P2PSIP
Internet-Draft
Intended status: Standards Track
Expires: September 15, 2011

C. Jennings
Cisco
B. Lowekamp, Ed.
Skype
E. Rescorla
RTFM, Inc.
S. Baset
H. Schulzrinne
Columbia University
March 14, 2011

REsource LOcation And Discovery (RELOAD) Base Protocol
draft-ietf-p2psip-base-13

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.

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 15, 2011.

Copyright Notice

Table of Contents

1. Introduction

This document defines REsource LOcation And Discovery (RELOAD), a peer-to-peer (P2P) signaling protocol for use on the Internet. It provides a generic, self-organizing overlay network service, allowing nodes to efficiently route messages to other nodes and to efficiently store and retrieve data in the overlay. RELOAD provides several features that are critical for a successful P2P protocol for the Internet:

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

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

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

High Performance Routing: The very nature of overlay algorithms introduces a requirement that peers participating in the P2P network route requests on behalf of other peers in the network. This introduces a load on those other peers, in the form of bandwidth and processing power. RELOAD has been defined with a simple, lightweight forwarding header, thus minimizing the amount of effort required by intermediate peers.

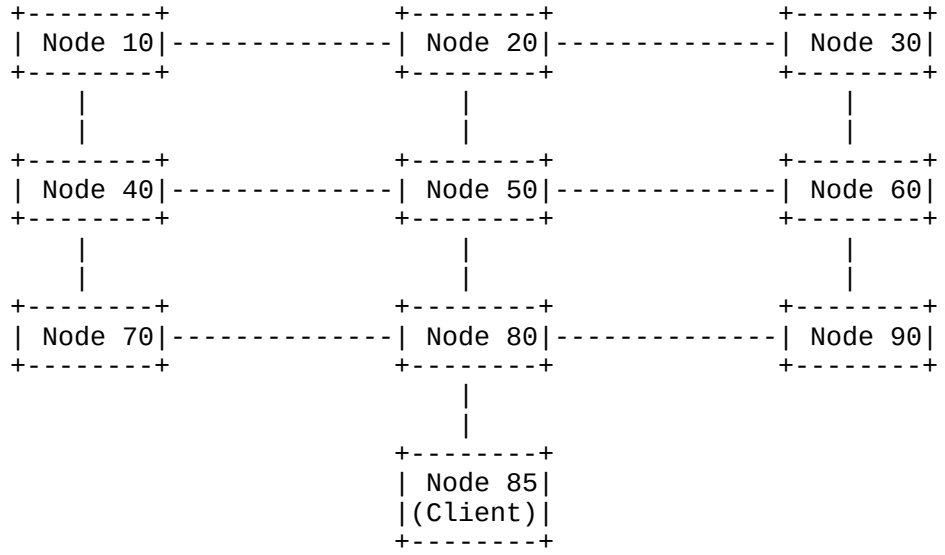
Pluggable Overlay Algorithms: RELOAD has been designed with an abstract interface to the overlay layer to simplify implementing a variety of structured (e.g., distributed hash tables) and unstructured overlay algorithms. This specification also defines how RELOAD is used with the Chord DHT algorithm,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

application.

These properties were designed specifically to meet the requirements for a P2P protocol to support SIP. This document defines the base protocol for the distributed storage and location service, as well as critical usages for NAT traversal and security. The SIP Usage itself is described separately in [I-D.ietf-p2psip-sip]. RELOAD is not limited to usage by SIP and could serve as a tool for supporting other P2P applications with similar needs. RELOAD is also based on the concepts introduced in [I-D.ietf-p2psip-concepts].

1.1. Basic Setting

In this section, we provide a brief overview of the operational setting for RELOAD. See the concepts document[I-D.ietf-p2psip-concepts] for more details. A RELOAD Overlay Instance consists of a set of nodes arranged in a connected graph. Each node in the overlay is assigned a numeric Node-ID which, together with the specific overlay algorithm in use, determines its position in the graph and the set of nodes it connects to. The figure below shows a trivial example which isn't drawn from any particular overlay algorithm, but was chosen for convenience of representation.



Because the graph is not fully connected, when a node wants to send a message to another node, it may need to route it through the network. For instance, Node 10 can talk directly to nodes 20 and 40, but not to Node 70. In order to send a message to Node 70, it would first

send it to Node 40 with instructions to pass it along to Node 70. Different overlay algorithms will have different connectivity graphs, but the general idea behind all of them is to allow any node in the graph to efficiently reach every other node within a small number of hops.

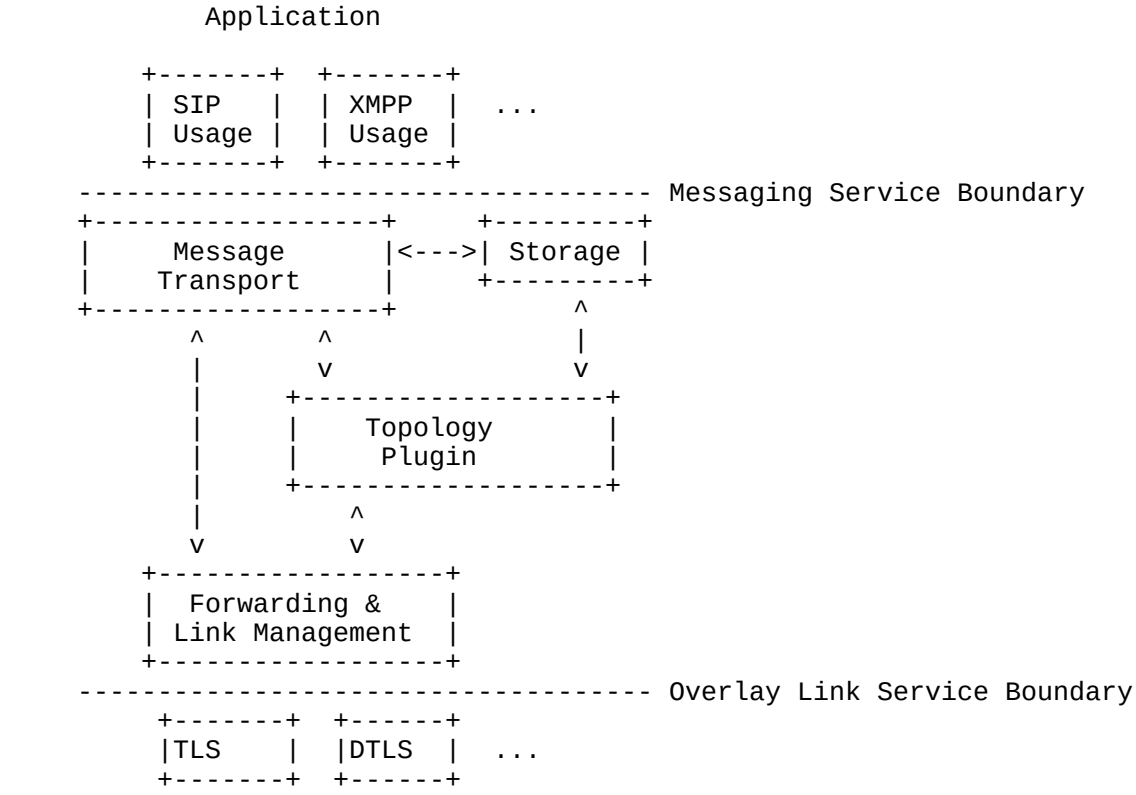
The RELOAD network is not only a messaging network. It is also a storage network. Records are stored under numeric addresses which occupy the same space as node identifiers. Peers are responsible for storing the data associated with some set of addresses as determined by their Node-ID. For instance,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

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

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

1.2. Architecture

RELOAD is fundamentally an overlay network. The following figure shows the layered RELOAD architecture.



The major components of RELOAD are:

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

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

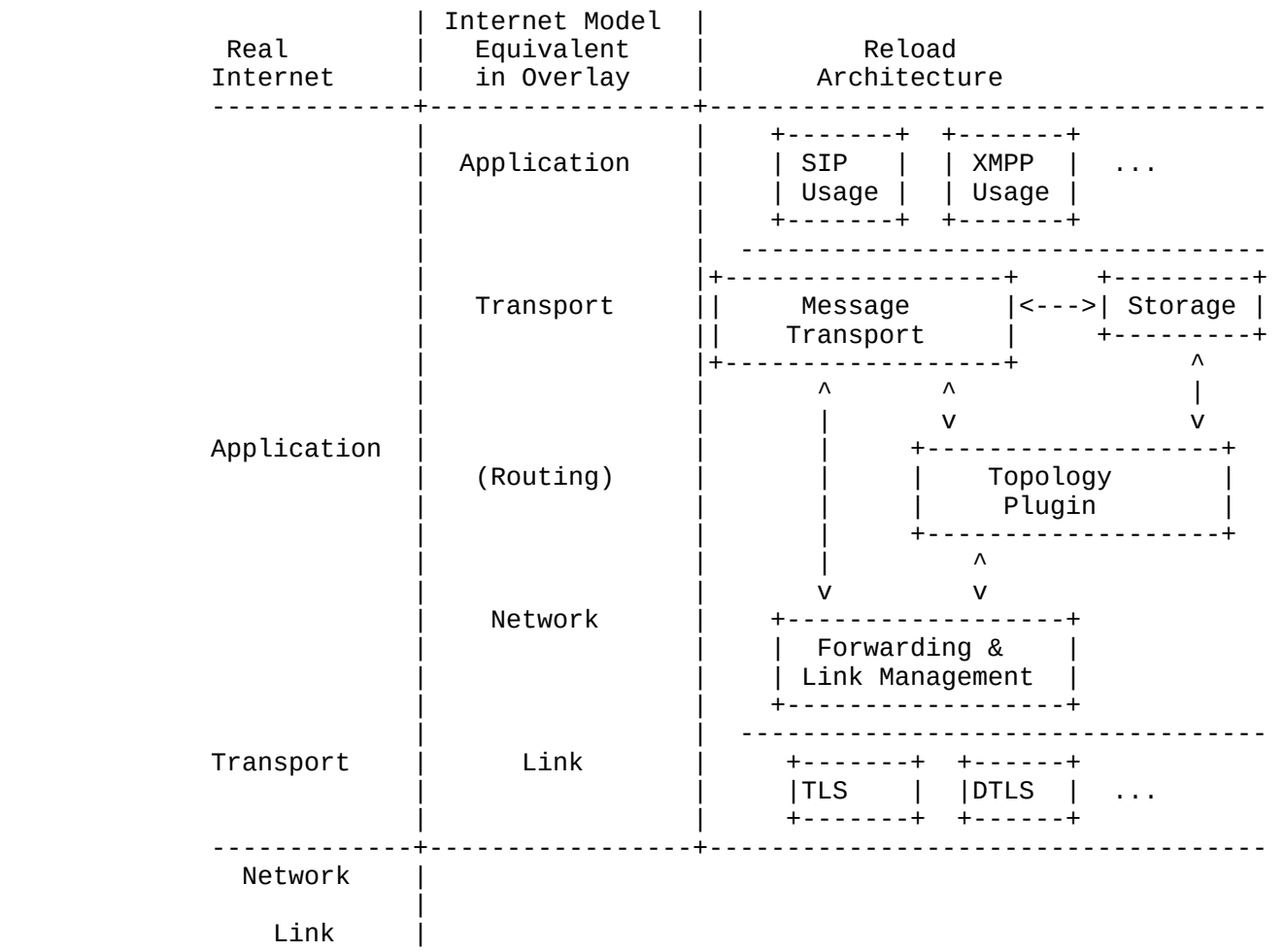
Storage: The Storage component is responsible for processing messages relating to the storage and retrieval of data. It talks directly to the Topology Plugin to manage data replication and migration, and it talks to the Message Transport component to send and receive messages.

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

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

Overlay Link Layer: Responsible for actually transporting traffic directly between nodes. Each such protocol includes the appropriate provisions for per-hop framing or hop-by-hop ACKs required by unreliable transports. TLS [RFC5246] and DTLS [RFC4347] are the currently defined "link layer" protocols used by RELOAD for hop-by-hop communication. New protocols MAY 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. An usage

may define multiple kinds of data that are stored in the overlay and may also rely on kinds originally defined by other usages.

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

1.2.2. Message Transport

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

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

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

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

1.2.3. Storage

One of the major functions of RELOAD is to allow nodes to store data in the overlay and to retrieve data stored by other nodes or by themselves. The Storage component is responsible for processing data storage and retrieval messages. For instance, the Storage component might receive a Store request for a given resource from the Message Transport. It would then query the appropriate usage before storing the data value(s) in its local data store and sending a response to the Message Transport for delivery to the requesting node. Typically, these messages will come from other nodes, but depending

on the overlay topology, a node might be responsible for storing data for itself as well, especially if the overlay is small.

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

1.2.4. Topology Plugin

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

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

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

1.2.5. Forwarding and Link Management Layer

The Forwarding and Link Management Layer is responsible for getting a message to the next peer, as determined by the Topology Plugin. This Layer establishes and maintains the network connections as required by the Topology Plugin. This layer is also responsible for setting up connections to other peers through NATs and firewalls using ICE,

and it can elect to forward traffic using relays for NAT and firewall traversal.

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

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

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

1.3. Security

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

Connection Level: Connections between peers are secured with TLS, DTLS, or potentially some to be defined future protocol.
Message Level: Each RELOAD message must be signed.
Object Level: Stored objects must be signed by the creating peer.

These three levels of security work together to allow peers to verify the origin and correctness of data they receive from other peers, even in the face of malicious activity by other peers in the overlay. RELOAD also provides access control built on top of these communications security features. Because the peer responsible for storing a piece of data can validate the signature on the data being stored, the responsible peer can determine whether a given operation is permitted or not.

RELOAD also provides an optional shared secret based admission control feature using shared secrets and TLS-PSK. In order to form a TLS connection to any node in the overlay, a new node needs to know the shared overlay key, thus restricting access to authorized users only. This feature is used together with certificate-based access control, not as a replacement for it. It is typically used when self-signed certificates are being used but would generally not be used when the certificates were all signed by an enrollment server.

1.4. Structure of This Document

The remainder of this document is structured as follows.

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

2. Terminology

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

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

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

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

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

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

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

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

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

Node-ID: A fixed-length value that uniquely identifies a node. Node-IDs of all 0s and all 1s are reserved and are invalid Node-IDs. A value of zero is not used in the wire protocol but can be used to indicate an invalid node in implementations and APIs. The Node-ID of all 1s is used on the wire protocol as a wildcard.

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

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

Resource-ID: A value that identifies some resources and which is used as a key for storing and retrieving the resource. Often this is not human friendly/readable. One way to generate a Resource-ID is by applying a mapping function to some other unique name (e.g., user name or service name) for the resource. The Resource-ID is used by the distributed database algorithm to determine the peer or peers that are responsible for storing the data for the overlay. In structured P2P networks, Resource-IDs are generally fixed length and are formed by hashing the resource name. In unstructured networks, resource names may be used directly as Resource-IDs and may be variable lengths.

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

Routing Table: The set of peers which a node can use to route overlay messages. In general, these peers will all be on the connection table but not vice versa, because some peers will have Attached but not sent updates. Peers may send messages directly to peers that are in the connection table but may only route messages to other peers through peers that are in the routing table.

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

Usage: A usage is an application that wishes to use the overlay for some purpose. Each application wishing to use the overlay defines a set of data kinds that it wishes to use. The SIP usage defines the location data kind.

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

3. Overlay Management Overview

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

3.1. Security and Identification

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

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

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

The certificate serves multiple purposes:

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

If a user has more than one device, typically they would get one certificate for each device. This allows each device to act as a separate peer.

RELOAD supports multiple certificate issuance models. The first is based on a central enrollment process which allocates a unique name and Node-ID and puts them in a certificate for the user. All peers in a particular Overlay Instance have the enrollment server as a

trust anchor and so can verify any other peer's certificate.

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

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

3.1.1. Shared-Key Security

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

3.2. Clients

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

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

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

In RELOAD, unlike some other designs, clients are not a first-class concept. From the perspective of a peer, a client is simply a node which has not yet sent any Updates or Joins. It might never do so (if it's a client) or it might eventually do so (if it's just a node that's taking a long time to join). The routing and storage rules for RELOAD provide for correct behavior by peers regardless of whether other nodes attached to them are clients or peers. Of course, a client implementation must know that it intends to be a client, but this localizes complexity only to that node.

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

3.2.1. Client Routing

Clients may insert themselves in the overlay in two ways:

- Establish a connection to the peer responsible for the client's Node-ID in the overlay. Then requests may be sent from/to the client using its Node-ID in the same manner as if it were a peer, because the responsible peer in the overlay will handle the final step of routing to the client. This may require a TURN relay in cases where NATs or firewalls prevent a client from forming a direct connections with its responsible peer. Note that clients that choose this option MUST 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 then MUST 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.

3.2.2. Minimum Functionality Requirements for Clients

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

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

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

3.3. Routing

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

RELOAD's routing capabilities must meet the following requirements:

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

Clients: RELOAD must support requests from and to clients that do not participate in overlay routing.

Client promotion: RELOAD must support clients that become peers at a later point as determined by the overlay algorithm and deployment.

Low state: RELOAD's routing algorithms must not require significant state to be stored on intermediate peers.

Return routability in unstable topologies: At some points in times, different nodes may have inconsistent information about the connectivity of the routing graph. In all cases, the response to a request needs to delivered to the node that sent the request and not to some other node.

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

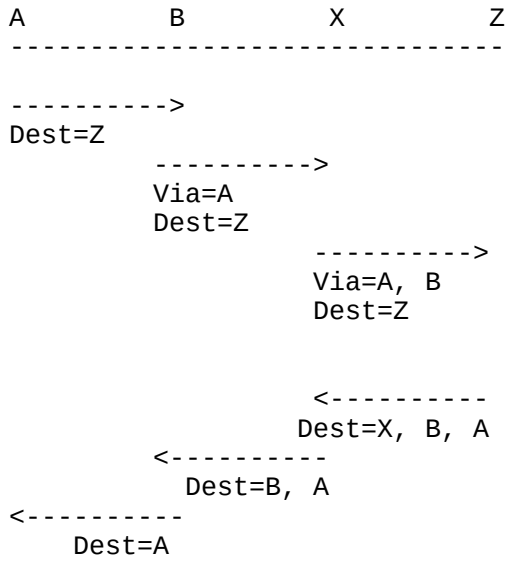
Destination Lists: While in principle it is possible to just inject a message into the overlay with a bare Node-ID as the destination, RELOAD provides a source routing capability in the form of "Destination Lists". A "Destination List provides a list of the nodes through which a message must flow.

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

RouteQuery: The RouteQuery method allows a node to query a peer for the next hop it will use to route a message. This method is useful for diagnostics and for iterative routing.

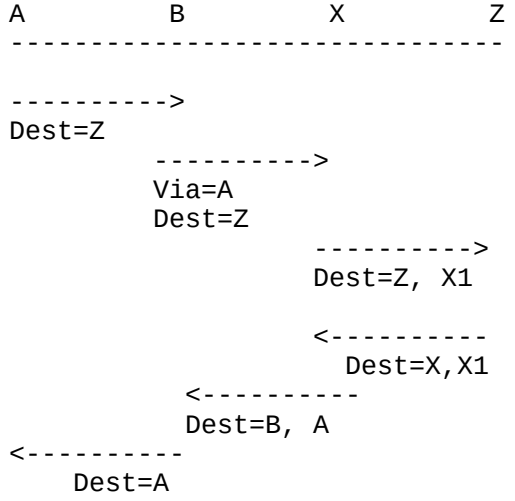
The basic routing mechanism used by RELOAD is Symmetric Recursive. We will first describe symmetric recursive routing and then discuss its advantages in terms of the requirements discussed above.

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



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

This figure shows use of full via-lists by intermediate peers B and X. However, if B and/or X are willing to store state, then they may elect to truncate the lists, save that information internally (keyed by the transaction id), and return the response message along the path from which it was received when the response is received. This option requires greater state to be stored on intermediate peers but saves a small amount of bandwidth and reduces the need for modifying the message en route. Selection of this mode of operation is a choice for the individual peer; the techniques are interoperable even on a single message. The figure below shows B using full via lists but X truncating them to X1 and saving the state internally.



RELOAD also supports a basic Iterative routing mode (where the intermediate peers merely return a response indicating the next hop, but do not actually forward the message to that next hop themselves). Iterative routing is implemented using the RouteQuery method, which requests this behavior. Note that iterative routing is selected only by the initiating node.

3.4. Connectivity Management

In order to provide efficient routing, a peer needs to maintain a set of direct connections to other peers in the Overlay Instance. Due to the presence of NATs, these connections often cannot be formed directly. Instead, we use the Attach request to establish a connection. Attach uses ICE [RFC5245] to establish the connection. It is assumed that the reader is familiar with ICE.

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

which it sends to B through usual overlay routing procedures. B does its own candidate gathering and sends back a response with its candidates. A and B then do ICE connectivity checks on the candidate pairs. The result is a connection between A and B. At this point, A and B can add each other to their routing tables and send messages directly between themselves without going through other overlay peers.

There is one special case in which Attach cannot be used: when a peer is joining the overlay and is not connected to any peers. In order to support this case, some small number of "bootstrap nodes" typically need to be publicly accessible so that new peers can directly connect to them. Section 10 contains more detail on this.

In general, a peer needs to maintain connections to all of the peers near it in the Overlay Instance and to enough other peers to have efficient routing (the details depend on the specific overlay). If a peer cannot form a connection to some other peer, this isn't necessarily a disaster; overlays can route correctly even without fully connected links. However, a peer should try to maintain the specified link set and if it detects that it has fewer direct connections, should form more as required. This also implies that peers need to periodically verify that the connected peers are still alive and if not try to reform the connection or form an alternate one.

3.5. Overlay Algorithm Support

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

3.5.1. Support for Pluggable Overlay Algorithms

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

3.5.2. Joining, Leaving, and Maintenance Overview

When a new peer wishes to join the Overlay Instance, it must have a Node-ID that it is allowed to use and a set of credentials which match that Node-ID. When an enrollment server is used that Node-ID will be in the certificate the node received from the enrollment

server. The details of the joining procedure are defined by the overlay algorithm, but the general steps for joining an Overlay Instance are:

- Forming connections to some other peers.
- Acquiring the data values this peer is responsible for storing.
- Informing the other peers which were previously responsible for that data that this peer has taken over responsibility.

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

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

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

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

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

1. JP (Joining Peer) sends a Join request to AP (Admitting Peer) announcing its intention to join.
2. AP sends a Join response.
3. AP does a sequence of Stores to JP to give it the data it will need.
4. AP does Updates to JP and to other peers to tell it about its own routing table. At this point, both JP and AP consider JP responsible for some section of the Overlay Instance.
5. JP makes its own connections to the appropriate peers in the Overlay Instance.

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

Note that the first node is a special case. When ordinary nodes cannot form connections to the bootstrap nodes, then they are not part of the overlay. However, the first node in the overlay can obviously not connect to other nodes. In order to support this case, potential first nodes (which must also serve as bootstrap nodes initially) must somehow be instructed (perhaps by configuration settings) that they are the entire overlay, rather than not part of it.

Note that clients do not perform either of these operations.

3.6. First-Time Setup

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

3.6.1. Initial Configuration

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

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

3.6.2. Enrollment

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

overlay is one of the cornerstones of RELOAD's security.

4. Application Support Overview

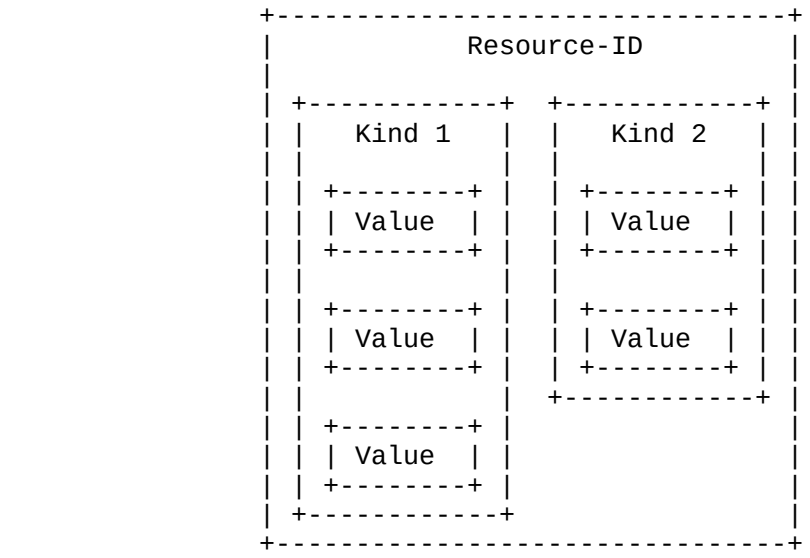
RELOAD is not intended to be used alone, but rather as a substrate for other applications. These applications can use RELOAD for a variety of purposes:

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

This section provides an overview of these services.

4.1. Data Storage

RELOAD provides operations to Store and Fetch data. Each location in the Overlay Instance is referenced by a Resource-ID. However, each location may contain data elements corresponding to multiple kinds (e.g., certificate, SIP registration). Similarly, there may be multiple elements of a given kind, as shown below:



Each kind is identified by a Kind-ID, which is a code point either assigned by IANA or allocated out of a private range. As part of the kind definition, protocol designers may define constraints, such as

limits on size, on the values which may be stored. For many kinds, the set may be restricted to a single value; some sets may be allowed to contain multiple identical items while others may only have unique items. Note that a kind may be employed by multiple usages and new usages are encouraged to use previously defined kinds where possible. We define the following data models in this document, though other usages can define their own structures:

single value: There can be at most one item in the set and any value overwrites the previous item.

array: Many values can be stored and addressed by a numeric index.

dictionary: The values stored are indexed by a key. Often this key is one of the values from the certificate of the peer sending the Store request.

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

4.1.1. Storage Permissions

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

The most natural rule is that a certificate authorizes a user to store data keyed with their user name X. This rule is used for all the kinds defined in this specification. Thus, only a user with a certificate for "alice@example.org" could write to that location in the overlay. However, other usages can define any rules they choose, including publicly writable values.

The digital signature over the data serves two purposes. First, it allows the peer responsible for storing the data to verify that this Store is authorized. Second, it provides integrity for the data.

The signature is saved along with the data value (or values) so that any reader can verify the integrity of the data. Of course, the responsible peer can "lose" the value but it cannot undetectably modify it.

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

4.1.2. Replication

Replication in P2P overlays can be used to provide:

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

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

The core RELOAD specification does not specify a particular replication strategy. Instead, the first level of replication strategies are determined by the overlay algorithm, which can base the replication strategy on its particular topology. For example, Chord places replicas on successor peers, which will take over responsibility should the responsible peer fail [Chord].

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

4.2. Usages

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

- o Registers Kind-ID code points for any kinds that the Usage defines.
- o Defines the data structure for each of the kinds.
- o Defines access control rules for each of the kinds.
- o Defines how the Resource Name is formed that is hashed to form the Resource-ID where each kind is stored.
- o Describes how values will be merged after a network partition. Unless otherwise specified, the default merging rule is to act as if all the values that need to be merged were stored and 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.maenpaa-p2psip-service-discovery].

4.4. Application Connectivity

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

For more common situations, however, it is the overlay itself - rather than an external authority such as DNS - which is used to establish a connection. RELOAD provides connectivity to applications using the AppAttach method. For example, if a P2PSIP node wishes to establish a SIP dialog with another P2PSIP node, it will use AppAttach to establish a direct connection with the other node. This new connection is separate from the peer protocol connection. It is

a dedicated UDP or TCP flow used only for the SIP dialog.

5. Overlay Management Protocol

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

5.1. Message Receipt and Forwarding

When a peer receives a message, it first examines the overlay, version, and other header fields to determine whether the message is one it can process. If any of these are incorrect (e.g., the message is for an overlay in which the peer does not participate) it is an error. The peer SHOULD generate an appropriate error but local policy can override this and cause the messages to be silently dropped.

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

- The first entry on the destination list is an ID for which the peer is responsible.
- The first entry on the destination list is an ID for which another peer is responsible.
- The first entry on the destination list is a private ID that is being used for destination list compression. This is described later (note that private IDs can be distinguished from Node-IDs and Resource-IDs on the wire; see 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 node is responsible, there are several 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 passes it up to the upper layers.

- o If the entry is a Node-ID which equals this node's Node-ID, then the message is destined for this node. If this is the only entry on the destination list, the message is destined for this node and is passed up to the upper layers. Otherwise the entry is removed from the destination list and the message is passed to the Message Transport. If the message is a response and there is state for the transaction ID, the state is reinserted into the destination list before the message is further processed.
- o If the entry is a Node-ID which is not equal to this node, then the node MUST drop the message silently unless the Node-ID corresponds to a node which is directly connected to this node (i.e., a client). In that case, it MUST forward the message to the destination node as described in the next section.

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

5.1.2. Other ID

If neither of the other three cases applies, then the peer MUST forward the message towards the first entry on the destination list. This means that it MUST select one of the peers to which it is connected and which is likely to be responsible for the first entry on the destination list. If the first entry on the destination list is in the peer's connection table, then it SHOULD forward the message to that peer directly. Otherwise, the peer consults the routing table to forward the message.

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

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

As an example of the first strategy, if node D receives a message from node C with via list (A, B), then D would forward to the next node (E) with via list (A, B, C). Now, if E wants to respond to the message, it reverses the via list to produce the destination list, resulting in (D, C, B, A). When D forwards the response to C, the destination list will contain (C, B, A).

As an example of the second strategy, if node D receives a message from node C with transaction ID X and via list (A, B), it could store

(X, C) in its state database and forward the message with the via list unchanged. When D receives the response, it consults its state database for transaction id X, determines that the request came from C, and forwards the response to C.

Intermediate peers which modify the via list are not required to simply add entries. The only requirement is that the peer be able to reconstruct the correct destination list on the return route. RELOAD provides explicit support for this functionality in the form of private IDs, which can replace any number of via list entries. For instance, in the above example, Node D might send E a via list containing only the private ID (I). E would then use the destination list (D, I) to send its return message. When D processes this destination list, it would detect that I is a private ID, recover the via list (A, B, C), and reverse that to produce the correct destination list (C, B, A) before sending it to C. This feature is called List Compression. It MAY either be a compressed version of the original via list or an index into a state database containing the original via list.

No matter what mechanism for storing via list state is used, if an intermediate peer exits the overlay, then on the return trip the message cannot be forwarded and will be dropped. The ordinary timeout and retransmission mechanisms provide stability over this type of failure.

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

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

5.1.3. Private ID

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

5.2. Symmetric Recursive Routing

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

5.2.1. Request Origination

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

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

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

Because messages may be lost in transit through the overlay, RELOAD incorporates an end-to-end reliability mechanism. When an originating node transmits a request it MUST set a 3 second timer. If a response has not been received when the timer fires, the request is retransmitted with the same transaction identifier. The request MAY be retransmitted up to 4 times (for a total of 5 messages). After the timer for the fifth transmission fires, the message SHALL be considered to have failed. Note that this retransmission procedure is not followed by intermediate nodes. They follow the hop-by-hop reliability procedure described in Section 5.6.3.

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

maximum request lifetime (15 seconds). For some requests (e.g., Fetch) this can be accomplished merely by processing the request again. For other requests, (e.g., Store) it may be necessary to maintain state for the duration of the request lifetime.

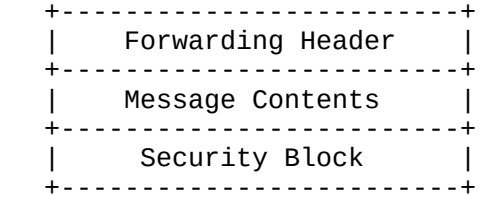
5.2.2. Response Origination

When a peer sends a response to a request using this routing algorithm, it MUST construct the destination list by reversing the order of the entries on the via list. This has the result that the response traverses the same peers as the request traversed, except in reverse order (symmetric routing).

5.3. Message Structure

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

Each message has three parts, concatenated as shown below:



The contents of these parts are as follows:

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

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

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

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

5.3.1. Presentation Language

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

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

Several idiosyncrasies of this language are worth noting.

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

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

5.3.1.1. Common Definitions

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

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

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

A boolean value is either a 1 or a 0. The max value of 255 indicates this is represented as a single byte on the wire.

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

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

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

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

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

Like a NodeId, a ResourceId is an opaque string of bytes, but unlike NodeIds, ResourceIds are variable length, up to 254 bytes (2040 bits) in length. On the wire, each ResourceId is preceded by a single length byte (allowing lengths up to 255). Thus, the 3-byte value "FOO" would be encoded as: 03 46 4f 4f. Note the < range > syntax defines a variable length element that does include the length of the element in the on the wire encoding. The number of bytes to encode the length on the wire is derived by range; i.e., it is the minimum number of bytes which can encode the largest range value.

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

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

struct  {
  uint32                  addr;
  uint16                  port;
} IPv4AddrPort;

struct  {
  uint128                 addr;
  uint16                  port;
} IPv6AddrPort;


struct {
  AddressType             type;
  uint8                   length;

  select (type) {
    case ipv4_address:
       IPv4AddrPort       v4addr_port;

    case ipv6_address:
       IPv6AddrPort       v6addr_port;

    /* This structure can be extended */
  };
} IpAddressPort;
```

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

type: the type of address (v4 or v6).
length: the length of the rest of the structure.

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

The rest of the IpAddressPort structure is either an IPv4AddrPort or an IPv6AddrPort. Both of these simply consist of an address represented as an integer and a 16-bit port. As an example, here is the wire representation of the IPv4 address "192.0.2.1" with port "6100".

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

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

5.3.2. Forwarding Header

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

```
     struct {
       uint32             relo_token;
       uint32             overlay;
       uint16             configuration_sequence;
       uint8              version;
       uint8              ttl;
       uint32             fragment;
       uint32             length;
       uint64             transaction_id;
       uint32             max_response_length;
       uint16             via_list_length;
       uint16             destination_list_length;
       uint16             options_length;
       Destination        via_list[via_list_length];
       Destination        destination_list
                            [destination_list_length];
       ForwardingOptions  options[options_length];
     } ForwardingHeader;
```

The contents of the structure are:

relo_token: The first four bytes identify this message as a RELOAD message. This field MUST contain the value 0xd2454c4f (the string 'RELO' with the high bit of the first byte set.).

overlay: The 32 bit checksum/hash of the overlay being used. The variable length string representing the overlay name is hashed with SHA-1 [RFC3174] and the low order 32 bits are used. The purpose of this field is to allow nodes to participate in multiple overlays and to detect accidental misconfiguration. This is not a security critical function.

configuration_sequence: The sequence number of the configuration file.

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

ttl: An 8 bit field indicating the number of iterations, or hops, a message can experience before it is discarded. The TTL value MUST be decremented by one at every hop along the route the message traverses. If the TTL is 0, the message MUST NOT be propagated further and MUST be discarded, and a "Error_TTL_Exceeded" error should be generated. The initial value of the TTL SHOULD be 100 unless defined otherwise by the overlay configuration.

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

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

transaction_id: A unique 64 bit number that identifies this transaction and also allows receivers to disambiguate transactions which are otherwise identical. In order to provide a high probability that transaction IDs are unique, they MUST be randomly generated. Responses use the same Transaction ID as the request they correspond to. Transaction IDs are also used for fragment reassembly.

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

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

destination_list_length: The length of the destination list in bytes.

options_length: The length of the header options in bytes.

via_list: The via_list contains the sequence of destinations through which the message has passed. The via_list starts out empty and grows as the message traverses each peer.

destination_list: The destination_list contains a sequence of destinations which the message should pass through. The destination list is constructed by the message originator. The first element in the destination list is where the message goes next. The list shrinks as the message traverses each listed peer.

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

5.3.2.1. Processing Configuration Sequence Numbers

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

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

5.3.2.2. Destination and Via Lists

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

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

select (destination_type) {
 case node:
        NodeId               node_id;

 case resource:
        ResourceId           resource_id;

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

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

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

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

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

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

length
The length of the destination_data.

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

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

A DestinationData can be one of three types:

node
A Node-ID.

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

resource
The Resource-ID of the resource which is desired. This type MUST only appear in the final location of a destination list and MUST NOT appear in a via list. It is meaningless to try to route through a resource.

One possible encoding of the 16 bit integer version as an opaque identifier is to encode an index into a connection table. To avoid misrouting responses in the event a response is delayed and the connection table entry has changed, the identifier SHOULD be split between an index and a generation counter for that index. At startup, the generation counters should be initialized to random values. An implementation could use 12 bits for the connection table index and 3 bits for the generation counter. (Note that this does not suggest a 4096 entry connection table for every node, only the ability to encode for a larger connection table.) When a connection table slot is used for a new connection, the generation counter is incremented (with wrapping). Connection table slots are used on a rotating basis to maximize the time interval between uses of the same slot for different connections. When routing a message to an entry in the destination list encoding a connection table entry, the node confirms that the generation counter matches the current generation counter of that index before forwarding the message. If it does not

match, the message is silently dropped.

5.3.2.3. Forwarding Options

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

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

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

Each ForwardingOption consists of the following values:

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

length
: The length of the rest of the structure.

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

option
The option value.

5.3.3. Message Contents Format

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

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

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

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

The contents of this structure are as follows:

message_code
This indicates the message that is being sent. The code space is broken up as follows.

0 Reserved

1 .. 0x7fff Requests and responses. These code points are always paired, with requests being odd and the corresponding response being the request code plus 1. Thus, "probe_request" (the Probe request) has value 1 and "probe_answer" (the Probe response) has value 2

0xffff Error
The message codes are defined in Section 13.8

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

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

All extensions have the following form:

type
The extension type.

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

extension_contents
The contents of the extension (extension-dependent).

5.3.3.1. Response Codes and Response Errors

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

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

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

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

The contents of this structure are as follows:

error_code
A numeric error code indicating the error that occurred.

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

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

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

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

Error_Request_Timeout: A response to the request has not been received in a suitable amount of time. The requesting node MAY resend the request at a later time.

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

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

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

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

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

Error_Unsupported_Forwarding_Option: A peer receiving the request with a forwarding options flagged as critical but the peer does not support this option. See section Section 5.3.2.3.

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

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

Error_Response_Too_Large: A peer would have generated a response that is too large per the max_response_length field.

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

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

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

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

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

5.3.4. Security Block

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

```
enum { x509(0), (255) } certificate_type;

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

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

The contents of this structure are:

certificates
: A bucket of certificates.

signature
A signature over the message contents.

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

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

type
The type of the certificate. Only one type is defined: x509 representing an X.509 certificate.

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

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

```
enum { reservedSignerIdentity(0),
       cert_hash(1),  (255)} SignerIdentityType;

struct {
  select (identity_type) {
    case cert_hash;
      HashAlgorithm      hash_alg;              // From TLS
      opaque             certificate_hash<0..2^8-1>;
    /* 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. 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.

The only currently permitted identity format is a hash of the signer's certificate. 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). The SignerIdentity structure is typed purely to allow for future (unanticipated) extensibility.

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

```
overlay || transaction_id || MessageContents || SignerIdentity
```

where overlay and transaction_id come from the forwarding header and || indicates concatenation.

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

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

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

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

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

5.4. Overlay Topology

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

5.4.1. Topology Plugin Requirements

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

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

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

5.4.2. Methods and types for use by topology plugins

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

5.4.2.1. Join

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

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

The minimal JoinReq contains only the Node-ID which the sending peer wishes to assume. Overlay algorithms MAY specify other data to appear in this request. Receivers of the JoinReq MUST verify that the joining_peer_id field matches the Node-ID used to sign the message and if not MUST reject the message with an Error_Forbidden error.

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

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

If the request succeeds, the responding peer MUST follow up by executing the right sequence of Stores and Updates to transfer the appropriate section of the overlay space to the joining peer. In addition, overlay algorithms MAY define data to appear in the response payload that provides additional info.

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

5.4.2.2. Leave

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

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

LeaveReq contains only the Node-ID of the leaving peer. Overlay algorithms MAY specify other data to appear in this request. Receivers of the LeaveReq MUST verify that the leaving_peer_id field matches the Node-ID used to sign the message and if not MUST reject the message with an Error_Forbidden error.

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

5.4.2.3. Update

Update is the primary overlay-specific maintenance message. It is used by the sender to notify the recipient of the sender’s view of the current state of the overlay (its routing state), and it is up to the recipient to take whatever actions are appropriate to deal with the state change. In general, peers send Update messages to all their adjacencies whenever they detect a topology shift.

When a peer receives an Attach request with the send_update flag set to "true" Section 5.4.2.4, it MUST send an Update message back to the sender of the Attach request after the completion of the corresponding ICE check and TLS connection. Note that the sender of a such Attach request may not have joined the overlay yet.

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

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

5.4.2.4. RouteQuery

The RouteQuery request allows the sender to ask a peer where they would route a message directed to a given destination. In other words, a RouteQuery for a destination X requests the Node-ID for the node that the receiving peer would next route to in order to get to X. A RouteQuery can also request that the receiving peer initiate an Update request to transfer the receiving peer’s routing table.

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

5.4.2.4.1. Request Definition

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

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

```
        } RouteQueryReq;
```

The contents of the RouteQueryReq message are as follows:

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

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

overlay_specific_data
: Other data as appropriate for the overlay.

5.4.2.4.2. Response Definition

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

5.4.2.5. Probe

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

5.4.2.5.1. Request Definition

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

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

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

The currently defined values for ProbeInformation are:

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

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

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

5.4.2.5.2. Response Definition

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

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

          case num_resources:
            uint32             num_resources;

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

        };
      } ProbeInformationData;

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

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

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

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

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

5.5. Forwarding and Link Management Layer

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

Attach: used to form RELOAD connections between nodes. 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.

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

An Attach in and of itself does not result in updating the routing table of either node. That function is performed by Updates. If node A has Attached to node B, but not received any Updates from B, it MAY route messages which are directly addressed to B through that channel but MUST NOT route messages through B to other peers via that channel. The process of Attaching is separate from the process of becoming a peer (using Join and Update), to prevent half-open states where a node has started to form connections but is not really ready to act as a peer. Thus, clients (unlike peers) can simply Attach without sending Join or Update.

5.5.1.1. Request Definition

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

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

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

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

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

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

The values contained in AttachReqAns are:

ufrag
: The username fragment (from ICE).

password

The ICE password.

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

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

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

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

addr_port
corresponds to the connection-address and port productions.

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

foundation
corresponds to the foundation production.

priority
corresponds to the priority production.

type
corresponds to the cand-type production.

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

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

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

5.5.1.2. Response Definition

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

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

An agent follows the ICE specification as described in [RFC5245] with the changes and additional procedures described in the subsections below.

5.5.1.4. Collecting STUN Servers

ICE relies on the node having one or more STUN servers to use. In conventional ICE, it is assumed that nodes are configured with one or more STUN servers through some out of band mechanism. This is still possible in RELOAD but RELOAD also learns STUN servers as it connects to other peers. Because all RELOAD peers implement ICE and use STUN keepalives, every peer is a capable of responding to STUN Binding requests [RFC5389]. Accordingly, any peer that a node knows about can be used like a STUN server -- though of course it may be behind a NAT.

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

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

Only peers to which the peer currently has connections may be used.

If the connection to that host is lost, it MUST be removed from the list of stun servers and a new server from the same group MUST be selected unless there are no others servers in the group in which case some other peer MAY be used.

5.5.1.5. Gathering Candidates

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

The ICE specification assumes that an ICE agent is configured with, or somehow knows of, TURN and STUN servers. RELOAD provides a way for an agent to learn these by querying the overlay, as described in Section 5.5.1.4 and Section 8.

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

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

No-ICE will not work in all of the scenarios where ICE would work, but in some cases, particularly those with no NATs or firewalls, it will work. Therefore it is RECOMMENDED that full ICE be used even for a node that has a public, unfiltered IP address, to take advantage of STUN connectivity checks, etc.

5.5.1.6. Prioritizing Candidates

At the time of writing, UDP is the only transport protocol specified to work with ICE. However, standardization of additional protocols for use with ICE is expected, including TCP and datagram-oriented protocols such as SCTP and DCCP. In particular, UDP encapsulations for SCTP and DCCP are expected to be standardized in the near future, greatly expanding the available Overlay Link protocols available for RELOAD. When additional protocols are available, the following prioritization is RECOMMENDED:

- o Highest priority is assigned to message-oriented 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.
- o Second highest priority is assigned to stream-oriented protocols, e.g. TCP.
- o Lowest priority is assigned to protocols encapsulated over UDP that do not implement well-established congestion control algorithms. The DTLS/UDP with SR overlay link protocol is an example of such a protocol.

5.5.1.7. Encoding the Attach Message

Section 4.3 of ICE describes procedures for encoding the SDP for conveying RELOAD candidates. Instead of actually encoding an SDP, the candidate information (IP address and port and transport protocol, priority, foundation, type and related address) is carried within the attributes of the Attach request or its response. Similarly, the username fragment and password are carried in the Attach message or its response. Section 5.5.1 describes the detailed attribute encoding for Attach. The Attach request and its response do not contain any default candidates or the ice-lite attribute, as these features of ICE are not used by RELOAD.

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

5.5.1.8. Verifying ICE Support

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

5.5.1.9. Role Determination

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

5.5.1.10. Full ICE

When neither side has provided an No-ICE candidate, connectivity checks and nominations are used as in regular ICE.

5.5.1.10.1. Connectivity Checks

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

5.5.1.10.2. Concluding ICE

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

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

5.5.1.10.3. Media Keepalives

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

5.5.1.11. No-ICE

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

5.5.1.12. Subsequent Offers and Answers

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

5.5.1.13. Sending Media

The procedures of Section 11 of ICE apply to RELOAD as well. However, in this case, the "media" takes the form of application layer protocols (RELOAD) over TLS or DTLS. Consequently, once ICE

processing completes, the agent will begin TLS or DTLS procedures to establish a secure connection. The node which sent the Attach request MUST be the TLS server. The other node MUST be the TLS client. The server MUST request TLS client authentication. The nodes MUST verify that the certificate presented in the handshake matches the identity of the other peer as found in the Attach message. Once the TLS or DTLS signaling is complete, the application protocol is free to use the connection.

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

5.5.1.14. Receiving Media

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

5.5.2. AppAttach

A node sends an AppAttach request when it wishes to establish a direct connection to another node for the purposes of sending application layer messages. AppAttach is nearly identical to Attach, except for the purpose of the connection: it is used to transport non-RELOAD "media". A separate request is used to avoid implementor confusion between the two methods (this was found to be a real problem with initial implementations). The AppAttach request and its response contain an application attribute, which indicates what protocol is to be run over the connection.

5.5.2.1. Request Definition

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

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

The values contained in AppAttachReq and AppAttachAns are:

ufrag
The username fragment (from ICE)

password
The ICE password.

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

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

candidates
One or more ICE candidate values

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

5.5.2.2. Response Definition

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

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

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

5.5.3. Ping

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

(2^128-1).

5.5.3.1. Request Definition

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

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

5.5.3.2. Response Definition

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

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

A PingAns message contains the following elements:

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

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

5.5.4. ConfigUpdate

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

5.5.4.1. Request Definition

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

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

struct {
  ConfigUpdateType       type;
  uint32                 length;

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

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

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

The ConfigUpdateReq message contains the following elements:

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

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

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

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

5.5.4.2. Response Definition

```
struct {
} ConfigUpdateAns
```

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

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

Otherwise appropriate errors MUST be generated.

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

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

The response for ConfigUpdate is empty.

5.6. Overlay Link Layer

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

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

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

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

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

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

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

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

5.6.1. Future Overlay Link Protocols

The only currently defined overlay link protocols are TLS and DTLS. 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) [I-D.ietf-hip-bone] 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] should allow 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. Baset et al. have proposed tunneling TCP over UDP for similar reasons [I-D.baset-tsvwg-tcp-over-udp]. Once ICE traversal has been specified for these tunneled protocols, they should be straightforward to support as overlay link protocols.

5.6.2. Framing Header

In order to support unreliable links and to allow for quick detection of link failures when using reliable end-to-end transports, each message is wrapped in a very simple framing layer (FramedMessage) which is only used for each hop. This layer contains a sequence number which can then be used for ACKs. The same header is used for both reliable and unreliable transports for simplicity of implementation.

The definition of FramedMessage is:

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

struct {
  FramedMessageType       type;

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

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

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

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

sequence

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

message
the message that is being transmitted.

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

When the receiver receives a message, it SHOULD immediately send an ACK message. The receiver MUST keep track of the 32 most recent sequence numbers received on this association in order to generate the appropriate ack.

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

ack_sequence
The sequence number of the message being acknowledged.

received
A bitmask indicating if each of the previous 32 sequence numbers before this packet has been among the 32 packets most recently received on this connection. When a packet is received with a sequence number N, the receiver looks at the sequence number of the previously 32 packets received on this connection. Call the previously received packet number M. For each of the previous 32 packets, if the sequence number M is less than N but greater than N-32, the N-M bit of the received bitmask is set to one; otherwise it is zero. Note that a bit being set to one indicates positively that a particular packet was received, but a bit being set to zero means only that it is unknown whether or not the packet has been received, because it might have been received before the 32 most recently received packets.

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

5.6.3. Simple Reliability

When RELOAD is carried over DTLS or another unreliable link protocol, it needs to be used with a reliability and congestion control mechanism, which is provided on a hop-by-hop basis. The basic principle is that each message, regardless of whether or not it carries a request or response, will get an ACK and be reliably

retransmitted. The receiver's job is very simple, limited to just sending ACKs. All the complexity is at the sender side. This allows the sending implementation to trade off performance versus implementation complexity without affecting the wire protocol.

5.6.3.1. Retransmission and Flow Control

Because the receiver's role is limited to providing packet acknowledgements, a wide variety of congestion control algorithms can be implemented on the sender side while using the same basic wire protocol. In general, senders MAY implement any rate control scheme of their choice, provided that it is REQUIRED to be no more aggressive then TFRC[RFC5348].

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

5.6.3.1.1. Trivial Retransmission

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

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

Implementations that use a dynamic estimate to compute the RTO MUST use the algorithm described in RFC 2988[RFC2988], 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 2988. 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 2988 to update the RTO estimate. (Note that because retransmissions receive new sequence numbers, all received ACKs are used.)

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

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

To determine when a link may be failing without waiting for the final timeout, observe when no ACKs have been received for an entire RTO interval, and then wait for three retransmissions to occur beyond that point. If no ACKs have been received by the time the third retransmission occurs, it is RECOMMENDED that the link be removed from the routing table. The link MAY be restored to the routing table if ACKs resume before the connection is closed, as described above.

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

5.6.4. DTLS/UDP with SR

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

5.6.5. TLS/TCP with FH, No-ICE

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

Because the TCP layer’s application-level timeout is too slow to be useful for overlay routing, the Overlay Link implementation MUST use the framing header to measure the RTT of the connection and calculate an RTO as specified in Section 2 of [RFC2988]. The resulting RTO is not used for retransmissions, but as a timeout to indicate when the link SHOULD be removed from the routing table. It is RECOMMENDED that such a connection be retained for 30s to determine if the failure was transient before concluding the link has failed

permanently.

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

5.6.6. DTLS/UDP with SR, No-ICE

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

5.7. Fragmentation and Reassembly

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

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

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

Upon receipt of a fragmented message by the intended peer, the peer holds the fragments in a holding buffer until the entire message has been received. The message is then reassembled into a single message and processed. In order to mitigate denial of service attacks, receivers SHOULD time out incomplete fragments after maximum request lifetime (15 seconds). Note this time was derived from looking at the end to end retransmission time and saving fragments long enough

for the full end to end retransmissions to take place. Ideally the receiver would have enough buffer space to deal with as many fragments as can arrive in the maximum request lifetime. However, if the receiver runs out of buffer space to reassemble the messages it MUST drop the message.

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

6. Data Storage Protocol

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

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

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

The contents of this structure are as follows:

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

storage_time
The time when the data was stored represented as the number of milliseconds elapsed since midnight Jan 1, 1970 UTC not counting leap seconds. This will have the same values for seconds as standard UNIX time or POSIX time. More information can be found at [UnixTime]. Any attempt to store a data value with a storage time before that of a value already stored at this location MUST generate a Error_Data_Too_Old error. This prevents rollback attacks. Note that this does not require synchronized clocks: the receiving peer uses the storage time in the previous store, not its own clock.
A node that is 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 of store.

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

signature
A signature as defined in Section 6.1.

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

6.1. Data Signature Computation

Each StoredData element is individually signed. However, the signature also must be self-contained and cover the Kind-ID and Resource-ID even though they are not present in the StoredData structure. The input to the signature algorithm is:

```
resource_id || kind || storage_time || StoredDataValue ||
SignerIdentity
```

Where || indicates concatenation.

Where these values are:

resource_id
: The resource ID where this data is stored.

kind
: The Kind-ID for this data.

storage_time

: The contents of the storage_time data value.

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

SignerIdentity
: The signer identity as defined in Section 5.3.4.

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

6.2. Data Models

The protocol currently defines the following data models:

- single value
- array
- dictionary

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

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

struct {
  select (dataModel) {
    case single_value:
      DataValue             single_value_entry;

    case array:
      ArrayEntry            array_entry;

    case dictionary:
      DictionaryEntry       dictionary_entry;

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

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

6.2.1. Single Value

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

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

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

value
: The stored data.

6.2.2. Array

An array is a set of opaque values addressed by an integer index. Arrays are zero based. Note that arrays can be sparse. For instance, a Store of "X" at index 2 in an empty array produces an array with the values [NA, NA, "X"]. Future attempts to fetch elements at index 0 or 1 will return values with "exists" set to False.

A array element is represented as an ArrayEntry:

```
struct {
  uint32                index;
  DataValue             value;
} ArrayEntry;
```

The contents of this structure are:

index
: The index of the data element in the array.

value
: The stored data.

6.2.3. Dictionary

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

A dictionary element is represented as a DictionaryEntry:

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

struct {
  DictionaryKey         key;
  DataValue             value;
} DictionaryEntry;
```

The contents of this structure are:

key
: The dictionary key for this value.

value
: The stored data.

6.3. Access Control Policies

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

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

6.3.1. USER-MATCH

In the USER-MATCH policy, a given value MUST be written (or overwritten) if and only if the request is signed with a key associated with a certificate whose user name 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 request is signed with a key associated with a certificate whose Node-ID hashes (using the hash function for the overlay) to the Resource-ID for the resource.

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 request is signed with a key associated with a certificate whose user name 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.

6.3.4. NODE-MULTIPLE

In the NODE-MULTIPLE policy, a given value MUST be written (or overwritten) if and only if the request is signed with a key associated with a certificate containing a Node-ID such that H(Node-ID || i) is equal to the Resource-ID for some small integer value of i. When this policy is in use, the maximum value of i MUST be specified in the kind definition.

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

6.4. Data Storage Methods

RELOAD provides several methods for storing and retrieving data:

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

These methods are each described in the following sections.

6.4.1. Store

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

6.4.1.1. Request Definition

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

The store request is defined by the StoreReq structure:

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

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

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

resource
The resource to store at.

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

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

If the replica number is zero, then the peer MUST check that it is responsible for the resource and, if not, reject the request. If the replica number is nonzero, then the peer MUST check that it expects to be a replica for the resource and that the request sender is consistent with being the responsible node (i.e., that the receiving peer does not know of a better node) and, if not, reject the request.

Each StoreKindData element represents the data to be stored for a single Kind-ID. The contents of the element are:

kind
The Kind-ID. Implementations MUST reject requests corresponding to unknown kinds.

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

values
The value or values to be stored. This may contain one or more stored_data values depending on the data model associated with each kind.

The peer MUST perform the following checks:

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

- For original (non-replica) stores, the peer MUST check that if the generation counter is non-zero, it equals the current value of the generation counter for this kind. This feature allows the generation counter to be used in a way similar to the HTTP Etag feature.
- For replica Stores, the peer MUST set the generation counter to match the generation counter in the message, and MUST NOT check the generation counter against the current value. Replica Stores MUST NOT use a generation counter of 0.
- The storage time values are greater than that of any value which would be replaced by this Store.
- The size and number of the stored values is consistent with the limits specified in the overlay configuration.

If all these checks succeed, the peer MUST attempt to store the data values. For non-replica stores, if the store succeeds and the data is changed, then the peer must increase the generation counter by at least one. If there are multiple stored values in a single StoreKindData, it is permissible for the peer to increase the generation counter by only 1 for the entire Kind-ID, or by 1 or more than one for each value. Accordingly, all stored data values must have a generation counter of 1 or greater. 0 is used in the Store request to indicate that the generation counter should be ignored for processing this request; however the responsible peer should increase the stored generation counter and should return the correct generation counter in the response.

When a peer stores data previously stored by another node (e.g., for replicas or topology shifts) it MUST adjust the lifetime value downward to reflect the amount of time the value was stored at the peer. The adjustment SHOULD be implemented by an algorithm equivalent to the following: at the time the peer initially receives the StoreReq it notes the local time T. When it then attempts to do a StoreReq to another node it should decrement the lifetime value by the difference between the current local time and T.

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

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

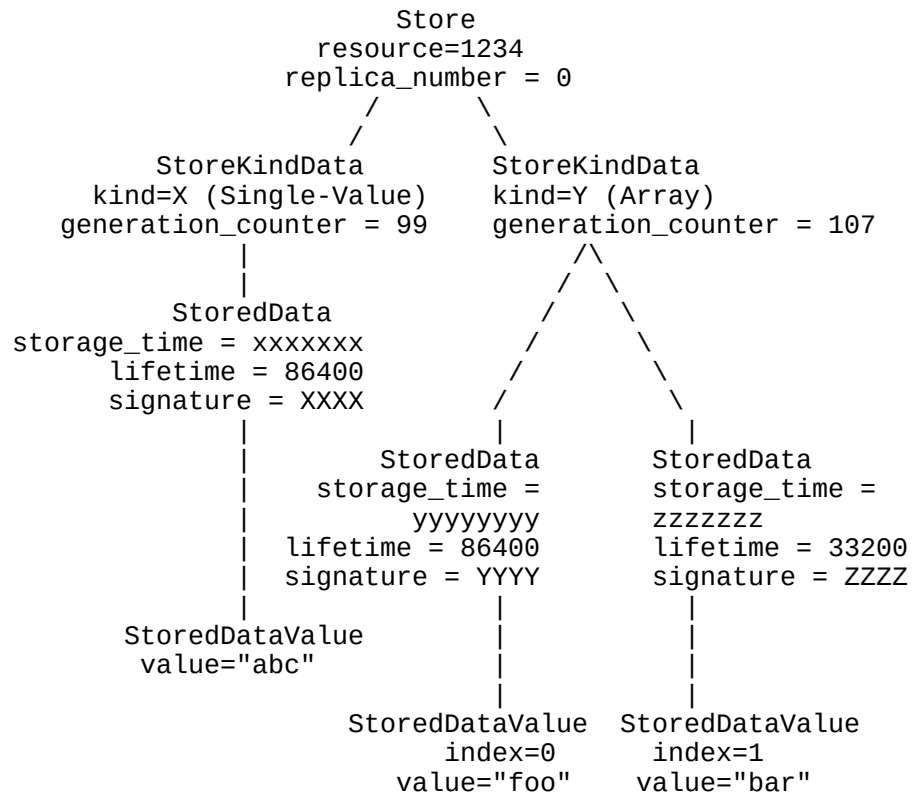
Single-value:
A store of a new single-value element creates the element if it does not exist and overwrites any existing value with the new value.

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

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

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

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



6.4.1.2. Response Definition

In response to a successful Store request the peer MUST return a StoreAns message containing a series of StoreKindResponse elements containing the current value of the generation counter for each Kind-ID, as well as a list of the peers where the data will be replicated by the node processing the request.

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


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

The contents of each StoreKindResponse are:

kind
The Kind-ID being represented.

generation_counter
The current value of the generation counter for that Kind-ID.

replicas
The list of other peers at which the data was/will be replicated. In overlays and applications where the responsible peer is intended to store redundant copies, this allows the storing peer to independently verify that the replicas have in fact been stored. It does this verification by using the Stat method. Note that the storing peer is not required to perform this verification.

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

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

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

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

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

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

6.4.1.3. Removing Values

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

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

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, this may create an implicit, unsigned "nonexistent" value to represent a gap in the data structure. However, this value isn't persistent nor is it replicated. It is simply synthesized by the storing node.

6.4.2. Fetch

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

6.4.2.1. Request Definition

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

        struct {
          KindId                  kind;
          uint64                  generation;
          uint16                  length;

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

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

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

            /* This structure may be extended */

          } model_specifier;
        } StoredDataSpecifier;

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

The contents of the Fetch requests are as follows:

resource
: The Resource-ID to fetch from.

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

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

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

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

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

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

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

The model_specifier is as follows:

- If the data model is single value, the specifier is empty.
- If the data model is array, the specifier contains a list of ArrayRange elements, each of which contains two integers. The first integer is the beginning of the range and the second is the end of the range. 0 is used to indicate the first element and 0xffffffff is used to indicate the final element. The first integer must be less than the second. While multiple ranges MAY be specified, they MUST NOT overlap.
- If the data model is dictionary then the specifier contains a list of the dictionary keys being requested. If no keys are specified, than this is a wildcard fetch and all key-value pairs are returned.

The generation counter is used to indicate the requester's expected state of the storing peer. If the generation counter in the request matches the stored counter, then the storing peer returns a response with no StoredData values.

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

6.4.2.2. Response Definition

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

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

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

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

The contents of the FetchKindResponse structure are as follows:

kind
: the kind that this structure is for.

generation
: the generation counter for this kind.

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

There is one subtle point about signature computation on arrays. If the storing node uses the append feature (where the index=0xffffffff), then the index in the StoredData that is returned will not match that used by the storing node, which would break the signature. In order to avoid this issue, the index value in the array is set to zero before the signature is computed. This implies that malicious storing nodes can reorder array entries without being detected.

6.4.3. Stat

The Stat request is used to get metadata (length, generation counter, digest, etc.) for a stored element without retrieving the element

itself. The name is from the UNIX stat(2) system call which performs a similar function for files in a file system. It also allows the requesting node to get a list of matching elements without requesting the entire element.

6.4.3.1. Request Definition

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

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

6.4.3.2. Response Definition

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

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

        struct {
          uint32                index;
          MetaData              value;
        } ArrayEntryMeta;

        struct {
          DictionaryKey         key;
          MetaData              value;
        } DictionaryEntryMeta;

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

            case array:
              ArrayEntryMeta        array_entry;

            case dictionary:
              DictionaryEntryMeta   dictionary_entry;

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

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

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

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

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

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

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

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

exists
: Same as in DataValue

value_length
: The length of the stored value.

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

hash_value
: A digest of the value using hash_algorithm.

6.4.4. Find

The Find request can be used to explore the Overlay Instance. A Find request for a Resource-ID R and a Kind-ID T retrieves the Resource-ID (if any) of the resource of kind T known to the target peer which is closest to R. This method can be used to walk the Overlay Instance by iteratively fetching R_n+1=nearest(1 + R_n).

6.4.4.1. Request Definition

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

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

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

resource
The desired Resource-ID

kinds
The desired Kind-IDs. Each value MUST only appear once, and if not the request MUST be rejected with an error.

6.4.4.2. Response Definition

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

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

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

If the processing peer is not responsible for the specified Resource-ID, it SHOULD return an Error_Not_Found error code.

For each Kind-ID in the request the response MUST contain a FindKindData indicating the closest Resource-ID for that Kind-ID, unless the kind is not allowed to be used with Find in which case a FindKindData for that Kind-ID MUST NOT be included in the response. If a Kind-ID is not known, then the corresponding Resource-ID MUST be 0. Note that different Kind-IDs may have different closest Resource-IDs.

The response is simply a series of FindKindData elements, one per kind, concatenated end-to-end. The contents of each element are:

kind
The Kind-ID.

closest
The closest resource ID to the specified resource ID. This is 0 if no resource ID is known.

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

6.4.5. Defining New Kinds

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

However a kind is defined, the definition must include:

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

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

While each kind needs to define what data model is used for its data, that does not mean that it must define new data models. Where practical, kinds should use the existing data models. The intention is that the basic data model set be sufficient for most applications/ usages.

7. Certificate Store Usage

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

A user/peer MUST store its certificate at Resource-IDs derived from two Resource Names:

- The user name in the certificate.

o The Node-ID in the certificate.

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

A peer MUST ensure that the user's certificates are stored in the Overlay Instance. New certificates are stored at the end of the list. This structure allows users to store an old and a new certificate that both have the same Node-ID, which allows for migration of certificates when they are renewed.

This usage defines the following kinds:

Name: CERTIFICATE_BY_NODE

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

Access Control: NODE-MATCH.

Name: CERTIFICATE_BY_USER

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

Access Control: USER-MATCH.

8. TURN Server Usage

The TURN server usage allows a RELOAD peer to advertise that it is prepared to be a TURN server as defined in [RFC5766]. When a node starts up, it joins the overlay network and forms several connections in the process. If the ICE stage in any of these connections returns a reflexive address that is not the same as the peer's perceived address, then the peer is behind a NAT and not a candidate for a TURN server. Additionally, if the peer's IP address is in the private address space range, then it is also not a candidate for a TURN server. Otherwise, the peer SHOULD assume it is a potential TURN server and follow the procedures below.

If the node is a candidate for a TURN server it will insert some pointers in the overlay so that other peers can find it. The overlay configuration file specifies a turn-density parameter that indicates how many times each TURN server should record itself in the overlay. Typically this should be set to the reciprocal of the estimate of

what percentage of peers will act as TURN servers. If the turn-density is not set to zero, for each value, called d, between 1 and turn-density, the peer forms a Resource Name by concatenating its Node-ID and the value d. This Resource Name is hashed to form a Resource-ID. The address of the peer is stored at that Resource-ID using type TURN-SERVICE and the TurnServer object:

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

The contents of this structure are as follows:

iteration
the d value

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

Note: Correct functioning of this algorithm depends on having turn-density be an reasonable estimate of the reciprocal of the proportion of nodes in the overlay that can act as TURN servers. If the turn-density value in the configuration file is too low, then the process of finding TURN servers becomes more expensive as multiple candidate Resource-IDs must be probed to find a TURN server.

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

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

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

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

finding a server relatively quickly.

9. Chord Algorithm

This algorithm is assigned the name chord-reload to indicate it is an adaptation of the basic Chord 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 topology plugin, the size of the Resource-ID is 128 bits. The hash of a Resource-ID is computed using SHA-1 [RFC3174]then truncating the SHA-1 result to the most significant 128 bits.

9.3. Routing

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

If a peer is not responsible for a Resource-ID k, but is directly connected to a node with Node-ID k, then it routes the message to that node. Otherwise, it routes the request to the peer in the routing table that has the largest Node-ID that is in the interval between the peer and k. If no such node is found, it finds the smallest Node-Id that is greater than k and routes the message to that node.

9.4. Redundancy

When a peer receives a Store request for Resource-ID k, and it is responsible for Resource-ID k, it stores the data and returns a success response. It then sends a Store request to its successor in the neighbor table and to that peer's successor. Note that these Store requests are addressed to those specific peers, even though the Resource-ID they are being asked to store is outside the range that they are responsible for. The peers receiving these check they came from an appropriate predecessor in their neighbor table and that they are in a range that this predecessor is responsible for, and then they store the data. They do not themselves perform further Stores because they can determine that they are not responsible for the Resource-ID.

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

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

9.5. Joining

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

1. JP MUST connect to its chosen bootstrap node.
2. JP SHOULD send an Attach request to the admitting peer (AP) for Node-ID n. The "send_update" flag should be used to acquire the routing table for AP.
3. JP SHOULD send Attach requests to initiate connections to each of the peers in the neighbor table as well as to the desired finger table entries. Note that this does not populate their routing tables, but only their connection tables, so JP will not get messages that it is expected to route to other nodes.
4. JP MUST enter all the peers it has contacted into its routing table.
5. JP MUST send a Join to AP. The AP sends the response to the Join.
6. AP MUST do a series of Store requests to JP to store the data that JP will be responsible for.
7. AP MUST send JP an Update explicitly labeling JP as its predecessor. At this point, JP is part of the ring and responsible for a section of the overlay. AP can now forget any data which is assigned to JP and not AP.

8. The AP MUST send an Update to all of its neighbors with the new values of its neighbor set (including JP).
9. The JP MUST send Updates to all the peers in its neighbor table.

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

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

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

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

9.6. Routing Attaches

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

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

9.7. Updates

A chord Update is defined as

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


      struct {
         uint32               uptime;
         ChordUpdateType        type;
         select(type){
          case peer_ready:                  /* Empty */
            ;

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

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

The "uptime" field contains the time this peer has been up in seconds.

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

peer_ready: this peer is ready to receive messages. This message is used to indicate that a node which has Attached is a peer and can be routed through. It is also used as a connectivity check to non-neighbor peers.

neighbors: this version is sent to members of the Chord neighbor table.

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

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

predecessors
The predecessor set of the Updating peer.

successors
The successor set of the Updating peer.

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

predecessors
The predecessor set of the Updating peer.

successors
The successor set of the Updating peer.

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

A peer MUST maintain an association (via Attach) to every member of its neighbor set. A peer MUST attempt to maintain at least three predecessors and three successors, even though this will not be possible if the ring is very small. It is RECOMMENDED that O(log(N)) predecessors and successors be maintained in the neighbor set.

9.7.1. Handling Neighbor Failures

Every time a connection to a peer in the neighbor table is lost (as determined by connectivity pings or the failure of some request), the peer MUST remove the entry from its neighbor table and replace it with the best match it has from the other peers in its routing table. If using reactive recovery, it then sends an immediate Update to all nodes in its Neighbor Table. The update will contain all the Node-IDs of the current entries of the table (after the failed one has been removed). Note that when replacing a successor the peer SHOULD delay the creation of new replicas for successor replacement hold-down time (30 seconds) after removing the failed entry from its neighbor table in order to allow a triggered update to inform it of a better match for its neighbor table.

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

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

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

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

9.7.2. Handling Finger Table Entry Failure

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

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

9.7.3. Receiving Updates

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

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

In the first case, no change is needed.

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

After any Pings and Attaches are done, if the neighbor table changes and the peer is using reactive recovery, the peer sends an Update request to each member of its Connection Table. These Update requests are what end up filling in the predecessor/successor tables of peers that this peer is a neighbor to. A peer MUST NOT enter itself in its successor or predecessor table and instead should leave

the entries empty.

If peer N is responsible for a Resource-ID R, and N discovers that the replica set for R (the next two nodes in its successor set) has changed, it MUST send a Store for any data associated with R to any new node in the replica set. It SHOULD NOT delete data from peers which have left the replica set.

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

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

9.7.4. Stabilization

There are four components to stabilization:

1. exchange Updates with all peers in its neighbor table to exchange state.
2. search for better peers to place in its finger table.
3. search to determine if the current finger table size is sufficiently large.
4. search to determine if the overlay has partitioned and needs to recover.

9.7.4.1. Updating neighbor table

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

9.7.4.2. Refreshing finger table

A peer MUST periodically search for new peers to replace invalid (repeated) 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))-2^(128-(i+1))]. Invalid entries occur in the finger table when a previous finger table entry has failed or when no peer has been found in that range.

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

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

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

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

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

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

9.7.4.3. Adjusting finger table size

If the finger table has less than 16 entries, the node SHOULD attempt to discover more fingers to grow the size of the table to 16. The value 16 was chosen to ensure high odds of a node maintaining connectivity to the overlay even with strange network partitions.

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

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

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

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

9.7.4.4. Detecting partitioning

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

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

9.8. Route query

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

```
struct {
   NodeId                  next_peer;
} ChordRouteQueryAns;
```

The contents of this structure are as follows:

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

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

9.9. Leaving

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

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

struct {
  ChordLeaveType         type;

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

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

from_succ
: The Leave request was sent by a successor.

from_pred
: The Leave request was sent by a predecessor.

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

successors
: The sender's successor list.

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

predecessors
: The sender's predecessor list.

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

10. Enrollment and Bootstrap

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

10.1. Overlay Configuration

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

```
<?xml version="1.0" encoding="UTF-8"?>
<overlay xmlns="urn:ietf:params:xml:ns:p2p:config-base"
   xmlns:ext="urn:ietf:params:xml:ns:p2p:config-ext1"
   xmlns:chord="urn:ietf:params:xml:ns:p2p:config-chord">
   <configuration instance-name="overlay.example.org" sequence="22"
       expiration="2002-10-10T07:00:00Z" ext:ext-example="stuff" >
       <topology-plugin> CHORD-RELOAD </topology-plugin>
       <node-id-length>16</node-id-length>
       <root-cert>
MIIDJDCCAo2gAwIBAgIBADANBgkqhkiG9w0BAQUFADBwMQswCQYDVQQGEwJVUzET
MBEGA1UECBMKQ2FsaWZvcm5pYTERMA8GA1UEBxMIU2FuIEpvc2UxDjAMBgNVBAoT
BXNpcGl0MSkwJwYDVQQLEyBTaXBpdCBUZXN0IENlcnRpZmljYXRlIEF1dGhvcml0
eTAeFw0wMzA3MTgxMjIxNTJaFw0xMzA3MTUxMjIxNTJaMHAxCzAJBgNVBAYTAlVT
MRMwEQYDVQQIEwpDYWxpZm9ybmlhMREwDwYDVQQHEwhTYW4gSm9zZTEOMAwGA1UE
ChMFc2lwaXQxKTAnBgNVBAsTIFNpcGl0IFRlc3QgQ2VydGlmaWNhdGUgQXV0aG9y
aXR5MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDDIh6DkcUDLDyK9BEUxkud
+nJ4xrCVGKfgjHm6XaSuHiEtnfELHM+9WymzkBNzZpJu30yzsxwfKoIKugdNUrD4
N3viCicwcN35LgP/KnbN34cavXHr4ZlqxH+OdKB3hQTpQa38A7YXdaoz6goW2ft5
Mi74z03GNKP/G9BoKOGd5QIDAQABo4HNMIHKMB0GA1UdDgQWBBRrRhcU6pR2JYBU
bhNU2qHjVBShtjCBmgYDVR0jBIGSMIGPgBRrRhcU6pR2JYBUbhNU2qHjVBShtqF0
pHIwcDELMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExETAPBgNVBAcT
CFNhbiBKb3NlMQ4wDAYDVQQKEwVzaXBpdDEpMCcGA1UECxMgU2lwaXQgVGVzdCBD
ZXJ0aWZpY2F0ZSBBdXRob3JpdHmCAQAwDAYDVR0TBAUwAwEB/zANBgkqhkiG9w0B
AQUFAAOBgQCWbRvv1ZGTRXxbH8/EqkdSCzSoUPrs+rQqR0xdQac9wNY/nlZbkR3O
qAezG6Sfmklvf+DOg5RxQq/+Y6I03LRepc7KeVDpaplMFGnpfKsibETMipwzayNQ
QgUf4cKBiF+65Ue7hZuDJa2EMv8qW4twEhGDYclpFU9YozyS1OhvUg==
       </root-cert>
       <root-cert> YmFkIGNlcnQK </root-cert>
       <enrollment-server>https://example.org</enrollment-server>
       <enrollment-server>https://example.net</enrollment-server>
       <self-signed-permitted
                digest="sha1">false</self-signed-permitted>
       <bootstrap-node address="192.0.0.1" port="6084" />
       <bootstrap-node address="192.0.2.2" port="6084" />
       <bootstrap-node address="2001:DB8::1" port="6084" />
       <turn-density> 20 </turn-density>
       <multicast-bootstrap address="192.0.0.1" />
```

```
          <multicast-bootstrap address="233.252.0.1" port="6084" />
          <clients-permitted> false </clients-permitted>
          <no-ice> false </no-ice>
          <chord:chord-update-interval>
              400</chord:chord-update-interval>
          <chord:chord-ping-interval>30</chord:chord-ping-interval>
          <chord:chord-reactive> true </chord:chord-reactive>
          <shared-secret> password </shared-secret>
          <max-message-size>4000</max-message-size>
          <initial-ttl> 30 </initial-ttl>
          <overlay-link-protocol>TLS</overlay-link-protocol>
          <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>

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

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

 </overlay>
```

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

The file can contain multiple "configuration" elements where each one contains the configuration information for a different overlay. Each "configuration" 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 peer SHOULD retrieve a new copy of the configuration at a randomly selected time that is before the expiration time.
sequence: a monotonically increasing sequence number between 1 and 2^16-2

Inside each overlay element, the following elements can occur:

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

self-signed-permitted This element indicates whether self-signed certificates are permitted. If it is set to "true", then self-signed certificates are allowed, in which case the enrollment-server and root-cert elements may be absent. Otherwise, it SHOULD be absent, but MAY be set to "false". This element also contains an attribute "digest" which indicates the digest to be used to compute the Node-ID. Valid values for this parameter are "sha1" and "sha256" representing SHA-1 [RFC3174] and SHA-256 [RFC4634] respectively. Implementations MUST support both of these algorithms.

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

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

multicast-bootstrap This element represents the address of a multicast, broadcast, or anycast address and port that may be used for bootstrap. Nodes SHOULD listen on the address. It has an attributed called "address" that represents the IP address and an optional attribute called "port" that represents the port and defaults to 6084. More than one "multicast-bootstrap" element may be present.

clients-permitted This element represents whether clients are permitted or whether all nodes must be peers. If it is set to "true" or absent, this indicates that clients are permitted. If it is set to "false" then nodes are not allowed to remain clients after the initial join. There is currently no way for the overlay to enforce this.

no-ice This element represents whether nodes are required to use the "No-ICE" Overlay Link protocols in this overlay. If it is absent, it is treated as if it were set to "false".

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

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

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

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

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

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

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

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

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.

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

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

In addition, the kind element contains the following elements:

max-count: the maximum number of values which members of the overlay must support.
data-model: the data model to be used.
max-size: the maximum size of individual values.
access-control: the access control model to be used.
max-node-multiple: This is optional and only used when the access control is NODE-MULTIPLE. This indicates the maximum value for the i counter. This is an integer greater than 0.

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

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

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

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

10.1.1. Relax NG Grammar

The grammar for the configuration data is:

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

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

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

start =  element p2pcf:overlay {
      overlay-element
}

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

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

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

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

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

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

   kind-block = element kind-block {
       element kind {
           (  attribute name { kind-names }
              | attribute id { xsd:int } ),
           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 first determines the overlay name. This value is provided by the user or some other out of band provisioning mechanism. The out of band mechanisms may also provide an optional URL for the configuration server. If a URL for the configuration server is not provided, the node MUST do a DNS SRV query using a Service name of "p2psip-enroll" and a protocol of TCP to find a configuration server and form the URL by appending a path of "/.well-known/p2psip-enroll" to the overlay name. This uses the "well known URI" framework defined in [RFC5785]. For example, if the overlay name was example.com, the URL would be "https://example.com//.well-known/p2psip-enroll".

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

For overlays that do not use a configuration server, nodes obtain the configuration information needed to join the overlay through some out

of band approach such an XML configuration file sent over email.

10.3. Credentials

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

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

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

- If authentication is required, there is a URL parameter of "password" and "username" containing the user's name and password in the clear (hence the need for HTTPS)
- The body is of content type "application/pkcs10", as defined in [RFC2311].
- The Accept header contains the type "application/pkix-cert", indicating the type that is expected in the response.

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

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

- o A single name this user is allowed to use in the overlay, using type rfc822Name.

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

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

10.3.1. Self-Generated Credentials

If the "self-signed-permitted" element is present in the configuration and set to "true", then a node MUST generate its own self-signed certificate to join the overlay. The self-signed certificate MAY contain any user name of the users choice.

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

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

10.4. Searching for a Bootstrap Node

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

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

10.5. Contacting a Bootstrap Node

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

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

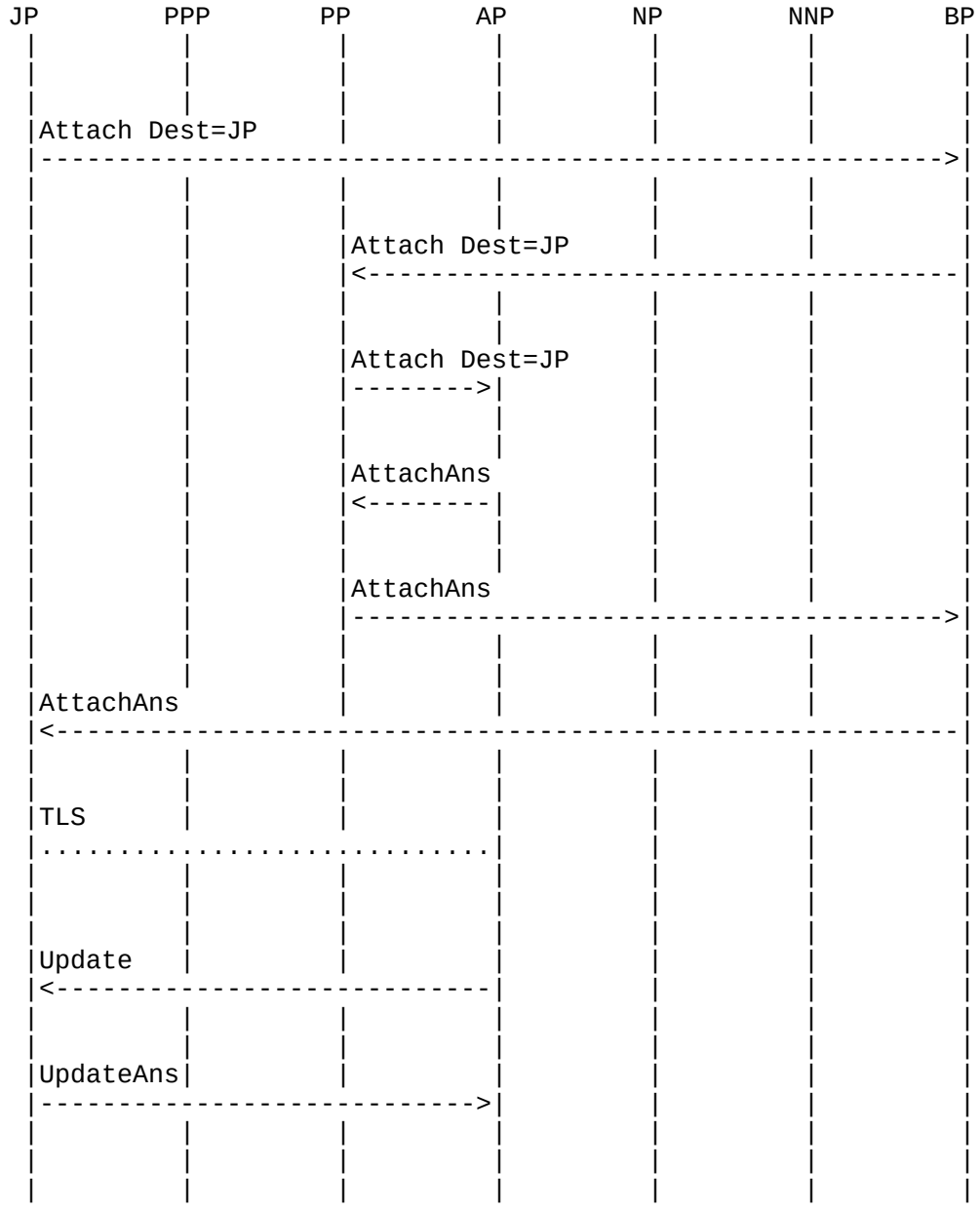
When the requester node finally does receive a response from some responding node, it can note the Node-ID in the response and use this Node-ID to start sending requests to join the Overlay Instance as described in Section 5.4.

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

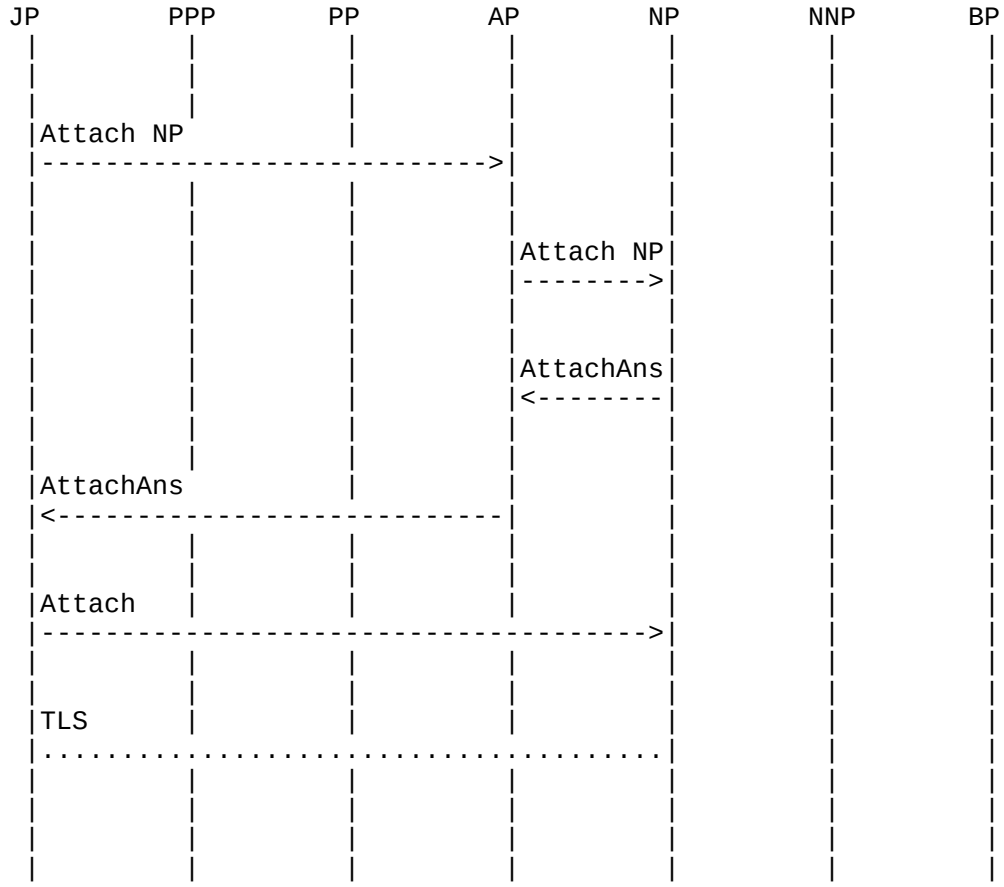
11. Message Flow Example

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

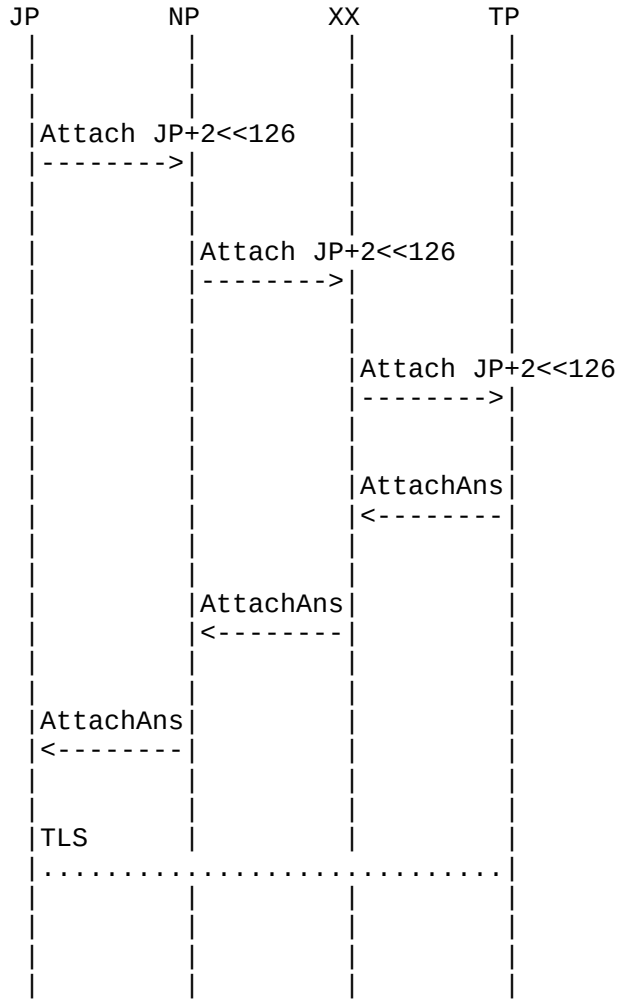
In the following example, we assume that JP has formed a connection to one of the bootstrap nodes. JP then sends an Attach through that peer to a resource ID of itself (JP). It gets routed to the admitting peer (AP) because JP is not yet part of the overlay. When AP responds, JP and AP use ICE to set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



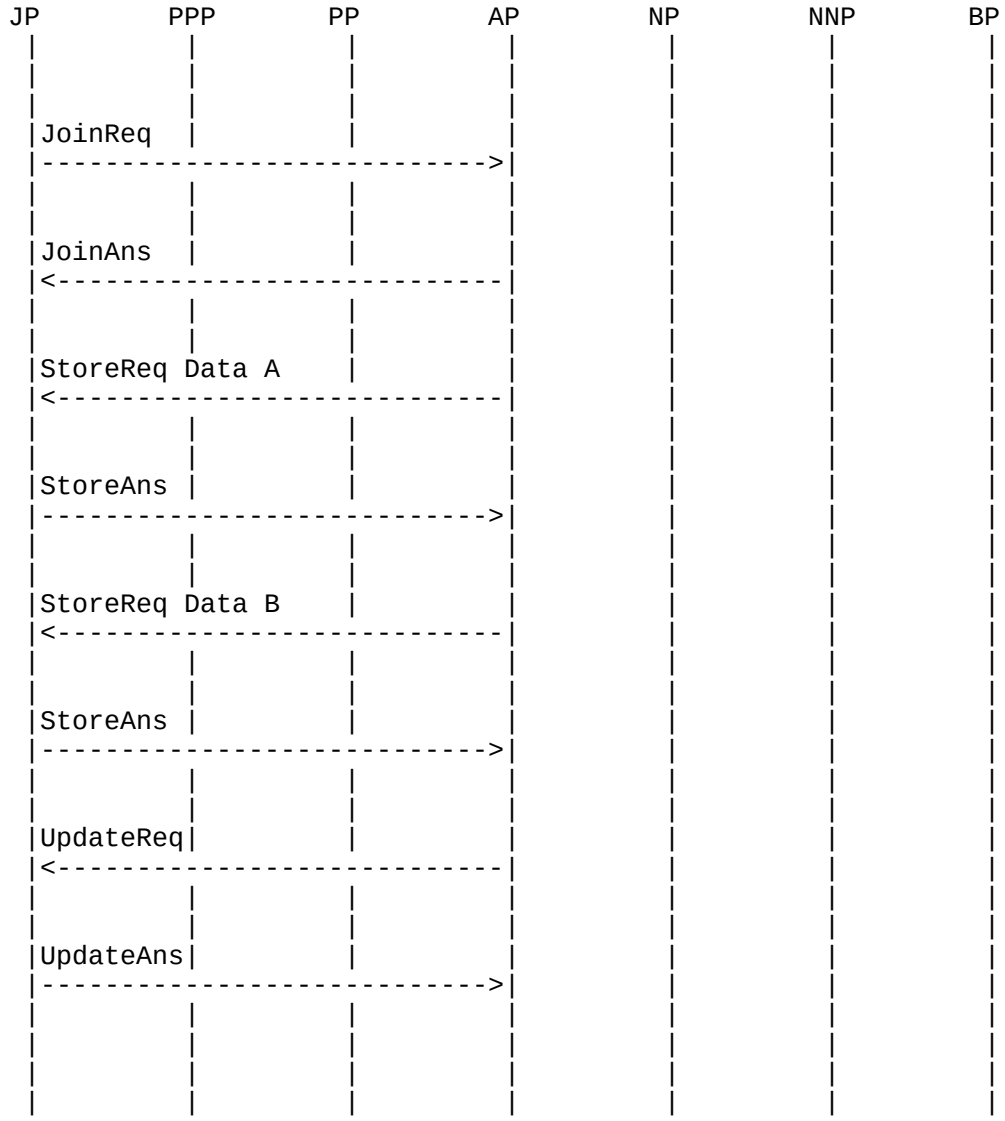
The JP then forms connections to the appropriate neighbors, such as NP, by sending an Attach which gets routed via other nodes. When NP responds, JP and NP use ICE and TLS to set up a connection.



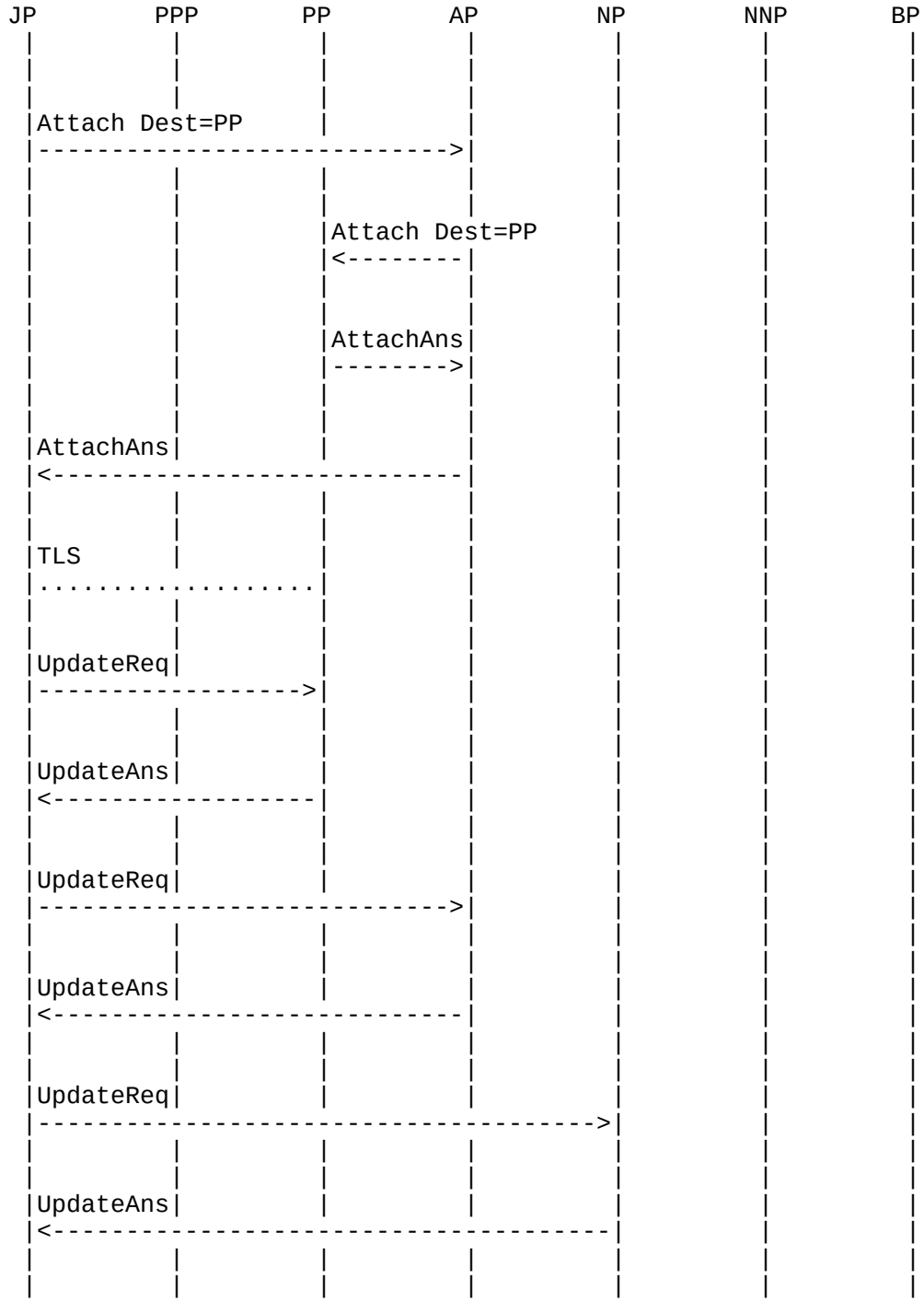
JP also needs to populate its finger table (for Chord). It issues an Attach to a variety of locations around the overlay. The diagram below shows it sending an Attach halfway around the Chord ring to the JP + 2^127.



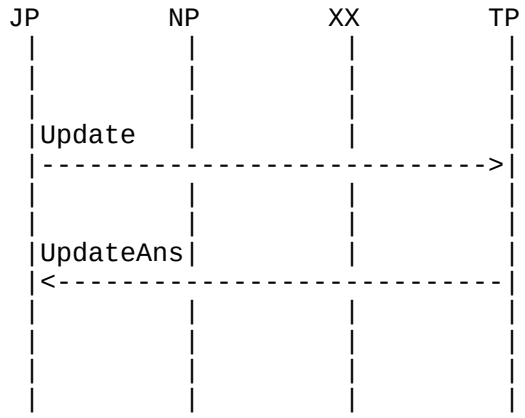
Once JP has a reasonable set of connections, it is ready to take its place in the DHT. It does this by sending a Join to AP. AP does a series of Store requests to JP to store the data that JP will be responsible for. AP then sends JP an Update explicitly labeling JP as its predecessor. At this point, JP is part of the ring and responsible for a section of the overlay. AP can now forget any data which is assigned to JP and not AP.



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


JP
PPP
PP
AP
NP
NNP
BP
Attach Dest=PP
Attach Dest=PP
AttachAns
AttachAns
TLS
UpdateReq
UpdateAns
UpdateReq
UpdateAns
UpdateReq
UpdateAns

Finally, now that JP has a copy of all the data and is ready to route messages and receive requests, it sends Updates to everyone in its Routing Table to tell them it is ready to go. Below, it is shown sending such an update to TP.



12. Security Considerations

12.1. Overview

RELOAD provides a generic storage service, albeit one designed to be useful for P2PSIP. In this section we discuss security issues that are likely to be relevant to any usage of RELOAD. More background information can be found in [RFC5765].

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

P2PSIP deployments require the ability to authenticate both peers and resources (users) without the active presence of a trusted entity in the system. We describe two mechanisms. The first mechanism is based on public key certificates and is suitable for general deployments. The second is an admission control mechanism based on an overlay-wide shared symmetric key.

12.2. Attacks on P2P Overlays

The two basic functions provided by overlay nodes are storage and routing: some node is responsible for storing a peer's data and for allowing a third peer to fetch this stored data. Other nodes are responsible for routing messages to and from the storing nodes. Each of these issues is covered in the following sections.

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

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

12.3. Certificate-based Security

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

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

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

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

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

a certificate, this usage does not require a global PKI. Instead, certificates are signed by a central enrollment authority which acts as the certificate authority for the Overlay Instance. This authority signs each peer's certificate. Because each peer possesses the CA's certificate (which they receive on enrollment) they can verify the certificates of the other entities in the overlay without further communication. Because the certificates contain the user/peer's public key, communications from the user/peer can be verified in turn.

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

Because all stored data is signed by the owner of the data the storing peer can verify that the storer is authorized to perform a store at that Resource-ID and also allow any consumer of the data to verify the provenance and integrity of the data when it retrieves it.

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

All implementations MUST implement certificate-based security.

12.4. Shared-Secret Security

RELOAD also supports a shared secret admission control scheme that relies on a single key that is shared among all members of the overlay. It is appropriate for small groups that wish to form a private network without complexity. In shared secret mode, all the peers share a single symmetric key which is used to key TLS-PSK [RFC4279] or TLS-SRP [RFC5054] mode. A peer which does not know the key cannot form TLS connections with any other peer and therefore cannot join the overlay.

One natural approach to a shared-secret scheme is to use a user-entered password as the key. The difficulty with this is that in TLS-PSK mode, such keys are very susceptible to dictionary attacks.

If passwords are used as the source of shared-keys, then TLS-SRP is a superior choice because it is not subject to dictionary attacks.

12.5. Storage Security

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

12.5.1. Authorization

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

When the storing peer receives the request, it must determine whether the storing client is authorized to store at this Resource-ID/Kind-ID pair. Determining this requires comparing the user's identity to the requirements of the access control model (see Section 6.3). If it satisfies those requirements the user is authorized to write, pending quota checks as described in the next section.

For example, consider the certificate with the following properties:

```
User name: alice@dht.example.com
Node-ID:   013456789abcdef
Serial:    1234
```

If Alice wishes to Store a value of the "SIP Location" kind, the Resource Name will be the SIP AOR "sip:alice@dht.example.com". The Resource-ID will be determined by hashing the Resource Name. Because SIP Location uses the USER-NODE-MATCH policy, it first verifies that the user name in the certificate hashes to the requested Resource-ID. It then verifies that the Node-Id in the certificate matches the dictionary key being used for the store. If both of these checks succeed, the Store is authorized. Note that because the access control model is different for different kinds, the exact set of checks will vary.

12.5.2. Distributed Quota

Being a peer in an Overlay Instance carries with it the responsibility to store data for a given region of the Overlay Instance. However, allowing clients to store unlimited amounts of data would create unacceptable burdens on peers and would also enable trivial denial of service attacks. RELOAD addresses this issue by requiring configurations to define maximum sizes for each kind of stored data. Attempts to store values exceeding this size MUST be rejected (if peers are inconsistent about this, then strange artifacts will happen when the zone of responsibility shifts and a different peer becomes responsible for overlarge data). Because each Resource-ID/Kind-ID pair is bound to a small set of certificates, these size restrictions also create a distributed quota mechanism, with the quotas administered by the central configuration server.

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

12.5.3. Correctness

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

12.5.4. Residual Attacks

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

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

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

12.6. Routing Security

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

12.6.1. Background

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

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

12.6.2. Admissions Control

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

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

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

12.6.3. Peer Identification and Authentication

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

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

Replay is typically prevented for messages that impact the topology

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

12.6.4. Protecting the Signaling

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

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

12.6.5. Residual Attacks

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

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

13. IANA Considerations

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

13.1. Well-Known URI Registration

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

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

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

13.2. Port Registrations

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

IANA will make the following port registration:

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

13.3. Overlay Algorithm Types

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

Algorithm Name	RFC
CHORD-RELOAD	RFC-AAAA

13.4. Access Control Policies

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

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

13.5. Application-ID

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

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

13.6. Data Kind-ID

IANA SHALL create a "RELOAD Data Kind-ID" Registry. Entries in this registry are 32-bit integers denoting data kinds, as described in Section 4.2. Code points in the range 0x00000001 to 0x7fffffff SHALL be registered via RFC 5226 Standards Action. Code points in the range 0x8000000 to 0xf0000000 SHALL be registered via RFC 5226 Expert Review. Code points in the range 0xf0000001 to 0xfffffffe are reserved for private use via the kind description mechanism described in Section 10. The initial contents of this registry are:

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

13.7. Data Model

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

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

13.8. Message Codes

IANA SHALL create a "RELOAD Message Code" Registry. Entries in this registry are 16-bit integers denoting method codes as described in Section 5.3.3. These codes SHALL be registered via RFC 5226 Standards Action. The initial contents of this registry are:

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

13.9. Error Codes

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

Error Code Name	Code Value	RFC
invalid	0	RFC-AAAA
Unused	1	RFC-AAAA
Error_Forbidden	2	RFC-AAAA
Error_Not_Found	3	RFC-AAAA
Error_Request_Timeout	4	RFC-AAAA
Error_Generation_Counter_Too_Low	5	RFC-AAAA
Error_Incompatible_with_Overlay	6	RFC-AAAA
Error_Unsupported_Forwarding_Option	7	RFC-AAAA
Error_Data_Too_Large	8	RFC-AAAA
Error_Data_Too_Old	9	RFC-AAAA
Error_TTL_Exceeded	10	RFC-AAAA
Error_Message_Too_Large	11	RFC-AAAA
Error_Unknown_Kind	12	RFC-AAAA
Error_Unknown_Extension	13	RFC-AAAA
Error_Response_Too_Large	14	RFC-AAAA
Error_Config_Too_Old	15	RFC-AAAA
Error_Config_Too_New	16	RFC-AAAA
Error_In_Progress	17	RFC-AAAA
reserved	0x8000..0xfffe	RFC-AAAA

13.10. Overlay Link Types

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

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

13.11. Overlay Link Protocols

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

13.12. Forwarding Options

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

Forwarding Option	Code	Specification
invalid	0	RFC-AAAA
reserved	255	RFC-AAAA

13.13. Probe Information Types

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

Probe Option	Code	Specification
invalid	0	RFC-AAAA
responsible_set	1	RFC-AAAA
num_resources	2	RFC-AAAA
uptime	3	RFC-AAAA
reserved	255	RFC-AAAA

13.14. Message Extensions

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

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

13.15. reload URI Scheme

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

- A peer.
- A resource inside a peer.

The reload URI is defined using a subset of the URI schema specified in Appendix A of RFC 3986 [RFC3986] and the associated URI Guidelines [RFC4395] per the following ABNF syntax:

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

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

The definitions of these productions are as follows:

destination: a hex-encoded Destination List object (i.e., multiple concatenated Destination objects with no length prefix prior to the object as a whole.)

overlay: the name of the overlay.

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

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

13.15.1. URI Registration

[[Note to RFC Editor - please remove this paragraph before publication.]] 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

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 J. Maenpaa, G. Camarillo, and J. Hautakorpi.

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

15. References

15.1. Normative References

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

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

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

[RFC2988] Paxson, V. and M. Allman, "Computing TCP's Retransmission Timer", RFC 2988, November 2000.

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

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

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

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

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

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

[RFC4634] Eastlake, D. and T. Hansen, "US Secure Hash Algorithms (SHA and HMAC-SHA)", RFC 4634, July 2006.

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

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

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

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

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

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

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

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

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

15.2. Informative References

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

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

[I-D.baset-tsvwg-tcp-over-udp] Baset, S. and H. Schulzrinne, "TCP-over-UDP", draft-baset-tsvwg-tcp-over-udp-01 (work in progress), June 2009.

[I-D.ietf-hip-bone] Camarillo, G., Nikander, P., Hautakorpi, J., Keranen, A., and A. Johnston, "HIP BONE: Host Identity Protocol (HIP) Based Overlay Networking Environment", draft-ietf-hip-bone-07 (work in progress), June 2010.

[I-D.ietf-hip-reload-instance] Keranen, A., Camarillo, G., and J. Maenpaa, "Host Identity Protocol-Based Overlay Networking Environment (HIP BONE) Instance Specification for REsource LOcation And Discovery (RELOAD)", draft-ietf-hip-reload-instance-03 (work in progress), January 2011.

[I-D.ietf-mmusic-ice-tcp]
Rosenberg, J., Keranen, A., Lowekamp, B., and A. Roach, "TCP Candidates with Interactive Connectivity Establishment (ICE)", draft-ietf-mmusic-ice-tcp-12 (work in progress), February 2011.

[I-D.ietf-p2psip-concepts]
Bryan, D., Matthews, P., Shim, E., Willis, D., and S. Dawkins, "Concepts and Terminology for Peer to Peer SIP", draft-ietf-p2psip-concepts-03 (work in progress), October 2010.

[I-D.ietf-p2psip-sip]
Jennings, C., Lowekamp, B., Rescorla, E., Baset, S., and H. Schulzrinne, "A SIP Usage for RELOAD", draft-ietf-p2psip-sip-05 (work in progress), July 2010.

[I-D.jiang-p2psip-relay]
Jiang, X., Zong, N., Even, R., and Y. Zhang, "An extension to RELOAD to support Direct Response and Relay Peer routing", draft-jiang-p2psip-relay-04 (work in progress), April 2010.

[I-D.maenpaa-p2psip-self-tuning]
Maenpaa, J., Camarillo, G., and J. Hautakorpi, "A Self-tuning Distributed Hash Table (DHT) for REsource LOcation And Discovery (RELOAD)", draft-maenpaa-p2psip-self-tuning-01 (work in progress), October 2009.

[I-D.maenpaa-p2psip-service-discovery]
Maenpaa, J. and G. Camarillo, "Service Discovery Usage for REsource LOcation And Discovery (RELOAD)", draft-maenpaa-p2psip-service-discovery-00 (work in progress), October 2009.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[handling-churn-usenix04] Rhea, S., Geels, D., Roscoe, T., and J. Kubiatowicz,

"Handling Churn in a DHT", In Proc. of the USENIX Annual Technical Conference June 2004 USENIX 2004.

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

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

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

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

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

Appendix A. Change Log

A.1. Changes since draft-ietf-p2psip-reload-12

- Clarified lifetime management.
- Made it clear how direct return response could be put in an extension .
- Clarified distinction between enrollment and configuration servers.
- Clarified that the KindId is 32 bits long. The -12 draft had some text that said 32 bits and one typo that said 16. Earlier drafts were said 32 bits.
- Miscellaneous editorial.
- Specified the Chord hash was SHA-1 truncated
- Clarified Attache procedures.
- Changed name in XML configuration from chord-reload-reactive to chord-reactive. Made other chord-reload- names consistent.
- Fixed the relax NG and made a more complex configuration example.

- Added Error_In_Progress to Error Code table

Appendix B. 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.

B.1. Iterative vs Recursive

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

Iterative routing is supported through the 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.

B.2. Symmetric vs Forward response

An alternative to the symmetric recursive routing method used by RELOAD is Forward-Only routing, where the response is routed to the requester as if it 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.

B.3. Direct Response

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

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

- 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.

B.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.

B.5. Symmetric Route Stability

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

The other aspect of this issue is whether the request itself can be successfully delivered. Assuming typical connection maintenance intervals, the time period between the last maintenance and the request being sent will be orders of magnitude greater than the delay between the request being forwarded and the response being received.

Therefore, if the path was stable enough to be available to route the request, it is almost certainly going to remain available to route the response.

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

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

Appendix C. Why Clients?

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

C.1. Why Not Only Peers?

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

- 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.

C.2. Clients as Application-Level Agents

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

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

Authors' Addresses

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

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

Bruce B. Lowekamp (editor)
Skype
Palo Alto, CA
USA

Email: bbl@lowekamp.net

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

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

Salman A. Baset
Columbia University
1214 Amsterdam Avenue
New York, NY
USA

Email: salman@cs.columbia.edu

Henning Schulzrinne
Columbia University
1214 Amsterdam Avenue
New York, NY
USA

Email: hgs@cs.columbia.edu

P2PSIP
Internet-Draft
Intended status: Standards Track
Expires: September 11, 2011

X. Jiang
N. Zong
Huawei Technologies
R. Even
Gesher Erove
Y. Zhang
China Mobile

March 10, 2011

An extension to RELOAD to support Direct Response and Relay Peer routing
draft-jiang-p2psip-relay-05

Abstract

This document proposes an optional extension to RELOAD to support direct response and relay peer routing modes. RELOAD recommends symmetric recursive routing for routing messages. The new optional extensions provide a shorter route for responses reducing the overhead on intermediary peers and describe the potential cases where these extensions 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 September 11, 2011.

Table of Contents

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 and RPR are advantageous over RPR in some scenarios 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. On the other hand, 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 these three routing modes 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].

In this draft, we first discuss the problem statement, then the relationship between the three routing modes is presented. In Section 5, we give comparison on the cost of SRR, DRR and RPR in both managed and open networks. An extension to RELOAD to support DRR and RPR is proposed in Section 6.

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.

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.

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 DRR and 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 three routing modes: SRR, DRR and RPR.

Some assumptions are made in the following illustrations.

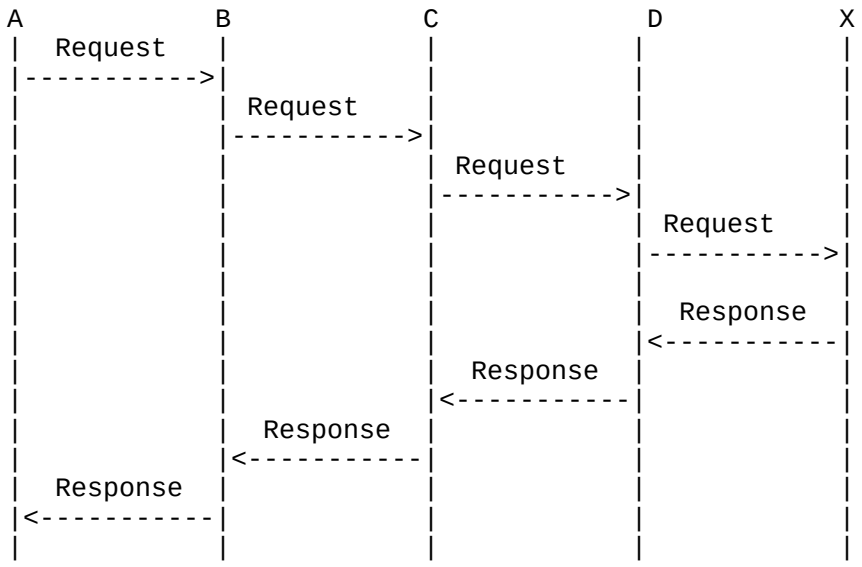
1) Peer A sends a request destined to a peer who is the responsible peer for Resource-ID k;

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.

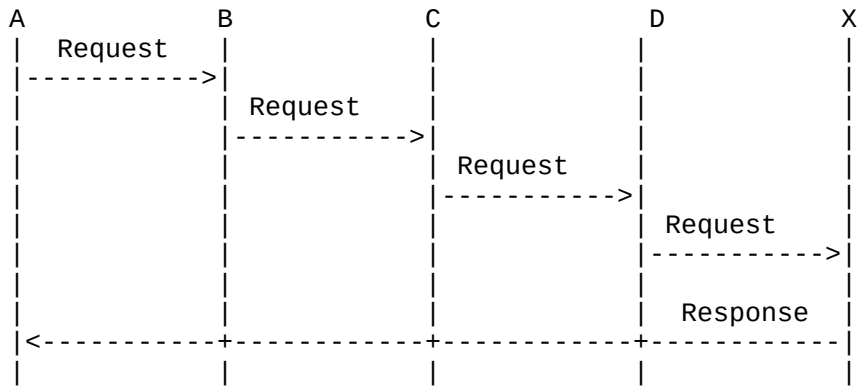


SRR works in any situation, especially when there are NATs or firewalls. A downside of this solution is that the message takes several hops to return to the client, increasing the bandwidth usage and CPU/battery load of multiple nodes.

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 secure connection requires multiple round trips. Please refer

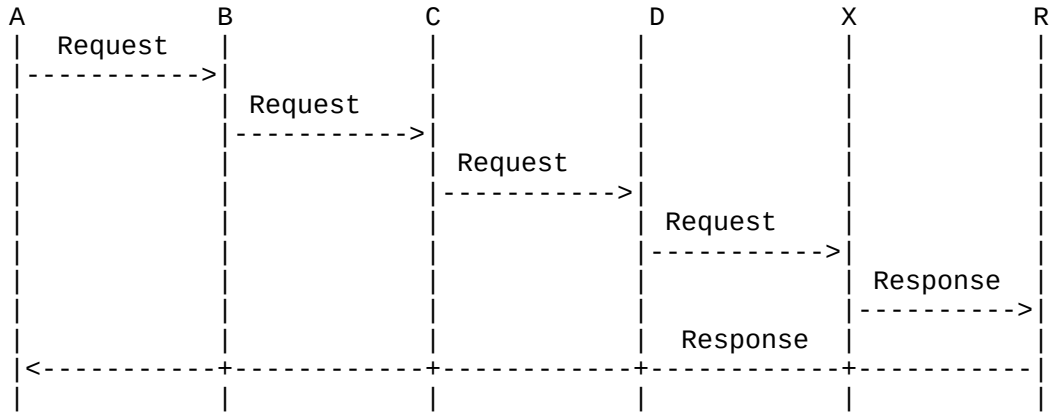
to Section 5 for cost comparison between SRR, DRR/RPR.



3.1.3. 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, DRR/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.



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 severs, 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 the responding peer receives a retransmit on a request with SRR as the preferred routing mode he should stop trying to establish a direct connection and use 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.

3.3. Scenarios Where RPR Benefits

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

3.3.1. 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. As with DRR, 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.

3.3.2. 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.

3.3.3. Wireless Scenarios

While some mobile deployments may use clients, in mobile networks using peers, RPR, like DRR, 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 DRR/RPR

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 RPR Works

RPR is a bit more complicated than DRR. 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.

(Editor's Note: 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.)

4.3. How These Three Routing Modes Work Together

DRR and RPR are not intended to replace SRR. As seen from Section 3, DRR or RPR have 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 three modes together to adapt to each peer's specific situation. In this section, we give some suggestions on how to transition between the routing modes in RELOAD.

Editor's Note: What this draft proposes are optional extensions to support DRR/RPR. There is no requirement for implementation to use the strategy described to choose the appropriate mode.

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 or RPR. Based on the success rate as seen from the lost message statistics or responses that used SRR, the node can either continue to offer DRR/RPR first or switch to SRR.

The node can decide whether to try DRR or RPR 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. If a relay peer is provided by the service provider, nodes may prefer RPR over SRR.

5. Comparison on cost of SRR and DRR/RPR

The major advantages in using DRR/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 DRR/RPR.

SRR brings out more routing hops than DRR and RPR. 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, DRR and RPR are listed in the following table. Establishing a secure connection between sending/relay 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/RPR; 2) still using DTLS in DRR/RPR 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.)

Mode	Success	No. of Hops	No. of Msgs
SRR	Yes	logN	logN
DRR	Yes	1	1
RPR	Yes	2	2
DRR(DTLS)	Yes	1	7+1
RPR(DTLS)	Yes	2	7+2

From the above comparison, it is clear that:

1) In most cases of N > 2^2=4, DRR/RPR has fewer hops than SRR. Shorter route means less overhead and resource usage on intermediary peers, which is an important consideration for adopting DRR/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 > 2^9=512, DRR/RPR also has fewer messages than SRR.

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

5.2. Open networks

In open network where DRR/RPR is not guaranteed, DRR/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, DRR and RPR are listed in the following table.

Mode	Success	No. of Hops	No. of Msgs
SRR	Yes	logN	logN
DRR	Yes	1	1
	Fail&Fall back to SRR	1+logN	1+logN
RPR	Yes	2	2
	Fail&Fall back to SRR	2+logN	2+logN
DRR(DTLS)	Yes	1	7+1
	Fail&Fall back to SRR	1+logN	8+logN
RPR(DTLS)	Yes	2	7+2
	Fail&Fall back to SRR	2+logN	9+logN

From the above comparison, it can be observed that:

1) Trying DRR/RPR 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)*(1+logN), N*logN, respectively. The condition for fewer hops in DRR is P*1+(N-P)*(1+logN) < N*logN, which is P/N > 1/logN. 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. Similar analysis can be easily applied to RPR. Therefore, the chance of trying DRR/RPR 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/RPR is good, it is still possible that DRR/RPR has fewer messages than SRR. Otherwise, the consideration to use DRR/RPR or SRR depends on other factors like using less resources from the intermediaries peers.

6. Extensions to RELOAD

Adding support for DRR and RPR 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 or RPR MUST support SRR.

All peers MUST be able to process requests for routing in SRR, and MAY support DRR or RPR routing requests.

Peers that do not support or do not wish to provide DRR or RPR MAY reject these messages.

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 and RPR modes. 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 or RPR 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 should be maintained in the intermediate peers.

flag : 0x3 IGNORE-STATE-KEEPING

If IGNORE-STATE-KEEPING is set, any peer receiving this message and which is not the destination of the message MUST forward the message with the full VIA list and MUST 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_TYPE:

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

```
enum { 0x0, 0x01 (DRR), 0x02(RPR), 255} RouteMode;
struct {
        RouteMode               routemode;
        OverlayLink             transport;
        IpAddressPort          ipaddressport;
        Destination             destination<1..2>;
} ExtensiveRoutingModeOption;
```

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

Route mode: refers to which type of routing mode is indicated to the destination peer. Currently, only DRR and RPR are defined.

Transport: refers to the transport type which is used to deliver responses from the destination peer to the sending peer or the relay 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 and a relay peer address for RPR.

Destination: refers to the relay peer or the sending node itself. if the routing mode is DRR, then the destination only contains the sending node’s node-id; 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 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_TYPE.

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 vaule, defined as type peer and set with the sending peer's own values.

6.3.2. 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_TYPE.

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 DRR and RPR are introduced. Here, we only describe the additional

procedures for supporting DRR and 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 a error response 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.

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 DRR and there is not exactly one destination, or the routing mode is set to RPR and there are not exactly two destinations the destination peer MUST try to send a error response 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 he is trying to response to now, the responding peer should abort the DRR/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]. 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.

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.

Editor note: This section will be extended if we adopt RPR, but like other configuration information, there may be many ways to obtain this.

8. Optional Methods to Investigate Node Connectivity

This section is for informational purposes only for providing some mecahnsism that can be used when the configuration information does not specify if DRR or 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 DRR and 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. If the peer believes it is publically reachable, DRR may be attempted. NATs and firewalls are two major contributors preventing DRR and 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 use DRR for responses and can also 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.

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.

8.1. 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.

8.2. Public Reachability 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 and RPR. 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, and may advertise that it is a candidate to serve as a relay peer.

9. Security Considerations

TBD

10. IANA Considerations

10.1. A new RELOAD Forwarding Option

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

Type: 0x1 - extensive_routing_mode

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 and Bruce Lowekamp for their constructive comments.

12. References

12.1. 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", draft-ietf-p2psip-base-12 (work in progress), March 2010.

[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.

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

12.2. Informative References

[ChurnDHT] Rhea, S., "Handling Churn in a DHT", Proceedings of the USENIX Annual Technical Conference. Handling Churn in a DHT, June 2004.

[DTLS] Modadugu, N., Rescorla, E., "The Design and Implementation of Datagram TLS", 11th Network and Distributed System Security Symposium (NDSS), 2004.

[I-D.ietf-behave-nat-behavior-discovery] MacDonald, D. and B. Lowekamp, "NAT Behavior Discovery Using STUN", draft-ietf-behave-nat-behavior-discovery-04 (work in progress), July 2008.

[I-D.ietf-behave-tcp] Guha, S., Biswas, K., Ford, B., Sivakumar, S., and P. Srisuresh, "NAT Behavioral Requirements for TCP", draft-ietf-behave-tcp-08 (work in progress), September 2008.

[I-D.lowekamp-mmusic-ice-tcp-framework] Lowekamp, B. and A. Roach, "A Proposal to Define Interactive Connectivity Establishment for the Transport Control Protocol (ICE-TCP) as an Extensible Framework", draft-lowekamp-mmusic-ice-tcp-framework-00 (work in progress), October 2008.

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

Authors' Addresses

Xingfeng Jiang
Huawei Technologies

Email: jiang.x.f@huawei.com

Ning Zong
Huawei Technologies

Email: zongning@huawei.com

Roni Even
Gesher Erove

Email: ron.even.tlv@gmail.com

Yunfei Zhang
China Mobile

Email: zhangyunfei@chinamobile.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2011

A. Knauf
G. Hege
T C. Schmidt
HAW Hamburg
M. Waehlisch
link-lab & FU Berlin
March 13, 2011

A RELOAD Usage for Distributed Conference Control (DisCo)
draft-knauf-p2psip-disco-02

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 September 14, 2011.

Copyright Notice

Table of Contents

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 der 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.

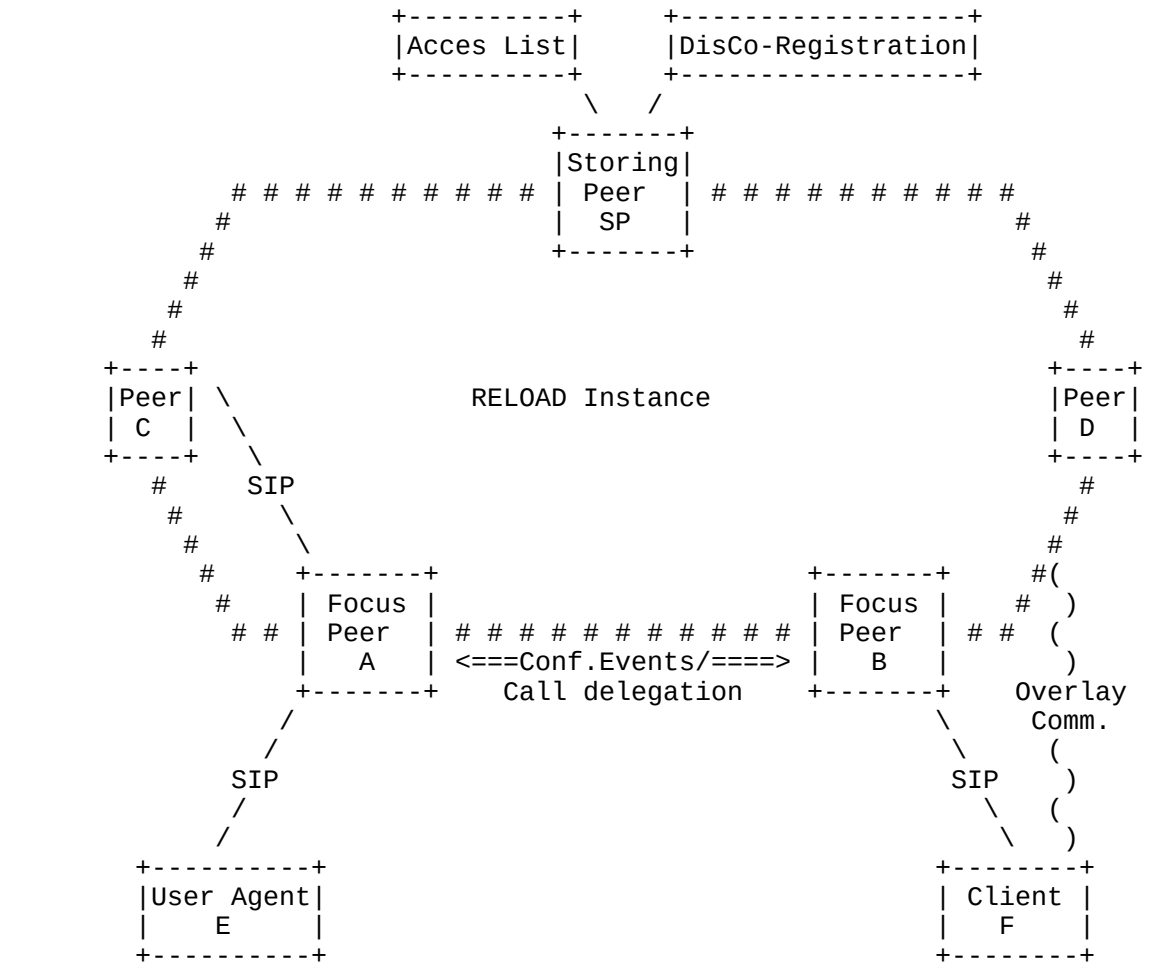


Figure 1: Reference Scenario: Focus peers A,B maintain a distributed conference

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.

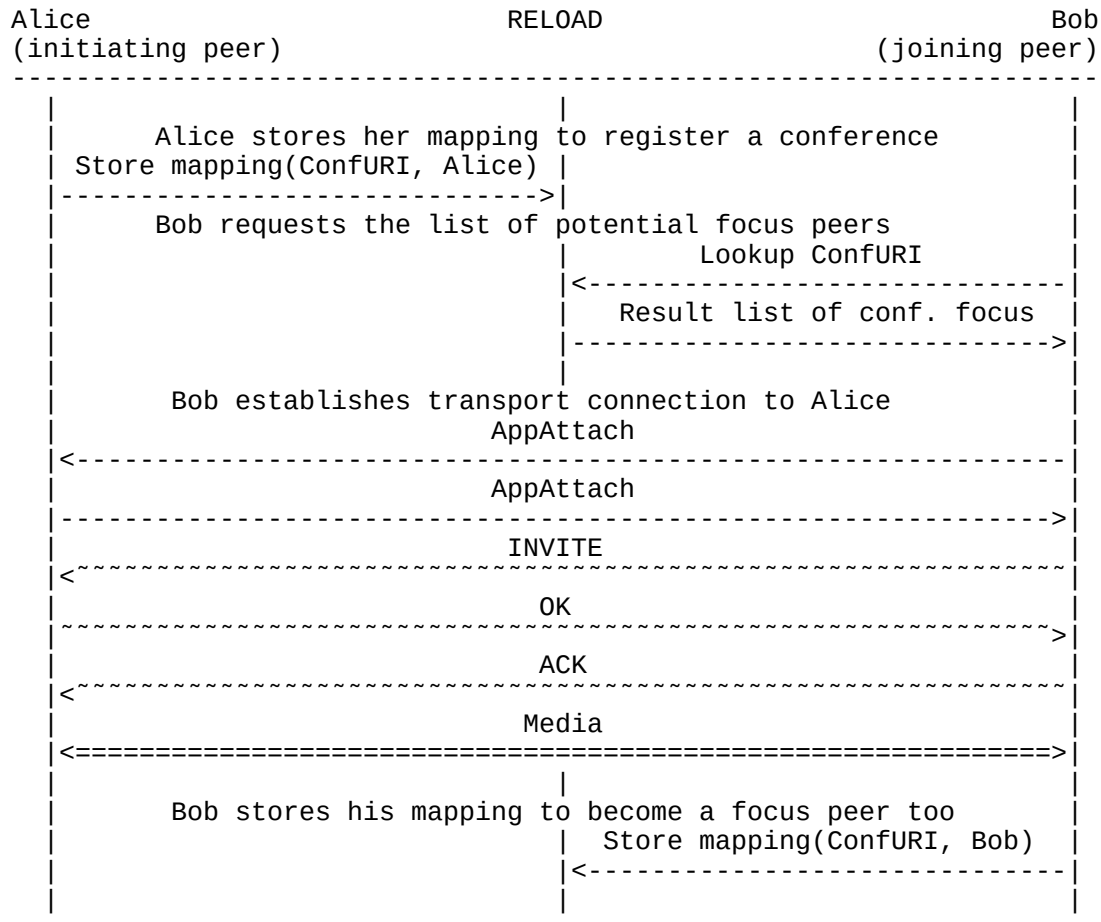


Figure 2: DisCo Usage generic Call Flow

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

local conference state document to represent a valid view of current total conference state.

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 loose 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 concept of Shared Resources [I-D.knauf-p2psip-share] is applied the DisCo-Registration to allow joint maintenance thereof by multiple focus peers. Thus, write permission to the resource location of a distributed conference's registration is shared by the conference creator with all authorized focus peers.

DisCo complies with the following requirements for Shared Resources:

Separated Data Storage: DisCo uses the dictionary data model. Each dictionary entry is associated with a single peer: a dictionary value MUST be written (or overwritten) if and only if the store request is signed with a private key associated with a certificate whose Node-ID is equal to the dictionary key.

Access Control Policy: Authorized focus peers are allowed to write the DisCo-Registration using the USER-CHAIN-ACL access policy. The conference creator (and resource owner) is the only exception: he is allowed to write based on the USER-MATCH or USER-PATTERN-MATCH policy.

User_name field: Each DisCo-Registration data provides a user_name field, which contains the user name of the originator of the stored data.

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:

```
struct {
  opaque resource_name<0..16^-1>;
  opaque user_name<0..16^-1>;
  opaque coordinate<0..2^16-1>;
  NodeId node_id;
} DisCoRegistrationData;

struct {
  uint16 length;
  DisCoRegistrationData data;
} DisCoRegistration;
```

The DisCoRegistration structure is composed of the following values:

length: This field denotes the length of the DisCoRegistrationData.

data: This field denotes the distributed conference registration data.

The DisCoRegistrationData structure is composed of the following values:

resource_name: This field contains the conference URI and meets the requirement for the USER-PATTERN-MATCH access policy defined in [I-D.knauf-p2psip-share]

user_name: This field contains the user name of the focus peer that stores the DisCoRegistration data. It meets the requirements for USER-CHAIN-ACL access policy defined in [I-D.knauf-p2psip-share].

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 NodeId 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 flexible Resource Name. The variable conference identifier follows the requirements for Kinds using variable Resource Names as defined in the ShaRe Usage [I-D.knauf-p2psip-share]. The ’kind’-attribute of the <pattern>-element used for DisCo-Registrations contains the string "DISCO-REGISTRATION".

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 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 List 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 Access List exist, the conference creator stores a valid conference identifier (see Section 4.3) and an Access List referring to the DisCo-Registration Kind-ID.

Else, if the conference creator has cached the previous Access List Kind it refreshes the root item of the existing Access-List and stores its registration as focus peer into the DisCo-Registration Kind. If no cached Kind data is present, the conference creator fetches the Access List Kind to determine the index of its root item in the Access List.

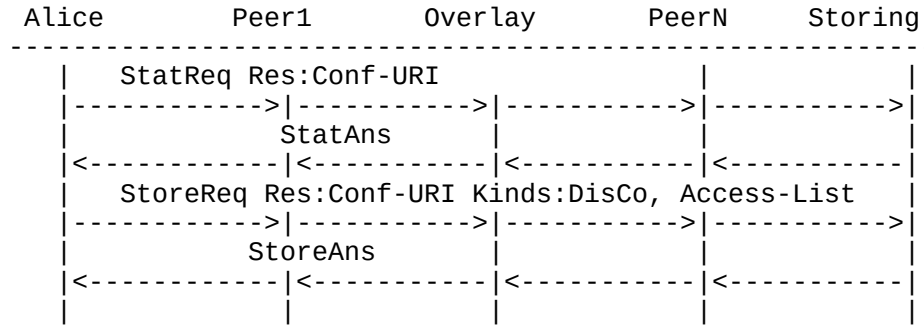


Figure 3: Creation of a Distributed Conference

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 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 constuct 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 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.

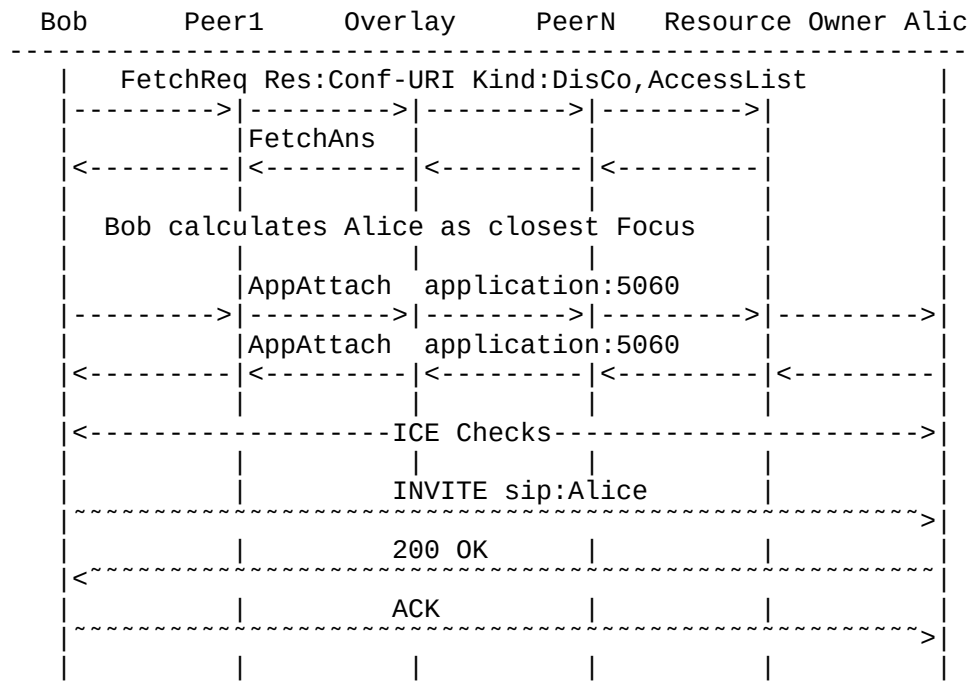


Figure 4: Participation of a Distributed Conference

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

2. The joining peer sends a FetchReq message for the DisCo and Access List Kind to the Resource-ID that corresponds to the hash over the conference URI using the overlay's hash-function. The FetchReq SHOULD NOT include any specific dictionary keys or array ranges in order to receive all focus peers of the conference. With the Access List data, the joining peer is able to verify which DisCo-Registrations are stored by authorized focus peers (see [I-D.knauf-p2psip-share]).

3. Using the retrieved coordinates values of the DisCo-Registrations, the joining peer MAY calculate which of the focus peers is the relatively closest to itself.

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

5. 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 Access List item that delegates write permission for the DisCo-Registration to the new participant. It sets the 'from_user' field in the Access List 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 Access List 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.

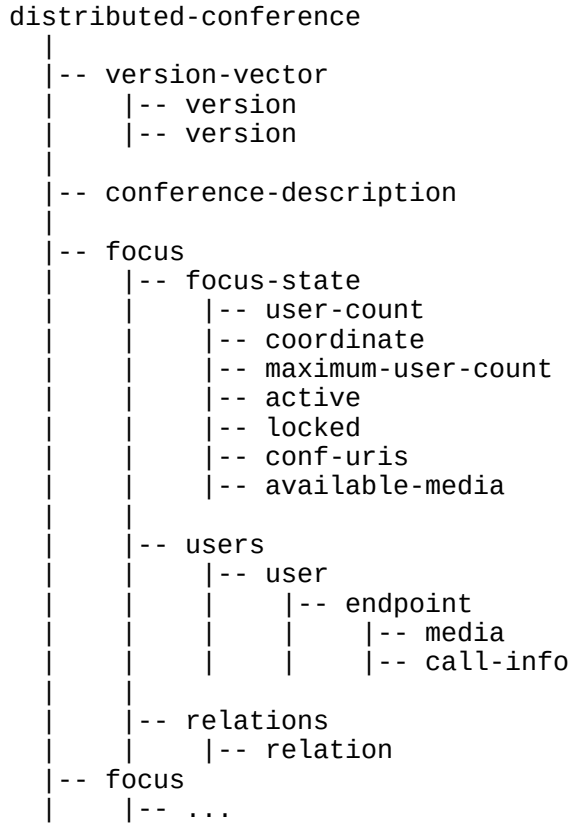


Figure 5: Overview of the event package for distributed conferences

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 an 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 it local XML document. If a received <version> number is higher that 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 a 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 sub elements 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 attributed 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 peers 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 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.

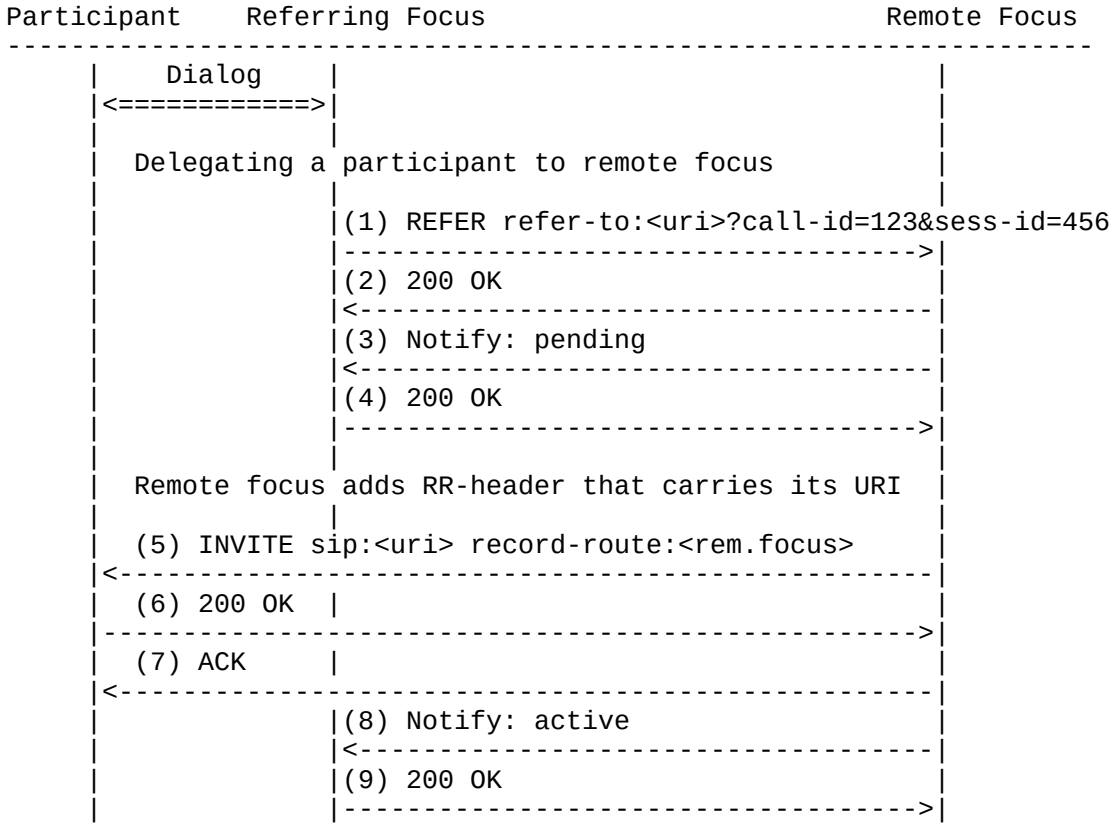


Figure 6: Delegating a participant with SIP REFER

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 steams 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 used in 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:

- 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 lasted version of the DisCo-Registration to obtain all potential focus peers.

- 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.

- 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-MATCH: By the conference creator
or
USER-PATTERN-MATCH: By the conference creator
or
USER-CHAIN-ACL: By authorized focus peers

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 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"/>
<xs:import namespace="http://www.w3.org/XML/1998/namespace"
     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>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:complexType>
  <!--
      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: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: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:

```
<!--
    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]
Jennings, C., Lowekamp, B., Rescorla, E., Baset, S., and H. Schulzrinne, "REsource LOcation And Discovery (RELOAD) Base Protocol", draft-ietf-p2psip-base-12 (work in progress), November 2010.

[I-D.knauf-p2psip-share]
Knauf, A., Hege, G., Schmidt, T., and M. Waehlisch, "A Usage for Shared Resources in RELOAD (ShaRe)", draft-knauf-p2psip-share-00 (work in progress), March 2011.

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

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

[RFC3264] Rosenberg, J. and H. Schulzrinne, "An Offer/Answer Model with Session Description Protocol (SDP)", RFC 3264, June 2002.

[RFC3265] Roach, A., "Session Initiation Protocol (SIP)-Specific Event Notification", RFC 3265, June 2002.

[RFC3515] Sparks, R., "The Session Initiation Protocol (SIP) Refer Method", RFC 3515, April 2003.

[RFC4574] Levin, O. and G. Camarillo, "The Session Description Protocol (SDP) Label Attribute", RFC 4574, August 2006.

[RFC4575] Rosenberg, J., Schulzrinne, H., and O. Levin, "A Session Initiation Protocol (SIP) Event Package for Conference State", RFC 4575, August 2006.

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

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

13.2. 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.ietf-p2psip-sip]
Jennings, C., Lowekamp, B., Rescorla, E., Baset, S., and H. Schulzrinne, "A SIP Usage for RELOAD", draft-ietf-p2psip-sip-05 (work in progress), July 2010.

[RFC4353] Rosenberg, J., "A Framework for Conferencing with the Session Initiation Protocol (SIP)", RFC 4353, February 2006.

[landmarks-infocomm02]
Ratnasamy, Handley, Karp, and Shenker, "Topologically-Aware Overlay Construction and Server Selection", Proc. of 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM ’02) pp. 1190-1199, 2002.

[timestamps-acsc88]
Fidge, C., "Timestamps in Message-Passing Systems that Preserve the Partial Ordering", Proceedings of 11th Australian Computer Science Conference, pp. 56-66, February 1988.

Appendix A. Change Log

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

P2PSIP Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 8, 2011

A. Knauf
G. Hege
T C. Schmidt
HAW Hamburg
M. Waehlisch
link-lab & FU Berlin

March 07, 2011

A Usage for Shared Resources in RELOAD (ShaRe)
draft-knauf-p2psip-share-00

Abstract

This document defines a RELOAD Usage for 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 September 8, 2011.

Copyright Notice

Table of Contents

1. Introduction

This document defines a RELOAD Usage for 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 of 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 List Kind that maintains a chain of Authorized Peers for a particular Shared Resource. Additionally, this document defines the USER-CHAIN-ACL access control policy that enables a 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 Lists. An Access list contains the Kind-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 user name or Node Id in the peer’s certificate. This document extends these policies and allows a controlled write access for multiple users at a common Resource Id.

Additionally, this specification defines a new access control policy that enables RELOAD users to store Resources with a variable Resource Name. The USER-PATTERN-MATCH policy allows the storage of Resources whose name complies to a specific pattern. Definition of the pattern is arbitrary, but must contain the user name 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 List: The term Access 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 List Kinds that map trust delegations from user A to user B, where A is allowed to write a Shared Resource and Access 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 provide one or more RELOAD Kinds that are designated for a shared write access with other RELOAD users. The mechanism to share those Resource/Kind pairs with a group of users consists of two basic steps. Storage of the Resource/Kind pair to be shared and storage of an Access List to those Kinds. Access Lists are initiated by the Resource Owner and contain Access List items, each delegating the permission to write the shared Kind to a specific user called Authorized Peer. This trust delegation to the Authorized Peer can include the right to further delegate the write permission to the Shared Resource. For each shared Kind data, the Resource owner stores a root item that initiates an Access List. The result is a tree of trust delegations with the Resource Owner as trust anchor.

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

Separated Data Storage: The specifications in this document ensure that concurrent writing does not effect race conditions. Each data stored within a Shared Resource MUST be exclusively maintained by the RELOAD user that created it. Hence, Usages that allow the storage of Shared Resources MUST use a RELOAD data model consisting of multiple objects (e.g. Array or Dictionary), each assigned to a single user.

Access Control Policy: To ensure write access to Shared Resource by Authorized Peers, each Usage MUST permit the USER-CHAIN-ACL access policy (see Section 5.4) in addition to its regular access policies (USER-MATCH, USER-NODE-MATCH, etc.).

user_name field: To identify the originator of a stored value, the Kind data structure of a Resource allowing shared write access MUST define a <0..2^16-1> long opaque user_name value. It contains the user name value of the RELOAD certificate which was used to store and sign Kind data. The user_name field allows any consumer of the data to request the public key certificate of the originator of the stored data and to verify its provenance and integrity.

4. Access List Definition

4.1. Access List

An Access List in this document specifies a logical list of AccessList data structures defined in Section 4.2. Each entry delegates write access to specific Kind data and is stored at the same overlay location as the Shared Resource. It allows the RELOAD user who is authorized to write at a specific Resources-ID to delegate his exclusive write access for the specified Kinds to further users of a RELOAD instance. Each Access List data structure therefore carries the information about who delegates write access to whom, the Kind-ID of the Resource to be shared, and whether delegation includes write access to the Access List itself. The latter condition grants the right to delegate write access further for an Authorized Peer. Access Lists are stored within a RELOAD array data model and are initially created by the Resource Owner.

Figure 1 shows an example of an Access List. The array entry at index #0 displays the initial storage of an Access list to a Shared Resource with Kind-ID 1234 at the same Resource-ID. It represents the root item of the trust delegation tree to the shared RELOAD Kind and initiates an Access List to the specified Kind data. The root entry MUST contain the mapping from Resource owner to Resource owner and MUST only be written by the owner of the public key certificate to this Resource-ID.

The array entry at index #1 represents the first trust delegation to an Authorized peer that is permitted write access to the Shared Resource with Kind-ID 1234. Additionally, the Authorized peer Alice is also granted write access to the Access List as indicated by the allow_delegation flag (AD) set to 1. It authorizes Alice to store further trust delegations to the Shared Resource, respectively, store items into the Access List. For instance, Alice permits Bob to access the Shared Resource, but Bob in turn is not allowed to write the Access List (AD = 0). The Authorized Peer Alice signs the Access List item with her own private key.

In order to share multiple Kinds at a single location, the Resource Owner can initiate new Access Lists that are referencing to another Kind-IDs as shown in array entry index #42. Note that overwriting existing items in an Access List that reference a different Kind-ID, revokes all succeeding trust delegations in the tree. Hence, Authorized Peers are not enabled to overwrite any existing Access List item (see Section 5.2). The Resource Owner is allowed to overwrite existing Access List items, but should be aware of its consequences.

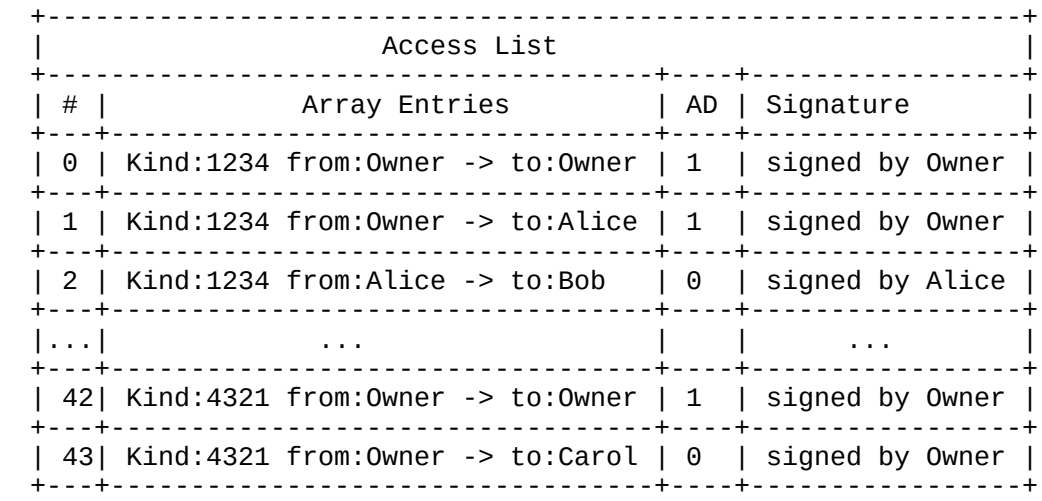


Figure 1: Access list example

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

4.2. Data Structure

The Kind data structure for the access list is defined as follows:

```
struct {
  opaque resource_name<0..2^16-1>;
  KindId kind;
  opaque from_user<0..2^16-1>;
  opaque to_user<0..2^16-1>;
  Boolean allow_delegation;
} AccessListData;

struct {
  uint16 length;
  AccessListData data;
} AccessListItem;
```

The AccessListItem structure is composed of:

length:
    Length of the Access List data structure
data:

Data of the Access List

The content of the AccessListData structure is defined as follows:

resource_name: This opaque string contains the Resource Name of the Shared Resource in an opaque string. Thus, the AccessListData meet the requirements for the USER-PATTERN-MATCH access policy (see Section 6.1).

kind: This field contains the Kind-ID of the Kind that will be shared.

from_user: This field contains the user name of that RELOAD peer the grants write permission to the Shared Resource. The user name is stored as an opaque string and contains the user name value of the certificate that is associated with the private key that signed this Access List item.

to_user: This field contains the user name of the RELOAD peer that obtains writing permission to the Shared Resource.

allow_delegation: This Boolean flag indicates if true, that the Authorized peer in the 'to_user' field is allowed write access to the Access List in order to delegate the write permission to the Shared Resource to further users.

The ACCESS-LIST kind is defined as follows:

Name ACCESS-LIST

Data model The Data model for the ACCESS-LIST data is array.

Access Control Initial storages of ACCESS-LIST data by the Resource Owner use the same Access Control Policy as the Shared Resource. For instance, if the access policy for the Shared Resource is USER-NODE-MATCH, then the access policy for the ACCESS-LIST data is USER-NODE-MATCH. Storages by Authorized Peers use the USER-CHAIN-ACL access policy (see Section 5.4).

5. Access Control to Shared Resources

5.1. 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. If the Resource owner shares an existing Resource/Kind pair, it should ensure that it does not unintentionally overwrite an existing Access List item. Hence, before sharing the Resource, its owner performs a fetch request for the Access List Kind that requests the entire array. If the retrieved array does not contain an Access List root item to the desired Kind, the Resource Owner stores a new root item for the desired Kind-ID and sets the AccessListData vales 'from_user' and 'to_user' to the user name of the Resource Owner. If an Access List root item exists, the Resource Owner delegates write access by storing an Access List item setting the 'from_user' to its user name and setting the 'to_user' equal to the name of the RELOAD user that obtains write access.

If an Authorized Peer intents to delegate write access to a Shared Resource, it likewise fetches the entire array of the Access List Kind to prevent an unauthorized write attempt to an existing Access List item. Afterwards it delegates write access to the specified Kind by storing an Access List item setting the 'from_user' value to its own user name and setting the 'to_user' value to RELOAD user that obtains write access. Note, that an Authorized Peer is only allowed to add items into an Access List it is registered in with the 'allow_delegation' flag set to true.

5.2. Revoking Write Access

Write permissions MAY be revoked by storing a non-existent value [I-D.ietf-p2psip-base] to the corresponding item in the Access Control List. A revoked permission automatically invalidates all delegations performed by that user and also 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 Access List item MUST only be written by the user who initially stored the corresponding entry. The only exception is by the Resource Owner that is allowed to overwrite Access list items at all times with a non-existent value for revoking write access.

5.3. Storage and Validation

5.3.1. Operations of the Storing Peer

The storing peer (the peer at which Shared Resource and Access List are physically stored) is responsible for enforcing the correct access policy when it is requested to store values of a Shared Resource. The storing peer first checks, whether the request is signed with the private key that corresponds to a certificate valid for this Resource-ID as enforced by the standard access policies (USER-MATCH, USER-NODE-MATCH, etc.) defined in the RELOAD base protocol [I-D.ietf-p2psip-base], or the policy USER-PATTERN-MATCH defined in this document (see Section 6.1).

If not, the storing peer continues by checking whether any of the received RELOAD Kinds of the store request allows the USER-CHAIN-ACL access control policy. If so, the storing peer fetches the Access Lists for those Kinds and enforce the USER-CHAIN-ACL access policy (see Section 5.4). Since the Access list MUST be stored at the same overlay location as the Shared Resource, this operation is a local lookup.

Analogously, a storing peer that is requested to store an Access List Kind first verifies whether the requester is allowed to store values at this Resource-ID by its certificate. Otherwise the storing peer MUST locally fetch the Access List of the requested Resource Id and enforce the USER-CHAIN-ACL policy.

5.3.2. Operations of the Accessing Peer

An accessing peer (a RELOAD peer that fetches a Shared Resource) SHOULD validate the provenance and integrity of a retrieved data value and the authorization of the data originator. The latter is verified using the Access List Kind. The accessing peer requests all Access Lists that are stored under the same Resource-ID as the Shared Resource by requesting the entire array range. This request could be sent in the same fetch request as the request for the Shared Resource. The accessing peer then checks, whether any of these Access Lists refers to the Kind of Shared Resource by its Kind-ID. If true, the accessing peer compares the 'to_user' value of each Access List item with the mandatory user_name value of the Shared Resource for equality. If the comparison fails, the accessing peer MUST ignore the data of the retrieved Shared Resource. Else, the accessing peer repeats this comparison with the value of the 'from_user' field of this item with each 'to_user' field of the Access List. This procedure continues until both 'from_user' and 'to_user' values are equal. The accessing peer then hashes the 'from_user' using the hash function of the overlay algorithm. If the

hash is equal to the Resource-ID of the Shared Resource, the authority of the originator of the stored data is validated. The accessing peer then proceeds with the provenance and integrity tests.

The accessing peer verifies provenance and integrity of the retrieved kind data using the certificate corresponding to the mandatory user_name field of the Shared Resource entry. The certificate can be retrieved by applying the Certificate Usage [I-D.ietf-p2psip-base] or other means (e.g., caching from a previous request).

The accessing peer MAY cache previously fetched Access List to a maximum of the individual items' lifetimes. Since stored values could have been changed or invalidated prior to their expiration an accessing peer uses a stat request to check for updates before using the cached data. If a change has been detected it fetches the latest Access List.

5.4. 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. Access is controlled by the values stored within the Access List Kind that explicitly permits Authorized Peers writing access to Shared Resources or the Access List (or both) by their user name. Hence, if a request in not signed with a private key that allows write access to a Resource by any access control policy defined in the RELOAD base specification, a storing peer MUST enforce the USER-CHAIN-ACL policy:

When accessing the Shared Resource, a given value MUST be written or overwritten if and only if the request is signed with a key that is associated with a certificate whose user name is stored in any 'to_user' value of an Access List associated to the Shared Resource. If true, this comparison has to be repeated for the 'from_user' value of that Access List item with each other 'to_user' value in this Access List. This procedure continuous until 'from_user' and to 'to_user' are equal. Then, if the hash over the 'from_user' equals the Resource-ID, the requester is authorized to write the Shared Resource.

When accessing the Access List, a given value MUST be written or overwritten if and only if the request is signed with a key that is associated with a certificate whose user name is stored in any 'to_user' value in the same Access List as the requested Access List. Additionally, the 'allow_delegate' value of this Access List item MUST be set true. If this query is successes, the comparison has to be repeated for the 'from_user' value of that Access List item with

each other 'to_user' value in the Access List. This procedure continuous until 'from_user' and to 'to_user' are equal. Then, if the hash over the 'from_user' equals the Resource-ID, the requester is authorized to write the Access List.

6. Extension for Variable Resource Names

In certain use cases such as conferencing (c.f. [I-D.knauf-p2psip-disco]) it is desirable to extend the set of Resource Names and thus Resource-IDs a peer is allowed to write beyond those defined through the user name or NodeId fields in its certificate. This is accomplished by the USER-PATTERN-MATCH access policy described here.

Each RELOAD node uses a certificate to identify itself using its user name (or Node-ID) while storing data under a specific Resource-ID. The USER-PATTERN-MATCH scheme follows this paradigm by allowing to store values whose Resource Name is derived from the user name in the certificate of a RELOAD peer, but extends the set of allowed Resource Names. This is done by using a Resource Name which contains a variable substring but matches the user name in the certificate using a pattern defined in the configuration document. Thus despite being variable an allowed Resource Name is closely related to the Owner's certificate. A sample pattern might be formed as the following:

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

When defining the pattern care must be taken that no conflict arises for two user names of witch one is a substring of the other. In this case the peer with the name which is the substring could choose the variable part of the Resource Name so that the resulting string contains the whole other user name and thus he could write the other user's resources. This can easily be prevented by delimiting the variable part of the pattern from the user name part by some fixed string, that is usually not part of a user name (e.g. the "-conf-" in the above Example).

6.1. USER-PATTERN-MATCH Access Policy

Thus, using the USER-PATTERN-MATCH policy, a given value MUST be written or overwritten if and only if the request is signed with a key that is associated with a certificate whose user name matches the Resource Name using the pattern specified in the configuration document. The Resource Name MUST be taken from an opaque resource_name field in the corresponding Kind data structure. Hence, each RELOAD Usage that utilizes the USER-PATTERN-MATCH policy, MUST define an opaque resource_name field within the Kind data structure, that contains the Resource Name whose hash equals the Resource-ID.

6.2. Overlay Configuration Document Extension

This document 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.

Each <pattern> element defines the pattern to be used for a single Kind. It is of type xsd:string, which is interpreted as a regular expression. In the regular expression $USER and $DOMAIN are used as variables for the corresponding parts of the string in the certificate user name field ($USER before and $DOMAIN after the '@'). Both variables MUST be present in any given pattern. The <pattern> element has the attribute "kind" which contains the Kind name for which this pattern is used.

A <pattern> element MUST be present for every Kind for which the variable resource names extension is allowed in an overlay.

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

```
<!--
    VARIABLE RESOURCE NAMES ELEMENT
-->
parameter &= element variable-resource-names {
    <!--
        RESOURCE NAME PATTERN ELEMENT
    -->
    element pattern {
        attribute kind { xsd:string },
        xsd:string
    }*
}?
```

7. Security Considerations

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

7.1. 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 ResourceId 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 the maximum number of entries a single peer is allowed to write at a single location.

7.2. 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.

7.3. 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.

8. IANA Considerations

TODO: register Kind-ID code point at the IANA

9. 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. This work was party funded by the German Federal Ministry of Education and Research, projects HAMcast and Mindstone.

10. References

10.1. 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", draft-ietf-p2psip-base-12 (work in progress), November 2010.

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

10.2. 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-01 (work in progress), December 2010.

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

P2PSIP
Internet-Draft
Intended status: Informational
Expires: September 15, 2011

Y. Peng
W. Wang
ZTE Corporation
J. Peng
L. Le
China Mobile
Z. Hao
Y. Meng
ZTE Corporation

March 14, 2011

Network Management Scenarios for RELOAD
draft-peng-p2psip-network-management-scenarios-02

Abstract

The RELOAD protocol can be applied in different kinds of scenarios, including the Internet, carrier's dedicated network, enterprise network, etc. This document summarizes the network management scenarios by analyzing typical application model for each of the above three kinds of scenarios.

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 15, 2011.

Copyright Notice

Table of Contents

1. Introduction

The RELOAD protocol is a peer-to-peer (P2P) signaling protocol, which provides an abstract storage and messaging service between a set of cooperating peers that form the overlay network. It can be applied in different kinds of scenarios, including the Internet, carrier's dedicated network, enterprise network, etc. This document summarizes the network management scenarios by analyzing typical application model for each of the above three kinds of scenarios.

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.

SNMP: Simple Network Management Protocol.

3. Network Management Scenarios for RELOAD Applied on The Internet

There are a variety of application models for RELOAD on the Internet, this document only analyses one of the typical application model named "Public P2P VoIP Service Providers" [cite P2PVoIP scenario]. As stated in the draft of application scenarios for RELOAD, centralized operation and management is required for RELOAD in this application model. Here we will analyse two aspects of the network management requirements for this application model.

From the viewpoint of the service provider, they need to ensure network stability and efficient operation. On the one hand, the provider needs to monitor and control their own devices, and view network utility and load of the devices; on the other hand, because of its openness to user nodes, it is necessary to prevent malicious user nodes from attacking the service network and abnormal user nodes from disturbing the service network, so the action of user nodes need be monitored and controlled. Furthermore, human intervention may be needed when the built-in mechanisms in RELOAD is not able to solve the network problems.

From the viewpoint of administrator of enterprise users who use the service from the public P2P VoIP service provider, they need to ensure their own network security, defense external network attacks

and viruses; It is needed to prevent commercial secret leakage; It is also needed to limit user function to prevent misuse enterprise VoIP services; It is needed to reasonably distribute traffic flows to improve user experience using limited bandwidth. As an example, Skype plans to provide administration tools for enterprise users to resolve the problem that Skype may be blocked by enterprise networks. (Note: This quotes from http://www.best-voip-review.com/news/2010_06_Skype_offers_network_management_tools_to_control_their_software.htmlz)

According to the above requirements, we put forward the following network management scenarios for RELOAD used on the Internet:

a. Monitoring and controlling the service provider's own devices, such as viewing and modifying the configuration parameters of the devices, monitoring the load and running state of accessed nodes(or super nodes) and the number of client nodes that access accessed nodes(client nodes include RELOAD client and application client).

b. Viewing and collecting various network data, such as routing table, the data stored in nodes, real-time status information of nodes, in order to find out the operational status of the network, such as capacity of network, quality of service, network topology information. These are the decision-making basis for optimization and adjustment of the network.

c. Alarm for abnormal events, such as congestion, message process failures, routing failures, link failures, etc.

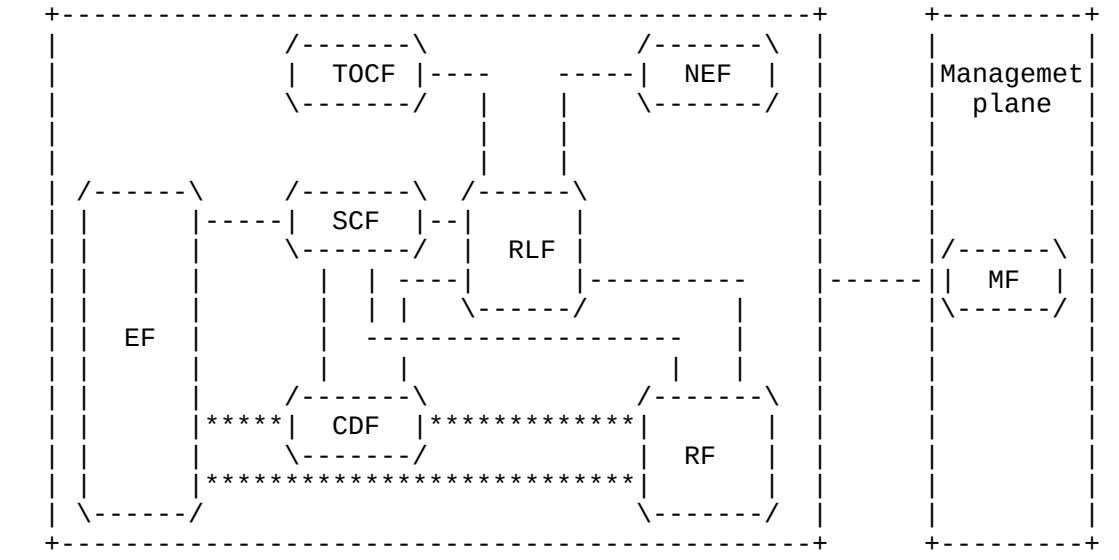
d. Disposal of abnormal nodes, for example, finding illegal user nodes and forcing it to exit the network, finding fault nodes that can't perform RELOAD related responsibilities and requiring them to rejoin or forcing them to quit. These ensure the health of the network.

e. Providing network management tool for enterprise users. The administrator of the enterprise may control specific data flow through RELOAD nodes, and limit application functions, and configure the communication port by this tool.

4. Network Management Scenarios for RELOAD Applied on Carrier's Dedicated Network

DHT-based VoIP service is studied in DSN project of ITU-T (SG13 Q19). Its architecture and processes were developed by referring RELOAD. Although it has not yet been clearly adopted which protocol will be

adopted in the project, but it is feasible that RELOAD is used here. And it is very possible that RELOAD is adopted by the VoIP service of DSN. The following figure is the architecture figure defined in DSN.



RLF in the above figure is equivalent to the peer of RELOAD. And NEF is equivalent to the Enrollment server of RELOAD. And MF is the function of network management. The specific descriptions of every function are as follows:

RLF (Resource Location Functions):

Resource registration;

Locate resources (content, Relay Node, subscription data, service capability, etc.);

Store and maintain resource information;

Routing of DSN message;

Construction and management (such as updating routing tables, transferring resources when a new node joins the overlay network, and so on.)of overlay network;

Retrieve optimization information from TOCF(Traffic Optimization Control Functions).

NEF (Node Enrolment Functions):

Provide bootstrap information for the DSN Node to join the DSN;

Assign globally unique Node ID to the DSN Nodes;

Configure parameters and information related to joining of the DSN Node;

Apply Authentication/Authorization to DSN Node;

Maintain the Node profile of all enrolled nodes (e.g. Node ID, Zone information and etc), which can be requested by RLF;

MF (Management Functions):

DSN network and service administration

DSN monitoring and diagnosis

Statistics and Accounting which includes collection of information related to usage and contribution of DSN services.

It is a telecom class application network built by telecom operators, and the Management Function is clearly defined in the architecture draft so as to achieve network management. In this kind of applications, the core network devices are provided by telecom operators. As the number of the devices is large and network topology changes frequently, it is very difficult to manage devices one by one, so the need for centralized operation and management is obvious. Moreover, the existing telecom networks were equipped with the appropriate network management systems. In this kind of application model, we will analyze the needs for network management mainly from the perspective of network operators:

Firstly, in order to ensure network stability and efficient operation, the network operators need to monitor and control the network devices to ensure utility and load of the core devices.

Secondly, the network operators need to effectively locate failure when an exception occurs in the system.

For these requirements, we propose the following specific network management scenarios:

a. Monitoring and controlling network devices. Such as viewing and modifying the configuration parameters of the devices, monitoring load and running state of the devices, controlling the functions

and roles of these devices in the network.

b. Viewing and collecting various network data, such as routing table, the data stored in nodes, real-time status information of nodes, in order to find out the operational status of the network, such as the capacity of network, the quality of service, network topology information. These are the decision-making basis for network optimization and adjustment.

c. Abnormal nodes alarm, such as congestion, message process failures, routing failures, link failures, etc.

d. When an exception occurs, the operators may find the location and the cause of failure by tracing process flow and signaling or doing diagnostic test. It will provide effective help to resolve the failure.

5. Network Management Scenarios for RELOAD Applied on Enterprise Network

RELOAD can be applied in a variety of application models in controlled private network, which was put forward in the draft of application scenarios for RELOAD. The need for centralized operation and management was clearly stated in the application model named "P2P for Redundant SIP Proxies" in this draft. This document analyses the need of network management only for this kind of application model.

Firstly, in order to ensure network stability and efficient operation, the IT department of enterprise needs to monitor and control network devices to ensure reasonable utilization rate and no overload.

Secondly, the IT department of enterprise needs to ensure the network security, to defense external network attacks and viruses; It is needed to prevent commercial secret leakage; It is needed to limit user functions to prevent misuse of network resources; It is needed to reasonably distribute traffic flows to improve the user’s experience under the limited bandwidth.

Thirdly, the network operators need to effectively locate failure when an exception occurs in the system.

For these requirements, we propose the following specific network management scenarios:

a. Monitoring and controlling the network devices, such as viewing and modifying the configuration parameters of the devices, monitoring load and running state of the accessed nodes(or super nodes) and the number of client nodes that access accessed nodes(client nodes include RELOAD Client and Application Client).

b. Viewing and collecting various network data, such as routing table, the data stored in nodes, real-time status information of nodes, in order to find out the operational status of the network, such as the capacity of the network, the quality of service, network topology information. These are the decision-making basis for optimization and adjustment of the network.

c. Abnormal nodes alarm, such as congestion, message process failures, routing failures, link failures, etc.

d. Disposal of abnormal nodes, for example, finding illegal user nodes and forcing it to exit the network, finding fault nodes that can’t perform RELOAD related responsibilities and requiring them to rejoin or forcing them to quit, and doing corresponding treatment. These ensure the health of the network.

e. The manager may control specific data flow through RELOAD nodes, and limit application functions, and configure the communication port, in order to control the actions of users.

f. When an exception occurs, the operators may find the location and the cause of failure by tracing process flow and signaling or doing diagnostic test. It will provide effective help to resolve the failure.

6. Summary of the Network Management Scenarios for RELOAD

Differences Among These Scenarios

Applications Category / Network Management Scenarios	Internet	Carrier's Dedicated Network	Enterprise Network
Applications Scenarios	"Public P2P VoIP Service Providers" in the RELOAD scenarios draft	Carrier's dedicated network application in the DSN project of ITU-T	"P2P for Redundant SIP Proxies" in the RELOAD scenarios draft
Monitoring Dedicated Device	Y	Y	Y
Viewing Network Data	Y	Y	Y
Fault Alarming	Y	Y	Y
Disposing Malicous/Fault User Nodes	Y		Y
Controlling User Node	Y		Y
Troubleshooting Quickly	Y (It is said that Skype will provide this tool to solve the problem of blocking by enterprise)	Y	Y
Control Level of Network Management	Loose	Strict	Medium

According to above analysis, we think whether RELOAD is applied on the Internet or carrier's dedicated network or enterprise network, network management is always involved in some application models and scenarios. So it is necessary to study the network management solution for RELOAD and to define its corresponding implementation protocol.

7. Relationship between Network Management Protocol and Diagnostic Protocol

A diagnostic mechanism was proposed in [I-D.ietf-p2psip-diagnostics], which is an extension to RELOAD protocol and defines the method how to monitor the connection between peers and the node status. While the SNMP usage for reload protocol focus on how to apply SNMP to manage DHT overlay considering its particular network circumstance. There are some correlations between the network management protocol and the diagnostic protocol. But they are applied respectively between different network elements. The network management protocol is used between the network management server and the managed peers. The diagnostic protocol is used between two peers in the overlay. These two protocols can fulfill network management functions through collaboration. For example, the network management server sends SNMP message to Peer to trigger diagnostic operation. After RELOAD Peer receives the message, it will do diagnostic test through RELOAD diagnostic message and generate result data. Finally, this REOAD Peer will report the diagnostic result to the network management server through SNMP message.

8. Security Considerations

The security requirements of the various application scenarios vary tremendously. They should be discussed in more detail in this document.

9. IANA Considerations

This document has no IANA Considerations.

10. 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 "Application Scenarios for Peer-to-Peer Session Initiation Protocol" draft by D. Bryan, E. Shim, B. Lowekamp, S. Dawkins, Ed.

Thanks to the many people of the IETF P2PSIP WG whose many drafts we have learned.

11. References

11.1. Normative References

[I-D.ietf-p2psip-app-scenarios]
Bryan, D., Shim, E., Rescorla, E., Lowekamp, B., Dawkins, S., and Ed. , "Application Scenarios for Peer-to-Peer Session Initiation Protocol", November 2007.

[I-D.ietf-p2psip-base]
Jennings, C., Lowekamp, B., Rescorla, E., Baset, S., and H. Schulzrinne, "REsource LOcation And Discovery (RELOAD)Base Protocol", November 2010.

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

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", July 2008.

[I-D.narten-iana-considerations-rfc2434bis]
Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", draft-narten-iana-considerations-rfc2434bis-09 (work in progress), March 2008.

[RFC2629] Rose, M., "Writing I-Ds and RFCs using XML", RFC 2629, June 1999.

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, July 2003.

Appendix A. Additional Stuff

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.wei108@zte.com.cn

Jin Peng
China Mobile Communication Corporation
Beijing,
China

Phone: +86 13911281193
Email: pengjin@chinamobile.com

LiFeng Le
China Mobile Communication Corporation
Beijing,
China

Phone: +86 13910019925
Email: lelifeng@chinamobile.com

ZhenWu Hao
ZTE Corporation
Nanjing, 210012
China

Phone: +86 13382087596
Email: hao.zhenwu@zte.com.cn

Meng Yu
ZTE Corporation
Nanjing, 210012
China

Phone: +86 18651806839
Email: meng.yu@zte.com.cn

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2011

M. Petit-Huguenin
Stonyfish, Inc.

March 13, 2011

Configuration of Access Control Policy in REsource LOcation And Discovery (RELOAD) Base Protocol
draft-petithuguenin-p2psip-access-control-01

Abstract

This document describes an extension to the REsource LOcation And Discovery (RELOAD) base protocol to distribute the code of new Access Control Policies without having to upgrade the RELOAD implementations in an overlay.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79. This document may not be modified, and derivative works of it may not be created, except to format it for publication as an RFC or to translate it into languages other than English.

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 14, 2011.

Copyright Notice

Table of Contents

1. Introduction

The RELOAD base protocol specifies an Access Control Policy as "defin[ing] whether a request from a given node to operate on a given value should succeed or fail." The paragraph continues saying that "[i]t is anticipated that only a small number of generic access control policies are required", but there is indications that this assumption will not hold. On all the RELOAD Usages defined in other documents than the RELOAD base protocol, roughly 50% defines a new Access Control Policy.

The problem with a new Access Control Policy is that, because they are executed when a Store request is processed, they need to be implemented by all the peers and so require an upgrade of the software. This is something that is probably not possible in large overlays or on overlays using different implementations. For this reason, this document proposes an extension to the RELOAD configuration document that permits to transport the code of a new Access Control Policy to each peer.

This extension defines a new element <access-control-code> that can be optionally added to a <kind> element in the configuration document. The <access-control-code> element contains ECMAScript [ECMA-262] code that will be called for each StoredData object in a StoreReq processed by a peer. The code receives four parameters, corresponding to the Resource-ID, Signature, Kind and StoredDataValue of the value to store. The code returns true or false to signal to the implementation if the request should succeed or fail.

For example the USER-MATCH Access Control Policy defined in the base protocol could be redefined by inserting the following code in an <access-control-code> element:

```
return resource.equalsHash(signature.user_name.bytes());
```

The <kind> parameters are also passed to the code, so the NODE-MULTIPLE Access Control Policy could be implemented like this:

```
for (var i = 0; i < kind.params['max-node-multiple']; i++) {
    if (resource.equalsHash(signature.node_id, i.width(4))) {
        return true;
    }
}
return false;
```

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].

3. Processing an extended Kind

A peer receiving a <kind> definition, either by retrieving it from the configuration server or in a ConfigUpdateReq message, MUST verify the signature in the kind-signature element before executing the code.

If the <access-control-code> element is present in the namespace allocated to this specification, and the Access Control Policy is not natively implemented, then the code inside the element MUST be called for each DataValue found in a received StoreReq for this Kind. For each call to the code, the following ECMAScript objects, properties and functions MUST be available:

resource: An opaque object representing the Resource-ID, as an array of bytes.
resource.equalsHash(Object...): Returns true if hashing the concatenation of the arguments according to the mapping function of the overlay algorithm is equal to the Resource-ID. Each argument is an array of bytes.
signature.user_name: The rfc822Name stored in the certificate that was used to sign the request, as a String object.
signature.node_id: The Node-ID stored in the certificate that was used to sign the request, as an array of bytes.
kind.id: The id of the Kind associated with the entry, as a Number object.
kind.name: The name of the Kind associated with the entry, as a String object.
kind.data_model: The name of the Data Model associated with the entry, as a String object.
kind.access_control: The name of the Access Control Policy associated with the entry, as a String object.
kind.params: An associative array containing the parameters of the Access Control Policy as specified in the configuration file.
  max-count: The value of the max-count element in the configuration file, as a String object.
  max-size: The value of the max-size element in the configuration file as a String object.

max-node-multiple: If the Access Control is MULTIPLE-NODE, contains the value of the max-node-multiple element in the configuration file, as a String object. If not, this property is undefined.

entry.index: If the Data Model is ARRAY, contains the index of the entry, as a Number object. If not, this property is undefined.

entry.key: If the Data Model is DICTIONARY, contains the key of the entry, as an array of bytes. If not, this property is undefined.

entry.storage_time: The date and time used to store the entry, as a Date object.

entry.lifetime: The validity for the entry in seconds, as a Number object.

entry.exist: Indicates if the entry value exists, as Boolean object.

entry.value: This property contains an opaque object that represents the whole data, as an array of bytes.

The properties SHOULD NOT be modifiable or deletable and if they are, modifying or deleting them MUST NOT modify or delete the equivalent internal values (in other words, the code cannot be used to modify the elements that will be stored).

If addition to the "max-count", "max-size" and eventual "max-node-multiple" properties in the kind.params associative array, any extension element in any namespace found in the <kind> element MUST be added to this array, using the element name as key and the content as value.

The value returned by the code is evaluated to true or false, according to the ECMAScript rules. If the return value of one of the call to the code is evaluated to false, then the StoreReq fails, the state MUST be rolled back and an Error_Forbidden MUST be returned.

4. Security Considerations

TBD

5. IANA Considerations

No IANA considerations.

6. Acknowledgements

This document was written with the xml2rfc tool described in [RFC2629].

7. References

7.1. Normative References

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

[I-D.ietf-p2psip-base] Jennings, C., Lowekamp, B., Rescorla, E., Baset, S., and H. Schulzrinne, "REsource LOcation And Discovery (RELOAD) Base Protocol", draft-ietf-p2psip-base-12 (work in progress), November 2010.

[ECMA-262] Ecma, "ECMAScript Language Specification 3rd Edition", December 2009.

7.2. Informative References

[RFC2629] Rose, M., "Writing I-Ds and RFCs using XML", RFC 2629, June 1999.

[I-D.ietf-p2psip-service-discovery] Maenpaa, J. and G. Camarillo, "Service Discovery Usage for REsource LOcation And Discovery (RELOAD)", draft-ietf-p2psip-service-discovery-02 (work in progress), January 2011.

[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-01 (work in progress), December 2010.

Appendix A. Examples

A.1. Standard Access Control Policies

This section shows the ECMAScript code that could be used to implement the standard Access Control Policies defined in [I-D.ietf-p2psip-base].

A.1.1. USER-MATCH

```
   String.prototype['bytes'] = function() {
       var bytes = [];
       for (var i = 0; i < this.length; i++) {
           bytes[i] = this.charCodeAt(i);
       }
       return bytes;
   };

   return resource.equalsHash(signature.user_name.bytes());
```

A.1.2. NODE-MATCH

```
   return resource.equalsHash(signature.node_id);
```

A.1.3. USER-NODE-MATCH

```
   String.prototype['bytes'] = function() {
       var bytes = [];
       for (var i = 0; i < this.length; i++) {
           bytes[i] = this.charCodeAt(i);
       }
       return bytes;
   };

   var equals = function(a, b) {
       if (a.length !== b.length) return false;
       for (var i = 0; i < a.length; i++) {
           if (a[i] !== b[i]) return false;
       }
       return true;
   };

   return resource.equalsHash(signature.user_name.bytes())
     && equals(entry.key, signature.node_id);
```

A.1.4. NODE-MULTIPLE

```
Number.prototype['width'] = function(w) {
    var bytes = [];
    for (var i = 0; i < w; i++) {
        bytes[i] = (this >>> ((w - i - 1) * 8)) & 255;
    }
    return bytes;
};

for (var i = 0; i < kind.params['max-node-multiple']; i++) {
    if (resource.equalsHash(signature.node_id, i.width(4))) {
        return true;
    }
}
return false;
```

A.2. Service Discovery Usage

[I-D.ietf-p2psip-service-discovery] defines a specific Access Control Policy (NODE-ID-MATCH) that need to access the content of the entry to be written. If implemented as specified by this document, the <kind> element would look something like this:

```
<kind name='REDIR'
  xmlns:acp='http://implementers.org/access-control-policy'
  xmlns:ext='http://implementers.org/my-ext'>
    <data-model>DICTIONARY</data-model>
    <access-control>NODE-ID-MATCH</access-control>
    <max-count>100</max-count>
    <max-size>60</max-size>
    <ext:branching-factor>2</ext:branching-factor>

    <acp:access-control-code>
        /* Insert here the code from
           http://jsfromhell.com/classes/bignumber
         */

       var toBigNumber = function(node_id) {
           var bignum = new BigNumber(0);
           for (var i = 0; i < node_id.length; i++) {
               bignum = bignum.multiply(256).add(node_id[i]);
           }
           return bignum;
       };

       var checkIntervals = function(node_id, level, node, factor) {
           var size = new BigNumber(2).pow(128);
           var node = toBigNumber(node_id);
           for (var f = 0; f < factor; f++) {
```

```
                var temp = size.multiply(new BigNumber(f)
                  .pow(new BigNumber(level).negate()));
                var min = temp.multiply(node.add(new BigNumber(f)
                  .divide(factor)));
                var max = temp.multiply(node.add(new BigNumber(f + 1)
                  .divide(factor)));
                if (node.compare(min) === -1 || node.compare(max) == 1
                  || node.compare(max) == 0) return false;
            }
            return true;
        };

        var equals = function(a, b) {
            if (a.length !== b.length) return false;
            for (var i = 0; i < a.length; i++) {
                if (a[i] !== b[i]) return false;
            }
            return true;
        };

        var level = function(value) {
            var length = value[16] * 256 + value[17];
            return value[18 + length] * 256 + value[18 + length + 1];
        };

        var node = function(value) {
            var length = value[16] * 256 + value[17];
            return value[18 + length + 2] * 256
              + value[18 + length + 3];
        };

        var namespace = function(value) {
            var length = value[16] * 256 + value[17];
            return String.fromCharCode(value.slice(18, length));
        };

        return equals(entry.key, signature.node_id)
          && (!entry.exists || checkIntervals(entry.key,
            level(entry.value), node(entry.value),
            kind.params['branching-factor']))
          && (!entry.exists
            || resource.equalsHash(namespace(entry.value),
              level(entry.value), node(entry.value)));
    </acp:access-control-code>
</kind>
```

Note that the code for the BigNumber object was removed from this example, as the licensing terms are unclear. The code is available

at <http://jsfromhell.com/classes/bignumber>.

The <branching-factor> parameter is used to match the <redirBranchingFactor> parameter that is not accessible to the code. The signer of the kind must be sure that the two match. In fact the branching factor could have been set directly in the code, but that would make it more difficult to change.

Appendix B. Release notes

This section must be removed before publication as an RFC.

B.1. Modifications between -01 and -00

- Changed reference from JavaScript to ECMAScript.
- Changed signature from equals() to equalsHash().
- Fixed the examples following implementation.
- Replaced automatic decoding of value by ECMAScript code.
- Added the type of each property.
- Specified that the code cannot be used to modify the value stored.

B.2. TODO List

- Need to present the complete list of certificates for the DisCo [I-D.knauf-p2psip-disco] Usage USER-CHAIN-MATCH.

Author's Address

Marc Petit-Huguenin
Stonyfish, Inc.

Email: petithug@acm.org

P2PSIP
Internet Draft
Intended status: Informational
Expires: April 2011

Yunfei.Zhang
China Mobile
Gang.Li
China Mobile
Jin.Peng
China Mobile
Baohong.He
China CATR
Shihui.Duan
China CATR
Wei.Zhu
China CATR

March 11, 2011

bcp for a large scale carrier-level VoIP system using p2psip
draft-zhang-p2psip-bcp-04.txt

Status of this Memo

Copyright Notice

Abstract

This document introduces a practical work for Peer-to-Peer (P2P) SIP system based on Distributed Service Network (DSN), which was proposed by China Mobile in ITU-T. In this document, it introduces some key problems of carrier grade P2PSIP VoIP system. Then it gives a brief introduction on DSN VoIP system and especially discusses some key technologies aimed at those mentioned problems. At last, we present some measurement works for validating DSN VoIP system call transaction performance, reliability and high Availability. These measurements include the measurement in lab environment and in real internet.

Table of Contents

1. Introduction

DSN [1],the abbreviation of Distributed Services Network, is a new question being standardized in ITU-T proposed by China Mobile. [3] outlines the DSN Architecture. As described in [3], DSN can support many applications, such as Multimedia telephony services, streaming services, content distribution services and so on.

DSN VoIP system is designed based on the theory of DSN and P2PSIP. According to [1], DSN VoIP system includes two layers over IP network: P2PSIP layer and the application layer. The application layer uses SIP protocol for call and P2PSIP layer uses RELOAD[4] protocol for call routing in overlay network.

It also analyzes some key problems in P2P VoIP system and defines the corresponding requirements in this document. It also includes the system architecture, components and key technologies. At last it presents some measurement work for validating the DSN VoIP system.

2. Requirements and key problems of carrier-grade P2P VoIP system

2.1. Requirements

Peer-to-Peer(P2P) systems have been widely deployed in current internet, due to its advantages such as high scalability and cost-effectiveness. Many P2P technologies have been to implement the traditional telecom voice service (for example, IMS or softswitch) such as Skype. But it hasn’t been discussed thoroughly that how to make P2P system meet the telecom infrastructure performance requirements, which is usually called as the "Carrier-Grade" requirements. According to the current performance of telecom voice service deployments, the requirements of the carrier-grade services can be summarized but not limited to those listed below:

1. Qos guarantee: Current IMS requires that any client-side operations should be able to get the final responses in 300 milliseconds. Therefore, P2P VoIP system should be able to guarantee the signaling response time which is comparable to this time and the media transmission time which is less than 400ms. There are also some specifications for voice quality, which is not mentioned in this document.

2. High Availability: The high availability includes network availability, service availability, and subscriber data availability and so on. The most common practice is to use some special mechanisms to acquire robustness and availability such as hot backup redundancy. High availability is a severe challenge for P2P VoIP systems which intends to achieve the same level of the telecom systems.

3. Scalability: The P2P VoIP system should be adapted to the expanding of the system scale, such as the increasing of the number of the clients or core nodes.

4. Load balance: The resource is distributed among P2P nodes and each node should be able to collaborate with one another to avoid the emergence of the centralization of resource and traffic.

5. Cost-effectiveness: It’s the aim that the P2P VoIP system can achieve the same performance as the telecom commercial system by fully utilizing only commodity PCs, not using the costly hardware.

6. Maintainability: it’s quite appealing that it needs to do a little work to configure the nodes and network in order to maintain the natural operation of the P2P VoIP system. P2P VoIP system can realize the self-organization even after any network failure, so as to reduce the demand for emergent response and well experienced operators.

7. Others: TBD.

2.2. Some Key problems

To obtain the same performance as the telecom system, there are many key problems needed to be solved. Some of them are listed as below:

1. Routing performance: The traditional P2P routing performance is relative to the network dimensions. The routing performance will be sharply descended as the increase of the network dimensions, for example, the traditional Chord routing performance is O(logN).

2. Redundant traffic: The P2P network does not match the underlying network, so the P2P nodes forward the traffic only according to the logic P2P routing and not to the real optimal IP layer network routing, and this will cause mass P2P redundant traffic iteratively across many Automatic Systems(AS).

3. The reliability and availability of subscriber data: Current P2P network can't provide an efficient solution for the reliability and availability of subscriber data in case of the random joining or exiting of the P2P nodes happened in the network.

4. Duplicate data consistency: There are many replicas for subscriber data to acquire enough availability. It's very common that the subscribers' data are often modified, so the system must allow for the updating and amendment information to be propagated to all replicas asynchronously.

5. Hot spot: The subscriber data are distributed unevenly in the network, which will cause that some nodes have much more subscribers' data than the others. Therefore, it might probably lead to more traffic due to those high-loaded nodes which might makes these nodes overload and even fault in some cases.

6. Single node failure: If a node fails, its clients can't use the services.

7. Others: TBD.

It discusses some key technologies in section 3.3 with a view to solve the above questions.

3. DSN VoIP System Overview

DSN VoIP system is a VoIP system based on P2P overlay. DSN VoIP system uses SIP protocol for call transaction and RELOAD for call routing.

3.1. System Architecture

According to the architecture of RELOAD, DSN VoIP system has the similar architecture and contains some essential components defined in RELOAD, such as Usage Layer, Routing Layer, Storage and so on. The architecture of DSN VoIP system is shown as figure 1.

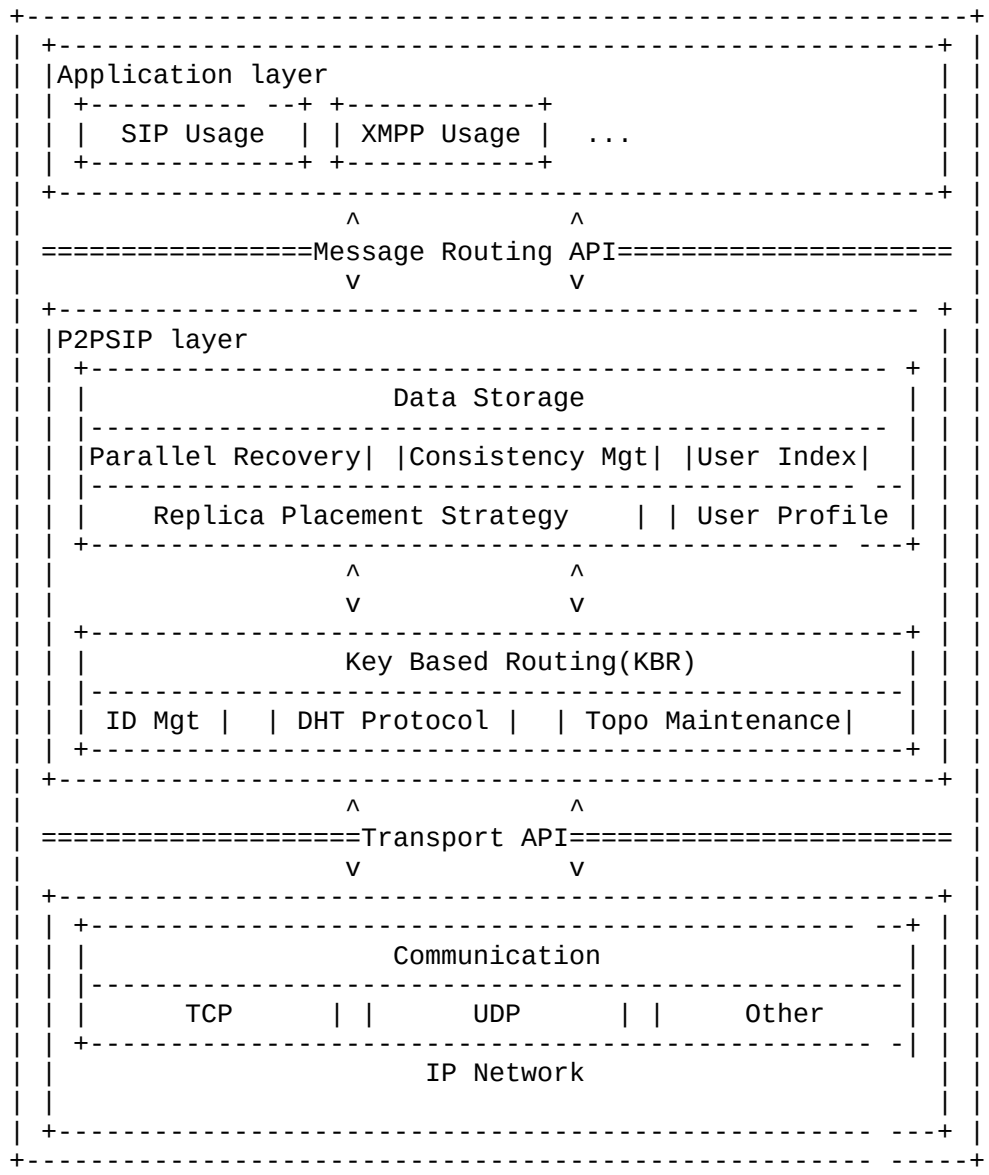


Figure 1 The architecture of DSN VoIP system

Based on the RELOAD architecture, DSN VoIP system implements the necessary functions required by RELOAD and some additional functions to make the system more practicability.

There are two basic modules in P2PSIP layer: KBR and Data Storage module.

The key based routing, KBR, refers to find the best suitable host for an input key, also includes ID management, DHT routing protocol, topology maintenance, peers failure detecting and etc. DSN VoIP system implements a unique KBR routing scheme with real-time response and a traffic localization mechanism, including a novel ID assignment method.

The data storage module takes charge of management of subscriber data, including storage, restoring, consistency verification and etc. Here DSN VoIP system implements a unique and practical replica replacement strategy and an effective consistency strategy.

3.2. System Implementation and Components

DSN VoIP system includes some distributed servers and subscriber terminals. These servers form the DSN VoIP network by using P2P protocol and share load for each other. According to the different functions, the server can be Peer Node(PN), Application Server(AS), Enrollment Server(ES), Edge Agent(EA), Super Peer-Maintenance (SPM),Relay Node(RN) and RN Cluster Server(RCS). According to the support for P2P protocol, the subscriber terminal can be the traditional SIP Client(SiC) and Peer Client(PeC). The DSN VoIP system framework is shown as figure 2.

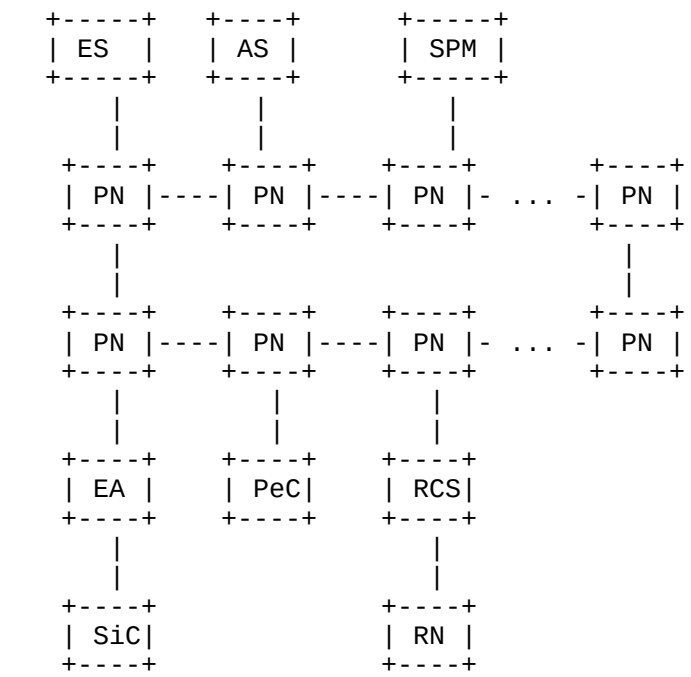


Figure 2 The framework and components of DSN VoIP system

Peer Node(PN): PN is the core node in DSN VoIP network and is deployed by telecom operators. PN is responsible for DHT topology maintenance, overlay routing, storage and access for subscriber/service data, subscriber registration and Authentication, BootStrap, SIP session control and VoIP service.

Application Server(AS): AS can provide other services which can't be provided by Peer Node(PN).

Enrollment Server(ES): ES implements BootStrap service when the PeC or PN joins the DSN VoIP network for the first time. When new PN joins, ES is responsible for authentication for the new node. ES is deployed by telecom operators and fixed in the network.

Edge Agent(EA):EA provides the relay service for SiC and adapt to all kinds of services and connections to PN. EA supports SIP proxy discovery for DSN and connection with PN.

SPM(Super Peer Maintenance): SPM is an administrator for one zone. SPM undertakes the registration of the PN nodes in the same zone, assigns the nodes IDs to PN nodes and maintain DHT routing information.

Relay Node(RN): RN is responsible for NAT traversal and media traffic forwarding. RN can discover the DSN and connection with PN. RN can be deployed by telecom operators or upgraded from PeC.

RN Cluster Server(RCS): RCS is responsible for the management of RN nodes according to topology information or RTT(Round-Trip Time). RCS manages one AS and assigns a suitable RN for PeC according to the information about PeC's request or location etc.

SIP Client(SiC): SiC support only SIP protocol and can't discover the DSN network if it is directly connected to the DSN. It must connect directly with Edge Agent (EA).

Peer Client(PeC): PeC support both SIP protocol and P2P protocol. PeC can be upgraded to be Relay Node (RN) or RN Cluster, but can't be Peer Node(PN).

PN nodes form the DSN VoIP network and interconnect via P2P protocol. The subscriber terminal can get valid service once it connects to any PN. If some but not all PN nodes fail or go offline, this will not affect the normal operation of DSN network. We can add more new PN nodes to augment the capacity of DSN network.

3.3. System Key technologies

In this section, it presents with some key technologies which is aiming to solve the problem in sec 2.2.

1. SPM assisted one hop route: DSN VoIP system is typically deployed as a two-layer DHT, including a global DHT (corresponding to countrywide) and several regional DHTs (corresponding to provinces). For each region, there are several special peers dedicated to routing information updating and some other management tasks and these special peers are SPM. Multiple SPMs can be deployed in a large region. Each one is responsible for a partition of PN belonging to that region, and these SPMs backup each other. SPMs from different regions are full-mesh connected, they run a gossip-based protocol to maintain a strongly consistent view of their existence. All PN nodes must register to the local SPM of the same region, when they join the network. During runtime, once the PN's state change is detected by its neighboring PN, the neighbor will notify the local SPM, and then the SPM will disseminate the event to other SPMs, finally each SPM will broadcast this notification to all PN nodes under the charge of it. In this way, each PN can obtain other PNs' information through SPM and can maintain routing table for all PNs. So SPM assisted one hop route can be implemented.

2. Traffic Localization: Recent studies showed that a large volume of inter-domain redundant traffic aroused by P2P mismatch problem already became a serious problem. As mentioned above, each PN node joins the countrywide global DHT and one regional DHT at the same time. We use a novel peer ID assignment scheme optimized for global load balancing, which can write the regional information into peer ID. With this unique ID scheme, the global DHT and regional DHT can be maintained by only one KBR (Key Based Routing) incarnation, that's different with many other layered DHT proposed. Every object will be put in the regional DHT and global DHT for disaster recovery, according to the normal DHT semantics internally.

3. Replica Placement strategy: The DSN VoIP system raises a unique N:B replica placement which provides an effective way to backup and recovery subscribers' data from local and remote region. Under this solution a parallel recovery factor N is defined as how many candidates should be used as the recovery. And B is defined as the number of Backups which means how many duplicated replicas the overlay hold. Subscriber data in each primary peer node is divided into S=N/B slices. After that, each B backup peers in a group for this primary peer will backup the same slice of subscriber data. One peer node could execute data backup procedure to send data to different backup peers simultaneously. Compare to 1:1 backup mechanism, not only does our N:B replica placement increase the efficiency of parallel recovery, but also avoids the reservation of half capacities (CPU cycles, bandwidth) for potential migration of the service backed up.

4. Consistency Strategy: In order to ensure better data consistency, DSN VoIP system will trigger "DataCheck" and "DataGoHome" mechanisms periodically. DSN VoIP system through "DataCheck" triggers the comparison between the data version information of the corresponding PN nodes, and updates the inconsistent data to ensure that the eventual consistency of subscriber profiles. "DataGoHome" takes charge of identifying those temporary irregular data, recalculate their key-value pair and push them to correct PN nodes. With the help of timing service always available in telecom offices, DSN VoIP system uses timestamp in order to capture causality between different versions of the same data. One can determine whether two versions of a data have a causal ordering by examine their timestamps.

5. Strip Segmentation method Based ID Assignment: We use this method to generate the subscriber ID. This method can evenly distribute the subscriber data on each PN node so as to achieve traffic load balance and avoid hot spot. The detailed method can be referred to [5].

6. Single node failure handle: If a PN node fails, we design some mechanisms to ensure the natural operation of VoIP service. For the session in progress, the session between the caller and the callee is established, the standby PN checks the primary PN in failure, and the standby PN notifies the associated nodes and switch the session to itself. For the new call originated from the caller, if the standby PN has switch to be the primary PN and notified the subscribers, the call originated from the subscriber will be routed to the standby PN. For the new call originated from the callee, if the callee knows the failure in the primary PN by requesting routing or querying routing table, the callee will route the query to the standby PN and establishes the session.

7. Others: TBD.

4. Validation of DSN VoIP system in lab environment

4.1. Overview

We did some measurements to validate the DSN VoIP system and its key technologies and all these were described in this document of version 01 which was submitted in IETF 77 meeting. In this document, we don’t repeat the results and conclusions in the first document.

Here we do some more thorough measurements to validate the performance and reliability of DSN VoIP system which are carried out in a bigger network.

We set up a DSN VoIP demo system in CMCC lab. The demo system includes 58 PN nodes, two SPM and one ES. All the PN nodes are deployed on VMWare machine and have the same configuration: two virtual CPU, 2G virtual memory, 75G hard disk space and SUSE Linux Enterprise Desktop 10 SP2 (i586). We deploy the four VM machines on one HP DL320 server and we use seventeen HP DL320 servers. All these HP servers connect through one Cisco 7500 layer3 switch.

Our emphases are system bulk call performance measurement and reliability measurement under churn.

4.2. System bulk call performance measurement

For measuring the bulk call performance of the demo system, we configure the following parameters:

1.call parameters:

each call last time=3 second

each measurement last time = 15 minutes

2. subscriber data parameters:

caller subscriber number = callee subscriber number = 16000

The format of SIP subscriber terminal ID is like 8xxxxxx@bj.dsn.com. The callers and callees must be registered before bulk call performance measurement. All these registered subscribers and calls handled by each PN are randomly averagely distributed in system. Our algorithm can ensure the approximate well-proportioned distribution for registered subscribers and calls handled by each PN.

We measured the bulk call transaction performance when the demo system respectively included 24, 32, 40, 48, 56 PN nodes. We collected the following parameters:

1.Registration statistics: average register time

2.Call Establishment time: average call setup time, average call tear down time, Post dial delay

3.Call finish: call success ratio, call attempts per second(CAPS)

If the call lose ratio is below 0.5% and don’t persist to increase, we think the call performance is under the limit of the system.

We do two measurements: 1)there are only callers and callees in the demo system; 2)except the callers and callees, there are some other users which originate registration for simulating the transfer from one place to another and the number of these users is one fifth of the total users.

4.2.1. Pure bulk call performance measurement results

Note: the maximal call capability of the demo system increases along with the number of PN nodes and it goes beyond the maximal capability

of the test device. So we can get accurate measurement results when the number of PN nodes is no more than 48.

The following are the results of the first measurement.

1.average register time

Node number	24	32	40	48
average register time(ms):	21	21	22	22

2. average call setup time

Node number	24	32	40	48
average setup time(ms):	338	423	311	223

3. average call tear down time

Node number	24	32	40	48
average tear down time(ms):	139	152	140	97

4. Post dial delay

Node number	24	32	40	48
Post dial delay (ms):	338	423	310	223

5. call success ratio

Node number	24	32	40	48
call success ratio(%):	99.97	99.95	99.89	99.91

6. call attempts per second(CAPS)

Node number	24	32	40	48
System CAPS:	1252	1708	2221	2562
PN’s average CAPS:	52.2	53.4	55.5	53.4

4.2.2. Mix Bulk call performance measurement results

The following are the results of the second measurement.

1.average register time

Node number	24	32	40	48
average register time(ms):	21	24	22	21

2. average call setup time

Node number	24	32	40	48
average setup time(ms):	346	376	319	244

3. average call tear down time

Node number	24	32	40	48
average tear down time(ms):	138	176	137	115

4. Post dial delay

Node number	24	32	40	48
Post dial delay (ms):	346	439	318	244

5. call success ratio

Node number	24	32	40	48
call success ratio(%):	99.98	99.06	99.96	99.77

6. call attempts per second(CAPS)

Node number	24	32	40	48
System CAPS:	1196	1591	1986	2267
PN’s average CAPS:	49.8	49.7	49.7	47.2

4.2.3. conclusions

Through above measurement, the conclusions are:

1. The DSN VoIP system has the capability for call transaction.

2. The system capability can approximately linearly increase as the number of PN nodes increase.

We have measure the bulk call performance for the demo system. However due to measurement time limited, these results are just elementary and further measurement will be done for more large scale network.

4.3. Churn measurement

4.3.1. Churn model and setting

The P2P overlay will churn when there are some nodes arriving or leaving. Generally we can give some assumption:

1. one node has the the probability p% to arise churn in an hour for leaving or poweroff etc and the churning node will equably distribute in the network.

2. The churn will occur with the periods of T1 minutes in the P2P network and p% of all nodes will leave the P2P overlay.

3. After T2 minutes, the total leaving nodes will return the P2P overlay with the probability q%. T2 must be less than T1.

Because we use backup mechanisms for user’s information, if one node leaves the overlay, its backup nodes will take on the incoming calls. When the node leaves the overlay, it will lead to call errors.

If one node return the overlay, it will get its previously possessed information from the backup nodes and this action will lead to generate new traffic(we call these traffic as recover traffic). These traffics will influence the network bandwidth and increase as the total number of the all users.

Based on the above discuss, for simplifying and accelerating the measurements, we set some supreme values for the measurement as follows:

1. p = 5, 10, 20, q = 80

2. T1 = 10 minutes, T2 = 4 minutes

3. Number of users = 100 thousands, 300 thousands, 500 thousands, 1000 thousands

We do churn measurements under different p and number of users in the demo system with 58 PN nodes. Because we don’t care the bulk call performance under churn, we use test device to originate 1800caps

call in each measurement and the settings for the calls are the same as section 4.2.

We collected the following parameters:

1. Call success ratio: call success establishment compared to all call.

2. Call error: call fail caused by either caller or callee. we randomly get the call errors from two arbitrary nodes which leave and return the overlay.

3. sampled recover traffics: we randomly get the recover traffics from two arbitrary nodes which leave and return the overlay.

4.3.2. Measurement results

The following are the results.

1. p = 5% (about 3 nodes involved in churn at the same time)

user number(thousands):	100	300	500	1000
call success ratio(%):	99.00	99.31	99.52	99.93
call error:	34283	22702	16240	3316
	32676	22290	15681	2545
Recover traffic(kbps):	999	490	198	12372
	880	117	441	11613

2. p = 10% (about 6 nodes involved in churn at the same time)

user number(thousands):	100	300	500	1000
call success ratio(%):	99.74	99.65	99.67	99.71
call error:	10299	13041	12532	15256
	9020	12038	11445	12055
Recover traffic(kbps):	1430	1360	2500	13840
	1170	1020	2670	9720

3. p = 20% (about 12 nodes involved in churn at the same time)

user number(thousands):	100	300	500	1000
call success ratio(%):	99.11	99.16	98.98	98.93
call error:	28276	26684	32474	51705
	28275	26684	32474	42942
Recover traffic(kbps):	250	450	660	5397
	220	410	600	10888

4.3.3. conclusions

Through above measurement, the conclusions are:

1. The recover traffic will increase as the total users increase.

2. If the churn involves a small quantity of PN nodes, the churn has little effect to the system call transaction performance and almost is independent of the total user’s number.

5. Validation of DSN VoIP system in real environment

5.1. Overview

We establish a small DSN VoIP demo network over the internet to validate the DSN VoIP system’s usability in real internet.

The demo network is deployed in three cities: Beijing, Shenzhen and Wuhan. This network includes 7 PN nodes, three in Beijing, two in Shenzhen and two in Wuhan.

These seven PNs have different configuration: the PNs in Wuhan use common PCs, the PNs in Shenzhen use virtual machines in a common PC and the PNs in Beijing use virtual machines in two HP DL320 servers.

The local network in each place belongs to different operator network, the PNs in Wuhan use CERNET, the PNs in Shenzhen use China Telecom’s IP network and the PNs in Beijing use China Mobile’s IP network. The access bandwidth for each point is quite different. The access bandwidth for these three locations is shared bandwidth, there are some other traffics on the same access point, we don’t know how much bandwidth is used by the other traffic and how the other traffic may affect the available bandwidth for the DSN VoIP traffic.

(Note: we do many measurements for different number of PNs in three cities and here only choose a typical measurement due to limited space.)

5.2. Single Capabilities measurement

For measuring the performance of the DSN VoIP demo network, we configure the following parameters:

1. call parameters:

call length =20 second

Total call number = 200,000

2. subscriber data parameters:

caller number = callee number = 100,000

The callers and callees must be registered before measurement. All these registered subscribers and calls handled by each PN are randomly averagely distributed in system. So there are one sixth calls between Beijing and Beijing, one sixth between Beijing and Wuhan, one six between Beijing and Shenzhen, one sixth between Shenzhen and Shenzhen, one sixth between Shenzhen and Wuhan, one sixth between Wuhan and Wuhan. We guess that these circs may occur, namely the least time for the calls is the calls between Beijing and Beijing, then the calls between Shenzhen and Shenzhen and the calls between Wuhan and Wuhan, the calls between different cities will take much longer time than the two cases.

Since there are no standards for Internet VoIP, we don’t know how to evaluate the results of this measurement. We refer to [2] and [2] gives some performance referenced parameters for mobile calls.

Every caller will finish two calls during the measurement if no error happens. We collected the following results:

1. Call Response time(msec)-CR time

Minimum: 63 Average: 195 Maximum: 6122

Call Response time is the time when the first 100 trying message is received after Invite is sent.

In Sec 3.2.3.1 of [2], the mean value for CR time is required to be less than 800ms and it doesn’t exceed 1000ms with 0.95 probability.

In our test about 85% CR times in successful calls doesn't exceed 1000ms.

2. Call setup(msec)-CS time

Minimum: 73 Average: 1548 Maximum: 24525

In Sec 3.2.3.8 of [2], the mean value for CS time is required to be less than 2200ms and it doesn't exceed 2400ms with 0.95 probability. In our test, about 76% CS times in successful calls doesn't exceed 2400ms.

3. Tear Down(msec)-TD time

Minimum: 17 Average: 919 Maximum: 19735

In Sec 3.2.3.5 of [2], the mean value for TD time is required to be less than 400ms and it doesn't exceed 700ms with 0.95 probability. In our test, about 82% TD times in successful calls doesn't exceed 700ms.

4. Post Dial delay(msec)-PD time

Minimum: 73 Average: 1544 Maximum: 24528

In Sec 3.2.3.7 of [2], the mean value for PD time is required to be less than 175ms and it doesn't exceed 350ms with 0.95 probability. In this measurement, less than 45% PD times in successful calls doesn't exceed 350ms.

From the above results, we can see these results show much difference with the results in lab environment, for example, the average call setup time is much higher in real network. We give a brief analysis as following:

1. The measurement spans three operator's network so that the traffic go cross too many different network and it can lead to long transmission delay. If the long transmission delay exceeds the threshold of some timers, it will lead the call fail. For example, we do traceroute from Shenzhen to Beijing and find there are 16 hops between these two points. In lab environment, there is only one hop between every pair of PNs and the ping delay is under 5ms.

2. For security, the PNs in these three points are all behind firewall though these PNs have public IP addresses. The firewall has its secure policy for every stream and inspects every ingoing/outgoing packet. So the firewall's action has some impact

on the communication between PNs. We find that the communication between two neighbor PNs is often lost for some reasons and this badly affect the call success ratio. In lab environment, there is no firewall and direction communication between PNs.

3. Since the PNs in different points have different configuration, the PN has different call transaction capability. In DSN VoIP system, the calls distributed for each PN are approximately uniform, as the call traffic increases gradually, the PN with the lowest configuration will lead to call fails firstly. In lab environment, all PNs have the same configuration and the call fail has an even distribution.

5.3. conclusions

Through above measurement, the conclusions are:

1. The DSN VoIP system can work in global internet and doesn’t perform well compared with the tests in lab environment.

2. Some issues obviously affect the performance of the system, e.g. limited IP interconnection bandwidth, firewall configuration, heterogeneous platform configuration.

Due to many unpredictable reasons in the WAN, these results show much ununiformity which can’t be seen in the lab environment.

6. Security Considerations

This draft does not introduce any new security issues.

7. IANA Considerations

This memo includes no request to IANA.

8. Conclusions

Through the measurement for the demo system, we validate that DSN VoIP system is a potential operationable system, which has good call transaction capability, high reliability and availability to settle some key problem mentioned in Sec2.2 at a certain extent.

The call transaction capability of DSN VoIP system can linearly increase as the server continuously joins the overlay. And this makes it possible that we can provide Carrier-grade call transaction capability by using normal computer server.

The DSN VoIP system has an excellent anti-churn capability so that this overcomes the single point failure and provides the high network and service reliability for subscribers.

The measurement in real WAN shows that the DSN VoIP system can be deployed in the internet and affected by many unpredictable issues.

9. References

9.1. Normative References

[1] ITU-T DSN, Proposed scope of DSN, http://www.itu.int/md/meetingdoc.asp?lang=en&parent=T09-SG13-090112-C&PageLB=50.

[2] 3GPP TS 43.005 V9.0.0, Technical Specification Group Core Network and Terminals, Technical performance objectives

9.2. Informative References

[3] [draft-zhang-ppsp-dsn-introduction-00]www.ietf.org/internet-draft/draft-zhang-ppsp-dsn-introduction-00.txt

[4] [draft-ietf-p2psip-base-07]www.ietf.org/id/draft-ietf-p2psip-base-07.txt

[5] Guangyu Shi, Jian Chen, Hao Gong, Lingyuan Fan, Haiqiang Xue, Qingming Lu, and Liang Liang, " SandStone: A DHT based Carrier Grade Distributed Storage System ", Proc. ICPP 2009

10. Acknowledgments

Thanks to Cheng Li, Haitao Yang, Xiaolin Jiang, Jian Zhang and Jiguang Cao for their hard efforts, comments and suggestions. Special thanks to Guangwu He for his elaborate technical supports for DSN VoIP system.

Authors' Addresses

Yunfei Zhang
China Mobile
Beijing 100045, China
Phone: 86-13601032119
Email: zhangyunfei@chinamobile.com

Gang Li
China Mobile
Beijing 100045, China
Phone: 86-13501279120
Email: ligangyf@chinamobile.com

Jin Peng
China Mobile
Beijing 100045, China
Phone: 86-13911281193
Email: pengjin@chinamobile.com

Baohong He
China CATR
Beijing 100045, China
Phone: 86-10-62300050
Email: hebaohong@catr.cn

Shihui Duan
China CATR
Beijing 100045, China
Phone: 86-10-62300068
Email: duanshihui@catr.cn

Wei Zhu
China CATR
Beijing 100045, China
Phone: 86-10-62300089
Email: zhuwei@catr.cn