SIPPEER: A Session Initiation Protocol (SIP)-based Peer-to-Peer Internet Telephony Client Adaptor*

Kundan Singh and Henning Schulzrinne
Department of Computer Science, Columbia University
{kns10,hgs}@cs.columbia.edu

Abstract

Our architecture for peer-to-peer Internet telephony using the Session Initiation Protocol (SIP) supports basic user registration and call setup as well as advanced services such as offline message delivery, voice/video mails and multi-party conferencing. In this paper, we describe the design and implementation of our P2P-SIP client adaptor, SIPPEER, using pseudo code and example messages. We also explain how to use existing SIP user agents such as X-lite and Cisco IP phone with SIPPEER to participate in P2P network.

Contents

1 Introduction **3**

2 Background **4**
2.1 SIP-based telephony . . . 4
2.2 Chord: a structured P2P algorithm . . . 4

3 Architecture **5**
3.1 SIP-layer . . . 5
3.2 Data structures and classes . . . 6
3.3 DHT module . . . 7
3.3.1 Initialization . . . 8
3.3.2 Peer discovery . . . 8
3.3.3 Joining the DHT . . . 9
3.3.4 Stabilization . . . 10
3.3.5 Node shutdown (graceful termination) . . . 13
3.3.6 Node failure and failover . . . 14
3.4 User registration . . . 16
3.4.1 Registration handling . . . 16
3.4.2 Node shutdown (graceful termination) . . . 18
3.4.3 Node failure and failover . . . 18
3.5 Call setup and message proxy . . . 18
3.5.1 Message proxy . . . 19
3.5.2 Multimedia call setup and instant messages . . . 19

4 Experiments **20**
4.1 Adaptor for existing SIP phones . . . 20
4.1.1 Columbia sipc . . . 21
4.1.2 Cisco IP phone . . . 21
4.1.3 X-lite . . . 22
4.1.4 Microsoft Windows Messenger . . . 23
4.2 Interoperability issues . . . 23

*The work is supported by grant from SIPquest Inc., and the National Science Foundation under Grant No. 0202063.

5 Security **25**
5.1 Threats . . . 25
5.2 Identity protection . . . 25
5.3 Misbehaving nodes . . . 26
5.4 Data privacy . . . 26
5.5 Programmable call routing . . . 26
5.6 User aliases . . . 27

6 Advanced services **28**
6.1 Presence and event notification . . . 28
6.2 PSTN interworking . . . 29
6.2.1 Registering a telephone prefix . . . 30
6.2.2 Calling a telephone number . . . 30
6.2.3 Incoming INVITE handling . . . 31
6.3 Firewall and NAT traversal . . . 31
6.3.1 Signaling . . . 32
6.3.2 Media . . . 32
6.3.3 Integrating media and signaling . . . 32
6.4 Inter-domain operation . . . 32
6.4.1 Registration . . . 33
6.4.2 Call setup . . . 34
6.4.3 Cross-domain . . . 34

7 Conclusions and future work **35**

A Message flow **36**

1 Introduction

Peer-to-peer (P2P) systems inherently have high scalability, robustness and fault tolerance because there is no centralized server and the network self-organizes itself [12]. This is achieved at the cost of higher latency for locating the resources of interest in the P2P overlay network. We have proposed a Session Initiation Protocol (SIP [20, 25])-based peer-to-peer Internet telephony (P2P-SIP) architecture [26, 27]. It uses distributed, robust and scalable key lookup of distributed hash table (DHT)-based structured P2P algorithms such as Chord [29]. It uses SIP as the underlying protocol so that it interoperates with existing infrastructure such as SIP-PSTN gateways.

We have implemented a P2P-SIP adaptor, SIPPEER, that allows existing or new SIP user agents to connect to the P2P-SIP network without modifying the user agent. For example, SIPPEER can run on the same host as the PC-based SIP user agent, sipc, and act as the outbound proxy for sipc. SIPPEER can also act as a standalone SIP user agent, proxy or registration server with command line user interface.

We describe the design and implementation of SIPPEER for basic user registration and call setup using pseudo-code and example messages. In particular, we answer the following questions: (1) how to implement a P2P-SIP node for basic functionality? (2) how to use a P2P-SIP node for existing user agents? and (3) how to extend it for advanced features such as NAT or firewall traversal and PSTN interworking.

This implementation report can be used to understand the SIPPEER implementation and to design similar P2P-SIP systems. This document should be used in conjunction with [27]. Our modular design allows reusable and replaceable components. For example, in future Chord may be replaced by another distributed hash table (DHT) without affecting the rest of the implementation. The open architecture allows installing new services without affecting the existing design. For example, a new voice mail module can be added to the existing node. Finally, we discuss the security aspects and advanced services such as firewall and NAT traversal, and PSTN interworking in the context of SIPPEER.

Section 2 gives an overview of SIP and Chord. Section 3 describes the detailed design. Some experiments on using SIPPEER with existing clients are in Section 4. Some security issues are discussed in Section 5. Section 6 provides guidelines to extend SIPPEER for advanced services.

In this section, we review the Chord algorithm and SIP-based telephony concepts needed for understanding SIPPEER's design. Readers familiar with these can skip the section.

2.1 SIP-based telephony

Existing SIP-based telephony has client-server architecture. SIP [20] is a signaling protocol for Internet conferencing, telephony, presence, event notification and instant messaging. As shown in Fig. 1, when a user, Bob, starts the SIP client on his PC, IP-phone or hand-held device, the client registers with the SIP server (or registrar) indicating the IP address of the device. The SIP server stores the mapping between the user identifier bob@home.com and the IP address or host name. When another user, Alice, makes a call or sends instant message for bob@home.com to the server in home.com domain, the server proxies the request to the current device of Bob. Further details of the protocol can be found in [20, 25].

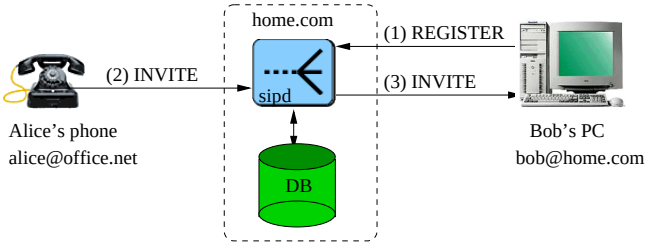

Figure 1: SIP call flow using proxy servers

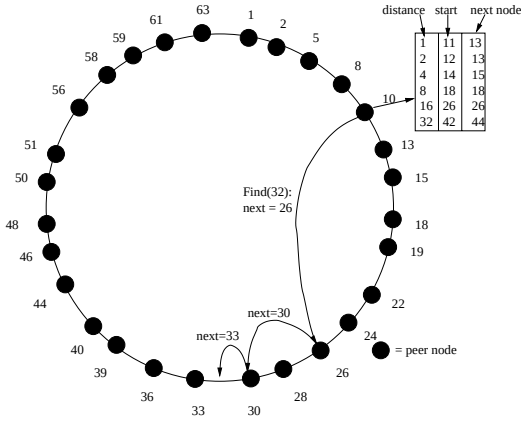

Figure 2: Example Chord network

2.2 Chord: a structured P2P algorithm

Structured P2P algorithms such as Chord [29] focuses on optimizing the P2P overlay for lookup latency and join or leave maintenance cost [10] instead of using inefficient blind search by flooding. We use Chord as the underlying distributed hash table (DHT) in our architecture for lookup. Chord has a ring-based topology where each node stores at most $log(N)$ entries (or state) in its *finger table* to point to other peers. Lookup is done in $O(log(N))$ time.

An example Chord network with six bits identifiers is shown in Fig. 2. The identifier range is [0-63]. Every node maintains a finger table of $log(N) = 6$ entries pointing to the next-hop node location at distance 2^{i-1} (for i=1,2,..,6) from this node identifier. Node 10's finger table is shown in Fig. 2. When node 10 wants to Find key 32, it looks up the finger table to find the closest match, and sends the query to node 26. Node 26 in turn sends it to node 30, which finally sends it to node 33. Node 33 is the successor of identifier 32, hence responsible for storing information about key 32. At each step the distance to destination is reduced by approximately half, resulting in $O(log(N))$ lookup latency, if there are N nodes in the ring.

The *iterative* and *recursive* lookup styles in Chord [29] directly map to the *redirect* and *proxy* behavior, respectively, in SIP. Research in DHT is complementary to our work, since our architecture can use new innovations or optimizations in the underlying DHT.

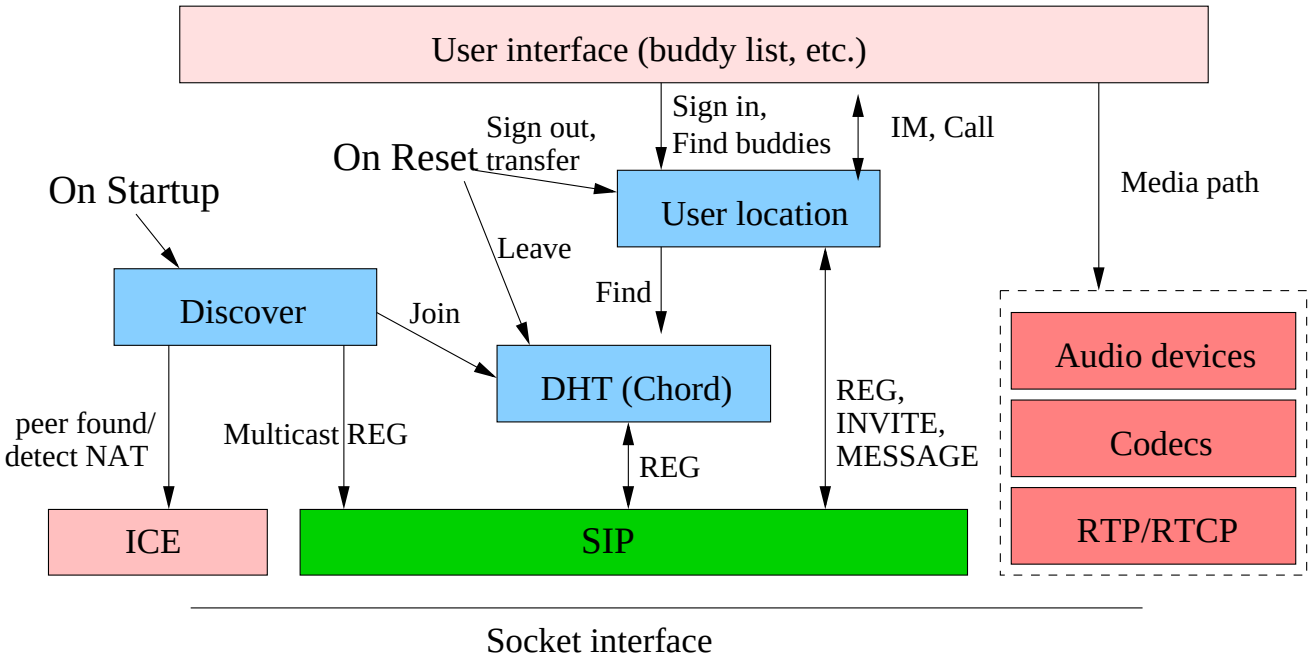

Figure 3: Block diagram of a P2P-SIP node

Fig. 3 shows the block diagram of the different components in SIPPEER. When the node starts up and the user signs-in with her identifier, and the discover module is activated to initiate NAT and firewall detection, peer discovery and SIP registration. Multicast SIP registration, cached peer addresses from last boot cycle and pre-configured bootstrap addresses are used to discover initial set of nodes. The user interface module keeps track of user's "friends list" and invokes the user location module to locate these friends. User location is obtained using the SIP module or, if this node is a super-node, the DHT module.

The node architecture can be logically divided into two parts: the DHT maintenance and the user account maintenance. The DHT module maintains the peer information (e.g., Chord *finger table*) and performs DHT operations such as find, join and leave. It provides the underlying topology for communication. The user account maintenance module deals with maintaining local user accounts as well as storing remote user registrations. It acts as a SIP registrar and proxy server.

3.1 SIP-layer

SIP is used as the underlying protocol for maintaining the DHT, locating another user, registering the user, call setup and instant messaging. The SIP REGISTER message is used in two context by the node: query and update. If a Contact header is present in the message, then it is an update request indicating that the sender wants to update the bindings for the node identifier in the To header. Otherwise, it is a query request, where the sender is requesting to get the Contact information of the node identifier in the To header. Initial discovery uses the REGISTER message for query. This behavior is semantically same as that defined by SIP.

Once the user location is done, the call setup or instant messages can be sent directly to the user's phone. SIP REGISTER refresh and OPTIONS messages are used to detect node failure. When a super-node shuts down, the registrations are transferred to other super-nodes in the DHT as appropriate. Other SIP functions such as third-party-call control and call-transfer can be implemented in the similar way. The media path (audio device, codecs and transport) is largely independent of the P2P-SIP operation.

Node and user identifiers are represented using SIP URI. For example, if a node is listening at transport address 128.59.15.32:8054 for SIP messages and the Chord's hash function gives the node identifier as H(128.59.15.32) = 17, then the node's URI becomes sip:17@128.59.15.32:8054. User Alice can register her identifier as sip:alice@example.com.

We use our C++ SIP library, libsip++ [2], that provides class SIPEndpoint representing the end system and SIPDialog representing a SIP dialog between two end systems. SIPCall and SIPOutgoingRegister are derived from SIPDialog to implement the call control and registration refresh state machines. SIPSession extends SIPCall to implement advanced features such as event notification and call transfer whereas SIPSessionEndpoint extends SIPEndpoint to support SIPSession.

3.2 Data structures and classes

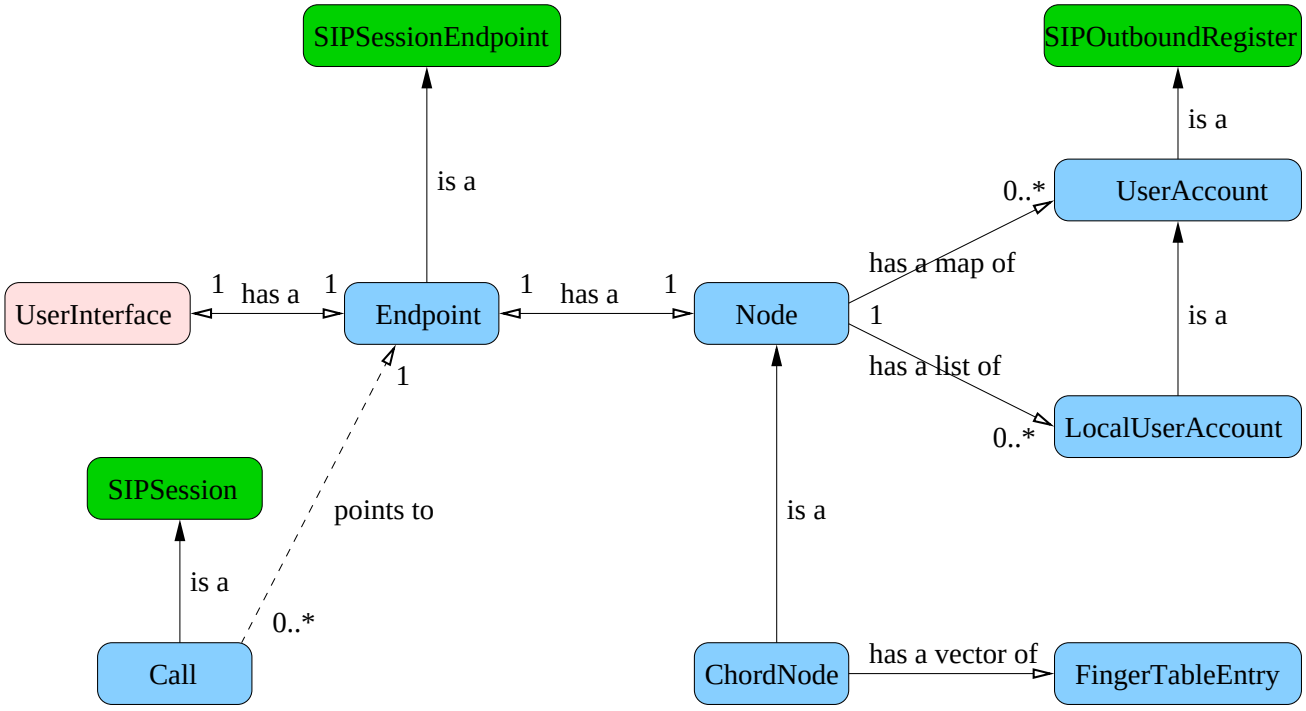

Figure 4: Class diagram in SIPpeer

Fig. 4 shows the class diagram including the relationship among different classes. The Endpoint class, derived from libsip++'s SIPSessionEndpoint, represents the P2P-SIP node. It is associated with one DHT node implementation, represented by the Node class. It gets all events such as the join operation is complete, or the node information changed (such as change in finger table or predecessor), or when this node is leaving the DHT. Endpoint also stores a list of LocalUserAccount objects for outgoing registrations from the local node. These registrations represent the user accounts activated from the local node. It also stores a map table of all the registered UserAccount objects indexed by the remote user identifier.

A UserAccount object represents a user registration (typically of a remote user) derived from SIPOutboundRegister class. It invokes the associated Node's callback methods when a new registration is received or is refreshed. A LocalUserAccount class represents a local user's registration.

A Node object represents a single node (this node) in the DHT. The ChordNode class derived from the Node class, implements the Chord algorithm. Various methods such as Find, Join, Leave are defined as virtual in Node, and implemented in ChordNode. Node class implements the super-node and ordinary-node distinction.

A ChordNode internally maintains other Chord related data structures such as finger table, successor list, and predecessor. The FingerTableEntry represents a single entry in the finger table consisting of the start and end identifiers in the ring, and the node location for the next hop node corresponding to this identifier range. The finger table is a map table of FingerTableEntry with integer index. The successor list is a list of node locations. Predecessor is a single node location.

The Identifier class represents a node or user identifier. The implementation is internal, typically using as hash function. The object stores the raw value such as 10@sippeer.net or alice@yahoo.com as well as the hashed value such as 10 and 24. The current implementation assumes number@host as the node identifier whereas anything else as the user identifier. The SIP requests (typically REGISTER) with To: header containing this node's identifier is handled by the ChordNode

module, where as those for the user identifier are handled by Endpoint, UserAccount or LocalUserAccount objects. A NodeLocation class represents the node identifier as well as its contact location (a SIP URI).

The Call object is similar to the SIPSession object except that it does lookup using both DNS (NAPTR and SRV [18]) as well as P2P overlay. The UserInterface object represents the command line user interface, which can later be extended to graphical UI.

Using various objects the Endpoint implements a SIP user agent, proxy as well as registrar. No external database is used. All state including registrations are maintained in the internal memory.

3.3 DHT module

The Node class represents a single node in the DHT. The actual implementation is done in a sub-class. For example, ChordNode implements the Chord algorithm. Alternatively, other DHT algorithms such as CAN can be implemented as a sub-class of Node. The Endpoint object creates this sub-class (e.g., ChordNode) object on startup.

The DHT module takes care of implementing three abstract methods: Join, Leave and Find. When the node starts up it needs to discover at least one other node in the DHT. Then, it joins the DHT through that node. When the node is gracefully shutting down, it leaves the DHT. Higher layer application such as user account maintenance module uses the Find method to locate the next hop node to send user registration or proxy other SIP messages for call setup or instant messaging.

This Section describes the implementation of ChordNode. We illustrate with simple examples of 5 bit identifiers. We represent the node N's identifier as N_{id}, transport address (IP and port number) as N_{addr}, IP address as N_{ip}, predecessor as N_{pred}, successor as N_{succ}, finger table entry of this node for index i as F^i, and corresponding start, end and next hop node URI as F^i_{start}, F^i_{end} and F^i_{node} respectively. The successor list is represented as N^{list}_{succ}, and i^{th} successor as N^i_{succ}. Note that N^0_{succ} is same as N_{succ}. Finger table entry for another node N is denoted as $N :: F^i$. Note that N_{succ} is same as $N :: F^1_{node}$.

Procedure 3.1 N.set-fingers (i:start index, $node$:node location)

```
/* Set the finger table entries starting at i */
/* Returns the index of the last finger entry that gets set. */
F^i_node ⇐ node
while i ≤ m − 1 do
  if F^{i+1}_start ∈ [N_id, F^i_node] then
    F^{i+1} ⇐ F^i_node
  else
    return i
  i ⇐ i + 1
return i
```

Pseudo code to set the finger table entry and to query the closest preceding finger are described in procedures 3.1 and 3.2, respectively. If F^i_{node} is set to $node$, then all subsequent F^j_{node} are also set to $node$ as long as F^j_{start} is before $node$ in the Chord ring. To find the closest preceding finger for a given key, the finger table is scanned in the reverse order and the alive node with highest identifier preceding the key is returned. Lookup is also done in the successor list, N^{list}_{succ}.

Procedure 3.2 N.closest-preceding-finger (key)

```
/* Find the closest preceding finger for the key */
node ⇐ N_id
for i ← m down to 1 do
  if F^i_node is alive and F^i_node ∈ (N_id, key) then
    node ⇐ F^i_node
    break
for all s in N^list_succ do
  if s is alive and s ∈ (N_id, node) then
    node ⇐ s
    break
return node
```

3.3.1 Initialization

When the node starts up, it creates a new Endpoint object and associated UserInterface and ChordNode objects. It allocates any available port for receiving SIP messages on TCP and UDP. The SIPPEER application can accept -p command line option to configure a fixed receiving port number, instead of using any available port. Typically there are three listening threads, (1) for TCP on INADDR_ANY interface and some port p, (2) for UDP unicast on INADDR_ANY interface and same port p, and (3) for UDP multicast on address 224.0.1.75 and port 5060. To allow receiving both multicast and unicast packets on ports p and 5060, threads (2) and (3) bind to multicast as well as unicast addresses. If p=5060, then thread (3) is not created. The UDP sockets are bound non-exclusively on port 5060, so that multiple instances of the node can run on the same host.

If the node is started with a pre-configured user identifier, it also creates a new LocalUserAccount object for this user identifier. For example, when the SIPPEER application is used as a SIP user agent, the user can specify her identifier, e.g., alice@example.com, on start up.

The ChordNode calculates its node Identifier using the IP address of the local interface of the Endpoint. For testing purpose we use both IP and port, so that we can start nodes with different node identifiers on the same host. However, in practice only the IP address should be used. This prevents a single IP address from disturbing random parts in the DHT on frequent join and leave by restarting the application. **TODO: Currently we use a simple hash function to generate up to 32 bit identifier. This needs to be changed to use SHA1 to generate 160 bit identifier as per the Chord specification.**

3.3.2 Peer discovery

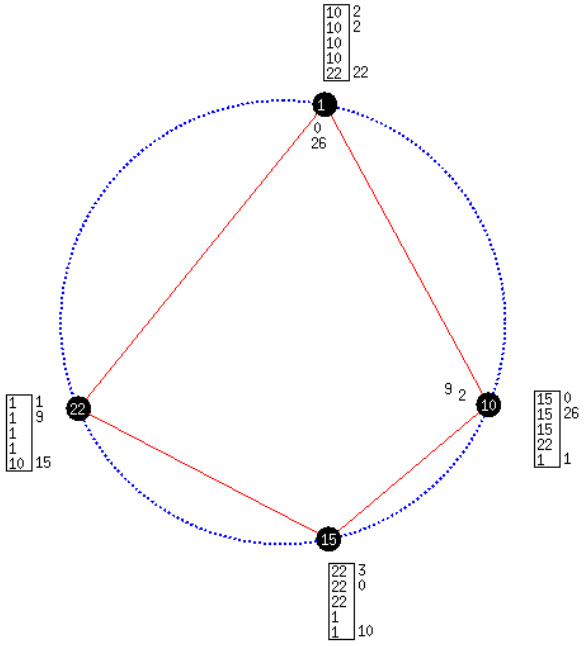

Figure 5: Example Chord network with 4 nodes

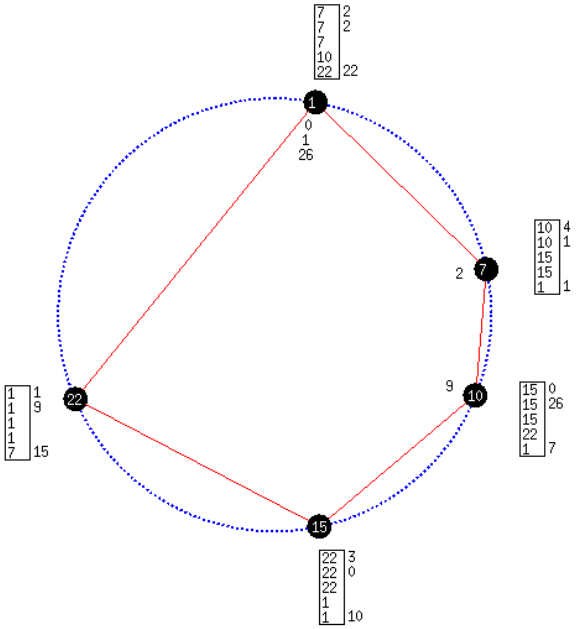

Figure 6: After node 7 joins the network

Consider an example four nodes Chord network as shown in Fig. 5. The node identifiers are 10, 22, 1 and 15, and the node transport addresses are 128.59.15.55:, 128.59.15.31:, 128.59.15.60: and 128.59.15.48:, respectively. When a new node, 7, (with transport address 7_{addr}=128.59.15.56:44452) starts up, it invokes its Discover method (procedure 3.3) to discover possible peers.

The Discover method of node 7 sends a SIP REGISTER message with request-URI as sip:224.0.1.75 (SIP REGISTER multicast IPv4 address) and the To header as the local node identifier, 7_{id}. The From header is always the local node identifier, 7_{id}, if the request is generated by the ChordNode module. (The mandatory SIP headers that are not needed for understanding P2P-SIP are not shown, but MUST be sent as per SIP specification. See Appendix A for details.)

```
REGISTER sip:224.0.1.75 SIP/2.0
To: <sip:7@128.59.15.56>
From: <sip:7@128.59.15.56>
```

Procedure 3.3 N.Discover

if discovery is allowed **then**
  send REGISTER sip:224.0.1.75
    To: N_{id}
else
  for $i := 1$ to m **do**
    $F^i_{node} \Leftarrow N_{id}$
  $N_{pred} \Leftarrow N_{id}$
  trigger join complete event

If the application is started with -N option to suppress node discovery, the node state is initialized to reflect a singleton node in the DHT. In that case, all finger table entries and predecessor in the ChordNode point to this node's location.

If some other node, say node 22, receives the multicast REGISTER request, and is already part of the DHT, it responds with its own unicast address, 22_{addr}=128.59.15.31, in the SIP 302 redirection response.

```
SIP/2.0 302 Redirect to unicast
Contact: <sip:128.59.15.31>
```

If the node receives multiple final responses, it can choose which one to use. Our implementation uses the first received response. If the node does not get any response within a timeout (we use about 30 seconds), it uses other means of discovery. Following possibilities exist but are not yet implemented:

Service discovery: The node can have a service location protocol (SLP [7]) user agent (UA), that discovers other nodes in the domain. Once the node joins the DHT it should register with the SLP directory agent so it other nodes can discover this node. For the Internet, some wide-area service discovery protocol is more suitable.

Bootstrap nodes: The node can be pre-configured with a set of IP addresses or domain names to probe for possible peers. For example, the node can query DNS for sippeer.net domain's SIP servers and send the initial REGISTER message to them. At least one of the the initial bootstrap P2P-SIP nodes is assumed to be active for this scheme to work. This may introduce the centralized component, but is limited in scope only to the initial bootstrap process. Once the node starts up it caches other peers addresses for subsequent reboots.

If the node can not discover any other peer, it assumes that it is the first node in the DHT and initializes its data structures (Chord finger table and predecessor location) accordingly. It also re-schedules the discovery procedure for a later time, say after five minutes.

3.3.3 Joining the DHT

Once other peer(s) are discovered, the node selects one and sends a SIP REGISTER message to its unicast address. For example, node 7 sends the following message to 22_{addr}=128.59.15.31.

```
REGISTER sip:128.59.15.31 SIP/2.0
To: <sip:7@128.59.15.56>
From: <sip:7@128.59.15.56>
```

When node 22 receives the REGISTER on its unicast address, it extracts the *destination key*, 7, from the To header. Depending on the destination key value, k, there are three cases for node N to process the request (procedure 3.4): (1) if $k \in (N_{pred}, N]$, then node N is responsible for storing k, (2) if $k \in (N, N_{succ}]$, then node N_{succ} is responsible for storing k, otherwise (3) some other node is responsible for storing k. For case (1), node N responds with a SIP 200 success response containing the Contact header as N_{id} and the predecessor parameter as N_{pred}. If the key, k, is same as N (subset of case (1)), but the addresses are different (e.g., two nodes happen to have the same hash value for the node identifier), then a global failure (SIP 600 response) is returned, with the Contact header as N_{id}. For case (2), it responds with a SIP 200 success response containing the Contact header as N_{succ} and the predecessor parameter as N_{id}. For case (3), it proxies the request to the next hop node based on the finger table $N :: F$. Eventually the request reaches the node responsible for k, which can respond back with the correct Contact header.

In our example, key 7 does not belong to node 22 or successor 1 (this is case (3)), so the finger table is used to find the next hop node. Since the largest i for which $N + 2^{i-1} \leq k$, where N=22 and k=7, is $i = 4$, the next hop is $22 :: F^4_{node}$=1_{addr}, hence the request is proxied to 1_{addr}. Node 1 decides that key k belongs to the successor node 10, (case (2) because $7 \in (1, 10]$) and responds with the success response containing Contact as 10_{addr}. Node 22 forwards this response back to node 7.

Procedure 3.4 N.OnRegister (R:registration object, M:request message)

if join is not complete **then**
  ignore M
else if M is a query, i.e., M.Contact is empty **then**
  $to \Leftarrow M$.To.user
  if $to \neq N_{id}$ **and** $to \notin (N_{id}, N_{succ}]$ **then**
    $node \Leftarrow$ closest-preceding-finger(k) /* *procedure 3.2* */
  else if to equals N_{succ}'s id, but has different address **then**
    $node \Leftarrow N_{succ}$
  else
    $node \Leftarrow N_{id}$
  if $node$ = N_{id} **then**
    if $to = N$ **then**
      send response 200 OK
        Contact: N_{id}; predecessor=N_{pred}
    else
      send response 200 OK
        Contact: N_{succ}; predecessor=N_{id}
  else
    proxy M **to** $node$

```
SIP/2.0 200 OK
To: <sip:7@128.59.15.56>
Contact: <sip:10@128.59.15.55>; predecessor=sip:1@128.59.15.60
```

When the discovering node, 7, receives the SIP 200 response with To header as 7_{id}, it updates its finger table with the successor node locations and goes on to find remaining nodes in the finger table (procedure 3.5). For example if the To header in the response is this node identifier, 7_{id}, and the successor for this node 7_{succ} is empty, then the successor is set to be the Contact header in the response, $7_{succ} := 10$. Now, $10 \geq 7 + 2^{i-1}$ for i=1 and 2, so node 7 updates its finger table $F^1_{node} := F^2_{node} := 10_{id}$. The next unassigned finger entry for index i=3 needs to be discovered. Node 7 sends a SIP REGISTER message for F^3_{start}=11. The domain sippeer.net is used as logical domain for node 11_{id} to indicate that the IP address of this key 11 is not known.

```
REGISTER sip:128.59.15.55 SIP/2.0
To: <sip:11@sippeer.net>
From: <sip:7@128.59.15.56>
```

Eventually node 7 receives the response for this registration, indicating that node 15 is responsible for key 11:

```
SIP/2.0 200 OK
To: <sip:11@sippeer.net>
Contact: <sip:15@128.58.15.48>; predecessor=sip:10@128.59.15.55
```

When the node gets the SIP 200 response for this REGISTER, it realizes that the To header corresponds to F^3_{start}, and updates the finger table based on the Contact header of the response, $F^3_{node} := 15_{id}$. Since $15 \geq F^4_{start}$, it updates $F^4_{node} := 15_{id}$. Finally, node 7 sends another SIP REGISTER message to discover node $F^5_{start} = 23$, and updates the finger table on response as $F^5_{node} := 1_{id}$.

3.3.4 Stabilization

At this point, node 7 realizes that all the finger table entries are filled, so it tries to stabilize the Chord DHT, if the Join procedure is not yet complete. To initiate the stabilization process, it sends a SIP REGISTER message to N_{succ}, sets the To header as N_{pred} (or N_{id}, if the predecessor is not known or is empty) and the Contact header pointing to N_{id}. Assuming the predecessor 7_{pred} is known as node 1, then the node 7 sends the following request:

```
REGISTER sip:128.59.15.60 SIP/2.0
To: <sip:1@128.59.15.60>
Contact: <sip:7@128.59.15.56>; predecessor=sip:1@128.59.15.60
```

Procedure 3.5 N.OnRegisterSuccess (R:registration object, M:response message)

if R was a query, i.e., R.Contact is absent **then**
  if M.To = N_{id} **and** N_{succ} is empty **then**
    / set the finger table. */*
    $k \Leftarrow$ set-fingers(1, M.Contact) + 1 */* procedure 3.1 */*
    if $k \leq m$ **then**
      / more empty entries in finger table. */*
      / query for the next empty entry. */*
      $id \Leftarrow N + 2^{k-1}$
      send REGISTER M.Contact
          To: sip:id@sippeer.net
      trigger join complete event
    else
      / stabilize here with predecessor. */*
      send REGISTER M.Contact
          To: N_{pred} (or N_{id} if predecessor is empty)
          Contact: N_{id}; predecessor=N_{pred}
  else
    if $\exists i$, such that $F^i_{start} = M$.To **then**
      if $i \leq m$ **then**
        / found a pending query for empty finger table entry. */*
        $i \Leftarrow$ set-fingers(i, M.Contact) */* procedure 3.1 */*
        if $i < m$ **then**
          / more empty entries in finger table. */*
          / query for next empty entry. */*
          $id \Leftarrow F^{i+1}_{start}$
          send REGISTER N_{succ}
              To: sip:id@sippeer.net
        else if $i = m$ **then**
          if join is not complete **then**
            / stabilize here. */*
            send REGISTER N_{succ}
                To: N_{pred} (or N_{id} if predecessor isempty)
                Contact: N_{id}; predecessor=N_{pred}
            trigger join complete event

Procedure 3.6 N.Stabilize

/ This is called periodically by the ChordNode thread. */*
if join is completed **then**
  if $N_{succ} \neq N_{id}$ **then**
    send REGISTER N_{succ}
        To: N_{succ}
        Contact: N_{id}; predecessor=N_{pred}
  else if N_{pred} is not empty **and** $N_{pred} \neq N_{id}$ **then**
    / this is a singleton node in the ring */*
    set-fingers(1, N_{pred}) */* procedure 3.1 */*
  if $N_{pred} \neq N_{succ}$ **and** N_{pred} is not empty **then**
    send REGISTER N_{pred}
        To: N_{pred}
        Contact: N_{id}; predecessor=N_{pred}

When the node joins the DHT, it also starts its stabilization algorithm in the ChordNode thread's Run method. The stabilization algorithm is periodically invoked by the node to refresh finger table entries, successor and predecessor locations. The stabilization algorithm just initiates the SIP Registration for the successor and predecessor nodes with the local contact address in the Contact header (procedure 3.6). It avoids sending duplicate messages if the successor and predecessor nodes

are same.

```
REGISTER sip:10@128.59.15.55 SIP/2.0
To: <sip:10@128.59.15.55>
Contact: <sip:7@128.59.15.56>; predecessor=sip:1@128.59.15.60

REGISTER sip:1@128.59.15.60 SIP/2.0
To: <sip:1@128.59.15.60>
Contact: <sip:7@128.59.15.56>; predecessor=sip:1@128.59.15.60
```

If the node, N, discovers that the predecessor node is not empty or not same as this node, and the successor is this node (i.e., $(N_{pred} \neq \phi | N_{pred} \neq N_{id}) \& N_{succ} = N_{id}$), then it concludes that there is only one node in the DHT. In that case it sets the successor as the predecessor node and adjusts the finger table accordingly: $N_{pred} := N_{id}$.

When the successor or predecessor of this node receives this SIP REGISTER message with the contact header, it updates its state (procedure 3.7). In particular, if the sending node identifier is closer to the receiving node than the existing predecessor in the Chord ring, then predecessor is set as the sending node identifier. The 200 response contains the successor-list so that the original stabilizing node can update its state with the successor's successor list. The successor-list is sent using the Contact headers with different preference values, $q : 0 \leq q \leq 1$. Suppose there are k successors, then i^{th} successor has $q := 1 - \frac{i}{k}$ for $i = 0, 1, .., k-1$. Chord specifies successor-list to be of size $O(log(N))$. Node 10 sends the following response to node 7, indicating node 7's successor list: $\{10, 15, 22, 1\}$.

```
SIP/2.0 200 OK
To: <sip:10@128.59.15.55>
From: <sip:7@128.59.15.56>
Contact: <sip:10@128.59.15.55>; q=1; predecessor=sip:7@128.59.15.56
Contact: <sip:15@128.59.15.48>; q=.8
Contact: <sip:22@128.59.15.31>; q=.6
Contact: <sip:1@128.59.15.60> ; q=.4
```

Procedure 3.7 N.OnRegister (R:registration object, M:request message)

```
/* This is appended to procedure 3.4 */
if M.Contact is present then
  if M is not unregister, i.e., expires ≠ 0 then
    if N_pred is empty or M.From ∈ (N_pred, N_id) then
      N_pred ⇐ M.From
  if M.To = N_id then
    send response 200 OK
                  Contact: N_id; predecessor=N_pred; q=1.0
                  Contact: successor-list[0]; q=.8
                  Contact: successor-list[1]; q=.6
                  ...
  else
    send response 200 OK
                  Contact: N_succ; predecessor=N_id; q=1.0
                  Contact: successor-list[1]; q=.8
                  ...
```

When the stabilizing node, 7, receives the SIP 200 success response from its successor, 10, it updates its successor list using the Contact headers in the response (procedure 3.8). If node 7 discovers that successor node 10's predecessor lies between this node and the successor, $(7, 10)$, then node 7 sets its successor pointer to 10's predecessor.

At this point, node 7 refreshes the remaining finger table entries beyond 10. For example, it locates the next hop for the next finger table entry $F^3_{start} = 11 > 10$, and sends a SIP REGISTER query for sip:11@sippeer.net to sip:10@128.59.15.55 as shown below. If the next hop node for this key belongs to node 7 itself, then the request is sent to the successor, N_{succ}.

```
REGISTER sip:128.59.15.55 SIP/2.0
To: <sip:11@sippeer.net>
From: <sip:7@128.59.15.56>
```

Procedure 3.8 N.OnRegisterSuccess (R:registration object, M:response message)

```
/* This is appended to procedure 3.5 */
if R was not a query then
  if N_succ = M.To then
    pred ⇐ M.Contact.predecessor
    if pred ≠ N_id and pred ∈ (N_id, N_succ) then
      set-fingers(1, pred) /* procedure 3.1 */
    if pred = N_id then
      N_succ^list ⇐ M.Contacts in decreasing q
    if join is completed then
      /* stabilize the next finger entry. */
      i ⇐ ⌈log N_succ⌉
      if i < m then
        id ⇐ F_start^(i+1)
        if id ∈ (N_id, N_succ] then
          node ⇐ N_id
        else
          node ⇐ closest-preceding-finger(id) /* procedure 3.2 */
        if node = N_id then
          node ⇐ N_succ
        send REGISTER node
             To: sip:id@sippeer.net
```

When node 7 receives a response for this query for key 11, it continues to refresh remaining finger table entries (procedure 3.5) by sending more REGISTER requests. Fig. 6 shows the stable Chord network after node 7 has joined.

3.3.5 Node shutdown (graceful termination)

Suppose, node 7 wants to gracefully leave the network. It unregisters with its successor 10 and predecessor 1 (procedure 3.9). Once the node 10 and 1 know that node 7 has left, other nodes will eventually know using the stabilization algorithm.

Procedure 3.9 N.Leave

```
if N_pred is valid and N_pred ≠ N_id then
  send REGISTER N_pred
       To: N_id
       Contact: N_id ∪ N_succ^list
       Expires: 0
if N_succ is valid and N_succ ≠ N_id then
  send REGISTER N_succ
       To: N_id
       Contact: N_id ∪ N_succ^list
       Expires: 0
```

To unregister, node 7 sends a SIP REGISTER request with Expires header with value 0 as follows. The Contact headers are also present in the request indicating the successor list.

```
REGISTER sip:1@128.59.15.60 SIP/2.0
To: <sip:7@128.59.15.56>
From: <sip:7@128.59.15.56>
Expires: 0
Contact: <sip:7@128.59.15.56>; q=1.0; predecessor=sip:1@128.59.15.60
Contact: <sip:10@128.59.15.55>; q=.8
...

REGISTER sip:10@128.59.15.55 SIP/2.0
To: <sip:7@128.59.15.56>
```

```
From: <sip:7@128.59.15.56>
Expires: 0
Contact: <sip:7@128.59.15.56>; q=1.0; predecessor=sip:1@128.59.15.60
```

Procedure 3.10 N.OnRegister (R:registration object, M:request message)

/ This is appended to procedure 3.7 */*
if M.Contact is present **then**
  $to \Leftarrow M$.To
  if M is unregister, i.e., expires = 0 **then**
    if $to = N_{pred}$ **then**
      $N_{pred} \Leftarrow M$.Contact.predecessor
    if $to = N_{succ}$ **then**
      $N_{succ}^{list} \Leftarrow M$.Contacts in decreasing q
      set-node-as-inactive(to) */* procedure 3.11 */*
  else
    / See procedure 3.7 */*

Procedure 3.11 N.set-node-as-inactive ($node$)

/ Set the node as inactive in F and N_{succ}^{list} */*
for all n in F and N_{succ}^{list} **do**
  if $node = n$ **then**
    $n.alive \Leftarrow false$
/ Update N_{succ}^{list} to replace dead nodes */*
$previous \Leftarrow N_{id}$
for $k \leftarrow m$ down to 1 **do**
  if F_{node}^{k} is **not** alive **then**
    $F_{node}^{k} \Leftarrow previous$
  else
    $previous \Leftarrow F_{node}^{k}$

When node 10 receives the unregistration from node 7, it realizes that its predecessor is leaving, so it updates its predecessor location using the predecessor value, 1, in the Contact header: $10_{pred} := 1$ (procedure 3.10). Similarly, when node 1 receives the unregistration from node 7, it realizes that its successor is leaving, so it updates its successor to be the next active successor in 1_{succ}^{list}. When a node location is made invalid, it is removed from the 1_{succ}^{list} and $1 :: F$. Any dead finger table node location is changed to the next alive entry in the finger table (procedure 3.11).

Node 7 should wait for confirmation responses (until a reasonable timeout) from 10 and 1 before shutting down.

3.3.6 Node failure and failover

Node failure, unlike graceful shutdown, needs to be detected automatically by other nodes when the SIP REGISTER message fails. If node 7 fails due to some reason, the neighbors 10 and 1 detect the failue and update their states to reflect it. The SIP library libsip++ generates OnRegisterFailed event when the outgoing REGISTER message gets a failure SIP response or times out. Registration failures can happen due to many reasons and at different stages (procedure 3.12):

Global failure or SIP 600 response may be received if duplicate node identifier is detected. For example, if the hash function generates the same node identifier value for two nodes, $H(A) = H(B)$, but A and B have different transport addresses. If A is already in the network, then REGISTER from B will be rejected with a SIP 600 global failure response. The response contains N_A, so node B can use this address, N_A, as the outbound proxy instead of joining the Chord network.

Discovery failure may happen if there is no other P2P-SIP node in the multicast domain. In this case, other means of discovery should be used, e.g., service location for P2P-SIP server using SLP, or using the bootstrap nodes to join the network. Alternatively, the node can assume singleton node in the Chord network, and adjust its states accordingly. This is useful within a single LAN environment such as P2P VoIP within an organization.

Procedure 3.12 N.OnRegisterFailed (R:registration object, M:response message)

```
to ⇐ M.To
uri ⇐ M.uri
if to = N_id and R is a 600-class global failure then
  /* Discovery failed. Probably duplicate node identifier. */
  if M.Contact is present then
    set-fingers(1, M.Contact) /* procedure 3.1 */
    trigger join failed event
else if uri is multicast discovery address, 224.0.1.75 then
  /* Discovery failed. Assuming singleton node in Chord. */
  for i ← 1tom do
    F^i_node ⇐ N_id
  N_pred ⇐ N_id
  trigger join complete event
else
  succ ⇐ N_succ
  if to = uri or uri ≠ N_succ or R.Contact is empty then
    set-node-as-inactive(uri) /* procedure 3.11 */
    if uri = succ then
      successor_failed ⇐ true
  else
    set-node-as-inactive(to) /* procedure 3.11 */
  if uri = N_pred then
    N_pred ⇐ empty
  node ⇐ N_id
  if successor_failed is true then
    /* select the next successor. */
    if N_succ is empty or N_succ = N_id then
      /* Successor not found. Ignore. */
    else if N_succ ∈ (N_id, to] then
      node ⇐ N_succ
    else
      /* Do not know where to send. */
  else
    node ⇐ closest-preceding-finger(uri) /* procedure 3.2 */
    if node = N_id then
      /* No more addresses left for successor. */
  /* Now resend only query messages if possible. */
  if R.Contact is absent and node ≠ N_id and node ∈ (N_id, to] then
    R.uri ⇐ node
    re-register using R
```

Besides duplicate node detection and discovery failure, we want to address the remote node failure so that the network can failover automatically. When a remote node fails, this node deactivates that remote node location from its finger table and successor list. There are following cases: (1) if the destination key (To header) and the next hop node (request-URI) were same, that node location is deactivated, (2) if the request was not sent to the successor ($uri \neq N_{succ}$), the next hop node location is deactivated, but not the destination key, (3) if the request was a query (no Contact header), the next hop node location is deactivated, but not the destination key, otherwise (4) the node represented by the destination key (To header) is deactivated. If the next hop node is the predecessor, then the predecessor variable is reset ($N_{pred} := \phi$).

The next step is to re-send the original query request to the new failover hop. If the successor node failed, then the next successor is chosen and the request is sent again to the new successor, if one is found. Otherwise, the query is sent to the next closest preceding finger to the destination key. Only the REGISTER query and not updates, are re-sent, because updates are refreshed anyway in the next stabilization interval. If the new next hop is after the destination key in the Chord ring, then the query is not re-sent, and is considered failed query.

Procedure 3.13 N.OnRegisterSuccess (R:registration object, M:response message)

```
/* This is appended to procedure 3.5 and 3.8 */
if R was a query, i.e., R.Contact is absent then
   to ⇐ M.To.user
   succ ⇐ M.Contact
   pred ⇐ M.Contact.predecessor
   if to = N and succ equals N_id but has different address then
      /* duplicate node identifier found */
      set-fingers(1, succ) /* procedure 3.1 */
      trigger join failed event
   else if M.To = N_id and N_succ is empty then
      /* See procedure 3.5 */
   else
      /* See procedure 3.5 */
```

The node checks for duplicate identifiers, when the initial discovery returns a a duplicate successor node identifier (procedure 3.13). For example, if node sip:7@128.59.15.56 discovers the successor as sip:7@128.59.15.45, it uses 128.59.15.45 as the IP address of the outbound proxy and does not join the Chord network directly.

3.4 User registration

The DHT module maintains the underlying P2P overlay network, whereas the user location module takes care of user profile and registrations. Both the modules use the SIP REGISTER messages. We describe the user account maintenance in this section.

Suppose the table of UserAccount in the node is represented as A such that $A[k]$ is the user registration for user identifier k. Suppose the list of LocalUserAccount is represented as L such that L^i is the i^{th} local user registration.

Procedure 3.14 RegisterUser (k:user account or identifier)

```
L.append(k)
node ⇐ N.Find(k_id) /* procedure 3.15 */
if node = N_id then
   A[k_id] ⇐ k
else
   send REGISTER node
        To: k_id, From: k_id, Contact: k_contact
```

Procedure 3.15 N.Find (key:identifier to find)

```
/* Find the next hop node for key */
if key ∈ (N_id, N_succ] then
   node ⇐ N_id
else
   node ⇐ closest-preceding-finger(key) /* procedure 3.2 */
if node = N_id then
   node ⇐ N_succ
return node
```

3.4.1 Registration handling

When a user registers her identifier, say k=alice@example.com, a new LocalUserAccount object is created to represent this user. The next step is to transfer this registration on to the P2P network to the responsible Chord node (procedure 3.14). Suppose the user identifier key is H(k)=1, then this user registration will be stored in the DHT on the node which is responsible for this key, 1. The DHT's Find method is invoked to get the next hop location and the request is forwarded (see procedure 3.15). If the local node is responsible for this key, then the registration is stored locally. For example, when user Alice registers from node 7 (Fig. 6), the next hop is 1_{id} so the following REGISTER request is sent:

```
REGISTER sip:128.59.15.60 SIP/2.0
To: <sip:alice@example.com>
From: <sip:alice@example.com>
Contact: <sip:alice@128.59.15.56>
```

The receiving node should authenticate any registrations (Section 5). The registration is replicated at all the nodes in the successor-list of the responsible node, 1^{list}_{succ}.

When node 1 receives the message, it recognizes that the destination key in To header belongs to a user rather than a node. As show in procedure 3.16, there are following cases: (1) if $key \in (N_{pred}, N_{id}]$, then this node is the responsible node, (2) if $N_{succ} = N_{id}$, then there is only this node in the Chord ring, so obviously this node is the responsible node, otherwise (3) find the closest preceding finger for this key and proxy the SIP request to that node location. In this example, node 1 uses itself as the responsible node and stores the registration for alice@example.com.

Procedure 3.16 N.OnRegister (R:registration object, M:request message)

```
to ⇐ M.To
if to is not a node identifier then
  if to ∈ (N_pred, N_id] or N_succ = N_id then
    /* Register the user locally. */
    A(to) ⇐ M
  else
    if to ∉ (N_id, N_succ] then
      node ⇐ closest-preceding-finger(to) /* procedure 3.2 */
    else
      node ⇐ N_id
    if node = N_id then
      proxy M to N_succ
    else
      proxy M to node
else
  /* to is a node identifier. see procedure 3.10 */
```

Now, node 1 replicates the registration to other nodes in N^{list}_{succ} (procedure 3.17). For example, it sends the following REGISTER message to node 10_{addr}, with To header containing the destination key alice@example.com, From header containing 1_{id} and Contact header containing original contact location of Alice.

```
REGISTER sip:128.59.15.55 SIP/2.0
To: <sip:alice@example.com>
From: <sip:1@128.59.15.60>
Contact: <sip:alice@128.59.15.56>
```

Procedure 3.17 $A[k] \Leftarrow M$

```
A[k] := M
if M.To = M.From or M.Reason = "leaving" then
  /* This node is responsible for k */
  for all S in N^list_succ do
    send REGISTER S_addr
         To: k_id, From: N_id, Contact: A[k]_contact
```

The receiving node 10 recognizes this to be a registration tranfer, since the To header and From header are different. It stores the registration without routing it further. It should authenticate the sending node 1 before storing the registration. If the From header is also a user identifier, then the REGISTER request is a third-party registration (e.g., secretary registering on behalf of her boss), and should be routed using the P2P-SIP routing algorithm based on the To header. Third-party and transferred registrations should be authenticated at each proxy.

3.4.2 Node shutdown (graceful termination)

When a node gracefully leaves the network, it should transfer all stored registrations to the new responsible nodes, which is its immediate successor (procedure 3.18). For example, when node 1 leaves, it sends the following REGISTER request to $1_{succ} = 7_{addr}$.

```
REGISTER sip:128.59.15.56 SIP/2.0
To: <sip:alice@example.com>
From: <sip:1@128.59.15.60>
Reason: leaving
Contact: <sip:alice@128.59.15.56>
```

Procedure 3.18 N.Leave

```
/* unregister this node using procedure 3.9 */
/* unregister local accounts in L */
for all u in L do
  R ⇐ N.Find(u_id)
  if R ≠ N_id then
    send REGISTER R_addr
        To: u_id, From: u_id, Expires: 0
/* transfer local registrations */
for all k in keys(A) do
  send REGISTER N_succ
      To: k_id, From: N_id, Expires: 0, Reason: leaving
```

When node 7 receives the registration transfer with the Reason field as leaving, it can decide to assume the responsibility for this registration. Alternatively, node 7 can conclude that node 1 is leaving from node unregistration message, and assume responsibility for all the keys that were transfered from node 1 before. The decision is local to node 7 since assuming responsibility for registration is an extra load. Alternatively, when Alice's user agent refreshes the registration, the appropriate responsible node (which may be 7) will get the new registration. Suppose node 7 decides to accept the responsibility for this destination key, it replicates the registration to all the nodes in 7^{list}_{succ}. For example, it sends a SIP REGISTER to its successor 10_{addr} as follows:

```
REGISTER sip:128.59.15.56 SIP/2.0
To: <sip:alice@example.com>
From: <sip:7@128.59.15.56>
Contact: <sip:alice@128.59.15.56>
```

Node 10 had earlier received the replicated registration for alice@example.com from node 1. When node 10 receives the the new REGISTER from node 7, it concludes that the responsibility for key alice@example.com has been transferred from node 1 to node 7.

3.4.3 Node failure and failover

Node failure is similar to node shutdown, except that the failed node does not transfer registrations. The immediate successor detects that its predecessor has failed and owns the responsibility for the keys from its immediate predecessor. For example, if node 7 fails, node 10 detects the failure, and can decide to assume responsibility for the destinations keys sent by node 7. If node 10 decides to not assume the responsibility, it will get the next registration refresh from Alice's user agent, at that time it can authenticate Alice and assume responsibility.

3.5 Call setup and message proxy

So far we have described only the registration request routing. A SIP request such as REGISTER or INVITE belongs to either an user or a node, based on the destination being the user identifier or the node identifier, respectively. For REGISTER request, the SIP To header is used for computing the key for routing decision. For all other requests (e.g., INVITE, MESSAGE), the request-URI is used to make the routing decision. However, this means that SIPPEER must not modify the request-URI on proxy for non-REGISTER requests.

3.5.1 Message proxy

When an incoming non-REGISTER request is received, and the request-URI is a user identifier (i.e., not a node identifier), and the request does not belong to an existing dialog or local user on this node, then SIPPEER looks up for the user key in its registered user map, A, as shown in procedure 3.19. If no registration is found, then a 404 response is returned if the key belongs to this node, otherwise the request is proxied to the next hop node. If valid registrations are found, the request is proxied to those registered contact locations.

Procedure 3.19 N.OnReceiveRequest(T:transaction, M:message)

if M.method == REGISTER **then**
  / user or node registration: procedure 3.16 */*
else if M.uri is some node identifier **then**
  / this is for the DHT module */*
else if M belongs to existing dialog **then**
  / let the dialog state-machine handle it */*
else if M.uri is in L^i **then**
  / M is for local user on this node */*
else if M.uri is **not** in $A[k]$ **then**
  / no registration found */*
  if M.uri $\in (N.prev, N]$ **or** N.Find(M.uri) failed **then**
    send response 404 User not found on P2P/SIP
  else
    $next$:= N.Find(M.uri)
    proxy M **to** $next$ without modifying uri
else
  / registration found */*
  $contacts$:= $A[M.uri]$.contacts
  proxy M **to** $contacts$ using parallel forking without modifying uri

3.5.2 Multimedia call setup and instant messages

SIPPEER allows initiating or terminating a SIP call using the command line interface. When the node initiates a request, or acts as an outbound proxy for an existing SIP client, it tries both traditional DNS lookup for the user domain and P2P lookup for next hop in Chord for the user identifier. When one branch gets a final response, the other branch is cancelled. **TODO: Do we handle request merging in libsip++.** Alternatively, some P2P-SIP node can try DNS first and fallback to P2P lookup on failure to get DNS NAPTR/SRV records.

If the node initiates a call or acts as an outbound proxy, it does both DNS and P2P lookup, otherwise it does only P2P lookup. To detect that this node is acting as an outbound proxy for a third party SIP client, SIPPEER uses the Reason header field. All requests initiated or proxied by SIPPEER has a Reason header field indicating that the DNS lookup was already done. When a SIPPEER node receives a message with this Reason it does not invoke another DNS lookup, but uses only P2P lookup.

Usually, BYE message is sent directly between the two endpoints to terminate the call, without involving P2P lookup. Other messages such as MESSAGE for instant messaging follow similar lookup mechanism as INVITE. The SUBSCRIBE message handling for locating users in the "friends list" on startup is described in Section 6.

This section describes the various experiments we have conducted with SIPPEER. **TODO: Should I add a subsection for Java GUI?**

4.1 Adaptor for existing SIP phones

In this section we describe how to use SIPPEER as a P2P-SIP adaptor for existing SIP user agents such as Columbia University's sipc [30], Cisco IP phone 7960, Pingtel IP phone, Xten Networks' X-Lite client v2.0 and Microsoft Windows Messenger v5.0. Similar mechanism can be used for other clients. The idea is to use SIPPEER as an outbound proxy for these clients.

Consider an example P2P-SIP network with five nodes with identifiers 1, 7, 10, 15 and 22 as shown in Fig. 7.

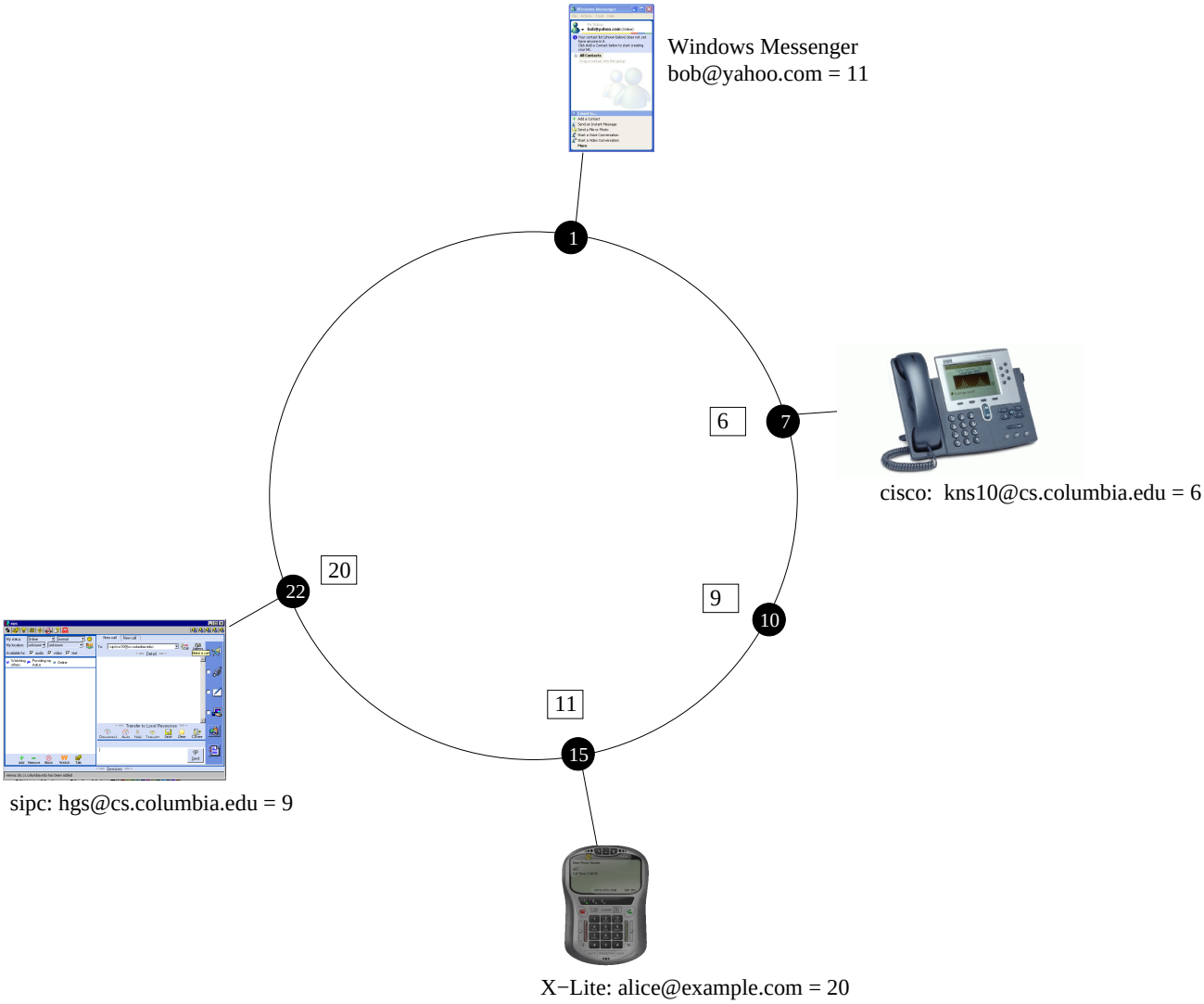

Figure 7: Example P2P-SIP network with existing phones

4.1.1 Columbia sipc

User hgs@cs.columbia.edu uses sipc. He sets the outbound proxy as node 22_{ip}, which is vienna.clic.cs.columbia.edu as shown in Fig. 8. This allows sipc to send all the outbound messages to node 22. He add a new account to register his SIP identifier as shown in Fig. 9. Note that the login information is not used unless the P2P-SIP network prompts for authentication. Sipc sends SIP REGISTER for hgs@cs.columbia.edu to node 22. Based on the hash value of the identifier, H(hgs@cs.columbia.edu)=9, the request gets routed to node 10 which stores the registration.

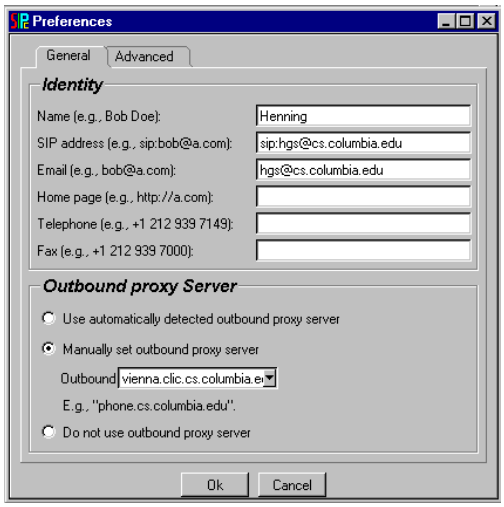

Figure 8: Sipc: preferences

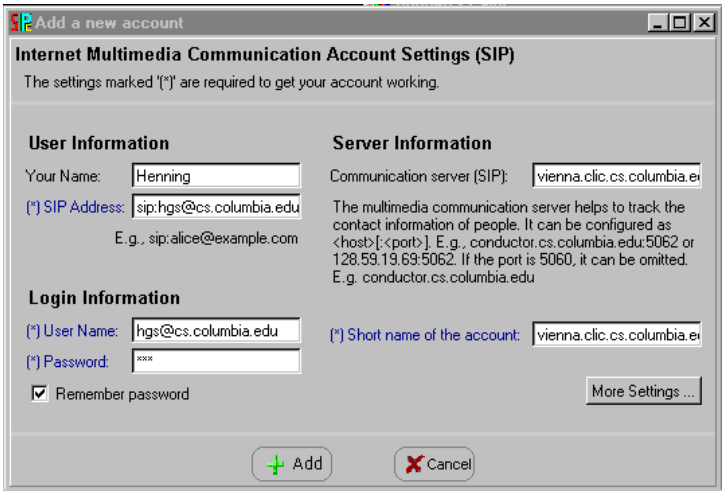

Figure 9: Sipc: new account

Sipclite is another user agent with simpler user interface than sipc but uses the same source code base and configuration mechanism.

```
serverhost      localhost
serverport
username        Henning Schulzrinne
sip             sip:hgs@cs.columbia.edu
proxycall       1
protocol        UDP
monitor         0
auth_method     Digest
stun            stunc
```

Figure 10: Sipclite configuration

For software-based SIP phones, a wrapper script can invoke both the SIPPEER and the SIP phone, so that no explicit configuration is needed. For example, we have written a shell script on Linux that modifies the sipclite configuration to use SIPPEER as outbound proxy on localhost as shown in Fig. 10. Then, it invokes SIPPEER, waits for three seconds, and starts sipclite. From the user's point of view, he just has to start one script to add P2P-SIP service to his SIP phone using SIPPEER.

Since SIPPEER listens on UDP packets on default SIP port, 5060, as well as another randomly allocated port (for N_{addr}), configuring the outbound proxy port as default works for the clients on UDP. For TCP, the client needs to set the port correctly based on N_{addr}. A command line option, -p, in SIPPEER allows using a fixed port for N_{addr}.

4.1.2 Cisco IP phone

User kns10@cs.columbia.edu uses Cisco IP phone. He sets the outbound proxy as node 7_{ip}, which is 128.59.15.56. An example configuration is shown in Fig. 11. When the phone starts up, it sends a SIP REGISTER message to the outbound proxy for identifier kns10@cs.columbia.edu. The registration for destination key, H(kns10@cs.columbia.edu)=6, gets stored on node 7.

At this point kns10 can call hgs, by dialing the URL hgs on the Cisco phone. Call is received by hgs on sipc who accepts the calls. When kns10 hangs up, the Cisco phone sends the SIP BYE message for hgs@cs.columbia.edu to the

```
line1_name: "kns10"
line1_shortname: "kns10 on p2p"
line1_authname: "kns10@cs.columbia.edu"
line1_password: "something"
line1_displayname: "Kundan Singh"
outbound_proxy: "128.59.15.56" ; budapest.clic.cs.columbia.edu
outbound_proxy_port: 5060
proxy1_address: "cs.columbia.edu"
proxy1_port: 5060
proxy_registrer: 1
```

Figure 11: Example Cisco phone configuration

outbound proxy node 10. This works but is not correct. A SIP [20] compliant user agent should send BYE directly to Contact address of the other endpoint, i.e., sipc's IP address, because SIPPEER node 10 does not do record-route.

When user hgs calls kns10@cs.columbia.edu, the Cisco phone rings. When kns10 picks up the phone, call setup is completed. When hgs terminates the call, sipc sends the SIP BYE message directly to the Cisco phone.

4.1.3 X-lite

User alice@example.com uses the X-Lite client and sets the proxy as node 15_{ip}, which is dhaka.clic.cs.columbia.edu as shown in Fig. 12. When the user agent starts up, it sends a SIP REGISTER request to node 15, which eventually gets stored on node 22 based on the key, H(alice@example.com)=20.

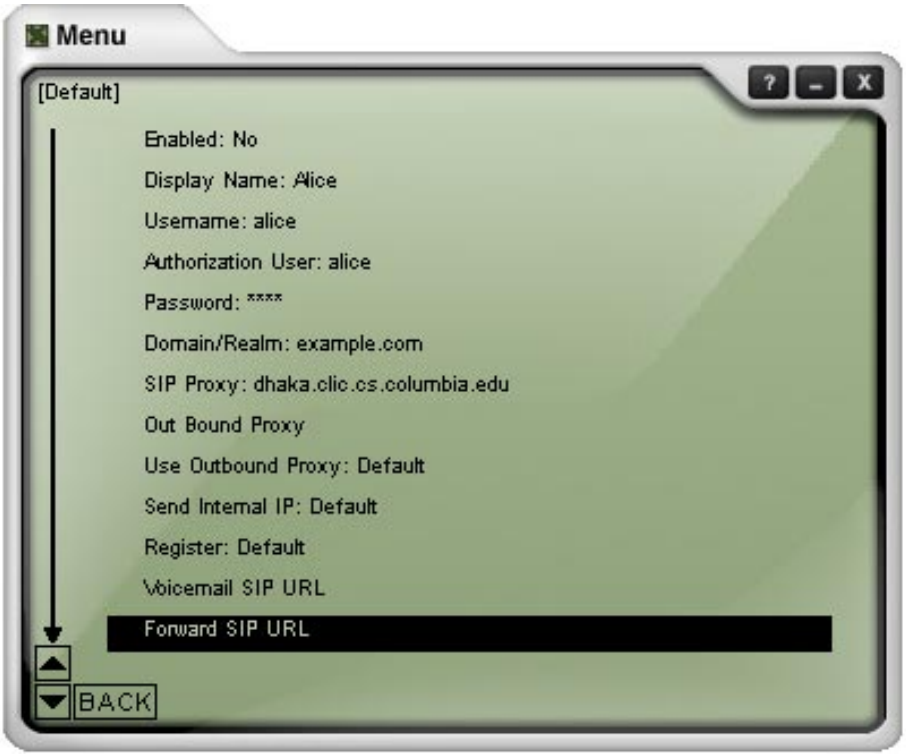

Figure 12: SIP proxy settings

At this point Alice can create address book entry for another user, say hgs@cs.columbia.edu as shown in Fig. 13. To make an outbound call to hgs, she dials the speed dial number of 447. The user agent sends a SIP INVITE message to node 15, which eventually gets routed to user hgs. The ACK and BYE messages are sent directly from X-Lite to sipc. The call flow is analogous in the reverse direction when hgs calls alice@example.com.

> A bug in X-Lite that ignored the branch parameter after a space caused initial interoperability problem. We modified our software to workaround this problem, however problem may appear with other clients trying to

Figure 13: Speed dial

Figure 14: Making a call

interoperate with X-Lite with or without P2P-SIP network.

4.1.4 Microsoft Windows Messenger

User bob@yahoo.com uses Microsoft Windows Messenger and sets the SIP communication service proxy as node 1, warsaw.clic.cs.columbia.edu, as shown in Fig. 15. When user logs in, it sends a SIP REGISTER message to node 1, which eventually gets stored in node 15 based on the key, H(bob@yahoo.com)=11.

To make an outgoing audio call, Bob selects that option and specifies the destination user identifier, hgs@cs.columbia.edu, as shown in Fig. 17. The ACK and BYE messages are sent directly to the other end point. The reverse direction call setup is similar.

4.2 Interoperability issues

This section describes some interoperability issues, certain design choices to work around these issues, and suggestions for future enhancements for interoperability.

SIP phones that one wants to use with SIPPEER adaptor should be SIP compliant [20]. Using an adaptor shields some of the interoperability issues, but does not solve all the problems. For example, if SIP phones A and B are not interoperable, but both can interoperate with SIPPEER, then the system may or may not work, depending on whether the non conformance issue affects the proxy/registrar or the user agent, respectively. We try to workaround specific proxy/registrar related interoperability problems without violating SIP. For user agent related problems, SIPPEER may act as a back-to-back user agent (B2BUA) to shield the non-conforming phones.

Some phones do not implement outbound proxy as per SIP specification [20], which says that the outbound proxy should be treated as a pre-loaded route set. This means if the proxy does not insert Record-route header in the initial INVITE request, then subsequent request in the dialog such as BYE should not be sent to the proxy. Suppose a user, hgs, using sipc sends the initial INVITE via P2P-SIP network, to kns10 using non-conforming Cisco IP phone. When kns10 wants to hang up, the Cisco phone sends BYE to the outbound proxy SIPPEER node. Since the destination hgs may not be registered in P2P-SIP, this does not work.

We have modified SIPPEER to work around this problem. SIPPEER proxies the request to the request-URI instead of P2P-SIP next hop, if the request-URI does not belong to local host and contains a port number, hence likely to be a contact location rather than an user identifier. This works in this scenario of Cisco phone, but will not always work.

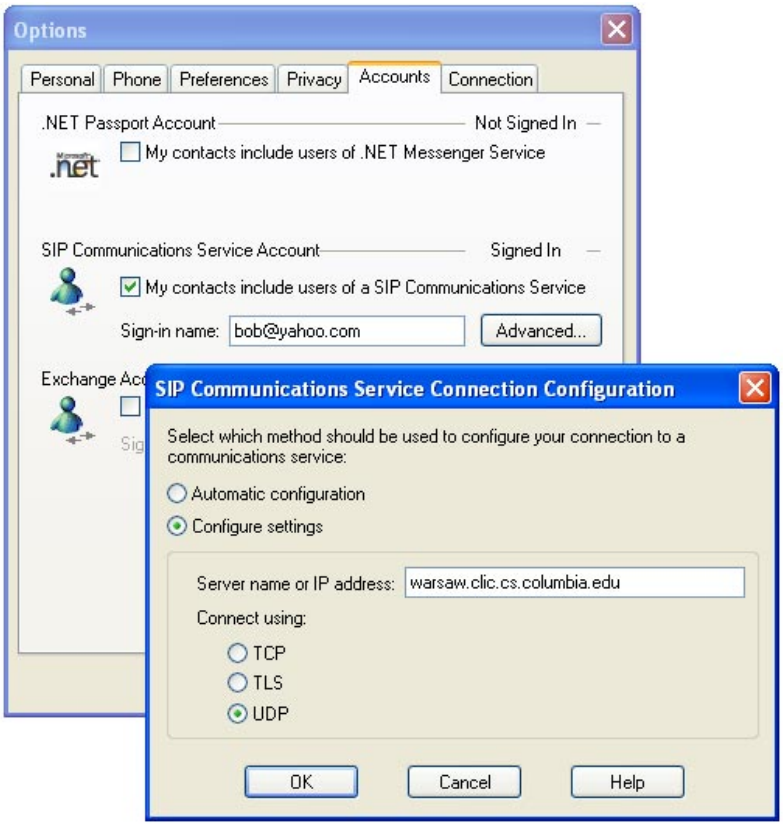

Figure 15: Windows Messenger configuration

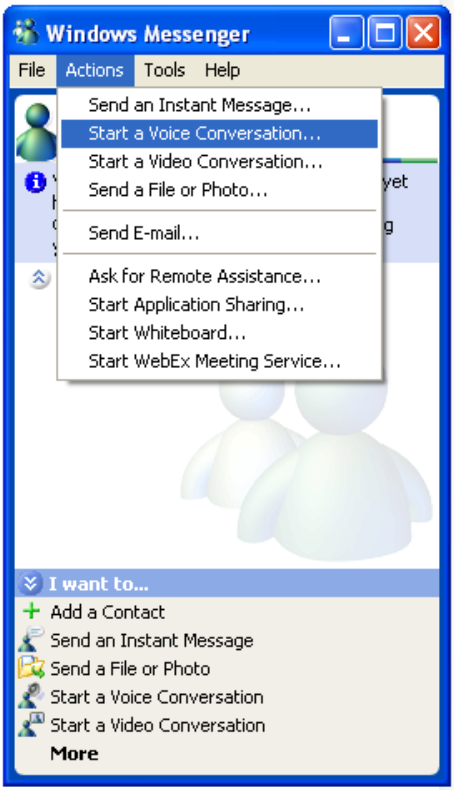

Figure 16: Select voice call

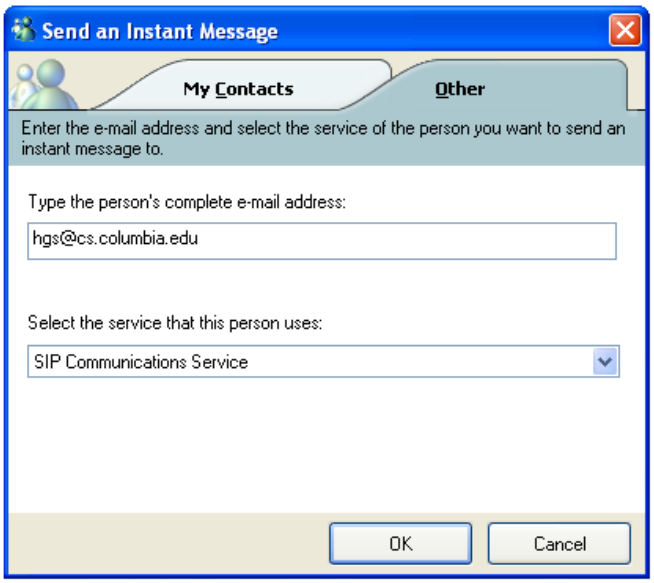

Figure 17: Specify remote address

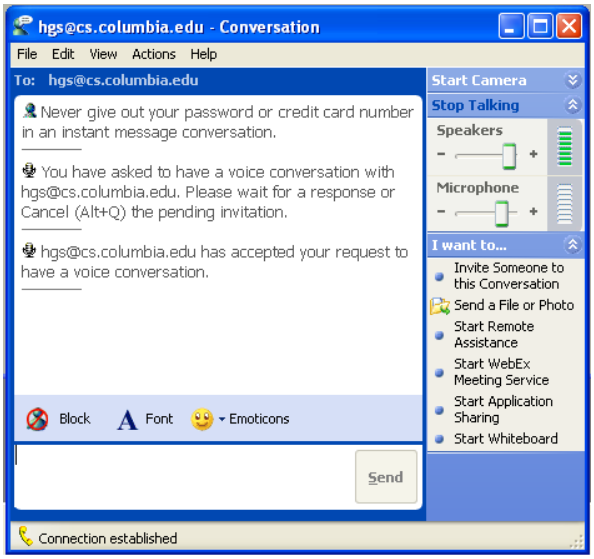

Figure 18: In a voice call

5 Security

P2P applications expose existing security threats such as virus and worms to larger networking community, even to corporate networks behind firewalls. In the context of P2P-SIP, there are a number of different types of threats, some of which exist in server-based SIP, whereas others in P2P. In this section we summarize various threats and try to provide solutions to some aspects of these threats. We have not yet implemented these mechanisms in our SIPPEER.

Some of the security problems in P2P are hard to solve. There is a tradeoff between security risks and convenience of server-less systems.

5.1 Threats

Malicious program: The application can allow various forms of attacks, break-in, spying or spread virus, spy-wares or worms. Software developed by trusted entities or open source community can reduce this risk. Even software bugs such as buffer overflow can be exploited by hackers. Running the application as a regular user instead of an administrator (on Windows) or super-user (on Unix) can reduce the risk to some extent.

Copyright violation: P2P-SIP architecture can be easily extended to support file transfer. For example, SIP INVITE can initiate a ftp session using appropriate SDP message body. The problem is similar to other P2P file sharing applications. P2P-SIP does not have an efficient search method, i.e., search for files using regex pattern matching. This also reduces the threat, since not many people will use this for sharing music files if the files can not be efficiently searched.

Stolen identity: The system should prevent a malicious user from stealing the identify of another user. In P2P-SIP, we reduce the risk by requiring that the user identify must be a valid email address. The system generates a password for the user identify and sends it to the email address. We describe this mechanism later in this section. The system should be able to authenticate and securely determine whether the peer is who she claims to be.

Privacy: Certain user information needs to be conveyed to the other peers to allow call routing. The system should ensure that no sensitive data is conveyed, which can be misused later. In particular all signaling and media communication should be encrypted. Privacy and confidentiality in a pure P2P system is difficult. Some parts of the problem is addresses in this section using public key mechanism.

Trust: Detection of misbehaving peers in Chord-based P2P network is yet an unsolved problem. There are guidelines that can help reduce the risk. In particular, it is hard to detect a misbehaving node that routes some calls correctly, but drops others. Secondly, the node may secretly log the call information for later misuse.

Besides the above threats there are more threats in P2P systems such as anonymity [12], free riding [3] and accounting. Caller anonymity can be provided by having the proxies hide the identity of the caller. A reward or credit system is needed to motivate peers to join the DHT, and reduce free riders. Call accounting is needed for PSTN calls, and can be provided by the gateway. Accounting within P2P-SIP nodes is not required.

5.2 Identity protection

When a user signs up with the P2P-SIP network for the first time, we need to verify that the user identifier is valid and indeed belongs to the user. In the absence of public key infrastructure (PKI), the system can generate a new password and send it in an email to the user. This requires that the user identifier be same as her email address. For example, when Alice signs up with identifier alice@example.com by sending a SIP REGISTER message, the responsible node generates a random password for Alice and sends it in an email to alice@example.com. It then challenges Alice with digest authentication [5]. We use the domain part of the user identifier as the digest realm for authentication. The responsible node maintains the authentication information (user identifier, realm and MD5 hash of "user:realm:password") on the DHT. The information is indexed by the user key. This information is required and sufficient for future authentication of any user signing up with the same identifier. A usable time-to-live, say one month, can be used. The information is refreshed when the user subsequently signs up. So if the user identifier is unused for a month, subsequent sign-in generates new password sent to the user's email

address. **TODO: Some mechanism for the user to request a new password is useful, e.g., new header in REGISTER message as Send-Password: yes, but can be misused as email spam**

The email sent to Alice contains the user identifier, realm and password. It also contains the IP address (or other identifying information) of the original sender so that Alice can report abuse if she was not the one trying to sign up with P2P-SIP. When Alice receives the password, she signs up again with the appropriate credentials. Subsequent sign-up follow the same procedure.

When registrar node (A) fails or shuts down, the registration is transferred to another DHT node (B). If node B trusts node A, it just needs to authenticate A, otherwise it re-generates a new password and sends it to the user's email address. We believe that once we have a P2P reputation system, only the "more" trusted nodes will be present in the DHT.

The problem is still there if the registrar node is malicious, and can cause denial of service (DoS).

5.3 Misbehaving nodes

Certain guidelines can be followed to detect and avoid misbehaving nodes [28]. For example, the caller can prefer redirect (iterate) mode of operation, so that it can monitor at each step whether the routing is as per DHT specification. There should be no single point of decision. In the current system, the responsible node also does replication. So a misbehaving responsible node can make the user unavailable. The P2P-SIP nodes can periodically verify the routing correctness, e.g., by making calls to itself through some other node.

Generally speaking there are three models to prevent misbehaving nodes in P2P: (1) hide the security algorithms and protocols, so that only the single vendor implementation will be running on the node (e.g., Skype), (2) form a social network of peers in unstructured P2P system, or (3) keep only trusted nodes in the structured P2P network. What we want is to have all kinds of nodes in Chord-based structured P2P, but nodes should be able to selectively trust other nodes in call routing and registrations.

A number of reputation systems have been proposed for P2P [6,8,9,31]. However, they focus on file sharing systems (not real-time), have centralized components, assume co-operating peers or have problems of collusion and multiple identities. Further study is needed to detect the peers who are known to drop calls or do other malicious behavior so that they are not used in the call routing path and not allowed to become part of the underlying DHT.

Malicious nodes cause two kinds of problems visible to the user: (1) DoS, i.e., the user identifier becomes unavailable, and (2) intercept, i.e., call goes to the wrong person. The latter can be detected using end-to-end authentication assuming a previous communication has happened. The former is difficult to eliminate without a P2P reputation system. "Node calling itself" mechanism can be used for detection to some extent.

5.4 Data privacy

There are three types of information about a user that can be stored on another node.

public: Node should be able to see the information for message routing, authentication, or other processing. For example, user's encrypted password, contact locations (SIP Contact header including q value, expiry and URI), programmable script (CPL), voicemail options (such as timeout to go to voicemail, maximum message size, etc).

private: Only the user should be able to see and modify this information. Private data must be encrypted by the user before storing on the node. For example, user's address book, groups, calendar appointments, watcher and watchee list, programmable scripts (e.g., LESS, CPL, SIP CGI or servlet) and other profile information.

protected: User should be able to see and modify the information, but some other user should be able to create the information. The storing node should not be able to see or modify the information. For example, voice/video mail, and offline messages. Protected data is encrypted by the sender, and decrypted by the recipient.

Public key mechanism can be used for these.

5.5 Programmable call routing

The responsible node can not trust the user except for storing her information and routing her calls. For example, un-trusted programmable call routing scripts such as SIP-CGI and SIP Servlet will not be run by the responsible node on the user's behalf. On the other hand, trusted and secure CPL scripts can be run by the responsible node. However, this is purely a local decision by the responsible node.

5.6 User aliases

Aliases or other names are treated as user identifiers and all profile information must be duplicated. Sharing the profile information among the aliases causes complicated trust requirements. On the other hand, user will typically have provisions in her user agent to register with multiple user identifiers or line presence, so that does not require support from P2P-SIP.

Alternatively, user can maintain a primary identifier such as alice@example.com and point all other identifiers such as Alice.Wonderland@yahoo.com and aw76@columbia.edu by registering them with contact as the primary identifier. This avoids duplicating the profile information for secondary identifiers, but increases the call setup latency when someone wants to reach the user by her secondary identifier. To avoid going into search loop, responsible node for the secondary identifier will typically redirect the call request to the primary identifier. The caller's phone then retries search for the primary identifier on the P2P-SIP network. **TODO: Does proxy mode work? How to inform the node to use search on P2P-SIP instead of DNS lookup for the contact location.**

Aliases follow the same procedure for first time log-in, i.e., aliases must be a valid email address and the password is sent to the email address represented by the alias identifier.

6 Advanced services

Many advanced services can be specified using SIP URI. For example, sip:staff-meet@conferencing.net can indicate the pre-scheduled conferencing service by conferencing.net domain, or sip:dialog.voicexml@ivr.net can reach the generic interactive voice response service. Such services can be built transparently in the basic implementation. For example, a SIP conference server can register all the pre-scheduled conferences in the P2P network, an answering machine module can register to receive incoming calls on behalf of all the registered users, and a VoiceXML browser can register the specific voice dialog service such as voice mail access.

In this section we describe how to extend SIPPEER to support some advanced services such as presence, PSTN interworking, firewall and NAT traversal. The procedures described in this section are not yet implemented in SIPPEER.

6.1 Presence and event notification

SIPPEER has only the rudimentary support for event subscription and notification [13, 16]. Other SIP users agents that support presence or other events can work in conjunction with SIPPEER. SIPPEER facilitates subscription migration from a P2P-SIP node to subscribee's endpoint and vice versa.

Suppose a subscriber, Alice (alice@example.com), subscribes for the presence status of Bob by sending SIP SUBSCRIBE message to bob@yahoo.com. Note that the P2P-SIP node may not be able to authenticate the subscription since the subscriber Alice may not be registered with P2P-SIP network at all.

```
SUBSCRIBE sip:bob@yahoo.com SIP/2.0
To: <sip:bob@yahoo.com>
From: <sip:alice@example.com>
```

If Bob does not have a valid registration in P2P-SIP network, the responsible node for Bob's user identifier keeps the subscription information. It responds with a SIP 202 pending response, and a SIP NOTIFY message with Subscription-State of pending (procedure 6.1). If SIPPEER understands the event-package (e.g., presence package may be implemented in some P2P-SIP nodes), then it can put appropriate message body in NOTIFY to indicate off-line status.

```
SIP/2.0 200 Pending

NOTIFY <sip:alice@example.com> SIP/2.0
Subscription-State: pending
```

When Bob registers, the subscription is terminated with reason as "deactivated" so that Alice can subscribe again (procedure 6.2).

```
NOTIFY <sip:alice@example.com> SIP/2.0
Subscription-State: terminated; reason=deactivated
```

If Bob has a valid registered contact, then SIPPEER proxies the SUBSCRIBE message to the contact. If there are multiple registered locations, then the request is forked to all the locations. Once the request is proxied, the SIPPEER node steps out of the subscription path.

When Bob unregisters with P2P-SIP, he sends NOTIFY message to Alice terminating the subscription with reason as "deactivated". Alice subscribes again, and the subscription gets migrated to the responsible P2P-SIP node.

When the responsible P2P-SIP node gracefully leaves the system, it also sends NOTIFY to terminate all the subscriptions for keys stored on that node. Alternatively, the node can send the SUBSCRIBE message to the new responsible node. However, this approach requires additional logic for the node authenticating on behalf of the subscriber to the subscribee, hence not recommended.

The subscription is stored in Subscription object. Each UserAccount, A, is associated with zero or more contact locations, C^i, and zero of more subscriptions, S^j. The algorithm for handing incoming SUBSCRIBE by the responsible node is shown in procedure 6.1, and incoming REGISTER for subscription migration in procedure 6.2.

One potential problem could be as follows. Suppose Bob registers with his user agent which does not support events. So the SUBSCRIBE request will be rejected, e.g., by "501 not implemented" error code. This terminates the subscription attempt by Alice, who may not retry subscribing. To work around this problem, SIPPEER may use the OPTIONS message

Procedure 6.1 N.OnSubscribe (S:subscription object,M:request message)

```
if no A such that A.to = S.to then
  /* No valid registrations found */
  send response 202 Pending
  send NOTIFY S.from
       Subscription-State: pending
else if S was active then
  /* Terminate existing subscription first */
  send NOTIFY S.from
       Subscription-State: terminated; reason=deactivated
  delete S
else
  for all C in A.contacts do
    proxy M to C
  if a valid 2xx, 401, or 407 response is received then
    delete S
    /* proxy the response upstream */
  else
    /* do not migrate. respond locally. */
    send response 202 Pending
    send NOTIFY S.from
         Subscription-State: pending
```

Procedure 6.2 N.OnRegister (R:registration object,M:request message)

```
/* This is appended to procedure 3.16 */
for all S such that S.to = R.to do
  if S.event is not reg then
    send NOTIFY S.from
         Subscription-State: terminated; reason=deactivated
    delete S
```

to Bob to find out if Bob's user agent supports SUBSCRIBE or not. It also intercepts the SUBSCRIBE response from Bob. If Bob's user agent fails without notifying Alice, there may be delay before Alice detects and retries.

SIPPEER implements the registration event package [16] since it acts as registrar for some users. The subscription for event reg is handled locally by the SIPPEER node that is responsible for storing user registrations. This subscription does not get migrated when the user registers or unregisters. When the SIPPEER node storing the subscription is leaving the network, it terminates the subscription so that the subscriber re-subscribes to the new responsible node for the user key. **TODO: How does authentication work for this? How does subscribee approve this subscription without migration?**

6.2 PSTN interworking

The system does not allow registering a telephone number of the form sip:number@domain as an alias because that is used as node identifier to form the Chord DHT in SIPPEER. Secondly, verifying that a user owns a particular telephone number or extension is beyond the scope of P2P-SIP. **TODO: How do we verify that the user owns the number if he registers his number.**

TRIP [17] is a policy-driven inter-administrative domain protocol for advertising the reachability of telephony destinations between location servers. We do not use TRIP because (1) it adds another protocol requirement for P2P-SIP nodes, and (2) it distributes telephony destinations to all the location servers, unlike lookup service of DHT. However, the hybrid architecture of P2P-SIP can still use TRIP servers through the service provider's SIP proxy servers, independent of P2P operations.

In this section, we provide an alternate SIP-based method to map telephone number ranges to P2P-SIP nodes. One problem with this approach is that every gateway node can reach potentially every phone number in the world – at very different rates. For example, a gateway in New York could advertise calling India, but one would imagine that a gateway in Delhi would have better rates. One way to restrict advertisements is by rates (cost). For example, a gateway in New York that is also a P2P-SIP node, queries the existing rates for calling India (country code 91) using the SIP OPTIONS method.

If the rates are more than its offer, then it should advertise its own offer, otherwise not. Typically nodes will periodically query for rates with period inversely proportional to the prefix match with its local coverage. For example, a gateway in New York with phone number +1-212-939-7130 can query for rates of +91 prefix every week, +1 prefix everyday, and +1-212 prefix every three hours. Another issue is related to currency conversion. Comparing two different gateways offering rates in different currency requires timely currency conversion factor, which is beyond the scope of P2P-SIP.

The gateway registers the various telephone prefixes it can reach in the DHT. The DHT node stores only a few best offers, based of the offered rates. The prefix is queried during call setup. Given all the rate determination and currency conversion problems, the method may not be useful for calls that incur cost, but is useful for detecting gateways to reach free telephone destinations.

6.2.1 Registering a telephone prefix

SIP [20] does not allow use of tel URI [23] in the To header of REGISTER message. This is because other schemes such as tel are supposed to be resolved externally, e.g., using ENUM [4]. If the resolution results in sip or sips URI, it can be registered with the SIP registrar. However, in P2P-SIP, we allow registering a tel URI [23] or a telephone prefix. A telephone prefix is similar to a tel URI except that the digits are empty and the phone-context parameter specifies the prefix, e.g., "tel:;phone-context=+1-212". **TODO: This is not comformant with RFC 3966 because of empty number portion.** Alternatively, "sip" URI can be used as "sip:+1212939xxxx@sippeer.net;user=phone" to make it RFC 3261 compliant, but introduces an unused sippeer.net domain.

For example, a PSTN gateway implementing P2P-SIP (or its SIPPEER adaptor) that can reach area code 212 in Manhattan, New York, can register with P2P-SIP as follows:

```
REGISTER sip:sippeer.net SIP/2.0
To: <tel:;phone-context=+1-212>
From: <gw@my_gateway.com>
Contact: <sip:my_gateway.com;lr>; cost=USD.05/60s
```

The lr parameter in Contact indicates that the server should not alter the original INVITE's request-URI when proxying to this gateway, and should use mechanism for loose-routing [20] to proxy the request. This preserves the original callee destination in the request-URI. This does not apply for redirect responses. The cost parameter provide basic billing information for the gateway.

The complete tel URI in the To header including the phone-context parameter after removing any fillers such as "(,), -", is used as the key for lookup and routing in Chord-based P2P-SIP network. For example, the responsible P2P-SIP node uses "tel:;phone-context=+1212" as the user key for the above registration.

When a responsible node receives an incoming REGISTER message for the prefix, it updates the Contact list of the destination and re-computes the preference q values for the various Contact headers. The node may choose to keep only the top few contacts for the prefix based on the calculated preference. Local policy or cost metric can be used for calculating the q values.

6.2.2 Calling a telephone number

When another user wants to reach a number, say "+1-212-939-7040", the caller's user agent sends the following SIP INVITE request:

```
INVITE tel:+1-212-939-7040 SIP/2.0
To: <tel:+1-212-939-7040>
From: <sip:bob@example.com>
Request-Disposition: redirect
```

Caller preference [21] is used to indicate that the caller wants the request to be redirected, i.e., receive "302 Redirect" responses from P2P-SIP nodes. This allows the caller to collect all the responses and select the best gateway to route the call to.

The next hop node in P2P-SIP network is computed using the key as follows. The basic key, K_0, is "tel:+12129397040" derived from To header after removing any fillers. Other keys, K_{n-i}, is computed as "tel:;phone-context=+*first i digits prefix*", for i=1,2,..n-1, where n is total number of digits. For example, K_4 is "tel:;phone-context=+1212939" and K_7 is "tel:;phone-context=+1212". The INVITE request is sent to multiple next hop nodes based on keys K_i. Either sequential or parallel forking can be used. The caller may prefer K_i over K_j for $i < j$ but is not required to do so. Note that only the first P2P-SIP node (which may the caller phone or the outbound proxy of an existing SIP phone), sends multiple INVITE

requests based on the keys. It also puts the key in the request-URI of the INVITE request. Subsequent nodes proxy or respond only a single request, without modifying the request-URI.

The caller user agent should utilize any local configuration or policy information available to reduce the number of INVITE requests sent. For example, in USA, a call made to "+1-212-555-1234" may be sent to nodes with keys "tel:+12125551234", "tel:;phone-context=+1212555", "tel:;phone-context=+1212" and "tel:;phone-context=+1", but other keys can be skipped. This is because a gateway typically handles an area code, local switch or all of USA telephone subscribers.

If the P2P-SIP node proxied the call to the gateway, then the caller receives 200 success response and call setup is complete. Otherwise, the caller receives 302 redirection response with the address of the gateway. Different responses may be received for different prefixes. Three example responses are shown below:

```
SIP/2.0 302 Moved temporarily
To: <tel:;phone-context=+1212939>
Contact: <sip:phone.cs.columbia.edu;lr>; cost=0; q=1.0

SIP/2.0 302 Moved temporarily
To: <tel:;phone-context=+1212>
Contact: <sip:phone.cs.columbia.edu;lr>; cost=USD.03/60s; q=1.0
Contact: <sip:manhattan.verizon.com;lr>; cost=USD.06/60s; q=.8

SIP/2.0 302 Moved temporarily
To: <tel:;phone-context=+1>
Contact: <sip:mci.com;lr>; cost=USD.10/120s; q=1.0
Contact: <sip:att.com;lr>; cost=USD.09/60s; q=.8
```

Redirect behavior is preferred since it allows the caller to select the best destination. When multiple responses are received as shown above, the caller user agent collects all the Contacts and selects the best destination. The q parameter is assigned by the registering P2P-SIP node for all the contacts within the same telephone prefix using some local policy or based on cost metrics. The q parameters are valid within the same response, and can not be used for comparison across different responses.

In this case, the best options is sip:phone.cs.columbia.edu;lr with zero cost for prefix +1-212-939. The caller sends another INVITE with original request URI (and other headers) to this destination. **TODO: What about authentication for INVITE to telephone destinations?**

6.2.3 Incoming INVITE handling

When a SIPPEER node receives the INVITE request, it checks if the request is for a telephone destination: either "tel" URI or "sip:number@sippeer.net;user=phone". If yes, the canonical "tel" URI is formed to compute the DHT key by removing fillers and converting "sip" URI to "tel" URI. For example, "sip:+1-212-939-xxxx@sippeer.net;user=phone" is converted to "tel:;phone-context=+1212939". This is used as the key to compute the next hop in routing.

If this SIPPEER is an outbound proxy for an existing SIP client, and the destination is the global "tel" URI without any phone-context or a "sip" URI with global telephone number in the user part, then it also calculates multiple keys and sends INVITE to those next hop nodes using forking, as described earlier.

If this SIPPEER node is responsible for the key, then it collects all the contacts for this prefix, and responds with 302 response. If the caller-preference requests "proxy" behavior, then the node proxies the call to the contact with the highest q value.

6.3 Firewall and NAT traversal

We refer to firewall or NAT as a *middlebox*, and the internal network behind the middlebox as a *private* network. If a P2P-SIP node in a private network, it does not join the global DHT, but instead uses an existing global DHT node as an outbound proxy. When an existing client (C) uses a P2P-SIP node (P) as an adaptor (outbound proxy), there are three cases: (1) if both P and C are in public network, it does not involve any middlebox, (2) if P is public and C is private, then C needs to implement various middlebox traversal mechanisms, and (3) if both P and C are in private network, then P does not join the global DHT, but uses an existin DHT node as outbound proxy.

Both signaling and media traffic needs to be traversed through the middlebox. SIP signaling traversal through middlebox is handled using symmetric response routing [19] and connection reuse [11]. Interactive connectivity establishment (ICE [14]) is used in conjunction with STUN [22] and TURN [15] to enable media traversal.

6.3.1 Signaling

We describe SIP symmetric response routing and connection reuse in the context of P2P-SIP.

Suppose the P2P-SIP node is listening for SIP messages on IP interface L_{ip} and port L_p. Default SIP port is L_p=5060. When the node sends or proxies a SIP request on UDP, it uses L_{ip} and L_p as source IP and source port. It also puts the "rport" parameter in the top Via header.

When the node receives an incoming SIP request on UDP from the remote client's IP R_{ip} and port R_p, the response is sent using source IP L_{ip}, source port L_p, and destination IP R_{ip}. If the "rport" parameter is present, then destination port is R_p, else the port number in Via header is used. If the "rport" parameter is present, the response also contains the parameter, and the value is set to R_p.

For TCP, the node maintains a list of existing connections. The connection C includes the source IP, source port, destination IP and destination port; and also include an attribute indicating direction (send, receive or both). When the node has to send or proxy a SIP request on TCP to destination IP R_{ip} and portR_p, it searches for existing connection to same destination and port, with direction as send or both. If found, the connection is used to send the message. If not found, a new connection is created with direction as send, and the local interface L_{ip} and port L'_p for the TCP connection are recorded. Note that the connection specific local port L'_p is different from default listening port L_p. The node now listens for incoming responses as well as requests on this new connection. The outgoing request has the Via header containing L_{ip} and L_p and the "alias" attribute, so that the response is sent to port L'_p.

When an incoming SIP request is received on TCP from remote IP R_{ip}, remote port R_p, local IP L_{ip} and local port L_p, and the connection is not found, a new connection is created with direction "receive". If the request's top Via contains "alias" parameter, the direction is changed to "both" and remote port of the connection is changed to the port specified in the Via header. If the new request is received on an existing connection, then the "alias" parameter is ignored.

The connection is terminated if there is no activity for a timeout, e.g., one hour. If new connection needs to be created, but no more resources (e.g., socket or file descriptors) are available, the least recently used connection(s) for which no response is pending are terminated to allow new connections.

TLS is required for connection reuse, to prevent connection hijacking.

6.3.2 Media

Since media and signaling are sent on separate ports, middlebox traversal for media also needs to be established.

SIPPEER implements ICE to establish media sessions. It also implements basic STUN and TURN client and server functions. The node listens on default STUN port 3478, for both STUN and TURN requests.

Before sending a call setup request, the caller's node gathers STUN and TURN derived addresses from its registrar P2P-SIP node. It uses these addresses in SDP of INVITE as described in ICE. Callee also gathers derived addresses from it's registrar, and uses these in the SDP in the 200 response.

TODO: Describe how media over TCP is handled.

6.3.3 Integrating media and signaling

An alternative is to use the same default SIP port (i.e., 5060) for media also, and do demultiplexing at the application layer between SIP and RTP/RTCP packets. For example, the node can use this port in the SDP's "m=" media line. Since RTCP on UDP requires separate port, it may not work. Both SIP and RTP on the same TCP connection requires packetization layer for RTP/RTCP above TCP as used in RTSP [24] (sec 10.12). Moreover, all media must traverse the same path as signaling, thereby turning the SIP proxies into back-to-back user agents, which is not scalable. Integrating signaling and media works reasonably well for residential-to-gateway calls, where the gateway is outside the firewall/NAT, similar to the IAX model [1]. Comparison of this model with STUN or TURN model is for further study. In particular, whether single TCP connection has advantage over having multiple TCP and UDP connections to the same box.

We explain how to interwork between P2P-SIP of a private network with the global P2P-SIP in the next section.

6.4 Inter-domain operation

In real deployment, it is useful to allow multiple P2P-SIP networks (DHTs) to be interconnected. For example, individual large organizations can have internal P2P-SIP network which is connected to the global P2P-SIP network. In this section, we propose a two level network: the global (public) DHT represented by sippeer.net and local organization wide DHT, which may be behind firewall and NAT. The local domain-specific DHT has representative server nodes, that are reachable in the global DHT also via the domain key. Any node in domain-specific DHT can reach the global DHT also, and any node in global DHT can reach the domain-specific DHT via the representative server nodes in the domain. The global DHT

computes the index based on user identifier of the form user@domain, and if not found then domain. Local one computes the index based on user for intra-domain calls.

6.4.1 Registration

Consider the architecture shown in Fig. 19 with one global DHT (nodes P, Q, R, S) and two domain specific DHTs. Domain private.com's DHT has nodes A, B, C, D and example.com has nodes X, Y, Z, where nodes C, D and X are representative server nodes.

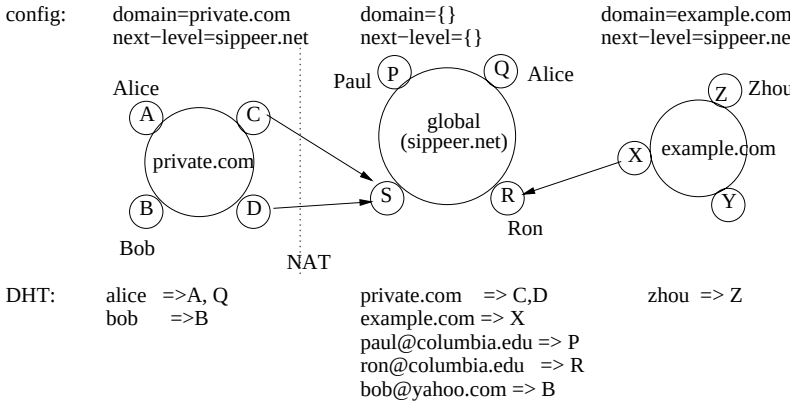

Figure 19: Inter-domain P2P-SIP

Every DHT has some bootstrap nodes identified in DNS of the domain. For example, bootstrap nodes for global DHT are identified by DNS record of sippeer.net, and those for local DHTs of private.com and example.com by their respective DNS records. When a node starts up, it uses its configured domain name and it does DNS lookup for NAPTR/SRV for SIP service of its domain, and resolves the addresses to IP addresses. If no domain name is configured, it assumes global sippeer.net domain. If any IP address matches any of the the local interface, the node assumes it is one of the bootstrap nodes for the domain. **TODO: Does the dependency on DNS cause P2P-SIP to become server-based system?** For example, private.com resolves to IP addresses of nodes C and D, where as example.com to node X.

There are two configuration properties for each node: domain and next-level. The former indicates the domain for the node, where sippeer.net indicates global DHT, whereas the latter indicates the next level DHT's domain. Bootstrap nodes in global DHT are configured with domain and next-level as empty. When the node starts up it does DNS query and detects that it should be a bootstrap node for global DHT. Representative server nodes, C and D in the private.com domain are started with domain as private.com and next-level as sippeer.net. When node C starts up, it detects that it is bootstrap node for its domain. Since C is a bootstrap node and the next-level is not empty, it registers its domain private.com in the next-level DHT via the bootstrap nodes in sippeer.net domain. The registration gets stored at appropriate global P2P node based on the key private.com. Similarly, nodes D and X register their domains in the next-level global DHT.

```
REGISTER sip:sippeer.net SIP/2.0
To: <sip:private.com>
From: <sip:C@private.com>
Contact: <sip:C_ip_address>
```

The global DHT stores the mapping that private.com is found at node C and D, whereas example.com is at node X.

When a domain-specific node, A, starts up, it discovers node C, e.g., using multicast discovery. Node A gets to know its domain and next-level parameters in the REGISTER response from node C in new headers. It then joins the domain-specific DHT in private.com domain. It also knows that it is not the bootstrap node, so it doesn't register its domain to the next level DHT. Existing clients such as X-lite do not need to understand domain and next-level parameters, because they will typically be connected to a P2P-SIP node (outbound proxy), and do not take part in DHT directly. Internal DHT nodes maintain their next-level and domain properties, and send to other new joining nodes in that DHT.

The domain-specific bootstrap nodes use the P2P-SIP nodes of next-level DHT as outbound proxy. If a domain-specific bootstrap node is in public network, it can directly join the global DHT, in addition to the domain-specific DHT.

Domain administrators may install multiple domain-specific bootstrap nodes to share load. The next-level may be configured as empty so that the domain specific bootstrap nodes do not connect to the global DHT. This allows restricting P2P-SIP calls to within a domain. Nodes may still use DNS [18] to reach outside networks directly without going through

the global DHT. Alternatively, administrators may install only bootstrap nodes in the domain as a replacement for SIP proxy and registrar of the domain. In this case, the internal SIP phones use server-based SIP architecture but the domain is connected to global DHT via P2P-SIP.

6.4.2 Call setup

When a user alice@private.com in a domain using node A, wants to call another user bob@private.com, it discovers that the domain portion of the destination is same as the domain property, so it calls Find(bob) in the domain-specific DHT. The domain-specific P2P-SIP nodes identify the domain, and build the lookup key using only the user part.

When alice@private.com wants to call paul@columbia.edu, the domains do not match, so it proxies the INVITE request to the domain's bootstrap node (C or D) resolved via DNS. Nodes C and D act as proxy to the global DHT, and perform lookup on the global DHT.

When a user paul@columbia.edu using node P in global DHT, wants to call ron@columbia.edu, the domains do not match. This is because node P is configured with domain as empty. In this case it looks up for both keys ron@columbia.edu and columbia.edu. Suppose, ron@columbia.edu is registered from node R in the global DHT, then the call is proxied to node R.

Using similar procedure, suppose paul@columbia.edu wants to call alice@private.com, then it first looks up for both alice@private.com and private.com keys in global DHT. The latter is found to be registered as nodes C and D, so the request gets proxied to C or D or both, which further proxies the request to internal node A which registered as alice@private.com. If such user identifier is not registered, the domain-specific DHT node sends back appropriate failure response, 480 or 404, to the caller.

Suppose alice@private.com on node A, wants to call zhou@example.com. The INVITE request is proxied to C, which in turn proxies to X, which then proxies to internal node Z which registered as this user.

6.4.3 Cross-domain

The system allows a user in private.com domain to register with user identifier containing another domain. For example, if user on node B registers as bob@yahoo.com, the registration should be propagated to global DHT. Similarly, a user visiting another network should be allowed to register with her home domain's DHT. We assume such cross registrations are limited in volume and are supported with appropriate authentication.

When a user on node B in domain private.com registers as identifier bob@yahoo.com, it compares the domain part, similar to call setup procedure. Since the domain does not match, the REGISTER message is proxied to the domain-specific bootstrap nodes C or D, which in turn proxies it to the global DHT.

When a user on node P in global DHT, registers as identifier alice@private.com, the REGISTER message is first sent with key as the domain private.com. If this fails, then the user key alice@private.com is used for routing. Alternatively, both can be tried in parallel, but will result in duplicate registrations. Since only a few users are expected to cross register, this is not bad.

The OPTIONS request to sip:private.com can be used before sending REGISTER for alice@private.com to detect if the domain-specific servers exist for private.com or not.

When user on node Q calls alice@private.com, it needs to send two INVITE requests, one to sip:alice@private.com and other to sip:private.com. The latter URI is not right since the nodes C or D can not tell where to proxy the request. There are two alternatives: use URI as sip:user@private.com?p2p-key=private.com or use OPTIONS method to sip:private.com to discover nodes C and D, and then send INVITE to one of those nodes with URI as sip:alice@private.com. Using p2p-key parameter reduces the call setup round-trips but looks like a hack. The problem with Q sending INVITE directly to C or D is that C or D may be behind NAT or firewall and reachable only via P or S, respectively.

TODO: Provide pseudo-code for inter-domain REGISTER and INVITE processing, both inbound and outbound, for regular nodes and bootstrap nodes.

7 Conclusions and future work

This paper gives an implementation report of the P2P-SIP architecture described in [26, 27]. We have implemented the basic registration and call setup modules in SIPPEER. We are working on NAT and firewall traversal, security, interworking with PSTN, and offline message storage and delivery.

A Message flow

This section details the example SIP message flow with all mandatory headers, when a new node joins the DHT. The message trace is captured using our implementation and is configured with 5-bits identifiers. In real world applications, the identifiers will be 32 bits or 160 bits.

Node 7 multicasts discovery REGISTER message:

```
REGISTER sip:224.0.1.75 SIP/2.0
Via: SIP/2.0/UDP 128.59.15.56:44452;branch=z9hG4bKMzWJQTgLAwAA
From: <sip:7@128.59.15.56:44452>; tag=3047841036
To: <sip:7@128.59.15.56:44452>
CSeq: 1 REGISTER
Call-ID: 978717896@128.59.15.56
Content-Length: 0
```

Node 22 responds:

```
SIP/2.0 302 Redirecting
Via: SIP/2.0/UDP 128.59.15.56:44452;branch=z9hG4bKMzWJQTgLAwAA
From: <sip:7@128.59.15.56:44452>; tag=3047841036
To: <sip:7@128.59.15.56:44452>; tag=2644643801
Call-ID: 978717896@128.59.15.56
Cseq: 1 REGISTER
Date: Wed, 03 Nov 2004 19:44:53 GMT
Server: P2P-SIP-Columbia-University/1.24
Contact: <sip:128.59.15.31:41253>
Content-Length: 0
```

Node 7 sends unicast REGISTER to node 22:

```
REGISTER sip:128.59.15.31:41253 SIP/2.0
Via: SIP/2.0/UDP 128.59.15.56:44452;branch=z9hG4bKMzWJQTgLAwAB
From: <sip:7@128.59.15.56:44452>; tag=3047841036
To: <sip:7@128.59.15.56:44452>
CSeq: 1 REGISTER
Call-ID: 591891853@128.59.15.56
Content-Length: 0
```

Node 22 => node 1.

```
REGISTER sip:128.59.15.60:35778 SIP/2.0
Via: SIP/2.0/UDP 128.59.15.31:41253;branch=z9hG4bK9DSJQdByDQAi
Via: SIP/2.0/UDP 128.59.15.56:44452;branch=z9hG4bKMzWJQTgLAwAB
From: <sip:7@128.59.15.56:44452>; tag=3047841036
To: <sip:7@128.59.15.56:44452>
CSeq: 1 REGISTER
Call-ID: 591891853@128.59.15.56
Content-Length: 0
```

Node 1 => node 22:

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 128.59.15.31:41253;branch=z9hG4bK9DSJQdByDQAi
Via: SIP/2.0/UDP 128.59.15.56:44452;branch=z9hG4bKMzWJQTgLAwAB
From: <sip:7@128.59.15.56:44452>; tag=3047841036
```

```
To: <sip:7@128.59.15.56:44452>; tag=1664201056
Call-ID: 591891853@128.59.15.56
Cseq: 1 REGISTER
Date: Wed, 03 Nov 2004 19:44:53 GMT
Server: P2P-SIP-Columbia-University/1.24
Contact: <sip:10@128.59.15.55:36550>; predecessor=sip:1@128.59.15.60:35778
Content-Length: 0
```

Node 22 => node 7:

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 128.59.15.56:44452;branch=z9hG4bKMzWJQTgLAwAB
From: <sip:7@128.59.15.56:44452>; tag=3047841036
To: <sip:7@128.59.15.56:44452>; tag=1664201056
Call-ID: 591891853@128.59.15.56
Cseq: 1 REGISTER
Date: Wed, 03 Nov 2004 19:44:53 GMT
Server: P2P-SIP-Columbia-University/1.24
Contact: <sip:10@128.59.15.55:36550>; predecessor=sip:1@128.59.15.60:35778
Content-Length: 0
```

Node 7 => Node 10:

```
REGISTER sip:128.59.15.55:36550 SIP/2.0
Via: SIP/2.0/UDP 128.59.15.56:44452;branch=z9hG4bKMzWJQTgLAwAC
From: <sip:7@128.59.15.56:44452>; tag=3047841036
To: <sip:11@sippeer.net>
CSeq: 1 REGISTER
Call-ID: 368501868@128.59.15.56
Content-Length: 0
```

Node 10 => Node 7:

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 128.59.15.56:44452;branch=z9hG4bKMzWJQTgLAwAC
From: <sip:7@128.59.15.56:44452>; tag=3047841036
To: <sip:11@sippeer.net>; tag=1829672944
Call-ID: 368501868@128.59.15.56
Cseq: 1 REGISTER
Date: Wed, 03 Nov 2004 19:44:53 GMT
Server: P2P-SIP-Columbia-University/1.24
Contact: <sip:15@128.59.15.48:34391>; predecessor=sip:10@128.59.15.55:36550
Content-Length: 0
```

Node 7 => Node 10:

```
REGISTER sip:128.59.15.55:36550 SIP/2.0
Via: SIP/2.0/UDP 128.59.15.56:44452;branch=z9hG4bKMzWJQTgLAwAD
From: <sip:7@128.59.15.56:44452>; tag=3047841036
To: <sip:23@sippeer.net>
CSeq: 1 REGISTER
Call-ID: 3390599958@128.59.15.56
Content-Length: 0
```

Node 10 => Node 7:

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 128.59.15.56:44452;branch=z9hG4bKMzWJQTgLAwAD
From: <sip:7@128.59.15.56:44452>; tag=3047841036
To: <sip:23@sippeer.net>; tag=2644643801
```

```
Call-ID: 3390599958@128.59.15.56
Cseq: 1 REGISTER
Date: Wed, 03 Nov 2004 19:44:53 GMT
Server: P2P-SIP-Columbia-University/1.24
Contact: <sip:1@128.59.15.60:35778>; predecessor=sip:22@128.59.15.31:41253
Content-Length: 0
```

The JOIN is complete now. This node 7 has become a super-node.
Now, node 7 updates its predecessor and successor as part of stabilization process.
Node 7 => Node 10:

```
REGISTER sip:128.59.15.55:36550 SIP/2.0
Via: SIP/2.0/UDP 128.59.15.56:44452;branch=z9hG4bKMzWJQTgLAwAF
From: <sip:7@128.59.15.56:44452>; tag=3047841036
To: <sip:7@128.59.15.56:44452>
CSeq: 1 REGISTER
Call-ID: 4170442560@128.59.15.56
Expires: 3600
Contact: <sip:7@128.59.15.56:44452>; predecessor=
Content-Length: 0
```

Node 10 => Node 7:

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 128.59.15.56:44452;branch=z9hG4bKMzWJQTgLAwAF
From: <sip:7@128.59.15.56:44452>; tag=3047841036
To: <sip:7@128.59.15.56:44452>; tag=1829672944
Call-ID: 4170442560@128.59.15.56
Cseq: 1 REGISTER
Date: Wed, 03 Nov 2004 19:44:53 GMT
Server: P2P-SIP-Columbia-University/1.24
Contact: <sip:15@128.59.15.48:34391>; q=1; predecessor=sip:10@128.59.15.55:36550
Contact: <sip:22@128.59.15.31:41253>; q=0.6
Contact: <sip:1@128.59.15.60:35778>; q=0.4
Content-Length: 0
```

Node 7 => Node 10:

```
REGISTER sip:128.59.15.55:36550 SIP/2.0
Via: SIP/2.0/UDP 128.59.15.56:44452;branch=z9hG4bKMzWJQTgLAwAH
From: <sip:7@128.59.15.56:44452>; tag=3047841036
To: <sip:10@128.59.15.55:36550>
CSeq: 1 REGISTER
Call-ID: 1037968559@128.59.15.56
Expires: 3600
Contact: <sip:7@128.59.15.56:44452>; predecessor=sip:10@128.59.15.55:36550
Content-Length: 0
```

Node 10 => Node 7:

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 128.59.15.56:44452;branch=z9hG4bKMzWJQTgLAwAH
From: <sip:7@128.59.15.56:44452>; tag=3047841036
To: <sip:10@128.59.15.55:36550>; tag=1829672944
Call-ID: 1037968559@128.59.15.56
Cseq: 1 REGISTER
Date: Wed, 03 Nov 2004 19:44:58 GMT
Server: P2P-SIP-Columbia-University/1.24
Contact: <sip:10@128.59.15.55:36550>; q=1; predecessor=sip:7@128.59.15.56:44452
Contact: <sip:15@128.59.15.48:34391>; q=0.8
```

```
Contact: <sip:22@128.59.15.31:41253>; q=0.6
Contact: <sip:1@128.59.15.60:35778>; q=0.4
Content-Length: 0
```

Now, it refreshes the finger table entries.
Node 7 => Node 10:

```
REGISTER sip:128.59.15.55:36550 SIP/2.0
Via: SIP/2.0/UDP 128.59.15.56:44452;branch=z9hG4bKMzWJQTgLAwAI
From: <sip:7@128.59.15.56:44452>; tag=3047841036
To: <sip:11@sippeer.net>
CSeq: 1 REGISTER
Call-ID: 3698689758@128.59.15.56
Content-Length: 0
```

Node 10 => Node 7:

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 128.59.15.56:44452;branch=z9hG4bKMzWJQTgLAwAI
From: <sip:7@128.59.15.56:44452>; tag=3047841036
To: <sip:11@sippeer.net>; tag=1829672944
Call-ID: 3698689758@128.59.15.56
Cseq: 1 REGISTER
Date: Wed, 03 Nov 2004 19:44:58 GMT
Server: P2P-SIP-Columbia-University/1.24
Contact: <sip:15@128.59.15.48:34391>; predecessor=sip:10@128.59.15.55:36550
Content-Length: 0
```

Node 7 => Node 10:

```
REGISTER sip:128.59.15.55:36550 SIP/2.0
Via: SIP/2.0/UDP 128.59.15.56:44452;branch=z9hG4bKMzWJQTgLAwAJ
From: <sip:7@128.59.15.56:44452>; tag=3047841036
To: <sip:23@sippeer.net>
CSeq: 1 REGISTER
Call-ID: 661685467@128.59.15.56
Content-Length: 0
```

Node 10 => Node 7:

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 128.59.15.56:44452;branch=z9hG4bKMzWJQTgLAwAJ
From: <sip:7@128.59.15.56:44452>; tag=3047841036
To: <sip:23@sippeer.net>; tag=2644643801
Call-ID: 661685467@128.59.15.56
Cseq: 1 REGISTER
Date: Wed, 03 Nov 2004 19:44:58 GMT
Server: P2P-SIP-Columbia-University/1.24
Contact: <sip:1@128.59.15.60:35778>; predecessor=sip:22@128.59.15.31:41253
Content-Length: 0
```

References

[1] Inter-asterisk exchange (iax) voice over ip protocol. http://www.asterisk.org.

[2] SIP library api in C++. http://www.cs.columbia.edu/~kns10/software/siplib.

[3] E. Adar and B. A. Huberman. Free riding on gnutella. *First Monday*, 5(10), Oct. 2000.

[4] P. Faltstrom. E.164 number and DNS. RFC 2916, Internet Engineering Task Force, Sept. 2000.

[5] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. J. Leach, A. Luotonen, and L. Stewart. HTTP authentication: Basic and digest access authentication. RFC 2617, Internet Engineering Task Force, June 1999.

[6] M. Gupta, P. Q. Judge, and M. Ammar. A reputation system for peer-to-peer networks. In *ACM NOSSDAV 2003*, June 2003.

[7] E. Guttman, C. E. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. RFC 2608, Internet Engineering Task Force, June 1999.

[8] S. Kamvar, M. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in P2P networks. In *International World Wide Web Conference (WWW)*, Budapest, Hungary, May 2003. International World Wide Web Conference Committee.

[9] S. Lee, R. Sherwood, and S. Bhattacharjee. Cooperative peer groups in NICE. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, Mar. 2003.

[10] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *ACM Conf. on Principles of Distributed Computing (PODC)*, Monterey, CA, USA, July 2002. ACM.

[11] R. Mahy. Connection reuse in the session initiation protocol (SIP). Internet Draft draft-ietf-sip-connect-reuse-00, Internet Engineering Task Force, Aug. 2003. Work in progress.

[12] D. Milojicic, V. Kalogeraki, R. M. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-peer computing. technical report HPL-2002-57 20020315, Technical Publications Department, HP Labs Research Library, Mar. 2002. http://www.hpl.hp.com/techreports/2002/HPL-2002-57.html.

[13] A. B. Roach. Session initiation protocol (sip)-specific event notification. RFC 3265, Internet Engineering Task Force, June 2002.

[14] J. Rosenberg. Interactive connectivity establishment (ICE): a methodology for network address translator (NAT) traversal for the session initiation protocol (SIP). Internet Draft draft-ietf-mmusic-ice-00, Internet Engineering Task Force, Oct. 2003. Work in progress.

[15] J. Rosenberg. Traversal using relay NAT (TURN). Internet Draft draft-rosenberg-midcom-turn-03, Internet Engineering Task Force, Oct. 2003. Work in progress.

[16] J. Rosenberg. A session initiation protocol (sip) event package for registrations. RFC 3680, Internet Engineering Task Force, Mar. 2004.

[17] J. Rosenberg, H. F. Salama, and M. Squire. Telephony routing over IP (TRIP). RFC 3219, Internet Engineering Task Force, Jan. 2002.

[18] J. Rosenberg and H. Schulzrinne. Session initiation protocol (SIP): locating SIP servers. RFC 3263, Internet Engineering Task Force, June 2002.

[19] J. Rosenberg and H. Schulzrinne. An extension to the session initiation protocol (SIP) for symmetric response routing. RFC 3581, Internet Engineering Task Force, Aug. 2003.

[20] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. R. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: session initiation protocol. RFC 3261, Internet Engineering Task Force, June 2002.

[21] J. Rosenberg, H. Schulzrinne, and P. Kyzivat. Caller preferences for the session initiation protocol (sip). RFC 3841, Internet Engineering Task Force, Aug. 2004.

[22] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN - simple traversal of user datagram protocol (UDP) through network address translators (nats). RFC 3489, Internet Engineering Task Force, Mar. 2003.

[23] H. Schulzrinne. The tel uri for telephone numbers. RFC 3966, Internet Engineering Task Force, Dec. 2004.

[24] H. Schulzrinne, A. Rao, and R. Lanphier. Real time streaming protocol (RTSP). RFC 2326, Internet Engineering Task Force, Apr. 1998.

[25] H. Schulzrinne and J. Rosenberg. Internet telephony: Architecture and protocols – an IETF perspective. *Computer Networks and ISDN Systems*, 31(3):237–255, Feb. 1999.

[26] K. Singh and H. Schulzrinne. Peer-to-peer internet telephony using SIP. In *New York Metro Area Networking Workshop*, New York, NY, Sep 2004.

[27] K. Singh and H. Schulzrinne. Peer-to-peer Internet telephony using SIP. Technical Report CUCS-044-04, Department of Computer Science, Columbia University, New York, NY, Oct. 2004.

[28] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *Electronic Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, USA, Mar 2002. IEEE.

[29] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, San Diego, CA, USA, Aug 2001.

[30] X. Wu and H. Schulzrinne. sipc, a multi-function SIP user agent. In *7th IFIP/IEEE International Conference, Management of Multimedia Networks and Services (MMNS)*, pages 269–281. IFIP/IEEE, Springer, Oct. 2004.

[31] L. Xiong and L. Liu. Peertrust: Supporting reputation-based trust for peer-to-peer electronic communities. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):843–857, July 2004.