P2PSIP
Internet-Draft
Intended status: Standards Track
Expires: January 2, 2008

C. Jennings
J. Rosenberg
Cisco
E. Rescorla
Network Resonance

July 1, 2007

# Address Settlement by Peer to Peer
# draft-jennings-p2psip-asp-00

## Status of this Memo



## Copyright Notice



## Abstract

This document defines Address Settlement by Peer-to-Peer (ASP), a peer-to-peer (P2P) binary signaling protocol for usage on the Internet. A P2P signaling protocol provides its clients with an abstract hash table service between a set of cooperating peers that form the P2P network. ASP is designed to support a P2P Session Initiation Protocol (SIP) network, but it can be utilized by other

applications with similar requirements. ASP introduces the notion of usages, which are a collection of data types that are required for a particular application. For SIP, these types include location, STUN and TURN servers. ASP defines a security model based on a certificate enrollment service that provides peers with unique identities. ASP also provides protocol extensibility and defines a migration methodology, allowing for major upgrades of the P2P network without service disruption.

Table of Contents

## 1. Introduction

> With thy sharp teeth this knot intrinsicate
> Of life at once untie: poor venomous fool
> Be angry, and dispatch.
>
> -Cleopatra, Act V, scene II,
> Antony and Cleopatra by William Shakespeare

This document defines Address Settlement by Peer-to-Peer (ASP), a peer-to-peer (P2P) signaling protocol for usage on the Internet. A P2P signaling protocol provides its clients with an abstract hash table service. Clients can both read and write entries into the hash table. The hash table is actually distributed: pieces of the table are stored by the various clients that access it. Such an abstract hash table service, in which the contents of the hash table are stored across many hosts, is called a Distributed Hash Table (DHT).

ASP is a lightweight, binary protocol. It provides several functions that are critical for a successful P2P protocol for the Internet. These are:

Security Framework: Security is one of the most challenging problems in a P2P protocol. A P2P network will often be established among a set of peers none of which trust each other. Yet, despite this lack of trust, the network must operate reliably to allow storage and retrieval of data. ASP defines an abstract enrollment server, which all entities trust to generate unique identifiers for each user. Using that small amount of trust as an anchor, ASP defines a security framework that allows for authorization of P2P protocol functions and DHT write operations. This framework mitigates many important threats, such as corruption of data in the DHT by malicious users. ASP itself runs only over TLS or DTLS.

Usage Model: It is anticipated that many applications, including multimedia communications with the Session Initiation Protocol (SIP) [RFC3261], will utilize the services of ASP. Consequently, ASP has the notion of a usage, one of which is defined to support each application (this document also defines the SIP usage for multimedia communications). Each usage identifies a set of data types that need to be stored and retrieved from the DHT (the SIP usage defines one for registrations, one for certificates, one for Traversal Using Relay NAT (TURN) [I-D.ietf-behave-turn] servers and one for Session Traversal Utilities for NAT (STUN) [I-D.ietf-behave-rfc3489bis] servers). Each type defines a data structure, authorization policies, size quota, and information required for storage and retrieval in the DHT. The usage concept allows ASP to be used with new applications through a simple

documentation process that supplies the details for each application.

Pluggable DHT Algorithms: Many algorithms have been developed for DHTs, including Chord, CAN, Kademlia, and so on. The goal of ASP is to make it very easy to define how ASP works with each DHT algorithm, and furthermore, to minimize the amount of specification work, protocol change, and coding that are required to support each DHT. To accomplish this, ASP defines an abstracted interface between ASP and the DHT algorithm. ASP is designed so as to minimize the amount of logic within the DHT algorithm itself, so that core ASP services are as generalized as possible. This specification also defines how ASP is used with Chord.

High Performance Routing: The very nature of DHT algorithms introduces a requirement that peers participating in the P2P network route requests on behalf of other peers in the network. This introduces a load on those other peers, in the form of bandwidth and processing power. ASP has been defined to reduce the amount of bandwidth and processing required of peers. It does so by using a very lightweight binary protocol, and furthermore, by defining a packet structure that facilitates low-complexity forwarding, including hardware-based forwarding. It borrows concepts in Multi-Protocol Label Switching (MPLS) around label stacks to minimize the computational costs of forwarding.

NAT Traversal NAT and firewall traversal are built into the design of the protocol. ASP makes use of Interactive Connectivity Establishment (ICE) [I-D.ietf-mmusic-ice] to facilitate the creation of the P2P network and the establishment of links for use by the application protocol (SIP and RTP, for example). ASP also defines how the peers in the P2P network can act as STUN and TURN servers and how those resources can be discovered through the DHT. ASP runs over both TLS and DTLS, so that its connections can support both bulk transfer and datagram connectivity. With these features, ASP can run in modes in which all the peers are behind NATs, yet are able to fully participate without imposing any constraints on the actual DHT algorithm or routing topology.

Multiple P2P Networks: ASP allows for multiple and unrelated P2P networks to operate at the same time. A single peer can participate in more than one, while at the same time running ASP on a single port.

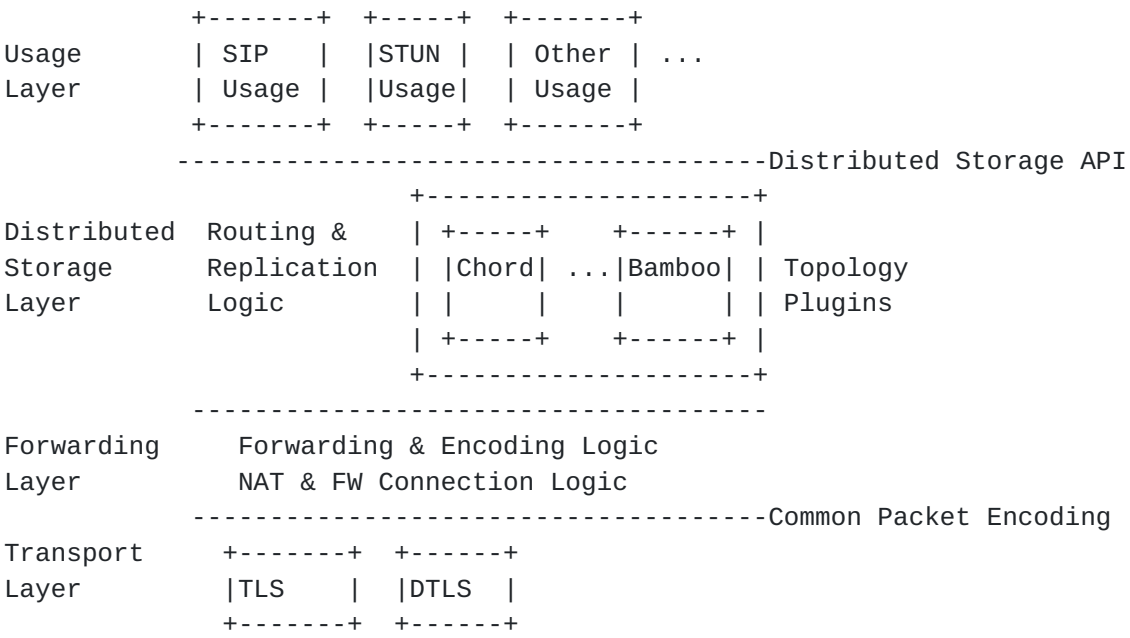
Extensible: Extending P2P protocols is a challenging task, due to the highly distributed nature of their behavior. ASP introduces a protocol extensibility model similar to the one used for the Border Gateway Protocol (BGP). BGP, like ASP, runs among a large number of peers to implement a highly distributed protocol. It does this by including bit flags for each command that indicate properties of that command. ASP also introduces a migration model, whereby parallel P2P networks are utilized during a cutover

interval while a major protocol change is in progress.

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

## 2. Overview

Architecturally this specification splits into several layers, as shown in the following figure.




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

The next layer is a Distributed Storage Layer. It can store and retrieve information, perform maintenance of the DHT as peers join and leave the DHT, and route messages. This layer is tightly bound to the specific DHT algorithm being used, as this algorithm determines how both routing and redundant storage are done in the DHT. The goal of this layer is to provide a fairly generic

distributed and redundant storage service.

The next layer down is the Forwarding Layer. This layer is responsible for getting a packet to the next peer in the DHT. It uses the routing layers above it to determine what the next hop is; this layer deals with actually forwarding the packet to the next hop. Forwarding can include setting up connections to other peers through NATs and firewalls using ICE; it can take advantage of relays for NAT and firewall traversal. This layer passes packets in a common packet encoding, regardless of what DHT algorithm is being used in the Transport Layer below it. The goal of the Forwarding Layer is to forward packets to other peers.

Finally, in the bottom layer, packets are sent using a Transport Layer which uses TLS and DTLS.

## 2.1. Distributed Storage Layer

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

### 2.1.1. Distributed Storage API

TODO

### 2.1.2. DHT Topology

Each DHT will have a somewhat different structure, but many of the concepts are common. The DHT defines a large space of loci, which can be thought of as addresses. In many DHTs, the loci are simply 128- or 160-bit integers. Each DHT also has a distance metric such that we can say that locus A is closer to locus B than to locus C. When the loci are n-bit integers, they are often considered to be arranged in a ring so that (2^n)-1 and (0) are consecutive and distance is simply distance around the ring.

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

### 2.1.3. Routing

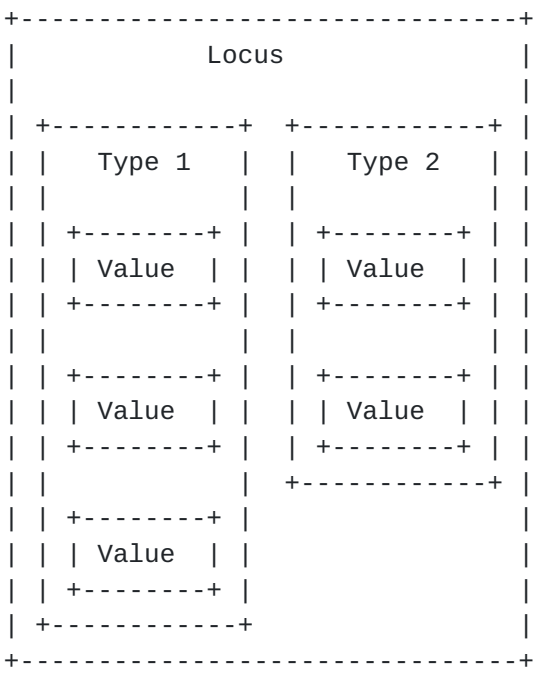
The way routing works in a DHT is specified by the specific DHT algorithm but the basic concepts are common to most systems. Each peer maintains connections to some other set of peers N. There need not be anything special about the peers in N, except that the peer has a direct connection to them: it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

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

In ASP, messages may either be COMMANDS or RESPONSES to COMMANDS. Messages are routed as described above. In principle, responses could be routed the same way, but this makes diagnosis of errors difficult. Instead, as commands travel through the network they accumulate a history of the peers they passed through and responses are routed in the opposite direction so that they follow the same path in reverse.

### 2.1.4. Storing and Retrieving Typed Data

The Data Storage Layer provides operations to STORE, FETCH, and REMOVE data from the DHT. Each location in the DHT is referenced by a single integer locus. However, each location may contain data elements of multiple types. Furthermore, there may be multiple values of each type, as shown below.




Each type-id is a code point assigned to a specific application usage by IANA. As part of the Usage definition, protocol designers may define constraints, such as limits on size, on the values which may be stored. For many types, the set may be restricted to a single item; some sets may be allowed to contain multiple identical items while others may only have unique items. Some typical types of sets that a usage definition would use include:

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

set: Many values can be stored and each store appends to the set, but there cannot be two entries with the same value.

bag: Similar to a set, but there can be more than one entry with the same value.

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 command.

### 2.1.5. Joining, Leaving, and Maintenance

When a new peer wishes to join the DHT, it must have a peer-id that it is allowed to use. It uses one of the peer-ids in the certificate it received from the enrollment server. The main steps in joining the DHT are:

o 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.

First, the peer ("JP," for Joining Peer) uses the bootstrap procedures to find some (any) peer in the DHT. It then typically contacts the peer which would have formerly been responsible for the peer's locus (since that is where in the DHT the peer will be joining), the Responsible Peer (RP). It copies the other peer's state, including the data values it is now responsible for and the identities of the peers with which the other peer has direct connections.

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

1. JP sends a JOIN command to RP announcing its intention to join.
2. RP sends an OK response.
3. RP does a sequence of STOREs to JP to give it the data it will need.
4. RP does a sequence of UPDATEs to JP to tell it about its own routing table. At this point, both JP and RP consider JP responsible for some section of the DHT.
5. JP makes its own connections to the appropriate peers in the DHT. Often this is done merely by copying RP's routing table.

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

### 2.1.6. Forming Direct Connections

As described in Section 2.1.3, a peer maintains a set of direct connections to other peers in the DHT. Consider the case of a peer JP just joining the DHT. It communicates with the responsible peer RP and gets the list of the peers in RP's routing table. Naively, it could simply connect to the IP address listed for each peer, but this works poorly if some of those peers are behind a NAT or firewall. Instead, we use the CONNECT command to establish a connection.

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

### 2.1.7. Data Replication

TODO - More is needed here but the short version is that the replication approach is defined by the specific DHT algorithm not the Usage. The reason is that when a peer comes or goes, specific knowledge of the DHT topology is required to understand where the replication set is stored for the data. Also need to explain how data is merged after a network partition event.

## 2.2. Forwarding Layer

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

- Deciding the message was destined for this peer and passing the message up to the layer above this.
- Looking at the label that represents the flow to which this message needs to be sent next and forwarding the message over that flow.
- Requesting the DHT Routing logic to tell the forwarding layer which flow the message needs to be forwarded on, and then sending the message on that flow.

### 2.2.1. Label Stacks

In a general messaging system, messages need a source and a destination. In an overlay network it is often useful to specify the source or destination as the path through the overlay. In addition, responses to commands need to retrace the command's path. To support this, each message has a source label stack and a destination label stack. Each label is 32 bits long, and the labels 0 to 254 are reserved for special use. 0 is an invalid label and 1 indicates that the next 4 labels are to be interpreted as a peer-id.

When a peer receives a message from the Transport Layer, it pushes a label on the source stack that indicates which TLS or DTLS flow the message arrived on. When a peer goes to transmit a message to the Transport Layer, it looks at the top label on the destination stack. If the top label is not one of the special use labels, it pops that label off the destination stack and sends the message over the TLS or DTLS flow that corresponds to that label. If the label is 1, then the next 4 labels are looked at and interpreted as a peer id. Note that these can be in the 0 to 254 range and still be interpreted as a peer-id. The routing logic in the Distributed Storage Layers is consulted to find out where to route this message. If this peer is responsible for the peer-id, then the 5 labels for the peer-id are popped off and the message is passed up to the Distributed Storage Layer for processing. Otherwise the labels are not popped off and the message is forwarded over the TLS or DTLS flow indicated in the

routing logic.

When a peer goes to send a response to a command, it can simply copy the source label stack from the command into the destination label stack of the response and then start forwarding the response.

Peers that are willing to maintain state may do label compression. They do this by taking some number of labels off the top of the source label stack and replacing them with a single label that uniquely represents all the labels removed. Later, if the peer sees the compressed label in a destination label set, it removes it and replaces it with all the labels it originally popped off the s source label stack. Doing this requires a peer to save state but it allows certain peers to provide services in which they reduce the size of messages going across bandwidth-constrained links. It can also help protect the privacy of the per-compression peer topology. (TODO need more on length of validity of compressed labels)

The label stack approach provides several features. First it allows a response to follow the same path as the request. This is particularly important for peers that are sending commands while they are joining and before other peers can route to them. It also makes it easier to diagnose and manage the system. Storing a label stack that includes a peer that does label compression provides the type of Local Network Protection described inRFC 4864 [RFC4864] without requiring a NAT.

## 2.3. Transport Layer

This layer sends and receives messages over TLS and DTLS. Each TLS or DTLS connection is referred to as a flow. For TLS it does the framing of messages into the stream. For DTLS it takes care of fragmentation issues. The reason for including TLS is the improved performance it can offer for bulk transport of data. The reason for including DTLS is that the percentage of the time that two devices behind NATs can form a direct connection without a relay is much higher for DTLS than for TLS. The way DTLS and TLS certificates are used does not require a global PKI, and therefore no option that uses only TCP or UDP without any security is included.

## 2.4. Enrollment

Before a new user can join the DHT for the first time, they must enroll in the P2P Network for the DHT they want to join. Enrollment will typically be done by contacting a centralized enrollment server. Other approaches are possible but are outside the scope of this specification. The user establishes his identity to the server's satisfaction and provides the server with its public key. The

centralized server then returns a certificate binding the user's user name to their public key. The properties of the certificate are discussed more in Section 2.5. The amount of authentication performed here can vary radically depending on the DHT network being joined. Some networks may do no verification at all and some may require extensive identity verification. The only invariant that the enrollment server needs to ensure is that no two users may have the same identity.

During the enrollment process, the central server also provides the peer/user with the root certificate for the DHT, information about the DHT algorithm that is being used, a P2P-Network-Id that uniquely identifies this ring, the list of bootstrap peers, and any other parameters it may need to connect to the DHT. The DHT also informs the peers what Usages it is required to support to be a peer on this P2P Network. Once the peer has enrolled, it may join the DHT.

## 2.5. Security

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

The certificate serves two purposes:

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

When a user enrolls, or enrolls a new device, the user is given a certificate. This certificate contains information that identifies the user and the device they are using. If a user has more than one device, typically they would get one certificate for each device. This allows each device to act as a separate peer.

The contents of the certificate include:

- A public key provided by the user.
- Zero, one, or more user names that the DHT is allowing this user to use. For example, "alice@example.org". Typically a certificate will have one name. In the SIP usage, this name corresponds to the AOR.

- Zero, one, or more peer-ids. Typically there will be one peer-id. Each device will use a different peer-id, even if two devices belong to the same user. Peer-IDs should be chosen randomly.
- A serial number that is unique to this certificate across all the certificates issued for this DHT.
- An expiration time for the certificate.

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

### 2.5.1. Storage Permissions

When a peer uses a STORE command to place data at a particular location X, it must sign with the private key that corresponds to a certificate that is suitable for storing at location X. Each data type in a usage defines the exact rules for determining what certificate is appropriate. However, the most natural rule is that a certificate with user name U allows the user to store data at locus H(U) where H is a cryptographic hash function characteristic of the DHT. The idea here is that someone wishing to look up identity U goes to locus H(U), which is where the user is permitted to store their data.

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.

### 2.5.2. Peer Permissions

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

Note that because the formation of a connection between two nodes generally requires traversing other nodes in the DHT, as specified in Section 2.1.6, those nodes can interfere with connection initiation. However, if they attempt to impersonate the target peer they will be unable to complete the TLS mutual authentication: therefore such attacks can be detected.

### 2.5.3. Expiry and Renewal

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

## 2.6. Migration

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

# 3. Usages Layer

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

- Register code points for any type that the Usage defines.
- Define the data structure for each of the types.
- Define access control rules for each type.
- Provide a size limit for each type.
- Define how the seed is formed that is hashed to form the locus where each type is stored.
- Describe how values will be merged after a network partition. Unless otherwise specified, the default merging rule is to act as if all the values that need to be merged were stored and that the order they were stored in corresponds to the timestamps on the signatures associated with their values.

TODO - Give advice on things that make bad usages - for example,

things that involve unlimited storage such as storing voice mail.

### 3.1. SIP Usage

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

This operation can take a number of forms, but in the simplest case, Bob's SIP UA has peer-ID "B". When Bob joins the DHT (i.e., turns on his phone), he stores the following mapping in the DHT:

o sip:bob@dht.example.com -> B

When Alice wants to call Bob, she starts with his URI and her UA uses the DHT to look up his peer-ID B. She then routes a message through the DHT to B requesting a direct connection. Once this connection is established she can send SIP messages over it, which allows her to set up the phone call.

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

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

#### 3.1.1. SIP Location

A peer acting as a SIP UA stores their registration information in the DHT by storing a label stack that routes to them at a locus in the DHT formed from the user's SIP AOR". When another peer wishes to find a peer that is registered for a SIP URI, the lookup of the user's name is done by taking the user's SIP Address or Record (AOR) and using it as the seed that is hashed to get a locus. A lookup for a data type of sip-location is done to this locus to find a set of values. Each value is a data structure contains a label stack that is used to reach a peer that represents a SIP UA registered for that AOR. The data structure also contains a string that would be a valid SIP header field value for the contact header in a 3xx response from a redirect server. This string can contain the caller-pref (TODO add reference) information for that SIP UA.

The seed for this usage is a user's SIP AOR, such as

"sip:alice@example.com", and the locus is formed by taking the top 128 bits of the SHA-1 hash of the seed. The set is a dictionary style set and is indexed by the peer-id of the certificate used to sign the STORE command. This allows the set to store many values but only one for each peer. The authorization policy is that STORE commands are only allowed if the user name in the signing certificate, when turned into a SIP URL and hashed, matches the locus. This policy ensures that only a user with the certificate with the user name "alice@example.com" can write to the locus that will be used to look up calls to "sip:alice@example.com".

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

### 3.1.2. SIP GRUUs

GRUUs that refer to peers in the P2P network are constructed by simply forming a GRUU, where the value of gr URI parameter contains a base64 encoded version of the label stack that will reach the peer. The base64 encoding is done with the alphabet specified in table 1 of RFC 4648 with the exception that ~ is used in place of =. An example GRUU is "sip:alice@example.com;gr=MDEyMzQ1Njc4OTAxMjM0NTY3ODk~". When a peer needs to route a message to a GRUU in the same P2P network, it simply decodes the label stack and connects to that peer.

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

### 3.1.3. SIP Connect

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

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

connection is being formed to.

### 3.2. Certificate Store Usage

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

This usage defines one new type, called "certificate." Each locus stores only a single value which is the X.509 certificate encoded using DER. The seed used to generate the locus is simply the serial number of the certificate. When a peer receives a command to STORE a particular certificate, it needs to be signed with the certificate with that serial number. This ensures that an attacker cannot overwrite the certificate of some other user.

Each user can store their current and previous certificate. This allows for transition from an old certificate to a new one. The certificate is stored as an X.509 certificate encoded with DER.

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

### 3.3. STUN Usage

This usage defines two new types, one for STUN servers and one for STUN-Relay servers.

Peers that provide the STUN server type need to support both UDP and TCP hole punching as defined in XXX, while peers that provide the STUN-Relay server type need to support the TURN extensions to STUN for media relay of both UDP and TCP traffic as defined in XXX.

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

Peers can find other servers by selecting a random locus and then doing a FIND command for the appropriate server type with that locus. The FIND command gets routed to a random peer based on the locus. 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 locus. As long as the ratio of servers relative to peers is not too low, this approach will result in finding a server relatively quickly.

Any peer that is not running in one of the RFC 1597 private address spaces MUST provide a STUN server. Open issues - what about requiring STUN-Relay servers? Should there be low and high bandwidth version of STUN-Relay one can find? Low would be usable for signaling type things and high would be usable for audio and more.

### 3.4. Other Usages

This will likely be left out of scope of the initial system but just to give people a flavor of how these issues might be dealt with....

#### 3.4.1. Storing Buddy Lists

Buddy lists with reciprocal subscribes - when see indication buddy might be online, such as SUBSCRIBE from buddy, retry SUBSCRIBE to buddy. Subscriber ends up doing composition.

Single users with different devices can synchronize buddy lists when both are online

#### 3.4.2. Storing Users' Vcards

#### 3.4.3. Finding Voicemail Message Recorder

Can register a voicemail URI that fetches a greeting from a web server, plays this, and records a message, and then email the result to specified location. Could define a server usage for this similar to STUN/TURN server usage - may not have enough of them to effectively find with random probing and FIND command.

Store a mailto contact in the SIP Location and have it mean you can record a G.711 wav file for this user and email it to them.

#### 3.4.4. ID/Locator Mappings

## 4. Conventions

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].

## 5. Terminology

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

DHT Algorithm: An algorithm that defines the rules for determining which peers in a DHT store a particular piece of data and for determining a topology of interconnections amongst peers in order to find a piece of data. Examples of DHT algorithms are Chord, Bamboo and Tapestry.

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

P2P Network: Another name for a DHT instance.

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

P2P Network ID: A 24 bit identifier formed by taking portions of the hash of the P2P network name. The P2P network ID is present in ASP protocol messages and identifies the P2P network to which those messages are targeted.

Hashspace: A range of integers from 0 to 2^N - 1 for some value of N (typically 128 or larger), defined by the DHT algorithm. Identifiers for peers and for resources stored in the DHT are taken from the hashspace.

Locus: A locus is a single point in the hashspace.

Seed: A seed is a string used as an input to a hash function, the result of which is a locus.

Peer: A host that is participating in the DHT. By virtue of its participation it can store data and is responsible for some portion of the hashspace.

Peer-ID: A locus that uniquely identifies a peer. Peer-IDs 0 and 2^N - 1 are reserved and are invalid peer-IDs. A value of zero is not used in the wire protocol but can be used to indicate an invalid peer in implementations and APIs. The peer-id is used on the wire protocol as a wildcard.

Resource: An object associated with an identifier. The identifier for the object is a string that can be mapped into a locus by using the string as a seed to the hash function. A SIP resource, for example, is identified by its AOR.

User: A human being.

Usage: A usage is an application that wishes to use the DHT for some purpose. Each application wishing to use the DHT defines a set of data types that it wishes to use. The SIP usage defines the location, certificate, STUN server and TURN server data types.

About In this specification, the word "About" followed by some time, X, is used to mean a time that is randomly distributed between 90% and 100% of X.

## 6. Common Packet Encodings and Semantics

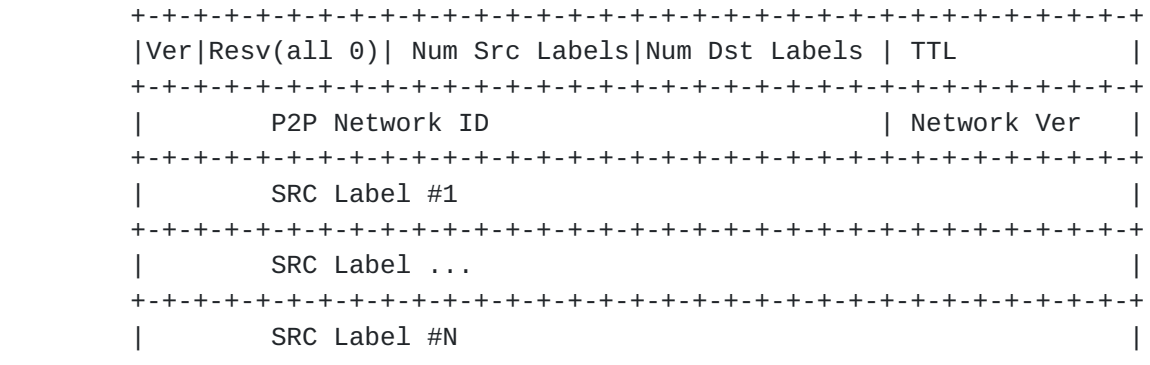
This section provides the normative description of what peers need to do when sending and receiving the actual protocol commands. The basic message consists of a Forwarding Block that determines the destination of the message, followed by one or more Command Blocks or Response Blocks. The support for multiples of the Command or Response Blocks is just to pipeline several Commands or Responses together. Each Command Block specifies an operation and will receive a response.

### 6.1. Forwarding Block

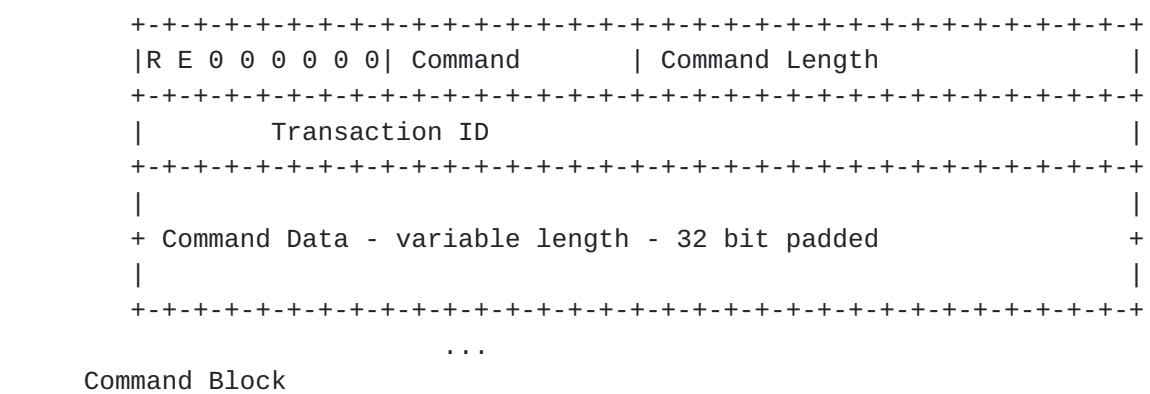
The common packet format consists of a forwarding block with a TTL, P2P-Network-Id and version for that network, a stack of source and destination labels, and finally a variable number of command blocks. The top two bits in the first byte indicate the version of the ASP protocol and are set to 0 for this version. When a label is pushed on the stack, it becomes the first label; label #1 is the top of the stack and #N is the bottom.

Open issue: Do we want a magic number at front of block to indicate the protocol.

Forwarding Block

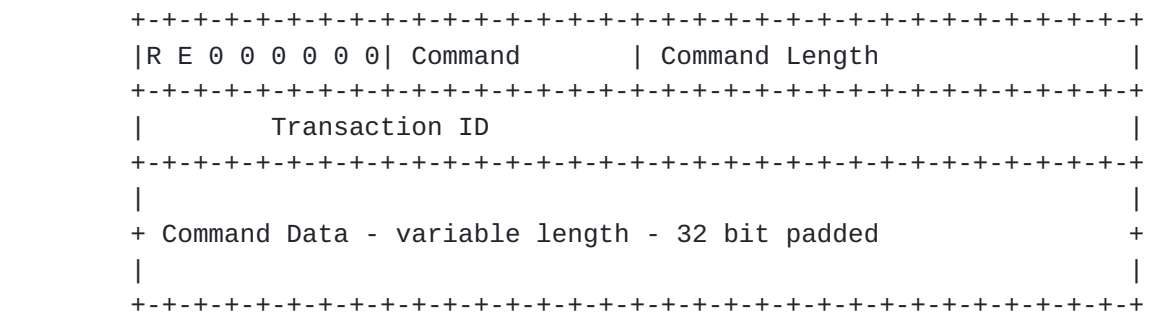



Command Block




...

Command Block



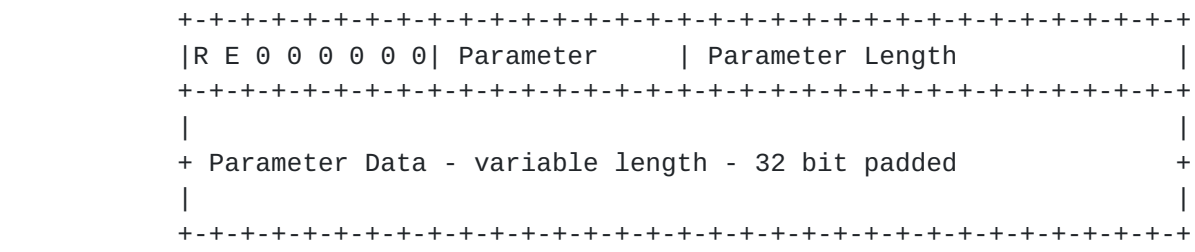

Each command block starts with a command header that includes extension bits, the command, and the length of the command data (not including the command header). The transaction-id is a random number and is preserved as the message is forwarded from one hop to the next. In the command block, the R bit, if set, indicates that the peer processing the request must be able to understand this command or else an error response MUST be returned (a peer that simply forwards is not required to look at or understand the command

blocks). The E bit indicates that even if this command is not understood, it MUST be echoed in any response.

The last Command Header Block in the message is typically a SIGNATURE command that computes a signature over all the previous command blocks.

Each command typically has some fixed format data at the beginning of it that carries the information that must occur in every command of that type, followed by a series of optional parameters. The first byte of the optional parameters has the same semantics as the first byte of the Command block that indicates whether the receiver needs to understand the parameter or not. The second byte defines the actual parameter type (which are IANA registered). The data length follows this in the third and forth byte.

Parameter Block




## 6.2. Data Storage and Retrieval

### 6.2.1. STORE

Stores a single copy of data in DHT. Includes a time to live for the data.

Parameters: locus, type, data, expiration time, data signature, signature, [etag]

Note the locus can be different than the destination when used for storing redundant data.

The expiration time is an absolute time to stop replay attacks, as described in the Security section.

Each time data is stored that is not bitwise identical to the previous data, the storing peer updates an entity-tag. If an etag is supplied in the command, then the operation will return an error if the current data does not have an entity-tag that matches the current etag.

#### 6.2.2. FETCH

Retrieves copy of data that is bitwise identical to the data in the store command

Parameters: locus, type, [etag]

If the entity tag of the data matches the optional etag in the FETCH, then a special response of SUCCESS-ETAG-MATCH is returned and no data is returned.

Response: data, data signature

#### 6.2.3. REMOVE

Removes data - can only be done by the user that stored the data.

Parameter: locus, signature, [etag]

#### 6.2.4. FIND

Returns the first instance of a stored data of a particular type that has a locus greater than or equal to the parameter

Need to also support returning the number of loci of the specified type that a peer is storing values for, as well as the range of locus space the peer is responsible for.

Parameter: locus, type

Responses: data locus, data, data signature, loci responsibility range, number of loci stored

### 6.3. DHT Maintenance

Many DHTs will not need all of these, but some will need to use them.

#### 6.3.1. JOIN

Used to indicate the sender is a new peer joining the DHT

Parameters: joining peer id , user-id, signature

Response: list of existing peers that the sender might be interested in knowing about

### 6.3.2. LEAVE

Used to indicate that the sender is about to leave the DHT

### 6.3.3. UPDATE

Used to indicate that the sender wishes to flag that they exist and that the receiver may want to take some action, as a result of their existence, to deal with the stability of the DHT.

This one is highly dependent on the actually DHT algorithm. It may be possible to define some common identifiable peers such as 1st successor, nth successor, nth predecessor, other peer in finger table, and so on.

## 6.4. Connection Management

### 6.4.1. CONNECT

A node sends a CONNECT command when it wishes to establish a direct TCP or UDP connection to another node for the purposes of sending ASP messages or application layer protocol messages, such as SIP. Detailed procedures for the CONNECT and its response are described in Section 8.3.

The attributes included in the CONNECT command and its response are:

- One or more candidate attributes. Each candidate attribute has an IP address, IP address family, port, transport protocol, priority, foundation, component ID, STUN type and related address.
- One username fragment.
- One password.
- One Next-Protocol attribute. This attribute contains a 16-bit port number. This port number represents the IANA registered port of the protocol that is going to be sent on this connection. For SIP, this is 5060 or 5061, and for ASP is TBD. By using the IANA registered port, we avoid the need for an additional registry and allow ASP to be used to set up connections for any existing or future application protocol.
- One fingerprint attribute (from RFC 4572 [RFC4572].
- An active/passive/actpass attribute from RFC 4145 [RFC4145].

TODO: fill in binary encoding formats

### 6.4.2. PING

Tests connectivity along a path. Can be addressed to a specific locus, in which case it is routed to the responsible peer to respond,

or can be addressed to any locus, in which case the first peer to receive it will respond. Can be sent with anycast or multicast so it must have a small response that does not fragment and the receiver needs to be able to deal with multiple responses. Probably need the responder to insert a random response id.

Nothing signed on this one.

Responses: peer id of actual responding peer, label stack that the responding peer received

### 6.5. Data Signature

#### 6.5.1. SIGNATURE

Time-stamp - not sure if this is needed to limit replay window or not

serial number of certificate used to sign

Signature

## 7. Forwarding Operations

## 8. Transport Operations

TODO - All transport flows need to have an associated label. SHOULD be unique to this peer or host and only use bottom 20 bits.

Number of retransmissions determines rate at which failure detection can occur - need to keep in lower than say SIP was - may have to be parameter of DHT instance

Need to make sure we can DEMUX this from other things - is a magic number needed at top of packet?

### 8.1. Framing for stream transports

For TLS session, first the length of the message is sent as a 32 bit integer followed by the message. If the top two bits of the length are not set to zero, the receiver should consider this an error and close this stream. These bits are reserved for future extensibility.

### 8.2. Framing for datagram transports

TODO - deal with retransmissions, TCP rate friendly congestion control, and fragmentation of large packets above the DTLS layer.

Is a peer that routes a command transaction state-full on the command? Who runs a timer on a command to time it out? Who deals with retransmissions - has to be link by link. Suspect we can make all retransmission and timer at the original commanding peer and allow all forwarding peers to be stateless other than the issue of DTLS retransmissions - which will be a nightmare.

### 8.3. ICE and Connection Formation

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

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

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

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

#### 8.3.1. Overview

To utilize ICE, the CONNECT method provides a basic offer/answer operation that exchanges a set of candidates for a single "stream". In this case, the "stream" refers not to RTP or other types of media, but rather to a connection for ASP itself or for SIP signaling. The CONNECT request contains the candidates for this stream, and the CONNECT response contains the corresponding answer with candidates for that stream. Though CONNECT provides an offer/answer exchange, it does not actually carry or utilize Session Description Protocol (SDP) messages. Rather, it carries the raw ICE parameters required for ICE operation, and the ICE spec is utilized as if these parameters had actually been used in an SDP offer or answer. In essence, ICE is utilized by mapping the CONNECT parameters into an SDP for the purposes of following the details of ICE itself. That

avoids the need for ASP to respecify ICE, yet allows it to operate without the baggage that SDP would bring.

ICE uses server reflexive and relayed candidates learned from STUN and TURN servers. With ASP, the nodes in the P2P network can provide TURN and STUN services for other nodes. Using a bootstrapping STUN server on the public Internet, a node learns with some probability that it is not behind a NAT or firewall. If it believes it is probably not behind one, it writes itself into the P2P network using a particular algorithm described below. When it comes time to gather a STUN or TURN server, an agent uses the algorithm described below to gather several servers of each type. Several servers are used for redundancy, to handle failures or cases where the server is not actually behind a NAT (which will result in the connectivity check through that server failing).

In addition, ASP 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.

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

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

### 8.3.2. TURN and STUN Server Insertion

Open Issue: We are still working on the algorithm in the this section and as it is currently described, there are some security issues. Expect improvement in the next release :-)

When a node starts up, it learns its bootstrap STUN server. It does this by taking the name of the DHT (for example, "example.com") and querying the DNS for the STUN server for that domain. The administrator of this domain MUST provide a STUN server. This

bootstrap STUN server MUST be on the public Internet. The node then utilizes the diagnostics STUN usage [I-D.ietf-behave-nat-behavior-discovery]. If, based on this, the agent believes it is not behind a NAT or firewall, it MUST consider itself a candidate STUN server and SHOULD consider itself a candidate TURN server.

Next, the node gets an estimate N of the number of nodes in the P2P network. This computation is actually very straightforward. A given node has connections to other nodes in the DHT. For each such node i, the node directs a FIND command to it, and will get back the range of loci that this neighbor is responsible for. For that node i, an estimate Ei of the total number of nodes is the size of the hashspace divided by the number of loci in this range. Then, the node takes the average Ei across all connections. The result is an estimate of N.

Each node is configured with an estimate of the typical fraction, d, of the population that will serve as STUN or TURN servers. For STUN servers, this SHOULD be d_stun=.1, and for TURN, d_turn=.01.

- OPEN ISSUE: Need to have a way to estimate this by ring measurements.

If the node is a candidate STUN server, it picks a random number uniformly distributed between 0 and d_stun*N. This number is used as a seed, and the resulting value is a locus in the hashspace. The node performs a STORE operation at this locus, using the STUN server data type. This operation SHOULD be repeated four more times (for a total of five stores to different loci). If the node is a candidate TURN server, it performs the same process, but using d_turn.

- This process causes each seed between 0 and Nd to have, on average, five values stored there. This allows the workload of storing TURN and STUN servers to be uniformly distributed across the ring. It also allows for a single query to return five TURN or STUN servers on average, the exact number needed in Section 8.3.3.

### 8.3.3. Gathering Candidates

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

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

The ICE specification assumes that an ICE agent is configured with, or somehow knows of, TURN and STUN servers. ASP provides a way for an agent to learn these by querying the ring. Using the procedures in Section 8.3.2, an agent estimates the number of nodes in the P2P network, N. If the node is utilizing TURN, it then computes a random number uniformly distributed between 0 and d_turn, and uses the resulting value as a seed. It then performs a FETCH targeted to the locus for that seed, asking for data of type TURN server. The result will, on average, return five TURN servers. The agent then uses each of these as its TURN servers for this CONNECT. If the agent is not utilizing TURN, it computes a random number uniformly distributed between 0 and d_stun, and uses the resulting value as a seed. It then performs a FETCH targeted to the locus for that seed, asking for data of type STUN server. The result will, on average, return five STUN servers. The agent then uses each of these as its STUN servers for this CONNECT.

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

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

### 8.3.4. Encoding the CONNECT Message

Section 4.3 of ICE describes procedures for encoding the SDP. Instead of actually encoding an SDP, the candidate information (IP address and port and transport protocol, priority, foundation, component ID, type and related address) is carried within the attributes of the CONNECT command or its response. Similarly, the username fragment and password are carried in the CONNECT message or its response. Section 6.4.1 describes the detailed attribute encoding for CONNECT. The CONNECT command and its response do not contain any default candidates or the ice-lite attribute, as these features of ICE are not used by ASP. The CONNECT command and its response also contain a Next-Protocol attribute, with a value of SIP or ASP, which indicates what protocol is to be run over the connection. The ASP CONNECT command MUST only be utilized to set up connections for application protocols that can be multiplexed with STUN and ASP itself.

Since the CONNECT command contains the candidate information and short term credentials, it is considered as an offer for a single

media stream that happens to be encoded in a format different than SDP, but is otherwise considered a valid offer for the purposes of following the ICE specification. Similarly, the CONNECT response is considered a valid answer for the purposes of following the ICE specification.

Since all messages with ASP are secured between nodes, the node MUST implement the fingerprint attribute of RFC 4572 [RFC4572], and encode it into the CONNECT command and response as described in Section 6.4.1. This fingerprint will be matched with the certificates utilized to authenticate the ASP CONNECT command and its response.

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

### 8.3.5. Verifying ICE Support

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

### 8.3.6. Role Determination

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

### 8.3.7. Connectivity Checks

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

### 8.3.8. Concluding ICE

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

answer exchanges.

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

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

### 8.3.9. Subsequent Offers and Answers

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

### 8.3.10. Media Keepalives

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

### 8.3.11. Sending Media

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

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

### 8.3.12. Receiving Media

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

## 9. DHT Algorithms

This section describes what needs to be specified when specifying a new DHT Algorithm.

Describe this from point of view of event driven system. Events include a user deciding to join, leave, etc. and protocol events such as receive update, join, etc. When an event is received, DHT defines a series of things to send and things to store - the DHT algorithm specifies what message gets sent on each event and what gets stored.

### 9.1. Generic Algorithm Requirements

TODO

- How to store redundant encoding
- Algorithm to go from a seed, such as a user name, to a locus
- Joining procedures
- Stabilization procedures
- Exit procedures
- Keep alive procedures
- Routing and loops
- Merging procedures to recovering from network partitions
- Detecting disconnection from rest of peers

### 9.2. DHT API

Note: This section need is just a very rough strawman to start thinking about the right issues.

In order to allow ASP to be used with existing and new DHT algorithms, it is important to define a clear model on how different DHTs are "plugged" into ASP. In order to make it easy to add new DHT algorithms, from the perspective of protocol changes, code changes and specification work, ASP defines an abstract API that exists between the Routing and Replication Logic and the DHT.

This API takes the form of an event driven system. Events arrive as a consequence of operations invoked by the usage and by arrival of messages over the wire. For certain events, the DHT layer is expected to provide a response. In other cases, the DHT layer is just notified of the event. In response, the DHT layer can inject messages, typically ones used for DHT maintenance.

The events passed to the DHT layer are:

onMessageToForward(Peer-ID DestinationPeerID): When a message is received by the transport layer, the destination label set is examined. If the top-most label does not identify the node itself, the message needs to be forwarded closer towards the destination. The routing and replication logic layer maintains a series of connections to other nodes. However, the decision about which connection to use is a function of the DHT. So, when such a message arrives, the routing and replication logic layer invokes this event and passes the target Peer-ID to the DHT. The DHT consults its routing tables and passes back to the routing and replication layer the specific connection on which to forward the message.

onStore(): When a STORE command is received, the actual storage of data, including authorization, quota management, and data processing are handled by the routing and replication logic layer. However, the determination of which peer nodes at which the data must be replicated is a function of the DHT. Thus, when a store is received, the DHT algorithm is notified, and it passes back the set of other nodes at which to perform the store by sending another STORE command to those nodes. Fetch and remove operations do not require interaction with the DHT layer.

onFind(): When a FIND command is received, the computing the number of loci of the particular type is handled by the routing and replication logic layer. However, the DHT layer must indicate the range of loci the peer is responsible for. The response to the onFind() operation returns this number.

onJoin(Peer-ID NewPeer): When a join is received and targeted for this node, the authentication is handled by the routing and replication logic layer. However the DHT algorithm does the real work of processing the join. It does so by passing back to the DHT a set of Peer-IDs that the joining node might be interested in. It can also send DHT maintenance messages as needed.

onLeave(Peer-ID LeavingPeer: When a LEAVE is received and targeted for this node, the authentication is handled by the routing and replication logic layer. However the DHT algorithm does the real work of processing the leave. It can send DHT maintenance messages as needed.

onUpdate(): When an UPDATE is received, its attributes are passed to the DHT. Update processing is entirely dependent on the DHT algorithm.

onConnectionFailure(Peer-ID Neighbor): The routing and replication logic layer will perform keepalives on each connection to other peers. When a connection fails or timeouts, the DHT algorithm is informed of this fact.

onJoinMyself(): When the routing and replication logic layer decides to join the network, it asks the DHT layer to do this for it. The DHT layer will generate messages as needed to affect the joining into the DHT.

onLeaveMyself(): When the routing and replication logic layer decides to leave the network, it asks the DHT layer to do this for it. The DHT layer will generate messages as needed to affect the leaving of the DHT.

The "commands" that the DHT layer can invoke include all of the commands supported by ASP. However, the DHT layer would not construct the message or perform authentication. Rather, it would instruct the routing and replication logic to send the message, and include attributes that the DHT layer wants to include in the message. When a response is received, this response is passed to the DHT layer.

## 10. Chord Algorithm

This algorithm is assigned the name chord-128-2-32 to indicate it is based on Chord, and it uses a 128 bit hash function, stores 2 redundant copies of all data, and has finger tables with 32 entries.

### 10.1. Overview

The algorithm described here is a modified version of the Chord algorithm. Each peer keeps track of a finger table of 32 entries and a neighborhood table of 6 entries. The neighborhood table contains the 3 peers before this peer and the 3 peers after it in the DHT ring. The first entry in the finger table contains the peer half-way around the ring from this peer; the second entry contains the peer that is 1/4 of the way around; the third entry contains the peer that is 1/8th of the way around, and so on. Fundamentally, the chord data structure can be thought of a double-linked list formed by knowing the successors and predecessor peers in the neighborhood table, sorted by the peer-id. As long as the successor peers are correct, the DHT will return the correct result. The pointers to the prior peers are kept to enable inserting of new peers into the list structure. Keeping multiple predecessor and successor pointers makes it possible to maintain the integrity of the data structure even when consecutive peers simultaneously fail. The finger table forms a skip list too, so that entries in the linked list can rapidly be found - it needs to be there so that peers can be found in O(log(N)) time instead of the typical O(N) time that a linked list would provide.

A peer, n, is responsible for a particular locus k if k is less than or equal to n and k is greater than p, where p is the peer id of the

previous peer in the neighborhood table. Care must be taken when computing to note that all math is modulo 2^128.

### 10.2. Routing

If a peer is not responsible for a locus k, then it routes a command to that location by routing it to the peer in either the neighborhood or finger table that has the largest peer-id that is still less than or equal to k.

### 10.3. Redundancy

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

### 10.4. Joining

[rewrite to be more event oriented]

When a peer (with peer-id n) joins the ring, it first does a PING to peer n to discover the peer, called p, that is currently responsible for the loci this peer will need to store. It then does a PING on p+1 to discover p0, a PING on p0+1 to discover p1, and finally a PING on p1+1 to discover p2. The values for p, p0,p1, and p2 form the initial values of the neighborhood table. (The values for the two peers before p will be found at a later stage when n receives an UPDATE.) The peer then fills the finger table by, for the i'th entry, doing a PING to peer (n+2^(numBitsInPeerId-i). The peer then uses the CONNECT command to form connections to all the peers in the neighborhood and finger tables. The finger table is initialized before starting to accept data so that certificates can be looked up to check signatures.

Next, peer n indicates it is ready to start receiving data by sending a JOIN command to peer p. At this point peer p transfers a copy of the data it will need to store on peer n by sending a series of STORE commands to transfer the data. Once peer p has finished sending all the STORE commands to transfer the data, it changes its neighborhood

table to include n and then sends an UPDATE command to all the peers in the neighborhood table. Each one of the UPDATES contains the peer-id of all the entries in peer p's neighborhood table as well as the id for peer n.

### 10.5. Receiving UPDATEs

When a peer, n, receives an UPDATE command, it looks at all the peer-ids in the UPDATE and at its neighborhood table and decides if this UPDATE would change its neighborhood table. If any peer, p, would be added or removed from the neighborhood table, the peer sends a PING to peer p; if this fails, peer p is removed from the neighborhood table, and if it succeeds, p is added to the table. After the PINGs are done, if the table has changed, peer n attempts to open a new connection to any new peers in the neighborhood table by sending them a CONNECT command. If the neighborhood table changes, the peer sends an UPDATE command to each of its neighbors.

### 10.6. Sending UPDATEs

Every time a connection to a peer in the neighborhood set is lost (as determined by connectivity pings), the peer should remove the entry from its neighborhood table and send an UPDATE to all the remaining neighbors. The update will contain all the peer-ids of the current entries of the table (after the failed one has been removed).

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

### 10.7. Stabilization

About every hour, a peer should send UPDATE commands to all of the peers in its neighborhood table.

About every hour a peer should select a random entry from the finger table and do a PING to peer (n+2^(numBitsInPeerId-i). If this returns a different peer than the one currently in this entry of the peer table, then a new connection should be formed to this peer and it should replace the old peer in the finger table.

### 10.8. Leaving

Unfortunately most peers leave by just disconnecting. This is not good. A more orderly way to disconnect is the following. First the

leaving peer stops responding to PINGS. It then sends CLOSE commands on any connections it has open. Next it sends an UPDATE to all of the peers in its neighbor set (both peers ahead and behind it in the ring) which includes its other neighbors but MUST NOT include its own peer id. It then does a STORE for each locus it has, to transfer that data to the new responsible peer. Finally it closes any connections that it has open.

## 11. Enrollment and Bootstrap

Fixes the DHT and DHT parameters

Provides user name and CERT

May provide multiple DHTs for insertions multiple rings during migration from one to another

Specify some XML over HTTP based enrollment process to a central server

Discuss P2P-Network-Id creation. The top 24 bits are a hash of the P2P-Network-ID name (for example, "example.org"), while the bottom 8 bits are controlled by the site and are used for different versions of the ring.

## 12. Usages

### 12.1. Generic Usage Requirements

### 12.2. SIP Usage

### 12.3. STUN/TURN Usage

### 12.4. Certificate Store Usages

## 13. Security Considerations

### 13.1. Overview

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

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

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

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

Note that since only users of this DHT need to validate a certificate, this usage does not require a global PKI. It does, however, require a central enrollment authority which acts as the certificate authority for the DHT.

## 13.2. General Issues

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

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

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

### 13.2.1. Storage Security

The foundation of storage security in ASP is that any given locus/ type code pair (a slot) is deterministically bound to some small set of certificates. In order to write data in a slot, the writer must

prove possession of the private key for one of those certificates. Moreover, all data is stored signed by the certificate which authorized its storage. This set of rules makes questions of authorization and data integrity - which have historically been thorny for DHTs - relatively simple.

#### 13.2.1.1. Authorization

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

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

For example, consider the certificate with the following properties:

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

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

#### 13.2.1.2. Distributed Quota

Being a peer in a DHT carries with it the responsibility to store data for a given region of the DHT. However, if clients were allowed to store unlimited amounts of data, this would create unacceptable burdens on peers, as well as enabling trivial denial of service attacks. ASP addresses this issue by requiring each usage to define maximum sizes for each type of stored data. Attempts to store values exceeding this size SHOULD be rejected. Because each slot is bound to a small set of certificates, these size restrictions also create a

distributed quota mechanism, with the quotas administered by the central enrollment server.

Allowing different types 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. Because peers know at joining time what usages they must support (see Section XXX), peers can to some extent predict their storage requirements.

#### 13.2.1.3. 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 "expiration time" values in each store. An expiration time 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 expiration times be monotonically increasing expiration time (see Section XXX ). Storing peers MUST reject STORE requests with expiration times smaller than those they are currently storing.

#### 13.2.1.4. Residual Attacks

The mechanisms described here provide a high degree of security, but some attacks remain possible. Most simply, it is possible for storing nodes to refuse to store a value (reject any request). In addition, a storing node can deny knowledge of values which it previously accepted. To some extent these attacks can be ameliorated by attempting to store to/retrieve from replicas, but a retrieving client at least has no way of knowing what it should do so.

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

### 13.2.2. Routing Security

Because the storage security system guarantees (within limits) the integrity of the stored data, routing security focuses on stopping the attacker from performing a DOS attack on the system by mis-routing requests in the DHT. There are a few obvious observations to make about this. First, it is easy to ensure that an attacker is at

least a valid peer in the DHT. Second, this is a DOS attack only. Third, if a large percentage of the peers on the DHT are controlled by the attacker, it is probably impossible to perfectly secure against this.

#### 13.2.2.1. Background

In general, attacks on DHT routing are mounted by the attacker arranging to route traffic through or two nodes it controls. In the Eclipse attack [REF: 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 [REF: Sybil], the attacker registers a large number of nodes and is therefore able to capture a large amount of the traffic through the DHT.

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

#### 13.2.2.2. Admissions Control

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

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

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

#### 13.2.2.3. Peer Identification and Authentication

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

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

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

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

#### 13.2.2.4. Residual Attacks

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

### 13.3. SIP-Specific Issues

#### 13.3.1. Fork Explosion

#### 13.3.2. Malicious Retargeting

#### 13.3.3. Privacy Issues

## 14. IANA Considerations

### 14.1. DHT Types

### 14.2. Stored Data Types

### 14.3. Command & Responses Types

### 14.4. Parameter Types

## 15. Examples

## 16. Open Issues

### 16.1. Peer-id and locus size

Should these be 128 bits? Should the messages signal the size of them and the implementations use variable size for them?

### 16.2. More efficient FIND command

It would be possible for a peer that had an empty list for a service like STUN to keep pointers to the previous and next peers that did have one a peer that performed the service and manage this as a linked list. When a FIND command came, it could return a hint of likely next and previous peers that might have pointers to a peer that provided the service.

### 16.3. Generation, E-Tags, link thing

Should all data have a generation ID so that instead of fetching all the data you can just see if it has changed?

### 16.4. Future upgrade support

How do we do required support like tags to add new commands?

What about extension blocks inside commands?

## 17. Acknowledgments

## 18. Appendix: Operation with SIP clients outside the DHT domain

## 19. Appendix: Notes on DHT Algorithm Selection

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

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

## 20. References

### 20.1. Normative References

[I-D.ietf-mmusic-ice]
Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", draft-ietf-mmusic-ice-16 (work in progress), June 2007.

[I-D.ietf-behave-rfc3489bis]
Rosenberg, J., "Session Traversal Utilities for (NAT) (STUN)", draft-ietf-behave-rfc3489bis-06 (work in progress), March 2007.

[I-D.ietf-behave-turn]
Rosenberg, J., "Obtaining Relay Addresses from Simple Traversal Underneath NAT (STUN)", draft-ietf-behave-turn-03 (work in progress), March 2007.

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

### 20.2. Informative References

[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.

[I-D.willis-p2psip-concepts]
Willis, D., "Concepts and Terminology for Peer to Peer SIP", draft-willis-p2psip-concepts-04 (work in progress), March 2007.

[RFC4864] Van de Velde, G., Hain, T., Droms, R., Carpenter, B., and E. Klein, "Local Network Protection for IPv6", RFC 4864, May 2007.

[I-D.ietf-behave-nat-behavior-discovery]
MacDonald, D. and B. Lowekamp, "NAT Behavior Discovery Using STUN", draft-ietf-behave-nat-behavior-discovery-00 (work in progress), February 2007.

[I-D.ietf-mmusic-ice-tcp]
Rosenberg, J., "TCP Candidates with Interactive Connectivity Establishment (ICE", draft-ietf-mmusic-ice-tcp-03 (work in progress), March 2007.

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

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

[RFC4572] Lennox, J., "Connection-Oriented Media Transport over the Transport Layer Security (TLS) Protocol in the Session Description Protocol (SDP)", RFC 4572, July 2006.

## 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

Jonathan Rosenberg
Cisco
Edison, NJ
USA

Email: jdrosen@cisco.com

Eric Rescorla
Network Resonance
3246 Louis Road
Palo Alto, CA 94303
USA

Phone: +1 650 320-8549
Email: fluffy@cisco.com

## Full Copyright Statement



## Intellectual Property



## Acknowledgment

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