UNIVERSITÀ DEGLI STUDI DI PARMA

Dottorato di Ricerca in Tecnologie dell'Informazione

XXIII Ciclo

A DHT-based Peer-to-peer Architecture for Distributed Internet Applications

Coordinatore:

Chiar.mo Prof. Carlo Morandi

Tutor:

Chiar.mo Prof. Luca Veltri

Dottorando: *Simone Cirani*

Gennaio 2011

Alla mia famiglia e a Paola
per l'amore e la fiducia
che mi donate ogni giorno...

Table of Contents

Introduction **1**

1 Peer-to-peer Networks **3**

1.1 Unstructured and structured P2P networks 4

1.2 Distributed Hash Tables 6

1.2.1 Chord 9

1.2.2 Kademlia 16

1.2.3 Other DHTs 23

1.2.4 One-hop DHTs 23

1.3 Bootstrapping 24

2 A Multicast-based approach to Peer-to-peer Bootstrapping **27**

2.1 Introduction 27

2.2 Solicited vs. Unsolicited Approach 30

2.2.1 Solicited (PULL) approach 30

2.2.2 Unsolicited (PUSH) approach 30

2.3 Simple Algorithm 31

2.3.1 Synchronized case 31

2.3.2 Unsynchronized case 33

2.3.3 Problems with the simple algorithm 33

2.4 Enhanced Algorithm 36

2.4.1 Estimation of the number of collaborating nodes 36

2.4.2 Scheduling 37
2.4.3 Algorithm description 43

3 Distributed Location Service 45
3.1 Motivations for a Distributed Location Service 45
3.1.1 URI information 47
3.1.2 The IETF P2PSIP Working Group 49
3.1.3 Distributed SIP Location Service 49
3.2 DLS Architecture 50
3.2.1 Information stored into the DHT 52
3.3 DLS Layers 53
3.3.1 DLS Layer 54
3.3.2 Peer Layer 54
3.3.3 RPC Protocol Layer 55
3.4 DHT-unaware clients and peer adapters 56
3.5 IETF P2PSIP WG Proposals 58
3.5.1 dSIP 58
3.5.2 RELOAD 59

4 DLS Framework Implementation 65
4.1 RPC Layer 66
4.2 Peer Layer 69
4.3 DLS Layer 72
4.4 Protocol Adapters 73
4.5 DLS Framework usage 77
4.6 DLS Framework extension 81
4.6.1 RPC Protocols 81
4.6.2 DHT Algorithms 82
4.6.3 DLS Client Protocols 83

5 DLS-based Peer-to-peer Applications 85
5.1 P2P VoIP 85

5.1.1 P2P VoIP Architecture 86
5.2 Distributed Web Server 88
5.2.1 DWS Node Architecture 89
5.2.2 DWS Deployment Example 91
5.3 Distributed File System with HDFS 93
5.3.1 HDFS Architecture 94
5.3.2 DLS-based HDFS 95
5.3.3 Possible Enhancements 96
5.4 Peer-to-peer Online Social Networks 98
5.4.1 P2P OSN Architecture 100
5.4.2 Data Storage 100
5.4.3 Distributed Storage Service 101
5.4.4 Privacy Enforcement Framework 102
5.4.5 Social Network Services 107

6 Conclusions **109**

Bibliography **111**

Acknowledgments **115**

List of Figures

1.1 Overlay network 5
1.2 Chord simple node lookup 10
1.3 Chord scalable node lookup 13
1.4 Probability of remaining online another hour as a function of uptime 17
1.5 Kademlia k-buckets management policy 20

2.1 Procedure to get the value of $\Gamma_{n,2}$ 40
2.2 Procedure to get the value of $\Gamma_{n,n}$ 40
2.3 Procedure to get the value of $\Gamma_{7,4}$ 41
2.4 Ratio of non empty slots using mathematical model 42
2.5 Ratio of non empty slots using simulations 42
2.6 Improving hit-ratio and collision-ratio by refining the choice after detecting a collision 44

3.1 Applications accessing the DLS 52
3.2 Location Service Layer 53
3.3 DLS Peer layered architecture 54
3.4 DLS Client layered architecture 57
3.5 DLS Client (left) with DLS Adapter Peer (right) 57
3.6 RELOAD architecture 61

4.1 The it.unipr.profiles.communicator package (RPC Layer) 68
4.2 The it.unipr.profiles.dSIP package 69

4.3 The it.unipr.profiles.peer package (Peer Layer) 71
4.4 Peer creation through PeerFactory 71
4.5 Information storage 72
4.6 The it.unipr.profiles.dls package (DLS Layer) 74
4.7 The it.unipr.profiles.adapter package 76
4.8 Extending the DLS framework to support other DHT algorithms . . 84

5.1 SIP trapezoid 86
5.2 P2P VoIP session with P2P-aware SIP UAs 87
5.3 P2P VoIP session with P2P-unaware SIP UAs 88
5.4 Distributed Web Server scenario 90
5.5 HDFS and DLS integration 97
5.6 Key distribution tree 105
5.7 Storing encrypted content in the DSS/DLS 106

List of Tables

3.1 Abstract DLS table representation 51

5.1 DLS content for the DWS for the domain *www.dws.org* 92
5.2 Evaluation of different key distribution approaches 103
5.3 Evaluation of different approaches for key addition 104
5.4 Evaluation of different approaches for key revocation 104

Listings

4.1 Creating a DLS-based application 78

Introduction

Peer-to-peer technology has become popular primarily due to file sharing applications, such as Napster, Gnutella, Kazaa, and eMule, which have been the dominant component of usage of Internet bandwidth for several years. However, peer-to-peer technology is not all about file sharing. Many famous applications used by millions of users every day, such as Skype, are applications based on the peer-to-peer paradigm.

The peer-to-peer (P2P) paradigm is a communication model in which multiple independent and heterogeneous devices interact as equals (peers). In a pure P2P network each node implements functions of both client and server, and either peer can initiate a communication session at any moment. Nodes are arranged on an overlay network, built on top of an existing network, such as the Internet. Many peer-to-peer applications are based on a particular class of peer-to-peer networks: Distributed Hash Tables (DHT). DHTs are structured peer-to-peer networks which provide a service of information storage and retrieval similar to a regular hash table where keys are mapped to values, in a scalable, flexible, and self-organizing fashion.

This thesis reports the results of the research activity on applying peer-to-peer technology beyond file sharing. The work has been focused first on the study and analysis of existing peer-to-peer network implementations, especially on Distributed Hash Tables, and the proposals for peer-to-peer protocols presented by the IETF P2PSIP Working Group. The main research activity has been the definition of a peer-to-peer architecture, called Distributed Location Service (DLS), which allows the establishment of direct connections among the endpoints of a communication without the need of central servers. The Distributed Location Service is a DHT-based peer-to-peer service which can be used to store and retrieve information about where resources can be accessed, thus eliminating the need to rely (partially) on the DNS system and on central location servers, such as SIP Location Services. Access information is stored in the DLS as key-to-value mappings, which are maintained by a number of nodes that participate in the DHT overlay the DLS is built upon. The DLS has been implemented as a framework, by defining a standard set of interfaces between the components of the DLS, in order to allow maximum flexibility on components such as the DHT algorithm and communication protocol in use, as no assumption has been made in the definition of the DLS architecture. The Kademlia DHT algorithm and the dSIP communication protocol have been implemented and integrated in the DLS framework in order to create real-world DLS-based application to show the feasibility of the DLS approach. These demonstrative DLS-based applications have been realized with the intent to show that peer-to-peer is not just about file sharing, but real-time communication applications, such as VoIP, distributed file systems, and Online Social Networks, can also be built on top of a peer-to-peer architecture.

Even though the research activity has been conducted independently from the IETF P2PSIP Working Group, the Distributed Location Service has been eventually found quite similar to the official proposal, named RELOAD, with whom it shares several concepts and ideas.

Another aspect that was studied is the issue of bootstrapping in peer-to-peer networks. When a node wants to join an existing P2P network, it needs to gather information about one node that already belongs to the P2P overlay network which will then admit the new node. Typically, the discovery of a node that is already participating in the overlay is made through mechanisms such as caching, pre-configured list of nodes, or the use of central servers. Even though these approaches have worked so far, they are not in the true philosophy of peer-to-peer networks, where decentralization, scalability, and self-organization are critical features. A Multicast-based approach has therefore been defined and validated, with the goal of achieving true scalability and self-organization.

Chapter 1

Peer-to-peer Networks

A distributed system is a collection of independent computers that appears to its users as a single coherent system. Relevant features of distributed systems should be:

- Use of heterogeneous computers
- Transparent communication
- Easy to expand and scale
- Permanently available (even though parts of it are not)

A peer-to-peer (or P2P) computer network relies primarily on the computing power and bandwidth of the participants in the network rather than concentrating it in a relatively low number of servers. P2P networks are typically used for connecting nodes via largely ad hoc connections. Such networks are useful for many purposes. Sharing content files containing audio, video, data or anything in digital format is very common, and real-time data, such as telephony traffic, is also passed using P2P technology. A pure peer-to-peer network does not have the notion of clients or servers, but only equal peer nodes that simultaneously function as both clients and servers to the other nodes on the network. This model of network arrangement differs from the client-server model where communication is usually to and from a central server.

An important goal in peer-to-peer networks is that all clients provide resources, including bandwidth, storage space, and computing power. Thus, as nodes arrive and demand on, the system increases and the total capacity of the system also increases. This is not true in client-server architectures with a fixed set of servers, in which adding more clients could mean slower data transfer for all users. The distributed nature of peer-to-peer networks also increases robustness in case of failures by replicating data over multiple peers, and, in pure P2P systems, by enabling peers to find the data without relying on a centralized index server. In the latter case, there is no single point of failure in the system. A P2P network usually forms an **overlay network**, that is, a computer network which is built on top of another network. Nodes in the overlay can be thought of as being connected by virtual or logical links, each of which corresponds to a path, perhaps through many physical links, in the underlying network. For instance, many peer-to-peer networks are overlay networks because they run on top of the Internet. Peer-to-peer networks may be categorized into the following categories:

- Centralized P2P network such as Napster
- Decentralized P2P network such as Kazaa
- Structured P2P network such as CAN
- Unstructured P2P network such as Gnutella
- Hybrid P2P network (Centralized and Decentralized) such as JXTA

1.1 Unstructured and structured P2P networks

The P2P overlay network consists of all the participating peers as network nodes. There are links between any two nodes that know each other: if a participating peer knows the location of another peer in the P2P network, then there is a directed edge from the former node to the latter in the overlay network.

Based on how the nodes in the overlay network are linked to each other, we can classify P2P networks as unstructured or structured. An unstructured P2P network is

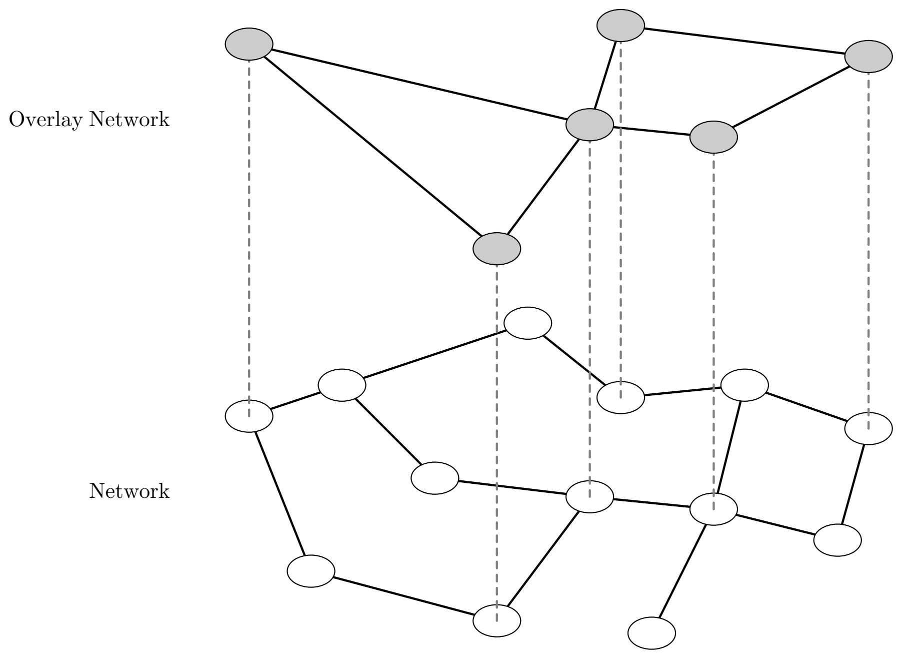


Figure 1.1: Overlay network

formed when the overlay links are established arbitrarily. Such networks can be easily constructed as a new peer that wants to join the network can copy existing links of another node and then form its own links over time. In an unstructured P2P network, if a peer wants to find a desired piece of data in the network, the query has to be flooded through the network to find as many peers as possible that share the data. The main disadvantage with such networks is that the queries may not always be resolved. Popular content is likely to be available at several peers and any peer searching for it is likely to find the same thing, but if a peer is looking for rare data shared by only a few other peers, then it is highly unlikely that search will be successful. Since there is no correlation between a peer and the content managed by it, there is no guarantee that flooding will find a peer that has the desired data. Flooding also causes a high amount of signaling traffic in the network and hence such networks typically have very poor search efficiency. Most of the popular P2P networks such as Gnutella and

FastTrack are unstructured. Structured P2P networks employ a globally consistent protocol to ensure that any node can efficiently route a search to some peer that has the desired resource, even if the resource is extremely rare. Such a guarantee necessitates a more structured pattern of overlay links. By far the most common type of structured P2P network is the Distributed Hash Table (DHT), in which a variant of *consistent hashing*[1] is used to assign ownership of each file to a particular peer, in a way analogous to a traditional hash table's assignment of each key to a particular array slot. Some well known DHTs are Chord, Kademlia, Pastry, Tapestry, and CAN.

1.2 Distributed Hash Tables

Distributed Hash Tables (DHTs) are a class of decentralized distributed systems that provide a lookup service similar to a hash table: $<key, value>$ pairs are stored in the DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from names to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows DHTs to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures. DHT research was originally motivated, in part, by peer-to-peer systems such as Napster, Gnutella, and Freenet, which took advantage of resources distributed across the Internet to provide a single useful application. In particular, they took advantage of increased bandwidth and hard disk capacity to provide a file sharing service. These systems differed in how they found the data their peers contained. Napster had a central index server: each node, upon joining, would send a list of locally held files to the

[1]Consistent hashing is a scheme that provides hash table functionality in a way that the addition or removal of one slot does not significantly change the mapping of keys to slots. In contrast, in most traditional hash tables, a change in the number of array slots causes nearly all keys to be remapped. By using consistent hashing, only K/n keys need to be remapped on average, where K is the number of keys, and n is the number of slots. Consistent hashing was introduced in 1997 as a way of distributing requests among a changing population of web servers. Each slot is then represented by a node in a distributed system. The addition (joins) and removal (leaves/failures) of nodes only requires K/n items to be re-shuffled when the number of slots/nodes change.

server, which would perform searches and refer the requester to the nodes that held the results. This central component left the system vulnerable to attacks and lawsuits. Gnutella and similar networks moved to a flooding query model; in essence, each search would result in a message being broadcast to every other machine in the network. While avoiding a single point of failure, this method was significantly less efficient than Napster. Finally, Freenet was also fully distributed, but employed a heuristic key based routing in which each file was associated with a key, and files with similar keys tended to cluster on a similar set of nodes. Queries were likely to be routed through the network to such a cluster without needing to visit many peers. However, Freenet did not guarantee that data would be found. Distributed Hash Tables use a more structured key based routing in order to attain both the decentralization of Gnutella and Freenet, and the efficiency and guaranteed results of Napster. One drawback is that, like Freenet, DHTs only directly support exact-match search, rather than keyword search, although that functionality can be layered on top of a DHT. DHTs characteristically emphasize the following properties:

- Decentralization: the nodes collectively form the system without any central coordination.
- Scalability: the system should function efficiently even with thousands or millions of nodes.
- Fault tolerance: the system should be reliable (in some sense) even with nodes continuously joining, leaving, and failing.

A key technique used to achieve these goals is that any one node needs to coordinate with only a few other nodes in the system - most commonly, $O(\log n)$ of the n participants - so that only a limited amount of work needs to be done for each change in membership. The structure of a DHT can be decomposed into several main components. The foundation is an abstract key space, such as the set of 160-bit strings. A key space partitioning scheme splits ownership of this key space among the participating nodes. An overlay network then connects the nodes, allowing them to find the owner of any given key in the key space. Most DHTs use some variant of consistent hashing to map keys to nodes. This technique employs a function $\delta(k_1,k_2)$ which defines an abstract notion of the distance from key k_1 to key k_2. Each node is assigned a single key called its identifier (ID). A node with ID i owns all the keys for which i is the closest ID, measured according to δ. Consistent hashing has the essential property that removal or addition of one node changes only the set of keys owned by the nodes with adjacent IDs, and leaves all other nodes unaffected. Contrast this with a traditional hash table in which addition or removal of one bucket causes nearly the entire key space to be remapped. Since any change in ownership typically corresponds to bandwidth-intensive movement of objects stored in the DHT from one node to another, minimizing such reorganization is required to efficiently support high rates of *churn* (node arrival and failure). Each node maintains a set of links to other nodes (its neighbors or routing table). Together these links form the overlay network. A node picks its neighbors according to a certain structure, called the network's topology. All DHT topologies share some variant of the most essential property: for any key k, the node either owns k or has a link to a node that is closer to k in terms of the key space distance defined above. It is then easy to route a message to the owner of any key k using the following greedy algorithm: at each step, forward the message to the neighbor whose ID is closest to k. When there is no such neighbor, then we must have arrived at the closest node, which is the owner of k as defined above. This style of routing is sometimes called **key based routing**. Beyond basic routing correctness, two key constraints on the topology are to guarantee that the maximum number of hops in any route (route length) is low, so that requests complete quickly; and that the maximum number of neighbors of any node (maximum node degree) is low, so that maintenance overhead is not excessive. Of course, having shorter routes requires higher maximum degree. Some common choices for maximum degree and route length are as follows, where n is the number of nodes in the DHT, using Big O notation:

- Degree $O(1)$, route length $O(\log n)$
- Degree $O(\log n)$, route length $O(\frac{\log n}{\log\log n})$
- Degree $O(\log n)$, route length $O(\log n)$

- Degree $O(n^{\frac{1}{2}})$, route length $O(1)$

The third choice is the most common, even though it is not quite optimal in terms of degree/route length tradeoff, because such topologies typically allow more flexibility in choice of neighbors. Many DHTs use that flexibility to pick neighbors which are close in terms of latency in the physical underlying network.

1.2.1 Chord

Chord [1] is a distributed lookup protocol that arranges nodes and keys on a circle. Nodes and keys are assigned a unique m-bit identifier, which is calculated by using consistent hashing. The SHA-1 algorithm is used as a base hashing function for the consistent hashing. Typically, a node ID is calculated by hashing the node's IP address and a resource's key is calculated by hashing some keyword that is related to the resource. IDs and keys map in the same key space. Consistent hashing assigns keys to nodes as follows. Identifiers are ordered on an identifier circle modulo 2^m. Key k is assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier space. This node is called the successor node of key k, denoted by `successor(k)`. If identifiers are represented as a circle of numbers from 0 to $2^m - 1$, then `successor(k)` is the first node clockwise from k. The identifier circle is also referred to as the Chord ring. The distance between a node whose ID is a and another node whose ID is b is computed as $(b-a)$ mod 2^m, and is therefore asymmetric.

Simple node lookup

A lookup operation returns the node that is responsible for a certain identifier, that is passed as an argument. In Chord, the responsible node is the successor node of the given id. Lookups could be implemented on a Chord ring with little per-node state. Each node needs only to know how to contact its current successor node on the identifier circle. Queries for a given identifier could be passed around the circle via these successor pointers until they encounter a pair of nodes that straddle the desired identifier; the second in the pair is the node the query maps to (Figure 1.2).

Algorithm 1 *find_successor()*: Simple node lookup

```
n:find_successor(id)
if $id \in (n, successor]$ then
    return $successor$;
else /* forward the query around the circle */
    return $successor : find\_successor(id)$;
end if
```

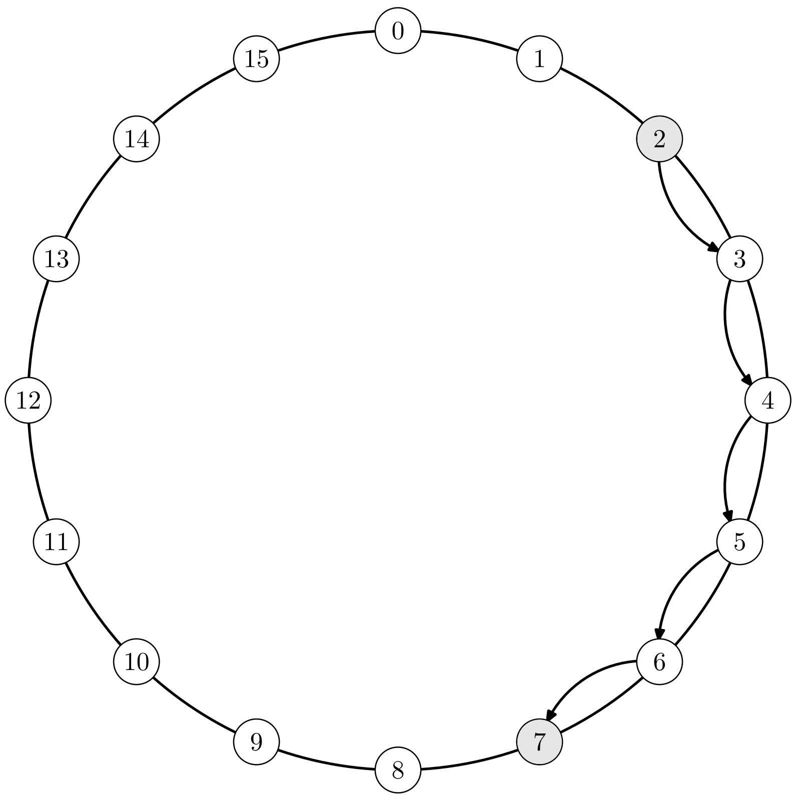


Figure 1.2: Chord simple node lookup

Scalable node lookup

In order to accelerate lookups, Chord maintains additional routing information. This additional information is not essential for correctness, which is achieved as long as each node knows its correct successor. Each node n maintains a routing table with up to m entries, called the **finger table**. The i^{th} entry in the table at node n contains the identity of the first node s that succeeds n by at least 2^{i-1} on the identifier circle, i.e., `s = successor(`$n+2^{i-1}$`)`, where $1 \leq i \leq m$ (and all arithmetic is modulo 2^m). We call node `s` the i^{th} finger of node n, and denote it by `n.finger[i]`. A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node. We can then report the following definitions:

- `successor`: the next node on the identifier circle
- `predecessor`: the previous node on the identifier circle
- `finger[k]`: first node on circle that succeeds $n+2^k \bmod 2^m$, $0 \leq k \leq m-1$

The lookup procedure for a given key is modified as follows:

- a node receives a query for the key
- if the node's successor is responsible for the given key, it returns the successor node
- else it returns the highest entry in the finger table that does precedes the key

This modified version of the lookup algorithm guarantees that lookups take at most $O(\log m)$ hops in order to succeed, as it halves the topological distance to the target (Figure 1.3). It is very important to ensure the correctness of the lookup procedure. The system is stable *if and only if each node knows its correct successor*.

Join

A node joins a Chord network by contacting the node whose ID immediately follows its own ID (called the *admitting node*). The discovery of the admitting node is

Algorithm 2 *find_successor()*: Scalable node lookup

```
n:find_successor(id)
if $id \in (n, successor]$ then
    return $successor$;
else /* forward the query around the circle */
    $\bar{n} = n : closest\_preceding\_node(id)$;
    return $\bar{n} : find\_successor(id)$;
end if
```

Algorithm 3 *closest_preceding_node()*: search the local table for the highest predecessor of *id*

```
n:closest_preceding_node(id)
for $i = m$ downto 1 do
    if $n : finger[i] \in (n, id)$ then
        return $n : finger[i]$;
    end if
end for
```

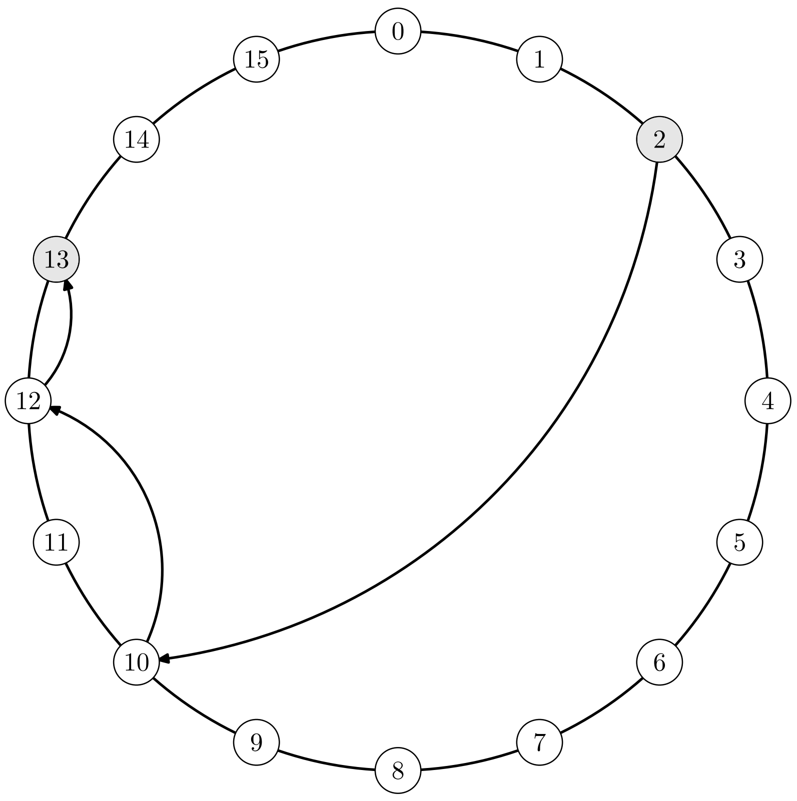


Figure 1.3: Chord scalable node lookup

performed by a executing a `find_successor()` with the joining node ID as an argument. Once the joining node has found its successor, it sets its successor node to be the admitting peer and sends to it a `notify()` procedure that informs the admitting peer that a new predecessor has joined the network. If the joining peer's ID falls in the interval between the admitting peer and its predecessor, the admitting peer sets its predecessor to be the joining peer.

Algorithm 4 *create()*: create a new Chord ring

```
n:create()
$n : predecessor = \textbf{null}$;
$n : successor = \textbf{n}$;
```

Algorithm 5 *join()*: node *n* joins the Chord ring by contacting node *x*

```
n:join(x)
$n : predecessor = \textbf{null}$;
$n : successor = x : find\_successor(n)$;
```

Algorithm 6 *notify()*: *n* believes it might be *x*'s predecessor

```
n:notify(x)
if $x : predecessor = \textbf{null}$ or $n \in (x : predecessor, x)$ then
    $x : predecessor = n$;
end if
```

Stabilization

Each node calls a `stabilize()` procedure to check the correctness of the successor node. The node contacts its successor, which responds with its predecessor. If the node and the successor's predecessor coincide, then the system is stable. Otherwise, the node needs to change its successor to be the successor's predecessor (as a new node between the node and the successor may have joined the network). Then the node would send a `notify()` to the new successor. Moreover, each node performs a `fix_fingers()` procedure to ensure its fingers point to the right nodes and a `check_predecessor()` procedure to check if the node's predecessor is still active or it has failed.

Algorithm 7 *stabilize()*: called periodically; verifies *n*'s immediate successor, and tells the successor about *n*

```
n:stabilize()
$x = n : successor : predecessor$;
if $x \in (n, n : successor)$ then
    $n : successor = x$;
    $successor : notify(n)$;
end if
```

Algorithm 8 *fix_fingers()*: called periodically; refreshes finger table entries

```
n:fix_fingers()
$next = 1$;
while $next \leq m$ do
    $n : finger[next] = n : find\_successor(n + 2^{next-1})$;
    $next{+}{+}$;
end while
```

Algorithm 9 *check_predecessor()*: called periodically; checks whether predecessor has failed

```
n:check_predecessor()
if $n : predecessor$ has failed then
    $n : predecessor =$ null;
end if
```

Node failure

The correctness of the Chord protocol relies on the fact that each node knows its successor. However, this invariant can be compromised if nodes fail. To increase robustness, each Chord node maintains a successor list of size r, containing the node's first r successors. If a node's immediate successor does not respond, the node can substitute the second entry in its successor list. All r successors would have to simultaneously fail in order to disrupt the Chord ring, an event that can be made very unlikely with modest values of r. Assuming each node fails independently with probability p, the probability that all r successors fail simultaneously is only p^r. Increasing r makes the system more robust. Handling the successor list requires minor changes in the pseudo-code: node n reconciles its list with its successor s by copying s's successor list, removing its last entry, and prepending s to it. If node n notices that its successor has failed, it replaces it with the first live entry in its successor list and reconciles its successor list with its new successor. At that point, n can direct ordinary lookups for keys for which the failed node was the success to the new successor. As time passes, `fix_fingers()` and `stabilize()` procedures will correct finger table entries and successor list entries pointing to the failed node.

Graceful leave

Since Chord is robust in the face of failures, a node voluntarily leaving the system could be treated as a node failure. However, two enhancements can improve Chord performance when nodes leave voluntarily. First, a node n that is about to leave may transfer its keys to its successor before it departs. Second, n may notify its predecessor p and successor s before leaving. In turn, node p will remove n from its successor list, and add the last node in n's successor list to its own list. Similarly, node s will replace its predecessor with n's predecessor. Here we assume that n sends its predecessor to s, and the last node in its successor list to p. After n's predecessor has changed its successor to n's successor, it sends a `notify()` to its new successor.

1.2.2 Kademlia

Foreword

An aspect that makes Chord hard to manage is the rigidity of the routing table which complicates recovery from failed nodes and routing table and precludes proximitybased routing. Moreover, in- and out- distribution are exactly opposite: this prevents from using incoming traffic to reinforce routing tables. Fixing these aspects has drawbacks: a bi-directional routing table would be more efficient and flexible but it would double the routing table size and the number of control messages in order to maintain the overlay. A main fact in the Chord algorithm is not taken into consideration: in an ideal case, once a node joins, it never leaves the network. In a realistic case, though, a randomly selected online node will stay online for another one hour with probability $\frac{1}{2}$. A statistical analysis of nodes' behavior in a Gnutella network has shown that the longer a node has been up, the more likely it is to remain up another hour. Figure 1.4 shows the probability of remaining online another hour as a function of uptime. The x-axis represents minutes. The y-axis shows the the fraction of nodes that stayed online at least x minutes that also stayed online at least $x+60$ minutes.

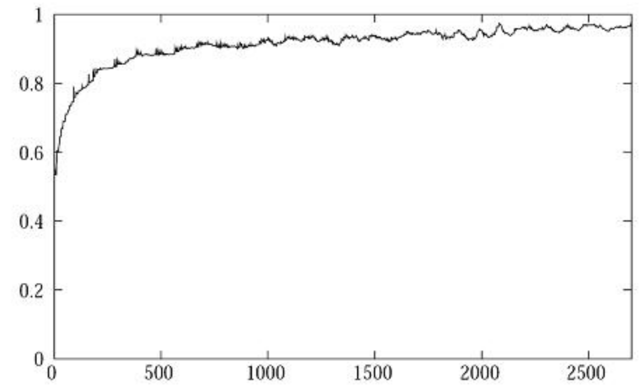


Figure 1.4: Probability of remaining online another hour as a function of uptime

Overview

Kademlia [2] is a DHT for decentralized peer-to-peer networks based on a XOR metric to compute the distance between two nodes on the network that exploits the fact the that long-time active nodes are most likely to stay active. Each node is assigned a unique 160-bit identifier, computed exactly as in Chord through the SHA-1 hash function. Keys are too 160-bit identifiers. The XOR metric defines distance between two nodes x and y as $d(x,y) = x \oplus y$. The topology has the property that every message exchanged conveys or reinforces useful contact information. In- and out- distributions are the same so the network reinforces itself. Kademlia thus minimizes the number of configuration messages that nodes must send to learn about each other. Configuration information spreads automatically as a side-effect of key lookups.

XOR metric properties

The XOR metric used by Kademlia offers some interesting and desirable properties that make it particularly efficient. First of all, it is obvious that $d(x,x) = 0$ and if $x \neq y, d(x,y) > 0$. Distance is, unlike Chord, symmetric, that is: $d(x,y) = d(y,x)$. Distance offers the triangular property: $d(x,y) + d(y,z) \geq d(x,z)$. Finally, distance is unidirectional, that is, given a point x and a distance $\Delta > 0$, there is only one point y such that $d(x,y) = \Delta$. Unidirectionality ensures that all lookups for the same key converge along the same path, regardless of the originating node.

System details

For each $0 \leq i < 160$, every node keeps a list of <IP address; UDP port; Node ID> triples for nodes of distance between 2^i and 2^{i+1} from itself. These lists are called *k-buckets*. Since XOR is used as a distance function, a node's i^{th} k-bucket contains nodes whose identifiers have the same most significant $159 - i$ bits. Since it is not relevant to calculate the exact distance between two nodes, but is sufficient to understand which k-bucket a node should belong to, the XOR metric is very efficient as the distance calculation is actually simplified to counting the number of equal most significant bits. Each k-bucket is kept sorted by time: least-recently seen node at the head, most-recently seen at the tail. For small values of i, the k-buckets will generally be empty (as no appropriate nodes will exist). For large values of i, the lists can grow up to size k, where k is a system-wide replication parameter. k is chosen such that any given k nodes are very unlikely to fail within an hour of each other (for example $k = 20$). When a Kademlia node receives any message (request or reply) from another node, it updates the appropriate k-bucket for the sender's node ID. If the sending node already exists in the recipient's k-bucket, the recipient moves it to the tail of the list. If the node is not already in the appropriate k-bucket and the bucket has fewer than k entries, then the recipient just inserts the new sender at the tail of the list. If the appropriate k-bucket is full, however, then the recipient pings the k-bucket's least-recently seen node to decide what to do. If the least-recently seen node fails to respond, it is evicted from the k-bucket and the new sender inserted at the tail. Otherwise, if the least-recently seen node responds, it is moved to the tail of the list, and the new sender's contact is discarded (Figure 1.5). k-buckets effectively implement a least-recently seen eviction policy, except that live nodes are never removed from the list. This policy derives from the statistical analysis discussed earlier. By keeping the oldest live contacts around, k-buckets maximize the probability that the nodes they contain will remain online. A second benefit of k-buckets is that they provide resistance to certain DoS attacks. One cannot flush nodes' routing state by flooding the system with new nodes. Kademlia nodes will only insert the new nodes in the k-buckets when old nodes leave the system.

Kademlia protocol

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

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

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

- FIND NODE: FIND NODE takes a 160-bit ID as an argument. The recipient of the RPC returns <IP address; UDP port; Node ID> triples for the k nodes

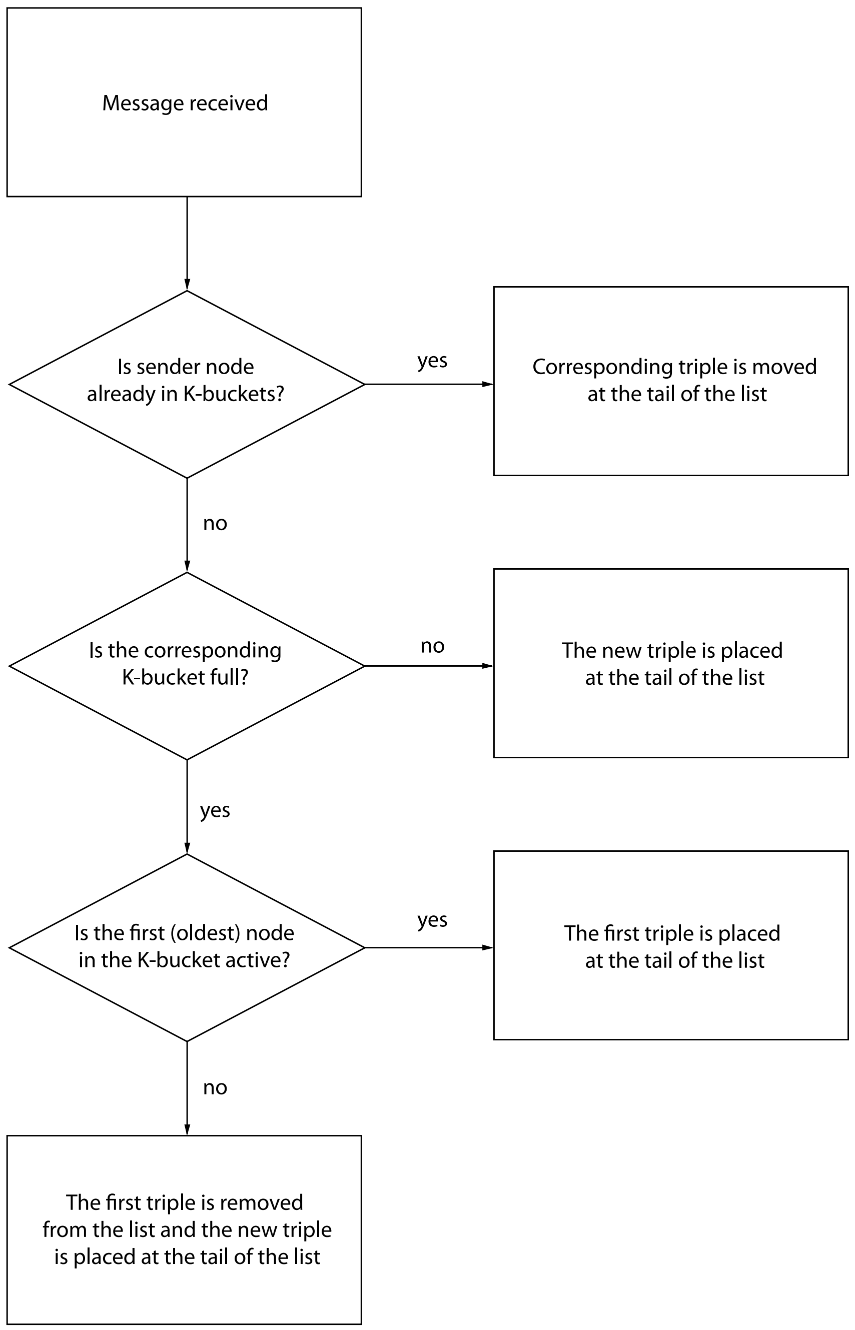

Message received
Is sender node already in K-buckets?
yes
Corresponding triple is moved at the tail of the list
no
Is the corresponding K-bucket full?
no
The new triple is placed at the tail of the list
yes
Is the first (oldest) node in the K-bucket active?
yes
The first triple is placed at the tail of the list
no
The first triple is removed from the list and the new triple is placed at the tail of the list

Figure 1.5: Kademlia k-buckets management policy

it knows about closest to the target ID. These triples can come from a single k-bucket, or they may come from multiple k-buckets if the closest k-bucket is not full. In any case, the RPC recipient must return k items (unless there are fewer than k nodes in all its k-buckets combined, in which case it returns every node it knows about). This is a primitive operation, not an iterative one.

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

Node lookup procedure

The most important procedure a Kademlia participant must perform is to locate the k closest nodes to some given node ID. We call this procedure a node lookup. Kademlia employs an iterative algorithm for node lookups (although the paper describes it as recursive). The lookup initiator starts by picking α nodes from its closest non-empty k-bucket (or, if that bucket has fewer than α entries, it just takes the α closest nodes it knows of). The initiator then sends parallel, asynchronous FIND NODE RPCs to the α nodes it has chosen. α is a system-wide concurrency parameter, such as 3. In the iterative step, the initiator resends the FIND NODE to nodes it has learned about from previous RPCs. This iteration can begin before all α of the previous RPCs have returned. Kademlia uses $\alpha = 3$, as its degree of parallelism. It appears that this value is optimal. There are at least three approaches to managing parallelism. The first is to launch α probes and wait until all have succeeded or timed out before iterating. This is termed **strict parallelism**. The second is to limit the number of probes in flight to α; whenever a probe returns a new one is launched. We might call this **bounded parallelism**. A third is to iterate after what seems to be a reasonable delay (duration unspecified), so that the number of probes in flight is some low multiple of α. This is called **loose parallelism**. Of the k nodes the initiator has heard of closest to the target, it picks α that it has not yet queried and resends the FIND NODE RPC to them. Nodes that fail to respond quickly are removed from consideration until and unless they do respond. If a round of FIND NODEs fails to return a node any closer than the closest already seen, the initiator resends the FIND NODE to all of the k closest nodes it has not already queried. The lookup terminates when the initiator has queried and gotten responses from the k closest nodes it has seen. When $\alpha = 1$, the lookup algorithm resembles Chord's in terms of message cost and the latency of detecting failed nodes. Most operations are implemented in terms of the above lookup procedure:

- **iterativeStore**: this is the Kademlia storing operation. The initiating node does an iterativeFindNode, collecting a set of k closest contacts, and then sends a primitive STORE RPC to each. iterativeStores are used for publishing or replicating data on a Kademlia network.
- **iterativeFindNode**: this is the basic Kademlia node lookup operation. As described above, the initiating node builds a list of k closest contacts using iterative node lookup and the FIND NODE RPC. The list is returned to the caller.
- **iterativeFindValue**: this is the Kademlia search operation. It is conducted as a node lookup, and so builds a list of k closest contacts. However, this is done using the FIND VALUE RPC instead of the FIND NODE RPC. If at any time during the node lookup the value is returned instead of a set of contacts, the search is abandoned and the value is returned. Otherwise, if no value has been found, the list of k closest contacts is returned to the caller.

Join and replication rules

A node joins the network as follows:

- it generates its ID n;
- it inserts the value of some known node c into the appropriate bucket as its first contact;
- it does an iterativeFindNode for n;

- it refreshes all buckets further away than its closest neighbor, which will be in the occupied bucket with the lowest index.

Data are stored using an iterativeStore procedure, which has the effect of replicating it over the k nodes closest to the key. If a node, during its uptime, discovers a node that is closer to some key it currently stores, it send a STORE for that key to the closer node.

1.2.3 Other DHTs

Chord and Kademlia are not the only DHT algorithm that have been defined. CAN [3], Pastry [4], and Tapestry [5] are examples of other important proposals of DHT algorithms. Every DHT algorithm defines its own metrics, topology, and routing scheme. In CAN, which stands for Content Addressable Network, items are placed in a d-dimensional lattice (coordinate system). The entire coordinate space is dynamically partitioned among all the nodes in the system such that every node is responsible for at least one distinct zone within the overall space. Pastry is similar to Chord, as nodes are arranged on a ring. However, the state information and routing are different: each Pastry node maintains a routing table, a neighborhood set and a leaf set. Tapestry is an extensible infrastructure that provides decentralized object location and routing focusing on efficiency and minimizing message latency. This is achieved since Tapestry constructs locally optimal routing tables from initialization and maintains them in order to reduce routing stretch. Furthermore, Tapestry allows object distribution determination according to the needs of a given application. Similarly Tapestry allows applications to implement multicasting in the overlay network.
All these DHT algorithms however share the same features, that is, they offer logarithmic upper bounds for information storage and retrieval and nodes maintain a logarithmic state.

1.2.4 One-hop DHTs

In traditional DHTs, nodes keep a limited amount of state information and lookup procedures typically require multiple hops before reaching the targeted node. Some scenarios require that lookups take as little time as possible to get executed. One-hop DHTs have been proposed in order to achieve faster lookup times, by ensuring that a single hop is needed to reach the target, with high probability. Among other benefits introduced by one-hop DHTs is the minimization of maintenance network traffic. The tradeoff to achieve these goals is to keep a much higher amount of state information ($O(n)$). D1HT [6] is an implementation of a one-hop DHT based on Event Detection and Reporting Algorithm (EDRA) technique. The topology of D1HT resembles Chord's ring. Since every node should keep information about any other node in the DHT, events are propagated throughout the overlay so that each node can detect when other nodes join or leave. Event propagation is essential to ensure that routing tables are kept consistent and updated reactively to churn events.

1.3 Bootstrapping

Peer-to-peer (P2P) overlay networks are used in those scenarios where decentralization, self-organization, and fault-tolerance are desired. However, P2P systems are never fully distributed as they typically rely on some centralized network elements or prior knowledge for bootstrapping, that is, to let new nodes join the overlay. Therefore, actual decentralization and self-organization cannot be achieved. Although this is not a problem in current P2P applications over the Internet where some nodes (*supernodes*) can be considered as permanently connected and sufficiently reliable, this becomes a problem when applied to very dynamic and self-organizing intranet or enterprise networks where all nodes may have a very dynamic behavior, leading to the impossibility to guarantee any sort of reliable and centralized bootstrap service. The bootstrapping mechanism can be very sophisticated and can include the verification of a certificate from the joining peer, as well as the generation of a peer-id that determines the position of the joining peer inside the overlay. The details of the admission mechanism are not discussed here, as they depend on the security policy of the overlay that the peer is willing to join. However, a peer which is willing to join any P2P network needs to discover the location of a bootstrap peer to send its join request to. Current solutions include the use of:

- cached mechanisms: a peer maintains a list of previously discovered peers and tries to contact them; this approach does not solve however the problem when the peer is trying to join the overlay for the first time as its cached list would be empty;
- server-based mechanisms: a peer contacts a pre-configured server node (or list of nodes); this approach has the obvious disadvantage of being server-centric, which is an antithetical solution for the goal of a purely distributed network and would create a bottleneck in the network and a possible point of failure; it is important to say that the failure of the bootstrap server does not affect the behavior of the P2P overlay, but only prevents new peers from joining the network;
- multicast-based mechanisms: multicast support is exploited as proposed in Chapter 2.

The first two approaches have drawbacks. For instance, it might not always be possible to know in advance a list of nodes that are always active to be used for bootstrapping, caching does not work in case the new node is joining the overlay for the very first time or if cached nodes are no longer enrolled in the overlay, and server-based mechanisms might be useless if the server is unreachable. The use of mechanisms that combine pre-configured lists which are hardcoded into the protocol and caching has proved to work in practice (i.e. eMule bootstrapping), but it is still potentially exposed to the risk of failure. Moreover, such mechanisms do not work in all those scenarios in which the P2P networks are built in a complete distributed and self-organized manner (for example in case of server-free distributed enterprise networks or ad-hoc networks). In Chapter 2, a scalable and distributed solution to bootstrapping based on Multicast is presented.

Chapter 2

A Multicast-based approach to Peer-to-peer Bootstrapping

2.1 Introduction

The peer-to-peer (P2P) network paradigm has been introduced in order to overcome some shortcomings of the client-server architecture by providing such features as decentralization, self-organization, scalability, and fault-tolerance. Bootstrapping is the initial process through which new nodes can join an existing P2P overlay network. Typically, a joining peer must first contact a bootstrap peer, which is a peer already enrolled in the overlay. The bootstrap peer is responsible for admitting the new peer by passing information about other peers so that the new peer can actively participate in the overlay. Finding a suitable bootstrap peer is therefore a critical issue. Although different P2P systems have been defined and deployed, the problem of bootstrapping has usually been solved by introducing such mechanisms as the use of a pre-configured list of nodes, caching, or server-based discovery. Unfortunately, although they work in P2P applications running over the Internet, they show some problems when applied to very dynamic and self-organizing intranet or enterprise network scenarios. In fact, in these cases all nodes may join and leave the network very dynamically, without the possibility of guaranteeing any sort of permanent centralized service as current bootstrap solutions may require. In this chapter, a multicast-based bootstrapping mechanism for dynamic and self-organized P2P networks is presented. The proposed approach describes a new mechanism for discovering a bootstrap node in a P2P network, which allows a joining peer to discover a proper bootstrap peer in a real distributed manner. The mechanism, named BANANAS (BootstrAp Node NotificAtion Service) [7], is based on multicast communications. It provides a completely distributed, self-organizing and scalable discovery service and employs an unsolicited approach and well-performs in terms of scalability, load-balancing, and mean frequency of information exchange. Although the use of IPv4 multicast is currently not supported amongst the public Internet, it is implemented in several ISPs, private, or enterprise networks and it is expected in the future to be supported within more and more IPv4 and IPv6 networks. Cramer et al. [8] have discussed some possible mechanisms for the bootstrapping process, such as based on: static bootstrap servers, dynamic web caches, random access probing, multicast, or IPv6 anycast. The first two mechanisms suffer of the well-known centralization, low reliability, and non-self-organization problems; random access probing, which consists in trying several random entry points until a success is reached, may result in large number of failures and large amount of network traffic. Anycast is also considered as mechanism for acquiring an IP address of a potential bootstrap peer, however it is also pointed out that such mechanism just moves the problem of node selection at the network layer, and has the drawback that it may limit the requesting nodes' freedom of choice. The authors also considered multicasting as possible mechanism combined with the expanding ring search (ERS). ERS in turn works by searching successively larger areas in the network centered around the source of broadcast. Searching areas may be limited by using increasing values of TTL (Time To Live). However this approach has some limitations such as:

- TTL scoping requires "successive containment" property and will not work with overlapping regions;

- by increasing the TTL, the multicast scope may rapidly expand to a large portion of the entire network resulting in flooding query packets to a very large number of nodes;

- for each TTL value, a proper maximum round-trip time (RTT) has to be considered (measured or pre-configured).

The authors in [9] propose a bootstrap service based on random access probing. The bootstrap service relies on a separate, dedicated, and unique P2P bootstrap overlay where bootstrapping information is stored. The bootstrap overlay is used in order to exploit random access probing, which proved to be more efficient in large P2P networks, and can be accessed through two basic methods (*lookup()* and *publish()*). The bootstrap overlay is based on a DHT in order to achieve load balancing among the participating nodes. However, the bootstrap service still suffers of the following drawbacks:

- it simply shifts the problem of joining the P2P overlay network to that of joining the P2P bootstrap overlay;
- it is based on DHTs, which may suffer of some security issues, such as poisoning or Sybil attacks;
- it relies on a PULL approach, that is, joining nodes issue a request and receive a response with bootstrapping information; even though a load balancing effort has been made in order to avoid overloading nodes responsible for the key of a popular overlay, each node possibly needs to handle an unpredictable number of requests.

Because of these reasons, we propose a different mechanism which is PUSH-based, that is, the joining node does not issue any request and bootstrap information is notified by the service. Moreover, our approach does not require any information storage system such as DHTs in order to keep bootstrapping information since each node simply notifies its own presence, thus avoiding the risk of overloading nodes.

2.2 Solicited vs. Unsolicited Approach

In this section, we will briefly discuss about the implication of the usage of the PUSH and PULL approaches in a multicast-based service. Let us consider the case in which all peers that act as bootstrap nodes or that directly know one or more bootstrap nodes are enrolled in a multicast group. There are two possible approaches for the discovery of a bootstrap peer in a multicast fashion.

2.2.1 Solicited (PULL) approach

A peer which tries to join the overlay sends a request to all the nodes in the group asking for bootstrap nodes; the nodes that would respond to such request would all be candidates to admit the peer. However, such a solicited mechanism would cause an overload of the network, especially when many bootstrap nodes are already in the overlay because all notification responses are sent for each joining node. Note that, amongst all response messages, only one is used by the requesting peer since only one bootstrap peer is needed to join the overlay. Moreover, also limiting the total number of responses from bootstrap nodes does not limit the total amount of messages spread over the network since it strictly depends on the total joining rate, multiplied by the cardinality of the multicast group.

2.2.2 Unsolicited (PUSH) approach

All the bootstrap nodes in the multicast group send unsolicited messages to all nodes in the group to advertise their presence in the overlay as well as their bootstrap information. When a node needs to discover a bootstrap node, it simply joins the multicast group and listens for these messages. This approach potentially may still have scalability issues due to the large number of messages sent over the network. However in this case, differently from the previous approach, such total amount of sent messages may be limited, regardless of the actual joining rate. This can be achieved if the nodes cooperate in order to ensure that the total amount of messages sent over the multicast group is constant or upper bounded, regardless of the effective number of collaborating bootstrap nodes. This second approach is the one followed in this work and appears to be the most efficient for the reasons reported above.

2.3 Simple Algorithm

The goals of the service are to provide a service characterized by an almost constant rate of messages received and to fairly balance the number of messages sent by nodes. Both bootstrap and joining nodes join the same multicast group. Each node sees a timeline divided into slots of length T, where $f = \frac{1}{T}$ is the average rate at which a message is to be received by any node in the group. A simple algorithm, which will be described next, can be used in order to achieve these goals.

2.3.1 Synchronized case

Suppose all nodes are synchronized, that is, their time slots are perfectly aligned. At the beginning of the slot, each node computes a random time t_i, uniformly distributed in the $[0,T]$ interval. When time t_i is reached, the node decides whether to actually send the message or not, depending on the fact that a message has already been received between time 0 and time t_i. If no message was received, then the node sends its message, otherwise it waits for the next slot, and repeats the above procedure. In a given time slot, the message will be sent by the node which computed the shortest time t_{min}, and all other nodes will cancel their scheduled message. Let $\mathbf{t_1}$, $\mathbf{t_2}$, ..., $\mathbf{t_n}$ be a set of n independent random variables, uniformly distributed in the interval $[0,T]$.

$$f_{T_i}(t_i) = \begin{cases} \frac{1}{T} & \text{if } 0 \le t_i \le T \\ 0 & \text{elsewhere} \end{cases} \tag{2.1}$$

Integration of $f_{T_i}(t_i)$ yields the cumulative distribution function of the random variable $\mathbf{t_i}$:

$$F_{T_i}(t_i) = \begin{cases} 0 & \text{if } t_i < 0 \\ \frac{t_i}{T} & \text{if } 0 \le t_i \le T \\ 1 & \text{if } t_i > T \end{cases} \tag{2.2}$$

Let $\mathbf{t_{min}} = min(\mathbf{t_1}, \mathbf{t_2}, ..., \mathbf{t_n})$. We wish to find the cumulative distribution function and mean value of the random variable $\mathbf{t_{min}}$.

$$\{\mathbf{t_{min}} \leq t\} = \{\mathbf{t_{min}} > t\}^{'} = \left\{ \bigcap_{i=1}^{n} \{\mathbf{t_i} > t\} \right\}^{'} \tag{2.3}$$

The cumulative distribution function of the random variable $\mathbf{t_{min}}$ can be calculated as follows:

$$\begin{aligned}
P(\mathbf{t_{min}} \leq t) &= \Pr(min(\mathbf{t_1}, \mathbf{t_2}, ..., \mathbf{t_n}) > t)^{'} = \\
&= \Pr\left(\left\{ \bigcap_{i=1}^{n} \{\mathbf{t_i} > t\} \right\}^{'} \right) = \\
&= 1 - \Pr\left(\left\{ \bigcap_{i=1}^{n} \{\mathbf{t_i} > t\} \right\} \right) = \\
&= 1 - \Pr(\mathbf{t_i} > t)^{n} = \\
&= 1 - (1 - P(\mathbf{t_i} \leq t))^{n} = \\
&= 1 - (1 - F_{T_i}(t))^{n} = \\
&= F_{T_{min}}(t)
\end{aligned}$$

Therefore

$$F_{T_{min}}(t) = \begin{cases} 0 & \text{if } t < 0 \\ 1 - \left(1 - \frac{t}{T}\right)^{n} & \text{if } 0 \leq t \leq T \\ 1 & \text{if } t > T \end{cases} \tag{2.4}$$

Derivation of $F_{T_{min}}(t)$ yields the probability density function $f_{T_{min}}(t)$:

$$f_{T_{min}}(t) = \begin{cases} \frac{n}{T}\left(1 - \frac{t}{T}\right)^{n-1} & \text{if } 0 \leq t \leq T \\ 0 & \text{elsewhere} \end{cases} \tag{2.5}$$

The mean value of the random variable $\mathbf{t_{min}}$ is:

$$\varepsilon = E\{\mathbf{t_{min}}\} = \int_{-\infty}^{+\infty} f_{T_{min}}(t)\, dt = \frac{T}{n+1} \tag{2.6}$$

As the number of nodes increases, the mean departure time value generated by the elected node decreases, and it will tend to 0 as the number of nodes tends to infinity. Therefore, a message will be sent at the beginning of each slot and all other messages will be dropped (not sent). The average rate of messages sent to the group would be $f = \frac{1}{T}$. Moreover, since each round of computation of the random time t_i is independent from the previous ones and from the other nodes, no assumption can be made about which node will be elected in a given round, so the probability of a node to be elected will be $\frac{1}{n}$, if n nodes are participating in the group.

2.3.2 Unsynchronized case

Let's remove the hypothesis about the synchronization of the time slots among the nodes. In this case, we use the same approach seen above, with one main difference: the reception of a message is used as a synchronization event among the nodes. To do so, instead of computing the time of the next scheduled message every T, we compute it starting from one slot after the time of reception or sending a message. This approach eliminates the need for synchronization among the nodes, but has the disadvantage that it may increase the mean time of a message being sent in the group. The time between two successive messages is the minimum computed time t_{min} plus T. Again, let $\mathbf{t_{min}} = min(\mathbf{t_1}, \mathbf{t_2}, ..., \mathbf{t_n})$. The mean time between two successive messages is:

$$E\{\mathbf{t_{min}} + T\} = E\{\mathbf{t_{min}}\} + T = \varepsilon + T = T \cdot \frac{n+2}{n+1} \tag{2.7}$$

However, as n increases, the average time tends to T.

2.3.3 Problems with the simple algorithm

The algorithms sketched above assume that all nodes are able to detect immediately a received notify message and at the same time stop the sending process scheduled for the same time slot. Although this is applicable in case of zero network delay, that is, the case in which the time needed for delivering a message is almost zero, it does not apply to real-world networks, because of the actual physical time requirements for a message to be delivered from end to end. On the contrary, it is possible that a message is still sent by one node that has computed a higher random value but that has not yet received the message from the real winning (elected) node. We call τ_i the *vulnerability interval* for the i-th node, that is, the time needed for the i-th node to receive the message sent from the elected node. Since in general such τ_i depends on node i, to the network topology, and to the current network traffic, we consider a *system vulnerability interval* of length τ, which is the worst-case estimation of the delivery time of a message. Due to the difficulty to estimate the worst-case network delivery time, any pre-configured upper bound can be considered. We refer to a *collision* as the event that a message is sent after the message sent from the actual elected node. A collision occurs any time a node that scheduled a departure time greater than the elected node's does not receive the message sent by the elected node before its own departure time, due to network delay. We wish to calculate the mean number of collisions for a network consisting of n nodes. Let t_{min} be the minimum amongst the values $t_1, t_2, \dots, t_n$ generated by the n nodes. Since the minimum value is t_{min}, all the nodes that have not generated t_{min} must have generated a value $\mathbf{t} > t_{min}$. Let's calculate the conditioned distribution of $\mathbf{t}$, given the event $\{\mathbf{t} > t_{min}\}$. Let $\bar{\mathbf{t}} = \{\mathbf{t} | \mathbf{t} > t_{min}\}$.

$$
\begin{aligned}
F_{\bar{T}}(t) &= \Pr\left(\bar{\mathbf{t}} \leq t\right) = \\
&= \Pr\left(\mathbf{t} \leq x | \mathbf{t} > t_{min}\right) = \\
&= \frac{\Pr\left(\mathbf{t} \leq t, \mathbf{t} > t_{min}\right)}{\Pr\left(\mathbf{t} > t_{min}\right)} = \\
&= \frac{\Pr\left(t \leq \mathbf{t} \leq t_{min}\right)}{1 - \Pr\left(\mathbf{t} \leq t_{min}\right)} = \\
&= \begin{cases} 0 & \text{if } t < t_{min} \\ \dfrac{F_T(t) - F_T(t_{min})}{1 - F_T(t_{min})} & \text{if } t_{min} \leq t \leq T \\ 1 & \text{if } t > T \end{cases} \\
&= \begin{cases} 0 & \text{if } t < t_{min} \\ \dfrac{t - t_{min}}{T - t_{min}} & \text{if } t_{min} \leq t \leq T \\ 1 & \text{if } t > T \end{cases}
\end{aligned}
$$

Therefore the c.d.f. of the random variable **t** is:

$$F_{\bar{T}}(t) = \begin{cases} 0 & \text{if } t < t_{min} \\ \dfrac{t - t_{min}}{T - t_{min}} & \text{if } t_{min} \leq t \leq T \\ 1 & \text{if } t > T \end{cases} \tag{2.8}$$

Derivation of $F_{\bar{T}}(t)$ yields:

$$f_{\bar{T}}(t) = \begin{cases} \dfrac{1}{T - t_{min}} & \text{if } t_{min} \leq t \leq T \\ 0 & \text{elsewhere} \end{cases} \tag{2.9}$$

The probability that one of the $n-1$ nodes that have not generated the minimum value t_{min} generated a value in the interval $[t_{min}, t_{min} + \tau]$ is:

$$F_{\bar{T}}(t_{min} + \tau) = \begin{cases} 0 & \text{if } t_{min} < 0 \\ \dfrac{\tau}{T - t_{min}} & \text{if } 0 \leq t_{min} \leq T - \tau \\ 1 & \text{if } t_{min} > T - \tau \end{cases} \tag{2.10}$$

We define

$$p(t_{min}) = F_{\bar{T}}(t_{min} + \tau) \tag{2.11}$$

The probability is function of the minimum value t_{min}. The probability that k of the the $n-1$ nodes generated a value in the interval $[t_{min}, t_{min} + \tau]$ (that is, the probability to get k collisions) is:

$$\begin{pmatrix} n-1 \\ k \end{pmatrix} \cdot p(t_{min})^k \, (1 - p(t_{min}))^{n-1-k} \tag{2.12}$$

This probability is function of n, t_{min}, and k. Given n and t_{min}, the distribution of the probability is a discrete binomial distribution, whose mean value is $(n-1)p(t_{min})$. We can then state that at each round, there will be k collisions in average, where:

$$E\{k\} = \begin{cases} (n-1)\dfrac{\tau}{T - t_{min}} & \text{if } 0 \leq t_{min} \leq T - \tau \\ n-1 & \text{if } t_{min} > T - \tau \end{cases} \tag{2.13}$$

As n increases, t_{min} tends to 0, and therefore the number of collisions tends to the ratio $\frac{\tau}{T} \cdot (n-1)$.

2.4 Enhanced Algorithm

In order to avoid the problems outlined in the previous section, an enhanced version of the algorithm is used. The algorithm first estimates the number of nodes that are currently enrolled in the group, and then exploits this information to schedule the time of sending of the message.

2.4.1 Estimation of the number of collaborating nodes

Let us suppose that, each time a node sends a notification message, it includes also its scheduled departure time t_i. The knowledge of the minimum t_{min} of a set of uniformly distributed random variables and the number of random variables k whose value belongs to the interval $[t_{min}, t_{min} + \tau]$, is used to estimate the number of such random variables.
We can invert equation (2.13) to get the number of nodes as a function of the minimum generated time t_{min} and the mean number of collisions $E\{k\}$:

$$n = \begin{cases} E\{k\} \frac{T - t_{min}}{\tau} + 1 & \text{if } 0 \le t_{min} \le T - \tau \\ E\{k\} + 1 & \text{if } t_{min} > T - \tau \end{cases} \tag{2.14}$$

If we assume that the number of messages that were received in the interval $[t_{min}, t_{min} + \tau]$ coincides with the mean number of k ($k = E\{k\}$), then it is possible to invert equation (2.13) to estimate n:

$$\hat{n} = \begin{cases} k\frac{T - t_{min}}{\tau} + 1 & \text{if } 0 \le t_{min} \le T - \tau \\ k + 1 & \text{if } t_{min} > T - \tau \end{cases} \tag{2.15}$$

Since the assumption that $k = E\{k\}$ was made, the estimation works well if k is big, and therefore if n is big. $\hat{n}$ tends to overestimate the actual value of n. We handle the possibility of undelivered messages by considering the probability of a lost message $p_{loss} = \Pr\{\text{message is lost}\}$. The number of collisions must be therefore adjusted by dividing it by p_{loss}.

2.4.2 Scheduling

We now want to exploit the information about the number of nodes that are currently participating in the group to generate a schedule in order to fairly balance the number of messages among the nodes while respecting the goal of having a constant rate of sent messages within the group. Let $\hat{n}$ be the estimated number of nodes. Any node in the group randomly selects a number between 1 and $\hat{n}$. The selected number will correspond to a particular time slot self-assigned to the node. The node will wait until its time slot and will send its message at a randomly selected time in that interval. The departure time within the slot is randomized in order to let that, in case two or more nodes will select the same slot, one node would send its message first and the other nodes may detect such message and reschedule their message in a successive timeslot. We will now evaluate this scheduling mechanism. Let n be the number of participants to the algorithm. Each participant selects a number between 1 and n. Let's define the event

$$E_{n,k} = \{k \text{ different values are selected out of } n\}.$$

We wish to calculate the mean value of the k different numbers selected by the n particpants. Since the selection of the number is independent from participant to participant, it may happen that the same number is selected by different participants. If n is the number of participants and each participant selects a number between 1 and n, the probability that a total of k different numbers are selected can be written as:

$$\Pr\{E_{n,k}\} = P_{n,k} = \frac{X_{n,k}}{N} \tag{2.16}$$

where N is the total number of possible outcomes of the event:

$$E_N = \{n \text{ particpants select a number between 1 and } n \text{ independently}\}$$

and $X_{n,k}$ is the total number of outcomes of the event $E_{n,k}$. E_N is clearly the event of generating all the dispositions of n elements from a set of n numbers. The number of all the outcomes is $N = n^n$. If $k = 1$, $X_{n,k} = n$ since there are n possible sets of n values that contain exactly one value (one for each value). If $k \geq 2$, $X_{n,k}$ can be written by the following formula:

$$X_{n,k} = \frac{n!}{(n-k)!} \sum_{i_1=0}^{n-k} k^{i_1} \sum_{i_2=0}^{n-k-i_1} (k-1)^{i_2} \cdots \sum_{i_{k-1}=0}^{n-k-\sum_{j=1}^{k-2} i_j} 2^{i_{k-1}} \tag{2.17}$$

Let's focus on the case $k \geq 2$. If we define

$$\mathbf{I_{n,k}} = \left\{ \underline{\mathbf{i}} \in \mathbf{N}^{k-1} : \|\underline{\mathbf{i}}\|_1 = \sum_{j=1}^{k-1} i_j \leq n-k \right\} \tag{2.18}$$

and the exponentiation of two vectors as

$$(a_1, a_2, \ldots, a_n)^{(b_1, b_2, \ldots, b_n)^T} = \prod_{i=1}^{n} (a_i)^{b_i} \tag{2.19}$$

then we can rewrite equation (2.17) as:

$$X_{n,k} = \frac{n!}{(n-k)!} \sum_{\underline{\mathbf{i}} \in \mathbf{I_{n,k}}} (\underline{\mathbf{K}})^{\underline{\mathbf{i}}^T} \tag{2.20}$$

where

$$\underline{\mathbf{K}} = (k, k-1, \ldots, 2) \tag{2.21}$$

and

$$\underline{\mathbf{i}} = (i_1, i_2, \ldots, i_{k-1}). \tag{2.22}$$

It is possible to find the cardinality $\theta_{n,k}$ of the set $\mathbf{I_{n,k}}$ by the following formula:

$$\theta_{n,k} = \frac{(n-1)!}{(n-k)! \cdot (k-1)!} = \begin{pmatrix} n-1 \\ k-1 \end{pmatrix} \tag{2.23}$$

Now we can define a matrix $Y_{n,k} \in \mathbf{M}(\theta_{n,k}, k-1)$ as:

$$Y_{n,k} = \begin{pmatrix} \underline{i_1} \\ \underline{i_2} \\ \vdots \\ \underline{i_{\theta_{n,k}}} \end{pmatrix} \tag{2.24}$$

We can extend the definition of vector exponentiation to the case of exponentiating a vector to a matrix. Let $\underline{a}$ be a vector of n elements and B a matrix $\in \mathbf{M}(n,m)$. Then, the result of exponentiating vector $\underline{a}$ to matrix B is a vector of m elements:

$$\underline{a}^B = (a_1, a_2, \dots, a_n)^{\left(\underline{b_1}, \underline{b_2}, \dots, \underline{b_m}\right)} = \left(a^{\underline{b_1}}, a^{\underline{b_2}} \dots a^{\underline{b_m}}\right) \tag{2.25}$$

where by $\underline{b_i}$ we denote the i-th column of B ($\underline{b_i}$ has n elements). Now we can write

$$\underline{\mathbf{K}}^{Y_{n,k}^T} = \left(\underline{\mathbf{K}}^{\underline{i_1}^T}, \underline{\mathbf{K}}^{\underline{i_2}^T}, \dots, \underline{\mathbf{K}}^{\underline{i_{\theta_{n,k}}}^T}\right) = \underline{\gamma_{n,k}} \tag{2.26}$$

and finally

$$\Gamma_{n,k} = \left\|\underline{\gamma_{n,k}}\right\|_1 = \sum_{j=1}^{\theta_{n,k}} \underline{\mathbf{K}}^{\underline{i_j}^T} = \sum_{\underline{\mathbf{i}} \in \mathbf{I_{n,k}}} (\underline{\mathbf{K}})^{\underline{\mathbf{i}}^T} \tag{2.27}$$

Therefore we can write:

$$X_{n,k} = \frac{n!}{(n-k)!} \cdot \Gamma_{n,k} \tag{2.28}$$

Equation (2.28) lets us find the number of different sets of n elements that contain k different elements when those elements are drawn from a set of n different elements. Since $\Gamma_{n,k}$ is clearly function of both n and k, our goal is to find a function $\phi : \mathbf{N}^2 \to \mathbf{N}$ such that $\phi(n,k) = \Gamma_{n,k}$. It is banal to see that $\Gamma_{n,n} = 1$. Let's construct a tree whose branches are all the vectors $\underline{i} \in \mathbf{I_{n,k}}$, that is, all the $\theta_{n,k}$ rows of the matrix $Y_{n,k}$. The tree has clearly $k-1$ levels. Level l is assigned a weight that corresponds to the value of the l-th element of the vector $\underline{\mathbf{k}} = \underline{k_l}$. Therefore, levels have weights that vary from k (the highest level), to 2. At level 1 (weight = k), the tree forks in $n-k+1$ branches. Each branch is labeled with a value that ranges from 0 to $n-k$. Since each branch corresponds to a vector of $\mathbf{I_{n,k}}$, the last branch (labeled with $n-k$) cannot fork anymore since the sum of all labels cannot exceed $n-k$. Therefore the path towards the leaf is subsequently followed by branches all labeled with 0. Therefore, such path leads to a value of k^{n-k}. All other branches create subtrees. The branch labeled with $n-k-1$ forks into two subtrees, one with label 0 and one with label 1. The branch with label 1 has saturated the value of the sum of the labels and therefore there only one path towards the leaf, which leads to a value of $k^{n-k-1} \cdot (k-1)^1$. The other branch creates again a subtree with labels 0 and 1. The branch with label 1 leads to a value of $k^{n-k-1} \cdot (k-2)^1$, while the other one creates another subtree. At level $k-1$, the subtree that results from all branches labeled with 0, create a final subtree with labels that range from 0 to $n-k$, leading to values that are 1, 2, 2^2, ..., 2^{n-k}. The sum of all values of this subtree is therefore $\sum_{i=0}^{n-k} 2^i = 2^{n-k+1} - 1$. Figures 2.1 and 2.2 show the solution for the limit cases $\Gamma_{n,2}$ and $\Gamma_{n,n}$.

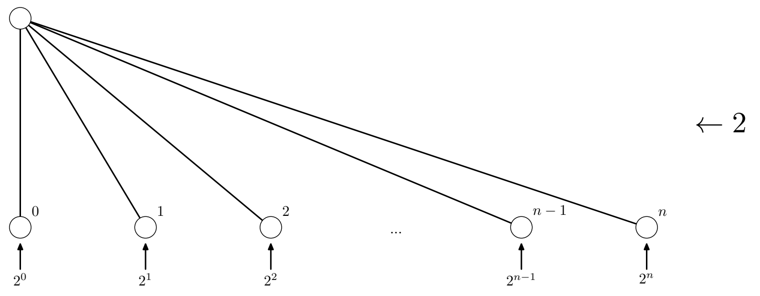


Figure 2.1: Procedure to get the value of $\Gamma_{n,2}$

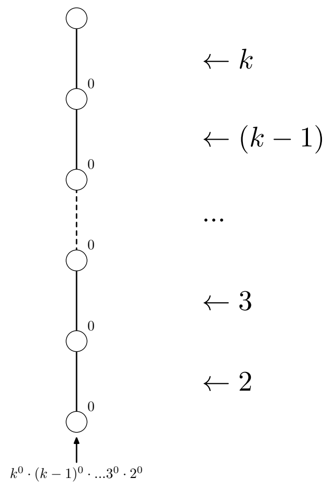
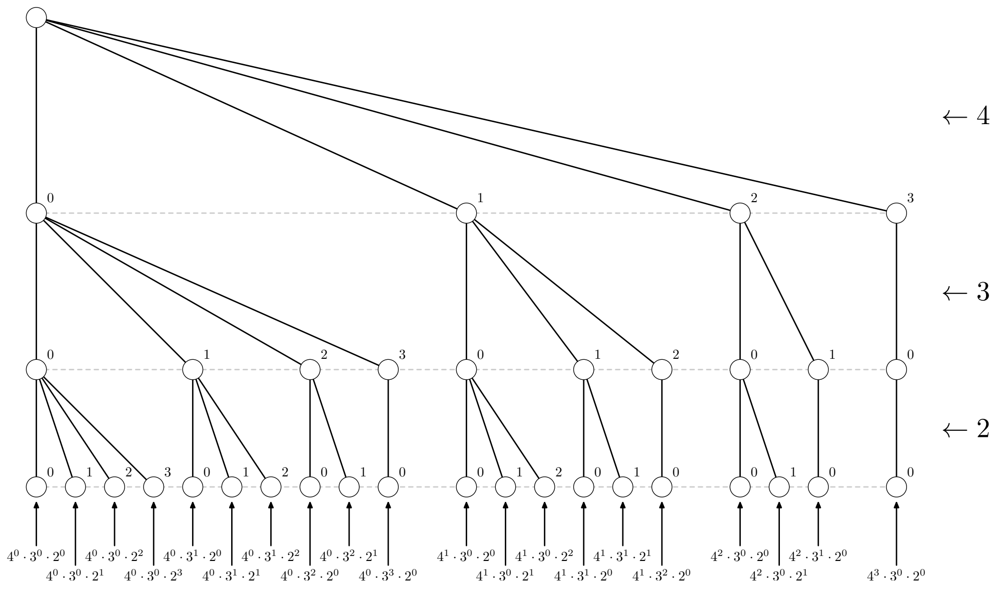


Figure 2.2: Procedure to get the value of $\Gamma_{n,n}$

Following the procedure described above, we can get the value of $\Gamma_{n,k}$ by reducing the problem to all the sub-problems of lower degree, in a recursive fashion. Such a reduction leads to an iterative formulation of the value of $\Gamma_{n,k}$:

$$\begin{cases} \Gamma_{n,k} = \sum_{i=0}^{n-k} k^i \cdot \Gamma_{n-1-i,k-1} \\ \Gamma_{n,n} = 1 \\ \Gamma_{n,2} = \sum_{i=0}^{n-2} 2^i = 2^{n-1} - 1 \end{cases} \tag{2.29}$$

Figure 2.3 shows an example of the procedure needed to get the value for $\Gamma_{7,4}$. With



← 4
← 3
← 2
$4^0 \cdot 3^0 \cdot 2^0$
$4^0 \cdot 3^0 \cdot 2^1$
$4^0 \cdot 3^0 \cdot 2^2$
$4^0 \cdot 3^0 \cdot 2^3$
$4^0 \cdot 3^1 \cdot 2^0$
$4^0 \cdot 3^1 \cdot 2^1$
$4^0 \cdot 3^1 \cdot 2^2$
$4^0 \cdot 3^2 \cdot 2^0$
$4^0 \cdot 3^2 \cdot 2^1$
$4^0 \cdot 3^3 \cdot 2^0$
$4^1 \cdot 3^0 \cdot 2^0$
$4^1 \cdot 3^0 \cdot 2^1$
$4^1 \cdot 3^0 \cdot 2^2$
$4^1 \cdot 3^1 \cdot 2^0$
$4^1 \cdot 3^1 \cdot 2^1$
$4^1 \cdot 3^2 \cdot 2^0$
$4^2 \cdot 3^0 \cdot 2^0$
$4^2 \cdot 3^0 \cdot 2^1$
$4^2 \cdot 3^1 \cdot 2^0$
$4^3 \cdot 3^0 \cdot 2^0$

Figure 2.3: Procedure to get the value of $\Gamma_{7,4}$

some calculations and index substitution it is easy to find an alternative and computationally more efficient representation for $\Gamma_{n,k}$. Given the above definition of $\Gamma_{n,k}$, we can write:

$$\Gamma_{n-1,k} = \sum_{j=0}^{n-k-1} k^j \cdot \Gamma_{n-2-j,k-1} \tag{2.30}$$

Therefore it is easy to demonstrate that:

$$\Gamma_{n,k} = \Gamma_{n-1,k-1} + k \cdot \Gamma_{n-1,k} \tag{2.31}$$

Even though there is no closed form to express $\Gamma_{n,k}$, a computer program can easily show that the mean of the number of non-empty slots tends to be approximately $k \approx 0.63 \cdot n$. Therefore, the scheduling proposed leaves approximately $0.37 \cdot n$ slots empty. This result can be derived either by computing the values of $\Gamma_{n,k}$ with the previous formulation, or by simulating the behavior of the community of participating nodes. Figures 2.4 and 2.5 show the alignment of our result with those extracted from simulation.

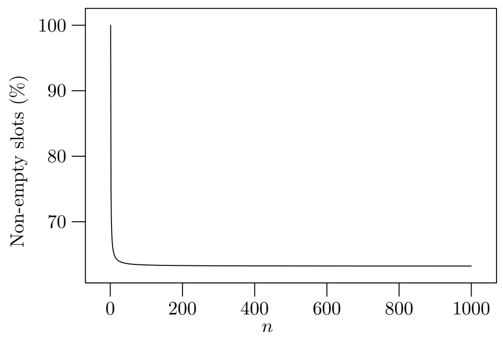


Figure 2.4: Ratio of non empty slots using mathematical model

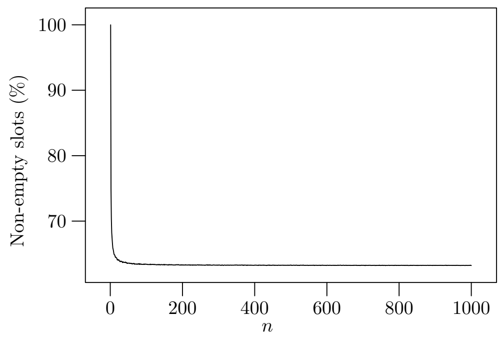


Figure 2.5: Ratio of non empty slots using simulations

In order to reduce the number of empty slots the picking could be made between 1 and $m = f(n)$ (with $m \leq n$), but collisions would be more luckily to occur. The handling of empty slots and collisions is discussed in the following section.

2.4.3 Algorithm description

The enhanced version of the algorithm requires two different stages:

1. a **synchronization round**: this stage is needed in order to let nodes learn about other nodes, that is, to let them estimate the number of nodes in the group ($\hat{n}$); the length of this stage is T;

2. a **notification round**: this stage is composed of $m = f(\hat{n})$ intervals of length T, during which nodes send the information about bootstrap nodes in the group;

Synchronization messages are received during synchronization rounds, which occur with variable period. A synchronization round's length is T. In this time interval, all synchronization messages are received. Synchronization messages are ordered by the reported t_i time generated by the sender. After all synchronization messages are received, the node counts all the distinct messages that report a t_i value that falls in the $[t_{min}, t_{min} + \tau]$ range (including its own t_i if it is in the range). Using the number k of such messages and the value of t_{min}, the node can estimate the number of nodes $\hat{n}$ that are present in the network using formula (2.15). The next synchronization round will occur after $m = f(\hat{n})$ slots.
The generation of a schedule can be made in a slightly different way than by picking a random value between 1 and m. Indeed, the statistics of the process remain the same if each node decides at the i-th slot whether to send the message or not with probability $p = \frac{1}{m-i}$. The node(s) that decide to send the message compute a random time t_i in the range $[0, T]$ to send the message. If a message is received prior to t_i, all the nodes that decided to send the message cancel their scheduled send. After sending a message, a node waits until the next synchronization round. After each node has determined when to send its own message, it also generates a random instant in the corresponding interval at which to send the message. By doing so, nodes that have

picked the same slot do not send their message simultaneously. If a node that has picked a slot receives a message during that slot, it cancels its scheduled message. In order to reduce the number of empty slots, a node that has picked an interval *x* that was also picked by another node, after canceling the sending of the message, it can pick another slot between *x* and *m*. The positive effect of refining the choice of a slot by a node after learning about a collision is shown in Figure 2.6.

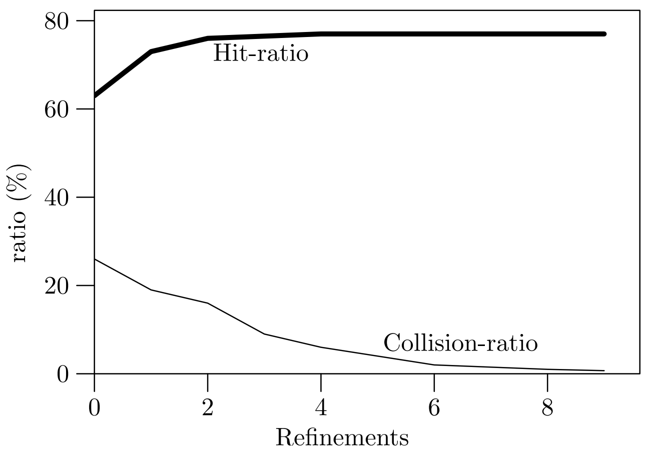


Figure 2.6: Improving hit-ratio and collision-ratio by refining the choice after detecting a collision

The presence of a synchronization round causes a slight decrease of the rate of information exchange since some round is dedicated to collecting information about the number of nodes that are participating. However, this decrement is not significant in the case of a high number of participating nodes.

Chapter 3

Distributed Location Service

3.1 Motivations for a Distributed Location Service

Peer-to-peer overlay networks allow participating nodes to communicate without relying on any central server. The communication, either connection-oriented or connectionless, is managed by a collection of intelligent nodes that are responsible for routing messages correctly towards the destination by implementing the overlay logic. There are basically two ways to implement peer-to-peer communication:

- to encapsulate the messages in a peer-to-peer protocol, which determines the next hop towards the recipient;
- to resolve the recipient of the message with a peer-to-peer resolution service to get a contact where the message will be directly sent.

Some application-layer protocols like SIP [10], SMTP [11], and HTTP [12] use URI or URL (RFC 3986 [13]) information to identify and address their resources, the endpoints (peers) of their communications, or simply the recipient of their messages. The use of an URI-based address mechanism instead of simply using IP addresses and port numbers is preferred since:

- IP addresses have a geographical distribution and are related to the specific (and sometimes temporary) network point of attachment of hosts;

- URI may contain more information on the addressed resource (user name, resource path, parameters, etc.).

The method used for mapping the URI to the actual next hop toward the recipient depends on the specific URI scheme, protocol type, or service logic. When no specific mechanism applies, a common way is to use the DNS system for resolving a part of the given URI representing a fully qualified domain name (FQDN), if present. For example, an HTTP request to *http://www.wonderland.net:8080/download/* is processed by an HTTP client agent (e.g. a web browser) by taking the hostport part (*www.wonderland.net:8080*) and resolving the hostname *www.wonderland.net* with a standard DNS query to the default DNS server. Similarly, when calling a user identified by the URI *sip:alice@wonderland.net;user=phone*, the SIP UACs often use DNS for resolving the FQDN *wonderland.net* to determinate the next hop SIP node through which the INVITE message has to be routed. However, such a DNS-based approach generally suffers of some problems like:

I1) it is server centric; by using standard DNS servers, it uses an architecture that is intrinsically distributed but not completely reliable, since an entire domain is normally managed by one or few servers;

I2) it does not allow dynamic name resolution; the association between the FQDN and the host address (IPv4 or IPv6) is statically configured on DNS servers; although updates of such binding are possible, the frequency of such updates is strongly limited by caching mechanisms implemented within the DNS architecture;

I3) it applies only to the FQDN part of the URI (when present) although some protocol-dependent information can also be provided within the URI.

Note that, although some protocols such as SIP provide their own location service in order to handle issue I2, the dependence on DNS still remains, resulting in issues I1 and I3. A possible solution to these issues could be to replace partially or completely the DNS dependence with a complete distributed and independent P2P location service (DLS), for example based on DHT. In such a scenario, whenever an agent wants to send a request to a resource or agent identified by a given URI, the following steps may be followed:

S1) the agent processes the URI and decides whether the URI already contains all information to route the message (for example because it includes a valid IP address and port number, or because it contains some explicit reference that can be locally and uniquely resolved to the proper IP address and port number), or requires an explicit DLS lookup; in the former case the procedure ends here, while in the latter case step S2 is executed;

S2) in order to obtain a fully routable address (URL) the agent performs a LS lookup by issuing a DLS get() query;

S3) the returned address (URL) is used as new reference as the next hop or recipient for the resource request.

This procedure can be extended to the case in which step S2 may return a new URI (or URL) not routable in the sense of S1. In this case, steps S1, S2, and S3 are repeated more times until step S2 returns a finally fully routable-URL. This simple DLS procedure can be further extended in order to provide dynamic name resolution. If an agent wants to update its location information or that of a given resource:

S4) the agent performs a LS insertion or update by issuing a DLS put() query with the proper new URI-to-routable-URL binding.

Since each binding has a time-to-live (expiration) date, such binding is refreshed through successive DLS put() operations. Step S4 can be used for inserting, refreshing, modifying, or deleting binding information.

3.1.1 URI information

The step S1 described above relies on the decision whether a URI is a pure resource identifier or it is a routable-URL, here defined as a URL that contains full information required to completely locate the resource. In order to overcome this ambiguity, two methods are hereafter considered:

R1) the URI is a routable-URL if it contains an IP address and port number in a proper hostport field; otherwise (for example if the hostport field contains a FQDN), an explicit LS query is needed;

R2) an explicit and proper new scheme or scheme parameter is used to distinguish a pure URI from a routable-URL.

Note that some already defined URI schemes may support parameters in different formats, may not support new parameters, or may not support parameters at all. Therefore, with solution R2, in case a proper URI parameter is defined to distinguish between a routable-URL and a generic URI, some hypothesis are required for the supported schemes. For this reason, with solution R2, the use of a new URI scheme is preferred. Examples according to R1 are:

- *http://192.168.1.2:8080/download/index.html*: it is a routable-URI and does not require an explicit LS lookup;
- *http://www.mydomain.com/download/index.html*: it is not routable and requires an explicit LS lookup.

Examples according to R2 are:

- *resource:http://www.mydomain.com/www/index.html*: it is a pure URI;
- *http://192.168.1.2:8080/download/index.html*: it is a routable-URI.

Note that these are only two possible approaches to solve step S1. Other mechanisms may be also defined and implemented. In the rest of this document we will refer to routable-URL or contact URI as a URL/URI that contains all the information to contact the identified resource (or to the next hop toward the resource) without the use of any external systems. According to this definition, the routable-URL returned by the iteration of steps S1, S2, and S3 (one or more times), eventually completed by some location information locally stored, should be resolved in one ore more IPv4 or IPv6 addresses together with complete protocol and port information. This is what should happen in the case of a real and complete P2P LS system.

3.1.2 The IETF P2PSIP Working Group

The Internet Engineering Task Force (IETF) is an open standards organization which develops and promotes Internet standards. The IETF is organized into a number of different working groups, each of them dealing with a specific topic, even though some topics require interactions among working groups. The Peer-to-Peer Session Initiation Protocol Working Group (P2PSIP WG[1]) is chartered to develop protocols and mechanisms for the use of the Session Initiation Protocol (SIP) in settings where the service of establishing and managing sessions is principally handled by a collection of intelligent endpoints, rather than centralized servers as in SIP as currently deployed. Since its establishment, the P2PSIP WG has been working to create a concept and terminology document, to define a P2PSIP Peer protocol and an optional P2PSIP Client protocol, and finally to produce a usage document. The work of the P2PSIP WG has been very productive, yielding hundreds of *Internet-drafts* and is currently in its final stage, with the goal of submitting final documents to the Internet Engineering Steering Group (IESG) between December 2010 and May 2011. During its lifetime, the P2PSIP WG has defined several proposals for a P2PSIP Peer protocol. The most notable of such proposals are dSIP [14] and RELOAD [15] which will be discussed in the following sections.

3.1.3 Distributed SIP Location Service

The SIP protocol is an application-level signaling protocol defined by the IETF RFC 3261 [10] and used to establish multimedia sessions. SIP is defined as a P2P protocol, in the sense that, once a session has been established, the multimedia stream flows among the participants directly. Moreover, some SIP scenarios, such as a “SIP P2P call”, require nothing but User Agents: in this case, the caller initiates a session by knowing the callee’s location (i.e. its IP address and port number). However, since this information is usually not known in advance, in order to be useful and practical for a public service, SIP relies on some network elements, such as Registrar servers or Proxy servers, that introduce some degree of centralization and possible failure points

[1] `http://datatracker.ietf.org/wg/p2psip/charter/`

of the architecture. SIP has been investigated for P2P capabilities in order to propose a version of SIP that does not require central servers. One possible solution to this problem has been found in realizing a DLS, in order to remove the need for central servers to resolve the contacts' addresses to route requests between User Agents. Because of their nature, DHTs have revealed as a perfect tool to accomplish this goal, as they can be used to store the user registration information (name address and location). The distributed SIP LS described above is one example of how a DHT can be used to create a LS for a given application. Other applications might exploit a DLS as well. We propose that these location services be merged into a unique DHT in order to achieve a single DLS that several application might use. This approach is preferred rather than creating separate location services for each application since:

- only one registry would be used as a single distributed access point;
- the increased number of collaborating nodes would result in a more robust DHT.

3.2 DLS Architecture

The P2P DLS system [16] should provide a storage and retrieval service for the binding between a URI, identifying the targeted resource, and one or more mapped contact URIs, which indicate the location where or through which the resource can be accessed. Together with each contact URI some other information should be stored like the expiration time, an access priority value, and, optionally, a displayable text (for example a description of the contact or a readable name). The distributed LS can be abstractly represented as in table 3.1.
The proposed P2P distributed LS may actually store and retrieve mappings between a URI (identifying the resource) and one or more URIs in a distributed and reliable manner. RFC 2397 [17] defines a method to encapsulate any (short) data within a standard URI. Our LS system in conjunction with RFC 2397 may also be seen as a system for storing any kind of short data in a distributed P2P manner, providing a sort of distributed database. It is important to point out that, although a real dis-

Table 3.1: Abstract DLS table representation

Key	**Value**
resource-URI-1	contact-URI-1; display-name-1; priority=P1; expires=T1 contact-URI-2; display-name-2; priority=P2; expires=T2 contact-URI-3; display-name-3; priority=P3; expires=T3
resource-URI-2	contact-URI-4; display-name-4; priority=P4; expires=T4
resource-URI-3	contact-URI-5; display-name-5; priority=P5; expires=T5 contact-URI-6; display-name-6; priority=P6; expires=T6
...	...

tributed database would require that the information stored in the DHT should be the actual data (such as files), the data stored in the DHT should be short as they would be moved often from node to node as the DHT reorganizes when nodes join and leave. This is why we prefer using the DHT as a location service, that is, a registry, where the information stored is not the actual data but rather show how to access that particular resource. This approach is also preferred as it is up to the application that looks up the DHT to decide what to do with the location information and what protocol to eventually use to actually access the resource. Therefore, applications may treat the DHT as an external registry to consult whenever they need location information. In order to implement a general purpose DLS we do not specify a particular DHT algorithm: different implementations may use different DHT algorithms (like Kademlia, Chord, etc.). The main components of such P2P distributed LS are:

C1) a DHT algorithm;

C2) a P2P protocol used for managing the DHT (inserting a new peer, updating the DHT, etc.);

C3) a protocol used to perform basic LS queries like put(), get() on the distributed LS, used by DHT-peer and possibly by non-DHT peers; since DHT peers use C2 for maintaining the DHT, protocol C3 is intended for pure DHT access at the border of the P2P system.

As pointed out, for component C1 we do not make any particular assumption since different DHT algorithms should be supported. For component C2 we do not make any assumption either, even though particular attention has been focused on SIP extensions defined by the IETF P2PSIP WG [14, 18, 19]. Finally, for component C3 we do not specify a new protocol. Any implementation can consider and use its own mechanism, according with the other systems it has to interact with. Some examples are described in the next sections. Note that protocol C2 is also a possible candidate for C3. Particularly, we have implemented a LS system in which SIP (actually with some extensions) is used for both C2 and C3 [19]. Note that, if the application includes the peer, C3 is not needed as the communication between the application and the peer occurs through basic API calls. In our realization both Kademlia and Chord DHT algorithm have been implemented and used for C1. Figure 3.1 shows how the DHT can be used by a generic application.

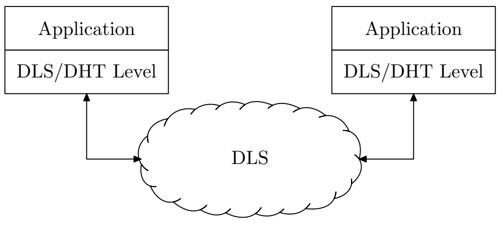


Figure 3.1: Applications accessing the DLS

3.2.1 Information stored into the DHT

The information stored into the DHT does not include only the contact of the resource (or service), that is, its routable-URI. Additional information such as:

- an optional display name, to be used as a description or a readable name,
- an expiration time, referring to the time for which the resource is to be considered fresh (this information can be also used to delete a resource from the DHT if set to 0),

- an access priority value

are also stored. This information can be included directly into a single URI, just like SIP specifies for contact information. Another approach could be to represent the resource information in XML format allowing also for adding additional parameters that might be considered useful for the resource.

3.3 DLS Layers

The DLS can be accessed through two simple API calls:

- *put(key,value)*
- *get(key)*

where *key* is a Resource URI (actually its hash), while *value* is a set of one or more tuples of display name, contact URI, expiration time, and priority value. The *get(key)* method should return the set of the corresponding values (actually the contact information) associated with the targeted resource. In a network-based application, the distributed LS could be implemented within a proper LS layer, as shown in Figure 3.2.

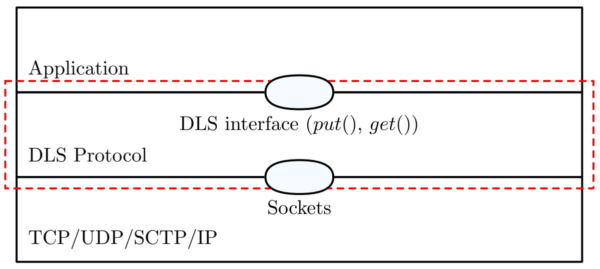


Figure 3.2: Location Service Layer

The DLS protocol layer includes all the mechanisms and functions to access the rest of the DLS system implemented on the other nodes, according to the proper P2P system. Considering a DHT-based P2P DLS infrastructure in which each node (actually a peer) cooperates to the maintainance of the DLS and underlying DHT, the previous architecture can be particularized as shown in Figure 3.3. In such architecture, the DLS protocol layer is composed of three sub-layers:

- DLS Layer;
- Peer Layer;
- RPC Protocol Layer.

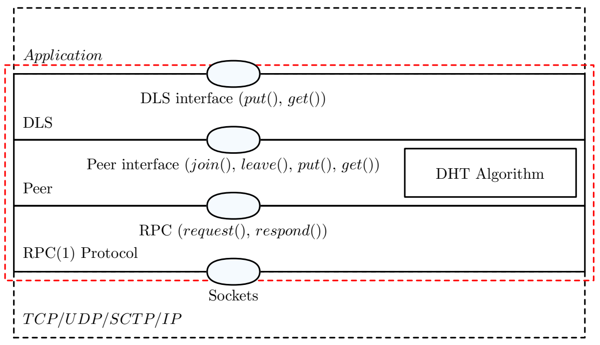

Application
DLS interface (*put*(), *get*())
DLS
Peer interface (*join*(), *leave*(), *put*(), *get*())
DHT Algorithm
Peer
RPC (*request*(), *respond*())
RPC(1) Protocol
Sockets
TCP/UDP/SCTP/IP

Figure 3.3: DLS Peer layered architecture

3.3.1 DLS Layer

The DLS Layer provides the basic LS service to the application layer. It mainly maps the *get()* and *put()* LS methods in the corresponding methods provided by the Peer Layer, which actually implements the specific DHT algorithm (Kademlia, Chord, etc.). All P2P-specific functions, such as *join()* and *leave()* and peer identification, are transparent for the application layer and are masqueraded by the DLS Layer.

3.3.2 Peer Layer

The Peer Layer is responsible for dynamic setup and maintenance of the DHT infrastructure, interacting with other peers, according to the specific DHT algorithm that is being implemented. It completely masquerades to the DLS Layer all the details about the adopted DHT algorithm by offering a transparent and uniform interface. As a result, it offers to the DLS Layer only basic operations (which we call *DHT API*), which are common to all DHT algorithms:

- *join()*: this operation is used to let the peer join the overlay;
- *leave()*: this operation is used to let the peer leave the overlay it is currently enrolled in, gracefully;
- *put(key,value)*: this operation is used to store a key/value pair in the DHT;
- *get(key)*: this operation is used to retrieve the information associated with the given key.

On the other side, the actual peer remote calls depend on the chosen DHT algorithm and are mapped on the underlying RPC protocol.

DHT Algorithm

The DHT Algorithm is the actual logic implemented by the peer and used to store and retrieve dynamic mappings between keys and values in a distributed fashion. At this level, the keys are the hashed Resource URIs, while the mapped values are a set of tuples containing the resource contact information. Note that, according to RFC 2397 [17], which defines a method for mapping any (short) data within a standard URI, the contact URI information may be used to encapsulate short data in place of or in addition to the actual resource contact URI. This in turn allows the DHT (and therefore the DLS) to be used as a generic system for storing any kind of short data in distributed P2P manner, thus providing a sort of distributed database.

3.3.3 RPC Protocol Layer

The DHT-based P2P system requires that all peers enrolled in the DHT overlay network exchange information for the DHT setup, update, and maintenance. The interaction between peers occurs through a request/response model, whose details depend on the specific DHT algorithm implemented by the Peer Layer. The mapping between the DHT algorithm logic and the actual communication protocol is provided by the RPC Protocol Layer. Hence this layer is responsible for transforming the Peer Layer's remote DHT methods to proper request/response communication messages. The RPC protocol may use an underlying transport such as TCP, UDP, SCTP, TLS or DTLS, depending on the type of the used RPC protocol (reliable/unreliable, message/stream oriented, etc.) and on the desired security level. On the receiver side, this layer is responsible for receiving messages from other peers, parsing them, and trigger the Peer Layer to execute the appropriate DHT algorithm logic.

3.4 DHT-unaware clients and peer adapters

According to the description of the DLS Layers above, a peer cooperates to maintain the DHT and the DLS system and provides an interface to the upper level application for accessing the DLS through the DLS interface at the same time. However, it could be also interesting to consider other application scenarios in which a node needs to access the LS service but for some reason (i.e., because it is not aware of the underlying P2P substrate, or it does not have enough resources to take part in the overlay) it does not belong to the DLS system. As a result, the architecture is decoupled between nodes (DLS peers) that are aware of the P2P substrate and nodes (DLS clients) that are not. The architecture of a DLS client is shown in Figure 3.4.
In a DLS client, the DLS layer still offers to the upper application layer the basic Location Service operations (*put()* and *get()*). However, differently from what happens within a DLS peer, the DLS layer does not interact directly with the DLS system, but it maps these operations to proper RPC calls that will be sent to a remote DLS server node. In general, such RPC protocol could be different from the one used by the Peer Layer within the DLS system, and for this reason it is here referred to as RPC(2), as shown in Figure 3.4. In order to effectively allow a DLS client to access the DLS system, a **DLS adapter peer** is required. A DLS adapter peer is a regular peer that participates in the DLS system, but it also acts as a DLS server, which means that it adds a sort of relay function that allows DLS client requests to be relayed to the

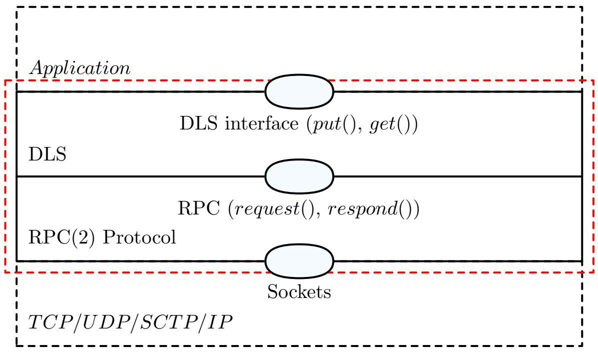


Figure 3.4: DLS Client layered architecture

P2P DLS system. The overall architecture of a DLS client and a DLS adapter peer is shown in Figure 3.5.

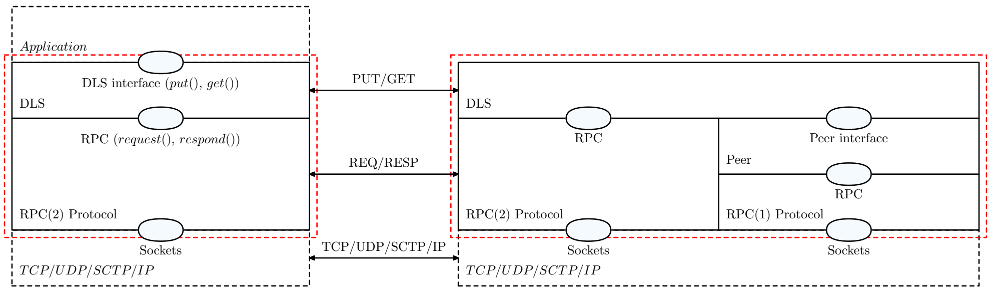


Figure 3.5: DLS Client (left) with DLS Adapter Peer (right)

Note that protocols RPC(1) and RPC(2) may or may not be the same, as this is just an implementation issue. It is important to remark that the proposed architecture is totally generic and independent from the DHT algorithm and RPC protocol used. Indeed, a DLS system instance is specified by the following three components (as previously assumed in section 3.2):

- a DHT algorithm;
- a RPC(1) protocol used for managing and maintaining the DHT;

- a RPC(2) protocol used by DLS clients at the border of the P2P infrastructure to access the DLS system.

3.5 IETF P2PSIP WG Proposals

3.5.1 dSIP

A mature proposal for a SIP-based P2P protocol was dSIP [14], which is a very simple extension of the SIP protocol with a few new headers added in order to maintain and manage the DHT. dSIP messages are based on the SIP REGISTER method; this choice was due to the fact that SIP REGISTER messages are intended to be processed only by those network elements, such as SIP Proxies, that provide a centralization point and that can decide whether to process the message or not by checking if they support the P2P capability. Depending on the implementation, peers can act either as proxy servers or redirect servers. Clients that are not aware of the P2P substrate can interact with their peer with the SIP protocol, thus allowing for backward compatibility towards legacy SIP applications. In this case, the peer would analyze the request (i.e. a normal SIP INVITE request for another client) from the client and retrieve the necessary information from the DHT; then the peer would forward the request to the appropriate endpoint or send a response back to the client. This role of the peer is called “adapter”. An adapter peer provides therefore a sort of gateway between SIP and P2PSIP. The peer thus has two main communication interfaces. The first one is the DHT communication interface, which is used to communicate with other peers in the DHT. Communication inside the DHT occurs using the dSIP protocol. dSIP messages are therefore received and sent at this interface. The second interface is what we call the SIP adapter, which is the interface responsible to provide the adapter functionality to the peer. The SIP adapter receives and sends regular SIP messages from and to clients. Other solutions for the P2PSIP architecture protocol which are not based on SIP are RELOAD [15] and XPP [20].

3.5.2 RELOAD

Internet-draft *draft-ietf-p2psip-base-12* [2] defines RELOAD as follows: REsource LOcation And Discovery (RELOAD), a peer-to-peer (P2P) signaling protocol for use on the Internet. It provides a generic, self-organizing overlay network service, allowing nodes to efficiently route messages to other nodes and to efficiently store and retrieve data in the overlay. RELOAD provides several features that are critical for a successful P2P protocol for the Internet:

- Security Framework: A P2P network will often be established among a set of peers that do not trust each other. RELOAD leverages a central enrollment server to provide credentials for each peer which can then be used to authenticate each operation. This greatly reduces the possible attack surface.
- Usage Model: RELOAD is designed to support a variety of applications, including P2P multimedia communications with the Session Initiation Protocol [I-D.ietf-p2psip-sip]. RELOAD allows the definition of new application usages, each of which can define its own data types, along with the rules for their use. This allows RELOAD to be used with new applications through a simple documentation process that supplies the details for each application.
- NAT Traversal: RELOAD is designed to function in environments where many if not most of the nodes are behind NATs or firewalls. Operations for NAT traversal are part of the base design, including using ICE to establish new RELOAD or application protocol connections.
- High Performance Routing: The very nature of overlay algorithms introduces a requirement that peers participating in the P2P network route requests on behalf of other peers in the network. This introduces a load on those other peers, in the form of bandwidth and processing power. RELOAD has been defined with a simple, lightweight forwarding header, thus minimizing the amount of effort required by intermediate peers.

[2]`http://tools.ietf.org/html/draft-ietf-p2psip-base-12.txt`

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

RELOAD is a binary protocol (contrast this with dSIP, which is a text-based protocol) that integrates the ICE (Internet Connectivity Establishment) protocol [21], which is intended to allow NAT traversal, a primary issue in the current Internet, as most clients reside in networks behind network devices, such as NATs and Firewalls. RELOAD is the official candidate of the P2PSIP Working Group to become an Internet standard.

RELOAD Architecture

RELOAD defines a layered architecture aimed to recreate on top of the Internet model's transport layer an overlay network which defines its own network, transport, and application layers. These equivalent layers are implemented in several components, as shown in Figure 3.6. The major components of the RELOAD architecture, from the upper layer to the lower layer, are:

- **Usage Layer**: this layer implements application-specific usages of the lower Message Transport; Usages define their own set of data types and behaviors that describe how to use the services provided by RELOAD.

- **Message Transport**: this component is used to handle end-to-end reliability, manage request state for usages, interacts with the Storage component to store and fetch resources, and delivers response messages to the component that initiates the request.

- **Storage**: this component processes messages relating to the storage and retrieval of resources; it talks to the Topology Plugin component in order to

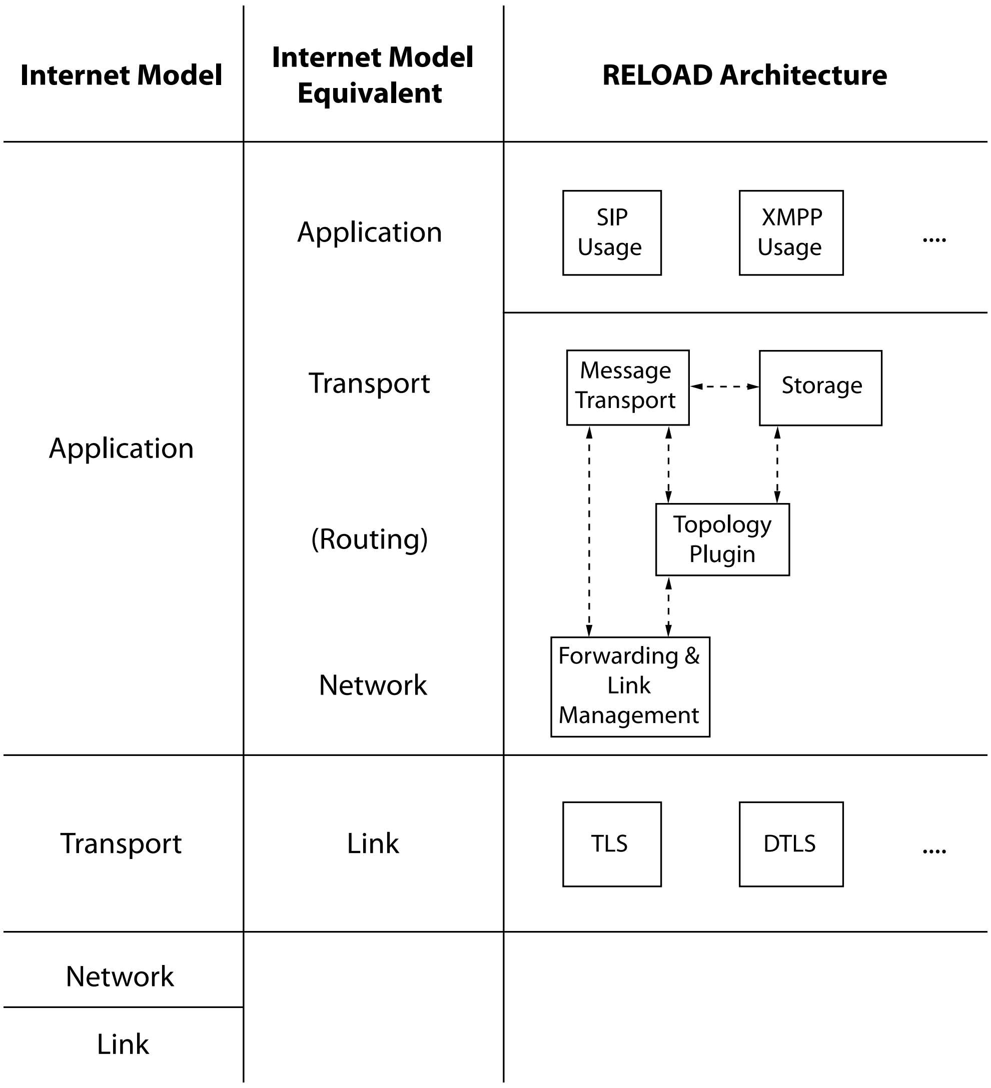


Figure 3.6: RELOAD architecture

manage data replication; it uses the Message Transport to send and receive resource-related messages.

- **Topology Plugin**: this component implements the actual logic of the overlay algorithm in use; it talks to the Message Transport component to send and receive messages for overlay management; it talks to the Storage component to manage data replication; it talks to the Forwarding Layer to control message forwarding.
- **Forwarding and Link Management**: this component stores and implements the routing table by providing packet forwarding services between nodes. It also handles establishing new links between nodes, including setting up connections across NATs using ICE.
- **Overlay Link Layer**: TLS/TCP [22] and DTLS/UDP [23] are used in RELOAD as "link-layer" for hop-by-hop communication.

It is possible to see some analogies between the components of the DLS architecture and those of the RELOAD architecture:

- the RPC Layer of the DLS resembles the functionalities of the Forwarding and Link Management Layer in RELOAD;
- the Peer Layer of the DLS resembles the functionalities of the Topology Plugin and Storage components in RELOAD;
- the DLS Layer of the DLS resembles the functionalities of the Usage Layer in RELOAD.

The Distributed Location Service is very similar to the RELOAD architecture, which is the official candidate to become an Internet standard. One difference between the two is that RELOAD natively addresses issues such as NAT/Firewall traversal, while the DLS needs to rely on external services that allow to establish connections among nodes that are behind NATs and Firewalls. Another difference, perhaps the major one, is the fact that RELOAD takes care of routing requests from the initiator to the target, while the DLS is used to lookup access information of the target, but the connection is then managed outside the DLS system.

Chapter 4

DLS Framework Implementation

The DLS Architecture described in Chapter 3 has been implemented and used as a basis to develop several distributed applications, which will be described in Chapter 5. The implementation is based on the Java programming language and provides a framework to create instances of Distributed Location Services based on different DHT algorithms and RPC protocols. The current implementation supports the Kademlia DHT algorithm and the dSIP protocol, but the framework is totally extensible to support other DHT algorithms and RPC protocols. The dSIP protocol, which is an extension of the SIP protocol defined in RFC 3261, has been implemented using the MjSIP library[1], which is a complete Java-based implementation of a SIP stack. MjSIP provides at the same time the API and implementation bound together into the MjSIP packages. MjSIP was developed within the Department of Information Engineering of the University of Parma and is available open source under the terms of the GNU GPL license (General Public License) as published by the Free Software Foundation.

The extensive use of design patterns in the implementation of the DLS Framework reflects the fact that no assumption was made on the DHT algorithm and RPC protocol that will be used for communication among the DLS nodes. The DLS framework can therefore be extended and particularized as needed in a clean and simple way,

[1]`http://www.mjsip.org`

while providing a consistent set of interfaces to any layer of the DLS architecture.

4.1 RPC Layer

The lowest layer of a DLS Peer is the RPC Layer. The RPC Layer is implemented in the *it.unipr.profiles.communicator* package. This layer is responsible for allowing the communication of the Peer Layer with other nodes. This layer maps the Peer Layer's DHT methods (*join()*, *leave()*, *get()*, and *put()*) into proper RPC protocol messages that will be sent to other nodes. The communication model adopted is a standard request/response model. This means that the DHT operations are performed as follows:

1. the Peer Layer calls a DHT method;
2. the method is mapped by the RPC Layer into a RPC protocol request message;
3. the RPC Layer sends the request to a node and waits for its response;
4. the response is received by the RPC Layer and parsed;
5. the response is forwarded to the Peer Layer which then subsequently takes the proper action as specified by the DHT algorithm logic.

The communication among nodes occurs asynchronously. This means that the RPC Layer's message forwarding procedure terminates after the message has been sent. When a response is received, the RPC Layer will then match the response message to the appropriate request message (*transactionality*) and notify the Peer Layer of the reception of the response. It is possible to realize a synchronous model by implementing a blocking behavior for the requests, even though this behavior is to be implemented on higher layers.
The package basically consists of three interfaces and two classes. The *IDHTCommunicator* interface defines the methods that are used to send request and response messages:

- void request(DHTRequest request, DHTCommunicatorListener listener)

- void respond(DHTRequest request, DHTResponse response)

The *DHTCommunicatorListener* interface defines the callback methods that are fired when messages are received:

- void onDHTRequestReceived(DHTRequest req)
- void onDHTResponseReceived(DHTRequest req, DHTResponse resp)
- void onDHTMessageSent(DHTMessage msg)

The *IDHTCommunicatorImpl* interface resembles the same methods of the *IDHT-Communicator* interface, but is used as a basis to implement the specific behavior of a particular communication protocol (Strategy Pattern). The *IDHTCommunicator* interface is implemented by the *DHTCommunicator* class, which is where the actual implementation of the *request()* and *respond()* methods resides. The *DHTCommunicator* class uses a Strategy Pattern for the implementation of the above methods: this class has a *DHTCommunicatorImpl* attribute, whose *request()* and *respond()* methods are called. This attribute is an object that implements the *IDHTCommunicatorImpl* interface and performs the actual sending of messages in its *request()* and *respond()* methods. The advantage of using of a *DHTCommunicatorImpl* instance makes it possible to have a clean and uniform interface at the upper layer to perform communication, with no reference to the actual RPC protocol in use. In order to implement communication using a particular RPC protocol, the only thing to do is just to define a class which extends the abstract *DHTCommunicatorImpl* class. This new class will then be used to set the *DHTCommunicatorImpl* class's *DHT-Communicator* implementation attribute. The *DHTCommunicatorListener* interface is implemented in the upper Peer Layer. Figure 4.1 shows the UML diagram for the *it.unipr.profiles.communicator* package. The RPC Layer is responsible for both sending and receiving messages to and from other peers and therefore the implementation of a RPC protocol needs to take care of the marshalling and unmarshalling operations. This is needed in order to allow the upper Peer Layer to work with neutral objects, independent from the RPC protocol (*DHTMessage*, *DHTRequest*, and

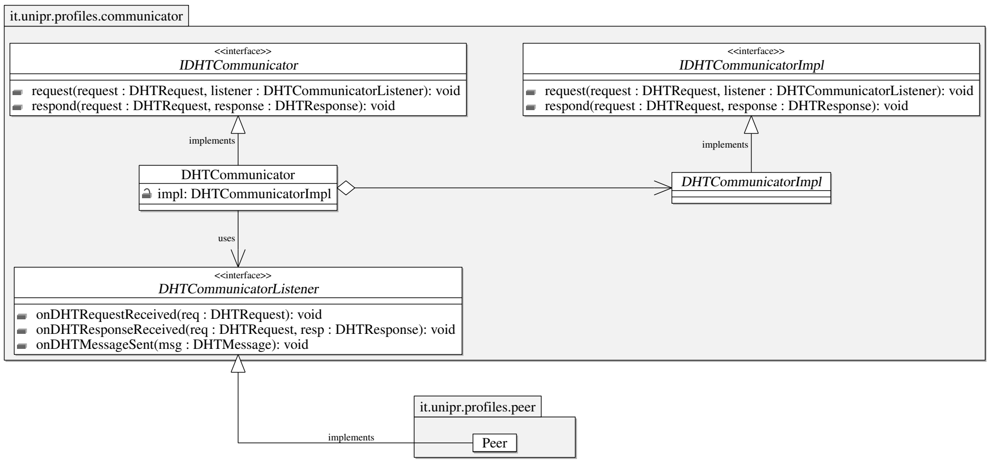

it.unipr.profiles.communicator
<<interface>>
IDHTCommunicator
request(request : DHTRequest, listener : DHTCommunicatorListener): void
respond(request : DHTRequest, response : DHTResponse): void
<<interface>>
IDHTCommunicatorImpl
request(request : DHTRequest, listener : DHTCommunicatorListener): void
respond(request : DHTRequest, response : DHTResponse): void
implements
implements
DHTCommunicator
impl: DHTCommunicatorImpl
DHTCommunicatorImpl
uses
<<interface>>
DHTCommunicatorListener
onDHTRequestReceived(req : DHTRequest): void
onDHTResponseReceived(req : DHTRequest, resp : DHTResponse): void
onDHTMessageSent(msg : DHTMessage): void
it.unipr.profiles.peer
implements
Peer

Figure 4.1: The it.unipr.profiles.communicator package (RPC Layer)

DHTResponse class objects, defined in the *it.unipr.profiles.message* package and sub-packages). The current implementation of the DLS Framework includes full support for the dSIP protocol described in the previous chapter. The dSIP protocol is implemented in the *it.unipr.profiles.dsip* package. Based on the interfaces defined in the *it.unipr.profiles.communicator* package, the *DSIPCommunicator* class, extending the *DHTCommunicatorImpl* class, was created. In order to support the marshalling and unmarshalling of dSIP messages, two additional classes were also created:

- *DSIPMessageFactory*: this class follows the Factory Pattern and provides static methods to create dSIP messages from *DHTMessage* objects (marshalling);

- *DSIPMessageParser*: this class performs the inverse operation by parsing incoming dSIP messages and creating appropriate *DHTMessage* objects to be used at the upper Peer Layer (unmarshalling).

Figure 4.2 shows the contents of the *it.unipr.profiles.dsip* package and its relation with the *it.unipr.profiles.communicator* package.

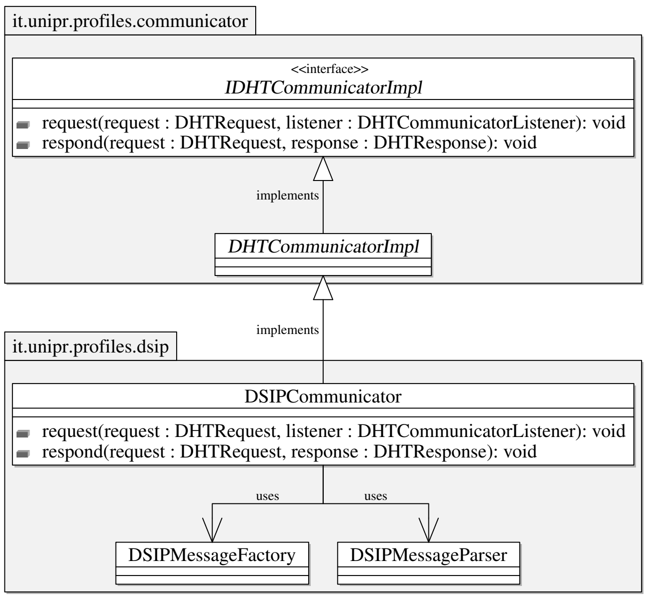

it.unipr.profiles.communicator
<<interface>>
IDHTCommunicatorImpl
request(request : DHTRequest, listener : DHTCommunicatorListener): void
respond(request : DHTRequest, response : DHTResponse): void
implements
DHTCommunicatorImpl
implements
it.unipr.profiles.dsip
DSIPCommunicator
request(request : DHTRequest, listener : DHTCommunicatorListener): void
respond(request : DHTRequest, response : DHTResponse): void
uses
uses
DSIPMessageFactory
DSIPMessageParser

Figure 4.2: The it.unipr.profiles.dSIP package

4.2 Peer Layer

The *it.unipr.profiles.peer* package holds all the interfaces and classes needed to implement to Peer Layer. The Peer Layer is located between the lower RPC Layer and the upper DLS Layer. The RPC Layer is transparent to the Peer Layer, which means that the Peer Layer has no knowledge of the details of the communication (i.e. the RPC protocol). The only knwoledge that Peer Layer has is the interface that the RPC Layer offers, that is the pair of *IDHTCommunicator* and *DHTCommunicatorListener* interfaces. The Peer Layer's responsibility is to manage the peer's participation in the DHT overlay. The Peer Layer is therefore responsible to actually let the peer join and leave the overlay, and perform storage-related operations such as storing and retrieving information to and from the DHT. The DHT algorithm logic resides at this level. The package consists of a set of interfaces and classes; the core of the Peer Layer is composed by the following:

- *IPeer* interface: it defines the methods that a peer must implement to actively participate in the DHT lifecycle; the methods are the ones defined in section 3.3.2 (*DHT API*);

- *IPeerImpl* interface: it defines all the methods that a particular DHT algorithm implementation must implement;

- *Peer* class: it is a concrete class which implements the *IPeer* interface; this class implements the basic behavior for these methods but leaves the details of the implementation to a class that implements the *IPeerImpl* interface (Strategy Pattern), using the same philosophy used for the *DHTCommunicatorImpl* class;

- *PeerImpl* class: it is an abstract class which implements the *IPeerImpl* interface; this class is the basis for implementing DHT algorithm behavior, which will be accessed by the *Peer* (Strategy Pattern);

- *DHTListener* interface: it defines the callback methods that are called when a DHT-related operation (the operations defined in the *Peer* interface) has completed; the *PeerImpl* class implements this interface as well;

Figure 4.3 shows the contents of the main components implemented in the *it.unipr.profiles.peer* package. Some methods do not report the exact signature for clarity reasons[2]. *Peer*s are created using a Factory Pattern. Instances are created by calling the *PeerFactory* class's *createPeer()* method. This approach is needed in order to have a unique interface for *Peer* creation while still allowing for new kinds of peer being implemented. The type of peer created depends on the supplied *PeerImpl* argument that is being passed. The model for *Peer* creation and the usage of such model to extend the framework with different DHT algorithm is shown in Figure 4.4. The Peer Layer is also responsible for the storage of the part of information for which it is responsible, according to the DHT algorithm. The *Peer* class also has a *ResourceMap* object which is where all the mappings between URIs and resource contacts are stored. The components relative to information storage are located in the *it.unipr.profiles.resource*

[2]N_ARGS in the argument list means that the method has one or more arguments but we do not report them, in order to have figures that are more comprehensible; this way we can still use the notation with no arguments without the risk of ambiguity.

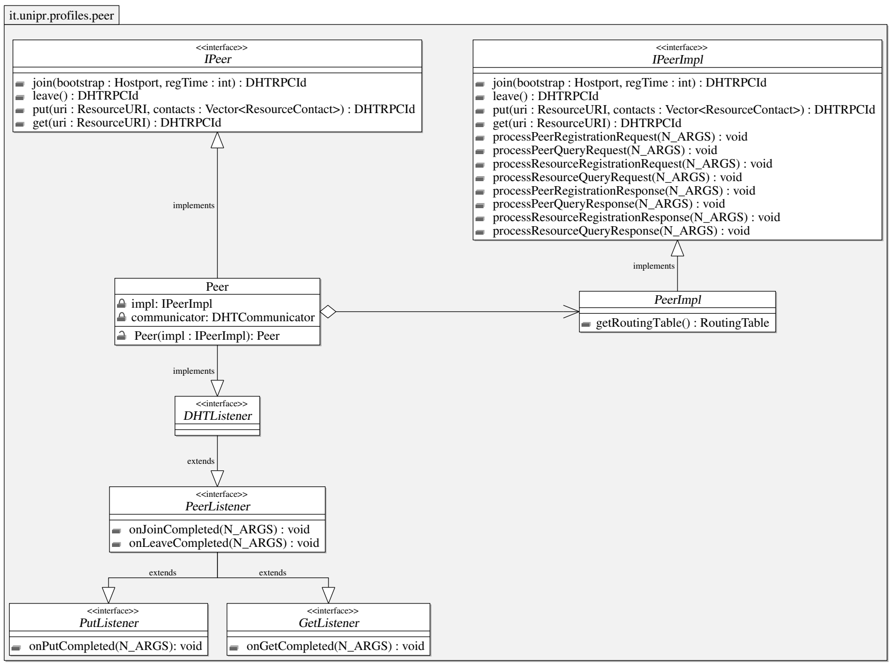

it.unipr.profiles.peer
<<interface>>
IPeer
join(bootstrap : Hostport, regTime : int) : DHTRPCId
leave() : DHTRPCId
put(uri : ResourceURI, contacts : Vector<ResourceContact>) : DHTRPCId
get(uri : ResourceURI) : DHTRPCId
<<interface>>
IPeerImpl
join(bootstrap : Hostport, regTime : int) : DHTRPCId
leave() : DHTRPCId
put(uri : ResourceURI, contacts : Vector<ResourceContact>) : DHTRPCId
get(uri : ResourceURI) : DHTRPCId
processPeerRegistrationRequest(N_ARGS) : void
processPeerQueryRequest(N_ARGS) : void
processResourceRegistrationRequest(N_ARGS) : void
processResourceQueryRequest(N_ARGS) : void
processPeerRegistrationResponse(N_ARGS) : void
processPeerQueryResponse(N_ARGS) : void
processResourceRegistrationResponse(N_ARGS) : void
processResourceQueryResponse(N_ARGS) : void
implements
implements
Peer
impl: IPeerImpl
communicator: DHTCommunicator
Peer(impl : IPeerImpl): Peer
PeerImpl
getRoutingTable() : RoutingTable
implements
<<interface>>
DHTListener
extends
<<interface>>
PeerListener
onJoinCompleted(N_ARGS) : void
onLeaveCompleted(N_ARGS) : void
extends
extends
<<interface>>
PutListener
onPutCompleted(N_ARGS): void
<<interface>>
GetListener
onGetCompleted(N_ARGS) : void

Figure 4.3: The it.unipr.profiles.peer package (Peer Layer)

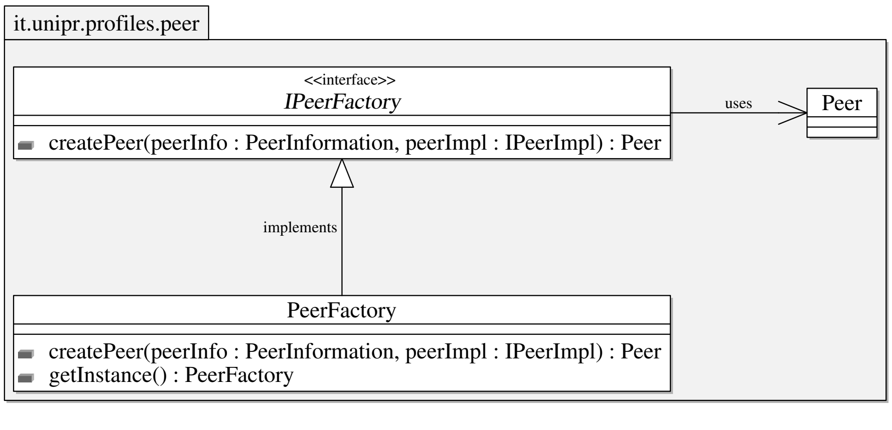

it.unipr.profiles.peer
<<interface>>
IPeerFactory
createPeer(peerInfo : PeerInformation, peerImpl : IPeerImpl) : Peer
uses
Peer
implements
PeerFactory
createPeer(peerInfo : PeerInformation, peerImpl : IPeerImpl) : Peer
getInstance() : PeerFactory

Figure 4.4: Peer creation through PeerFactory

package. The *ResourceMap* class offers methods to add, get, and remove mappings in behalf of the *Peer* class. Again, following a Strategy Pattern, the actual storage operation is not specified but is left to a *ResourceMapImpl* object. In fact, it might be convenient to have different storage policies (such as using a SQLite database or keeping all the resources in memory) depending on factors, such as the available memory of the device. A *ResourceMap* object is created by a *ResourceMapFactory*. The storage component of the *Peer* class is shown in Figure 4.5.

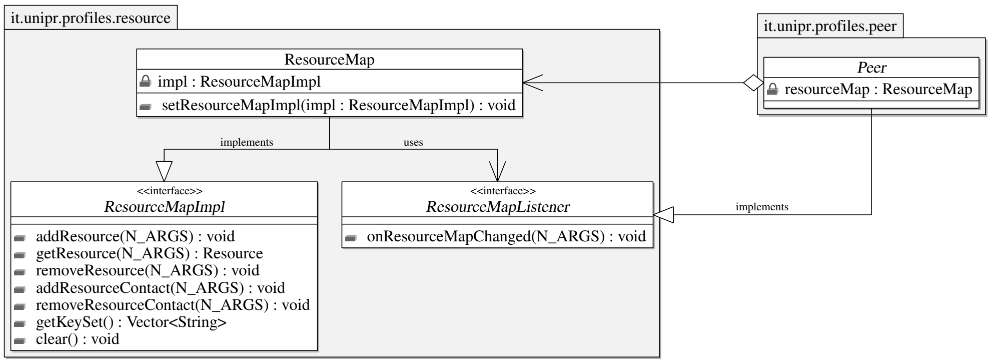

it.unipr.profiles.resource
ResourceMap
impl : ResourceMapImpl
setResourceMapImpl(impl : ResourceMapImpl) : void
implements
uses
<<interface>>
ResourceMapImpl
addResource(N_ARGS) : void
getResource(N_ARGS) : Resource
removeResource(N_ARGS) : void
addResourceContact(N_ARGS) : void
removeResourceContact(N_ARGS) : void
getKeySet() : Vector<String>
clear() : void
<<interface>>
ResourceMapListener
onResourceMapChanged(N_ARGS) : void
it.unipr.profiles.peer
Peer
resourceMap : ResourceMap
implements

Figure 4.5: Information storage

4.3 DLS Layer

The DLS Layer is the uppermost layer of the DLS. It is the access point for applications that need to access the location serivce provided by the DLS system. The Peer Layer hides all the details relative to the DHT algorithm and information storage policy implemented and offers to the DLS Layer only Location service-related functionalities, in the same way as the RPC Layer hides all the details of communcation to the Peer Layer.
The DLS Layer is implemented in the *it.unipr.profiles.dls* package. This package consists of two interfaces and an abstract class. The *IDLSApplication* defines the methods that can be called to interact with the underlying Peer Layer. This interface is very

similar to the *IPeer* interface defined in the *it.unipr.profiles.peer* package, but it differs from it since the methods return void. The *DLSApplication* class is the entry point for any application that uses the DLS system. It implements the *IDLSApplication* interface as this class has to deal directly with the Peer Layer and it also implements the *DHTListener* interface in order to be notified by the Peer Layer when DHT-related operation are executed. The *DLSApplication* class wraps an instance of *Peer* in order to forward requests to the Peer Layer. The *DLSApplication* class is based on the Delegation Pattern. The delegation pattern is a design pattern in object-oriented programming where an object, instead of performing one of its stated tasks, delegates that task to an associated helper object (*Inversion of Responsibility*). The helper object is called the delegate. Delegation is dependent upon dynamic binding, as it requires that a given method call can invoke different segments of code at runtime. In the DLS framework, the *DLSApplication* is the class that deals directly with the underlying Peer Layer to perform DHT-related operations. The *DLSApplicationDelegate* is an interface that defines a set of methods that the *DLSApplication* will fire at certain times during its execution. The delegate gets informed of the status of the operations that the *DLSApplication* is performing. The decoupling of these components makes it very easy to change easily the behavior of the application without the need to extend the application class. Subclassing should be used carefully as it introduces a tight coupling between the superclass and the subclass, which is considered bad practice in object-oriented development. The *DLSApplicationDelegate* is notified by the *DLSApplication* when a DHT-related operation will begin and when it ended its execution. The execution of these operations is therefore asynchronus. However, it is possible to realize a synchronous behavior at this level, as stated in section 4.1, by implementing a blocking behavior in the *DHTListener* methods of a *DLSApplication* subclass. Figure 4.6 shows the contents of the *it.unipr.profiles.dls* package.

4.4 Protocol Adapters

Protocol Adapters are the components that allow DLS Clients, which do not participate actively in the DLS system, to perform requests to the DLS. Protocol Adapters

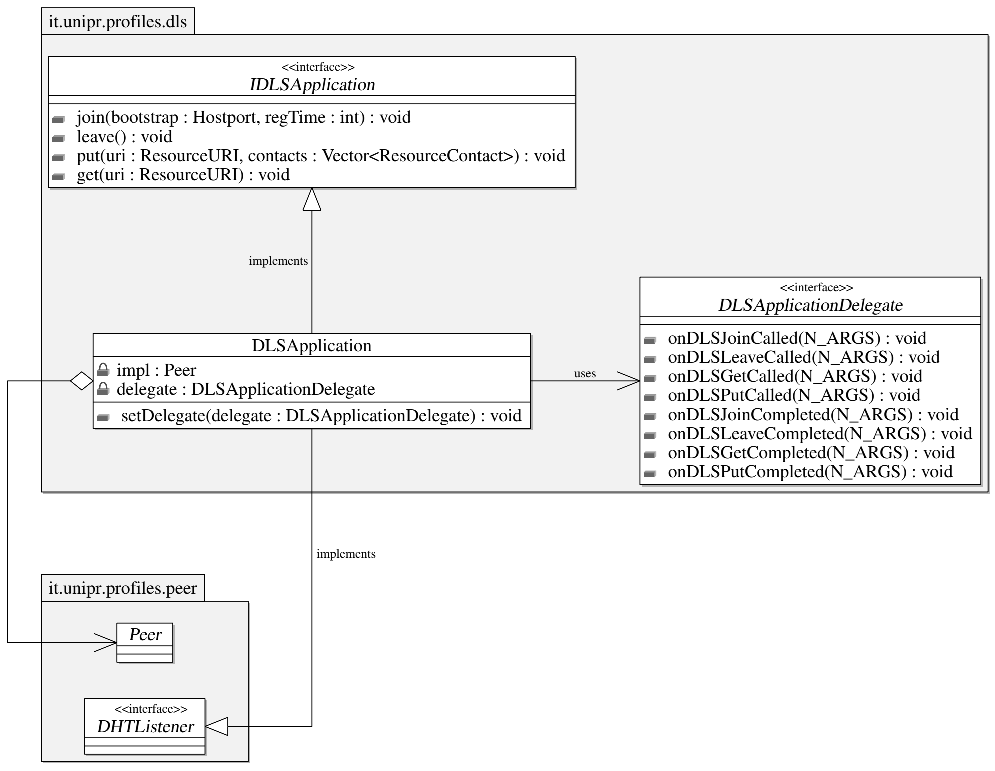

it.unipr.profiles.dls
<<interface>>
IDLSApplication
join(bootstrap : Hostport, regTime : int) : void
leave() : void
put(uri : ResourceURI, contacts : Vector<ResourceContact>) : void
get(uri : ResourceURI) : void
implements
<<interface>>
DLSApplicationDelegate
onDLSJoinCalled(N_ARGS) : void
onDLSLeaveCalled(N_ARGS) : void
onDLSGetCalled(N_ARGS) : void
onDLSPutCalled(N_ARGS) : void
onDLSJoinCompleted(N_ARGS) : void
onDLSLeaveCompleted(N_ARGS) : void
onDLSGetCompleted(N_ARGS) : void
onDLSPutCompleted(N_ARGS) : void
DLSApplication
impl : Peer
delegate : DLSApplicationDelegate
setDelegate(delegate : DLSApplicationDelegate) : void
uses
implements
it.unipr.profiles.peer
Peer
<<interface>>
DHTListener

Figure 4.6: The it.unipr.profiles.dls package (DLS Layer)

act like proxy servers to DLS Clients; when a DLS Client needs to access the DLS to store or retrieve information, the following steps are executed:

1. the DLS Client sends a request (using its own communication protocol, such as SIP or HTTP) to a DLS Peer's Protocol Adapter, which is set as a proxy server;
2. the Protocol Adapter receives the message, parses it, and determines the type of DLS request that must be performed;
3. the Protocol Adapter tells the peer to execute the appropriate request in the DLS;
4. when the peer has executed the request, it notifies the protocol adapter about the results of the request;
5. the Protocol Adapter then handles the results in the most appropriate way (i.e. forwards the original request to the targeted resource);
6. the Protocol Adapter replies to the DLS Client with the results of the original request.

The operations performed by the Protocol Adapter and peer are completely hidden to the DLS Client, which can therefore access the DLS system in a clean way, without any knowledge of the P2P-substrate. The transparency of these operations to the DLS Client allows any P2P-unaware application to exploit the DLS with no need to change their implementation.
Support for Protocol Adapters in the DLS Framework resides in the *it.unipr.profiles.adapter* package. The package defines the *ProtocolAdapter* class, which implements the behavior described previously. The *ProtocolAdapter* class implements the *GetListener* and *PutListener* interfaces defined in the *it.unipr.profiles.peer* package, since it directly deals with the DHT for information storage and retrieval. The *ProtocolAdapter* class includes a *ProtocolCommunicator* object. A *ProtocolCommunicator* is an object that acts as a server for incoming DLS Client requests. The *ProtocolCommunicator* is defined as abstract as it can only implement the general behavior needed

to interact with the *ProtocolAdapter*, but the actual implementation of the communication is left to subclasses. The *ProtocolAdapter* implements the *ProtocolCommunicatorListener* so that the *ProtocolCommunicator* can forward incoming Location Service requests to it. The other classes in the package are utility classes for internal operations.
Figure 4.7 shows the contents of the *it.unipr.profiles.adapter* package.

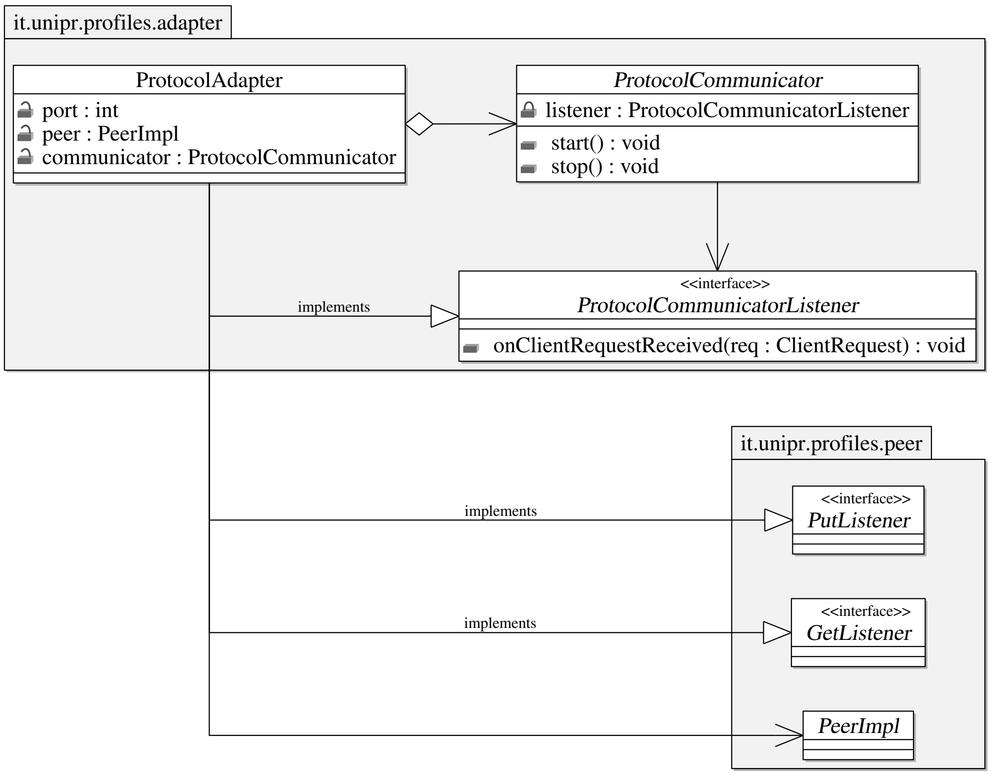


Figure 4.7: The it.unipr.profiles.adapter package

4.5 DLS Framework usage

The DLS framework can be easily used to create DLS-based applications. If there is no need to override the default behavior of the *DLSApplication* class, it is sufficient to instantiate a *DLSApplication* object and bind it to provided a *DLSApplicationDelegate* object. The *DLSApplication* object needs a *Peer* object to be created, which can be done through the *createPeer()* method of *PeerFactory*. Since the *DLSApplication* object deals with the lower layers, there is no need to care about the details of the DHT algorithm or communication. The *DLSApplication* uses the *Peer* it wraps to exploit the P2P substrate in order to store and retrieve information from the DLS. This way, the user can just focus on developing application-specific behavior and ignore the details of P2P communication and DHT algorithm.
The typical procedure to create a DLS application is the following:

1. create a *Peer* object by calling the *PeerFactory*'s *createPeer()* method and passing a new *PeerImpl* object in order to define the behavior of the peer;

2. set the implementation of the *ResourceMapFactory* class by calling the method *setResourceMapImpl()* and passing an instance of a *ResourceMapImpl*;

3. create a *ResourceMap* object by calling the method *createResourceMap()*;

4. set the *Peer* object's resource map to the *ResourceMap* object just created by calling *setResourceMap()*;

5. set the *Peer* object's communicator by calling *setDHTCommunicator()* and passing a new instance of a *DHTCommunicatorImpl* (i.e., a *DSIPCommunicator* object); at this point the *Peer* instance is ready to join the peer-to-peer overlay;

6. create a new instance of a *DLSApplication* (or one of its subclasses) by passing the *Peer* instance;

7. bind an instance of *DLSApplicationDelegate* to the *DLSApplication* instance;

8. start working with the *DLSApplication* object; the *DLSApplicationDelegate* instance will be notified when DLS-related events (i.e., when DLS RPCs are executed and when they complete), so that application-specific logic can stay separate from the DLS logic.

It is possible to add Protocol Adapters to the peer by calling the *addProtocolAdapter()* method on the *Peer* instance, if the peer is willing to act as adapter. Listing 4.1 shows the steps described above to create a DLS application. In this case, there is no need to override the basic behavior of the *DLSApplication* class; only a *DLSApplication-Delegate* class is created.

```
/* This class implements the DLSApplicationDelegate interface and shows a
    simple usage of the DLS framework.  */

package it.unipr.profiles.test;

import java.util.Vector;

import it.unipr.profiles.communicator.DHTCommunicatorImpl;
import it.unipr.profiles.communicator.DHTRPCId;
import it.unipr.profiles.dls.DLSApplication;
import it.unipr.profiles.dls.DLSApplicationDelegate;
import it.unipr.profiles.dsip.DSIPCommunicator;
import it.unipr.profiles.kademlia.KademliaPeerImpl;
import it.unipr.profiles.net.Hostport;
import it.unipr.profiles.peer.PeerFactory;
import it.unipr.profiles.peer.Peer;
import it.unipr.profiles.peer.PeerInformation;
import it.unipr.profiles.resource.ResourceContact;
import it.unipr.profiles.resource.ResourceMapFactory;
import it.unipr.profiles.resource.basic.BasicResourceMap;
import it.unipr.profiles.uri.GenericURI;
import it.unipr.profiles.util.SHA1Generator;

public class MyDLSAppDelegate implements DLSApplicationDelegate {

        /* Returns a DLSApplication instance whose peer runs on the specified
             hostport and belongs to the given overlay */
        private static DLSApplication createDLSApplication(Hostport hp,
            String overlay){
                String id = SHA1Generator.SHA1(hp.toString());
                PeerInformation pi = new PeerInformation(id, hp,
                    KademliaPeerImpl.algorithm, overlay, KademliaPeerImpl.
                    dht, 3600);
                Peer peer = PeerFactory.getInstance().createPeer(pi, new
                    KademliaPeerImpl());
                DHTCommunicatorImpl dSIPimpl = new DSIPCommunicator(peer);
                peer.setDHTCommunicatorImpl(dSIPimpl);
                ResourceMapFactory.getInstance().setResourceMapImpl(new
                    BasicResourceMap());
                peer.setResourceMap(ResourceMapFactory.getInstance().
                    createResourceMap());
                DLSApplication app = new DLSApplication(peer);
                return app;
        }

        /* Constructor */
        public MyDLSAppDelegate(){

        }

        /* Application main method */
        public static void main(String[] args){
                DLSApplicationDelegate delegate = new MyDLSAppDelegate();
                DLSApplication app = MyDLSAppDelegate.createDLSApplication(
                    new Hostport(args[0],Integer.parseInt(args[1]), args[2])
                    ;
                app.setDelegate(delegate);
                app.join(new Hostport(args[3], Integer.parseInt(args[4]),
                    3600);
        }

        /* DLSApplication callback: GET RPC was called */
        public void onDLSGetCalled(DLSApplication app, GenericURI target,
            DHTRPCId rpcId) {
                System.out.println("GET RPC called");
        }

        /* DLSApplication callback: GET RPC has completed */
        public void onDLSGetCompleted(DLSApplication app, boolean status,
            GenericURI target, Vector<ResourceContact> results, DHTRPCId
            rpcId) {
                System.out.println("GET RPC completed: " + status);
        }
        /* DLSApplication callback: JOIN RPC was called */
        public void onDLSJoinCalled(DLSApplication app, Hostport bootstrap,
            DHTRPCId rpcId) {
                System.out.println("JOIN RPC called");
        }

        /* DLSApplication callback: JOIN RPC has completed */
        public void onDLSJoinCompleted(DLSApplication app, boolean status,
            DHTRPCId rpcId) {
                System.out.println("JOIN RPC completed: "  + status);
        }

        /* DLSApplication callback: LEAVE RPC was called */
        public void onDLSLeaveCalled(DLSApplication app, DHTRPCId rpcId) {
                System.out.println("LEAVE RPC called");
        }

        /* DLSApplication callback: LEAVE RPC has completed */
        public void onDLSLeaveCompleted(DLSApplication app, boolean status,
            DHTRPCId rpcId) {
                System.out.println("LEAVE RPC completed");
        }

        /* DLSApplication callback: PUT RPC was called */
        public void onDLSPutCalled(DLSApplication app, GenericURI uri, Vector
            <ResourceContact> contacts, DHTRPCId rpcId) {
                System.out.println("PUT RPC called");
        }

        /* DLSApplication callback: PUT RPC has completed */
        public void onDLSPutCompleted(DLSApplication app, boolean status,
            GenericURI uri, Vector<ResourceContact> contacts, DHTRPCId rpcId
            ) {
                System.out.println("PUT RPC completed");
        }

}
```

Listing 4.1: Creating a DLS-based application

4.6 DLS Framework extension

The DLS framework has been defined and realized in order to be totally extensible, thus making it possible to implement different DHT algorithms, signaling protocols, resource map management policies, and protocol adapters. The extensive usage of design patterns in the implementation of the framework allows to implement extensions in a clean and simple way, without changing the usage patterns of the framework. It is therefore possible to think about all the components as pluggable modules that can be changed independently one from each other, in order to create custom DLS instances depending on the application's needs. The DLS framework has already been extended, by implementing the Kademlia DHT algorithm and the dSIP protocol in order to deploy and test it for real-world applications, but it may happen that for some applications other DHT algorithm might outperform Kademlia. In this section, the extension process is explained in detail.

4.6.1 RPC Protocols

As stated above, the dSIP signaling protocol has been implemented and is part of the DLS framework. Based on the framework architecture, it is possible to implement and integrate other protocols by creating a new *DHTCommunicatorImpl* subclass. In order to do so, it is necessary to implement the following methods:

- *request(DHTRequest request, DHTCommunicatorListener listener)*: this is the method that specifies how a DHT-related request should be marshaled, sent to the recipient of the request and which object should be notified when the response is received;

- *respond(DHTRequest request, DHTResponse response)*: this method specifies how a response (and its relative request) should be marshaled and sent back to the sender;

- *getSupportedProtocol()*: this method returns the supported communication *Protocol*;

- *getPeerURI(PeerContact pc)*: this method returns the *PeerURI* for a given *Peer-Contact*, that is, the universal identifier used by the protocol to identify the nodes;
- *halt()*: this method is used to stop the *DHTCommunicator*; messages will no longer be received or sent by the peer.

DHTCommunicatorImpl subclasses also need to provide server-like mechanisms in order to receive incoming requests. Typically, a *DHTCommunicatorImpl* listens for incoming requests on a port, which may be specified by the RPC protocol. When a request is received, the *DHTCommunicatorImpl* unmarshals the request message and passes the request object to the *Peer*, which will then process the request. Therefore, proper marshaling/unmarshaling methods must also be implemented.
The use of a standard interface for the *DHTCommunicator* hides all the details of the actual communication. The *Peer* class is not bound to a specific protocol, but delegates the responsibility of the communication to the *DHTCommunicator*.

4.6.2 DHT Algorithms

Similarly to RPC protocols, new DHT algorithms can be implemented and integrated in the framework for use within a DLS system. The DLS framework has built-in support for the Kademlia DHT algorithm. The process of extending the framework is extremely simple, as for the integration of new communication protocols, and it basically consists of creating a class that implements the desired behavior.
First of all, DHT logic resides in the *Peer* class. When adding support for a new DHT algorithm, a subclass of the *PeerImpl* class must be created. The methods that the subclass must implement are the following (as defined in the *IPeerImpl* interface):

- *join()*
- *leave()*
- *get()*
- *put()*

- *processPeerRegistrationRequest()*
- *processPeerQueryRequest()*
- *processResourceRegistrationRequest()*
- *processResourceQueryRequest()*
- *processPeerRegistrationResponse()*
- *processPeerQueryResponse()*
- *processResourceRegistrationResponse()*
- *processResourceQueryResponse()*

These methods provide the DHT logic that the peer must implement when performing requests and when processing incoming messages from other nodes in the DHT. Other methods can be overridden if needed to accomplish specific behaviors that are not implemented by default.
Figure 4.8 shows how to extend the DLS framework to support other DHT algorithms.

4.6.3 DLS Client Protocols

As previously described, the DLS framework can be used to allow access to the DLS service by nodes that are not part of the DLS/DHT platform. These nodes are called DLS clients. When a DLS client wants to access the DLS in order to store or retrieve information, it can do so through the mediation of a node (acting as a Proxy) that belongs to the DLS system, which is called a DLS adapter peer. The DLS adapter peer is a regular peer with some added functionalities, which are plugged as modules, called Protocol Adapters. Protocol Adapters are therefore proxy servers for some client protocol which can be used to store and retrieve information to and from the DLS. Creating a new Protocol Adapter is easy with the DLS framework, as the whole process can be reduced to the creation of a subclass of the *ProtocolAdapter* class. The

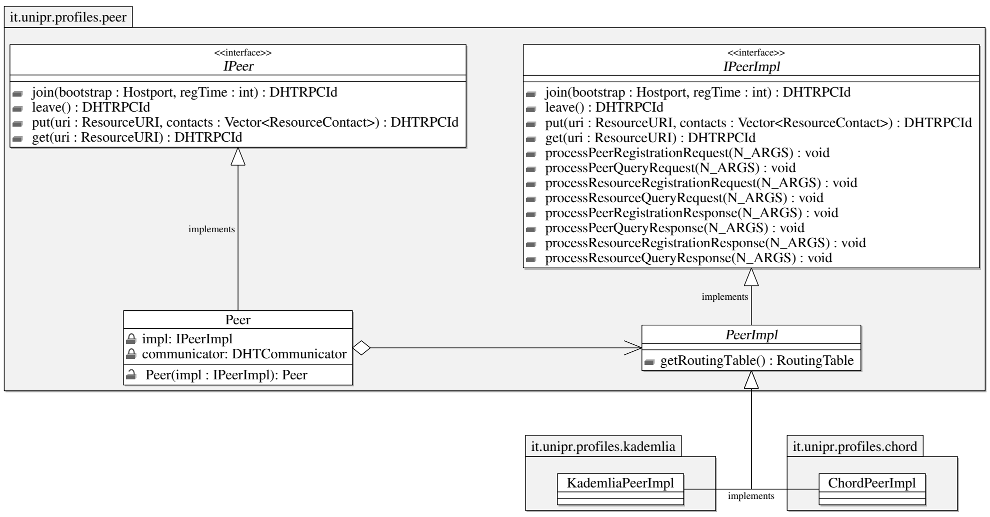

it.unipr.profiles.peer
<<interface>>
IPeer
join(bootstrap : Hostport, regTime : int) : DHTRPCId
leave() : DHTRPCId
put(uri : ResourceURI, contacts : Vector<ResourceContact>) : DHTRPCId
get(uri : ResourceURI) : DHTRPCId
<<interface>>
IPeerImpl
join(bootstrap : Hostport, regTime : int) : DHTRPCId
leave() : DHTRPCId
put(uri : ResourceURI, contacts : Vector<ResourceContact>) : DHTRPCId
get(uri : ResourceURI) : DHTRPCId
processPeerRegistrationRequest(N_ARGS) : void
processPeerQueryRequest(N_ARGS) : void
processResourceRegistrationRequest(N_ARGS) : void
processResourceQueryRequest(N_ARGS) : void
processPeerRegistrationResponse(N_ARGS) : void
processPeerQueryResponse(N_ARGS) : void
processResourceRegistrationResponse(N_ARGS) : void
processResourceQueryResponse(N_ARGS) : void
implements
implements
Peer
impl: IPeerImpl
communicator: DHTCommunicator
Peer(impl : IPeerImpl): Peer
PeerImpl
getRoutingTable() : RoutingTable
it.unipr.profiles.kademlia
KademliaPeerImpl
it.unipr.profiles.chord
ChordPeerImpl
implements

Figure 4.8: Extending the DLS framework to support other DHT algorithms

ProtocolAdapter class holds already all the details about the interaction with the DLS platform, so it is sufficient to implement a server-like behavior, that is:

- listen for incoming DLS client requests;
- parse incoming requests;
- forward the request in the DLS in behalf of the DLS client;
- send the response back to the DLS client.

The SIP and HTTP protocols are natively supported in the DLS framework by the *ProtocolAdapter* subclasses called *SIPAdapter* and *HttpAdapter*. These *ProtocolAdapter* classes have been used to implement some of the demonstrative DLS-based applications, which will be described in Chapter 5.

Chapter 5

DLS-based Peer-to-peer Applications

5.1 P2P VoIP

In section 3.1.3, DHTs were introduced to create a distributed SIP Location Service to implement a peer-to-peer VoIP platform.
The Session Initiation Protocol (SIP) [10] is the IETF standard signaling protocol defined for initiating, coordinating and tearing down any multimedia real-time communication session between two or more endpoints. Such endpoints are commonly referred to as SIP User Agents (UAs). According to SIP, in order to setup a multimedia session a caller UA sends an INVITE request to the callee UA, addressed by a SIP URI, which may identify either the callee or the actual contact IP address and port where the callee UA can currently be found. Since the actual contact address of the callee UA is usually not known in advance by the caller, currently implemented VoIP systems use the SIP URI to identify the callee. This mechanism requires a way to dynamically map a user URI to the actual contact address of one or more UAs where he can be reached. In the standard SIP architecture, this is achieved by SIP intermediate nodes (like Proxy or Redirect SIP servers) and by a proper registration mechanism through which SIP UAs update their contact addresses. This results into a

call scheme referred to as *SIP trapezoid* (Figure 5.1) and formed by the caller UA, an outbound proxy (optional), the destination proxy (which the callee is registered with), and the callee UA. Unfortunately such an architecture is server-centric and suffers of

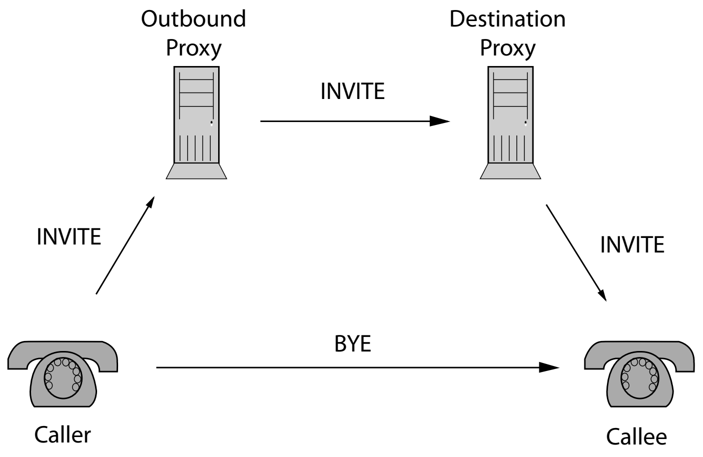


Figure 5.1: SIP trapezoid

well-known scalability and availability problems. In order to setup a session in a real P2P fashion, a fully distributed SIP architecture is needed.

5.1.1 P2P VoIP Architecture

When a caller UA wants to initiate a session with a callee UA, it needs a way to obtain the actual network address (IP address and port) of the callee. In the standard SIP architecture, this is usually achieved through the SIP trapezoid scheme. The goal of the architecture is basically to collapse the SIP trapezoid into a single line, connecting UAs directly. In order to create a fully distributed architecture for VoIP applications, SIP URI resolution can be provided by a peer-to-peer network, allowing for storing and lookup operations on SIP URIs. The most suitable peer-to-peer network type to do this is represented by the Distrubuted Location Service, described

in Chapter 3. According to our P2P VoIP architecture [24], the DLS stores mappings between a URI identifying the resource (the callee UA) and a set of contacts for the resource (where and how the UA is currently reachable). Such information includes the routable URL of the UA (containing IP address and port number), an optional human-readable display name, an expiration time, and an access priority value.
In order to register a UA's contact in the DLS, it is just needed to perform a *put()* RPC in the DLS, either directly (if the UA encapsulates a peer belonging to the DLS) or through an adapter peer (for instance, if the application is a legacy SIP UAs).
Session establishment requires some additional steps to be performed, which depend on whether the UA encapsulates a DLS peer or not. In the former case, the following steps are performed:

1. the caller UA, which encapsulates a DLS peer, perform a DLS *get()* RPC to retrieve the contact address of the callee;

2. when the address of the callee has been resolved, the caller UA sends an INVITE request directly to the callee UA and the session can be established.

An example of session setup between two SIP UAs wich encapsulate DLS peers is depicted in Figure 5.2. In case the UA does not encapsulate a DLS peer, it needs to set

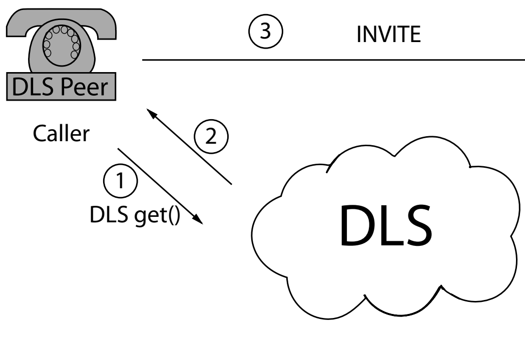


Figure 5.2: P2P VoIP session with P2P-aware SIP UAs

a DLS peer with a proper SIP Adapter module as its outbound proxy. In this scenario, the steps required to establish a session are the following:

1. the caller UA sends an INVITE request to its outbound proxy, that is, the DLS SIP adapter peer;
2. the SIP Adapter parses the request and performs a DLS *get()* RPC to resolve the callee's contact address;
3. when the address of the callee has been resolved, the SIP Adapter forwards the INVITE request to the callee UA and the session can be established.

An example of session setup between two SIP UAs which encapsulate DLS peers is depicted in Figure 5.3.

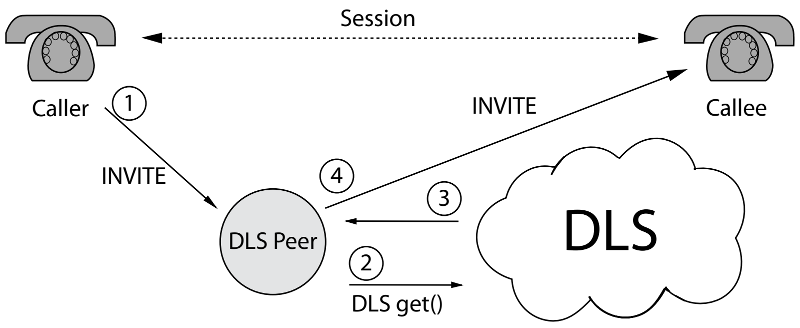


Figure 5.3: P2P VoIP session with P2P-unaware SIP UAs

5.2 Distributed Web Server

The *Distributed Web Server* (DWS) is a DLS-based application which implements a distributed HTTP server. When using a DWS, contents, such as files, are stored on a number of collaborating nodes rather than on a single host. The DWS uses the

DLS service in order to store and retrieve the mappings between resource URIs and their actual locations. The DWS allows for data replication, which can be achieved by storing the same information on different hosts. Moreover, contents can be partitioned among the collaborating nodes, for instance for security issues.

5.2.1 DWS Node Architecture

In the DWS system, nodes are applications composed of some modules:

- a DLS Application, which integrates a DLS peer, in order to interact with the DLS system; the DLS peer also has an HTTP Adapter module in order to be able to receive requests from P2P-unaware clients;
- an HTTP server to serve requests for resources;
- a *Resource Publish daemon* (RPD) in order to store and refresh the mappings of the resources hosted by the HTTP server;
- an optional *Resource Replication daemon* (RRD), which is used to force replication of the resources hosted by the HTTP server on other nodes.

The typical scenario for the Distributed Web Server is as follows:

1. an HTTP client (i.e. web browser) issues a request for a particular resource, hosted in the DWS; the request is sent to the HTTP Adapter of a DLS Peer that the HTTP client has set to be its proxy;
2. the HTTP Adapter module parses the incoming request and performs a DLS *get()* RPC for the targeted resource;
3. when the resource's address has been resolved, the HTTP Adapter relays the original request to the appropriate HTTP server, which belongs to a DWS node;
4. when the HTTP server receives the request from the DLS HTTP Adapter peer, it serves the request, and the response is relayed to the DLS HTTP Adapter peer and then to the original HTTP client.

Such a procedure, depicted in Figure 5.4, is totally transparent to the end user (i.e. a web browser), which only needs to set a valid DLS HTTP Adapter peer as its proxy. The proxy behavior can follow different policies. However, the suggested pol-

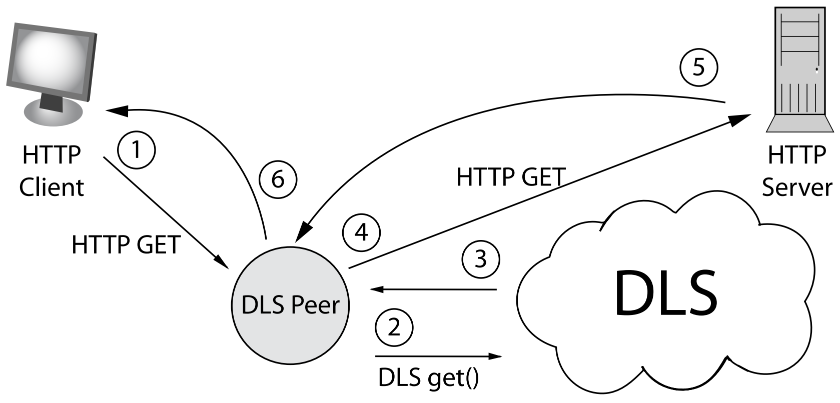


Figure 5.4: Distributed Web Server scenario

icy, which is the most general, is to try to resolve in the DLS any resource. If the resource cannot be resolved in the DLS, the proxy should act as a regular HTTP proxy server, using the DNS system to resolve the FQDN part of the resource URI. Once the resource has been resolved, either totally through the DLS or partially through the DNS system, the request is relayed and responses are sent back to the originator client through the DLS HTTP Adapter peer.

The *Resource Publish daemon* (RPD) is the module responsible for publishing the mappings between the resources hosted by the HTTP server and their actual location. The RPD periodically checks the resources hosted by the HTTP server and translates their path to a URI to be used as a key and a routable-URL. For each hosted resource, the key-to-value mapping is stored by the RPD through the DLS peer.

The *Resource Replication daemon* (RRD) is the module responsible for ensuring data replication in the overall system. Data replication might be needed in order to make resources available upon the failure of nodes. The RRD can follow different policies for data replication. Typically, the RRD periodically checks how many copies of a given resource are available in the DWS. This can be done by executing a DLS *get()* RPC for the given resource. If the number of available copies is less than a desired threshold, the RRD selects some nodes belonging to the DWS and sends an HTTP PUT request to their HTTP server, so that they can be instructed to store the given resource. The DWS that receives the resource will then make it available in the DWS system when its RPD will publish the mapping.

5.2.2 DWS Deployment Example

Let us suppose a Distributed Web Server for the domain *www.dws.org* is established among the nodes h_1, h_2, h_3, and h_4 and the resources distributed among the nodes as follows:

- $h_1 \rightarrow r_1, r_2, r_3, r_4$
- $h_2 \rightarrow r_1, r_3, r_5, r_7$
- $h_3 \rightarrow r_5, r_6, r_7, r_8$
- $h_4 \rightarrow r_2, r_4, r_6, r_8$

Resources are therefore assigned as follows:

- $r_1 \rightarrow h_1, h_2$
- $r_2 \rightarrow h_1, h_4$
- $r_3 \rightarrow h_1, h_2$
- $r_4 \rightarrow h_1, h_4$
- $r_5 \rightarrow h_2, h_3$
- $r_6 \rightarrow h_3, h_4$
- $r_7 \rightarrow h_2, h_3$

- $r_8 \to h_3, h_4$

Each node h_i has the following information associated:

- the HTTP server port $p_{s,i}$
- the HTTP adapter port $p_{a,i}$

The DLS contents for the DWS are depicted in table 5.1.

Table 5.1: DLS content for the DWS for the domain *www.dws.org*

Key	**Value**
http://*www.dws.org*/r_1	http://$h_1 : p_{s,1}/.../r_1$ http://$h_2 : p_{s,2}/.../r_1$
http://*www.dws.org*/r_2	http://$h_1 : p_{s,1}/.../r_2$ http://$h_4 : p_{s,4}/.../r_2$
http://*www.dws.org*/r_3	http://$h_1 : p_{s,1}/.../r_3$ http://$h_2 : p_{s,2}/.../r_3$
http://*www.dws.org*/r_4	http://$h_1 : p_{s,1}/.../r_4$ http://$h_4 : p_{s,4}/.../r_4$
http://*www.dws.org*/r_5	http://$h_2 : p_{s,2}/.../r_5$ http://$h_3 : p_{s,3}/.../r_5$
http://*www.dws.org*/r_6	http://$h_3 : p_{s,3}/.../r_6$ http://$h_4 : p_{s,4}/.../r_6$
http://*www.dws.org*/r_7	http://$h_2 : p_{s,2}/.../r_7$ http://$h_3 : p_{s,3}/.../r_7$
http://*www.dws.org*/r_8	http://$h_3 : p_{s,3}/.../r_8$ http://$h_4 : p_{s,4}/.../r_8$

Let us suppose that an HTTP client C wants to access the resource r_5. In order to do so, it sets its proxy to be the HTTP Adapter of any of the DWS nodes. C decides to use the HTTP adapter of node h_1 as its proxy. The message flow is the following:

- C sends an HTTP GET request for r_5 to the HTTP adapter of node h_1, which is reachable at http://$h_1 : p_{a,1}$ (i.e., GET r_5 HTTP/1.1);

- the HTTP adapter of node h_1 receives the request and performs a DLS lookup for r_5 on behalf of C;
- the lookup returns two possible locations for r_5 (nodes h_2 and h_3);
- the HTTP adapter of node h_1 sends a request for r_5 to the selected HTTP server (i.e., h_2, which is reachable at http://$h_2 : p_{s,1}$); the choice on which node to use might depend on some policy, such as using the access priority value;
- the HTTP server of node h_2 receives the request, serves it, and sends its response back to the HTTP adapter of node h_1;
- the HTTP adapter of node h_1 receives the response and relays it back to C.

The whole process of retrieving the resource is transparent to C, which doesn't need to know anything about the DWS infrastructure (i.e., h_1 belongs to the DWS), but only sees that node h_1 is acting as a regular HTTP proxy.
In the case of failure of node h_2, the resource would still be available on node h_3, so the request could still be fulfilled.

5.3 Distributed File System with HDFS

Dealing with huge amounts of data in data mining and user data collection applications, like Facebook[1] and Yahoo[2], has become more and more frequent. A solution to cope with this problem is to spread data over multiple network-connected physical devices, which increases system complexity and introduces additional potential points of failure. Moreover, despite the capacity of hard drives as massive storage systems has increased extremely during years, the speed at which data can be accessed has not. In order to address this problem, over the years, distributed file systems, such as Network File System (NFS), Hadoop Distributed File System (HDFS), Amazon Simple Storage Service (Amazon S3), and Google File System (GFS), have been designed and deployed. Such systems provide access to files stored on multiple hosts

[1] `http://www.facebook.com`
[2] `http://www.yahoo.com`

connected through a computer network transparently to users.
The peer-to-peer network paradigm has been introduced to overcome some limitations of the client-server architecture by adding features, such as scalability, fault-tolerance, and self-organization. In this section, a solution that integrates peer-to-peer network support to HDFS is presented, in order to realize a flexible, low-cost and, dynamic distributed file system [25].

5.3.1 HDFS Architecture

HDFS uses the concept of blocks of data. The block size is the minimum amount of data that the system can read and/or write. Stored data are broken into chunks according to the block size, which are then stored as independent units. Splitting data simplifies the storage management as every size computation is related to multiples of blocks. Blocks are also used as base units for data replication.
In an HDFS cluster, blocks are persistently stored into one or more workers, each one called *datanode*, whose job is to read and write data, and periodically send reports of which data they contain. The file system manager is another node, called *namenode*. The namenode manages the filesystem namespace, maintains its hierarchical tree and the metadata for files and directories in the tree. This information is stored persistently on the namenode's local disk. The namenode also keeps a registry of all the blocks of a given file and on which datanodes these blocks are located. The structure of the entire system is therefore highly dependent on the namenode. Without the namenode, the entire file system would be unusable since there is no way to reconstruct a file from the blocks stored on the datanodes. For this reason, it is important to make the namenode resilient to failure. This can be achieved by making regular backups of the persistent state of the file system metadata and writing them to multiple storage devices. These backup jobs can be done continuously by a special node called *secondary namenode*, which has to be dedicated to these tasks only as continuous registry merging could be computing intensive.
HDFS also provides replication methods to avoid data loss. Each time a new block is written, the namenode should also locate where replicas of the block should be placed. In common scenarios, every block is replicated three times. A CRC function

applied on every 512 bytes of each block is also normally used to achieve data integrity. A HDFS cluster is typically made by several (even hundreds) datanodes (data center), organized in different racks, which are redundant in nature, and one namenode, that is indeed extremely susceptible to hardware failure.
In order to ensure that HDFS works correctly, all the machines in the cluster should be pre-configured accordingly: every datanode must know who the namenode is and where it is located in order to run successfully, and the namenode must know every datanode's location to reach it and make it active during a working session. The HDFS working environment is extremely static. Therefore, if more machines are needed to perform a job (because their number has been underestimated), new machines need to be added to the cluster. On the other hand, if less machines are needed (because they have been overestimated), some machines would not be used and would not be operant; in both cases the entire system should be opportunely rebalanced and configured.

5.3.2 DLS-based HDFS

HDFS is intended to scale well on commodity machines, providing good data transfer performances with high reliability. HDFS provides methods for balancing the cluster load and adding/removing datanodes to and from the system. Each machine should be pre-configured to act as a datanode, which involves knowing exactly where the namenode is and how to reach it. Also, during system startup, the namenode should know an initial list of datanodes to work properly. These constrains can limit system's functionality in those scenarios which do not guarantee complete staticity, as every datanode should be configured for accessing a well-known namenode. This means that if the namenode changes because of a failure or if there is more than one namenode, each one for a specific, time-limited, working session, every datanode must be reconfigured. In order to overcome these limitations, a peer-to-peer layer (Kademlia-based DLS) could be introduced in the architecture to provide a way for datanodes to dynamically connect and disconnect from an HDFS working session without compromising overall system's performance.
The DLS is used to connect cluster machines one to each other and exchange in-

formation among them, with no static configuration needed. The namenode, which integrates a DLS peer, connects to the overlay and publishes information. When the namenode joins the overlay it publishes its status and service addresses in the DLS through successive put RPCs, thus making them accessible to other DLS peers. This information is then maintained collectively by all the peers participating in the DLS, but only the namenode can and must keep them fresh. All the information a node needs in order to become an active HDFS datanode is therefore available in the DLS and can be retrieved through get RPCs. As a new datanode enters the system, it can read the needed namenode information from the DLS layer, discover the namenode's location and contact it for authentication. As the namonode accepts the new datanode, the datanode becomes part of the HDFS cluster and all the following operations are done using the HDFS RPC protocol, so that the DLS is not involved anymore and HDFS performances are preserved. Figure 5.5 shows how namenodes and datanodes interact with the P2P substrate.
Since the information stored in the DLS has an expiration time associated, if the namenode fails, its access information is going to be deleted by the system. In order to better react to failures, a modified version of the Kademlia protocol has been defined. In case of failure of a node, when the failure is discovered, this information is propagated in the DHT, thus forcing other DHT nodes to remove the failed node from their routing tables. This mechanism improves considerably the average time for DLS-related RPCs with little overhead due to additional network traffic. While the namenode is unavailable, datanodes begin to poll on the DLS, waiting for the namenode or its replacement to become active. As soon as a namenode joins the DLS, it publishes its access information, so that datanodes can contact it to register and the system returns to a fully functional status. Therefore, the system itself is responsible for its functional consistency, thus eliminating the drawbacks of assistance in the event of failures.

5.3.3 Possible Enhancements

One-hop DHTs [6] appear to fit better into the architecture for our reference scenario. One-hop DHTs provide O(1) lookup procedures. This performance boost can

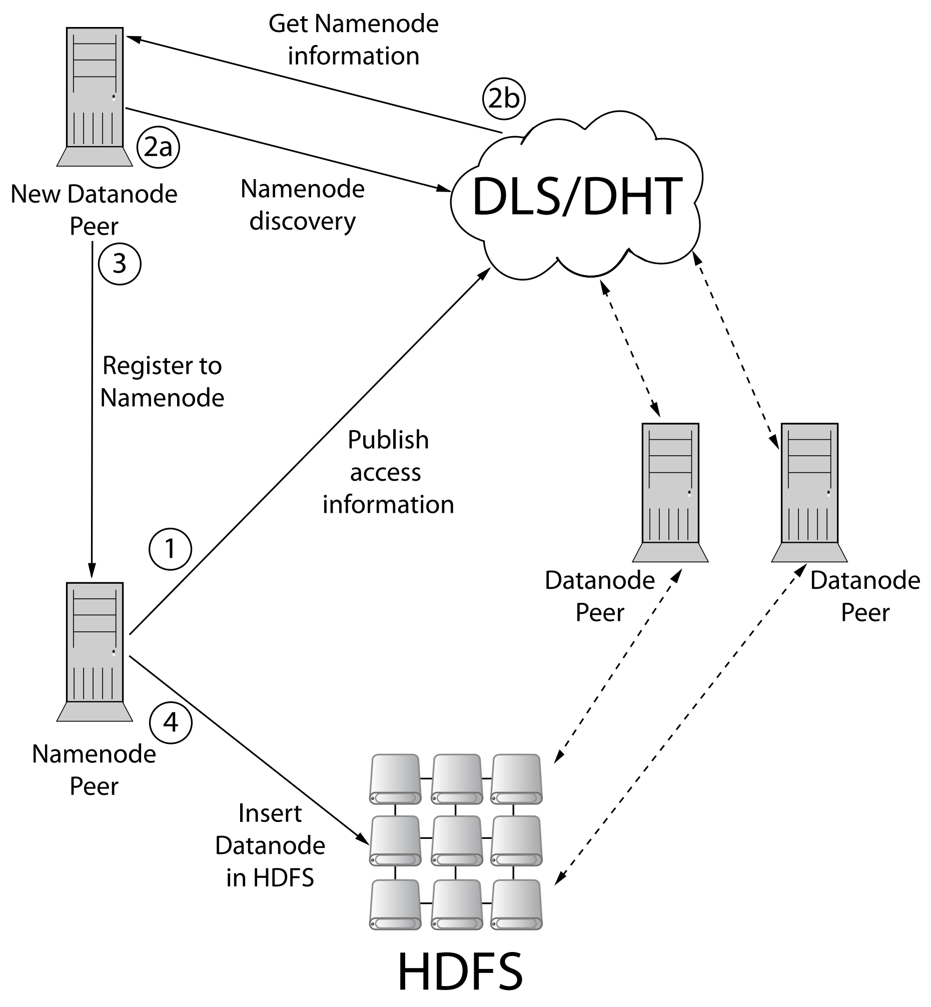


Figure 5.5: HDFS and DLS integration

be achieved only with O(N) state information, which make these DHTs suitable only for relatively low numbers of participating nodes. However, this is actually the kind of applications where such DHTs might offer their full benefits.
Another possible enhancement is to fully distribute the functionalities of the namenode. This feature would allow the deployment of a HDFS architecture in a purely P2P fashion. All nodes would collaborate to implement the management of the file system collectively, thus making the entire architecture more robust, since the namenode functionalities would be divided among all the peers. The advantages would therefore be the elimination of single point of failure and increment the performance for the system since the intermediation of the namenode bottleneck would be no longer required. Replacing the namenode with a distributed system comes at the price of overloading all datanodes of all the management of all the information related file system and the synchronization with the overlay participants. This approach increases the overall complexity and must be properly designed to ensure that full functionalities are available at any time, even when nodes fail.

5.4 Peer-to-peer Online Social Networks

Over the last few years, peer-to-peer and online social network (OSN) applications have dramatically changed the world of the Internet, by increasingly gaining their share of overall network traffic. Both kinds of applications are mainly focused on single users, who are no longer just service and content consumers but have also acquired the crucial role of content creators (Web 2.0) and service enablers and providers (P2P). Despite their growth, these two families of applications are still considered to be two separate worlds. The reason for this is mainly due to what these applications are used for: P2P applications are mainly used for file sharing, while social networks are essentially a means of communications.
The P2P network paradigm has been conceived to overcome some shortcomings of the client-server architectures by introducing the features of decentralization, scalability, robustness, and self-organization. P2P networks benefit from the aggregate resources of users, in terms of bandwidth, computational power, storage, rather than

on a limited centralized architecture. P2P networks actually improve their performance when the number of users increases as the overall capacity of the system also increases. Contrast this with centralized systems, where the growth of the number of accesses makes the service worse, as the slice of resources allocated to each user gets smaller and smaller. P2P networks are typically built on an overlay network, which is built on top of an existing network such as the Internet. The overlay network is built by establishing virtual (logical) links among the nodes, according to specific rules. The rules and mechanisms that regulate the creation of overlay links determine the nature of the P2P network, which might be structured or unstructured.

Social network services present some issues in terms of scalability, robustness, interoperability, and privacy. All these features are achieved mainly by the presence of centralized entities, whose redundancy allows to basically nullify the downtime of the service. However, this solution introduces some concerns for a series of reasons. First, the infrastructure needed to sustain huge, and inherently increasing, amounts of information and traffic requires enormous economical efforts. Because of this, it is very difficult for newly born social networks to compete with well-established services, such as Facebook[3] and MySpace[4]. Second, despite the development of open service APIs to allow third-party applications to access and publish content on existing services, full interoperability is still far from being achieved. Finally, privacy is basically achieved through user authentication and access privilege policies, which are usually deployed on centralized systems. Since social network services typically store (very) personal information about users, efforts have been made to implement and ensure security. However, there is still some concern about the fact the so many private information about people are in the hands of few, who may appear as third-millennium “big brothers”.

In this section, a solution to integrate the world of P2P with social networking is proposed, in order to address the issues that we have described. The goal is to create a P2P infrastructure that resembles existing social networks in terms of available services, but tries to overcome the problems of centralized architectures by exploi-

[3]`http://www.facebook.com`

[4]`http://www.myspace.com`

ting the typical features of distributed systems: scalability, fault-tolerance, and self-organization.

5.4.1 P2P OSN Architecture

The P2P OSN architecture is based on three basic building blocks or components: a Distributed Location Service (DLS), a Distributed Storage Service (DSS), and a Privacy Enforcement Framework (PEF). The DLS (Chapter 3) is a P2P service based on a Distributed Hash Table (DHT) which allows the storage and retrieval of access information for any kind of resource, thus allowing the establishment of direct connections among the endpoints of a communication in a pure P2P fashion, without the need for intermediate nodes. The DLS is also used for storing off-line messages, posts, and news, and for addressing other resources like images, videos, and other multimedia resources. The DSS, which is still based on the location service offered by the DLS, is a distributed platform for the actual storage, publishing, and retrieval of any multimedia resource. Finally, the PEF provides proper security and privacy functionalities through public key and symmetric cryptography combined with a key distribution architecture. Over these building blocks, there are the actual Social Network Services, such as Presence, Instant Messaging (IM) and VoIP, offline messaging, posts, and multimedia content sharing. Let us consider in more details all these components.

5.4.2 Data Storage

In the proposed OSN architecture, the DLS provides the link function between all used resources (users, services, multimedia resources, etc.) and their actual contact address, access protocol, and policy rules. In addition to such access information, the OSN platform should also provide support for the actual storage and retrieval of all physical resources (such as profile contents, messages, posts, news, multimedia files, etc.), in a reliable, scalable and efficient way. For practical convenience, we divide data into two main categories: small data and big data. Small data include all resources that are not big in nature and that require very small amount of bytes to

be recorded, ranging from few bytes (user profile information, such as name, email address, etc.) to some hundreds or thousands of bytes (offline messages, posts, news, etc.). Instead, big data include all other resources (such as images, audio/video files, programs, etc.) that range from hundreds of Kbytes to Gbytes or more. Due to the relatively small amount of bytes required for small data, according to our architecture, they are stored directly within the DLS and managed (inserted, replicated, fetched and deleted) according to the corresponding DHT algorithm. On the other hand, big data are stored in the Distributed Storage Service, a proper scalable and efficient platform.
Note that, regarding small data, in the proposed implementation, RFC 2397 [17] was exploited, thus allowing the encapsulation of small data within a URI, in order to directly use the DLS for both resource registration, and small data storing.

5.4.3 Distributed Storage Service

In section 5.2, the implementation of a distributed web server (DWS) based on a underlying DLS service was proposed. Such DWS may act as a regular web server to end users, where resources can be transparently navigated by using any standard web browser. Under the hood, resources are actually stored in a distributed environment of different nodes, thus allowing more robustness, scalability, and load balancing. Such simple distributed web server service has been further extended in order to provide a distributed, efficient, and scalable storing and publishing mechanism. Storing peers are chosen from the DLS routing tables (peer lists), and registered on the DLS, which offers the proper resource location lookup service. Persistence is implemented by a replication mechanism that allows resource copies to be transferred among the DWS nodes. Resources are stored to and retrieved from other peers through standard HTTP or HTTPS protocols. This results in a completely Distributed Storage Service (DSS). The DSS can be used to share any kind of files.

5.4.4 Privacy Enforcement Framework

A crucial aspect in online social networks is the exposure of very personal information about users to the world. Concerns about privacy become even more when considering a distributed environment, where no central authentication authority is present. In the DLS, data are stored at unrelated points of the overlay network, according to the rules defined by the DHT algorithm in use. This means that private information might be stored on any node, even those which the user does not want to make his information available to.
In order to solve this problem, a Privacy Enforcement Framework (PEF) is proposed, that includes both encryption rules and a key distribution architecture used for storing and retrieving both small and big data.

Evaluation of Key Distribution Techniques

In order to create the PEF, different strategies for key distribution have been considered. Assume there is a user u. u has a set of friends $U = \{u_1, u_2, ..., u_n\}$. $|U| = n$.
u wants to publish a set of resources: $R = \{r_1, r_2, ..., r_r\}$. $|R| = r$.
Assume each user x (or friend) has a pair of public/private keys $(K^+{}_x, K^-{}_x)$.
The PEF should make it easy to distribute and revoke keys for encrypting contents and change access privileges dynamically.

Case 1 - Single user key u creates a key K_u. u communicates K_u to each friend $u_i \in U$, encrypted using the public keys of his friends. This is accomplished by performing n put RPCs in the DLS. u publishes his r resources, encrypted using K_u. This is accomplished by performing r put RPCs in the DWS.

Case 2 - A key for each resource u creates a key K_{ri} for each resource $r \in R$: $\{K_{r1}, K_{r2}, ..., K_{rr}\}$. u publishes his resoruces in the DWS, ecncrypted using resource keys. This is accomplished by performing r put RPCs in the DWS. u publishes the resource keys in the DLS, encrypted using the public keys of his friends. This is accomplished by performing nr put RPCs in the DLS.

Case 3 - A key for each user, a key for each resource u assigns a key K_{ui} to each user $u_i \in U$ and a key K_{ri} to each user $r_i \in R$. u publishes in the DLS the keys for each user, encrypted using the public keys of his friends. This is accomplished by performing n put RPCs in the DLS. u publishes in the DWS his resources, encrypted using each resource key K_{ri}. This is accomplished by performing r put RPCs in the DWS. u publishes in the DLS the keys for resources, encrypted using the keys of friends. This is accomplished by performing nr put RPCs in the DLS.

Case 4 - Binary tree key assignment This technique is based on the binary tree method for deterministic revocation of privileges [26]. u creates a binary tree, where every node has a key associated to it, and each leaf is assigned to a friend. Each friend is assigned a set of keys (the keys that are on the path form the root to their assigned leaf). This is accomplished by performing $n \cdot log_2 n$ put RPCs in the DLS. u assigns a key K_{ri} to each user $r_i \in R$. u publishes his resources in the DWS, encrypted using each resource key. This is accomplished by performing r put RPCs in the DWS. u publishes the keys associated to resources in the DLS, encrypted using the smallest subset of keys (let's say n_k) that cover the leaves of desired users (best (typical) case 1, worst case $n/2$). This is accomplished by performing $n_k \cdot r$ put RPCs.

	case 1	**case 2**	**case 3**	**case 4**
total keys for users	1	0	n	$2n-1$
total keys for resources	1	r	r	r
total DLS put procedures	n	nr	$(n+1)r$	$n \cdot log_2 n + n_k \cdot r$
total DWS put procedures	r	r	r	r

Table 5.2: Evaluation of different key distribution approaches

Evaluation In the case of adding a friend, the operations that need to be performed in the DLS/DWS are shown in table 5.3. In the case of removing a friend, the ope-

	case 1	**case 2**	**case 3**	**case 4**
DLS put procedures	1	r	r	1
DWS put procedures	0	0	0	0

Table 5.3: Evaluation of different approaches for key addition

rations that need to be performed in the DLS/DWS are shown in table 5.4. Case 1

	case 1	**case 2**	**case 3**	**case 4**
DLS put procedures	$2n-1$	$(2n-1)\cdot r$	$(r+1)(2n-1)$	$n_k \cdot r$
DWS put/remove procedures	$2r$	$2r$	$2r$	0

Table 5.4: Evaluation of different approaches for key revocation

requires less keys to be stored in the DLS, but it suffers of inefficiencies in the case of changes in U. Case 2 and 3 require more keys to be stored in the DLS, but suffer great inefficiencies in the case of changes in U. Case 4, finally, requires even more keys to be stored in the DLS, but offers great flexibility and efficiency in the case of changes in U. If removing a friend, in case 4 there is no need to remove anything from the DWS, which is quite desirable since the complexity of this operation is high.

Binary Tree Key Distribution

Suppose any user x in the social network has a pair of public/private keys $({K_x}^+, {K_x}^-)$. Each user also has a list of friends $U_x = \{u_1, u_2, ..., u_m\}$ (buddy list) that defines which users have the privilege to access the private contents of x. Let's also assume, for simplicity, that $|U_x| = m = 2^n$. x constructs a binary tree T_{k_x} with m leaves, each one assigned to a user in its buddy list. A key is assigned to each node in the tree as in Figure 5.6. Each user $u_i \in U_x$ then stores all the $O(log_2 m + 1)$ keys K_i in the path from the root to its assigned leaf. User x stores all the $2m-1$ keys of the tree. x publishes all the K_i sets in the DLS with a URI that associates x with each u_i and a value that is the K_i encrypted using ${K_x}^-$ and ${K_{u_i}}^+$ in order to ensure both authenticity and confidentiality.

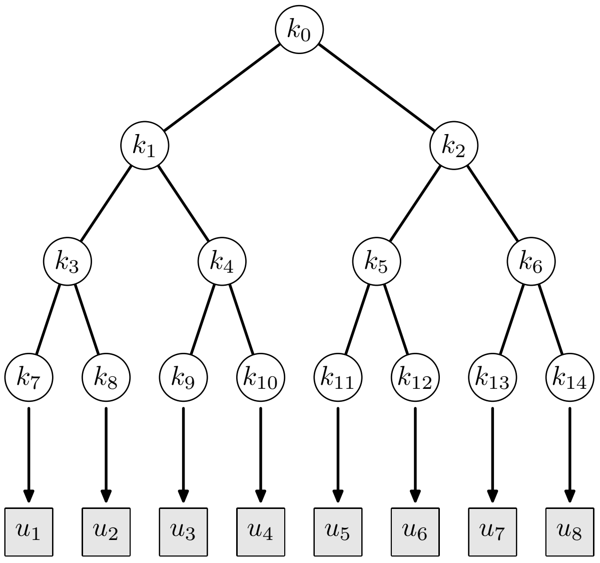


Figure 5.6: Key distribution tree

Let M be a message that x is willing to share with a subset $L \subseteq U$ of his buddies. x can encrypt M using a session key K_M and obtain a ciphered message $C = E_{K_M}(M)$. x then stores the ciphered message C in the DSS (that is, it stores the reference to the message M in the DLS, and C is stored on a node that belongs to the DSS). In order to make M accessible to the users in L, x stores in the DLS a so-called *encryption header* H_M, which is computed by selecting the smallest subset of keys of the tree T_{k_x} that covers all the users is L and then encrypting with such keys the session key K_M. Figure 5.7 shows how a user can store encrypted content in the DSS/DLS that is decryptable only by some selected users.

When a user u_i wants to access the message M, it can retrieve C and H_M from the DLS/DSS system. Only if $u_i \in L$, H_M can be decrypted using one of the keys assigned to u_i to get the session key K_M. Once C and K_M are available, M can be obtained by decrypting C: $M = D_{K_M}(C)$.

If the message M needs to be accessed by a different set of users L', in the case x adds or removes a user from its buddy list, x just needs to select the new smallest subset of

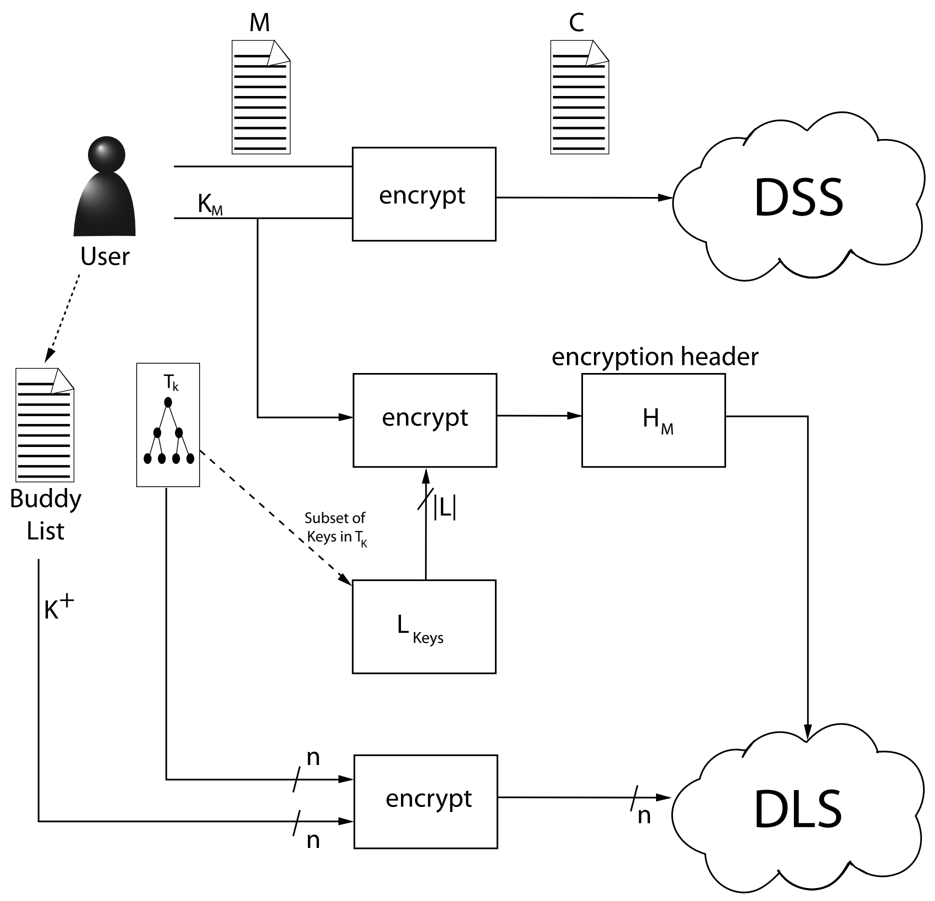


Figure 5.7: Storing encrypted content in the DSS/DLS

keys that covers all the users in L', compute the new encryption header H'_M, and store it in the DLS.

5.4.5 Social Network Services

User profiles are persistently stored into the DLS. This information includes the user's name, email address, and other significant attributes. Implementing some services, such as presence, messaging, and media sessions into the proposed P2P social network is quite easy. In general, all the information stored by a user is encrypted using, at least, its private key K^-, in order to ensure authenticity.

Presence

Presence can be implemented by storing presence information about users directly in the DLS. When a user logs in, it stores his presence information into the DLS (which has an expiration time and must be kept fresh). Then it checks his list of friends (which is, in terms of presence services, the list of users who subscribed for user's presence notifications) and sends a notification message directly to each of them.
When a user logs out, it removes its presence information from the DLS. In the case a user does not explicitly log out, its presence information will automatically expire.

IM and VoIP

IM and VoIP services are traditionally implemented through a centric server-based approach. Within IETF a specific Working Group (P2PSIP) is currently working on a proper architecture and protocol for moving toward a completely decentralized distributed VoIP architecture. According to the work done in IETF, any SIP-based instant messaging and VoIP session can be established by simply performing a lookup operation in a DLS that in turn can be based on a DHT, such as in our architecture. In case of a call or a messaging session has to be set-up between two or more user, the targeted user's contact is simply retrieved from the DLS, and a direct media session can then be established between the two endpoints.

Offline messaging and media contents

Posting short messages or news relies again on the direct storage of short data within the DLS DHT, as described in previous sections. Posting multimedia contents, instead, relies on the DSS platform. When a user wants to publish content, it stores into the DLS a reference to its locally stored resource. When other users want to access it, they just need to look it up in the DLS and then access it directly. In order for this data to be available even when the user disconnects, a replication mechanism is implemented in order to ensure that at least a certain number of copies is maintained in the DSS. Since this kind of operation replicates the media on unrelated points of the overlay network it is necessary to encrypt the media contents according to the PEF described earlier, so that no one but authorized users can access it.

Chapter 6

Conclusions

Nowadays, peer-to-peer technology is no longer used to create just file sharing applications. Many applications have shown how the peer-to-peer paradigm can be efficiently used to implement other kinds of services, such as Skype did with VoIP. The peer-to-peer model offers great opportunities to market newcomers as it permits to keep costs down as the infrastructure needed to create the service can be very cheap or even free.

Several peer-to-peer applications use Distributed Hash Tables (DHT) as a building block. DHTs are a class of decentralized distributed systems that provide a lookup service similar to a hash table; (key, value) pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

This thesis has addressed the problem of defining a DHT-based peer-to-peer architecture to create distributed applications beyond file sharing. The goal of the architecture was to create a scalable, fault-tolerant, self-organizing lookup service to store and retrieve access information for resources in order to allow the establishment of direct connections between the endpoints of the communication with no need of central servers.

The activity has been focused on the definition of a Distributed Location Service (DLS), which acts as a distributed registry where resource access information can be stored and retrieved. Access information, in the form of key-to-value mappings (i.e., resource URI to resource URL) is maintained collectively by the peers that participate in the DHT that is used to build the DLS. The implementation of a DLS framework has also been realized, by defining a standard set of interfaces between the components of the DLS, in order to allow maximum flexibility on components such as the DHT algorithm and communication protocol in use, as no assumption has been made in the definition of the DLS architecture. The DLS framework allows to create instances of Distributed Location Services, particularized by their own DHT algorithm and communication protocol, as needed by the application that is going to access the DLS.

Based on the DLS architecture and framework, some demonstrative applications have also been realized, such as peer-to-peer VoIP, a Distributed Web Server, a distributed File System, and a peer-to-peer Online Social Network. These applications show how easy distributing Internet applications can be with the support of the DLS and that peer-to-peer technology is a very efficient tool to create other application than file sharing. Other interesting applications that have been considered are P2P-based streaming systems for WebTV, even though the issues that must be addressed in this kind of applications make them more difficult to realize.

Another direction of the research was the definition of a decentralized, scalable, and self-configuring bootstrap service for peer-to-peer networks, which is a problem usually solved with non-P2P approaches. The solution presented is a Multicast-based notification service which allows new nodes to gather information about nodes that are already part of a peer-to-peer network in order to be admitted in the overlay. The proposed solution has been validated and can be easily integrated with the DLS in order to obtain a true distributed, scalable, fault-tolerant, and self-organizing architecture that can be used to eliminate the need of central servers in several scenarios.

Bibliography

[1] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[2] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. *Lecture Notes in Computer Science*, 2429:53–??, 2002. Available from: http://link.springer.de/link/service/series/0558/bibs/2429/24290053.htm;http://link.springer.de/link/service/series/0558/papers/2429/24290053.pdf.

[3] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31:161–172, August 2001. Available from: http://doi.acm.org/10.1145/964723.383072, doi:http://doi.acm.org/10.1145/964723.383072.

[4] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Rachid Guerraoui, editor, *Middleware 2001*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer Berlin / Heidelberg, 2001. Available from: http://dx.doi.org/10.1007/3-540-45518-3_18.

[5] B.Y. Zhao, Ling Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *Selected Areas in Communications, IEEE Journal on*, 22(1):41 – 53, January 2004. doi:10.1109/JSAC.2003.818784.

[6] Luiz R. Monnerat and Claudio L. Amorim. D1HT: A distributed one hop hash table. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006. Available from: http://www.cos.ufrj.br/~monnerat/papers/Monnerat_et_Amorim_D1HT_2006.pdf.

[7] Simone Cirani and Luca Veltri. A Multicast-based bootstrap mechanism for self-organizing P2P networks. In *Proceedings of the 28th IEEE conference on Global telecommunications*, GLOBECOM'09, pages 6243–6248, Piscataway, NJ, USA, 2009. IEEE Press. Available from: http://portal.acm.org/citation.cfm?id=1811982.1812417.

[8] C. Cramer, K. Kutzner, and T. Fuhrmann. Bootstrapping locality-aware P2P networks. In *Networks, 2004. (ICON 2004). Proceedings. 12th IEEE International Conference on*, volume 1, pages 357 – 361 vol.1, November 2004. doi:10.1109/ICON.2004.1409169.

[9] Michael Conrad and Hans-Joachim Hof. A generic, self-organizing, and distributed bootstrap service for peer-to-peer networks. In David Hutchison and Randy Katz, editors, *Self-Organizing Systems*, volume 4725 of *Lecture Notes in Computer Science*, pages 59–72. Springer Berlin / Heidelberg, 2007. Available from: http://dx.doi.org/10.1007/978-3-540-74917-2_7.

[10] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002. Updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630, 5922, 5954, 6026. Available from: http://www.ietf.org/rfc/rfc3261.txt.

[11] J. Klensin. Simple Mail Transfer Protocol. RFC 2821 (Proposed Standard), April 2001. Obsoleted by RFC 5321, updated by RFC 5336. Available from: `http://www.ietf.org/rfc/rfc2821.txt`.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785. Available from: `http://www.ietf.org/rfc/rfc2616.txt`.

[13] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Standard), January 2005. Available from: `http://www.ietf.org/rfc/rfc3986.txt`.

[14] D. Bryan, B. Lowekamp, and C. Jennings. dSIP: A P2P Approach to SIP Registration and Resource Location. Internet-Draft draft-bryan-p2psip-dsip-00, Internet Engineering Task Force, February 2007. Available from: `http://tools.ietf.org/id/draft-bryan-p2psip-dsip-00.txt`.

[15] C. Jennings, B. Lowekamp, E. Rescorla, S. Baset, and H. Schulzrinne. REsource LOcation And Discovery (RELOAD) Base Protocol. Internet-Draft draft-ietf-p2psip-base-12, Internet Engineering Task Force, November 2010. Available from: `http://tools.ietf.org/id/draft-ietf-p2psip-base-12.txt`.

[16] S. Cirani and L. Veltri. Implementation of a framework for a DHT-based distributed location service. In *Software, Telecommunications and Computer Networks, 2008. SoftCOM 2008. 16th International Conference on*, pages 279 – 283, September 2008. `doi:10.1109/SOFTCOM.2008.4669495`.

[17] L. Masinter. The “data” URL scheme. RFC 2397 (Proposed Standard), August 1998. Available from: `http://www.ietf.org/rfc/rfc2397.txt`.

[18] M. Zangrilli and D. Bryan. A Chord-based DHT for Resource Lookup in P2PSIP. Internet-Draft draft-zangrilli-p2psip-dsip-dhtchord-00, Internet Engi-

neering Task Force, February 2007. Available from: `http://tools.ietf.org/id/draft-zangrilli-p2psip-dsip-dhtchord-00.txt`.

[19] Simone Cirani and Luca Veltri. A Kademlia-based DHT for Resource Lookup in P2PSIP. Obsolete Internet draft, October 2007.

[20] E. Marocco and E. Ivov. Extensible Peer Protocol (XPP). Internet-Draft draft-marocco-p2psip-xpp-01, Internet Engineering Task Force, November 2007. Available from: `http://tools.ietf.org/id/draft-marocco-p2psip-xpp-01.txt`.

[21] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. RFC 5245 (Proposed Standard), April 2010. Available from: `http://www.ietf.org/rfc/rfc5245.txt`.

[22] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878. Available from: `http://www.ietf.org/rfc/rfc5246.txt`.

[23] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347 (Proposed Standard), April 2006. Updated by RFC 5746. Available from: `http://www.ietf.org/rfc/rfc4347.txt`.

[24] Simone Cirani, Riccardo Pecori, and Luca Veltri. A peer-to-peer secure VoIP architecture. In *21st International Tyrrhenian Workshop on Digital Communications (ITWDC)*, September 2010.

[25] Simone Cirani, Lorenzo Melegari, and Luca Veltri. Peer-to-peer technologies applied to data warehouses. In *Workshop on the Application of Communication Theory to Emerging Memory Technologies (ACTEMT 2010)*, December 2010.

[26] Sara Checcoli. A study of combinatorial revocation schemes. Master's thesis, University of Padova, June 2007.

Acknowledgements

I would like to thank first my Ph.D. tutor, Prof. Ing. Luca Veltri, for the great help, support, and opportunities he granted to me in these years. It has been a honor and a pleasure to collaborate with you, and I could not find a better friend to help me and guide me throughout this path.

I would like to express all my gratefulness and love to my wonderful family, who supported and encouraged me all my life. Mom and Dad, thank you for your trust and patience, for being always by my side, helping me, and believing in me. Always. My wonderful sisters, Alessandra and Maddalena, you have been always a constant inspiration, I love you so much, and I could never say or do anything that could express how blessed I am to have you in my life. Jonas, thanks for all your help, if I have taken this path in my studies, this is because of you. Emil and Emma, I am the luckiest and proudest uncle in the world! Finally, my grandma Jolanda, thank you for your support and always remembering me.

Paola, staying with you is the most wonderful gift I could ever receive in my life. Thank you for loving me, taking care of me, bearing with me, with your sweetness. Your help and support have been precious and valuable, and I could not make it without you. You are just too important to me. I love you so much and I wish to stay with you forever.

A huge thanks goes to my friends Lorenzo, Gianni, Duiz, Fra, Cliff, Lollo, and Imer for being supportive and giving me such a great time!

Another huge thanks goes to all my band mates, Grinning Shadows (Mario, Silvia, Duiz, Lorenz, Rove, Gigi), Dark Lunacy (Mike, Baijkal, Imer, Mary), and Le

Chatiment De La Secte (Lollo, Thomas), for giving me a great relief valve in music.

I would also say thanks to all the people that I met during these years in lab, especially Alessandro, with whom it's been really great to work, Riccardo, with whom I shared this path since the beginning, and Natalya, with whom it has been great to share ideas. Another great thanks goes to Marco who helped me many times with valuable insights.

Finally, I would like to thank all the people that supported me in my life and helped me in many ways. Thanks, you know who you are...