resolution of the conference identifier.
2. Establishment of transport connection.
3. SIP signaling to join a conference.

Figure 4 and the following specifications give a more detailed view
on the joining procedure.

<table>
<thead>
<tr>
<th>Bob</th>
<th>Peer1</th>
<th>Overlay</th>
<th>PeerN</th>
<th>Storing Peer</th>
<th>Alice</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>StatReq Res:Conf-URI</td>
<td>---------&gt;</td>
<td>---------&gt;</td>
<td>---------&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>StatAns</td>
<td>&lt;---------</td>
<td>&lt;---------</td>
<td>&lt;---------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FetchReq Res:Conf-URI Kind:DisCo,ACL</td>
<td>---------&gt;</td>
<td>---------&gt;</td>
<td>---------&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FetchAns</td>
<td>&lt;---------</td>
<td>&lt;---------</td>
<td>&lt;---------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bob calculates Alice as closest Focus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AppAttach application:5060</td>
<td>---------&gt;</td>
<td>---------&gt;</td>
<td>---------&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AppAttach application:5060</td>
<td>&lt;---------</td>
<td>&lt;---------</td>
<td>&lt;---------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICE Checks</td>
<td>&lt;-------------------ICE Checks-----------------------&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INVITE sip:Alice</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200 OK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: Participation of a Distributed Conference

1. The joining peer MAY determine its own coordinate value (if used).

2. The joining peer sends a StatReq message to obtain all indexes of the Access Control List (ACL) Kinds stored.

3. The joining peer sends a FetchReq message for the DisCo and ACL Kind to the Resource-ID of the conference URI. The FetchReq SHOULD NOT include any specific dictionary keys, but SHOULD fetch for those array ranges previously determined the StatReq. With the ACL items, the joining peer is able to verify whether DisCo-Registrations are stored by authorized focus peers (see [I-D.knauf-p2psip-share]).

4. Using the retrieved coordinates values of the DisCo-Registrations, the joining peer MAY calculate which of than
opaque <0..2^16-1> initial field in the Kind data structure focus peers is the relatively closest to itself.

5. The joining peer then establishes a transport using RELOADs AppAttach, respectively, ICE procedures to create a transport to the chosen focus peer.

6. Finally, the established transport is used to create a SIP dialog from the joining peer to the focus peers.

The SIP INVITE request MAY contain the joining peer’s topological descriptor as a URI-parameter called ‘coord’ in the contact-header in base64 encoded form [RFC4648]. An example contact URI is "sip:alice@example.com;coord=PEknbSBhIHRvcG9sb2dpY2FsIGRlc2NyaXB0b3I+". When the called focus is full and needs to delegate the call it uses the contents of the ‘coord’-parameter. It determines the next available focus closest to the calling peer (Section 4.6) using the received descriptor and the other focuses’ descriptors from the conference state synchronization document and delegates the call accordingly (see Section 6.1).

A conference focus MAY allow the joining peer to also become a focus (depending on the conference policy see Section 6.2). Therefore, it stores a new ACL Kind that delegates write permission for the DisCo-Registration to the new participant. It sets the ‘user_name’ field in the ACL Kind to its own user name and sets the ‘to_user’ field to the user name of the joining peer. If no other conference policy is defined, the focus peer MAY set the allow_delegation flag to true. For further details about the trust delegation using the ACL Kind see [I-D.knauf-p2psip-share].

4.8. Configuration Document Extension

This section defines an additional parameter for the <configuration> element that extends the RELOAD XML configuration document. The proposed <landmarks> element allows RELOAD provider to publish a set of accessible and reliable hosts that SHOULD be used if RELOAD peers use landmarking algorithms to determine relative position in the network topology.

The <landmarks> element serves as container for the <landmark-host> sub-elements, each representing a single host that serves as a landmark. The <landmark-host> uses the following attributes:

- address: The IP address (IPv4 or IPv6) of the landmark host.
port: The port on which the landmark host responds for distance estimation.

More than one landmark hosts SHOULD be present in the configuration document.
5. Conference State Synchronization

The global knowledge about signaling and media relations among the conference participants and focus peers defines the global state of a distributed conference. It is composed of the union of every local state at the focus peers. To enable focus peers to provide conference control operations that modify and/or require the global state of a conference, this document defines a mechanism for inter-focus state synchronization. It is based on mutual subscriptions for an Event Package for Distributed Conferences and allows to preserve a coherent knowledge of the global conference state. This XML based event package named 'distributed-conference' MUST be supported by each RELOAD peer that is registered with a DisCo-Registration. Participants of a distributed conference MAY support it. To provide backward compatibility to conference members that do not support the distributed-conference event package, this document defines a translation to the Event Package for Conference State [RFC4575].

5.1. Event Package Overview

The 'distributed-conference' event package is designed to convey information about roles and relations of the conference participants. Conference controllers obtain the global state of the conference and use this information for load balancing or conference restructuring mechanisms in case of a focus failure. Figure Figure 5 gives a general overview of the document hierarchy.
distributed-conference

-- version-vector
   -- version
   -- version

-- conference-description

-- focus
   -- focus-state
      -- user-count
      -- coordinate
      -- maximum-user-count
      -- active
      -- locked
      -- conf-uris
      -- available-media

-- users
   -- user
      -- endpoint
         -- media
         -- call-info

-- relations
   -- relation

-- focus
   -- ...

Figure 5: Overview of the event package for distributed conferences

The document structure is designed to allow concurrent change events at several focus peers. To prevent race conditions each focus peer has exclusive writing permission to the 'focus' sub element that describes itself. It is achieved by a unique mapping from a focus peer to its XML element using the 'Element Keys' mechanisms for partial notification [RFC4575](sections 4.4-5.). A focus peer is only allowed to update or change that <focus> sub element, whose 'entity' Element Key contains its RELOAD user name. This restriction also applies to the child elements of the 'version-vector' element. Each 'version' number belongs to a specific focus peer maintaining the version number.

The local state of a focus peer is described within a 'focus' element. It provides general information about a focus peer and its signaling and media relations to participants and focus peers. The Conference participants are aggregated within 'users' elements, respectively, 'user' sub elements.
General information about the conference as a whole, is provided within a 'conference-description' element. In contrast to the 'focus' and 'version-vector' elements, conference description is not meant for concurrent updating. Instead, it provides static conference descriptions that rarely change during a multiparty session.

More detailed descriptions about the event package and its elements are provided in the following sections. The complete XML schema definition is given in Section 8.

5.2. <distributed-conference>

The <distributed-conference> element is the root of a distributed conference XML document. It uses the following attributes:

entity: This attribute contains the conference URI that identifies the distributed conference. A SIP SUBSCRIBE request addressed to this URI initiates a subscribe/notify relation between participants and their related focus peer.

state: This attribute indicates whether the content of a distributed conference document is a 'full', 'partial' or 'deleted' information. It is in accordance with [RFC4575] to enable the partial notification mechanism.

The <distributed-conference> child elements are <vector-version>, <conference-description> and the <focus> elements. An event notification of state 'full' MUST include all these elements. An event notification of state 'partial' MUST contain at least <version-vector> and its sub elements.

5.3. <version-vector>/<version>

The Event Package for Distributed Conferences uses the <version-vector> and its <version> sub elements to enable a coherent versioning scheme. It is based on vector clocks as defined in [timestamps-acsc88]. Each <version> element contains a unsigned integer that describes the state of a specific focus peer and contains the following attributes:

entity: This attribute contains the user name of the focus peer whose local version number is described by this element.

node-id: This attribute contains the Node-ID of the focus peer.

Whenever the local status of a focus peer changes (e.g. a new participant arrived) the version number of the corresponding
<version> element MUST be incremented by one. Each change in the local state also triggers a new event notification containing the entire <version-vector> and the changes contained in a <focus> element.

The recipient of a change event needs to update its local XML document. If a received <version> number is higher than the local, it updates the <version> element and its associated <focus> element with the retrieved elements. All other elements remain constant.

If the length or contents of the retrieved <version-vector> is different to the local copy it indicates an incoherent knowledge about the entire conference state. If the retrieved <version-vector> contains any unknown focus peers and any local version numbers for the known focus peers is lower, the receiver SHOULD request a 'full' XML notification.

If any local <version> number is retarded more than one, the receiver SHOULD request a 'full' event notification from the sender. The full state notification updates all <focus> elements whose corresponding <version> element is out of date.

5.4. <conference-description>

The <conference-description> element provides general information and links to auxiliary services for the conference. The following subelements provide human-readable text descriptions about the conference:

<display-text>: Contains a short textual description about the conference

<subject>: Contains the subject of the conference

<free-text>: Contains a longer textual description about the conference

<keywords>: Contains a list of keywords that match the conference topic. The XML schema definition and semantic is imported from the 'conference-info' event package [RFC4575].

The <service-uris> sub element enables focus peers to advertise auxiliary services for the conference. The XML schema definition and semantic is imported from the 'conference-info' event package [RFC4575].

The <conference-description> element uses the 'state' Element Key to enable the partial notification mechanism.
5.5. <focus>

Each <focus> element describes a focus peer actively controlling the conference. It provides general information about a focus peer (e.g., display-text, languages, etc.), contains conference specific information about the state of a focus peer (user-count, available media types, etc.) and announces signaling and media information about the maintained participants. Additionally, it describes signaling or media relations to further focus peers.

The <focus> element uses the following attributes:

- **entity**: This attribute contains the user name of the RELOAD peer acting as focus peer. It uniquely identifies the focus peer that is allowed to update or change all sub elements of <focus>. All other focus peers SHOULD NOT update or change sub elements of this <focus> element. A SUBSCRIBE request addressed to the user name initiates a conference state synchronization with the focus peer.

- **Node-ID**: This attribute contains the Node-ID of the peer acting as conference focus.

- **state**: In accordance to [RFC4575], this attribute indicates whether the content of the <focus> element is a 'full', 'partial' or 'deleted' information. A 'partial' notification contains at maximum a single <focus> element.

The following sub elements of <focus> provide general information about a focus peer:

- **<display-text>**: Contains a short text description of the user acting as focus peer.

- **<associated-aors>**: This element contains additional URIs that are associated with this user.

- **<roles>**: This element MAY contain human-readable text descriptions about the roles of the user in the conference.

- **<languages>**: This element contains a list of tokens, each describing a language understood by the user.

The XML schema definition and semantic for <associated-aors>, <roles> and <languages> are imported from the 'conference-info' event package [RFC4575].

Following, a detailed description of the remaining sub elements.
5.5.1.  <focus-state>

The <focus-state> element aggregates a set of conference specific information about the RELOAD user acting as focus peer. The following attribute is defined for the <focus-state> element:

status: This attribute indicates whether the content of the <focus-state> element is a 'full', 'partial' or 'deleted' information.

The <focus-state> element has the following sub elements:

<user-count>: This element contains the number of participants that are connected to the conference via this focus peer at a certain moment.

<coordinate> This element contains the coordinate value Section 4.2 of the focus peer

<maximum-user-count>: This number indicates a threshold of participants a focus peer is able to serve. This value might change during a conference, depending on the focus peer's current load.

<conf-uris>: This element MAY contain other conference URIs in order to access the conference via different signaling means. The XML schema definition and semantic is imported from [RFC4575].

<available-media>: This element is imports the <conference-media-type> type XML scheme definitions from [RFC4575]. It allows a focus peer to list its available media streams.

<active>: This boolean element indicates whether a focus peer is currently active. Conference participation requests or a call delegation request SHOULD succeed.

<locked>: In contrast to <active>, this element indicates that a focus peer is not willing to accept anymore participation or call delegation request.

5.5.2.  <users>/<user>

The <users>, respectively, each <user> element describes a single participant that is maintained by the focus peer described by the parent <focus> element. The <users> element XML schema definition and its semantic is imported from the 'conference-info' event package [RFC4575].
5.5.3.  <relations>/<relation>

The <relations> element serves as container for <relation> elements, each describing a specific connection to another focus peer. The parent element <relations> uses the 'state' attribute to enable the partial notification mechanism. For the <relation> element the following attributes are defined:

entity:  This attribute contains the user name of the remote focus.

node-id  This attribute contains the Node-ID of the remote focus peer.

The content of each <relation> is a comma separated string that describes the tuple <CONNECTION-TYPE:IDENTIFIER>. The CONNECTION-TYPE is a string token describing the type of connection (e.g. media, signaling, etc.) and the IDENTIFIER contains a variable connection identifier. It is a generic method to announce any kind of connection to a remote focus. This specification defines following tuples:

media:<label>:  This tuple identifies a single media stream. The 'label' variable contains the SDP "label" attribute. (see [RFC4574]).

sync:<call-id>:  This tuple indicates a subscription for the Event Package for Distributed Conferences. The 'call-id' variable contains the call-id of the SIP subscription dialog.

5.6.  Distribution of Change Events

Each focus peer in a distributed conference must be able to retrieve all change events from all other focus peers. A simple approach would be to inter-connect each focus with all other focus in a full meshed topology. To avoid a full mesh, this document describes a 'mutual' subscription scheme that constructs a spanning tree topology among the focus peers.

A conference participant that becomes a focus peer MUST send a SIP SUBSCRIBE to request the Event Package for Distributed Conferences to its own focus peer. The subscription request is addressed to user name of the active focus peer. The latter interprets this subscription as a request for conference state synchronization. The corresponding NOTIFY message contains a 'full' distributed-conference state XML document (see section Section 5.1).

The subscribed focus peer in turn subscribes the upcoming focus peer for the distributed conference event package. The corresponding
NOTIFY message carries a ‘partial’ conference state XML document. It contains the received <version-vector> including a new <version> element for itself and a new <focus> element that describes its local state (see Section 5.5).

Resulting by this subscription scheme, each focus peer has at least one subscription to obtain updates for the conference state and is a notifier for change events originated itself. In a incrementally increasing conference, the 1st and 2nd focus peer have a mutual subscription for conference change events. A 3rd focus could have a mutual subscription with the 1st focus, a 4th focus to the 2nd focus and so forth. The result is a spanning tree topology among the focus peers in which each focus peer is a possible root for distribution tree for conference change events.

However, the fact that event notifications need to traverse one or more intermediate focus peers until conference-wide delivery, demands a forwarding mechanism for change events. On receiving a change event, a notified focus validates based on the <version-vector> whether the incoming state event is not a duplicate to previews notifications. If its not a duplicate, the received change event triggers a new event notification at the receiver of the change event. It notifies all its subscribers with excepting the sender of the event notification. In this way, the change event will be ‘flooded’ among the focus peer of a conference.

5.7. Translation to Conference-Info Event Package

The Event Package for Distributed Conferences imports several XML element definitions of the Event Package for Conference State [RFC4575]. This is caused by two reasons. First, the semantic of these elements are fitting the demands to describe the global state of a distributed conference and, second, it facilitates a re-translation to [RFC4575] to enable a backward compatibility to DisCo-unaware clients. Therefore, each focus peer MAY provide a separate [RFC4575] conform event notification service to its connected participants.

The following sections describe the translation to the Event Package for Conference State [RFC4575] by defining translation rules for the root element and its direct sub elements. For a better understanding, the following sections use a notation ci.<ELEMENT> to refer to a sub element of the conference-info element, and disco.<ELEMENT> to refer to an element of the distributed-conference event package.
5.7.1. <conference-info>

The root element of Event Package for Conference State uses the attributes 'entity', 'state' and 'version' and is the counterpart of the <distributed-conference> root element in the DisCo Event Package. The former two attributes 'entity' and 'state' are equal in both root elements and can be seamlessly translated.

According to [RFC4575], the 'version' attribute SHOULD be incremented by one at any time a new notification being sent to a subscriber. Hence, in DisCo the 'version' attributed increments with each change event that originated by focus peer and each reception of a change events of remote focus peer.

5.7.2. <conference-description>

The <conference-description> element exists in both event packages, conference-info and distributed-conference. Thus, the following elements are seamlessly translatable: <keywords>, <display-text>, <subject>, <free-text> and <service-uris>.

The sub elements <conf-uris>, <maximum-user-count> and <available-media> in conference-info have there counterparts below the \focus\focus-state element of the distributed-conference event package. Each describes a local state of a focus peer in the conference. Hence, the intersection of every disco.<conf-uris>, disco.<available-media> and the sum over each disco.<maximum-user-count> element of each disco.<focus> element in distributed-conference, specifies the content of the corresponding conference-info elements.

5.7.3. <host-info>

According to [RFC4575] the ci.<host-info> element contains information about the entity hosting the conference. For participants in a distributed conference, the hosting entity is their focus peer. Thus, the ci.<host-info> element contains information about a focus peer.

5.7.4. <conference-state>

The ci.<conference-state> element allows subscribers obtain information about overall state of a conference. Its sub elements ci.<user-count>, ci.<active> and ci.<locked> are reused as sub elements of \focus\focus-state to describe the local state of a focus peer in a distributed conference. The translation rules from the distributed-conference to the conference-info event package are the following:
<user-count>: The sum over each value of the disco.<user-count> element defines the corresponding ci.<user-count>.

<active>: The boolean ci.<active> element is the logical concatenation over all disco.<active> elements by an OR-operator.

<locked>: The boolean ci.<locked> element is the logical concatenation over all disco.<locked> elements by an AND-operator.

5.7.5. <users>/<user>

The distributed-conference event package imports the definitions of the ci.<users> and ci.<user> elements under a parent disco.<focus> element for each focus peer in a conference. Thus, the aggregation over all disco.<users> elements specifies the content of the corresponding ci.<users> element.

5.7.6. <sidebars-by-ref>/<sidebars-by-value>

In accordance to [RFC4575], if a participant is connected to a sidebar, its responsible focus peer creates a new <user> by referencing to the corresponding sidebar conference.
6. Distributed Conference Control with SIP

Distributed conference control with SIP defined in this document refers to multiparty conversation in a tightly coupled model that is controlled by several independent entities. It enables a resilient conferencing service for P2P scenarios and provides mechanisms for load-balancing among the focus peers. This section describes additional control operations for distributed conferences with SIP.

6.1. Call delegation

Distributed conference control enables load-balancing by a mechanism for call delegation. Call delegations are performed by focus peers that are running out of capacities to serve more participants. Incoming participation requests are then transferred to other established focus peer or conference participants that are registered as potential focus peers in the overlay. Call delegations use SIP REFER requests [RFC3515] that contain additional session information and are achieved transparently to the transferred party.

A focus peer initiates a call delegation by sending SIP REFER request containing the URI of the participant in the Refer-To header field. Additionally, the focus peer appends the following parameter to the URI of the participant:

- **call-id**: Contains the call-ID of the initial SIP dialog between the referred participant and the referring focus peer.

- **sess-id**: Contains the 'session identifier' value of the original SDP 'o=' field of the original offer/answer process between referred participant and referring focus peer.

If the recipient accepts the REFER request, it generates a re-INVITE towards the referred party and sets the SIP call-id header and the SDP 'session-identifier' field in the SDP offer, according to the URI parameter values of the initial REFER request. The From header field and contact header are set to the conference URI with setting the 'isfocus' tag to contact header. This identifies the peer as a focus to the conference and identifies this re-INVITE as a request of the SIP dialog between the party and the conference. To ensure that further signaling messages will be routed correctly, the new focus adds a Record-Route header field that contains its contact information (URI, IP-address,..).

An example call flow for call delegation is shown in Figure 6.
Delegating a participant to remote focus

(1) REFER refer-to:<uri>?call-id=123&sess-id=456
(2) 200 OK
(3) Notify: pending
(4) 200 OK

Remote focus adds RR-header that carries its URI

(5) INVITE sip:<uri> record-route:<rem.focus>
(6) 200 OK
(7) ACK

(8) Notify: active
(9) 200 OK

Figure 6: Delegating a participant with SIP REFER

Note, subscriptions for the event packages 'distributed-conference' and 'conference-info' are in scope of a specific focus peer and its connected participants. Hence, after a successful call delegation, the referring focus peer SHOULD terminate any subscription to the referred participant. The notifier SHOULD include a reason parameter "deactivated" to indicate a migration of the subscription as defined in [RFC3265]. The new SUBSCRIBE request by the party MUST be sent via the SIP dialog to the conference.

6.2. Conference Access

A conference policy defines who is allowed to participate in a multimedia conference. In many cases, a group conversation can be an open discussion free to participate, while in other occasions a closed privacy of a multiparty session is demanded. In distributed conferences, it is also an issue which of the conference participants is allowed to become a controller of the multiparty session.
Thus it must be decided whether:

- A peer is allowed to participate in a conference
- A peer is allowed to become a focus of the conference

Standard SIP authentication mechanisms can be used to authenticate and accordingly authorize joining participants.

6.3. Media Negotiation and Distribution

This section describes a basic scheme for media negotiation and distribution, which is done in an ad-hoc fashion. Each focus peer forwards all media streams it receives from the conference to all connected peers it is responsible for and similarly all streams from its peers to its responsible focus. This results in the media stream naturally following the SIP signalling paths. Implementations MAY choose to use a more sophisticated scheme, e.g. employing cross connections between different sub-trees of the conference, but MUST take measures to prevent loops in media routing.

When a new peer has been attached to a focus, new media streams may be available to the focus, which need to be forwarded to the conference. To accomplish this, the new media streams need to be signalled to the other participants. This is usually done by sending a SIP re-INVITE and modifying the media sessions, adding the new streams to the SDP. This renegotiation can be costly since it needs to be propagated through the whole conference. Also, distributing all media streams separately to all participants can be quite bandwidth intensive. Both problems can partially be mitigated by focus peers performing mixing of media streams, thus trading bandwidth and signalling overheads for computational load on focus peers.

6.3.1. Offer/Answer

A peer joining a conference negotiates media types and media parameters with its designated focus using the standard SDP offer/answer protocol [RFC3264]. The focus SHOULD offer all existing media streams that it receives from the conference.

A new participant does not necessarily know about all media streams present in a conference beforehand, and thus some of the media streams might not be included in the initial SDP offer sent by the joining peer. An SDP answer sent by the corresponding focus MUST NOT contain any media streams not matching an offer (cf. [RFC3264] Section 6). A joining peer which is aware of the distributed nature of the conference, SHOULD NOT send an SDP offer in the initial INVITE message, but instead send an empty INVITE to which the focus replies...
with an OK, containing the offer. This prevents the need for a second offer/answer-dialog to modify the session. But for compatibility the normal behavior with the INVITE containing the offer MUST be supported.

For new media streams (e.g. those sent by the new participant) the focus SHOULD only offer media types and codecs which are already used in the conference and which will probably be accepted when forwarded to neighboring peers, unless the focus is prepared to do transcoding and/or mixing of the received streams.

A focus has two options when distributing media streams from a new participant to the conference. The focus can either mix the new streams into his own, thus averting the need to modify the already established media sessions with neighboring peers or in case the focus is not willing or able to do mixing of the media streams, he SHOULD modify the media sessions with all attached peers by sending a re-INVITE, adding the new media streams coming from the newly joined participant to the SDP. This SHOULD subsequently be done by all other focus peers upon receiving the new stream, resulting in the stream being distributed across the conference.

6.4. Restructuring a Conference

Distributed conference control provides the possibility to delegate calls to remote focus peers. This feature is used to restructure a conference in case of departure of a focus peer. Following, this section presents restructuring schemes for graceful and unexpected leaves of conference focus peers.

6.4.1. On Graceful Leave

In a graceful case, the leaving focus peer (LP) accomplishes the following procedure:

o LP deletes its mapping in the DisCo-Registration by storing the "non-existing" value as described in the RELOAD base document [I-D.ietf-p2psip-base]. Afterwards, it fetches the lastest version of the DisCo-Registration to obtain all potential focus peers.

o LP calculates for all its participants the closest focus among all active and potential focus peer using the algorithm described in Section 4.6. LP then delegates all participants to those focus peers.

o LP then announces its leave by sending a NOTIFY to all its subscribers for the extended conference event package, setting its <focus> state to 'deleted'. Thereafter, it ends its own SIP
conference dialog by sending by to its related focus peer.

Since the state synchronization topology in a distributed conference is commonly arranged in a spanning tree, a leave of a focus peer effects a gab in the tree structure. Those focus peers which had the leaving focus peer as their parent, are supposed to reconnect to the synchronization graph by subscribing the parent focus of the leaving peer.

6.4.2. On Unexpected Leave

If an unexpected leave is detected by a participant (e.g. missing signaling and/or media packets) it MUST repeat the joining procedure as described in Section 4.7.
7. DisCo Kind Definition

This section formally defines the DisCo kind.

Name
DISCO-REGISTRATION

Kind IDs
The Resource name DISCO-REGISTRATION Kind-ID is the AoR of the conference. The data stored is the DisCoRegistrationData, that contains the Node-ID of a peer acting as a focus for the conference and optionally a coordinates value describing a peer’s relative network position.

Data Model
The data model for the DISCO-REGISTRATION Kind-ID is dictionary. The dictionary key is the Node-ID of the peer action as focus.

Access Control
USER-CHAIN-ACL

The data stored for the Kind-ID DISCO-REGISTRATION is of type DisCoRegistration. It contains a "coordinates" value, that describes the peers relative network position and the Node-ID of the registered focus peer.
8. XML Schema

The XML schema for the event package for distributed conferences is:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:ci="urn:ietf:params:xml:ns:conference-info"
    xmlns="urn:ietf:params:xml:ns:distributed-conference"
    targetNamespace="urn:ietf:params:xml:ns:distributed-conference"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
  <!--
      This imports the definitions in conference-info
  -->
  <xs:import namespace="urn:ietf:params:xml:ns:conference-info"
      schemaLocation="http://www.iana.org/assignments/xml-registry/
                     schema/conference-info.xsd"/>
      schemaLocation="http://www.w3.org/2001/03/xml.xsd"/>
  <!--
      A DISTRIBUTED CONFERENCE ELEMENT
  -->
  <xs:element name="distributed-conference"
      type="distributed-conference-type"/>
  <!--
      DISTRIBUTED CONFERENCE TYPE
  -->
  <xs:complexType name="distributed-conference-type">
    <xs:sequence>
      <xs:element name="version-vector"
          type="version-vector-type" minOccurs="1"/>
      <xs:element name="conference-description"
          type="conference-description-type" minOccurs="0" maxOccurs="1"/>
      <xs:element name="focus"
          type="focus-type" minOccurs="0" maxOccurs="unbounded"/>
      <xs:any namespace="##other" processContents="lax"/>
    </xs:sequence>
    <xs:attribute name="state" type="ci:state-type"/>
    <xs:attribute name="entity" type="xs:anyURI"/>
    <xs:anyAttribute namespace="##other" processContents="lax"/>
  </xs:complexType>
  <!--
      VERSION VECTOR TYPE
  -->
  <xs:complexType name="version-vector-type">
```

<xs:sequence>
  <xs:element name="version"
    type="version-type"
    minOccurs="1"
    maxOccurs="unbounded"/>
  <xs:any namespace="##other" processContents="lax"/>
</xs:sequence>

<!-- CONFERENCE DESCRIPTION TYPE -->
<xs:complexType name="conference-description-type">
  <xs:sequence>
    <xs:element name="display-text" type="xs:string" minOccurs="0"/>
    <xs:element name="subject" type="xs:string" minOccurs="0"/>
    <xs:element name="free" type="xs:string" minOccurs="0"/>
    <xs:element name="keywords" type="ci:keywords-type" minOccurs="0"/>
    <xs:element name="service-uris" type="ci:uris-type" minOccurs="0"/>
    <xs:any namespace="##other" processContents="lax"/>
  </xs:sequence>
  <xs:attribute name="state" type="ci:state-type"/>
  <xs:anyAttribute namespace="##other" processContents="lax"/>
</xs:complexType>

<!-- FOCUS TYPE -->
<xs:complexType name="focus-type">
  <xs:sequence>
    <xs:element name="display-text" type="xs:string" minOccurs="0"/>
    <xs:element name="associated-aors" type="ci:uris-type" minOccurs="0"/>
    <xs:element name="roles" type="ci:user-roles-type" minOccurs="0"/>
    <xs:element name="languages" type="ci:user-languages-type" minOccurs="0"/>
    <xs:element name="focus-state" type="focus-state-type" minOccurs="0"/>
    <xs:element name="users" type="ci:users-type" minOccurs="0"/>
    <xs:element name="relations" type="relations-type" minOccurs="0"/>
    <xs:any namespace="##other" processContents="lax"/>
  </xs:sequence>
</xs:complexType>
<xs:attribute name="entity" type="xs:anyURI"/>
<xs:attribute name="node-id" type="xs:string"/>
<xs:attribute name="state" type="ci:state-type"/>
<xs:anyAttribute namespace="##other" processContents="lax"/>
</xs:complexType>

<!-- VERSION TYPE -->
<xs:complexType name="version-type">
  <xs:simpleContent>
    <xs:extension base="xs:unsignedInt">
      <xs:attribute name="entity" type="xs:anyURI"/>
      <xs:attribute name="node-id" type="xs:string"/>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<!-- FOCUS STATE TYPE -->
<xs:complexType name="focus-state-type">
  <xs:sequence>
    <xs:element name="user-count" type="xs:unsignedInt" minOccurs="0"/>
    <xs:element name="coordinate" type="xs:string" minOccurs="0"/>
    <xs:element name="maximal-user-count" type="xs:unsignedInt" minOccurs="0"/>
    <xs:element name="conf-uris" type="ci:uris-type" minOccurs="0"/>
    <xs:element name="available-media" type="ci:conference-media-type" minOccurs="0"/>
    <xs:element name="active" type="xs:boolean" minOccurs="0"/>
    <xs:element name="locked" type="xs:boolean" minOccurs="0"/>
    <xs:any namespace="##other" processContents="lax"/>
  </xs:sequence>
  <xs:attribute name="state" type="ci:state-type"/>
  <xs:anyAttribute namespace="##other" processContents="lax"/>
</xs:complexType>

<!-- RELATIONS TYPE -->
<xs:complexType name="relations-type">
  <xs:sequence>
    <xs:element name="relation" type="relation-type" minOccurs="0" maxOccurs="unbounded"/>
    <xs:any namespace="##other" processContents="lax"/>
  </xs:sequence>
</xs:complexType>
</xs:sequence>
<xs:attribute name="state" type="ci:state-type"/>
<xs:anyAttribute namespace="##other" processContents="lax"/>
</xs:complexType>
<!--
RELATION TYPE
-->
<xs:complexType name="relation-type">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="entity" type="xs:anyURI"/>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
</xs:schema>
9. Relax NG Grammar

The grammar for the Landmark configuration document extension is:

```xml
<!--
  LANDMARKS ELEMENT
-->
parameter &= element landmarks {
  attribute version { xsd:int }
  <!--
    LANDMARK-Host ELEMENT
  -->
  element landmark-host {
    attribute address { xsd:string },
    attribute port { xsd:int }
  }*
}
```
10. Security Considerations

10.1. Trust Aspects

TODO
11. IANA Considerations

TODO: register Kind-ID code point at the IANA
12. Acknowledgments

This work was stimulated by fruitful discussions in the P2PSIP working group and SAM research group. We would like to thank all active members for constructive thoughts and feedback. In particular, the authors would like to thank (in alphabetical order) David Bryan, Toerless Eckert, Lothar Grimm, Cullen Jennings, Peter Musgrave, Joerg Ott, Peter Pogrzeba, Brian Rosen, and Jan Seedorf.
13. References

13.1. Normative References

[I-D.ietf-p2psip-base]

[I-D.knauf-p2psip-share]


13.2. Informative References

[I-D.ietf-p2psip-concepts]

[I-D.ietf-p2psip-sip]


[landmarks-infocomm02]

[timestamps-acsc88]
Appendix A. Change Log

The following changes have been made from version draft-knauf-p2psip-disco-02.

1. Adapted mechanisms for storing DisCo-Registrations to new requirements of Shared Resources draft [I-D.knauf-p2psip-share]

The following changes have been made from version draft-knauf-p2psip-disco-02.

1. DisCo-Registration uses now only the USER-CHAIN-ACL access control policy.

2. Adapted mechanisms for storing DisCo-Registrations to new requirements of Shared Resources draft [I-D.knauf-p2psip-share]

The following changes have been made from version draft-knauf-p2psip-disco-01.

1. The conference registration is now based on the Shared Resources draft [I-D.knauf-p2psip-share]:
   * DisCo-Registration Kind now meets the requirements for ShaRe.
   * Conference creation procedure now uses the ShaRe Access List.
   * Replaced USER-CHAIN-MATCH access policy for DisCo-Registration. Now uses USER-CHAIN-ACL or USER-PATTERN-MATCH.

2. Allow focus peers behind NAT

3. Added a 'node-id' attribute to the event package XML <version> element.

4. Added a 'node-id' attribute to the event package XML <focus> element.

5. Added a 'coordinate' child element to the event package XML <focus> element.

6. Corrected typos/wording

The following changes have been made from version draft-knauf-p2psip-disco-00.

1. Updated references.
2. Corrected typos.

3. New Section: Conference State Synchronization

4. XML Event Package for Distributed Conferences

5. New mechanism for generating chained conference certificates

6. Allow shared writing of resources using Access Control Policy USER-CHAIN-MATCH

7. Media Negotiation mechanism in a distributed conference

8. New Section: Distributed Conference Control with SIP
Authors’ Addresses

Alexander Knauf
HAW Hamburg
Berliner Tor 7
Hamburg  D-20099
Germany

Phone: +4940428758067
Email: alexander.knauf@haw-hamburg.de
URI:   http://inet.cpt.haw-hamburg.de/members/knauf

Gabriel Hege
HAW Hamburg
Berliner Tor 7
Hamburg  D-20099
Germany

Phone: +4940428758067
Email: hege@fhtw-berlin.de
URI:   http://inet.cpt.haw-hamburg.de/members/hege

Thomas C. Schmidt
HAW Hamburg
Berliner Tor 7
Hamburg  D-20099
Germany

Email: schmidt@informatik.haw-hamburg.de
URI:   http://inet.cpt.haw-hamburg.de/members/schmidt

Matthias Waehlisch
link-lab & FU Berlin
Hoenower Str. 35
Berlin  D-10318
Germany

Email: mw@link-lab.net
URI:   http://www.inf.fu-berlin.de/~waehl
A Usage for Shared Resources in RELOAD (ShaRe)
draft-knauf-p2psip-share-02

Abstract

This document defines a RELOAD Usage for managing shared write access
to RELOAD Resources. Shared Resources in RELOAD (ShaRe) form a basic
primitive for enabling various coordination and notification schemes
among distributed peers. Access in ShaRe is controlled by a
hierarchical trust delegation scheme maintained within an access
list. A new USER-CHAIN-ACL access policy allows authorized peers to
write a Shared Resource without owning its corresponding certificate.
This specification also adds mechanisms to store Resources with a
variable name which is useful whenever peer-independent rendezvous
processes are required.

Status of this Memo

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

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

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

This Internet-Draft will expire on April 26, 2012.

Copyright Notice

Copyright (c) 2011 IETF Trust and the persons identified as the
document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal
Table of Contents

1. Introduction ........................................ 3
2. Terminology ...................................... 4
3. Shared Resources in RELOAD .......................... 5
   3.1. Mechanisms for Isolating Stored Data .............. 6
4. Access Control List Definition ...................... 7
   4.1. Overview ..................................... 7
   4.2. Data Structure ................................ 8
5. Extension for Variable Resource Names ............... 10
   5.1. Overview ..................................... 10
   5.2. Data Structure ................................ 10
   5.3. Overlay Configuration Document Extension ........ 11
6. Access Control to Shared Resources .................. 13
   6.1. Granting Write Access .......................... 13
   6.2. Revoking Write Access ......................... 14
   6.3. Validating Write Access through an ACL .......... 14
   6.4. Operations of Storing Peers .................... 15
   6.5. Operations of Accessing Peers .................. 15
   6.6. USER-CHAIN-ACL Access Policy ................... 15
7. ACL Kind Definition .................................. 17
8. Security Considerations ............................. 18
   8.1. Resource Exhaustion ............................. 18
   8.2. Malicious or Misbehaving Storing Peer ............ 18
   8.3. Privacy Issues ................................ 18
9. IANA Considerations .................................. 19
10. Acknowledgments .................................... 20
11. References ........................................ 21
    11.1. Normative References .......................... 21
    11.2. Informative References ....................... 21
Appendix A. Change Log .................................. 22
Authors’ Addresses .................................... 23
1. Introduction

This document defines a RELOAD Usage for managing shared write access to RELOAD Resources and a mechanism to store Resources with a variable name. The Usage for Shared Resources in RELOAD (ShaRe) enables overlay users to share their exclusive write access to specific Resource/Kind pairs with others. Shared Resources form a basic primitive for enabling various coordination and notification schemes among distributed peers. Write permission is controlled by an Access Control List (ACL) Kind that maintains a chain of Authorized Peers for a particular Shared Resource. A newly defined USER-CHAIN-ACL access control policy enables shared write access in RELOAD.

The Usage for Shared Resources in RELOAD is designed for jointly coordinated group applications among distributed peers (c.f. [I-D.knauf-p2psip-disco]). Of particular interest are rendezvous processes, where a single identifier is linked to multiple, dynamic instances of a distributed cooperative service. Shared write access is based on a trust delegation mechanism. It transfers the authorization to write a specific Kind data by storing logical Access Control Lists. An ACL contains the ID of the Kind to be shared and contains trust delegations from one authorized to another (previously unauthorized) user.

Shared write access extends the RELOAD security model, which is based on the restriction that peers are only allowed to write resources at a small set of well defined locations (Resource-IDs) in the overlay. Using the standard access control rules in RELOAD, these locations are bound to the username or Node-ID in the peer’s certificate. This document extends the base policies to enable a controlled write access for multiple users to a common Resource Id.

Additionally, this specification defines an optional mechanism to store Resources with a variable Resource Name. It enables the storage of Resources whose name complies to a specific pattern. Definition of the pattern is arbitrary, but must contain the username of the Resource creator.
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 [RFC2119].

This document uses the terminology and definitions from the RELOAD base [I-D.ietf-p2psip-base] and the peer-to-peer SIP concepts draft [I-D.ietf-p2psip-concepts]. Additionally, the following terms are used:

Shared Resource: The term Shared Resource in this document defines a RELOAD Resource with its associated Kinds, that can be written or overwritten by multiple RELOAD users following the specifications in this document.

Access Control List: The term Access Control List in this document defines a logical list of RELOAD users allowed to write a specific RELOAD Resource/Kind pair by following the specifications in this document. The list items are stored as Access Control List Kinds that map trust delegations from user A to user B, where A is allowed to write a Shared Resource and the Access Control List, while B is a user that obtains write access to specified Kinds from A.

Resource Owner: The term Resource Owner in this document defines a RELOAD peer that initially stored a Resource to be shared. The Resource Owner possesses the RELOAD certificate that grants write access to a specific Resource/Kind pair using the RELOAD certificate-based access control policies.

Authorized Peer: The term Authorized Peer in this document defines a RELOAD peer that was granted write access to a Shared Resource by permission of the Resource Owner or another Authorized Peer.
3. Shared Resources in RELOAD

A RELOAD user that owns a certificate for writing at a specific overlay location can maintain one or more RELOAD Kinds that are designated for a non-exclusive write access shared with other RELOAD users. The mechanism to share those Resource/Kind pairs with a group of users consists of two basic steps.

1. Storage of the Resource/Kind pairs to be shared.

2. Storage of an Access Control List (ACL) associated with those Kinds.

ACLs are created by the Resource Owner and contain ACL items, each delegating the permission of writing the shared Kind to a specific user called Authorized Peer. For each shared Kind data, its Resource owner stores a root item that initiates an Access Control List. Trust delegation to the Authorized Peer can include the right to further delegate the write permission, enabling a tree of trust delegations with the Resource Owner as trust anchor at its root.

The Resource/Kind pair to be shared can be any RELOAD Kind that complies to the following specifications:

Isolated Data Storage: To prevent concurrent writing from race conditions, each data item stored within a Shared Resource SHALL be exclusively maintained by the RELOAD user who created it. Hence, Usages that allow the storage of Shared Resources are REQUIRED to use either the array or dictionary data model and apply additional mechanisms for isolating data as described in Section 3.1.

Access Control Policy: To ensure write access to Shared Resource by Authorized Peers, each Usage MUST use the USER-CHAIN-ACL access policy as described in Section 6.6.

Resource Name Extension: To enable Shared Resources to be stored using a variable resource name, this document defines an optional ResourceNameExtension structure. It contains the Resource Name of the Kind data to be stored and allows any receiver of a shared data to validate whether the Resource Name hashes to the Resource-ID. The ResourceNameExtension is made optional by configuration. The ResourceNameExtension field is only present in the Kind data structure when configured in the corresponding kind-block of the overlay configuration document (for more details see Section 5.3). If the configuration allows variable resource names, a Kind using the USER-CHAIN-ACL policy MUST use the ResourceNameExtension as initial field within the Kind data.
structure definition. Otherwise the Kind data structure does not contain the ResourceNameExtension structure.

3.1. Mechanisms for Isolating Stored Data

This Section defines mechanisms to avoid race conditions while concurrently writing an array or dictionary of a Shared Resource.

If a dictionary is used in the Shared Resource, the dictionary key MUST be the Node-ID of the certificate that will be used to sign the stored data. Thus data access is bound to the unique ID-holder.

If the data model of the Shared Resource is an array, the following algorithm will generate an array index that avoids collisions.

1. Obtain the Node-ID of the certificate that will be used to sign the stored data

2. Take the least significant 24 bits of that Node-ID

3. Concatenate an 8 bit long short individual index value to those 24 bit of the Node-ID

The resulting 32 bits long integer MUST be used as the index for storing an array entry in a Shared Resource. The 8 bit individual index can be incremented individually for further array entries and allows for 256 distinct entries per Peer.

The mechanism to create the array index is related to the pseudo-random algorithm to generate an SSRC identifier in RTP, see Section 8.1 in [RFC3550] for calculating a collision probability.
4. Access Control List Definition

4.1. Overview

An Access Control List (ACL) is a (self-managed) shared resource that contains a list of AccessControlListItem structures as defined in Section 4.2. Each entry delegates write access for a specific Kind data to a single RELOAD user. An ACL enables the RELOAD user who is authorized to write a specific Resource-ID to delegate his exclusive write access to a specific Kind to further users of a RELOAD instance. Each Access Control List data structure therefore carries the information about who obtains write access, the Kind-ID of the Resource to be shared, and whether delegation includes write access to the ACL itself. The latter condition grants the right to delegate write access further for the Authorized Peer. Access Control Lists are stored at the same overlay location as the Shared Resource and use the RELOAD array data model. They are initially created by the Resource Owner.

Figure 1 shows an example of an Access Control List. We omit the res_name_ext field to simplify illustration. The array entry at index 0x123abc001 displays the initial creation of an ACL for a Shared Resource of Kind-ID 1234 at the same Resource-ID. It represents the root item of the trust delegation tree for this shared RELOAD Kind. The root entry MUST contain the username of the Resource owner in the "to_user" field and can only be written by the owner of the public key certificate associated with this Resource-ID. The allow_delegation (ad) flag for a root ACL item is set to 1 by default. The array index is generated by using the mechanism for isolating stored data as described in Section 3.1. Hence, the most significant 24 bits of the array index (0x123abc) are the least significant 24 bits of the Node-ID of the Resource Owner.

The array item at index 0x123abc002 represents the first trust delegation to an Authorized Peer that is thus permitted to write to the Shared Resource of Kind-ID 1234. Additionally, the Authorized peer Alice is also granted (limited) write access to the ACL as indicated by the allow_delegation flag (ad) set to 1. This configuration authorizes Alice to store further trust delegations to the Shared Resource, i.e., add items to the ACL. On the contrary, index 0x456def001 illustrates trust delegation for Kind-ID 1234, in which the Authorized Peer Bob is not allowed to grant access to further peers (ad = 0). Each Authorized Peer signs its ACL items with its own private key, which makes the item ownership transparent.

To manage Shared Resource access of multiple Kinds at a single location, the Resource Owner can create new ACL entries that refer to another Kind-ID as shown in array entry index 0x123abc003. Note that
overwriting existing items in an Access Control List that reference a
different Kind-ID revokes all trust delegations in the corresponding
subtree (see Section 6.2). Authorized Peers are only enabled to
overwrite existing ACL item they own. The Resource Owner is allowed
to overwrite any existing ACL item, but should be aware of its
consequences.

+------------------------------------------------------+
|                Access Control List                   |
|-----------+------------------------------+-----------|
|  #Index   |       Array Entries          | signed by |
|-----------+------------------------------+-----------|
| 123abc001 | to_user:Owner Kind:1234 ad:1 |   Owner   |
| 123abc002 | to_user:Alice Kind:1234 ad:1 |   Owner   |
| 123abc003 | to_user:Owner Kind:4321 ad:1 |   Owner   |
| 123abc004 | to_user:Carol Kind:4321 ad:0 |   Owner   |
| ...       |       ...                     |   ...     |
| 456def001 | to_user:Bob   Kind:1234 ad:0 |   Alice   |
| ...       |       ...                     |   ...     |
+-----------+------------------------------+-----------+

Figure 1: Simplified example of an Access Control including entries
for two different Kind-IDs and varying delegation (ad) configurations

Implementations of ShaRe should be aware that the trust delegation in
an Access Control List need not be loop free. Self-contained
circular trust delegation from A to B and B to A are syntactically
possible, even though not very meaningful.

4.2. Data Structure

The Kind data structure for the Access Control List is defined as
follows:

struct {
   /* res_name_ext is optional, see documentation */
   ResourceNameExtension  res_name_ext;
   opaque                 to_user<0..2^16-1>;
   KindId                 kind;
   Boolean                allow_delegation;
} AccessControlListItem;
The AccessControlListItem structure is composed of:

res_name_ext: This optional field contains the Resource Name of a ResourceNameExtension (see Section 5.2) to be used by a Shared Resource with variable resource name. This name serves the storing peer for validating, whether a variable resources name matches one of the predefined naming pattern from the configuration document for this Kind. The presence of this field is bound to a variable resource name element in the corresponding kind-block of the configuration document whose "enable" attribute is set to true (see Section 5.3). Otherwise, if the "enable" attribute is false, the res_name_ext field SHALL NOT be present in the Kind data structure.

to_user: This field contains the username of the RELOAD peer that obtains write permission to the Shared Resource.

kind: This field contains the Kind-ID of the Shared Resource.

allow_delegation: If true, this Boolean flag indicates that the Authorized Peer in the 'to_user' field is allowed to add additional entries to the ACL for the specified Kind-ID.
5. Extension for Variable Resource Names

5.1. Overview

In certain use cases such as conferencing (c.f. [I-D.knauf-p2psip-disco]) it is desirable to increase the flexibility of a peer in using Resource Names beyond those defined by the username or Node-ID fields in its certificate. For this purpose, this document presents the concept for variable Resource Names that enables providers of RELOAD instances to define relaxed naming schemes for overlay Resources.

Each RELOAD node uses a certificate to identify itself using its username (or Node-ID) while storing data under a specific Resource-ID. The specifications in this document scheme adhere to this paradigm, but enable a RELOAD peer to store values of Resource Names that are derived from the username in its certificate. This is done by using a Resource Name with a variable substring that still matches the username in the certificate using a pattern defined in the overlay configuration document. Thus despite being variable, an allowable Resource Name remains tied to the Owner’s certificate. A sample pattern might be formed as follows.

Example Pattern:
\*.*-conf-\$USER@$DOMAIN

When defining the pattern, care must be taken to avoid conflicts arising from two usernames of which one is a substring of the other. In such cases, the holder of the shorter name could threaten to block the resources of the longer-named peer by choosing the variable part of a Resource Name to contain the entire longer username. This problem can easily be mitigated by delimiting the variable part of the pattern from the username part by some fixed string, that by convention is not part of a username (e.g. the "-conf-" in the above Example).

5.2. Data Structure

This section defines the optional ResourceNameExtension structure for every Kind that uses the USER-CHAIN-ACL access control policy.
enum { pattern (1),
      (255)} ResourceNameType;

struct {
    ResourceNameType type;
    uint16 length;
    select(type) {
        case pattern:
            opaque resource_name<0..2^16-1>;
            /* Types can be extended */
    }
} ResourceNameExtension

The content of the ResourceNameExtension consist of

length: This field contains the length of the remaining data
structure. It is only used to allow for further extensions to
this data structure.

The content of the rest of the data structure depends of the
ResourceNameType. Currently, the only defined type is "pattern".

If the type is "pattern", then the following data structure contains
an opaque <0..2^16-1> field containing the Resource Name of the Kind
being stored. The type "pattern" further indicates that the Resource
Name MUST match to one of the variable resource name pattern defined
for this Kind in the configuration document.

The ResourceNameType enum and the ResourceNameExtension structure can
be extended by further Usages to define other naming schemes.

5.3. Overlay Configuration Document Extension

This section extends the overlay configuration document by defining
new elements for patterns relating resource names to user names.

The <variable-resource-names> element serves as a container for one
or multiple <pattern> sub-elements. It is an additional parameter
within the kind block and has a boolean "enable" attribute that
indicates, if true, that the overlay provider allows variable
resource names for this Kind. The default value of the "enable"
attribute is "false". In the absence of a <variable-resource-names>
element for a Kind using the USER-CHAIN-ACL access policy (see
Section 6.6), implementors SHOULD assume this default value.

A <pattern> element MUST be present if the "enabled" attribute of its
parent element is set to true. Each <pattern> element defines a
pattern for constructing extended resource names for a single Kind. It is of type xsd:string and interpreted as a regular expression. In this regular expression, $USER and $DOMAIN are used as variables for the corresponding parts of the string in the certificate username field (with $USER preceding and $DOMAIN succeeding the '@'). Both variables MUST be present in any given pattern definition. If no pattern is defined for a Kind or the "enabled" attribute is false, allowable Resource Names are restricted to the username of the signer for Shared Resource.

The Relax NG Grammar for the Variable Resource Names Extension reads:

```xml
<!-- VARIABLE RESOURCE URN SUB-NAMESPACE -->

<!-- VARIABLE RESOURCE NAMES ELEMENT -->
kind-parameter &= element share:variable-resource-names {
    attribute enable { xsd:boolean }

    <!-- PATTERN ELEMENT -->
    element pattern { xsd:string }*
}
```
6. Access Control to Shared Resources

6.1. Granting Write Access

Write access to a Kind that is intended to be shared with other RELOAD users can solely be issued by the Resource Owner. A Resource Owner can share RELOAD Kinds by using the following procedure.

- The Resource Owner stores an ACL root item at the Resource-ID of the Shared Resource. The root item contains the resource name extension field (see Section 5.2), the username of the Resource Owner and Kind-ID of the Shared Resource. The allow_delegation flag is set to 1. The index of array data structure MUST be generated as described in Section 3.1.

- Further ACL items for this Kind-ID stored by the Resource Owner will delegate write access to Authorized Peers. These ACL items contain the same resource name extension field, the username of the Authorized Peer and the Kind-ID of the Shared Resource. Optionally, the Resource Owner sets the "ad" flag to 1 (the default equals 0) to enable the Authorized Peer to further delegate write access. Each succeeding ACL item created by the Resource Owner can be stored in the numerical order of the array index starting with the index of the root item incremented by one.

An Authorized Peer with delegation allowance ("ad"=1) can extend the access to an existing Shared Resource as follows.

- An Authorized Peer can store additional ACL items at the Resource-ID of the Shared Resource. These ACL items contain the resource name extension field, the username of the newly Authorized Peer, and the Kind-ID of the Shared Resource. Optionally, the "ad" flag is set to 1 for allowing the Authorized Peer to further delegate write access. The array index MUST be generated as described in Section 3.1. Each succeeding ACL item can be stored in the numerical order of the array index.

A store request by an Authorized Peer that attempts to overwrite any ACL item signed by another Peer is unauthorized and causes an Error_Forbidden response from the Storing Peer. Such access conflicts could be caused by an array index collision. However, the probability of a collision of two or more identical array indices will be negligibly low using the mechanism for isolating stored data (see Section 3.1).
6.2. Revoking Write Access

Write permissions are revoked by storing a non-existent value [I-D.ietf-p2psip-base] at the corresponding item of the Access Control List. Revoking a permission automatically invalidates all delegations performed by that user including all subsequent delegations. This allows to invalidate entire subtrees of the delegations tree with only a single operation. Overwriting the root item with a non-existent value of an Access List invalidates the entire delegations tree.

An existing ACL item MUST only be overwritten by the user who initially stored the corresponding entry, or by the Resource Owner that is allowed to overwrite all ACL items for revoking write access.

6.3. Validating Write Access through an ACL

Access Control Lists are used to transparently validate authorization of peers for writing a data value at a Shared Resource. Thereby it is assumed that the validating peer is in possession of the complete and most recent ACL for a specific Resource/Kind pair. The corresponding procedure consists of recursively traversing the trust delegation tree and proceeds as follows.

1. Obtain the username of the certificate used for signing the data stored at the Shared Resource.

2. Validate that an item of the corresponding ACL (i.e., for this Resource/Kind pair) contains a "to_user" field whose value equals the username obtained in step 1. If the Shared Resource under examination is an Access Control List Kind, further validate if the "ad" flag is set to 1.

3. Select the username of the certificate that was used to sign the ACL item obtained in step 2.

4. Validate that an item of the corresponding ACL contains a "to_user" field whose value equals the username obtained in step 3. Additionally validate that the "ad" flag is set to 1.

5. Repeat steps 3 and 4 until the "to_user" value is equal to the username of the signer of the previously selected ACL item. This final ACL item is expected to be the root item of this ACL which SHALL be further validated by verifying that the root item was signed by the owner of the ACL Resource.

The trust delegation chain is valid if and only if all verification steps succeed. In this case, the creator of the data value of the
Shared Resource is an Authorized Peer.

Note that the ACL validation procedure can be omitted whenever the creator of data at a Shared Resource is the Resource Owner itself. The latter can be verified by its public key certificate as defined in Section 6.6.

6.4. Operations of Storing Peers

Storing peers, i.e., peers at which Shared Resource and ACL are physically stored, are responsible for controlling storage attempts to a Shared Resource and its corresponding Access Control List. To assert the USER-CHAIN-ACL access policy (see Section 6.4), a storing peer MUST perform the access validation procedure described in Section 6.3 on any incoming store request using the most recent Access Control List for every Kind that uses the USER-CHAIN-ACL policy. It has further to ensure that only the Resource Owner stores new ACL root items for Shared Resources.

6.5. Operations of Accessing Peers

Accessing peers, i.e., peers that fetch a Shared Resource, MAY validate that the originator of a Shared Resource was authorized to store data at this Resource-ID by processing the corresponding ACL. To enable an accessing peer to perform the access validation procedure described in Section 6.3, it first needs to obtain the most recent Access Control List in the following way.

1. Send a Stat request to the Resource-ID of the Shared Resource to obtain all array indexes of stored ACL Kinds.

2. Fetch all indexes of existing ACL items at this Resource-ID by using the array ranges retrieved in the Stat request answer.

Peers can cache previously fetched Access Control Lists up to the maximum lifetime of an individual item. Since stored values could have been modified or invalidated prior to their expiration, an accessing peer SHOULD use a Stat request to check for updates prior to using the data cache.

6.6. USER-CHAIN-ACL Access Policy

This document specifies an additional access control policy to the RELOAD base draft [I-D.ietf-p2psip-base]. The USER-CHAIN-ACL policy allows Authorized Peers to write a Shared Resource, even though they do not own the corresponding certificate. Additionally, the USER-CHAIN-ACL allows the storage of Kinds with a variable resource name that are following one of the specified naming pattern. Hence, on an
inbound store request on a Kind that uses the USER-CHAIN-ACL access policy, the following rules MUST be applied:

In the USER-CHAIN-ACL policy, a given value MUST be written or overwritten, if either one of USER-MATCH or USER-NODE-MATCH (mandatory if the data model is dictionary) access policies of the base document [I-D.ietf-p2psip-base] applies.

Otherwise, the value MUST be written if the certificate of the signer contains a username that matches to one of the variable resource name pattern (c.f. Section 5) specified in the configuration document and, additionally, the hashed Resource Name matches the Resource-ID. The Resource Name of the Kind to be stored MUST be taken from the mandatory ResourceNameExtension field in the corresponding Kind data structure.

Otherwise, the value MUST be written if the ACL validation procedure described in Section 6.3 has been successfully applied.
7. ACL Kind Definition

This section defines the ACCESS-CONTROL-LIST Kind previously described in this document.

Name: ACCESS-CONTROL-LIST

Kind IDs: The Resource Name for ACCESS-CONTROL-LIST Kind-ID is the Resource Name of the Kind that will be shared by using the ACCESS-CONTROL-LIST Kind.

Data Model: The data model for the ACCESS-CONTROL-LIST Kind-ID is array. The array indexes are formed by using the mechanism for isolated stored data as described in Section 3.1

Access Control: USER-CHAIN-ACL (see Section 6.6)
8.  Security Considerations

In this section we discuss security issues that are relevant to the usage of shared resources in RELOAD.

8.1.  Resource Exhaustion

Joining a RELOAD overlay inherently poses a certain resource load on a peer, because it has to store and forward data for other peers. In common RELOAD semantics, each Resource ID and thus position in the overlay may only be written by a limited set of peers - often even only a single peer, which limits this burden. In the case of Shared Resources, a single resource may be written by multiple peers, who may even write an arbitrary number of entries (e.g., delegations in the ACL). This leads to an enhanced use of resources at individual overlay nodes. The problem of resource exhaustion can easily be mitigated for Usages based on the ShaRe-Usage by imposing restrictions on size, i.e., <max-size> element for a certain Kind in the configuration document.

8.2.  Malicious or Misbehaving Storing Peer

The RELOAD overlay is designed to operate despite the presence of a small set of misbehaving peers. This is not different for Shared Resources since a small set of malicious peers does not disrupt the functionality of the overlay in general, but may have implications for the peers needing to store or access information at the specific locations in the ID space controlled by a malicious peer. A storing peer could withhold stored data which results in a denial of service to the group using the specific resource. But it could not return forged data, since the validity of any stored data can be independently verified using the attached signatures.

8.3.  Privacy Issues

All data stored in the Shared Resource is publicly readable, thus applications requiring privacy need to encrypt the data. The ACL needs to be stored unencrypted, thus the list members of a group using a Shared Resource will always be publicly visible.
9. IANA Considerations

TODO: register Kind-ID code point at the IANA
10. Acknowledgments

This work was stimulated by fruitful discussions in the P2PSIP working group and SAM research group. We would like to thank all active members for constructive thoughts and feedback. In particular, the authors would like to thank (in alphabetical order) Lothar Grimm, Cullen Jennings, Peter Musgrave, Joerg Ott, Marc Petit-Huguenin, Peter Pogrzeba, and Jan Seedorf. This work was partly funded by the German Federal Ministry of Education and Research, projects HAMcast and Mindstone.
11. References

11.1. Normative References

[I-D.ietf-p2psip-base]
   Jennings, C., Lowekamp, B., Rescorla, E., Baset, S., and H. Schulzrinne,
   "REsource LOcation And Discovery (RELOAD) Base Protocol",

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels",

11.2. Informative References

[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.knauf-p2psip-disco]
   Knauf, A., Hege, G., Schmidt, T., and M. Waehlisch,
   "A RELOAD Usage for Distributed Conference Control (DisCo)",
   draft-knauf-p2psip-disco-03 (work in progress), July 2011.

[RFC3550] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson,
   "RTP: A Transport Protocol for Real-Time Applications",
   STD 64, RFC 3550, July 2003.
Appendix A. Change Log

The following changes have been made from version draft-knauf-p2psip-share-01:

1. Simplified the ACL data structure in response to WG feedback
2. Added ResourceNameExtension data structure to simplify the use of variable resource names
3. Restructured document
4. Many editorial improvements

The following changes have been made from version draft-knauf-p2psip-share-00:

1. Integrated the USER-PATTERN-MATCH access policy into USER-CHAIN-ACL
2. Access Control List Kind uses USER-CHAIN-ACL exclusively
3. Resources to be shared use USER-CHAIN-ACL exclusively
4. More precise specification of mandatory User_name and Resource_name fields for Shared Resources
5. Added mechanism for isolating stored data to prevent race conditions while concurrent storing
6. XML Extension for variable resource names uses its own namespace
7. Many editorial improvements
Authors’ Addresses

Alexander Knauf  
HAW Hamburg  
Berliner Tor 7  
Hamburg D-20099  
Germany  

Phone: +4940428758067  
Email: alexander.knauf@haw-hamburg.de  
URI: http://inet.cpt.haw-hamburg.de/members/knauf

Gabriel Hege  
HAW Hamburg  
Berliner Tor 7  
Hamburg D-20099  
Germany  

Phone: +4940428758067  
Email: hege@fhtw-berlin.de  
URI: http://inet.cpt.haw-hamburg.de/members/hege

Thomas C. Schmidt  
HAW Hamburg  
Berliner Tor 7  
Hamburg D-20099  
Germany  

Email: schmidt@informatik.haw-hamburg.de  
URI: http://inet.cpt.haw-hamburg.de/members/schmidt

Matthias Waehlisch  
link-lab & FU Berlin  
Hoenower Str. 35  
Berlin D-10318  
Germany  

Email: mw@link-lab.net  
URI: http://www.inf.fu-berlin.de/~waehl
An SNMP Usage for RELOAD
draft-peng-p2psip-snmp-04

Abstract

This document defines a SNMP Usage for REsource LOcation And Discovery (RELOAD). The SNMP Usage provides the functionality of managing the RELOAD network. The SNMP Usage provides lookup service for the network manager’s address stored in the overlay. The SNMP Usage also defines the method that allow the registrations to map a network manager’s name to a specific node reachable through the overlay. The AppAttach method is used to establish a direct connection between nodes through which SNMP messages are exchanged.

Status of this Memo

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

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

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

This Internet-Draft will expire on September 7, 2012.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents.
carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction ................................................. 3
2. Terminology ............................................... 3
3. Network Management Requirements ........................... 4
4. Basic Operations and SNMP ................................. 5
5. Overview of SNMP Usage ..................................... 5
6. SNMP Usage Architecture ................................. 6
7. Abstract Service Interfaces ............................... 7
  7.1. SNMP-RELOAD Application Primitive ................... 7
     7.1.1. getNodeForResource ASI .......................... 7
     7.1.2. returnNodeForResource ASI ....................... 7
     7.1.3. getAddressForNode ASI .......................... 8
     7.1.4. returnAddressForNode ASI ....................... 8
  7.2. RELOAD Node(M-Node/O-Node) primitive ................. 8
     7.2.1. getNodeForResource ASI .......................... 8
     7.2.2. returnNodeForResource ASI ....................... 8
     7.2.3. exchangeCandidateAddressList ASI ............... 9
     7.2.4. registerManagerReq ASI .......................... 9
     7.2.5. registerManagerAns ASI .......................... 10
8. Managed Object Definitions for RELOAD .................... 10
9. Network Manager Registering and Looking up ............... 15
10. An SNMP Entity Forms a Direct Connection with Another SNMP Entity ................................................. 17
11. O-Node information Collection ........................... 19
12. Network Manager Looks up the O-Node for a Resource .... 19
13. SNMP-REGISTRATION Kind Definition ..................... 21
14. Security Considerations .................................. 21
15. IANA Considerations ...................................... 22
16. Acknowledgments .......................................... 22
17. References ............................................... 22
  17.1. Normative References ................................. 22
  17.2. Informative References ............................... 23
Appendix A. Additional Stuff ................................. 23
Authors’ Addresses ........................................... 23
1. Introduction

This document defines an SNMP Usage for RELOAD, which can be used to manage the RELOAD network. It can provide important network management functions, such as adjusting the network configuration, monitoring the performance of the network, collecting real-time failure information, etc. These network management functions are essential for stable operation and high-quality services of the network. As traditional network management protocols (e.g., SNMP) cannot be directly applied to RELOAD network management, it is necessary to introduce new RELOAD usage of SNMP.

As defined in [I-D.ietf-p2psip-base], there are two kinds of nodes in RELOAD network: centralized servers, such as the Enrollment Server; distributed nodes, such as Peer and Client. The management function of centralized servers can be carried out by traditional management methods, so we don’t discuss the management of the centralized server in this document, and only focus on the management of the distributed nodes called as RELOAD Nodes.

When the manager starts up, it needs to register the mapping between its name and Node ID into the RELOAD network, in order to be found by the managed RELOAD Nodes. So only the name of manager needs be fixed, and needs be knowned beforehand by RELOAD Nodes. Then, the RELOAD Nodes can get the manager’s address and connect with it, and then Nodes and manager can send management messages each other through this link.

Not only RELOAD Nodes are managed object, but RELOAD resource is managed object as well, such as some data stored in RELOAD network by SNMP-RELOAD application.

The basic mechanism of SNMP Usage is the same as SIP Usage for RELOAD [I-D.draft-ietf-p2psip-sip]. If you have read the draft of SIP Usage, SNMP Usage will be more easy understood.

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 Concepts and Terminology for Peer to Peer SIP [I-D.ietf-p2psip-concepts] and the RELOAD Base Protocol [I-D.ietf-p2psip-base] and SNMPv3 [RFC3411] and TLS TM for SNMP [RFC5953] extensively in this document.

Entity: SNMP Entity, including both Manager and Agent, which resides in RELOAD Node.

Manager: SNMP Manager, which resides in RELOAD Node.

Agent: SNMP Agent, which resides in RELOAD Node.

Node: RELOAD Node, including both Peer and Client, which SNMP manager or agent resides in.

M-Node: Management Node, which is the RELOAD Node which SNMP Manager resides in.

O-Node: Objective Node, which is the RELOAD Node managed by a network manager, which SNMP agent resides in.

R-Node: Responsibility Node, which is the RELOAD Node responsible for storing the data according to P2P algorithm.

SNMP-RELOAD Application: It provides the functions related to RELOAD for SNMP applications, such as getting available address for Node ID and getting Node ID for Resource ID. SNMP applications can implement the management for RELOAD network by SNMP-RELOAD Application.

3. Network Management Requirements

SNMP usage SHOULD or MAY provide these functions and mechanisms as below:

1. SNMP usage for RELOAD SHOULD provide the management functions for RELOAD Nodes. Such as setting node name, software version or other configuration information, monitoring the number of the messages initiated, forwarded or processed by nodes, reporting program failure, message forwarding failure or other error on nodes.

2. SNMP usage for RELOAD SHOULD provide the management functions for RELOAD resource. Such as tracing forwarded the RELOAD messages, processing flows of resources.

3. SNMP usage for RELOAD SHOULD provide mechanisms for SNMP entities to discover each other based on RELOAD NodeID.

4. SNMP usage for RELOAD SHOULD provide mechanisms for SNMP entities to establish a secure connection between each other.
5. SNMP usage for RELOAD SHOULD provide mechanisms for SNMP manager to discover the RELOAD NodeID associated to a given ResourceID.

6. SNMP usage for RELOAD SHOULD provide mechanisms for SNMP entities to traverse the NAT in front of the SNMP entities which they will connect to.

7. SNMP usage for RELOAD MAY provide mechanisms for SNMP entities to discover the SNMP manager based on manager names or functions.

4. Basic Operations and SNMP

The management interactions between nodes can be abstracted into a few basic operations: 1). the network manager requests data of nodes and resources; 2). the network manager sets data of nodes and resources; 3). nodes initiate data reports to the network manager. A variety of management functions can be carried out by these basic operations or their combinations. This document adopts SNMP as a RELOAD Usage to achieve the management of the RELOAD network. The basic operations described above can be implemented by messages defined in SNMP, such as GetRequest, GetNextRequest, GetBulkRequest, Response, SetRequest, Trap, InformRequest.

5. Overview of SNMP Usage

The SNMP entity is deployed as an application on RELOAD Nodes in the SNMP usage for RELOAD. In other words, each SNMP entity is associated with a RELOAD Node. SNMP entities discover other entities (agents or managers) by RELOAD mechanisms and to set up connects with other SNMP entities. Therefore, SNMP entities talk to each other using SNMP protocol on dedicated connections, while RELOAD protocols are used for Node discovery and connection setup. The diagram of system composition and protocol is as follow:
6. SNMP Usage Architecture

This document defines SNMP Usage Architecture, which includes SNMP-RELOAD application and SNMP applications in the Simple Network Management Protocol (SNMP) architecture defined in [RFC3411]. The SNMP-RELOAD Application will provide the functions related to RELOAD, such as getting available address for Node ID and getting Node ID for Resource ID, to SNMP applications to implement the management for RELOAD network. This document identifies and describes some key aspects that need to be considered for SNMP usage for RELOAD. The following diagram depicts SNMP usage architecture.
7. Abstract Service Interfaces

7.1. SNMP-RELOAD Application Primitive

7.1.1. getNodeForResource ASI

The getNodeForResource ASI is provided for SNMP application by SNMP-RELOAD application, and it is used to get Node ID of RELOAD Node that is responsible for resource.

```plaintext
getNodeForResource(
    IN     resourceName     -- managed resource name
)
```

7.1.2. returnNodeForResource ASI

The returnNodeForResource ASI is used to return the Node ID of RELOAD Node that is responsible for resource to SNMP applications by SNMP-RELOAD application.

```plaintext
result =     -- SUCCESS or errorIndication
returnNodeForResource(
    IN     resourceName     -- managed resource name
)
IN     nodeID     -- node that responsible for managed resource name
)

7.1.3.  getAddressForNode ASI

The getAddressForNode ASI is provided for SNMP Applications(e.g. Command Application, Notification Application) by SNMP-RELOAD application, and it is used to get the address of the other side for SNMP communication.

getAddressForNode(
    IN     nodeID     -- destination node
)

7.1.4.  returnAddressForNode ASI

The returnAddressForNode ASI is used to return the address of the other side for SNMP communication.

result =     -- SUCCESS or errorIndication
returnAddressForNode(
    IN     nodeID     -- destination node
    IN     transportAddress     -- destination network address
)

7.2.  RELOAD Node(M-Node/O-Node) primitive

7.2.1.  getNodeForResource ASI

The getNodeForResource ASI is provided for SNMP-RELOAD Application by RELOAD Node(M-Node/O-Node), and it is used to get Node ID of RELOAD Node that is responsible for resource. The definition of getNodeForResource is above.

7.2.2.  returnNodeForResource ASI

The returnNodeForResource ASI is used to return the Node ID of RELOAD Node that is responsible for resource to SNMP-RELOAD Application by RELOAD Node. The definition of returnNodeForResource is above.
7.2.3. exchangeCandidateAddressList ASI

The exchangeCandidateAddressList ASI is used by SNMP-RELOAD Application and RELOAD Node to exchange the address list each other for SNMP communication, and these address lists will be used to do NAT traversal by the way of ICE.

exchangeCandidateAddressList(
    IN    nodeID     -- destination node
    IN    ufrag     -- the username fragment (from ICE)
    IN    password   -- the ICE password
    IN    candidateAddressList -- sender’s candidate address list
)

the Elements of candidateAddress of candidateAddressList including: IP address, LinkType, etc.

In order to implement ICE, these items need to be added into LCD:

Ufrag
Password

LinkType: DTLS-UDP-NO-ICE, DTLS-UDP-ICE, TLS-TCP-NO-ICE, TLS-TCP-ICE.

7.2.4. registerManagerReq ASI

The registerManagerReq ASI is provided for RELOAD Application by RELOAD M-Node, and it is used to register the Node ID of the RELOAD Node which Manager resides in.

registerManagerReq(
    IN    managerName   -- the name of Manager
    IN    nodeID        -- the Node ID of the RELOAD Node which Manager resides in
)
7.2.5. registerManagerAns ASI

The registerManagerAns ASI is used to return the result of registering to SNMP-RELOAD Application by RELOAD M-Node.

result = -- SUCCESS or errorIndication

8. Managed Object Definitions for RELOAD

SNMP-RELOAD-MIB DEFINITIONS ::= BEGIN

IMPORTS
MODULE-IDENTITY, OBJECT-TYPE, OBJECT-IDENTITY, mib-2, Counter32,
Gauge32, Integer32, NOTIFICATION-TYPE
FROM SNMPv2-SMI -- RFC 2578 or any update thereof
MODULE-COMPLIANCE, OBJECT-GROUP,
NOTIFICATION-GROUP
FROM SNMPv2-CONF -- RFC 2580 or any update thereof
TimeStamp
FROM SNMPv2-TC -- RFC 2579 or any update thereof
;

snmpReloadMIB MODULE-IDENTITY
LAST-UPDATED "201202280000Z"
ORGANIZATION "P2PSIP Working Group"
CONTACT-INFO "WG-EMail: p2psip@ietf.org"

DESCRIPTION "SNMP Usage for RELOAD MIB

Copyright (c) 2011 IETF Trust and the persons identified as
the document authors. All rights reserved.

Redistribution and use in source and binary forms, with or
without modification, is permitted pursuant to, and subject
to the license terms contained in, the Simplified BSD License
set forth in Section 4.c of the IETF Trust's Legal Provisions
Relating to IETF Documents
(http://trustee.ietf.org/license-info)."

REVISION "201202280000Z"
DESCRIPTION "This version of this MIB module is part of
draft-peng-p2psip-snmp-04; see the draft itself
for full legal notices."
 ::= { mib-2 199 }
-- subtrees of the SNMP-RELOAD-MIB

snmpReloadNotifications OBJECT IDENTIFIER ::= { snmpReloadMIB 0 }
snPmReloadObjects OBJECT IDENTIFIER ::= { snmpReloadMIB 1 }
snPmReloadConformance OBJECT IDENTIFIER ::= { snmpReloadMIB 2 }

-- snmpReloadObjects - Objects

-- Configuration Objects
snmpReloadConfig OBJECT IDENTIFIER ::= { snmpReloadObjects 1 }

snmpReloadConfigVersion OBJECT-TYPE
SYNTAX Gauge32
MAX-ACCESS read-only
STATUS current
DESCRIPTION "The version of snmpReloadConfigItem."
::= { snmpReloadConfig 1 }

snmpReloadConfigLastChanged OBJECT-TYPE
SYNTAX TimeStamp
MAX-ACCESS read-only
STATUS current
DESCRIPTION "The value of sysUpTime.0 when the snmpReloadConfigItem was last modified through any means, or 0 if it has not been modified since the command responder was started."
::= { snmpReloadConfig 2 }

snmpReloadConfigItem OBJECT IDENTIFIER ::= { snmpReloadConfig 3 }

snmpReloadNodeName OBJECT-IDENTITY
STATUS current
DESCRIPTION "The name of RELOAD Node."
::= { snmpReloadConfigItem 1 }

snmpReloadNodeId OBJECT-IDENTITY
STATUS current
DESCRIPTION "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."
 ::= { snmpReloadConfigItem 2 }

snmpReloadNodeType OBJECT-TYPE
 SYNTAX    Integer32(0..9)
 MAX-ACCESS read-write
 STATUS     current
 DESCRIPTION
 "the type of RELOAD Node.  
 Definition of values as follows:
 Client(0),
 Peer(1)
"
 ::= { snmpReloadConfigItem 3 }

snmpReloadLogPrintLevel OBJECT-TYPE
 SYNTAX    Integer32(0..9)
 MAX-ACCESS read-write
 STATUS     current
 DESCRIPTION
 "the type of RELOAD Node.  
 Definition of values as follows:
 debug(3),
 info(2),
 warn(1),
 error(0)
"
 ::= { snmpReloadConfigItem 4 }

snmpReloadNotificationEnable OBJECT-TYPE
 SYNTAX    Integer32(0..9)
 MAX-ACCESS read-write
 STATUS     current
 DESCRIPTION
 "Whether are notifications sent.  
 Definition of values as follows:
 Disable(0),
 Enable(1)
"
 ::= { snmpReloadConfigItem 5 }

-- The snmpReloadFailures Group
snmpReloadFailures OBJECT IDENTIFIER ::= { snmpReloadObjects 2 }

snmpReloadMessageForwardFailures OBJECT-TYPE
 SYNTAX    Counter32
 MAX-ACCESS read-only
STATUS current
DESCRIPTION
"The number of times RELOAD message failed to be forwarded,
for any reason."
::= { snmpReloadFailures 1 }

snmpReloadDataUpdateFailures OBJECT-TYPE
SYNTAX Counter32
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"The number of times RELOAD data failed to be updated,
for any reason."
::= { snmpReloadFailures 2 }

-- ************************************************
-- snmpReloadNotifications - Notifications Information
-- ************************************************

snmpReloadMessageForwardFailNotification NOTIFICATION-TYPE
OBJECTS { snmpReloadMessageForwardFailures }  
STATUS current
DESCRIPTION
"Notification that RELOAD message failed to be forwarded."
::= { snmpReloadNotifications 1 }

snmpReloadDataUpdateFailNotification NOTIFICATION-TYPE
OBJECTS { snmpReloadDataUpdateFailures }  
STATUS current
DESCRIPTION
"Notification that RELOAD data failed to be updated."
::= { snmpReloadNotifications 2 }

-- ************************************************
-- snmpReloadComplings - Conformance Information
-- ************************************************

snmpReloadComplings OBJECT IDENTIFIER ::= {snmpReloadConformance 1}

snmpReloadGroups OBJECT IDENTIFIER ::= { snmpReloadConformance 2 }

-- ************************************************
-- Compliance statements
-- ************************************************

snmpReloadCompliance MODULE-COMPLIANCE
STATUS current
DESCRIPTION "The compliance statement for SNMP engines that support
the SNMP-RELOAD-MIB"
MODULE
  MANDATORY-GROUPS { snmpReloadConfigGroup,
                   snmpReloadFailuresGroup,
                   snmpReloadNotificationGroup }::= { snmpReloadCompliances 1 }

-- ************************************************
-- Units of conformance
-- ************************************************

snmpReloadConfigGroup OBJECT-GROUP
  OBJECTS {snmpReloadConfigVersion,
            snmpReloadConfigLastChanged,
            snmpReloadNodeType,
            snmpReloadLogPrintLevel,
            snmpReloadNotificationEnable}
  STATUS current
DESCRIPTION "A collection of objects for maintaining configuration
information of an SNMP engine that implements
the SNMP Usage for RELOAD."
::= { snmpReloadGroups 1 }

snmpReloadFailuresGroup OBJECT-GROUP
  OBJECTS {snmpReloadMessageForwardFailures,
            snmpReloadDataUpdateFailures}
  STATUS current
DESCRIPTION "A collection of objects for failures
information of an SNMP engine that implements
the SNMP Usage for RELOAD."
::= { snmpReloadGroups 2 }

snmpReloadNotificationGroup NOTIFICATION-GROUP
  NOTIFICATIONS {
    snmpReloadMessageForwardFailNotification,
    snmpReloadDataUpdateFailNotification
  }
9. Network Manager Registering and Looking up

The Node ID of the network manager which acts as a provider of management service should be able to be found by agents on RELOAD nodes, thus agents can send messages to the manager. The Node ID of network manager may not be fixed or predefined in advance. So a recognizable name is necessary and the managed nodes should find the Node ID of the manager through this fixed name. Therefore, it is necessary for the manager to register itself in the network after joining the network. In other words, the manager needs to store the mapping between its name and its Node ID in the RELOAD network. When an agent wants to contact the manager, it needs to first look up the manager’s Node ID corresponding to the predefined management service name. This registration is achieved by storing the name of the network manager and the structure of SnmpRegistration into the RELOAD network. The corresponding SNMP-REGISTRATION Kind-ID will be formally defined in the following chapter. It is proposed to store the mapping of the manager’s name to a destination list in this document. Therefore, a single Node ID as a special case for a destination list. The contents of a SnmpRegistration structure are as follows:

```c
struct {
    opaque contact_prefs<0..2^16-1>;
    Destination destination_list<0..2^16-1>;
} SnmpRegistrationData;

struct {
    uint16 length;
    SnmpRegistrationData data;
} SnmpRegistration;
```

The contents of the SnmpRegistration PDU are:
length

the length of the rest of the PDU

data

the contents of the registration data is an opaque string
containing the network manager’s contact preferences and a
destination list for the peer.

When an agent needs to contact to a network manager, it must perform
a query of SnmpRegistration by FetchReq message to get the manager’s
Node ID.

The process diagram for Network Manager Registering and Looking up is
as follow:
10. An SNMP Entity Forms a Direct Connection with Another SNMP Entity

Because the targets of the management tasks and reports of RELOAD network are Node ID of RELOAD or snmeEngineID of SNMP. (Note: In this document, snmeEngineID constructed from Node ID.) So when a SNMP Entity needs to send SNMP messages to another SNMP Entity, it must get the other side of available IP address firstly. Due to the existence of NAT, they need to exchange ICE addresses each other and checks connectivity, then selects a pair of available IP address to
establish connection. (Of course, if a connection has been established between this pair of IP address, the initiator node will directly send messages to the target node.) The process of forming a direct connect between SNMP Entities as shown below:

```
Entity 1
SNMP  SNMP-RELOAD  RELOAD
applications application  M/O-Node
```
```
Entity 2
RELOAD  SNMP-RELOAD
O/M-Node  application
```

getAddressForNode
-----------------

```
Get ICE ufrag/
password from
LCD, collect
candidate
address list
```

exchangeCandidateAddressList
-----------------

```
AppAttach
```
```
AppAttach
```

exchangeCandidateAddressList
-----------------

```
ICE Check
```


11. O-Node information Collection

Before a network manager performs management tasks for nodes, it must first collect the Node ID and its status information of managed nodes. The way of a manager collecting the information of RELOAD nodes (including Peer and Client) is as follows: when an agent starts up, its associated RELOAD Node joins the RELOAD network, it needs to get the name of a network manager from its configuration or an Enrollment Server; then this node connects to the network manager and registers its own information, such as node name, Node ID, status, etc., to the manager. The procedure of finding the manager and connecting to it has been introduced in the previous section. There are many other ways to collect information of managed nodes, which could be studied in future documents.

12. Network Manager Looks up the O-Node for a Resource

When a network manager needs to send a management task for resource, it is necessary that the network manager first gets the Node ID of the O-Node responsible for the resource so as to judge whether there is a connection with the O-Node. One way for the manager to get the Node ID of the O-Node responsible for the resource is to acquire the Node ID of the O-NODE responsible for the target resource through via_list of Forwarding Header in FindAns. The process is as follows:

Firstly, the network manager sends an FindReq to the RELOAD network, the target Resource ID is put into the destination_list of the
FindReq. Then the RELOAD network routes FindReq to the node responsible for the target Resource ID according to its routing algorithm.

Secondly, the O-Node returns FindAns to the network manager through the RELOAD network. The first Node ID in the via_list of the Forwarding Header of the FindAns is the Node ID of the O-Node responsible for the target resource.

The process diagram for Network Manager Looking up the O-Node for a Resource is as below:

```
+---------------------------------------+ +-------------+
|Manager                                | | Agent       |
|                                       | |             |
|   SNMP      SNMP-RELOAD      RELOAD   | |    RELOAD   |
|applications application      M-Node   | |    O-Node   |
|                                       | |             |
|              |              |              |
|+-------------+              |+-------------+|
|getNodeForResource           |              |
|------------->|              |              |
|              |returnNodeForResource        |
|              |<-------------|              |
|              |              |    Find      |
|              |<------------>|
|              |              |              |
|              |returnNodeForResource        |
|              |<-------------|              |
After the network manager gets the Node ID of O-Node, it will be able to judge whether there is a connection between itself and the O-Node. If the connection exists, the network manager may directly send SNMP
```
message to the O-Node, otherwise it is necessary to establish a new connection to the O-Node.

13. SNMP-REGISTRATION Kind Definition

This section defines the SNMP-REGISTRATION kind.

Name          SNMP-REGISTRATION

Kind ID       The Resource Name for the SNMP-REGISTRATION Kind-ID is the Name of the network manager. The data stored is a SnmpRegistrationData, which can contain a destination list and contact preferences to the peer which is acting for the network manager.

Data Model    The data model for the SNMP-REGISTRATION Kind-ID is single value.

Access Control USER-NODE-MATCH.

Data stored under the SNMP-REGISTRATION kind is of type SnmpRegistration. A destination list can be used to reach the network manager.

14. Security Considerations

There are three solutions to the security problem in SNMP Usage for RELOAD. The first option is shared key based solution, which is SNMPv3 security solution (USM). The second option is PKI based security solution, which is to use the certificate of RELOAD to authenticate and encrypt the SNMP messages. The third option is (D)TLS based security solution, which uses the secure (D)TLS links to transfer the SNMP message.

USM was designed to be independent of other existing security infrastructures. USM therefore uses a separate principal and key management infrastructure. Operators have reported that deploying another principal and key management infrastructure in order to use SNMPv3 is a deterrent to deploying SNMPv3. [RFC5590] And the efficiency of the second option is questionable. So we recommend the third option.

The special detail of (D)TLS based security for SNMP is defined in RFC5953, and it won’t be described again in this document. In short, we propose to use RELAOD certificate for setting up the connection using (D)TLS based security. When the Mapping certificate’s
subjectAltName to a tmSecurityName is used in the SNMP-TLS-TM-MIB’s snmpT1stmCertToTSNTable, tmSecurityName is derived from the user name value of the SubjectAltName field in RELOAD certificate.

15. IANA Considerations

This document has no IANA Considerations.

16. Acknowledgments

This draft is based on "REsource LOcation And Discovery (RELOAD) Base Protocol" draft by C. Jennings, B. Lowekamp, E. Rescorla, S. Baset and H. Schulzrinne.

This draft make a reference to "A SIP Usage for RELOAD" draft by C. Jennings, B. Lowekamp, Ed., E. Rescorla, S. Baset, H. Schulzrinne.

This draft is based on "Architecture for Describing Simple Network Management Protocol (SNMP) Management Framewoks" RFC by Harrington, D., Presuhn, R., and B. Wijnen.

This draft is based on "Transport Layer Security (TLS) Transport Model for the Simple Network Management Protocol (SNMP)" RFC by Hardaker, W..

Thanks to David Harrington and the many people who give us much significative advice.

Thanks to the many people of the IETF P2PSIP WG and Network WG whose many drafts and RFCs we have learned.

17. References

17.1. Normative References


[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
17.2. Informative References

[I-D.ietf-p2psip-concepts]

[I-D.narten-iana-considerations-rfc2434bis]


Authors’ Addresses

YongLin Peng
ZTE Corporation
Nanjing, 210012
China

Phone: +86 13776637274
Email: peng.yonglin@zte.com.cn

Wei Wang
ZTE Corporation
Nanjing, 210012
China

Phone: +86 13851658076
Email: wang.weil08@zte.com.cn

ZhenWu Hao
ZTE Corporation
Nanjing, 210012
China

Phone: +86 13382087596
Email: hao.zhenwu@zte.com.cn

Yu Meng
ZTE Corporation
Nanjing, 210012
China

Phone: +86 18651806839
Email: meng.yu@zte.com.cn