A Distributed SIP P2P Telephony System

Iyadh Sidhom

A Thesis

In

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science (Electrical Engineering) at
Concordia University
Montreal, Quebec, Canada

April 2008

Your file Votre référence
ISBN: 978-0-494-42342-4
Our file Notre référence
ISBN: 978-0-494-42342-4

ABSTRACT

A Distributed SIP P2P Telephony System

Iyadh Sidhom

Since the telephone was invented, it has revolutionalized the telecommunications and enabled people to talk to each other over long distances. Through the years, the telephone has been greatly improved and many new models have been developed to serve the different needs of the diversified set of users.

Peer-to-peer systems inherently have high scalability, robustness and fault tolerance because there is no centralized server and the network self-organizes itself. These features can be used in order to construct peer-to-peer internet telephony systems where there is no central server and with a lower cost than the current internet telephony systems.

In this thesis, we propose a fully distributed peer-to-peer architecture, called SIP2P, for the Session Initiation Protocol (SIP). Peers connect directly to other peers, constructing an overlay network of peers which communicate to each other in order to provide the same services as the central server in the conventional SIP. The nodes that define the overlay not only act as an ordinary SIP user agent but they can also perform all together the role of the SIP proxy, SIP register and the SIP location service. The behavior of the conventional SIP servers is distributed over all the nodes participating in the overlay.

An implementation of the SIP2P system concludes the thesis in order to validate and evaluate its distributed design.

Table of contents

List of figures

List of tables

List of acronyms

DHT:	Decentralized Hash Table
NAT:	Network Address Translation
P2P:	Peer-to-peer
SC:	Skype Client
SN:	Super Node
SIP:	Session Initiation Protocol
TCP:	Transmission Control Protocol
UA:	User Agent
UDP:	User Datagram Protocol
VoIP:	Voice over Internet Protocol
XML:	Extensible Markup Language

1 Introduction

Telephone has been invented in 1877 by Alexander Graham Bell. Ever since it was invented, it revolutionalized telecommunications, enabling people to talk to each other over long distances.

Through the years, it has been greatly improved and many new models have been developed to serve the different needs of the diversified set of users. From these models, we have first seen the development of the circuit switched networks such as PSTN. Next, we saw the move to the packet switched networks such as VoIP and more recently there has been a lot of efforts to make the move to the peer-to-peer (P2P) network [44]. In this thesis, we focus on presenting an architecture model of a fully distributed P2P telephony system.

Current Internet telephony client-server architectures based on IETF's Session Initiation Protocol (SIP [37][34]) or ITUT recommendation H.323 [43] typically use a registration server for every domain. On the one hand, the main part of the system cost is with the maintenance and the configuration setting, typically by a dedicated system administrator in the domain. It is also difficult to quickly deploy a client-server system in a small environment (e.g., for emergency communications or at a conference). On the other hand, P2P systems [26] are inherently scalable and reliable because of the lack of a single point of failure and they are also easy to set up. The purest form of a P2P system has no concept of servers. All participants are equal peers and communicate in a distributed environment, to achieve a given objective such as locating data files or users [40].

All computer systems can be classified into either centralized and distributed (Figure 1.1). Distributed systems can be further classified into client-server models and the P2P models. The client-server model can be flat and such that all clients communicate with a single server (possibly replicated for improved reliability), or it can be hierarchical for improved scalability. In a hierarchal model, the servers of one level are acting as clients of the higher level servers. Examples of a flat model include traditional middleware solutions, such as object request brokers and distributed objects. Examples of a hierarchical model include DNS server and mounted file systems [28].

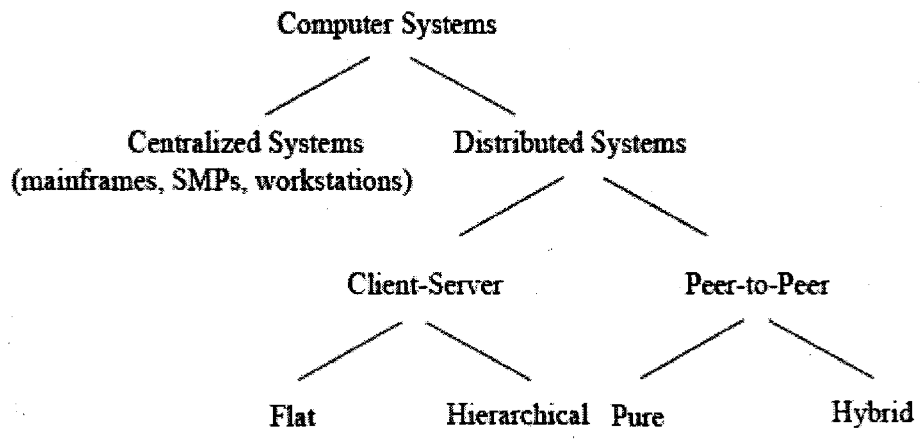


Figure 1.1 : Taxonomy of computer systems.

A P2P model can be either pure or hybrid. In a pure model, there is no centralized server. Examples of pure P2P models include Gnutella [15] and Freenet [26]. In a hybrid model, a server is first approached to obtain some meta-information, such as the identity of a peer on which some information is stored, or to verify some security credentials (Figure 1.2 (a)). From then on, a P2P communication is performed (Figure 1.2 (b)). Examples of hybrid models include Napster, Groove [17] and Magi [24]. There are also intermediate solutions with SuperPeers, such as KaZaa [19] or Skype [42]. SuperPeers contain some of the information that other peers may not have. Peers typically lookup information at SuperPeers if they cannot find the information they are looking for.

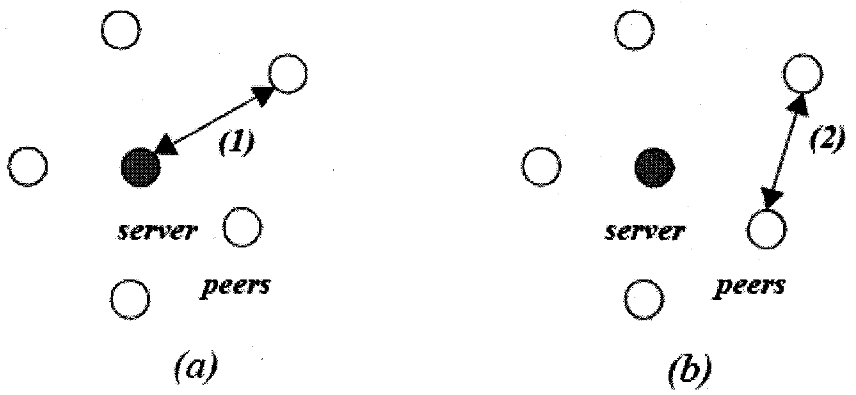


Figure 1.2: Hybrid peer-to-peer model

Peer-to-peer (P2P) networks have traditionally been used for file and information sharing in particular and for resource sharing in general. The key idea behind the development of P2P systems is to distribute processing and bandwidth requirements by sharing resources across many different peers. This idea has been extended by Skype to demonstrate the possibility of extending P2P networks to voice services. While Skype uses many P2P techniques [5], it is proprietary and closed. Though, there are other systems that explore inherently peer-to-peer VoIP protocols such as SIP. The work of Kundan *et al* [40] and Bryan *et al* [7] demonstrate the integration of the P2P Chord [29] algorithm within SIP [34]. However, such integration through SIP is open, the P2P structure which relies on Chord has been hard coded in the form of SIP headers. Therefore, it does not offer an appropriate flexibility in order to integrate new structures and consequently, it is not open to dynamic changes as required by the heterogeneous P2P voice systems.

Hence, a separation of P2P properties from the underlying voice and transport protocols is desirable for heterogeneous P2P voice systems in order to allow the use of future P2P structures.

Let us discuss about the research project where we propose a new P2P VoIP system called SIP2P. The thesis is devoted to the design of the architecture and the implementation of the SIP2P telephony system using SIP protocol. In the SIP2P system, we propose to layer P2P overlays and to separate the P2P related issues from the underlying voice and the transport layer. However, these overlays are logical overlays and can be realized with one or more physical overlays.

Unlike P2P, the conventional SIP has a client-server architecture. When a user Carl wants to start his user agent (UA), he must register with the SIP register indicating the IP address of his user agent. The SIP location service stores the mapping between the SIP ID: Carl@concordia.ca and the IP address of Carl's UA.

If another user wants to make a call to Carl, his user agent must send a request to the home server which proxies it to Carl's UA (Figure 1.3).

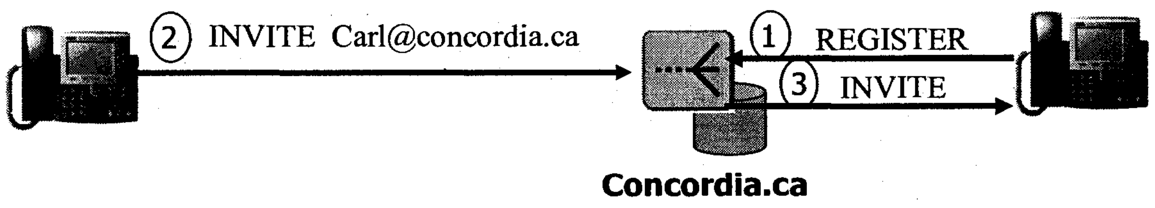


Figure 1.3: Conventional SIP call flow

In the SIP2P system architecture that we propose in this thesis, we do not require any central server. Peers connect directly to other peers, constructing an overlay network of peers which communicate to each other in order to provide the same services as the central server in the conventional SIP. The nodes that define the overlay not only act as an ordinary SIP user agent but they can also perform all together the role of the SIP proxy, SIP register and the SIP location service. The behavior of the conventional SIP servers is distributed over all the nodes participating in the overlay.

The main contributions of this thesis are:

- The design of a pure P2P system telephony with a scalable and decentralized architecture;
- An architecture where there is a separation between the P2P and the SIP layer to avoid changes in SIP protocol;
- Implementation and validation of the proposed SIP2P system.

The thesis is organized as follows. Chapter 2 contains a description of the main existing P2P telephony systems, i.e., Skype [42] and SoSimple [9] and their variants. Chapter 3 is a review of the four most used decentralized location algorithms in P2P networks. Chapter 4 is the core part of the thesis and contains the architecture description of the new proposed fully distributed SIP2P telephony system. Chapter 5 contains the implementation and evaluation of the proposed SIP2P system. Finally, Chapter 6 contains the conclusion and future work.

2 Existing P2P telephony systems

In this chapter, we analysis the two main P2P telephony systems. The first one, Skype [42], is a hybrid P2P system and the second one, SoSimple [9], is a full decentralized P2P system. At the end of the Chapter, we briefly discuss one of their variants.

2.1 Skype

Skype is a VoIP P2P telephony system developed by Kazaa [19] in 2003 that allows its users to make calls and send text messages to other Skype's users. In essence, it is similar to MSN messenger and Yahoo IM, since it has the capability to make voice calls, transmit instant messages and to set voice conferences. Though, the protocols and the core architecture are totally different.

The main difference between Skype and other VoIP clients is that Skype operates on a peer-to-peer model rather than on the more traditional server-client model. The Skype user directory is entirely decentralized and distributed among the nodes in the network, which means that the network can scale very easily to large sizes without a complex and costly centralized infrastructure. Skype also routes calls through other peers in the network, which allows it to traverse NATs and firewalls, unlike most other VoIP programs.

2.1.1 Skype architecture

We present here an overview of the Skype architecture, the reader is referred to [6] for more details.

There are two types of nodes in Skype, the ordinary node and the super nodes (SN).

The ordinary nodes are hosts where a Skype client is installed. They have the capability to make voice communications and send text messages.

The super nodes (SN) are link points between the ordinary nodes and the dedicated P2P network. Any ordinary node with a public IP address and sufficient memory, CPU and bandwidth can become a super node.

An ordinary host must connect to a super node and must also register itself to a Skype login server successfully. Although, the Skype server is not part of the dedicated P2P network, it is an important entity in the Skype system. User names and their passwords are stored on the Skype server. Also, this server ensures the user authentication and the uniqueness of the user names in the P2P overlay. Except for the server login, there is no other centralized entity in Skype.

2.1.2 Skype functions

This section discusses the four most important functions of Skype which are the startup, login, NAT discover and conference functions.

2.1.3 Startup

When a Skype client wishes to connect for the first time after a Skype setup, it sents a HTTP 1.1 "GET request" to the Skype server (skype.com). This request informs the server that the client is connecting for the first time.

During the subsequent startups, the client sends only a HTTP 1.1 to the Skype server to determine if a new version is available.

2.1.4 Login

Login is a very important operation for the Skype functioning. During the login process, a Skype client authenticates its user names and passwords with the login server. Also, it announces its presence to other peers and its buddy list, i.e., the contact list, determines if it is behind a NAT or a firewall and discovers the online super peer nodes.

When a node wants to join the Skype network, it first sends an UDP packet to the first entry of the host cache which contains a list made of a super node IP address and a port pairs that the Skype client builds and refreshes regularly. If there is no response after five seconds, the Skype client tries to establish a TCP connection with this entry at port 80 (HTTP port). If there is a failure again, the client tries to connect to port 443 (HTTPS port). Then the Skype client waits for a few seconds and repeats the whole process four more times. If still unsuccessful, it reports a login failure error (Figure 2.1).

After a Skype client has been connected to a super node, it must authenticate the user name and password with the Skype login server which is the only centralized component in Skype. The login server stores user names and passwords and ensures the uniqueness of the user name.

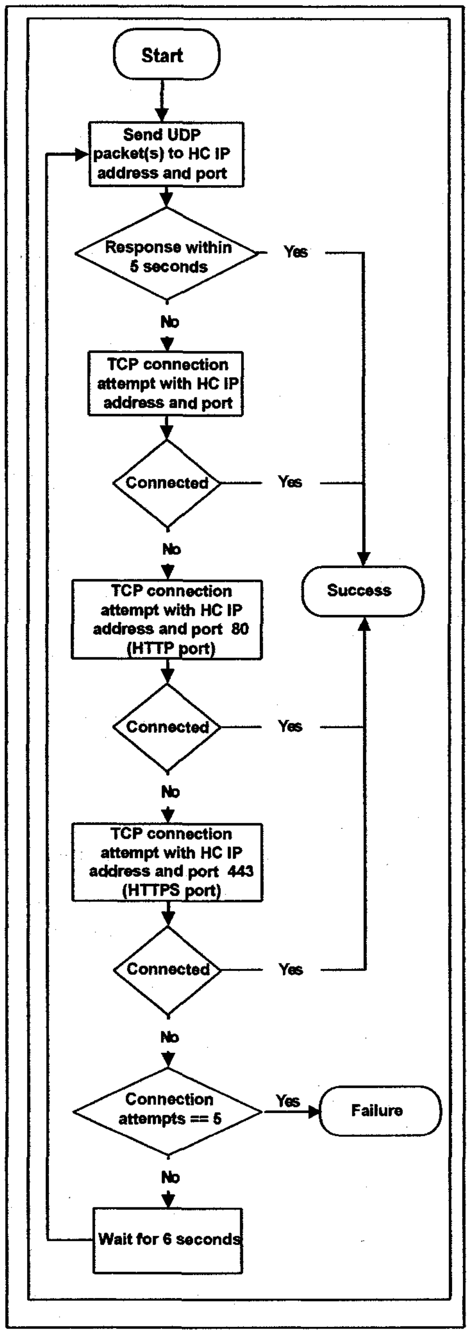

Start
Send UDP packet(s) to HC IP address and port
Response within 5 seconds
Yes
No
TCP connection attempt with HC IP address and port
Connected
Yes
No
TCP connection attempt with HC IP address and port 80 (HTTP port)
Success
Connected
Yes
No
TCP connection attempt with HC IP address and port 443 (HTTPS port)
Connected
Yes
No
Connection attempts == 5
Yes
Failure
No
Wait for 6 seconds

Figure 2.1: Skype login process [6].

2.1.5 First time login

As described in [6], “the host cache of the Skype client is empty upon installation. Thus, a Skype client must connect to well known Skype nodes in order to log on to the Skype network. It does so by sending UDP packets to some bootstrap super nodes and then waits for their response over UDP for some time. It is not clear how Skype client selects among bootstrap super nodes to send UDP packets to. Skype client then established a TCP connection with the bootstrap super node that responded. Since more than one node can respond, a Skype client can establish a TCP connection with more than one bootstrap node. However, a Skype client must maintain a TCP connection with at least one bootstrap node and can close the TCP connections with other nodes. After exchanging some packets with a bootstrap super node over TCP, it then eventually acquires the address of the login server (80.160.91.11). SC then establishes a TCP connection with the login server, exchanges authentication information with it, and finally closes the TCP connection. The initial TCP data exchange with the bootstrap super node and the login server shows the existence of a challenge-response mechanism [6]”.

2.1.6 NAT and firewall identification

Skype client is able to determine at login if it is behind a NAT or a firewall. There are at least two ways in which a Skype client can determine this information. One possibility is to determine this information by exchanging messages with its super nodes using a variant of the STUN [32] protocol. Another possibility is that during login, a Skype client sends and possibly receives data from some nodes after it has made a TCP connection with the super node. Once determined, the Skype client stores this information in the Windows registry.

2.1.7 Conference

Skype also permits conference communications. In this case, the most powerful machine will always get elected as the conference host and mixer.

2.2 SoSimple

SoSimple [9] is a fully decentralized P2P telephony system. Its design is based on currently available open-source software. SoSimple combine SIP/SIMPLE of IETF standards for VoIP and IM with a distributed hash table (DHT) protocol to make a decentralized P2P telephony system with self organizing properties.

SoSimple has many advantages like creating a fully distributed voice system and preserving many advantages of a centralized network. However, there is a big drawback with the integration of the DHT protocol in the SIP layer. This renders the architecture system inflexible and not easy to adapt to new DHT protocols.

System architecture

SoSimple suggests a valid email address to identify the user name, owing to the unique nature of email addresses and uses Chord as a DHT protocol.

As there is no more centralized server in the network such as proxy or registrar, resource location should be modified to adapt to these new changes. Instead of always using the outbound proxy for new calls, UAs instead consult the overlay to locate resources, and connect directly. UAs joining directly must support SIP register messages, store mappings between SIP addresses and IP addresses and maintain expirations for them. Nodes act both as UAs and as proxies simultaneously. Collectively, the peers replace the functionality of SIP registrars and proxies, each node being responsible for both roles for some parts of the overlay.

To get ride of the central proxy and meet several of P2P requirements, the registration process is modified by changing where registration messages are sent. The UA constructs a SIP REGISTER message containing its contact information; the endpoint hashes the username, and sends the SIP message embedded in a P2P message using the overlay. Upon arrival, the message is extracted and a reply is sent. To meet the loss intolerance and persistence requirements, the REGISTER is also sent to the *k* successors nodes in the overlay. Each node now serves the function of registrar, and knows where some users can be contacted.

Figure 2.2 illustrates an example of an endpoint joining the SoSimple network. Alice starts a SoSimple enabled UA at 1.1.1.1:5060 and connects to the overlay.

Assume her username, alice@alice.com hashes to node *a*, and the *k=2* successor nodes are nodes *b* and *c*. Alice embeds a SIP REGISTER in a P2P message and transmits it to each of these two nodes using the overlay. These 3 nodes now store a mapping from alice@alice.com to 1.1.1.1:5060 [9].

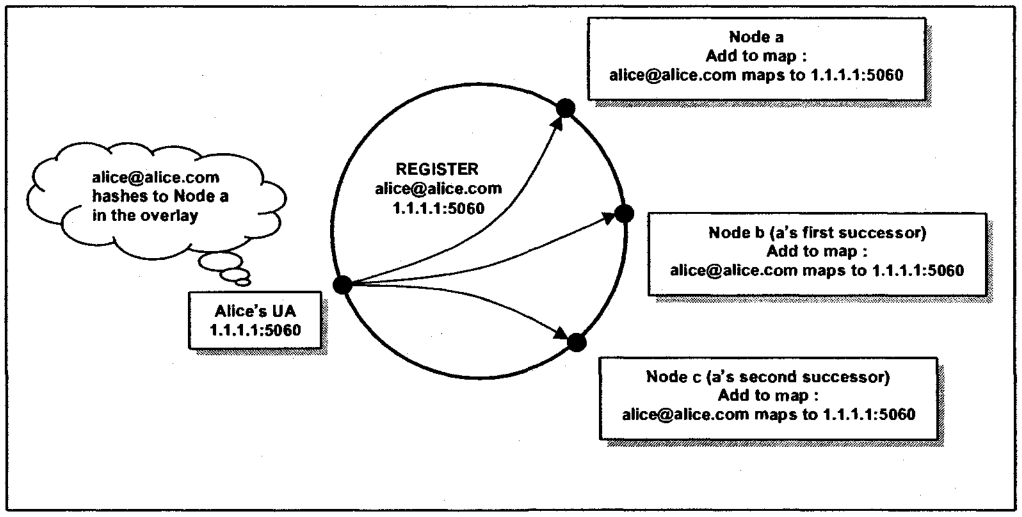

Node a
Add to map :
alice@alice.com maps to 1.1.1.1:5060
alice@alice.com
hashes to Node a
in the overlay
REGISTER
alice@alice.com
1.1.1.1:5060
Node b (a's first successor)
Add to map :
alice@alice.com maps to 1.1.1.1:5060
Alice's UA
1.1.1.1:5060
Node c (a's second successor)
Add to map :
alice@alice.com maps to 1.1.1.1:5060

Figure 2.2: Joining request example [9]

According to Bryan *et al* [9], Message routing is similarly modified to use the overlay. Rather than sending the SIP messages to a proxy, the UA hashes the destination user name. The message is embedded in a P2P message and sent to the destination using the overlay.

Returning to the previous example, let us assume that Bob wishes to send a SIMPLE MESSAGE to Alice. Bob hashes Alice's ID and sends the message using the overlay. A SIP redirect, including Alice's address, is embedded in a P2P message and sent back to Bob using the overlay. Bob sends the message (and future messages) to that address using a direct SIP message. Alice's responses are also sent directly using standard SIP messages. In order to meet the completeness requirement, if Alice has not been registered, a SIP *"404 Not Found"* would have been embedded in the P2P response that Bob received instead [9].

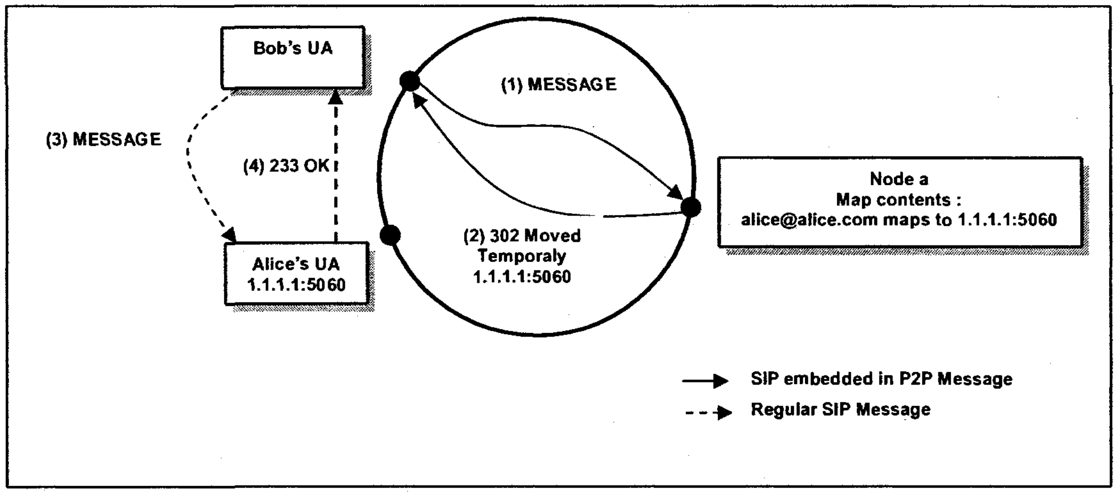


Figure 2.3: Message example [9]

2.3 A Skype variant

Kundan *et al* [40] propose a P2P telephony system using the SIP [34] protocol. Their system is distributed but they use Super peer as Skype. Their integration through SIP is open, the P2P structure which relies on Chord has been hard coded in the form of SIP headers. Therefore, it does not offer an appropriate flexibility in order to integrate new structures and consequently, it is not open to dynamic changes required by heterogeneous P2P voice systems.

3 Location algorithms in peer to peer networks

Peer-to-Peer systems and applications are distributed systems without any centralized control or hierarchical organization, in which each node runs software with the same functionalities. A review of the features of recent peer-to-peer applications yields a long list: redundant storage, permanence, selection of nearby servers, anonymity, search, authentication, and hierarchical naming. Despite this rich set of features, the core operation in most peer-to-peer systems is the efficient location of data items.

3.1 Centralized directory model

This model was made popular by Napster. The peers of the community are connected to a central directory where they publish information about the contents which they offer for sharing (Figure 3.1). Moreover, the central directory stores all the contents' location. Upon a request from a peer, the central index matches the request with the best peer in its directory. The best peer could be the one which is the least expensive, fastest or most available, according to the users' needs. Then, a file exchange occurs directly between the two peers. This model requires the control and maintenance of the infrastructure (the directory server) which hosts information on all the participants of the community. This can cause some limits of scalability because it requires larger servers when the number of requests increase and a greater storage when the number of users increase.

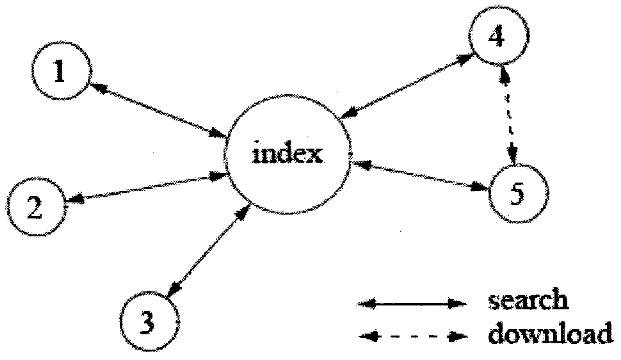


Figure 3.1: Central index algorithm

3.2 Flooded request model

The flooded request model is different from the central index one. It is a model of pure P2P in which no advertisement of the shared resources occurs. Instead, each peer request is flooded (broadcast) to the peers directly connected, that themselves broadcast to their peers and so on, until the request is answered or a maximum number of hops is reached (Figure 3.2). This model, used by Gnutella [15], requires a lot of bandwidth and is not scalable. However, it is effective in limited communities such as a company network. To avoid the scalability issue, some companies have developed super-peer client software, which concentrates a good number of requests. This leads to a lower need of bandwidth, at the price of a higher CPU consumption. Also, caching the recent search requests is used to improve scalability.

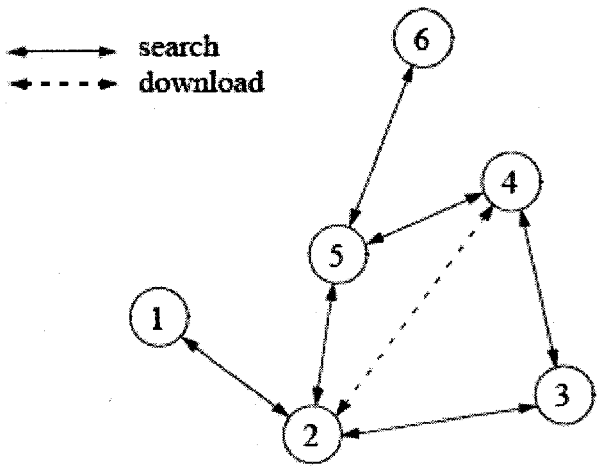


Figure 3.2: Flooded request model

3.3 Decentralized hash table (DHT) model

The decentralized hash table model, used by Freenet [14], is the most recent approach. Each peer from the network is assigned a random ID and each peer is assumed to know a given number of peers (Figure 3.3). When a document is published (shared) on such a system, an ID is assigned to the document based on hashed information depending of its name and contents. Each peer routes the document towards the peer with the ID which is the most similar to the document ID. This process is repeated until the nearest peer ID is the current peer. Each routing operation makes sure that a local copy of the document is kept. When a peer requests a document from the P2P system, the request goes to the peer with the ID most similar to the document ID. This process is repeated until a copy of the document is found. Then the document is transferred back to the request originator, while each peer participating in the routing of the document will keep a local copy. Although the decentralized hash table model is very efficient for large communities, it has the drawback that the document ID must be known before posting a request for a given document. Therefore, it is more difficult to implement a search than in the flooded request model. There are essentially four algorithms which have implemented a model with a decentralized hash table: Chord[29], CAN [31], Tapestry [45], and Pastry[13]. Their goals are similar. The primary goal is to reduce the number of P2P nodes which must be considered in order to locate a document and to reduce the amount of routing states that must be maintained by each peer. Each of the four algorithms has a guaranteed logarithmic complexity with respect to the size of the peer community. The differences among the algorithms are rather limited. However, each one is appropriate to a slightly different environment. In the next section, we will recall the key features of these system.

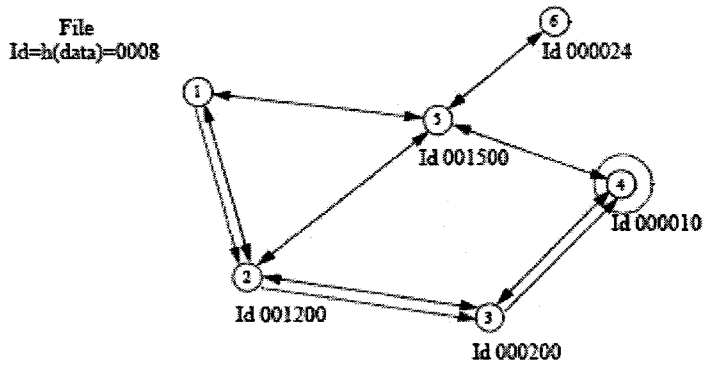


Figure 3.3: Decentralized hash table (DHT) model

3.4 Chord

Chord [29] is the algorithm that we have selected for the proposed SIP2P telephony system. The main motivations for its selection are with respect to its reliability and efficiency. Hence we will give more details about this algorithm.

3.4.1 Overview

Chord [29] is a peer-to-peer protocol which presents a new approach to the problem of identifying efficiently the location of a document or a user. Chord uses routed queries to locate a key with a small number of hops, which remains small even if the system contains a large number of nodes.

The most important features that distinguish Chord from the other DHT algorithm are its simplicity, its provable performance and correctness.

Basically, Chord protocol supports just one operation: given a key, it maps the key onto a node.

Chord uses a variant of consistent hashing [18], to assign keys to the nodes of Chord. Consistent hashing tends to balance the load, since each node receives the same number of keys and comprises relatively few moves of the keys when the nodes join and leave the system.

Previous works on consistent hashing assumed that the nodes were aware of the majority of the other nodes in the system, making it impractical to support a large number of nodes. In contrast, each node of Chord needs information of routing on only some other nodes.

In the steady state, in a system with N nodes, each node maintains information about only O(log N) nodes and resolves all the lookups by sending O(log N) messages to other nodes. Chord maintains its routing information as the nodes join and leave the system. Chord ensures that the maintenance event do not exceed O($\log^2$ N) messages.

While Chord maps keys onto nodes, the traditional location services provide a direct mapping between the keys and the values. A value can be an address, a document or an arbitrary data item. Chord can easily implement this functionality by storing each pair of key/value onto the node to which this key maps.

DNS [28] maps a host name at an IP address. Chord can provide the same service such as the name represents the key and the associated IP address represents the value. Chord does not require any special server while the DNS is based on a set of routing servers. The names of DNS are structured to reflect administrative boundaries, Chord does not impose any naming structure. The DNS is designed especially to find the host names or the services, while Chord can also be used to find the data objects which are not attached to particular machines.

Chord routing mechanism may be considered as one dimensional analogy of Grid [23] location system. Grid is based on the real geographical position information to route its requests, Chord maps its nodes to one artificial dimensional space where routing is handled by an algorithm similar to Grid. Chord can be used as a location service in a variety of systems and helps them to avoid the problem of a single point failure since there is no central server. There is also no scalability issue as in the Gnutella system.

3.4.2 Chord features

What distinguish Chord from the other peer to peer systems are its features.

- Decentralization

Chord is a fully distributed system, no node is more important than any others. This improves the system robustness since there is no single point failure.

- Availability

Chord adjusts automatically its routing table to reflect the new joining nodes and the node failures in the system. Also Chord ensures that the node responsible for a key can always be found even if the system is in a continuous state of change.

- Scalability

The cost of a Chord lookup grows as the log of the number of nodes, so Chord can be used in very large systems.

- Load balance

Chord is a load balanced system since it uses a hash function that spreads the key evenly over the nodes.

➢ Flexible naming

Chord does not require any constraint on the structure naming, thus the application has a big flexibility to name the data.

3.4.3 The Chord ring

The Chord protocol uses Sha-1 [38] as consistent hash function to assign an *m* bit identifier to each node and key. Consistent hash functions are hash functions with some advantageous properties, such as, i.e., they let nodes join and leave the system with a minimal disruption [29] [22]. The *m* is an integer which should be selected large enough to make the probability that two nodes or two keys receiving the same identifier is negligible. The hash function calculates the key identifier by hashing the key and the node identifier by hashing the IP address of the node.

Consistent hashing assigns keys to nodes as follows. Key and node identifier are arranged in a ring of size 2^m called "the Chord ring". Identifiers in the Chord ring are numbered from 0 to 2^{m-1}. Key *k* is assigned to the first node whose identifier is equal or follows the identifier of *k*. This node is called successor node of *k*, denoted by successor(*k*), and it is the first node clockwise from *k* on the circle. Figure 3.4 shows a Chord ring with $m = 6$. The Chord ring has ten nodes and stores five keys, the successor of key K10 is node N14 and therefore key K10 is located at N14. Similarly, key K24 and K30 are located at node N32, key K38 at node N38 and key K54 at node N56.

To maintain the consistent hashing mapping, when a node *n* joins the overlay, some previously keys stored by the successor of *n* become now assigned to *n*. Also, when node *n* leaves the overlay, all its assigned keys are reassigned to its successor.

In Figure 3.4, if a node was to join with an ID K26, it would capture key K24 from the node identifier N32.

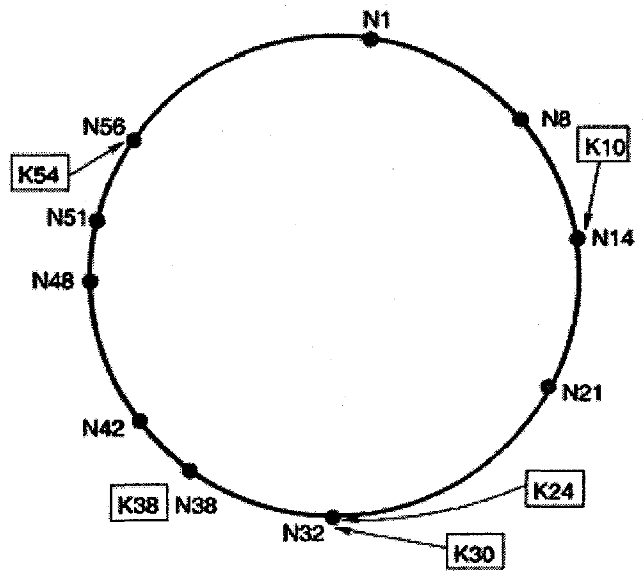


Figure 3.4: Chord ring with 10 nodes and 5 keys [29]

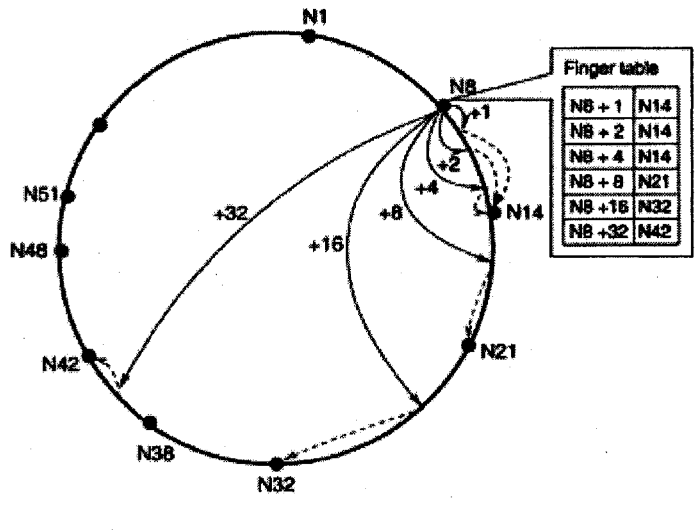
3.4.4 Key location

Key location is the core function of the Chord protocol. Chord uses a scalable function to locate a key and in order to provide an efficient lookup function: Chord nodes store some additional routing information in addition to the successor pointers. This additional information is not essential for the lookup correctness which is achieved as long as the successor information is maintained correctly; it is only used to accelerate the search process.

Each node *n* maintains a routing table of *m* entries (where *m* is the number of bits of the identifier) called “the finger table”. The i^th entry in the table at node *n* contains the first node *s* that succeeds *n* by at least 2^i, where $0 \leq i < m$ and all the arithmetic is modulo 2^m. This node *s* is called the i^th finger of node *n* and it is calculated with the formula:

$$\text{finger}[i] = (\text{successor } (n + 2^i)) \bmod 2^m.$$

A finger entry contains the Chord identifier and the IP address and port of the relevant node and maybe some other information. Figure 3.5 shows the finger table of node N8.



N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 +16	N32
N8 +32	N42

Figure 3.5: Finger table entry of a Chord node [29]

This scheme has three important characteristics:

- Each node stores information about only a small number of other nodes (*m*).
- Each node knows more about nodes closely following it on the ring than about nodes farther away.
- The finger table does not generally contain enough information to directly determine the successor of an arbitrary key *k*. A node has to contact other nodes in order to resolve the key lookup request.

When a node is asked to lookup for a given key, it will first determine the closest preceding node in its routing table and then forward the key to that node. This procedure is recursively repeated until the node responsible for the key is found. The lookup time is O(log N) since the query request is forwarded at least half the remaining distance around the ring in each step (see Figure 3.6 for an illustration and 3.7 for the description of the lookup algorithm).

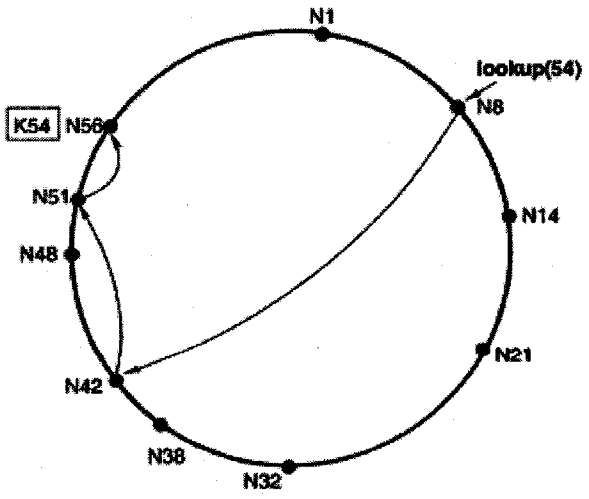


Figure 3.6: A query forward example [29]

```
//ask node n to find the successor of id
n.find_successor(id)
        if( id∈ (n,successor))
                return successor,
        else
                n'=closest_preceding_node(id);
                return n'.find_successor(id);


//search the local table for the highest predecessor of id
n.closest_preceding_node(id)
                                for i=m downto 1
                if (finger[i] ∈ (n,id))
                        return finger[i];
        return n;
```

Figure 3.7: Scalable key lookup using the finger table.

3.4.5 Node joins and stabilization

In order to ensure that the lookup is executed correctly while a status of some nodes is changing, Chord must ensure that the successor pointer of each node is up to date. It does this by using a "stabilization" protocol that each node executes it periodically to update the "finger table" and successor pointers. Figure 3.8 shows the pseudo code for the join and stabilization algorithm.

// create a new Chord ring.	Node n is the first node to start a new
	Chord ring. It does not have a predecessor
n.create()	and it is its own successor.
predecessor = nil;	
successor = n;	
// join a Chord ring containing node n'.	Node n wants to join the Chord ring.
n.join(n')	Knowing that n' is part of the ring, it will
predecessor = nil;	ask n' to find the successor of n
successor = n'.find successor(n);	
//Note: this line was missing in the original paper[29]	
successor.notify(n);	

Figure 3.8: Stabilization pseudo code [29]

//called periodically, verifies n's immediate successor, and tells the successor about n.
 	
n.**stabilize**()
    x = *successor.predecessor*;
    if (x ϵ (n; *successor*))
        successor = x;
    successor.notify(n);
	stabilize() is run periodically by each node. Denoted by x the predecessor of the successor of n, which will be n except for the case that a new node has joined recently between x and its successor. In this case, n will set its successor to x and notifies x of its own existence.

// n' thinks it might be our predecessor.
n.notify(n')
    if (*predecessor* **is nil or** n' ϵ (*predecessor*; n))
        predecessor = n';
	n' notifies n of its existence. If n does not yet have a predecessor and if n' is closer to n than its current predecessor, n sets its predecessor pointer to n'.

Figure 3.8 continued: Stabilization pseudo code [29]

n.fix _fingers()
    next = next + 1 ;
    if (next > m)
        next = 1 ;
finger[next] = *find successor*(n + $2^{\text{next-1}}$);
	each node runs fix_fingers periodically. This is how nodes update their finger tables and how new nodes initialize their finger table

n.check_predecessor()
    if (*predecessor* **has failed)**
predecessor = nil;
	Each node checks periodically whether its predecessor has failed, so it can clear its pointer and accept a new predecessor.

Figure 3.8 continued: Stabilization pseudo code [29]

Let us take the same example as the one presented in [29]. Assume that node n joins the system, and its ID is located between node n_p and node n_s. When it calls the *join ()* method, n acquires n_s as its successor. Node n_s, once notified by n, acquires n as its predecessor. The next time when n_p runs the *stabilize ()* method, it asks n_s for its predecessor (which is now n); n_p acquires then n as its successor. In conclusion, n_p notifies n, and n acquires n_p as its predecessor. At that time, all the predecessors and the successor pointers are correct. At each step of the process, n_s is reachable from n_p using its successor pointer, this means that concurrent lookups are not disturbed.

Figure 3.9 illustrates the join procedure, when n's ID is 26, and the ID of n_s and n_p are 21 and 32 respectively. N26 joins the system between nodes 21 and 32. The arcs represent the successor relationship. (a) Initial state: node 21 points to node 32. (b) Node 26 finds its successor (i.e., node 32) and points to it. (c) Node 26 copies all keys less than 26 from node 32. (d) The stabilize procedure updates the successor of node 21 to node 26 [29].

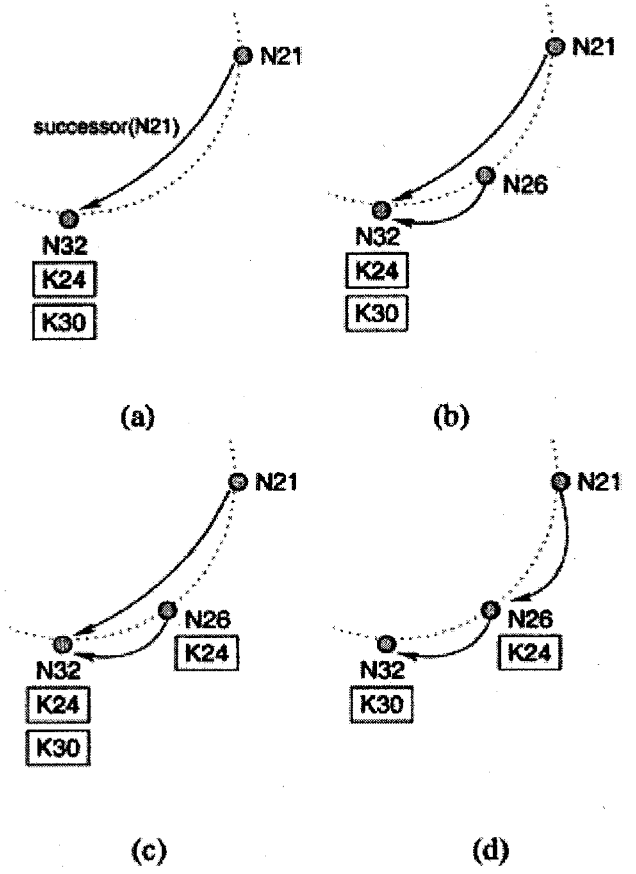


Figure 3.9: Example illustrating the join operation. [29]

3.4.6 Impact of node joins on correctness

When the stabilization procedure has not yet finished and a lookup request is recently launched, the overlay can exhibit one of the three following behaviors:

- All routing table entries (successors, predecessor and finger table entries) are correct at the time of the lookup and so there will be no impact on correctness. The lookup method will be successful with a O(log N) complexity.

- The successor pointers are correct but the finger entries are inaccurate. This yields to a successful lookup but it might be a little slower: in case a larger number of nodes have joined between the target and the target's predecessor, the find_successor method will initially undershoot and some of the hops will be in linear time (Figure 3.10).

- The visited node has an incorrect successor pointer and incorrect finger entries: in this case, the lookup function may fail but this situation could be handled by the application layer and the lookup will retry after a pause.

More generally, as long as the time it takes to adjust fingers is less than the time it takes the network to double its size, lookups will continue to take O(log N) hops [29].

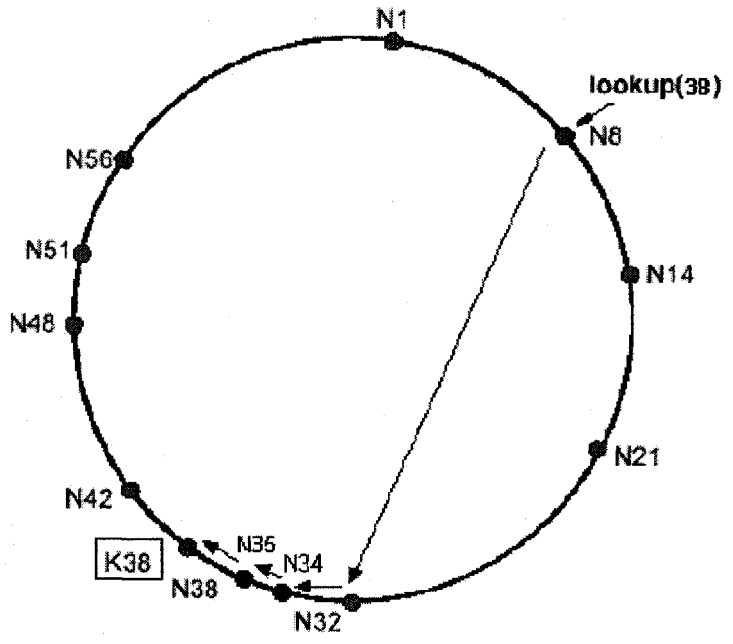
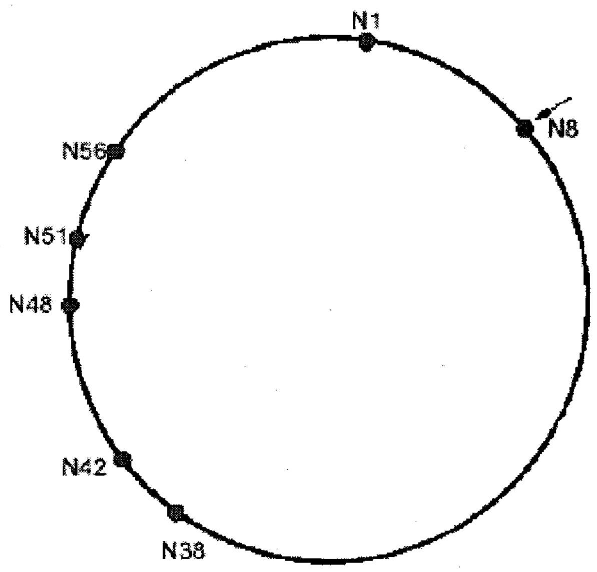
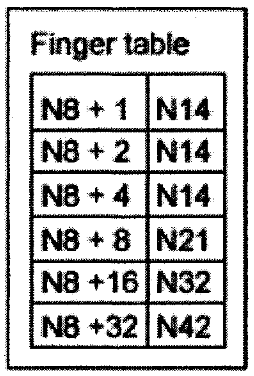


Figure 3.10: Impact of node joins on correctness.
The lookup might be a little slower when a large number of nodes join between the target and the target's predecessor [29]

3.4.7 Node failure

The correctness of the Chord protocol relies on the fact that each node knows its successor. When a multiple node failure occurs, it is possible that a node does not know its new successor, and that it does not have any chance to get information about it. Figure 3.11 provides an illustration of such a case.




Finger table

N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 +16	N32
N8 +32	N42

Figure 3.11: Simultaneous failure
When nodes N14, N21 and N32 fail simultaneously, N8 has no chance to learn about its new successor N38, since it does not show up in the finger table of N8 [29]

To increase robustness and avoid this situation, each Chord node maintains a successor list of size r, containing the node's first r successors. When the immediate successor node does not respond, the node simply contacts the next node on its successor list.

Assuming each node fails with probability p, the probability that every node in the list fails is p^r. Increasing r makes the system more robust. Even with a modest value of r, the failure probability is very small. In a network that is initially stable, if every node then fails with probability ½, then the expected time to execute the lookup method is O(log N). Proof can be found in [41].

3.5 Pastry

Pastry [13] is considered as a general substrate for the construction of a variety of peer-to-peer applications as file sharing, file storage and group communication. Several applications were established based on Pastry, including a persistent utility of storage called PAST [12] [36] and a scalable "publish/subscribe" system called SCRIBE [36].

Each Pastry node has a unique numerical identifier (nodeID). When a message and a key are presented to a node, this node can efficiently route this message to the node with a nodeID that is numerically closest to the key.

The expected number of routing steps is O(log N), where N is the number of Pastry nodes in the network. Pastry takes the network locality into account; it seeks to reduce the distance message travel, according to a proximity metric scalar such as the number of hops in the IP routing. Each Pastry node keeps track of its immediate neighbors in the nodeID space, and informs applications of new node arrivals, node failures and recoveries.

3.5.1 Pastry design

Pastry is fully decentralized, scalable, and a self organizing system, where each node can route client requests and interact with other Pastry nodes. Every computer that runs a Pastry client and connected to the Internet or a local network can be a Pastry node, subject only to satisfying the requirements of some security policies.

Every node in Pastry overlay is assigned a 128 bit node identifier. This ID is used to indicate the position of the node in the circular node ID space, which ranges from 0 to $2^{128} - 1$. The node IDs are randomly generated and Druschel *et al* [13] assume that they are uniformly distributed in the overlay. The node IDs are obtained by hashing the IP node address or a public key (name).

Consider a network consisting of N nodes. Pastry can route to the numerically closest node to a given key in less than $O(\log 2^{b} N)$ steps under normal operation (b is a configuration parameter with a typical value of 4). Despite concurrent node failures, delivery is guaranteed unless $\lfloor L/2 \rfloor$ nodes with their adjacent node IDs fail simultaneously

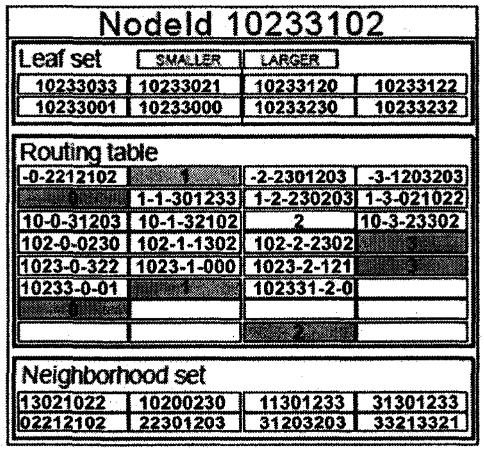
(L is a configuration parameter with a typical value of 16 or 32).

For the purpose of routing, node IDs and keys are thought of as a sequence of digits with base 2^{b}. Pastry routes messages to the node whose node IDs are numerically closest to the given key. This is accomplished as follows. In each routing step, a node forwards the message to a node whose node ID shares with the key a prefix that is at least one digit (or b bits) longer than the prefix that the key shares with the present node's ID. If no such node is known, the message is forwarded to a node whose node ID shares a prefix with the key as long as the current node, but is numerically closer to the key than the present node's ID. [13]

In order to perform the routing in a Pastry overlay, every node maintains a leaf set (L), a routing table (R) and a neighborhood set (M). A node's leaf set is a set of L nodes numerically closest to the current node, with L/2 closest larger and L/2 closest smaller node IDs relative to current node's node ID. The leaf set is used in the routing process as described in the routing Section 3.5.2.

Let us look at a routing table *R* of a specific Pastry node *X*. the routing table *R* is composed of ($\log_2^b$ N) rows with $2^b - 1$ entries each. Every entry in row n refers to a node whose node ID shares the first n digits with *X*'s node ID, but whose n+1th digit has one of the $2^b - 1$ remaining other possible values than the present nth digit of the current node *X*. Each entry in the routing table stores the IP address of a node whose node ID has the appropriate prefix and it should also be close to the actual node *X* according to proximity metric. If there is no node with an appropriate node ID then the entry in the routing table is left empty. Therefore, on average, only ($\log_2^b$ N) rows are filled in the routing table.

The neighborhood set M contains the node IDs and IP address of the M nodes that are closest to the current node *X* according to proximity metric. The neighborhood set is normally not used in the routing process but it is useful to maintain the locality properties. The typical value of L and M are 2^b or 2×2^b (Figure 3.12).



NodeId 10233102

Leaf set

SMALLER		LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	102331-2-0	
0			
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Figure 3.12: State of a Pastry node with a node ID 10233101, b = 2 and L = 8. The top row of the routing table is row zero. The shaded cell in each row of the routing table shows the corresponding digit of the present node's node ID. [13]

3.5.2 Routing

In this section we show the routing process pseudo code in Pastry (Figure 3.13). The process is executed each time a message with a key D arrives at a node with node ID A. We start by defining some notation.

R^i_l: the entry in the routing table R in column i, $0 \leq i < 2^b$ and row l, $0 \leq l < \lfloor 128/b \rfloor$.

L_i: the ith closest node ID in the leaf set L, $-L/2 \leq i \leq L/2$, where negative/positive indices indicate node IDs smaller/larger than the present node ID, respectively.

D_l: the value of the l's digit in the key D.

shl(A;B): the length of the prefix shared between A and B, in digits.

```
if ($L_{-\lfloor|L|/2\rfloor} \leq D \leq L_{\lfloor|L|/2\rfloor}$) {
    // D is within range of our leaf set
    Forward to $L_i$, s.th.$|D-L_i|$ is minimal;
} else {
        //use the routing table
        Let l = shl (D, A);
        If ( $R_i^{D_i} \neq$ null) {
                Forward to $R_i^{D_i}$ ;
        }
        Else {
                //rare case
                Forward to T $\in$ L $\cup$ R $\cup$M, s.th.
                        Shl (T, D) $\geq$ l,
                        $|T-D| < |A-D|$
        }
}
```

Figure 3.13: Pseudo code for Pastry core routing algorithm [13]

According to Druschel *et al* [13], For a given message, the node starts first by checking if the key falls in the range of the leaf's node ID (line 1). If so, the message is directly forwarded to the destination node that has the closest node ID to the key (line 3). If the key is not covered by the leaf set, then the routing table is used and the message is forwarded to a node which shares a common prefix with at least one digit longer than the key (line 6-8). In some case, it happens that the appropriate entry in the routing table is empty or the associated node is not reachable (line 11-14). In such a case, the message is forwarded to a node which shares the same prefix as the actual node but it is numerically closer to the key than the actual node (Figure 3.14).

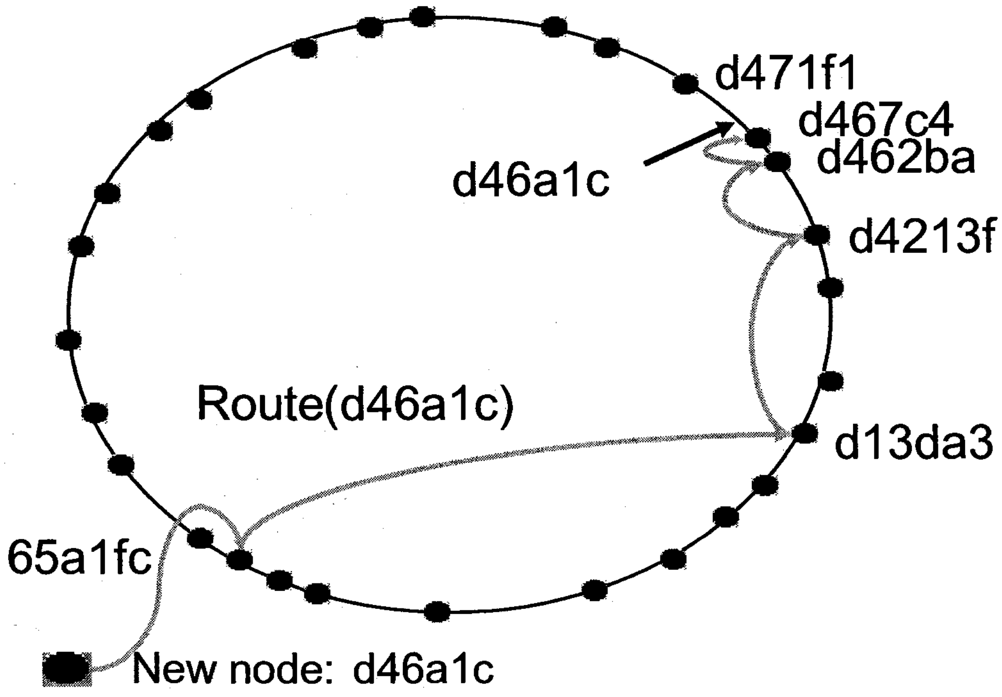


Figure 3.14: Pastry routing example [13]

3.5.3 Node arrival and departure

In this section, we describe the Pastry's protocol for handling the arrival and departure of a node in the Pastry system.

3.5.3.1 Node arrival

When a node joins the Pastry system, it needs to initialize its state table and then to inform the other nodes of its presence. Pastry assumes that the new joining node knows initially at least one nearby node *A* according to a proximity metric.

Assume that the node ID of the joining node is *X*. Node *X* asks *A* to route a join message with a key *X*. Like any message, Pastry routes the join message to the numerically closest node to *X* which is *Z*.

In reply to the join request, nodes *A, Z* and all nodes encountered on the path from *A* to *Z* send their state table to *X*. The new joining node inspects this information and uses it to fulfill its state table as described below.

Node *X* uses the *A*'s neighbors to set its neighborhood set since node *A* is assumed to be in its proximity. Further, the new node *X* uses the *Z*'s leaf set to fill its leaf set since *Z* is numerically the closest node to *X*. Next, the new node X starts filling the routing table row by row. Let us assume that A_i denotes node A's routing table at level *i*. Moreover, the entries in the routing table at row zero are independent of a node's node ID, thus A_0 contains appropriate values for X_0. Node X does not use other entries from *A* since *A*'s and *X*'s IDs share no common prefix.

However, appropriate values for X_1 can be taken from B_1, where B is the first node encountered along the route from A to Z. Indeed, observe that entries in B_1 and X_1 share the same prefix, because X and B have the same first digit in their node ID. Similarly, X obtains the appropriate entries for X_2 from node C, the next node encountered along the route from A to Z, and so on.

3.5.3.2 Node departure

The nodes in the Pastry network may fail or leave without any warning. A Pastry node is considered in a failure state when its immediate neighbors in the node ID space cannot communicate any more with it.

"To replace a failed node in the leaf set, its neighbor in the node ID space contacts the live node which has the first greater index than the failed node and asks that node for an appropriate entry from its leaf set that could replace the failed node.

To replace a failed entry R^d_l in the routing table, a node first contacts the node referred to by another entry R^i_l, $i \neq d$ of the same row, and asks for that node's entry for R^d_l. In the event that none of the entries in row l has a pointer to a live node with the appropriate prefix, the node next contacts an entry R^i_{l+1}, $i \neq d$." [13]

If an entry in the neighborhood set is not responding, the node asks other members for their neighborhood tables, checks the distance of each of the newly discovered nodes, and updates it own neighborhood set accordingly.

3.6 Tapestry

In this section, we present the Tapestry routing [45] architecture, a self-organizing, scalable and robust infrastructure that routes efficiently requests to content even in the presence of heavy load and node faults. Tapestry can route a message to the right destination using only point to point link and without centralized services.

Tapestry is inspired by the distributed architecture of Plaxton [16], augmented with additional mechanisms to provide availability, scalability, and adaptation in the presence of failures and attacks. The directory and routing information within this infrastructure is purely soft state and easily repaired. Also, Tapestry is self administrating, fault-tolerant, and resilient under load, and is a fundamental component of the OceanStore system [21][33].

Further, Tapestry proposes a dynamic and stable environment. Faulty components are transparently masked, failed routes are bypassed, nodes under attack are removed from service, and communication topologies are rapidly adapted to circumstances [45].

In Tapestry, every node can play the role of a server (where objects are stored), a router (which routes messages) and a client (origins of a request). Also, objects and nodes have names independent of their location and semantic properties, in the form of random fixed-length bit-sequences represented by a common base (e.g., 40 Hex digits representing 160 bits). The system assumes that entries are roughly evenly distributed over both node and object namespaces, which can be achieved by using the output of hashing algorithms, such as SHA-1 [38].

3.6.1 Routing and Object Location

According to Zhao *et al* [45], tapestry dynamically maps each data's identifier *G* to a unique live node called the identifier's *root Gr*. If a node has a node ID $N_{id} = G$ then this node is the root of *G*.

To deliver messages, every node maintains a routing table composed of node IDs and IP addresses of nodes with which it communicates. These nodes are considered *neighbors* of the local node.

To establish the routing toward *Gr*, messages are forwarded across neighbor links to nodes whose IDs are progressively closer to *G* in the ID space (i.e., matching larger prefix).

Tapestry uses local tables at each node called *neighbor map*, to route overlay messages to the destination ID digit by digit (e.g., 6*** => 63** => 635* => 635A, where *'s represent wildcards). This approach of routing is similar to that of Pastry (see Section 3.5).

A node has a neighbor map with multiple levels, where each level contains links to nodes matching a prefix up to a digit position in the ID and contains a number of entries equal to the ID's base. The primary *i*th entry in the *j*th level is the ID and location of the closest node that begins with prefix (N,*j*-1) + "*i*" (Figure 3.15) [45].

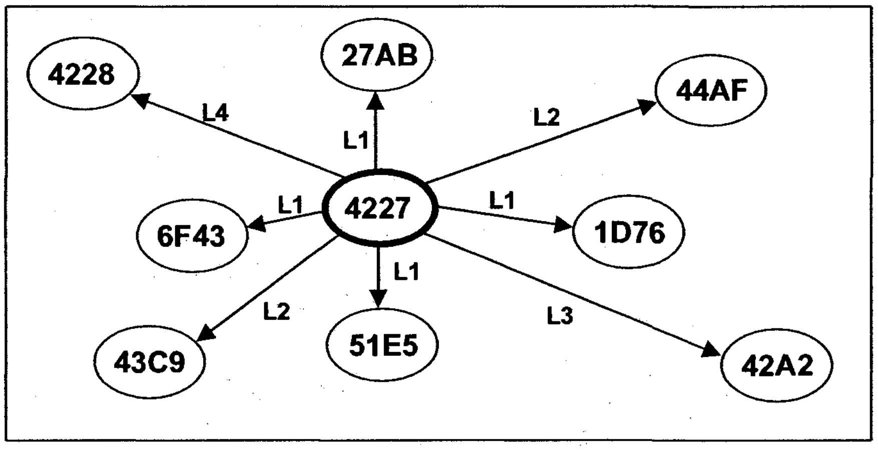


Figure 3.15: Tapestry routing mesh from the perspective of a single node. Outgoing neighbor links point to nodes with a common matching prefix. Higher level entries match more digits. Together, these links form the local routing table. [45]

Therefore, the router of the *n*th hop shares a prefix of at least length *n* with the destination ID. To find the next hop, Tapestry looks for the *n*+1th level map and searches the entry matching the value of the next digit in the destination ID. This method guarantees that any existing live node in the overlay can be found within at most $\log_b$ N hops, where N is the number of nodes and *b* is the ID's encoding base (Figure 3.16). When a digit cannot be matched, Tapestry looks for the numerically closest digit in the routing table, this is called *surrogate routing* [45].

As described in [45], to continue to route reliably even when intermediate links are changing or faulty and to help provide resilience in a dynamic environment, Tapestry exploits network path diversity in the form of redundant routing paths.

Primary neighbor links shown in Figure 3.17 are augmented by backup links, each sharing the same prefix. At the *n*th routing level, the *c* neighbor links differ only on the *n*th digit. There are $c \times b$ pointers on a level, and the total size of the neighbor nodes is $c \times b \times \log_b N$. Each node also stores reverse references (backpointers) to other nodes that point at it (Figure 3.17) [45].

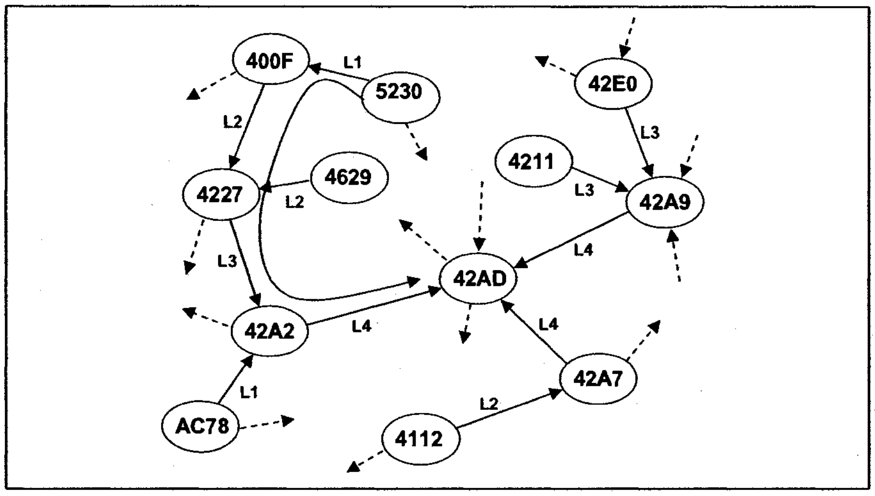


Figure 3.16: Path of a message.
The path taken by a message originating from node 5230 destined for node 42AD in a Tapestry mesh. [45]

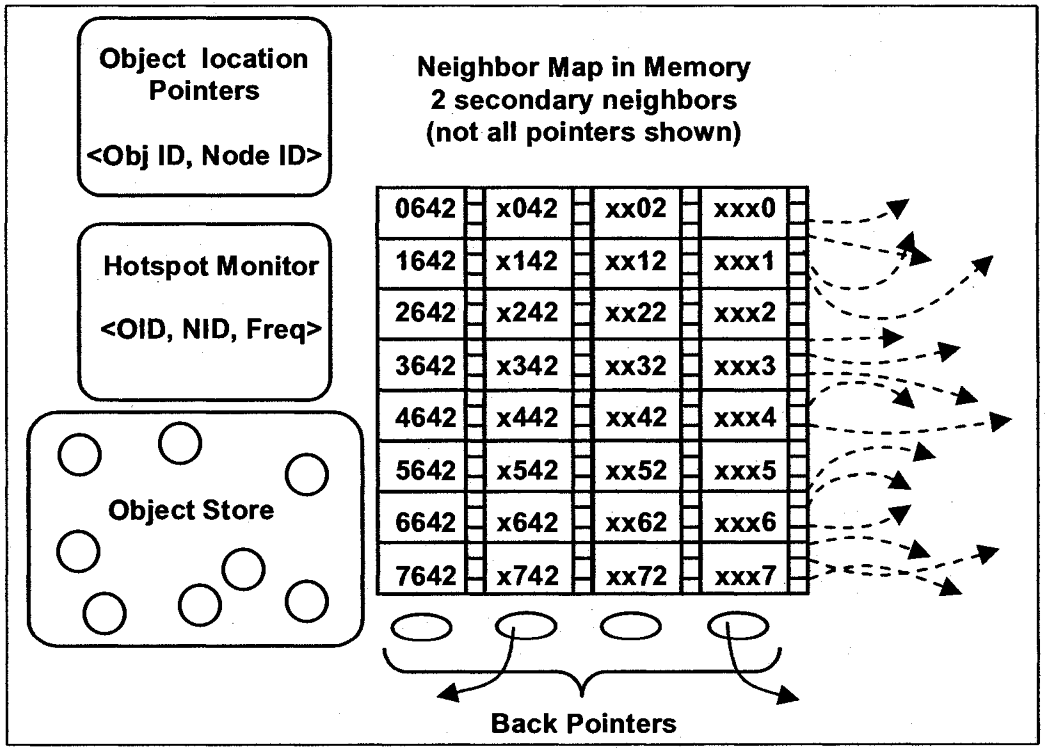


Figure 3.17: A complete routing table of the node 0642 [45]

3.6.2 Location and Object Publication

According to Zhao *et al* [45], the location mechanism allows a client to locate and send messages to an object *O* residing on a server. A server *S* periodically publishes its object *O* by routing a message to the root node of *O* (Figure 3.18). The root node is a unique node in the overlay used to store the location of an object *O*. The publish process consists mainly in sending a message toward the root node. At each hop along the way, the publish message stores location information in the form of a mapping <Object-ID(*O*), Server-ID(*S*)>. Note that these mappings are simply pointers to the server *S* where *O* is being stored, and not a copy of the object *O* itself. When multiple objects exist, only the reference to the closest object is saved at each hop to the root.

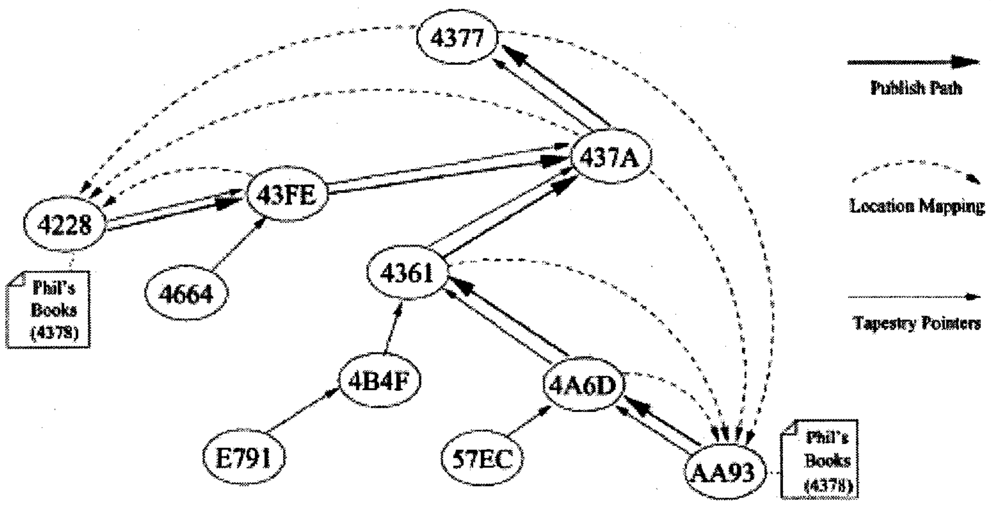


Figure 3.18: Tapestry object publish example.
Two copies of an object (4378) are published to their root node at 4377. [45]

When a client wishes to query for an object *O*, it routes a message toward the root of *O*. At each step, if the hop contains the location mapping for *O*, it is immediately redirected to the server containing the object. Otherwise, the message is forwarded one step closer to the root. If the message reaches the root, it is guaranteed to find a mapping for the location of *O* (Figure 3.19).

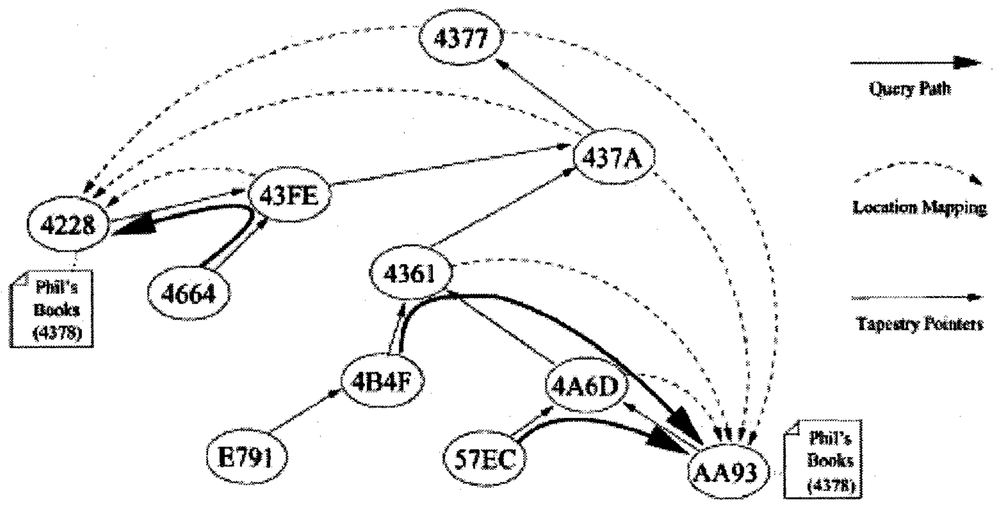


Figure 3.19: Object query example in the Tapestry network. [45]

3.7 Comparison of DHT location algorithms

The Distributed Hash Table (DHT) is a structured P2P system in which data object or value location information is placed at the peers with closest identifiers to the data object's unique key. Most DHT based systems consistently assign uniform random NodeIDs to the set of peers into a large space of identifiers. Data objects are assigned unique identifiers called keys, chosen from the same identifier space. Keys are mapped by the overlay network protocol to a unique live peer in the overlay network. Most DHT based systems share some features as the scalable storage and retrieval of (key,value) pairs on the overlay network, which involves routing requests to the peer corresponding to the key.

Each peer maintains only some information about other nodes in the overlay consisting of its neighboring peers' NodeIDs and IP addresses that form its routing table. Lookup queries or message routing are forwarded across overlay paths to peers in a progressive manner, with the NodeIDs that are closer to the key in the identifier space. Different DHT based systems have different organization schemes for the data objects and its key space and routing strategies. In theory, most DHT based systems can ensure that any data object can be found in a small O(logN) overlay hops on average, where N is the number of peers in the system [33].

In [33], Keong Lua *et al* explains that the algorithm of Plaxton was originally designed to route web queries to nearby hosts, and this influenced the design of Pastry, Tapestry and Chord. The algorithm of Plaxton has a logarithmic expected join/leave complexity. Plaxton guarantees that queries never travel further in network distance than the peer where the key is stored. However, Plaxton has several drawbacks: it requires global knowledge to construct the overlay; it has the problem of the single point of failure since it uses only one object's root peer; no insertion or deletion of peers; no avoidance to hotspot congestion. Chord, Pastry and Tapestry provides some enhancements to eliminate these problems. They rely on DHT to provide a semantic-free Node ID and perform efficient request routing among lookup peers using an efficient and dynamic routing infrastructure [33].

According to Morris *et al* [29], Tapestry and Pastry are very similar and are based on the idea of a Plaxton mesh. Identifiers are assigned based on a hash of the IP address of each peer. When a peer joins the network, it contacts a bootstrap peer and routes toward the peer in the network with the ID that most closely matches its own ID. Routing state for the new peer is built by copying the routing state of the peers along the path toward the new peer's location. For a given peer n, its routing table will contain *i* levels where the *i* th level contains references to b nodes (where b is the base of the identifier) that have identifiers that match the first *i* positions of n. Routing is based on a longest prefix protocol that selects the next hop to be the peer that has a prefix that matches the desired location in the greatest number of positions. Robustness in this protocol relies on the fact that at each hop, multiple nodes, and hence multiple paths, may be traversed.

In spite of many similarities between Pastry and Tapestry in choosing keys, locating nodes and the insertion of nodes, there are also differences that distinguish them. First, objects in Pastry are replicated without any control by the owner and these replicas are placed on several nodes whose node IDs are closest in the namespace to that of the

object's ID. Second, while Tapestry places references to the object location on hops between the server and the root, Pastry assumes that clients use the object ID to attempt to route directly to the vicinity where replicas of the object are stored.
While placing replicas at different nodes in network may reduce location latency, it comes at the price of security, storage overhead and confidentiality.

As described in Chord paper [29], the Chord algorithm models the identifier space as a uni-dimensional, circular identifier space. Peers are assigned IDs based on a hash on the IP address of the peer. When a peer joins the network, it contacts a gateway peer and routes toward its successor. The routing table at each peer n contains entries for *log N* other peers where the i-th peer succeeds n by at least 2^{i-1} with respect to its node ID. To route to another peer, the routing table at each hop is consulted and the message is forwarded toward the desired peer. When the successor of the new peer is found, the new peer takes responsibility for the set of documents that has identifiers less than or equal to its identifier and establishes its routing table. It then updates the routing state of all other peers in the network that are affected by the insertion. To increase the robustness of the algorithm, each document can be stored at several successive peers. Therefore, if a single peer fails, the network can be repaired and the document can be found at another peer.
Chord guarantees that the routing requests cross a logarithmic number of hops and that the keys are well balanced. It uses a "Consistent Hashing" algorithm to let peers enter and leave the network with minimal interruption. Consistent Hashing ensures that the total number of nodes responsible for a particular object is limited and when the state of these nodes change, the minimum number of object references will move to maintain a load

balancing. In Chord, the network is maintained appropriately by a background maintenance process which guarantees scalability and fault tolerant even in a very dynamical network which is not the case for Tapestry. All these comments are summarized in Table 3.1.

Algorithm Comparison Criteria	Chord	Tapsestry	Pastry
Decentralization	DHT functionality on Internet-like scale		
Architecture	Uni-directional and Circular Node ID space.	Plaxton-style global mesh network.	Plaxton-style global mesh network.
Lookup protocol	Matching Key and Node ID.	Matching prefix in Node ID.	Matching prefix in Node ID.
System parameters	N-number of peers in network.	N-number of peers in network and B-base of the chosen peer identifier.	N-number of peers in network and b-number of bits (B= 2^b) used for the base of the chosen identifier.

Table 3.1: Comparison of DHT location algorithms

Routing performance	O(log N)	O($\log_b$ N)	O($\log_b$ N)
Hops to locate data	Log N	$\log_b$ N	b.$\log_b$ N + b
Peers join/leave	$(\text{Log N})^2$	Log N	Log N
Reliability/Fault Resiliency	Failure of peers will not cause network wide failure. Replicate data on Multiple consecutive peers. On failures, application retries.	Failure of peers will not cause network wide failure. Replicate data across multiple peers. Keep track of multiple paths to each peer.	Failure of peers will not cause network wide failure. Replicate data across multiple peers. Keep track of multiple paths to each peer.
Proximity metric	No	Yes	Yes
Load balancing	Yes	No	No

Table 3.1 continued: Comparison of DHT location algorithms

In this Chapter, We have presented some P2P location algorithms and we have compared the features of the main four DHT location algorithms.

4 The SIP2P Telephony System

This chapter is the core part of the thesis. We describe the architecture of the proposed SIP2P telephony system and its design goals. The SIP2P system differs from the other P2P telephony systems by its scalability, its decentralized management, i.e., it does not use any central server or super node and its modularity (see Section 4.3.6).

This chapter is organized as follows. In Section 1, we present a general overview about SIP [34]. In Section 2, we describe the XML syntax which is used in the SIP message. Section 3 defines the design goals of the proposed SIP2P system. In Section 4, 5, 6, 7, 8, and 9, we describe the architecture of the SIP2P system. Finally, in Section 10, we present two registration examples.

4.1 SIP

The Session Initiation Protocol (SIP) [34] is a control protocol. Its job is to set up, modify, and tear down sessions between session users. One of its functions is to act as a signaling protocol, and another is to define the type of session for which it is signaling. The session may be a multimedia conference, a point-to-point telephone call, and so on.

SIP has other capabilities than just signaling operations. It can support sessions via multicast or single unicast, a mesh of unicast sessions, or a combination of these choices. SIP does not act as a media gateway, in that it does not transport any media streams.

One of its more distinctive features is its ability to support mobile users. If a user registers his location with a SIP server, SIP will direct SIP messages to the user or invoke a proxying operation to another server to the user's current location. The mobile capability is keyed to the individual user and not to the user's terminal (telephone, computer, etc.). SIP is an attractive support tool for IP telephony for the following reasons.

- It can operate as stateless or stateful. A stateless implementation provides a good scalability since the servers do not have to maintain information on the call state once a transaction has been processed. Moreover, a stateless approach is very robust, since the server does not need to remember anything about a call.
- It uses many of the formats and syntax of HTTP, thus providing a convenient way of operating with ongoing browsers.
- The SIP message i.e., (the message body) is opaque; it can be of any syntax. Therefore, it can be described in more than one way. For instance, it can be described with the Multipurpose Internet Mail Extension (MIME) or the Extensible Markup Language (XML).
- It identifies a user with an URL, thus providing the user the ability to initiate a call by clicking on a web link.

SIP supports four major functions:

- Determination a user location;
- Determination of media for the session;
- Determination of willingness of a user to participate;
- Call establishment, call transfer, and termination;

4.1.1 Architecture of SIP

One of the most important architecture features of SIP is that it relies on a client/server model.

The following entities are defined inside this model.

- *User Agent (UA):* It is the SIP entity that interacts with the user. It usually has an interface towards the user. SIP UAs are implemented on top of many different systems. They can run, for instance, in a computer as one among many applications, or they can be implemented in a dedicated device as a SIP phone.

- *Proxy server:* It acts both as a server and a client for the purpose of making requests on behalf of other clients. Requests are serviced internally or being passed on to other servers. A proxy interprets and may rewrite a SIP request message before forwarding it to another server or to a user agent.

- *Redirect server:* It is a server that accepts a SIP request, maps the address in the request to a new address, and returns the message to the client. Unlike a proxy server, it does not initiate its own SIP requests and does not forward SIP requests to other servers. Unlike a user agent server, it does not accept calls.

- *Registrar:* It is a SIP entity that accepts *REGISTER* requests. A registrar is typically collocated with a proxy or redirect server and offers location services.

The registrar is used to register SIP parties in a SIP domain.

- *Location servers:* They are not SIP entities, but they are important part of any architecture that uses SIP. A location server stores and returns possible locations for users. It can make use of information from registrars or from other databases. Most registrars upload location updates to a location server upon receipt.

4.1.2 SIP response messages

A SIP response is a message generated by a SIP server to reply to a request generated by an UA. A response may contain additional header fields containing the information needed by the SIP server or it may be a simple acknowledgement to prevent retransmission of the request by the UA.

There are six classes of SIP responses. The first five classes were borrowed from HTTP; the sixth one was created for SIP. The classes are shown in Table 4.1.

Class	Description	Action
1xx	Informational	Indicates the status of a call prior to its completion.
2xx	Success	Request has succeeded. If it is for an INVITE, ACK should be sent; otherwise, stops retransmissions of request.
3xx	Redirection	Server has returned possible locations. The client should resend request at another server.
4xx	Client error	The request has failed due an error by the client. The client may try to resend its request if reformulated according to response.
5xx	Server failure	The request has failed due an error by the server. The client may try at another server.
6xx	Global Failure	The request has failed. Do not try to resend the request again at this or other servers.

Table 4.1: SIP response classes

4.2 XML

In this section, we present a brief overview of XML since it is the language that we have selected to describe the content of a SIP message.

Extensible Markup Language (XML) [3] originated from Standard Generalised Markup Language (SGML) [2], defined by ISO 8897, which was originally designed for the markup of the text. XML can be used to define elements of textual documents that are used for marking up their data types and their structure. Since XML is platform independent, it is also a valuable way for encapsulating data and for use it as middleware in a networked environment.

The basic structure of XML is very similar to most other applications of SGML, including Hyper Text Language (HTML), which is a very simple version of SGML. XML documents can be very simple, with no document type declaration (DTD), and straightforward nested markup of one's own design.

XML provides a text-based means to describe and apply a tree-based structure to information. At its base level, all the information is expressed through as text, interspersed with markup that indicates the information's separation into a hierarchy of character data, container-like elements, and attributes of those elements.

4.2.1 XML syntax

Tags are used as directives to applications reading XML-text and are enclosed text strings in angle brackets, for example < TAG >. In HTML, these tags are used to tell the web-browser what colours to use, the size of a font or to include images etc. In XML, it is up to the application reading the XML-file, what different tags mean. Figure 4.1 shows how it can be used.

```
<?XML version = "1.0" ?>

<STUDENTGRADE>

        <STUDENT>

                <NAME>

                        <LASTNAME>Smith</LASTNAME>

                        <FIRSTNAME>John</FIRSTNAME>

                </NAME>

                <GRADE>A</GRADE>

                <ID>12345678</ID>

        </STUDENT>

</ STUDENTGRADE >
```

Figure 4.1: Example of XML message

The first tag shown in the above example of Figure 4.1 is a processing instruction that tells the application that this document is an XML document and uses XML version 1.0. The other tags in the example are tags that are defined for this example only. The XML-file structures the data in a document consisting of one or more students. To create hierarchical structures, a start tag, like < STUDENTGRADE >, and an end tag, like
</ STUDENTGRADE > is used. Start and end tags are used to put data in context and to group data. Between a start tag and an end tag, we can either put data or more tags to define a multi-level hierarchy.

4.2.2 Correctness in an XML document

For an XML document to be correct, it must be:

- **Well-formed**

A well-formed document conforms to all of XML's syntax rules. For example, if a non-empty element has an opening tag with no closing tag, it is not well-formed. A document that is not well-formed is not considered to be XML; a parser is not allowed to process it.

- **Valid**

A valid document has data that conforms to a particular set of user-defined content rules, or XML schemas, that describe correct data values and locations. For example, if an element in a document is required to contain text that can be interpreted as being an integer numerical value, and if instead of an integer numerical value, it contains the text "hello", or is empty, or has other elements in its content, then the document is not valid.

4.3 Design goals and challenges

We propose a P2P telephony system architecture aiming at the goals and challenges described below.

4.3.1 Decentralization

In the proposed SIP2P telephony system, there will be no central server or super peer. We aim at designing a system where each node has the same importance as any other node. Therefore, it corresponds to a very robust system without any single point of failure.

4.3.2 Self Organizing

The system should be able to automatically adapt and configure itself in order to handle the changes caused by peers connecting or disconnecting from the P2P system. It should also perform the discovery and setup of neighbors. The system does not need on site administrator, the user should be able to just plug in it and the system can handle all the setup operations automatically.

4.3.3 Efficient Lookup

The flooding search is a pure P2P model in which no advertisement of shared resources occurs but it is not efficient due to its limitation with respect to scalability and correctness.

We decided to use an underlying DHT to optimize the lookup and following the analysis of Section 3.4, we have chosen Chord [29] as DHT algorithm because of its performance and correctness even in the case of concurrent join and leave operations.

4.3.4 Scalability and resiliency

An immediate benefit of decentralization is an improved scalability. The cost of a Chord lookup grows only logarithmically in the number of nodes in the system, so the proposed SIP2P telephony system can be used for very large systems (it can reach 2^{160} when we use SHA-1 as hashing algorithm), see Section 3.4 for more details.

One of the primary design goals of a P2P system is to avoid a single point of failure, so that the system can be robust enough to face simultaneous node failures.

4.3.5 Advanced services

The system should be able to provide advanced services as call forward, multi party conference and offline messages.

4.3.6 Modularity

The purpose of the modularity is to allow dynamic changes in the overlay and to provide a mechanism that separates the DHT overlay and SIP. Therefore, no modification is required for SIP protocol. This is one of the main features of the proposed SIP2P system. Furthermore, modularity allows us to adapt easily with new DHT protocols.

The proposed SIP2P telephony system is divided in 3 layers (see Figure 4.2): the application layer, the DHT layer and the SIP layer which are next described.

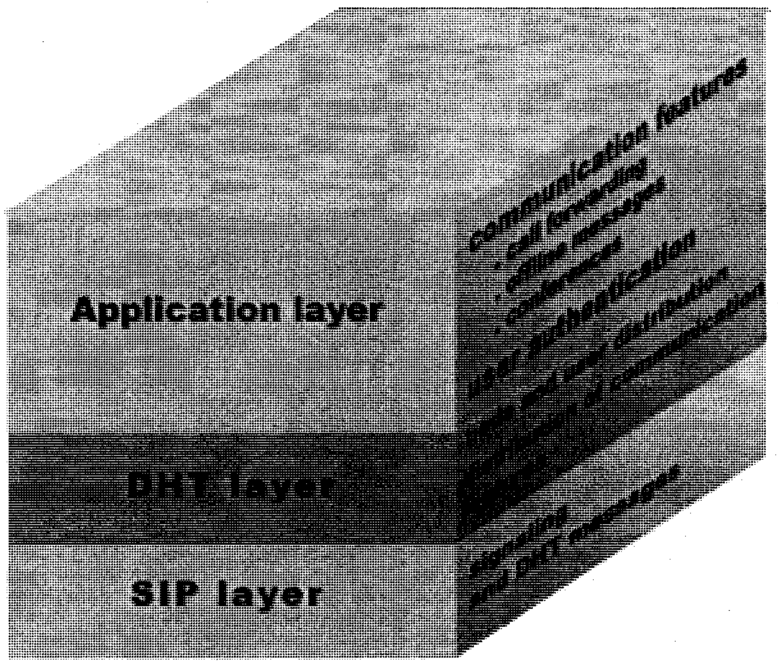


Figure 4.2: Layered architecture of the SIP2P system

4.3.6.1 Application Layer

The application layer is the interface between the user and the SIP2P system. It provides the communication functions like call forwarding, offline messages and conferences. It also ensures the security and the user authentication.

4.3.6.2 DHT Layer

The DHT layer uses Chord and serves as a lookup protocol. This layer not only permits us to construct update and query messages without changing the SIP protocol but also allows dealing easily with the new DHT protocol without changing SIP protocol.

4.3.6.3 SIP Layer

The SIP layer is a signaling, presence and instant messaging protocol used in order to set up, modify, and tear down multimedia session and request and in order to deliver presence and instant messages over the network. All the messages used in the overlay are SIP messages.

4.4 The Overlay Structure

Nodes are organized using a Distributed Hash Table (DHT) structure based on Chord. The system uses SHA-1 [38] as a consistent hash function to assign a m-bit identifier to each node and each resource.

Consistent hash functions are hash functions with some additional advantageous properties, i.e., they let nodes join and leave the system with minimal disruption [18][22]. The parameter *m* is an integer which should be chosen big enough to make the probability that two nodes or two keys receive the same identifier negligible (see Section 3.4).

Let us explain how Chord has been used in the SIP2P system. Following the description of Chord in Section 3.4, we proceed as follows.

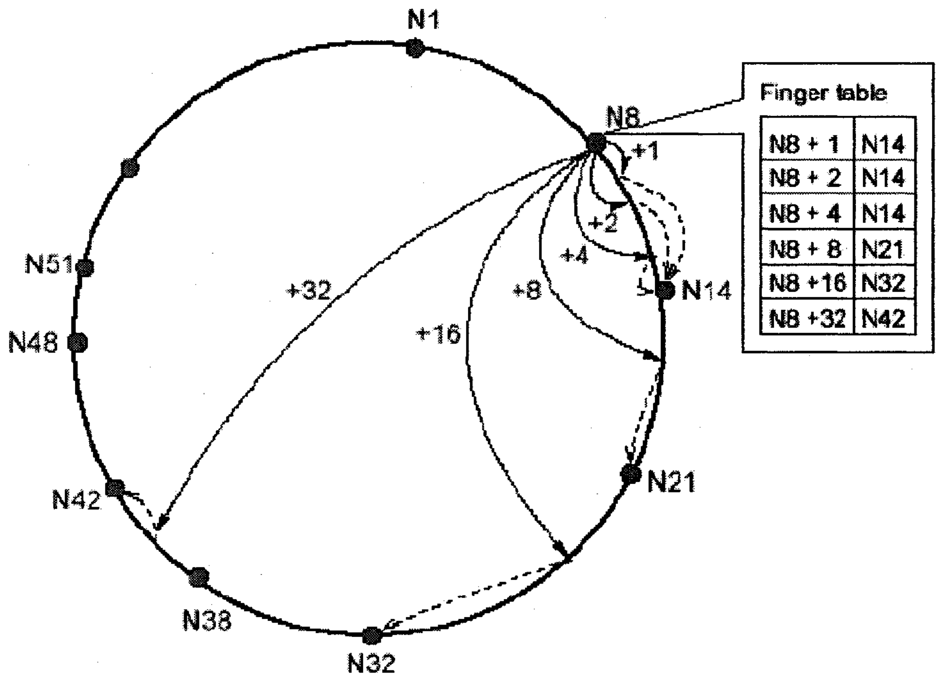
The hash function calculates the resource-ID identifier by hashing the user-ID, and the node identifier by hashing the IP address of the node. Both the resource-ID and the node-ID are mapped to the same hash space.

The resource-ID and the node identifiers are arranged on an identifier circle of size 2^m called the Chord ring. The identifiers on the Chord ring are numbered from 0 to 2^{m-1}. A resource-ID is assigned to a node whose identifier is equal to or greater than the resource-ID.

In order to provide more efficient lookups, additional routing information is stored to accelerate lookups. Each node n maintains a routing table with up to m entries (where m is the number of bits of the identifiers) which is called the finger table (Figure 4.3).

The ith entry in the finger table at node n contains the first node s that succeeds n by at least 2^{i-1}, finger[i] = successor $(n + 2^{i-1})$.

A finger table entry includes mappings between node-ID and node URL (IP address + port number) of neighbors to the relevant node. In addition, each node store *n* successor nodes for redundancy which is used to protect against node failure and loss of information. In the proposed SIP2P system, we have used four successors as is recommended for Chord algorithm [29]. Also, every node keeps the URL of its immediate predecessor node.



N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 +16	N32
N8 +32	N42

Figure 4.3: An example of a finger table entries [29]

The proposed SIP2P system overlay has some important characteristics which are as follows.

- Each node stores information about only a small number of nodes, e.g., log N.
- Each node knows more about nodes closely following it than about nodes farther away.
- A finger table generally does not contain enough information to directly determine the successor of an arbitrary resource-ID. A node has to contact other nodes in order to find the successor of the resource-ID.

The nodes take advantages of the three above characteristics to route a message. When a node wants to locate a resource-ID, it looks into its finger table for the closest node to the desired resource-ID. Since the receiving node has a closest node to this resource-ID, it sends the appropriate entry in its finger table as a response to the query request. The process is repeated until the node responsible for the resource-ID is located.

4.5 Message types

As mentioned before, all messages used in the overlay are SIP messages but we can distinguish two classes of messages: node messages and user messages. The first class is used to maintain the DHT layer, such as the message needed to join or leave the network, to update the finger table or to query for a node. The second class of messages is performed by the users in order to establish a service such as inviting or registering a user.

4.5.1 Node messages

These messages are used for maintenance purpose and they are exchanged among DHT nodes. We know that a SIP URI is constructed from two parts: a username and a host portion. In the node messages, the username part is represented by the hashing of the IP address and the port number of the appropriate node.

The hashing result must also be concatenated to the host portion which is the IP address of the appropriate node.

In order to distinguish between a node message and a user message we propose to add the specific tag “NodeOperation” in the XML body to indicate that this is an URI node.

4.5.2 User messages

The username part of a user message is represented by the unhashed user identifier and the host portion is constructed using the rules in RFC3261 [34].

In order to distinguish between a node message and a user massage, we add the specific tag "users-Operation" in the XML body to indicate that this is a URI user.

4.6 Node registration

4.6.1 Overview

Node registration is an internal operation which can be only performed by nodes wishing to join the overlay. The node should first construct its node-ID by hashing its IP address and port number. Then it sends a register request to a bootstrap node which has been discovered by a mechanism described below.

The bootstrap node analyses the node-ID of the joining node. Then it lookups for the nearest node in its finger table and responds with "*302 redirect*" to redirect it to this nearest node.

The joining node will repeat this process until it reaches the admitting node which is the node currently responsible for the joining node location. Additional messages can be exchanged with the admitting node to move resource-ID which the new joining node is now becoming responsible for them.

4.6.2 Bootstrapping

When a node wants to establish its first request to join the overlay, it has to discover an existing node in the overlay called bootstrap node. It is the first node which the joining node enters in contact with. A number of mechanisms can be used to discover these bootstrap nodes. We describe them below.

- Multicast with a small TTL (Time To Live) value

This mechanism can be used to discover an initial node. This can be done by using the SIP multicast registration address (224.0.1.75). If the node receives multiple responses, it can choose which one to use.

- Cached addresses

This mechanism cannot be used the first time that a node starts but it is a good solution for subsequent join. The mechanism consists of storing the previous bootstrap nodes and using them for the future connections. We recommend to cache some of the last bootstrap nodes (for example 20) and to try to use them in subsequent connections starting by the last bootstrap peer.

If all the peers fail, we use the multicast mechanism.

- Static node

As the last resort, some persistent bootstrap nodes can be used. These addresses could be pre-configured into the phone application or obtained from a DNS [28] query to a well known domain or could be entered manually.

4.6.3 Node registration type

We distinguish two types of node registration requests. The first one is the node update request that is used when a node wishes to join the overlay, the second one is the node query request which permits the lookup for a node already in the overlay.

4.6.3.1 Constructing the node update

In a node update request, the register-URI is constructed only by the IP address of the node wishing to be contacted. The To and From header must contain a SIP URI; the hashed IP address of the sending node as a username and the IP address in the host portion (see Section 4.5.1).

The node must provide a contact header when registering in order to distinguish that this is a node update and not a node query. The URI in the contact header must be constructed as described in Section 4.1. The *expire* field or the *Expires* header must be positive rather than equal to zero since as in standard SIP, *Expires* header with zero value is used to remove registrations.

The node must include *Require* and *Supported* headers with the option tag "P2P-DHT"

In the XML body, we include the overlay name and the hashing algorithm. The content of the < DHT-Operation > tag must be:

< DHT-Operation >nodeRegistration</ DHT-Operation >

since it is a node registration and not a user registration.

The body must also provide a DHT-node tag containing the node ID and the IP address of the appropriate node.

Example 4.1: Node registration request

Assume that a node running on IP address 10.0.0.32 wishes to join an overlay named "Concordia DHT" and contact a bootstrap node at address 10.0.0.21.

Further assume that the hash of the IP address 10.0.0.32 is 54ab7cd8 using SHA1.

A node register message will look like as described in Figure 4.4.

```
Register sip:10.0.0.21 SIP/2.0
TO: sip: 54ab7cd8@10.0.0.32
From: sip: 54ab7cd8@10.0.0.32
Contact: sip: 54ab7cd8@10.0.0.32
Expires: 800
Content-Type: application/DHT+XML
Content-Length: 142
Require: P2P-DHT
Supported: P2P-DHT
//empty line
<?xml version = "1.0"?>
<DHT-info>
        <DHT-description>
                <Overlay-name>Concordia DHT </Overlay-name>
                <Hash-alg>SHA-1</Hash-alg>
        </DHT-description>
        <DHT-operation> nodeRegistration </DHT-operation>
        <DHT-node> sip: 54ab7cd8@10.0.0.32;expires=800</DHT-node>
</DHT-info>
```

Figure 4.4: An example of a node registration request

4.6.3.2 Interpretation of node update

When a node receives a node registration request, it can determine that this is a DHT SIP message based on the presence of the P2P-DHT *Require* and *Supported* fields. If the node does not support the DHT extension, it must reply with a *501 Not Implemented* message which indicates that the server is unable to process the request because it is not supported. If a node examines the overlay name and determines that it has not the same overlay name, the node must reject the message with a *488 Not Acceptable Here* response which indicates that some aspect of the proposed session is not acceptable and may contain a *Warning* header field indicating the exact reason.

If a DHT node receives a non DHT request, it should reject the request with a "*421 Extension Required*" message which indicates that a server requires an extension to process the request that was not present in a *Supported* header field in the request.

The presence of the Contact header and a valid expiration time indicates that the message is a registration update. To determine that the node is a registration one, we must parse the XML body and analyses the *DHT-operation* tag that must be in this case *nodeRegistration.*

The process of the node registration is performed by the DHT layer following the steps described below.

Step 1: Checking the node ID

When the bootstrap node receives the registration request, it should verify that the calculated hash of the sending node ID corresponds to its URI address by hashing the IP address and the port number found in *DHT-Node* tag in the XML body and comparing it to the sending node ID. If the hash does not match with the node ID, the receiving node must reject this request with "*493 Undecipherable*" message.

Step 2: Checking the *TO* field

The bootstrap node should also verify that the calculated hash node ID in the *TO* field corresponds to the URI address of the sending node. If the hash does not match with the node ID, the receiving node must reject the request with "*493 Undecipherable*" message.

Step 3: The node routing

The bootstrap node analyses the node ID to determine if its successor is responsible for this ID space portion. If it does, the bootstrap node responds with a "*200 OK*" message including its successor in the XML body. If not it will lookup into its finger table and provide the joining node with a closer node that can have more information about the namespace where the joining node will be stored.

If the successor's bootstrap node is not responsible of the namespace of the joining ID, the bootstrap generates a "*302 Moved Temporarily message*". The 302 message is constructed following the rules in RFC3261 and the following two rules.

Rule 1

The bootstrap node looks up into in its finger table and successor list for the node nearest to the joining node's Node-ID, and uses it to create a contact field in the form of a node URI, as specified in the node message (see Section 4.5.1).

Rule 2

The response must contain a valid DHT-Node tag in the XML body. This response is sent to the joining node.

Example 4.2: Node registration redirect

Using the node registration example of the previous section (see Example 4.1), the bootstrap node at address 10.0.0.21 analyses the node registration request, performs a lookup operation in its finger table and then determines that the nearest node for the joining node N32 is N38. Finally, the bootstrap node redirects the joining node to the node N38 by putting the address of the nearest node N38 in the *Contact* headers.

Further assume that the hash of N38's IP address 10.0.0.38 is 64cb1ad2 using SHA1.

A "*302 redirect message*" will look like as described in Figure 4.5.

```
SIP/2.0 302 Moved Temporarily
TO: sip: 54ab7cd8@10.0.0.32
From: sip: 54ab7cd8@10.0.0.32
Contact: sip: 64cb1ad2@10.0.0.38
Content-Type: application/DHT+XML
Content-Length: 142
Require: P2P-DHT
Supported: P2P-DHT
//empty line
<?xml version = "1.0"?>
        <DHT-info>
                <DHT-description>
                        <Overlay-name>Concordia DHT </Overlay-name>
                        <Hash-alg>SHA-1</Hash-alg>
                </DHT-description>
                <DHT-operation>nodeRegistration</DHT-operation>
                <DHT-node> sip: a4fb79d2@10.0.0.21;expires=1800</DHT-node>
        </DHT-info>
```

Figure 4.5: A 302 redirect message

Step 4: The node redirection

When the joining node receives the "*302 redirect message*", it uses the address in the *Contact* header as the new bootstrap node, and sends again a node registration request.

This process is repeated until the joining node finds the nearest node which is responsible for this namespace area where the joining node will be stored. The bootstrap node knows that its successor is responsible for this joining node by checking if the node ID of the joining node falls between its node ID and its successor node. Once the admitting node is found, the bootstrap responds with a "*200 OK*" and must provide its four successors and its predecessor in the XML body. The successors are transmitted in a *DHT-Successor-list tag* and the predecessor is transmitted in a *DHT-Predecessor-node* tag.

Step 5: Notifying the successor

The joining node sends a final register request to its first successor in the *DHT-Successor-list* obtained by the previous "*200 OK*" response.

The DHT-operation in the XML body must be "Notify and copy entry". The admitting node knows that it must store the mapping of the joining node and it does not route it by checking the *DHT-operation* tag in the XML body. The admitting node should also check if the node ID hash is valid and falls between its node ID and its predecessor before storing the joining node. If the node ID hash is not valid, the request should be rejected with a response of "*493 Undecipherable*" and if admitting node determines that it is not responsible for the namespace of the joining ID, it responds with a 302 message to redirect the request to its predecessor. If this situation happens, it means that the predecessor of the admitting node has recently joined the overlay and it is not yet stable.

In addition to verifying that the node ID was properly calculated, the admitting node may require an authentication challenge before registering the joining node.

Step 6: Confirming the registration

Once this security aspect is done, the admitting node will reply with a "*200 OK*" message to the joining node. As in a traditional registration, the *Contact* in the "*200 OK*" will be the same as in the request.

Further the admitting node is responsible for helping the joining node to become a member in the overlay. This is done by providing the joining node with the neighbors of the admitting node.

The admitting node replies with a "*200 OK*" message if and only if the joining node will reside between its node ID and its predecessor. The admitting node must provide its predecessor and a list of four successors to help the joining node to enter the overlay.

The predecessor and the list of successors are provided in the "*200 OK*" message in the XML body.

The predecessor must be transmitted in a *DHT-Predecessor-node* tag. The successor must be transmitted in a *DHT-Successor-list* tag and using a *DHT-Successor-node* tag followed by the depth level of the successor. The "*200 OK*" response should contain the next four successor nodes, to use for redundancy. All nodes should maintain 4 successors at any time for redundancy. Additionally, the admitting node must include a *DHT-Node* tag containing the admitting node's node ID and its IP address:

```
<DHT-node> sip: 64cb1ad2@10.0.0.38 </DHT-node>
< DHT-Predecessor-node> sip: a4fb79d2@10.0.0.21</ DHT-Predecessor-node>
< DHT-Successor-list >
        <!for clarity we show only 2 entries!>
        < DHT-Succesor-node1>sip: c4db68a1@10.0.0.51</ DHT-Succesor-node1>
        < DHT-Succesor-node2> sip: c8ac58b2@10.0.0.56</ DHT-Succesor-node12>
</ DHT-Successor-list >
```

When a node joins the overlay, it may split the area namespace of an admitting node in two parts and some bindings will be the responsibility of the joining node and they will be transferred to it. The admitting node must include these immigrant bindings into a *DHT-Resource-list* tag in the XML body.

Step 7: Updating the joining node

Once the joining node receives the "*200 OK*" response, it obtains the node ID and the IP address of the admitting node and sets it as its successor node. Also the admitting node must set its predecessor to the joining node and it obtains this last information from the *DHT-node* tag in the register request. The admitting node's successor must not be changed.

Once the joining node becomes a member of the overlay, the admitting node can help it to fulfill its finger table by sending it a copy of the entries of its finger table, using a *DHT-Finger-table* tag in the XML body of the "*200 OK*" response.

As the joining node will likely be nearby the admitting node in the hash namespace, the successor list of the joining node can be fulfilled from the successor list of the admitting node. We can also use temporarily the finger table entries of the admitting node if they fit the joining node finger table intervals. The temporarily table can improve the performance of the query requests but the joining node must recalculate all the entries. It must also reconstruct the intervals by applying the offset of each finger and performing a search for each start of these intervals. This process is similar to the *fix finger* method in the Chord algorithm with the difference that all the used messages are SIP query requests as described below.

Example 4.3: Node registration confirmation

Continuing the node registration example described above (see Example 4.1), assume that the node ID 64cb1ad2 with the IP address 10.0.0.38 is responsible for the namespace of the node ID 54ab7cd8. The admitting node sends its predecessor, a list of 4 successors and its finger table.

For clarity we show only two successors and three entries of the finger table.

A node registration confirmation would look like as described in Figure 4.6.

```
SIP/2.0 200 OK
TO: sip: 54ab7cd8@10.0.0.32
From: sip: 54ab7cd8@10.0.0.32
Contact: sip: 54ab7cd8@10.0.0.38
Expires: 800
Content-Type: application/DHT+XML
Content-Length: 142
```

Figure 4.6: Node registration confirmation

```
Require: P2P-DHT
Supported: P2P-DHT
//empty line
<?xml version = "1.0"?>
<DHT-info>
<DHT-description>
<DHT-Overlay-name>Concordia DHT </DHT-Overlay-name>
<DHT-Hash-alg>SHA-1</DHT-Hash-alg>
</DHT-description>
<DHT-Operation> nodeRegistration</DHT-Operation>
<DHT-node> sip: 64cb1ad2@10.0.0.38 </DHT-node>
<DHT-Predecessor-node> sip: a4fb79d2@10.0.0.21</DHT-Predecessor-node>
<DHT-Successor-list >
<!for clarity we show only 2 entries!>
< DHT-Succesor-node1>sip: c4db68a1@10.0.0.51</ DHT-Succesor-node1>
< DHT-Succesor-node2> sip: c8ac58b2@10.0.0.56</ DHT-Succesor-node2>
</ DHT-Successor-list >
<DHT-Finger-table>
<!for clarity we show only 3 entries!>
<DHT-finger-node>sip: c4db68a1@10.0.0.51</ DHT-finger-node >
< DHT-finger-node > sip: c4db68a1@10.0.0.51</ DHT-finger-node >
< DHT-finger-node > sip: c4db68a1@10.0.0.51</ DHT-finger-node >
</DHT-Finger-table>
</DHT-info>
```

Figure 4.6 continued: Node registration confirmation

4.6.4 Node query

4.6.4.1 Construction of node query

The node query is an operation used to find or to locate the node that it is responsible for a particular hash space. This operation is similar to the *find_successor* in Chord.

The node query request is the same as a node update request except that there is no *Contact* header in the message. As with traditional SIP, *REGISTER* messages that are sent without a *Contact* header are assumed to be queries.

If a user wants to search for a particular resource or finds a specific node, a register node request with no *Contact* header is used.

To construct a node query message, the node *n* must follow the steps described below.

Step 1: lookup for the closest preceding node

Node *n* looks for the entry of the closest preceding node among its finger entries that covers the range of the resource-ID or node ID that it wishes to search following these three rules.

Rule 1

The querying node *n* should start searching for the appropriate node in its finger table; the querying node must start by the last entry in the finger table *finger[i]* and checks if the finger of this last entry falls between the querying node *n* and the key ID of the node which we are looking for, i.e., finger[i] ∈]n, key ID[.

Rule 2

The querying node *n* should also perform the same search in the successor list, i.e., successor $\in$]n, key ID[.

Rule 3

Finally the querying node *n* should try if the key ID falls between the predecessor and the querying node, i.e., key ID $\in$]predecessor, n[.

Step 2: Sending the query request

The querying node should send the request to the best node (closest node) which satisfies the appropriate conditions.

Even if the finger table is not up to date, i.e., the node has not yet completed calculating its entries and reconstructing its intervals, these initial searches may be less efficient but they will succeed and the probability that they occur is very low.

All the headers are constructed in the same manner as in a node update request.

The only difference is that there is no *Contact* and *Expires* header since it is a location request and not a binding message.

The node should include the same XML body as in the update request and it must also include the *Require* and *Supported* headers to indicate that this request must be processed as a DHT level.

Example 4.4: A node query request

Assume that a node running on IP address 192.168.1.100 on port 5321 wishes to locate the node responsible for the node ID 1fa3bd78 and asks the node with IP address 192.168.1.151.

An example of the message's header would look like as described in Figure 4.7.

```
Register sip: 192.168.1.151 SIP/2.0
TO: sip: 1fa3bd78@192.168.1.100
From: sip: 2ca5b4c3@192.168.1.100
Content-Type: application/DHT+XML
Content-Length: 142
Require: P2P-DHT
Supported: P2P-DHT
//The body is not shown
```

Figure 4.7: An example of a node query request

The username part of the *To* URI must be set to the node ID we want to search and the IP address part must be set to the IP of the sending node. The *From* URI must be constructed from the hashed node ID of the sending node and its IP address.

The receiving node must also check the validity of the *From* and *To* URI.

4.6.4.2 Interpretation of node query

When a node receives a node registration query, it can determine that the request is a DHT SIP message based on the presence of the P2P-DHT *Require* and *Supported* fields. If the node does not support the DHT extension, it must reply with *501 Not Implemented* message which indicates that the peer is unable to process the request because it is not supported.

If a node examines the overlay name and determines that it has not the same overlay name, the node must reject the message with *488 Not Acceptable Here* response which indicates that some aspect of the proposed session is not acceptable and may contain a *Warning* header field indicating the exact reason.

If a DHT node receives a non DHT request, it should reject the request with a "*421 Extension Required*" message which indicates that the peer requires an extension to process the request that was not present in a *Supported* header field in the request.

The lack of the *Contact* header and the expiration time indicate that the request is a node query. To determine that this is a node registration and not user registration, we must parse the XML body and analyses the *DHT-Operation-type* tag that must be in this case *nodeRegistration.*

The process of the node query is performed by the DHT layer and follows steps described below.

Step 1: Checking the node ID

The receiving node should verify that the calculated hash of the sending node ID corresponds to its URI address by hashing the IP address and the port number found in *DHT-Node* tag in the XML body and comparing it to the sending node ID.

If the hash does not match with the node ID, the receiving node must reject this request with "*493 Undecipherable*" message.

Step 2: Checking if the successor of the receiving node is the responsible for the searched node ID

The receiving node analyses the node ID to determine if its successor is responsible for this ID space portion. If it does, the receiving node responds with a *"200 OK"* message if the node ID that we are looking for is equal to its successor node ID and responds with a "*404 Not found*" if the node ID is different from the successor node.

If the successor of the receiving node is not responsible for the searched node ID, it will lookup a closer node into its finger table, successor list and its predecessor as described above (see Section 4.6.4).

Step 3: Lookup for the closest node

The receiving node selects the closest node among its finger table entries which could have more information about the namespace where the resource-ID or the node ID is stored.

Then, the receiving node generates a "*302 Moved Temporarily message*" and sends it to the querying node. The 302 message is constructed following the rules in the RFC 3261 and the following rules.

Rule 1

The receiving node uses its closest node to create a contact field in the form of a node URI, as specified in the Message Node (see Section 4.5.1) section of this document.

Rule 2

The 302 response message must contain a valid DHT-Node tag in the XML body.

Step 4: Routing of the query request

When the querying node receives the "*302 redirect message*", it uses the address in the *Contact* header as the new query node and sends again a node query request.

This process is repeated until the querying node finds the nearest node which is responsible for the namespace area where the node which we are looking for resides.

Step 5: Sending back the result of the query request

If the responsible node (successor of the receiving node) matches exactly the search key, the receiving node sends a *"200 OK"* message to the querying node. If it is responsible for the node's namespace but its node ID is different from the search key, the receiving node sends a "*404 Not Found*" message.

The predecessor and the list of successors are provided in the *"200 OK"* or *"404 Not Found"* message in the XML body.

The predecessor is transmitted in a *DHT-Predecessor-node* tag. The successor is transmitted in a *DHT-Successor-list* tag and using a *DHT-Successor-node* tag followed by the depth level of the successor. The *"200 OK"* or *400 Not found* response contains the next four successor nodes since the first successor is the responsible node for the search key namespace. In addition, the admitting node includes a *DHT-Node* tag containing the admitting node's node ID and its IP address

4.7 Node shutdown or failure

There are two ways for a node to leave the overlay; the graceful leaving and the node failure.

The graceful leaving consists to shutdown the user agent and the node updates the state of its neighbors before leaving the overlay. If a node is shutting down, the registration refresh operation ensures that its user records will gracefully be transferred to its successors. The UA sends a SIP Register message to the DHT nodes that will hold the user records after the node leaves (see Section 4.6.4). This guarantees that other users can locate the record when the DHT node will have gracefully shut down. In addition, the leaving node must unregister itself from its successors and its predecessor. This is done by sending a node registration message to these neighbors with the only exception that the *expires* header or field must be set to zero. The leaving node must include in its request the predecessor node and the list of 4 successors to allow the node receiving the request to correct their predecessor.

When a node fails abnormally, its predecessor node detects the failure by performing the stabilizing task as described below (see Section 4.8), and adjust its successor links. The 4-way redundancy registration ensures that the mapping will not be lost, unless the four successors of the leaving node fail simultaneously.

If a node fails to contact another node, the caller node may perform a search operation to update its neighbor's links if it is a finger link or it may remove this link if it is a successor link.

If the link that has failed was in the finger table, a querying node request should be performed to the failed entry of the finger table. If the successor fails, the next successor in the successor list must replace it. After updating these links, the request should be repeated based on the new entries.

4.8 DHT stabilization task

In practice, the overlay needs to deal with nodes joining the system and with nodes that fail or leave voluntarily. This section describes how Chord handles these situations.

In order to ensure that lookups execute correctly as the set of participating nodes changes, the DHT overlay must ensure that each node's successor pointer is up to date. It does this using a "stabilization" protocol that each node runs periodically in the background and which updates the finger tables, successor pointers and the predecessor.

Every node must run the stabilize protocol periodically to learn about the nodes which recently joined or left the system.

The stabilize task is divided into three operations which are successor refresh, finger refresh and predecessor refresh.

4.8.1 Successor refresh

Every node must periodically contact its first successor and analyses the response. Let us consider a given node *X*. If the predecessor node received in the response is different from the node *X*, *X* must update its own successor with the returned predecessor. Also, it should try to insert the successors received in the response in its own successor list and finger table if they are better, i.e., closer than its current entries. Further the contacted node must insert the node *X* as its predecessor if it does not yet have a predecessor or if the node *X* is closest than its own predecessor.

If the node which has been contacted does not respond, it must be removed from the successor list and replaced by the second successor in the list. As we should always keep four successors in the successor list, the node should try to find a potential successor in its finger table and insert it at the end of the successor list to fill the gap.

To contact its successor, the node *X* sends a *Register* request with a *Node notification* as a DHT-operation-type in the XML body.

The contacted node knows that this is a node refresh request and not a node join request based on the DHT-operation-type.

This process is used to update the overlay when a new node joins the system. It is similar to *stabilize* and *notify* process in Chord protocol.

4.8.2 Finger refresh

Every node must periodically check its finger intervals and recalculate its entries.

Every period, the node n should check one entry starting by the finger at position $n + 2^i$ where $i \in [0,$ *last entry*$]$. We recommend setting the value of i to the *last entry* at the start of the stabilize task and decrement it by 1 at each period. When i becomes equal to 0, we set it to the *last entry*.

The node uses the node query operation (see Section 4.6.4) to lookup for the finger entries.

If the new entry found is better than the current one, the new entry is inserted in the appropriate interval.

4.8.3 Predecessor refresh

This operation is also called periodically to check whether the predecessor has failed.

This request consists in sending a ping to the predecessor to check if it is still alive.

If not, the node must set the predecessor value to Null.

4.9 User registration

4.9.1 Overview

When a node enters the overlay and becomes a stable member, it should register the contact user it is responsible for. This registration is called user registration and consists to map the SIP ID to the node IP address.

4.9.2 User registration update

The user registrations updates are similar to the ordinary SIP user registrations described in RFC 3261. The register-URI is constructed only by the IP address of the node wishing to be contacted. The To and From header must contain a DHT user URI (as it is a user message), the username must be the unhashed user ID and the IP address must be the address of the UA (node) participating in the overlay (see Section 4.2).

The caller node must provide a contact header when registering in order to distinguish that this is a user update and not a user query. The URI in the contact header must be constructed as described in Section 4.2. The *expires* field or header must have a positive value rather than zero as in standard SIP, *Expires* header with zero value is used to remove registrations.

The node must include *Require* and *Supported* headers with the option tag "P2P-DHT"

We include in the XML body, the overlay name and the hashing algorithm. The content of the *DHT-operation* tag must be:

<DHT-Operation>userRegistration</DHT-Operation>

The body must also provide a DHT-node tag containing the node ID and the IP address of the appropriate node and should include a <DHT-Resource-ID> tag which is the hashing of the user ID.

To route the registration message, the UA hashes the user ID to obtain a resource ID corresponding to the user's user ID. This request is then forwarded to the closest neighbor covering the resource ID interval. The message is routed in the same manner as a node query request (see Section 4.6.1).

When the joining user receives the "*302 redirect message*", he uses the address in the *Contact* header as the new register node and sends again a user registration request.

This process is repeated until the joining user finds the nearest node which is responsible for this namespace area where the joining user will be stored.

Once the responsible node is found, the joining user sends it a user registration request as described above.

The register node determines that the request is a user registration rather than a node registration based on the *DHT-operation* tag content which must be *userRegistration.*

This register node responds with a *"200 OK"* message and includes a list of four successor nodes.

The caller node should use the received successors to send redundant registrations to these nodes. The purpose of these registrations is to improve the robustness and the lookup correctness of the system.

A node determines that the request is a redundant registration and it doesn't route it based on *DHT-operation* tag content which must be *userRedundantRegistration.*

If a registering user wants to call another user in the overlay, he should send an *Invite* message following these steps.

- The caller hashes the user ID of the called and obtains a resource ID for that user.
- The caller searches for this resource ID as described in the user registration query section.
- Once the node responsible for this resource ID is found, it will provide either a *200* message with a *Contact* header for the called UA or a *404* message if the mapping information of the called node is not found.
- Finally the communication establishment will follow the same steps as in a standard SIP communication.

4.9.3 User registration query

The user registration query is an operation used to find or to locate a user. This operation is usually done before establishing a connection. The only difference between the registration query and the registration update is that the list of bindings is left unchanged when it is a query request and there is no *Contact* header in this type of request.

The user query request is composed of two operations; the first one consists in looking for the responsible node which will hold the user mapping. This first operation is performed by the node query request (see Section 4.6.1). The second operation consists in checking if the responsible node still stores the mapping of the querying node.

If yes, the responsible node sends a *"200 OK"* with a Contact header for the called UA and if not, it sends a *"404 Not found"* message.

4.9.4 User registration refresh

Each UA is responsible for refreshing the bindings that it has previously established.

The UA issues a *REGISTER* request for each of its bindings before the expiration interval has elapsed. Further the UA may cache the address of the admitting node when it attempts to register for the first time and uses it to refresh its registration time. However, the refreshing request is not guaranteed to succeed since it may be that another node that has recently joined the overlay becomes now responsible of this binding.

In this case, the agent receives a 302 from the node with which it communicated and it must locate the new responsible node for this binding as if it is a new registration.

Also, when the UA receives the *"200 OK"* response from the register node, it should send a redundant registration to the successors provided by the register node.

4.9.5 Removing user registration

Registrations are soft states and expire unless refreshed, but can also be explicitly removed. A UA requests the immediate removal of a binding by specifying an expiration interval of zero for that contact address in a *REGISTER* request.

Further, the UA must send a registration removal to the successors provided by the *"200 OK"* response in order to remove the redundant bindings.

4.10 Examples

In this section, we illustrate the two main functions of the SIP2P system described in the previous sections which are node registration and user registration.

Consider an overlay of size 64 with 5 nodes such that their IDs are 8, 21, 38, 51and 56. We will assume a 6 bit hash to simplify the exposition. All five nodes are assumed stable with their neighbor entries filled as described in Table 4.2.

Every node represents a user agent (UA) and it is responsible for registering one user. Let us focus on users Bob and Alice. Bob‘s UA is collocated with N8 and for simplicity we assume that his IP address is 10.0.0.8. Alice‘s UA is collocated with N21 and her IP address is 10.0.0.21.

The resource ID of each user is stored in the first node which has a node ID greater or equal to resource ID. Assume that Bob's resource ID is 30 and Alice's resource ID is 48 so they will be stored respectively in N38 and N51.

For the purpose of the example, each node is assumed to have only two successors in its successor list, six entries in its finger table and one predecessor.

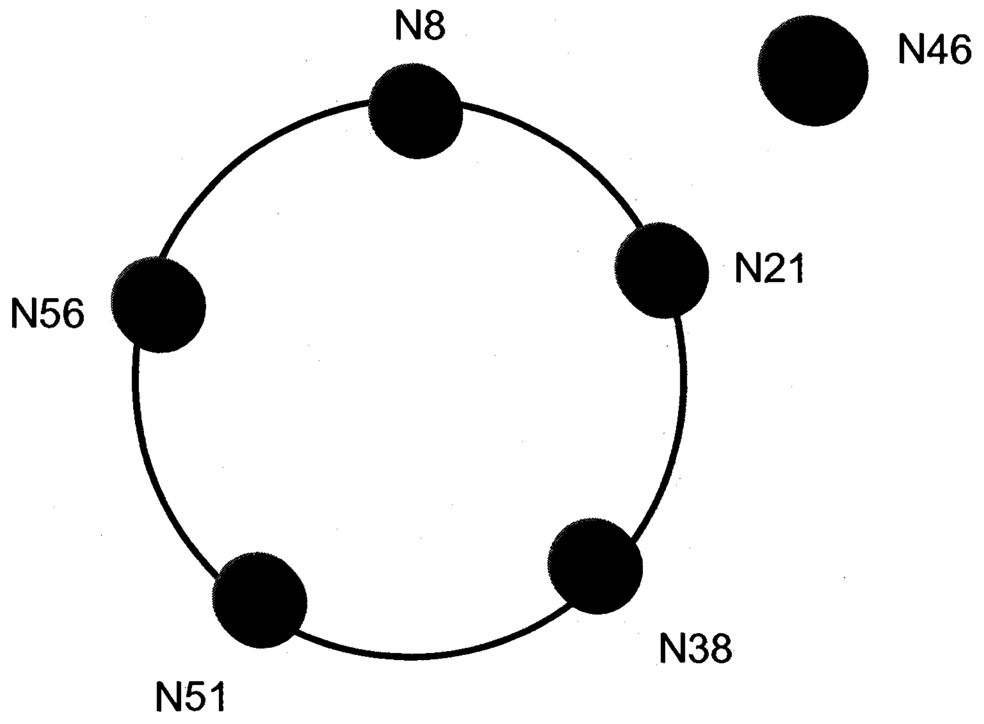


Figure 4.8: a 6 bit overlay with 5 nodes and 2 resources ID

Node N	N8	N21	N38	N51	N56
	Finger table				
N+1	N14	N38	N51	N56	N8
N+2	N14	N38	N51	N56	N8
N+4	N14	N38	N51	N56	N8
N+8	N21	N38	N51	N8	N8
N+16	N38	N38	N56	N8	N8
N+32	N51	N56	N8	N21	N38
	Successor list				
	N21	N38	N51	N56	N8
	N38	N51	N56	N8	N21
	Predecessor				
	N56	N8	N21	N38	N51

Table 4.2: neighbor entries

4.10.1 Node registration

Assume further that a new node wishes to join the overlay (see Figure 4.9). The node has an IP address of 10.0.0.46, which we shall assume hashes to a Node-ID of 46. The bootstrap node that was discovered is N21.

We also assume that node N51 stores the mapping of Carl's user whose user agent is collocated with node N38 and the hash of its user ID is 42.

- The joining node N46 constructs a Register message and sends it to the bootstrap node N21. Node N21 checks if the IP address of N46 hashes to 46 and then verifies if its successor controls this part of the namespace. Since, it is not the case here, it looks up in its neighbor entries and determines the best node (closest preceding node) that it can route the look up message of node N46 which is node N38 in the example.
- Node N46 constructs another Register message to node N38. The latter node verifies first if the hash of the IP address 10.0.0.46 is 46 and then determines that its successor is responsible of node N46 since node N46 falls between node N38 and its successor.

- Node N38 sends back a *"200 OK"* message including the responsible node as the first successor in the successor list.

- Node N46 sends its final registration message to the responsible node (N51) which stores the mapping of the joining node. When node N46 joins the overlay, it splits the area namespace of node N51 in two parts. Some user bindings whose user ID are between N38 and N46 will be the responsibility of joining node N46 and they will be transferred to it in the XML body of the "200 OK" response. In the example, Carl's user binding will move the new joining node N46.

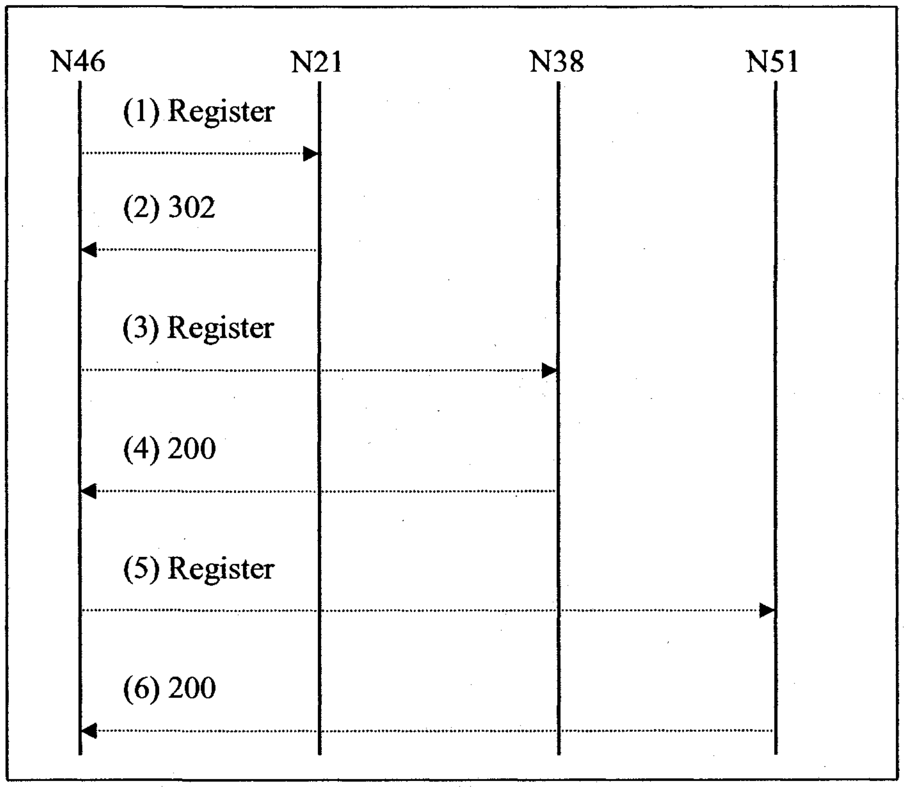


Figure 4.9: Example of a node registration

SIP message sent from node N46 to node N21.

Node 46 → Node 21

Register sip:10.0.0.21 SIP/2.0

TO: sip:46@10.0.0.46

From: sip:46@10.0.0.46

Contact: sip:46@10.0.0.46

Expires: 800

Content-Type: application/DHT+XML

Content-Length: 142

Require: P2P-DHT

```
Supported: P2P-DHT
//empty line
<?xml version = "1.0"?>
<DHT-info>
        <DHT-description>
                <DHT-Overlay-name>Concordia DHT </ DHT-Overlay-name >
                <DHT-Hash-alg>SHA-1</ DHT-Hash-alg >
        </DHT-description>
        <DHT-operation >nodeRegistration</ DHT-operation>
        <DHT-node> sip:46@10.0.0.46</DHT-node>
</DHT-info>
```

SIP message sent from node N21 to node N46.

Node 21 → Node 46

```
SIP/2.0 302 Moved Temporarily
TO: sip:46@10.0.0.46
From: sip:46@10.0.0.46
Contact: sip:38@10.0.0.38
Expires: 800
Content-Type: application/DHT+XML
Content-Length: 142
Require: P2P-DHT
```

```
Supported: P2P-DHT

//empty line

<?xml version = "1.0"?>

<DHT-info>

        <DHT-description>

                <DHT-Overlay-name>Concordia DHT </ DHT-Overlay-name >

                <DHT-Hash-alg>SHA-1</ DHT-Hash-alg >

        </DHT-description>

        <DHT-operation >nodeRegistration</ DHT-operation>

        <DHT-node> sip:21@10.0.0.21</DHT-node>

</DHT-info>
```

SIP message sent from node N46 to node N38.

Node 46 → Node 38

```
Register sip:10.0.0.38 SIP/2.0

TO: sip:46@10.0.0.46

From: sip:46@10.0.0.46

Contact: sip:46@10.0.0.46

Expires: 800

Content-Type: application/DHT+XML

Content-Length: 142

Require: P2P-DHT
```

```
Supported: P2P-DHT
//empty line
<?xml version = "1.0"?>
<DHT-info>
        <DHT-description>
                <DHT-Overlay-name>Concordia DHT </ DHT-Overlay-name >
                <DHT-Hash-alg>SHA-1</ DHT-Hash-alg >
        </DHT-description>
        <DHT-operation >nodeRegistration</ DHT-operation>
        <DHT-node> sip:46@10.0.0.46</DHT-node>
</DHT-info>
```

SIP message sent from node N38 to node N46.

Node 38 → node 46

```
SIP/2.0 200 OK
TO: sip:46@10.0.0.46
From: sip:46@10.0.0.46
Contact: sip:46@10.0.0.46
Expires: 800
Content-Type: application/DHT+XML
Content-Length: 142
Require: P2P-DHT
```

```
Supported: P2P-DHT

//empty line

<?xml version = "1.0"?>

<DHT-info>

        <DHT-description>

                <DHT-Overlay-name>Concordia DHT </ DHT-Overlay-name >

                <DHT-Hash-alg>SHA-1</ DHT-Hash-alg >

        </DHT-description>

        <DHT-operation >nodeRegistration</ DHT-operation>

        <DHT-node> sip:38@10.0.0.38</DHT-node>

        <DHT-Predecessor-node> sip:21@10.0.0.21</DHT-Predecessor-node>

        <DHT-Successor-list >

        !for clarity we show only 2 entries!>

                < DHT-Succesor-node1>sip:51@10.0.0.51</ DHT-Succesor-node1>

                < DHT-Succesor-node2> sip:56@10.0.0.56</ DHT-Succesor-node2>

        </DHT-Successor-list >

</DHT-info>
```

SIP message sent from node N46 to node N51.

Node 46 → node 51

```
Register sip:10.0.0.51 SIP/2.0

TO: sip:46@10.0.0.46

From: sip:46@10.0.0.46

Contact: sip:46@10.0.0.46

Expires: 800

Content-Type: application/DHT+XML

Content-Length: 142

Require: P2P-DHT

Supported: P2P-DHT

//empty line

<?xml version = "1.0"?>

<DHT-info>

        <DHT-description>

                <DHT-Overlay-name>Concordia DHT </ DHT-Overlay-name >

                <DHT-Hash-alg>SHA-1</ DHT-Hash-alg >

        </DHT-description>

        <DHT-operation > Notify and copy entry </ DHT-operation>

        <DHT-node> sip:46@10.0.0.46</DHT-node>

</DHT-info>
```

SIP message sent from node N51 to node N46.

Node 51 → node 46

```
SIP/2.0 200 OK

TO: sip:46@10.0.0.46

From: sip:46@10.0.0.46

Contact: sip:46@10.0.0.46

Expires: 800

Content-Type: application/DHT+XML

Content-Length: 142

Require: P2P-DHT

Supported: P2P-DHT

//empty line

<?xml version = "1.0"?>

<DHT-info>

        <DHT-description>

                <DHT-Overlay-name>Concordia DHT </ DHT-Overlay-name >

                <DHT-Hash-alg>SHA-1</ DHT-Hash-alg >

        </DHT-description>

        <DHT-operation > Notify and copy entry </ DHT-operation>

        <DHT-node> sip:51@10.0.0.51</DHT-node>

        <DHT-Predecessor-node> sip:21@10.0.0.21</DHT-Predecessor-node>
```

```
<DHT-Successor-list >

!for clarity we show only 2 entries!>

          < DHT-Succesor-node1>sip:51@10.0.0.51</ DHT-Succesor-node1>

          < DHT-Succesor-node2> sip:56@10.0.0.56</ DHT-Succesor-node2>

</DHT-Successor-list >

<DHT-Finger-table>

<!for clarity we show only 3 entries!>

          < DHT-finger-node >sip:51@10.0.0.51</ DHT-finger-node >

          < DHT-finger-node > sip:51@10.0.0.51</ DHT-finger-node >

          < DHT-finger-node > sip:51@10.0.0.51</ DHT-finger-node >

</DHT-Finger-table>

<DHTResourceList>

          <DHTResource>

          <DHTResourceID>42</DHTResourceID>

          <DHTResourceURI>sip:carl@10.0.0.38</DHTResourceURI>

          <DHTResourceUserName>carl@concordia.ca</DHTResourceUserName

>

          <DHTResourceExpire>45</DHTResourceExpire>

          </DHTResource>

</DHTResourceList>

</DHT-info>
```

4.10.2 User Registration

Assume further that Alice wishes to join the overlay. Alice's UA is collocated with node N21 and Alice's user ID is Alice@concordia.ca. The hash of Alice's user name leads to a resource ID of 48 and her contact is Alice@10.0.0.21. In the example below (see Figure 4.10), Alice's resource ID is stored by the node responsible of this part of namespace in the overlay, i.e., node N51.

- Alice's UA performs a lookup for the admitting node of Alice's ID. So Alice's UA makes a node query request for ID 48.

- Alice's UA determines the best node in its neighbor entries that can route this message. In the context of the example, the Register message will be routed to node N38. Node N38 verifies the resource ID by comparing it with the hash of the user ID. Then node N38 determines that its successor is responsible for this part of the namespace so it sends back a *404* message with the responsible node which is node N51 as a first successor in the successor list. In this case, the responds is for a node query request, so it is a *404* and not a 200 message because the admitting node ID is different from Alice's ID

- Alice's UA sends a user registration request to the admitting node (node N51). Node N51 answers with a 200 message to confirm its admission to the resource ID.

- Node N21 sends redundant registration to the four successors of node N51.

In the example described in Figure 4.10, we show only one redundant message which it is send to node N56.

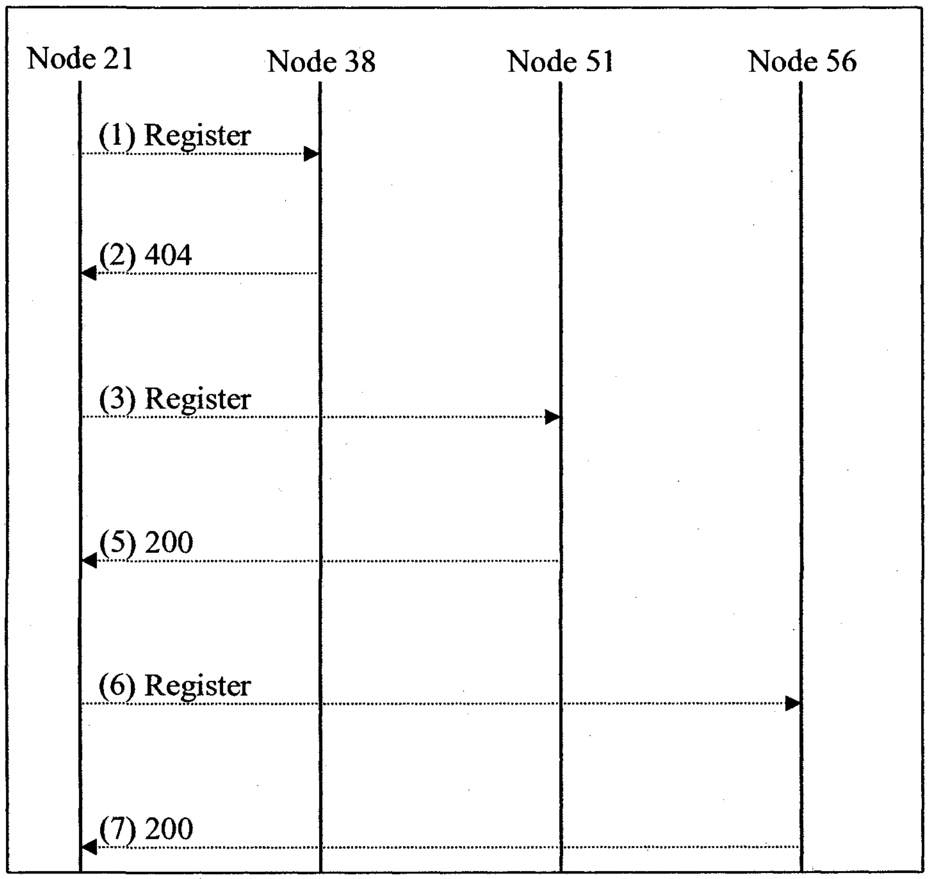


Figure 4.10: An example of a user registration

SIP message sent from node N21 to node N38.

Node 21 → Node 38

```
Register sip:10.0.0.38 SIP/2.0
TO: sip:Alice@concordia.ca
From: sip:Alice@concordia.ca
Content-Type: application/DHT+XML
Content-Length: 142
Require: P2P-DHT
Supported: P2P-DHT
//empty line
<?xml version = "1.0"?>
<DHT-info>
        <DHT-description>
                <DHT-Overlay-name>Concordia DHT </ DHT-Overlay-name >
                <DHT-Hash-alg>SHA-1</ DHT-Hash-alg >
        </DHT-description>
        <DHT-operation > nodeRegistration </ DHT-operation>
        <DHT-node> sip:21@10.0.0.21</DHT-node>
</DHT-info>
```

SIP message sent from node N38 to node N21.

Node 38 → Node 21

```
SIP/2.0 404 Not Found

TO: sip:Alice@concordia.ca

From: sip:Alice@concordia.ca

Content-Type: application/DHT+XML

Content-Length: 142

Require: P2P-DHT

Supported: P2P-DHT

//empty line

<?xml version = “1.0”?>

<DHT-info>

        <DHT-description>

                <DHT-Overlay-name>Concordia DHT </ DHT-Overlay-name >

                <DHT-Hash-alg>SHA-1</ DHT-Hash-alg >

        </DHT-description>

        <DHT-operation > nodeRegistration </ DHT-operation>

        <DHT-Successor-list >

        !for clarity we show only 2 entries!>

                < DHT-Succesor-node1>sip:51@10.0.0.51</ DHT-Succesor-node1>

                < DHT-Succesor-node2> sip:56@10.0.0.56</ DHT-Succesor-node2>

        </DHT-Successor-list >
        <DHT-node> sip:38@10.0.0.38</DHT-node>
</DHT-info>
```

SIP message sent from node N21 to node N51.

Node 21 → Node 51

```
Register sip:10.0.0.51 SIP/2.0
TO: sip:Alice@concordia.ca
From: sip:Alice@concordia.ca
Contact: sip:Alice@10.0.0.21
Content-Type: application/DHT+XML
Content-Length: 142
Require: P2P-DHT
Supported: P2P-DHT
//empty line
<?xml version = "1.0"?>
<DHT-info>
        <DHT-description>
                <DHT-Overlay-name>Concordia DHT </ DHT-Overlay-name >
                <DHT-Hash-alg>SHA-1</ DHT-Hash-alg >
        </DHT-description>
        <DHT-operation > userRegistration </ DHT-operation>
        <DHT-Resource-List>
        <DHT-Resource>
            <DHT-Resource-ID>48</DHT-Resource-ID>
                <DHT-Resource-    URI>sip:Alice@10.0.0.21    </DHT-
            Resource-URI>
                <DHT-Resource-UserName>Alice@concordia.ca</DHT-
            Resource-UserName>
            <DHT-Resource-Expire>45</DHT-Resource-Expire>
        </DHT-Resource>
    </DHT-Resource-List>

    <DHT-node> sip:21@10.0.0.21</DHT-node>
</DHT-info>
```

SIP message sent from node N51 to node N21.

Node 51 → Node 21

```
SIP/2.0 200 OK
TO: sip:Alice@concordia.ca
From: sip:Alice@concordia.ca
Contact: sip:Alice@10.0.0.21
Content-Type: application/DHT+XML
Content-Length: 142
Require: P2P-DHT
```

```
Supported: P2P-DHT
//empty line
<?xml version = "1.0"?>
<DHT-info>
        <DHT-description>
                <DHT-Overlay-name>Concordia DHT </ DHT-Overlay-name >
                <DHT-Hash-alg>SHA-1</ DHT-Hash-alg >
        </DHT-description>
        <DHT-operation > userRegistration </ DHT-operation>
        <DHT-Successor-list >
        !for clarity we show only 2 entries!>
                < DHT-Succesor-node1>sip:56@10.0.0.56</ DHT-Succesor-node1>
                < DHT-Succesor-node2> sip:8@10.0.0.8</ DHT-Succesor-node2>
        </DHT-Successor-list >
        <DHT-node> sip:51@10.0.0.51</DHT-node>
</DHT-info>
```

SIP message sent from node N21 to node N56.

Node 21 → Node 56

```
Register sip:10.0.0.56 SIP/2.0
TO: sip:Alice@concordia.ca
From: sip:Alice@concordia.ca
Contact: sip:Alice@10.0.0.21
Content-Type: application/DHT+XML
Content-Length: 142
Require: P2P-DHT
Supported: P2P-DHT
//empty line
<?xml version = “1.0”?>
<DHT-info>
    <DHT-description>
        <DHT-Overlay-name>Concordia DHT </ DHT-Overlay-name >
        <DHT-Hash-alg>SHA-1</ DHT-Hash-alg >
    </DHT-description>
    <DHT-operation > userRedundantRegistration </ DHT-operation>
    <DHT-Resource-List>
        <DHT-Resource>
            <DHT-Resource-ID>48</DHT-Resource-ID>
```

```
                    <DHT-Resource-    URI>sip:Alice@10.0.0.21    </DHT-
                Resource-URI>
                    <DHT-Resource-UserName>Alice@concordia.ca</DHT-
                Resource-UserName>
                <DHT-Resource-Expire>45</DHT-Resource-Expire>
            </DHT-Resource>
        </DHT-Resource-List>
        <DHT-node> sip:21@10.0.0.21</DHT-node>
</DHT-info>
```

SIP message sent from node N56 to node N21.

Node 56 → Node 21

```
SIP/2.0 200 OK
TO: sip:Alice@concordia.ca
From: sip:Alice@concordia.ca
Contact: sip:Alice@10.0.0.21
Content-Type: application/DHT+XML
Content-Length: 142
Require: P2P-DHT
Supported: P2P-DHT
//empty line
<?xml version = "1.0"?>
```

```
<DHT-info>
        <DHT-description>
                <DHT-Overlay-name>Concordia DHT </ DHT-Overlay-name >
                <DHT-Hash-alg>SHA-1</ DHT-Hash-alg >
        </DHT-description>
        <DHT-operation > userRedundantRegistration </ DHT-operation>
        <DHT-Successor-list >
        !for clarity we show only 2 entries!>
                < DHT-Succesor-node1>sip:8@10.0.0.8</ DHT-Succesor-node1>
                < DHT-Succesor-node2> sip:21@10.0.0.21</ DHT-Succesor-node2>
        </DHT-Successor-list >
        <DHT-node> sip:56@10.0.0.56</DHT-node>
</DHT-info>
```

In this chapter, we have proposed a new P2P architecture for the SIP telephony. The new architecture does not need any central server since the client end device can play the role of the client and of the server at the same time. Thus the SIP2P system has a low cost property and does not need a dedicated administrator for maintenance.

5 Implementation and evaluation of the SIP2P System

In order to validate the architecture of the SIP2P system, we have implemented the system and conducted performance tests on it.

5.1 Data structures and classes

Following the description of the proposed SIP2P system architecture (see Section 4), we have made an implementation and validated the SIP2P system. We used the library *open chord [1]* for the DHT layer and we chose the open source library *Jain SIP [4]* as a SIP stack for its performance and reliability.

SIP2P is currently implemented in Java and consists of roughly 120000 lines of code in 587 source files and has the following class diagram (see Figure 5.1).

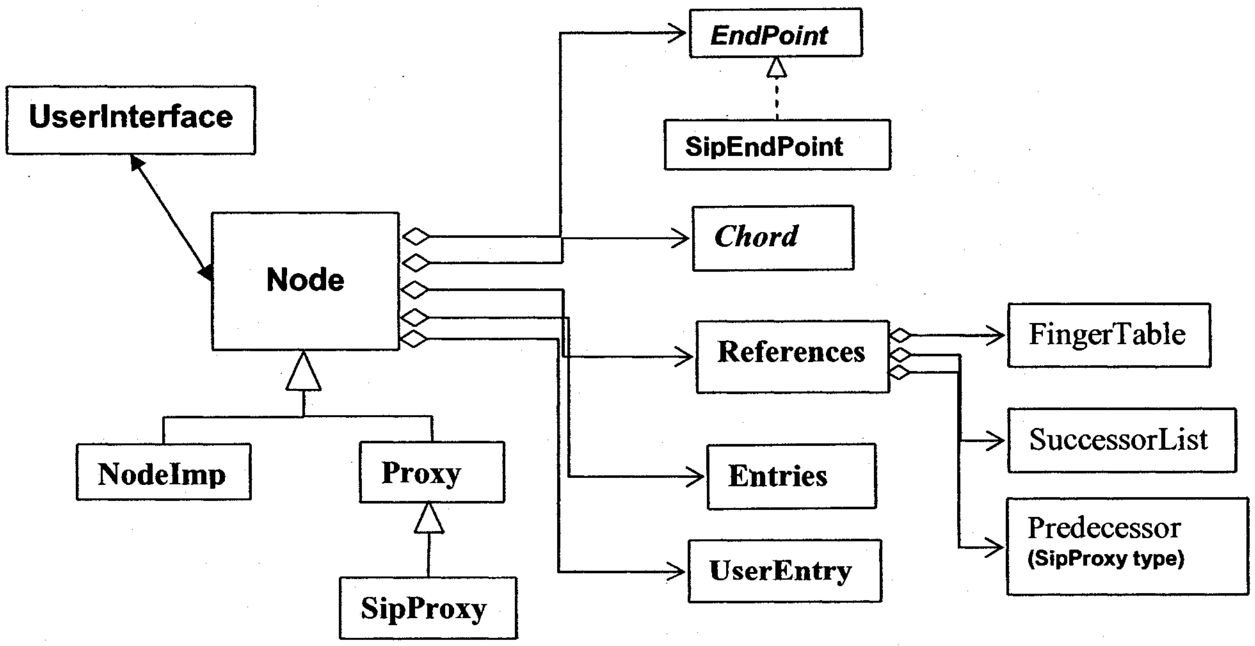


Figure 5.1: Class diagram in the SIP2P

Node

It represents a peer entity, containing all operations which can be invoked remotely by other nodes.

Chord

It represents the logic of the Chord system, including methods for changing connectivity to the network (create, join, leave), as well as those for manipulating the content (resource) (insert, retrieve, remove).

References

It includes the routing table (finger table), the successor list and the predecessor.

Entries

It represents the mapping between user ID and IP addresses stored in the nodes.

EndPoint and SipEndPoint

Represent the communication service, using the SIP stack.

They provide an interface to handle the incoming requests and responses.

UserEntry

It represents a user of the local node.

SipProxy

A SipProxy represents a reference to a remote nodes which a local node could connect directly to them.

All operations tacking place in a local Node (NodeImpl), are available for a SipProxy reference.

All elements (in finger table, successor list and predecessor) are of SipProxy type.

Local node can send a request to any other peer through the SipProxy that represents it.

Figure 5.2 shows the different type of requests that a local node could send.

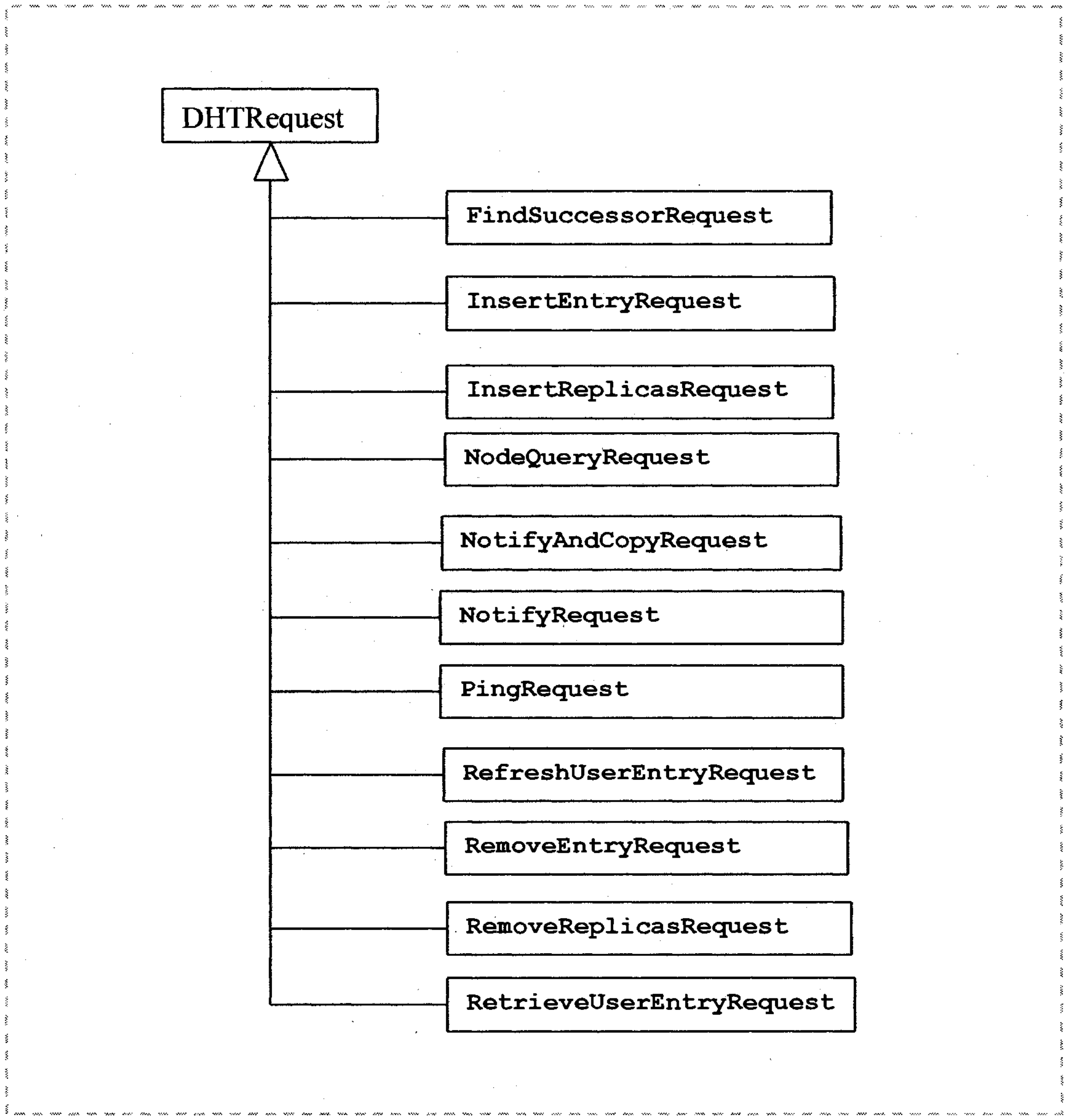


Figure 5.2: Different types of requests

5.2 Flow description

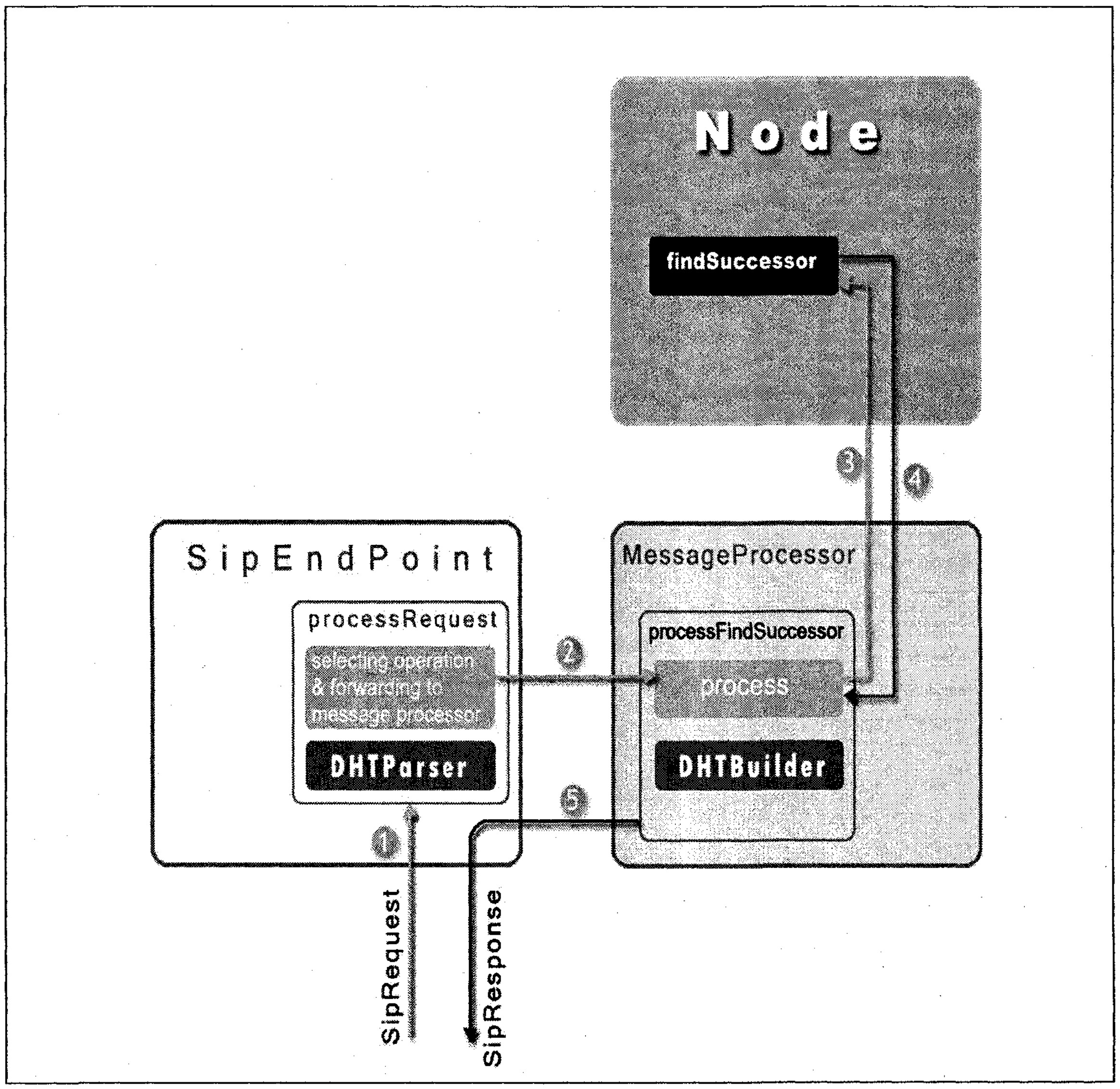


Figure 5.3: Flow description

As soon as a SIP request comes from a remote peer **(1)**, it is received by the local node through the *processRequest* method in the *Sip End Point* module, where it is first parsed by the *DHT Parser.*

After identifying the network operation that will be performed

(for example: `NetworkOperation.findSuccessor`) then the flow is forwarded to the *message processor* module where it will be handled by the *processFindSuccessor* method, i.e., **(2)** in the illustration provided by Figure 5.3.

Then the flow is transmitted to the *Node* module where it is processed by the appropriate operation handler (*findSuccessor* **(3)**). Once the *processFindSuccessor* method gets the result **(4)**, it builds the DHT body of the response with the *DHT Builder*, and finally sends the SIP response **(5)**.

5.3 Main features

➢ **Recovery mechanism**

Assume that each node has N successors.

There is a critical event where all the N successors leave the network or fails simultaneously. Thus the node is detached from the network and the ring is broken.

We solved this problem by replacing any failed successor in the successor list with a new node from the finger table. The fact that finger table contains references to nodes that are far (according to the logarithmic distribution) comparing to those in the successor list, helps any node to keep references that can be used as a recovery form such a critical situation.

Surely, right after the recovery, the successor list is not consistent but relying on the *stabilize task* of its neighbors, the node becomes consistent after a given latency.

- **Iterative lookup**

As we have already seen in Section 4.6.3.2, the successor lookup is an iterative process. However this feature could help us in the future to improve the ring security.

During the iterative lookup process, we can observe that the node ID becomes closer to the searched node ID at each step.

Detection of malicious nodes

Moreover, if there is any malicious node in the overlay, especially any intermediate nodes during the lookup process that would reply with a wrong node ID, i.e., sending a reference of next node having an ID far from the local node ID.

Therefore, as the local node keeps a history of the intermediate node during the lookup process, it would be able to detect this kind of malicious behavior, and identify the malicious node. It could also publish to other peers the presence of a malicious node, so it could be removed from all tables in the ring.

After the detection during the lookup process, the local node would recover by backtracking to the last previous safe intermediate node, and could ask for another path.

5.4 Evaluation

We conducted some performance measurements for the reliability and the scalability on the SIP2P system instead of using simulations. Since the implementation is based on Chord, more simulations would not add any research value to the existing simulation results as intensive simulations were already done in the context of Chord.

We begin with a short description of our experimental methodology. All experiments used a Java SIP2P implementation and Jain SIP stack running in SUN's JDK 1.5 with node virtualization.

Node Virtualization: To enable a larger number of nodes in our experiments, we placed multiple SIP2P node instances on each physical machine. To minimize computational overhead between nodes while maximizing the number of instances on each physical machine, we run each node instances inside its own Java Virtual Machine (JVM). This technique enables the execution of many simultaneous instances of SIP2P on a single machine.

A side effect of virtualization is the delay introduced by central processing unit (CPU) scheduling between nodes. During periods of high CPU load, scheduling delays can significantly impact performance results and artificially increase routing and location latency results. This is exacerbated by unrealistically low network distances between nodes on the same machine. These node instances can exchange messages in less than 10μs, making any overlay network processing overhead and scheduling delay much more expensive in comparison. These factors should be considered while interpreting results.

Our microbenchmarks are run on local cluster composed of 13 machines of duo core 1.8-GHz servers (2 GB RAM). We have run five nodes per machine (node virtualization) so we have a network of 65 nodes which is considered a reasonable size of a telephony network in a small enterprise. Each node in our SIP2P test network has a successor list of size 4 and it runs a test-member stage that listens to the network for commands sent by a central test driver.

Note that the results of experiments using node virtualization may be skewed by the processing delays associated with sharing CPUs across node instances on each machine.

The tests consist of analyzing SIP2P's scalability and stability under dynamic conditions.

- **Parallel node insertion**

We measure the effects of multiple nodes simultaneously entering the SIP2P network by examining the stabilization time for parallel insertions. Starting with a stable network of 20 nodes, we repeat each parallel insertion 5 times, and plot the median values versus the number of nodes being simultaneously inserted (see Figure 5.4 and Table 5.1).

Note that in Table 5.1, the first line in the first row represents the percentage of the number added nodes in the network and the second line represents the number of added nodes.

We note that the median time for stabilization scales roughly logarithmically with the number of simultaneously inserted nodes. Furthermore, we consider that we have achieved a good scalability since we can simultaneously double the initial size of the network.

Ratio of simultaneous inserted nodes	0.25 (5)	0.5 (10)	0.75 (15)	1 (20)	1.25 (25)	1.5 (30)	1.75 (35)	2 (40)	2.25 (45)
Stabilization time (ms)	9891	16000	27800	31728	33871	36745	43283	47477	49101

Table 5.1: Stabilization times of parallel node insertion

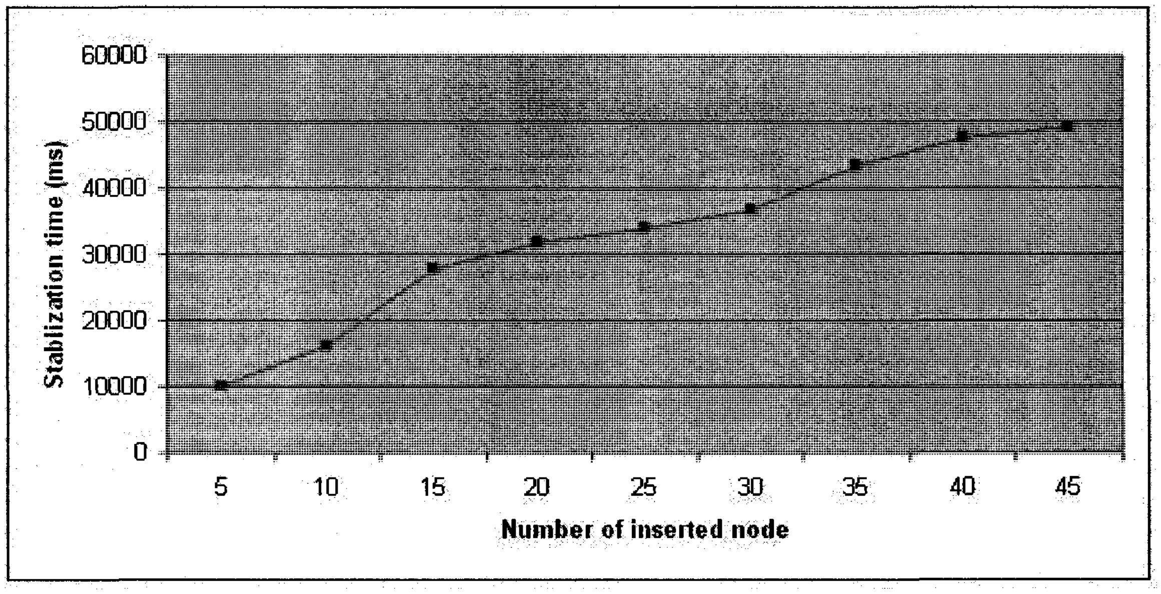


Figure 5.4: Stabilization times of parallel node insertion

➢ **Parallel node failure**

We measure the effects of multiple random nodes simultaneously failing in the SIP2P network by examining the stabilization time for parallel failure. Starting with a stable network of 65 nodes, we repeat each parallel failure 5 times, and plot the median values versus the number of nodes being simultaneously inserted (see Figure 5.5 and Table 5.2). Note that in Table 5.2, the first line in the first row represents the percentage of the number of added nodes in the network and the second line represents the number of added nodes.

We also note that the median time for stabilization scales roughly logarithmically with the number of simultaneously random failed nodes and we also achieve a good scalability since we can simultaneously disconnect more than the half of the network.

Ratio of simultaneous failed nodes	**0.07 (5)**	**0.15 (10)**	**0.23 (15)**	**0.30 (20)**	**0.38 (25)**	**0.46 (30)**	**0.53 (35)**	**0.61 (40)**	**0.69 (45)**
Stabilization time (ms)	**15978**	**40238**	**63000**	**70012**	**81673**	**93341**	**97483**	**101808**	**103325**

Table 5.2: Stabilization times of parallel random failed nodes

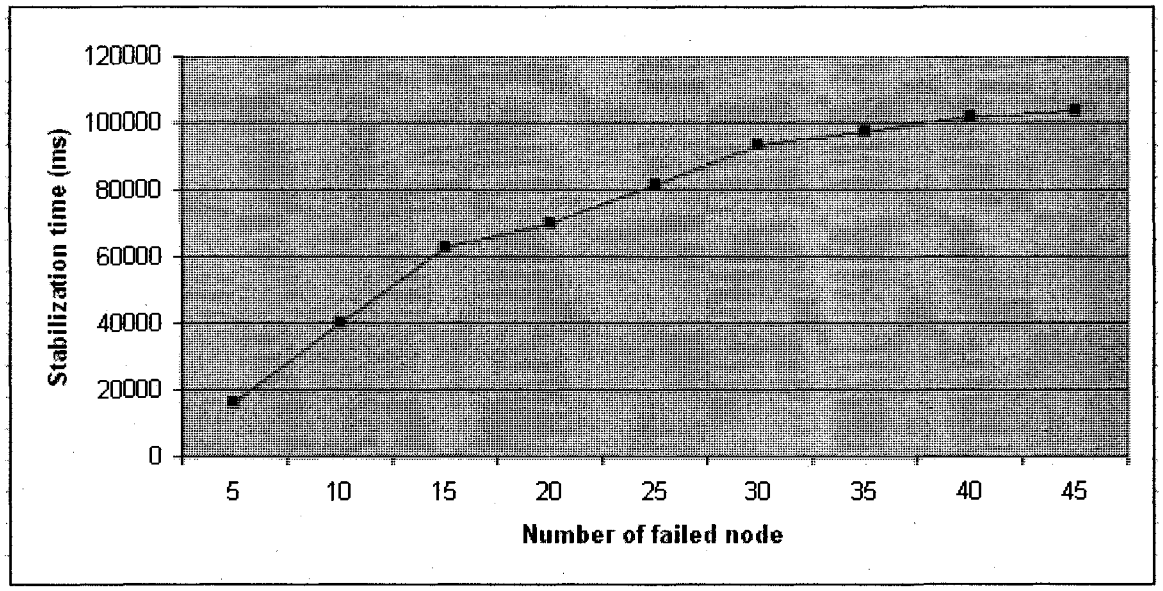


Figure 5.5: Stabilization times of parallel random failed nodes

➢ Parallel node insertion and failure

Finally we measure the effects of multiple nodes simultaneously entering and failing in the SIP2P network by examining the stabilization time for parallel node insertion and node failure. Starting with a stable network of 20 nodes, we insert and disconnect the same number of nodes each time. We also repeat each parallel insertion and failure 5 times, and plot the median values versus the number of nodes being simultaneously inserted and failed (see Figure 5.6 and Table 5.3).

Note that in Table 5.3, the first line in the first row represents the percentage of the number of added nodes in the network and the second line represents the number of added nodes.

We also note that the median time for stabilization scales roughly logarithmically with the number of simultaneously random failed nodes.

Ratio of simultaneous failed nodes	0.25 (5)	0.5 (10)	0.75 (15)	1 (20)	1.25 (25)
Stabilization time (ms)	11891	33998	42224	46291	49572

Table 5.3: Stabilization times of parallel node insertion and failure

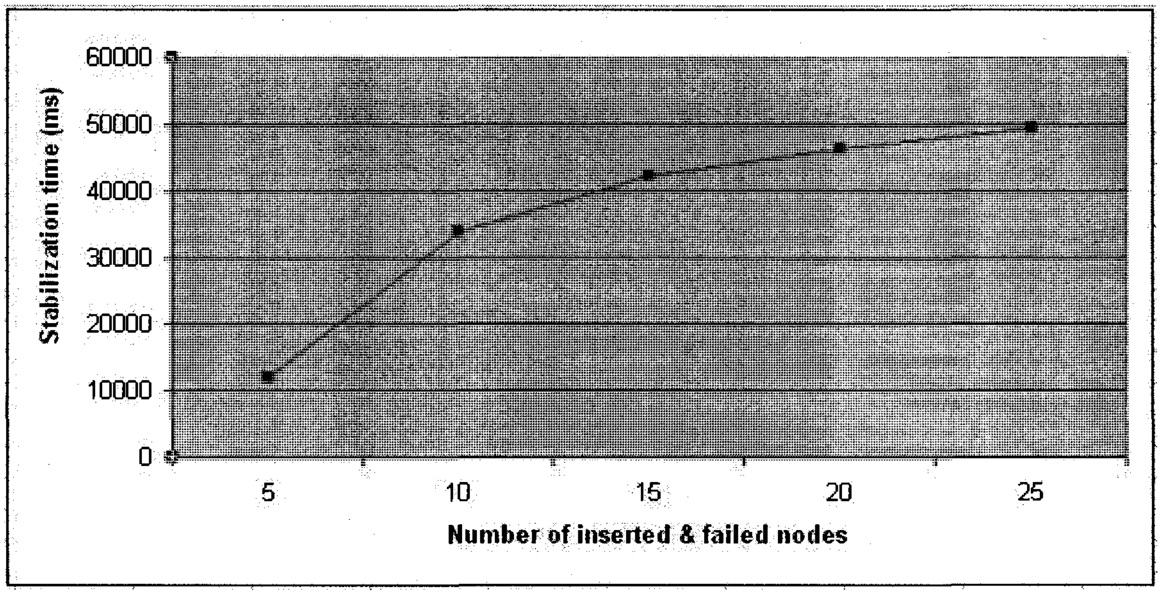


Figure 5.6: Stabilization times of parallel node insertion and failure

In this chapter, we have implemented and validated a prototype of the SIP2P system. As shown through experiments, the proposed architecture is scalable and reliable.

6 Conclusion and future directions

We have proposed a new P2P architecture for the SIP telephony. The new architecture does not need any central server since the client end device can play the role of the client and of the server at the same time. Thus the SIP2P system has a low cost property and does not need a dedicated administrator for maintenance. We have also implemented and validated a prototype of the SIP2P system. In addition, the proposed architecture is scalable and reliable as shown through experiments in Chapter 5.

Furthermore, the SIP2P architecture is flexible to integrate new structures. It is open to dynamic changes as required by heterogeneous P2P voice systems since the proposed architecture separate P2P properties from the underlying voice and transport protocols.

The P2P telephony can be used in a variety of business scenarios. Voice is one of the business applications where the P2P model can offer many benefits such as reducing deployment costs and increasing flexibility and recoverability. From a small business to small branches of a large enterprise, P2P voice can be used to rapidly deploy fully featured phone systems without the added cost of an IP PBX. Also, the company does not need anymore to replace its IP PBX with a bigger one each time its set of users grow. All is needed is to plug a new P2P phone for each new user.

In [8], we can find the answer of the question "Where the P2P telephony will be used?". According to the author, the P2P telephony solution gets much of the attention in the enterprise environment where the first real inroads are being made, due to the relative simplicity of building smaller systems and the perceived rapid revenue path for vendors. Systems like the Avaya one-X Quick Edition or the Siemens HiPath BizIP product can quickly recognize new phones as they are connected to the network, and they require minimal provisioning beyond entering phone extensions into their devices.

As a result, the deployments of these new phones are already beginning to make inroads in the market. Frost and Sullivan published a report in January 2007 indicating that P2P telephony had captured 0.3 percent of the European business telephone market, a number they expect to grow to nearly 4 percent in 2012. While that number seems modest, this technology is just emerging, and is only available from a few manufacturers to date.

Several open research problems need to be solved for a feature-rich SIP2P telephony system that would allow communication between nodes behind a NAT since a lot of users will be typically behind NAT and firewall. Nodes behind NAT will have a private address in the 192.168.0.x and 192.168.1.x range, thus there is a small probability for two users to use the same port number and to have the same ID. A solution of this problem could be the use of the STUN [32] or the TURN [35] mechanism. One commonly held belief is that the deployment of IPv6 [11] will eliminate the problem of NATs within the Internet. One argument is that there will be plentiful public IPv6 address space available and no particular reason to deploy NATs in an IPv6 realm. That does not say that IPv6 NATs will not be implemented, nor used. Indeed IPv6 NATs are already available.

Because P2P systems are inherently different from client-server systems, new challenges for security arise [39]. For instance, the lack of a central authority makes authentication in a pure P2P network difficult. Without authentication, adversary nodes can spoof identity and falsify messages in the overlay. This enables malicious nodes to launch man-in-the-middle or denial-of-service attacks. Castro *et al.* identify three requirements for secure P2P telephony network: secure node-ID assignment, secure routing table maintenance, and secure message forwarding [10].

Throughout the above discussion, we can see that the contribution of this thesis is a first step towards the deployment of the P2P telephony system.

7 References

[1] http://www.uni-bamberg.de/en/pi/bereich/research/software_projects/openchord/

[2] http://www.w3.org/MarkUp/SGML/

[3] http://www.w3.org/XML/

[4] https://jain-sip.dev.java.net/

[5] Baset, S., and H. Schulzrinne, "An analysis of the skype peer-to-peer internet telephony protocol," in Proceedings of the 25th IEEE InternationalConference on Computer Communications - INFOCOM, April 2006, pp.1–11.

[6] Baset, S., and H. Schulzrinne. An Analysis of the Skype P2P Internet Telephony Protocol. Sept 2004.

[7] Bryan, D., B. Lowekamp, and C. Jennings, "SoSimple: A serverless, standards-based, P2P SIP communication system," in First International Workshop on Advanced Architectures and Algorithms for Internet Delivery and Applications - AAA-IDEA, June 2005, pp. 42– 49.

[8] Bryan, D., http://www.sipeerior.com/docs/BCR_Apr_07_P2PSIP.pdf, white paper

[9] Bryan, D., and B. Lowekamp. SoSimple: a SIP/SIMPLE Based P2P VoIP and IM System. 2004. Tech. Rep.

[10] Castro, M., "Secure Routing for Structured Peer-to-Peer Overlay Networks," Proc. 5th Symp. Op. Sys. Design and Implementation, Boston, MA, Dec. 2002.

[11] Deering, S., R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification" RFC2460.

[12] Druschel, P., and A. Rowstron. PAST: A large-Scale, Persistent Peer-To-Peer Storage Utility. In Proc. HotOS VIII, Schloss Elmau, Germany, May 2001.

[13] Druschel, P., and A. Rowstron. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-To-Peer Systems. IN Proc. IFIP/ACM, Germany, Nov 2001.

[14] Freenet, www.freenetproject.org

[15] Gnutella: Peer-To-Peer File Sharing Software Application, http://www.gnutella.com.

[16] Greg Plaxton, C., R. Rajaraman, and W. Richa. Accessing Nearby Copies of Replicated Objects In a Distributed Environment. ACM, June 1997.

[17] Groove Workspace Software, http://www.groove.net.

[18] Karger, D. R., E. Lehman, F. Leighton, M. Levine, D. Lewin, and R.Panigrahy, "Consistent Hashing and Random Trees: Distributed caching protocols for relieving hot spots on theWorldWideWeb," in Proc. 29th Annu. ACM Symp. Theory of Computing, El Paso, TX, May 1997, pp. 654–663.

[19] Kazaa, www.kazaa.com

[20] Keong Lua, E., J. Crowcroft, M. Pias, R. Sharma and S. Lim. A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. IEEE Communication Survey, March 2004.

[21] Kubiatowicz, J., D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture For Global-Scale Persistent Storage. in Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS, vol. 35, November 2000, pp. 190 – 201.

[22] Lewin, D. "Consistent Hashing and Random Trees: Algorithms for caching in distributed networks," Master's thesis, Department of Electric. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, 1998.

[23] Li, J., J. Jannoti, D. Karger, and R. Morris, A Scalable Location Service for Geographic Ad hoc Routing. In Proceedings of the 6th ACM International Conference on Mobile Computing and Networking (Boston, Massachusetts, August 2000), pp. 120–130.

[24] Magi Peer-To-Peer Technology Being Adopted Across Vertical Industries," http://www.endeavors.com/PressReleases/partners1.htm.

[25] Milojicic, D., V. Kalogeraki, R. Lukose, K. Nagaraja1, J. Pruyne, B. Richard. Peer-to-Peer Computing. HP Laboratories Palo Alto. Tech. Rep. March 2002.

[26] Milojicic, D., 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.

[27] Mockapertris, P., Development of the Domain Name System. In Proc. ACM SIGCOMM (Stanford, CA, 1988), pp. 123–133.

[28] Mockapetris, P., and K.J. DUNLAP, Development of the Domain Name System. In Proc. ACM SIGCOMM (Stanford, CA, 1988), pp. 123–133.

[29] Morris, R., M.F. Kaashoek, D. Karger, H. Balakrishnan, I. Stoica, D. Liben-nowell, and E. Dabek, Chord: A scalable Peer-to-peer Lookup Protocol for Internet Applications, IEEE/ACM 2001.

[30] PEER-TO-PEER Working Group. 2001. Bidirectional Peer-to-Peer Communication with Interposing Firewalls and NATs. p2pwg White Paper, Revision 0.091. May 23, 2001. http://www.peer-to-peerwg.org/tech/nat/

[31] Ratnasamy, S., P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content Addressable Network," in SIGCOMM Symposium on Communications Architectures and Protocols, San Diego, CA, USA, Aug. 2001, ACM.

[32] RFC 3489, STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)

[33] Rhea, S., C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz, "Maintenance-free global data storage," IEEE Internet Computing, vol. 5, pp. 40–49, September/October 2001.

[34] Rosenberg, J., 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.

[35] Rosenberg, J., R. Mahy , C. Huitema " Traversal Using Relay NAT (TURN)", http://www.ietf.org/internet-drafts/draft-rosenberg-midcom-turn-07.txt

[36] Rowstron, A., and P. Druschel. Storage Management and Caching in PAST, a Large-Scale,Persistent Peer-To-Peer Storage Utility. In Proc. ACM SOSP'01, Banff, Canada, Oct. 2001.

[37] Schulzrinne, H., and J. Rosenberg, "Internet Telephony: Architecture and Protocols – an IETF perspective," Computer Networks and ISDN Systems, vol. 31, no. 3, pp. 237–255, Feb. 1999.

[38] "Secure Hash Standard," U.S. Dept. Commerce/NIST, National Technical Information Service, Springfield, VA, FIPS 180-1, Apr. 1995.

[39] Seedorf, J., Security Challenges for P2P-SIP, Special Issue on Securing Voice over IP, IEEE Network, vol. 20, no. 5, September 2006, pp. 38 – 45

[40] Singh, K., and H. Schulzrinne. Peer-To-Peer Internet Telephony Using SIP. In NOSSDAV, pages 63–68, 2005.

[41] Stoica, I., R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. C. Balakrishnan, "A Scalable Peer-To-Peer Lookup Service for Internet Applications," Lab. Comput. Sci., Massachusetts Inst. Technol., Tech. Rep. TR-819, 2001.

[42] Skype web site. http://www.Skype.com

[43] Toga, J., and J. Ott, "ITU-T Standardization Activities for Interactive Multimedia Communications on Packetbased Networks: H.323 and Related Recommendations," Computer Networks and ISDN Systems, vol. 31, no. 3, pp. 205–223, Feb. 1999.

[44] Wolfer, C., "Peer-to-peer voip: Will it ever work?" New Telephony Magazine, pp. 6–8, 2005.

[45] Zhao, B. Y., J. D. Kubiatowicz, and A. D. Joseph, "Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing," Univ. California, Berkeley, CA, Tech. Rep. CSD-01-1141, Apr. 2001.