A Kademlia-based DHT for Resource Lookup in P2PSIP

draft-cirani-p2psip-dsip-dhtkademlia-00

Status of this Memo

By submitting this Internet-Draft, each author represents that any applicable patent or other IPR claims of which he or she is aware have been or will be disclosed, and any of which he or she becomes aware will be disclosed, in accordance with Section 6 of BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on April 27, 2008.

Copyright Notice

Copyright (C) The IETF Trust (2007).
Abstract

This draft describes a Kadmeia-based protocol for Resource Lookup in P2P SIP. The proposed protocol is based on dSIP, a SIP-based protocol proposed by other authors as generic framework for a distributed SIP Location Service. Although the dSIP authors have obsoleted the draft by a newer approach based on a binary protocol named RELOAD, we are still considering this SIP-based approach, due to implementation simplicity, possibility of reuse of already available SIP stack implementations, easy integration into existing UAs, minimization of the number of required protocols for a P2P UA, and widespread support for (and relative maturity of) the SIP standard.
Table of Contents

1. Introduction ................................................. 4
2. Terminology .................................................. 5
   2.1. Definitions ............................................ 5
3. Background .................................................. 6
   3.1. Kademlia ................................................. 6
4. Routing Table and Connection Table .......................... 7
   4.1. The Kademlia protocol ................................. 8
   4.2. Node lookup procedure ................................ 8
5. Message Syntax ............................................... 10
   5.1. The DHT-PeerID Header ................................ 10
      5.1.1. Hash Algorithms .................................. 10
      5.1.2. DHT Name Parameter ............................... 10
   5.2. The DHT-Link Header ................................... 10
6. Kademlia Overlay Algorithm .................................. 11
   6.1. Routing Table .......................................... 11
   6.2. Starting a New Overlay ................................. 11
   6.3. Peer Admission ......................................... 11
      6.3.1. Constructing a Peer Registration ................. 11
      6.3.2. Processing and Routing the Peer Registration .. 12
      6.3.3. Admitting the Joining Peer ...................... 12
   6.4. Kademlia query processing .............................. 13
   6.5. Kademlia Graceful Leaving ............................. 13
   6.6. DHT Maintenance ....................................... 13
   6.7. Peer Failure .......................................... 13
   6.8. Resource Replicas ..................................... 13
7. Examples ...................................................... 15
   7.1. Peer registration ...................................... 15
   7.2. Peer query ............................................. 17
   7.3. User registration ...................................... 20
   7.4. Session establishment .................................. 21
8. Implementation ................................................. 25
9. Security Considerations ....................................... 26
10. IANA Considerations ......................................... 27
11. References .................................................. 28
   11.1. Normative References ................................ 28
   11.2. Informative References ............................... 29
Authors’ Addresses .............................................. 30
Intellectual Property and Copyright Statements .................. 31
1. Introduction

This draft describes a Kademlia-based protocol for Resource Lookup in P2PSIP. The proposed protocol is based on dSIP [I-D.bryan-p2psip-dsip], a SIP-based protocol proposed by other authors as generic framework for a distributed SIP Location Service. On top of dSIP, different DHTs can be supported in a pluggable module fashion. The dSIP specification for the support of Chord has already been defined in [I-D.zangrilli-p2psip-dsip-dhtchord]. Although the dSIP authors have obsoleted the draft by a newer approach based on a binary protocol named RELOAD [I-D.bryan-p2psip-reload], at this moment we are not sure that a binary protocol should be preferred to a text-based or SIP-based one, and, due to this, we are still considering this approach. A SIP-based solution might have some advantages such as simplicity of the implementation of a text-based approach, the possibility of reuse of already available SIP stack implementations, easy integration into existing UAs, minimization of the number of required protocols for a P2P UA, and widespread support for (and relative maturity of) the SIP standard. Moreover the drawbacks of such an approach do not appear so relevant by now to justify an abrupt change of course. Particularly, in this draft we chose to use dSIP as basis for the specification of a Kademlia-based protocol to be used for Resource Lookup in P2PSIP, due to relative maturity of its specification [I-D.bryan-p2psip-dsip]. Kademlia was selected because of its simplicity, performance, and widespread use in P2P application such as eMule and BitTorrent.

As practical demonstration of the feasibility of our solution we have already provided a dSIP implementation of both Chord and Kademlia, available from <http://www.mjsip.org/projects/p2psip/p2psip_dsip_071025.zip> based on the MjSip SIP stack [mjsip].

For simplicity of presentation this draft follows the same structure and style of the other proposed draft specifying the use of Chord with dSIP [I-D.zangrilli-p2psip-dsip-dhtchord].

Next versions of this draft may or may not rely on dSIP, which could be replaced by new text-based (SIP-based) or binary protocols.
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].

Terminology defined in RFC 3261 [RFC3261] is used without definition.

We use the terminology and definitions from the dSIP: A P2P Approach to SIP Registration and Resource Location [I-D.bryan-p2psip-dsip] and the Concepts and Terminology for Peer to Peer SIP [I-D.willis-p2psip-concepts] drafts extensively in this document. Other terms relating to P2P or new to this document are defined when used and are also defined in Definitions (Section 2.1). We suggest reviewing these drafts and the Definitions (Section 2.1) section before reading the remainder of this document.

In many places in this document, 10 hexadecimal digit values are used in examples as SHA-1 hashes. In reality, these hashes are 40 digit. They are shortened in this document for clarity only.

2.1. Definitions

Please also see the dSIP: A P2P Approach to SIP Registration and Resource Location [I-D.bryan-p2psip-dsip] draft and the P2PSIP concepts and terminology [I-D.willis-p2psip-concepts] draft for additional terminology. We do not redefine terms from that draft here.

- Kademlia: a DHT for decentralized peer-to-peer networks based on a XOR metric to compute the distance between two nodes on the network that exploits the fact that long-time active nodes are most likely to stay active.

- Routing Table: the list of peers that a peer uses to send messages to. The finger table contains many entries about peers with similar IDs, and fewer entries about more remote IDs.
3. Background

3.1. Kademlia

The Kademlia system [Kademlia] is one particular popular DHT algorithm. In Kademlia, resource with Resource-ID r will be stored by the k peers with Peer-ID closest to n, ensuring that every Resource-ID is associated with some peer.
4. Routing Table and Connection Table

For each 0 <= i < 160, every node keeps a list of <IP address; UDP port; Node ID> triples for nodes of distance between $2^i$ and $2^{i+1}$ from itself. These lists are called k-buckets. Each k-bucket is kept sorted by time: least-recently seen node at the head, most-recently seen at the tail. For small values of i, the k-buckets will generally be empty (as no appropriate nodes will exist). For large values of i, the lists can grow up to size k, where k is a system-wide replication parameter. k is chosen such that any given k nodes are very unlikely to fail within an hour of each other (for example k = 20). When a Kademlia node receives any message (request or reply) from another node, it updates the appropriate k-bucket for the sender’s node ID. If the sending node already exists in the recipient’s k-bucket, the recipient moves it to the tail of the list. If the node is not already in the appropriate k-bucket and the bucket has fewer than k entries, then the recipient just inserts the new sender at the tail of the list. If the appropriate k-bucket is full, however, then the recipient pings the k-bucket’s least-recently seen node to decide what to do. If the least-recently seen node fails to respond, it is evicted from the k-bucket and the new sender inserted at the tail. Otherwise, if the least-recently seen node responds, it is moved to the tail of the list, and the new sender’s contact is discarded.

k-buckets effectively implement a least-recently seen eviction policy, except that live nodes are never removed from the list. By keeping the oldest live contacts around, k-buckets maximize the probability that the nodes they contain will remain online. A second benefit of k-buckets is that they provide resistance to certain DoS attacks.

Each peer keeps information about how to contact some number of other peers in the overlay. In terms of the overlay network, these are the neighbors of the peer, since they are reachable in one hop. In Chord [Chord] the peer keeps track of one or more of its immediate predecessor peers, as well as one or more successor peers. The peer also keeps a table of information about other neighbors called a finger table, consisting of peers distributed around the overlay.

Note that dSIP defines a routing table as the set of peers that a peer knows about and uses to send messages to when routing. The routing table is the combination of the predecessor, successor and finger table.
4.1. The Kademlia protocol

The Kademlia protocol consists of four RPCs: PING, STORE, FIND NODE, and FIND VALUE.

- **PING**: The PING RPC probes a node to see if it is online. This RPC involves one node sending a PING message to another, which presumably replies. This has a two-fold effect: the recipient of the PING must update the bucket corresponding to the sender; and, if there is a reply, the sender must update the bucket appropriate to the recipient.

- **STORE**: STORE instructs a node to store a \(<key, value>\) pair for later retrieval. This is a primitive operation, not an iterative one.

- **FIND NODE**: FIND NODE takes a 160-bit ID as an argument. The recipient of the RPC returns \(<IP address; UDP port; Node ID>\) triples for the k nodes it knows about closest to the target ID. These triples can come from a single k-bucket, or they may come from multiple k-buckets if the closest k-bucket is not full. In any case, the RPC recipient must return k items (unless there are fewer than k nodes in all its k-buckets combined, in which case it returns every node it knows about). This is a primitive operation, not an iterative one.

- **FIND VALUE**: FIND VALUE behaves like FIND NODE, returning \(<IP address; UDP port; Node ID>\) triples, with one exception: if the RPC recipient has received a STORE RPC for the key, it just returns the stored value. This is a primitive operation, not an iterative one.

4.2. Node lookup procedure

The most important procedure a Kademlia participant must perform is to locate the k closest nodes to some given node ID. We call this procedure a node lookup. Kademlia employs an iterative algorithm for node lookups (although the paper describes it as recursive). The lookup initiator starts by picking alpha nodes from its closest non-empty k-bucket (or, if that bucket has fewer than alpha entries, it just takes the alpha closest nodes it knows of). The initiator then sends parallel, asynchronous FIND NODE RPCs to the alpha nodes it has chosen. alpha is a system-wide concurrency parameter, such as 3. In the iterative step, the initiator resends the FIND NODE to nodes it has learned about from previous RPCs. This iteration can begin before all alpha of the previous RPCs have returned. Kademlia uses alpha = 3, the degree of parallelism used. It appears that this value is optimal. There are at least three approaches to managing
parallelism. The first is to launch alpha probes and wait until all have succeeded or timed out before iterating. This is termed strict parallelism. The second is to limit the number of probes in flight to alpha; whenever a probe returns a new one is launched. We might call this bounded parallelism. A third is to iterate after what seems to be a reasonable delay (duration unspecified), so that the number of probes in flight is some low multiple of alpha. This is called loose parallelism. Of the k nodes the initiator has heard of closest to the target, it picks alpha that it has not yet queried and resends the FIND NODE RPC to them. Nodes that fail to respond quickly are removed from consideration until and unless they do respond. If a round of FIND NODEs fails to return a node any closer than the closest already seen, the initiator resends the FIND NODE to all of the k closest nodes it has not already queried.

The lookup terminates when the initiator has queried and gotten responses from the k closest nodes it has seen. Most operations are implemented in terms of the above lookup procedure:

- **iterativeStore**: this is the Kademlia store operation. The initiating node does an iterativeFindNode, collecting a set of k closest contacts, and then sends a primitive STORE RPC to each. iterativeStores are used for publishing or replicating data on a Kademlia network.

- **iterativeFindNode**: this is the basic Kademlia node lookup operation. As described above, the initiating node builds a list of k closest contacts using iterative node lookup and the FIND NODE RPC. The list is returned to the caller.

- **iterativeFindValue**: this is the Kademlia search operation. It is conducted as a node lookup, and so builds a list of k closest contacts. However, this is done using the FIND VALUE RPC instead of the FIND NODE RPC. If at any time during the node lookup the value is returned instead of a set of contacts, the search is abandoned and the value is returned. Otherwise, if no value has been found, the list of k closest contacts is returned to the caller.
5. Message Syntax

5.1. The DHT-PeerID Header

The routing algorithms used to implement the overlay is specified in the dht-param parameter in the DHT-PeerID header. The format of the DHT-PeerID header is defined in the dSIP [I-D.bryan-p2psip-dsip] draft.

5.1.1. Hash Algorithms

Implementations MUST support the SHA-1 [RFC3174] algorithm, which produces a 160 bit hash value. An implementation MAY rely on a secret initialization vector, key, or other shared secret to use the identifier as an HMAC, from RFC 2104 [RFC2104] such that no peer may join the overlay without knowledge of the shared secret, however this technique by itself does not protect the overlay against replay attacks. Security Extensions to dSIP [I-D.lowekamp-p2psip-dsip-security] provides information on how to protect against replay attacks and hash algorithms defined in that draft MAY be used in Kademlia implementations.

5.1.2. DHT Name Parameter

For this protocol, the dht-param token MUST be set to "Kademlia1.0".

A peer receiving a message with a dht-param other than "Kademlia1.0" SHOULD reject the message and return a 488 Not Acceptable Here response message.

Examples:

A peer with an SHA-1 hashed Peer-ID of a04d371e24 on IP 192.0.2.1. We include the required algorithm, and overlay as well as the optional expires header parameter.

DHT-PeerID: <sip:peer@192.0.2.1;peer-ID=a04d371e24>;algorithm=sha1; dht=Kademlia1.0;overlay=chat;expires=600

5.2. The DHT-Link Header

The DHT-Link header is not needed to implement this protocol. Information about other peers is carried in the Contact headers. Therefore there is no need to define the linktype and depth values used in the header.
6. Kademlia Overlay Algorithm

Each peer keeps track of a list of up to $k$ peers in its $k$-buckets. The Kademlia paper recommends keeping a number of $k$-buckets equal to the size in bits of the identifier, which is 160. However, it is possible to use a smaller number of $k$-buckets, as this affects only the efficiency, that is, node lookups would possibly return more distant contacts than with 160 $k$-buckets. There is no actual need to do this, though, unless to minimize memory usage by the peer. This would be useful for mobile devices, which feature a smaller amount of memory. Other than this case, the full number of $k$-buckets is recommended to increase performance. This is the choice made in this work.

6.1. Routing Table

A peer starting an overlay for the first time needs not to do anything special in order to construct the overlay. Its $k$-buckets are initialized as empty and will be populated as other peers join the overlay and messages are routed to the peer.

6.2. Starting a New Overlay

A peer that wishes to join an overlay (called the joining peer), constructs a Peer Registration message and sends it to the bootstrap peer. The bootstrap peer is also the admitting peer. After receiving a response from the bootstrap peer, the joining peer performs a node lookup targeting its own id in order to populate its $k$-buckets.

6.3. Peer Admission

To initiate the joining process, the joining peer constructs a Peer Registration and sends it to the bootstrap peer. The joining peer MUST construct the Peer Registration according the rules outlined in the dSIP [I-D.bryan-p2psip-dsip] draft. The joining peer MUST provide a DHT-PeerID header field in the Peer Registration and the dht-param part of the DHT-PeerID MUST be set to "*" or the token specified in the DHT Name Parameter Section 5.1.2.

6.3.1. Constructing a Peer Registration

Assume that a peer running on IP address 192.0.2.2 on port 5060 attempts to join the network by contacting a bootstrap peer at address 192.0.2.129. Further assume that 192.0.2.2 hashes to 463ac4b449 under SHA-1 (using a 10 digit hash for example simplicity), and that the overlay name is chat. An example message would look like this (neglecting tags):
REGISTER sip:192.0.2.129 SIP/2.0
To: <sip:peer@192.0.2.2;peer-ID=463ac4b449>
From: <sip:peer@192.0.2.2;peer-ID=463ac4b449>
Contact: <sip:peer@192.0.2.2;peer-ID=463ac4b449>
Expires: 600
DHT-PeerID: <sip:peer@192.0.2.2;peer-ID=463ac4b449>;algorithm=sha1;
  dht=Kademlia1.0;overlay=chat;expires=600
Require: dht
Supported: dht

6.3.2. Processing and Routing the Peer Registration

The Peer Registration is processed and routed according the rules
outlined in the dSIP [I-D.bryan-p2psip-dsip] draft.

6.3.3. Admitting the Joining Peer

When handling a Peer Registration from a joining peer, the admitting
peer MUST reply with a 200 response and update the appropriate
k-bucket according to the rules outlined in Section 4.

The admitting peer MUST verify that the joining peer’s Peer-ID is
valid. If the joining peer’s credentials are not valid, the message
should be rejected with a response of 493 Undecipherable. In
addition to verifying that the joining peer’s Peer-ID is valid, the
admitting peer MAY require an authentication challenge to the
REGISTER message. Once any challenge has been met, the admitting
will reply with a 200 OK message to the joining peer. As in a
traditional registration, the Contact in the 200 OK will be the same
as in the request, and the expiry time MUST be provided.

After receiving the 200 response the joining peer must populate its
routing table. This is done by performing a node lookup targeting
its own id. This process will let the joining peer know about the k
nodes that are closest to itself, which is the basic information that
a node needs in order to be an active node in the Kademlia network,
as it needs to be able to reply to a lookup request. As a side
effect, other nodes would know about the joining peer. The messages
exchanged in this process are peer query messages, which will be
shown in Section 6.4.

Continuing the example Peer Registration from the section above,
assume now that the peer with Peer-ID 47e46fa2cd and IP address
192.0.2.7 is currently responsible for 463ac4b449 in the namespace.
The response would look something like:
SIP/2.0 200 OK
To: <sip:peer@192.0.2.2;peer-ID=463ac4b449>
From: <sip:peer@192.0.2.2;peer-ID=463ac4b449>
Contact: <sip:peer@192.0.2.2;peer-ID=463ac4b449>
Expires: 600
DHT-PeerID: <sip:peer@192.0.2.2;peer-ID=463ac4b449>;algorithm=sha1;
dht=Kademlia1.0;overlay=chat;expires=600
Require: dht
Supported: dht

*After* the admitting peer sends the 200 response, it MUST update the appropriate k-bucket, and MUST obtain the information about the joining peer from the DHT-Peer header in the register request. It MUST NOT update the k-bucket prior to sending the 200.

### 6.4. Kademlia query processing

If the target node is the receiving node itself, it MUST reply with a 200 message. Otherwise, the receiving peer MUST provide the contacts of the k closest nodes to the target id in the 302 message. These MUST be placed in Contact headers. If the receiving node knows less than k nodes it must report all the nodes it knows about. Additionally, the replying peer MUST include its DHT-PeerID header.

### 6.5. Kademlia Graceful Leaving

Peers MUST unregister themselves. This unregister is constructed exactly the same as the Peer Registration message used to join, with the following exceptions. The expires parameter or header MUST be provided, and MUST be set to 0.

### 6.6. DHT Maintenance

No operations are needed in order to keep the overlay stable, as the Kademlia algorithm is designed to update routing table and resource storing information anytime a message is received. Therefore, the Kademlia DHT needs no Chord-like stabilization procedure.

### 6.7. Peer Failure

Peer failure is discovered through timed-out requests. Redundancy prevents against lost registrations.

### 6.8. Resource Replicas

When a resource is registered, the registering peer MUST create at least 2 redundant replicas to ensure the registry information is
secure in the DHT. The registering peer is responsible for maintaining these replicas along with the primary entry. Moreover, Kademia provides an inner replication mechanism by storing a registration in the k closest nodes to the resource’s id.
7. Examples

Instead of the SHA-1 algorithm, which would create a $2^{160}$ nodes network, a simpler network is used. The hash is a 4 bit hash, which yields a namespace of size 16. Moreover, $k$ (the maximum size of the k-buckets) is set to 4.

Assume, for the sake of example simplicity, assume the peer Peer-ID 3 has IP address 192.0.2.3, the peer with Peer-ID 5 has IP address 192.0.2.5, and so on. Further assume that peer 1, peer 3, peer 7, peer 10, and peer 12 are currently enrolled in the overlay. In the case that each peer has been previously contacted by all other peer in the overlay, the k-buckets for each peer are shown in Figure 4.

<table>
<thead>
<tr>
<th>k-bucket</th>
<th>Peer1</th>
<th>Peer3</th>
<th>Peer7</th>
<th>Peer10</th>
<th>Peer12</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>K1</td>
<td>Peer3</td>
<td>Peer1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>K2</td>
<td>Peer7</td>
<td>Peer7</td>
<td>Peer1</td>
<td>Peer12</td>
<td>Peer10</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>Peer3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>K3</td>
<td>Peer10</td>
<td>Peer10</td>
<td>Peer10</td>
<td>Peer1</td>
<td>Peer1</td>
</tr>
<tr>
<td>Peer12</td>
<td>-</td>
<td>-</td>
<td>Peer12</td>
<td>Peer3</td>
<td>Peer3</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Peer7</td>
<td>Peer7</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4

7.1. Peer registration

Assume peer 5, running on IP 192.0.2.5, port 5060, wants to join the overlay. From an out of band mechanism, peer 5 discovers peer 10 and uses it as a bootstrap. Peer 5 constructs a peer registration request and sends it to peer 10. Peer 10 verifies that the request is valid and executes the following steps:
1. updates its k-buckets: the sending peer’s id is 5, that is 0101; the distance between peer 10 and peer 5 is therefore 1010 XOR 0101 = 1111 = 15; the contact of peer 5 should then be stored in the k-bucket of index 3 as \(d(1010, 0101)\) belongs to the \([2^3, 2^4)\) interval;

2. replies with a 200 OK response.

At this point, peer 5 has joined the overlay and the k-buckets of the peers are shown in Figure 5.

<table>
<thead>
<tr>
<th>k-bucket</th>
<th>Peer1</th>
<th>Peer3</th>
<th>Peer5</th>
<th>Peer7</th>
<th>Peer10</th>
<th>Peer12</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>K1</td>
<td>Peer3</td>
<td>Peer1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>K2</td>
<td>Peer7</td>
<td>Peer7</td>
<td>-</td>
<td>Peer1</td>
<td>Peer12</td>
<td>Peer10</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Peer3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>K3</td>
<td>Peer10</td>
<td>Peer10</td>
<td>Peer10</td>
<td>Peer10</td>
<td>Peer1</td>
<td>Peer1</td>
</tr>
<tr>
<td>Peer12</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Peer7</td>
<td>Peer7</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Figure 5**

The registration process completes after peer 5 populates its routing table by performing a node lookup targeting its own id. This process is described in Section 7.2. The message flow for the registration is shown in Figure 6.
7.2. Peer query

After joining the overlay, peer 5 needs to populate its routing table. To do so, it performs a node lookup targeting its own id. As peer 10 is the only contact in its routing table, peer 5 sends a peer query to it and sets the closest contacted node to NULL. Peer 10 receives the peer query, updates its k-buckets as a new message was received, and replies with a 302 response, reporting the k closest nodes to the target (peer 5) that is currently aware of. These k contacts will be, in order of proximity to the target node: peer 7,
peer 1, peer 3, and peer 12. Peer 5 receives the reply from peer 10 and retrieves all the contacts from the Contact header. Peer 5 first updates its k-buckets and then sets peer 10 as the closest contacted peer to the target at a distance of 15. Now peer 5 selects alpha = 3 contacts from the k contacts it has received by peer 10 and sends asynchronous peer queries to these contacts. Peer 5 chooses peer 7, peer 1, and peer 3. All these peers receive the request from peer 5, and therefore update their routing table by adding peer 5’s contact. The peers then reply with a 302 response reporting the closest nodes to peer 5 they are aware of in the Contact header. When peer 5 receives the responses from each peer it updates its k-buckets and retrieves the contacts reported in the responses. After receiving all the replies, or after they have timed out, peer 5 updates the closest node contacted to be peer 7 at a distance of 2. Now peer 5 need to contact peer 12, as it is the only peer it has not yet queried. Peer 12 receives the request, updates its k-buckets, and replies with 302 response reporting the closest nodes it knows. Peer 5 receives the reply, updates its k-buckets, but it does not update its closest contacted node. At this point, since the round of queries has not yielded an improvement in proximity to the target, the peer query process terminates and peer 5 has populated its routing table. The k-buckets of the peers at this point are shown in Figure 7.

<table>
<thead>
<tr>
<th>k-bucket</th>
<th>Peer1</th>
<th>Peer3</th>
<th>Peer5</th>
<th>Peer7</th>
<th>Peer10</th>
<th>Peer12</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>K1</td>
<td>Peer3</td>
<td>Peer1</td>
<td>Peer7</td>
<td>Peer5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>K2</td>
<td>Peer7</td>
<td>Peer7</td>
<td>Peer1</td>
<td>Peer1</td>
<td>Peer12</td>
<td>Peer10</td>
</tr>
<tr>
<td></td>
<td>Peer5</td>
<td>Peer5</td>
<td>Peer3</td>
<td>Peer3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>K3</td>
<td>Peer10</td>
<td>Peer10</td>
<td>Peer10</td>
<td>Peer10</td>
<td>Peer1</td>
<td>Peer1</td>
</tr>
<tr>
<td></td>
<td>Peer12</td>
<td>Peer12</td>
<td>Peer12</td>
<td>Peer12</td>
<td>Peer3</td>
<td>Peer3</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Peer7</td>
<td>Peer7</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Peer5</td>
<td>Peer5</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
The message flow for the registration is shown in Figure 8.

Peer5 -> Peer10
REGISTER sip:192.0.2.10 SIP/2.0
To: <sip:peer@0.0.0.0;peer-ID=5>
From: <sip:peer@192.0.2.5;peer-ID=5>
Contact: <sip:peer@192.0.2.5;peer-ID=5>
Expires: 600
DHT-PeerID: <sip:peer@192.0.2.5;peer-ID=5>;algorithm=sha1;
Figure 8

7.3. User registration

Assume user Carl starts a UA co-located with peer 5. Carl’s contact is carl@192.0.2.5 and his user name is carl@p2psip.org. Peer 5 hashes Carl’s user name and determines that the corresponding resource-ID is 11. The registration proceeds through the following steps:

1. peer 5 performs a node lookup targeting id 11;
2. when the lookup has terminated, peer 5 has discovered the k peers closest to the id 11; in this case the lookup for id 11 results in peer 10, peer 12, peer 3, and peer 1;
3. peer 5 sends a resource registration request to each of the discovered nodes;
4. peers receive the registration request, store the registration, and reply with a 200 OK response.

The dSIP messages for a user registration are (we show only one registration request and its relative response) shown in Figure 9.
After the user registration has completed, the registration for Carl is stored by peers 10, 12, 3, and 1. This approach has therefore a built-in replication of registrations.

7.4. Session establishment

Assume user Carl wishes to call user Simon. Simon is co-located with peer 10 and Simon’s user name (simon@p2psip.org) hashes to 9. Further assume that the registration for Alice is stored by peer 10,
12, 1, and 3. Peer 5 performs a resource query for id 4. First, peer 5 finds k nodes in its k-buckets that are closest to the target id 4. It then selects of these nodes and sends a resource query request to each of these. Assume that peer 5’s routing table does not contain peer 12. The k = 4 closest nodes to the target id in peer 5’s k- buckets are peer 10, peer 1, peer 3, and peer 7. Since Kademlia does not specify any criteria for choosing the alpha nodes to contacts, suppose peer 5 decides to send the request to peer 7, peer 3, and peer 10. Peer 7 replies with a 302 response indicating the contacts in its routing table that are closest to the resource id, which would be peer 10, peer 12, peer 1, and peer 3. Peer 3, instead, replies with a 200 OK response, as it is currently storing the registration for Simon. When peer 5 receives the 200 OK response, the resource query is completed and the call can complete by Carl sending an INVITE request to Simon. The dSIP message flow for the session establishment is shown in Figure 10.
Peer5 -> Peer7; Peer5 -> Peer3; Peer5 -> Peer10;

REGISTER sip:192.0.2.7 SIP/2.0
To: <sip:simon@p2psip.org;resource-ID=9>
From: <sip:carl@p2psip.org;resource-ID=11>
DHT-PeerID: <sip:peer@192.0.2.5;peer-ID=5>;algorithm=sha1;
   dht=Kademlia1.0;overlay=p2psip;expires=800
Require: dht
Supported: dht

REGISTER sip:192.0.2.3 SIP/2.0
To: <sip:simon@p2psip.org;resource-ID=9>
From: <sip:carl@p2psip.org;resource-ID=11>
DHT-PeerID: <sip:peer@192.0.2.5;peer-ID=5>;algorithm=sha1;
   dht=Kademlia1.0;overlay=p2psip;expires=800
Require: dht
Supported: dht

REGISTER sip:192.0.2.10 SIP/2.0
To: <sip:simon@p2psip.org;resource-ID=9>
From: <sip:carl@p2psip.org;resource-ID=11>
DHT-PeerID: <sip:peer@192.0.2.5;peer-ID=5>;algorithm=sha1;
   dht=Kademlia1.0;overlay=p2psip;expires=800
Require: dht
Supported: dht

Peer7 -> Peer5

SIP/2.0 302 Moved Temporarily
To: <sip:simon@p2psip.org;resource-ID=9>
From: <sip:carl@p2psip.org;resource-ID=11>
Contact: <sip:peer@192.0.2.10;peer-ID=10>,
   <sip:peer@192.0.2.12;peer-ID=12>,
   <sip:peer@192.0.2.1;peer-ID=1>,
   <sip:peer@192.0.2.3;peer-ID=3>
DHT-PeerID: <sip:peer@192.0.2.7;peer-ID=7>;algorithm=sha1;
   dht=Kademlia1.0;overlay=p2psip;expires=600
Require: dht
Supported: dht

Peer10 -> Peer5
SIP/2.0 200 OK
To: <sip:simon@p2psip.org;resource-ID=9>
From: <sip:carl@p2psip.org;resource-ID=11>
Contact: <sip:simone@192.0.2.10>
DHT-PeerID: <sip:peer@192.0.2.3;peer-ID=3>;algorithm=sha1;
dht=Kademlia1.0;overlay=p2psip;expires=1200
Require: dht
Supported: dht

Figure 10
8. Implementation

The dSIP system has already been successfully implemented. The implementation includes support for the Chord DHT, as required in the old dSIP draft [I-D.bryan-p2psip-dsip], and for the Kademlia DHT, following the outline sketched in this document.

The implementation relies on the mjSip library [mjSip], developed at the University of Parma (Italy), which is a complete Java-based open source implementation of a SIP stack, and available under the terms of the GNU GPL license as published by the Free Software Foundation. mjSip implements the complete layered stack architecture as defined in [RFC3261] (Transport, Transaction, and Dialog sublayers), and is fully compliant with the standard. Moreover, it includes higher level interfaces for Call Control and User Agent implementations.

The complete source code for the implementation of the dSIP architecture, including support for the Chord and Kademlia DHTs, is available at the following location: <http://www.mjsip.org/projects/p2psip/p2psip_dsip_071025.zip>. Other DHTs can be easily implemented by extending the currently available API.
9. Security Considerations

[TO BE INVESTIGATED]
10. IANA Considerations

This document defines the "dht-param" value to be "Kademlia.0".
11. References

11.1. Normative References

[I-D.bryan-p2psip-dsip]

[I-D.bryan-p2psip-reload]
Bryan, D., "REsource LOcation And Discovery (RELOAD)", draft-bryan-p2psip-reload-01 (work in progress), July 2007.

[I-D.lowekamp-p2psip-dsip-security]

[I-D.willis-p2psip-concepts]

[I-D.zangrilli-p2psip-dsip-dhtchord]


11.2. Informative References


[mjSip] Veltri, L., "mjSIP".
Authors’ Addresses

Simone Cirani
Department of Information Engineering - University of Parma
Parco Area delle Scienze, 181/A
Parma 43100
Italy

Email: simone.cirani@gmail.com

Luca Veltri
Department of Information Engineering - University of Parma
Parco Area delle Scienze, 181/A
Parma 43100
Italy

Email: luca.veltri@unipr.it
URI: http://www.mjsip.org