UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

P2P overlay over Simctl

Author:
Gerard Carles Sicart López

Advisor:
Dr. Óscar Esparza Martín

A Master's Thesis Submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
in Partial Fulfillment of the Requirements for the Degree of

MASTER IN TELECOMMUNICATIONS ENGINEERING

October 2016

Acknowledgements

I would like to express my gratitude to my supervisor, Óscar Esparza, for his valuable advice and guidance, as well as the trust and support he has shown me throughout this time.

Preface

P2P systems became an interesting area since early 2000. Researchers conducted a large amount of research in some challenging areas and, to check their experiments, several implementations and simulators were created.

Over time, Internet has evolved and P2P has been widely used for file sharing, but the main structured P2P overlays were progressively abandoned. However, these overlays are still taught in network engineering courses. The aim of this master thesis is to find and implement a p2p overlay over the `simctl` platform, used both for teaching and research in the network engineering department. First, it is tried to locate and test the original structured P2P overlays source codes developed by the authors of the original publications. Then, several available P2P implementations developed by researchers and users have been tested. Once the software was selected, different scenarios for `simctl` were developed. Finally, a lab session was created by using the previous scenarios.

Contents

1 General Introduction **1**

1.1 Context . . . 1

1.2 Objectives . . . 2

1.3 Organization . . . 3

2 Background **5**

2.1 Simctl . . . 5

2.1.1 UML . . . 5

2.1.2 VNUML . . . 6

2.1.3 Simctl . . . 6

2.2 Peer-to-Peer (P2P) Networks . . . 7

2.2.1 Three Generations of P2P Networks . . . 9

2.2.2 Identity Management in Existing P2P Overlays . . . 11

2.2.2.1 CAN (Content-Addressable Network) . . . 11

2.2.2.2 Chord . . . 12

2.2.2.3 Pastry . . . 14

2.2.2.4 Tapestry . . . 14

2.2.2.5 Kademlia . . . 15

2.2.2.6 BitTorrent . . . 17

2.2.2.7 JXTA . . . 18

3 Tested solutions **21**

3.1 CAN . . . 22

3.2 Chord . . . 24

3.2.1 MIT PDOS Chord implementation . . . 24

3.2.2 Open Chord . . . 26
3.2.3 Chordial . . . 29
3.2.4 JChord . . . 30
3.2.5 Chordjerl . . . 30
3.2.6 jDHTUQ . . . 30
3.3 Pastry . . . 33
3.3.1 FreePastry . . . 34
3.3.2 PAST . . . 36
3.4 Tapestry . . . 39
3.4.1 Tapestry project . . . 39
3.4.2 Chimera . . . 40
3.4.3 OceanStore . . . 41
3.5 Kademlia . . . 43
3.5.1 OpenDHT . . . 43
3.5.2 Pydht . . . 45
3.5.3 TomP2P . . . 45
3.5.4 Hive2Hive . . . 46
3.5.5 MaidSafe . . . 48
3.6 JXTA . . . 51
3.6.1 JXTA/JXSE . . . 51
3.7 Edonkey . . . 54
3.7.1 aMule . . . 54
3.8 Routing algorithm simulators . . . 58
3.8.1 Overlay Weaver . . . 58
3.8.2 Entangled . . . 61
3.8.3 OverSim . . . 63
3.8.4 P2PSim . . . 65
3.8.5 PeerSim . . . 66
3.8.6 PeerFactSim.KOM . . . 67

4 Implementation **71**
4.1 OpenChord . . . 71
4.2 aMule . . . 78

5 Lab session 85

6 Conclusions and Further Work 99

References 102

List of Figures

2.1 `simctl` operation. 7
2.2 Global traffic by application category. 8
2.3 CAN bootstrapping phase. 12
2.4 Example of a Chord identifier circle modulo 2^3. 13
2.5 Generation of *buckets* in the node 111...110. 16

3.1 OpenChord interface with available commands. 28
3.2 jDHTUQ in Structure mode. 31
3.3 FreePastry capture. First node. 35
3.4 FreePastry capture. Second node. 35
3.5 FreePastry capture. Third node. 36
3.6 Past capture. FreePastry's DHT with 5 nodes on a single machine. . . . 37
3.7 FreePastry Demo capture. Network with 6 nodes running on a single machine. 38
3.8 Hive2Hive demo main menu. 47
3.9 Kademlia demo main menu (screenshot from MaidSafe-DHT documentation). 48
3.10 SAFE launcher capture. 50
3.11 Distributed Storage Leasing capture. Two peers in a local network. . . . 53
3.12 aMuleCMD client console. 56
3.13 Simple test of Chord running in a single host. 60
3.14 Messaging visualizer showing a Chord ring (extracted from the project's website). 61
3.15 Entangled GUI. 62
3.16 Kademlia demo GUI. 63

3.17 Chord ring in OverSim. 65
3.18 Visualization environment of PeerfactSim. 69

4.1 OpenChord test scenario. 74
4.2 Chord ring evolution. 76
4.3 aMule test scenario. 81
4.4 eserver commands. 82
4.5 aMule client connection to eD2k server. 82
4.6 eserver console showing clients connected to it. 83
4.7 Search and download process. 84

Chapter 1

General Introduction

Contents

1.1 Context 1

1.2 Objectives 2

1.3 Organization 3

1.1 Context

Most of our daily activities are carried out over the Internet, from home-banking and on-line teaching to watching on-line content, social networking, and file sharing. Considering file sharing as one of the Internet top activities, different generations of P2P networks have been proposed, designed and implemented. Over the last years, these networks have evolved from centralized approaches to fully decentralized and structured systems. Today, structured P2P networks, also known as P2P overlays, are considered the best way to share and distribute large amounts of data, regardless of the involved devices and underlying networks used below them. This is mainly due to they are mostly based on Distributed Hash Tables (DHT).

In the last few years, P2P networks have mainly been used for file sharing or multimedia streaming. But P2P based content distribution systems have emerged as a form for content distribution on the Internet, which can greatly reduce the distribution cost of content providers and improve the overall system scalability. Moreover, with the growth of Internet of Things (IoT) and the expected appearance of 5G technology in

2020, millions of devices will be used and P2P could take an interesting role to provide communication between them in device-to-device (D2D) and machine-to-machine (M2M) communication networks.

1.2 Objectives

This master project aims to find and implement a P2P overlay over the `simctl` platform, used both for teaching and research in the network engineering department. Therefore, the objectives of this project are as follows:

- To examine the evolution of P2P overlays from its origin to today, watching the current state of P2P development and research.
- To find implementations of the main P2P overlays. If possible, to find the original implementations developed by the authors of the reference paper publications.
- To install and test each implementation code on a virtual local network with four virtual machines. This virtual network is composed by four Ubuntu 14.04 virtual machines, each one with 1024MB of RAM memory, and it is created by using Oracle VM Virtual Box.
- To create a scenario for `simctl` with several machines. Then, to install and test over `simctl` those working P2P implementations that can run by console and that let the user interact with the software.
- To create a lab session for the Network Engineering department where the students can run the scenario with the implementation installed, and then analyse its behaviour and the traffic sent between nodes.

The different virtual machines in `simctl` are executed by console and have no graphical interface. Moreover, most `simctl` scenarios simulate a local network with no Internet connection, so the main requirements for an implementation to be used in `simctl` are:

- To run in console mode, it is, with no graphical user interface (GUI).
- To be able to run in a real network between different machines. Those implementations that run a virtual network with hundreds of nodes on a single host (physical machine) are not valid.

- To be able to run on Debian 6 Squeeze or Debian 5 Lenny, since they are the typical file systems used by `simctl`.

1.3 Organization

This master project is organized in 6 Chapters. Chapter 2 begins by explaining briefly what simctl is, and some concepts related to its operation. Then, it introduces the evolution of the P2P networks over the years and explains in detail the operation of the most representative DHT-based structured P2P networks.

Chapter 3 collects some of the codes that have been tested. Several projects and source codes have been installed and tested, but only the most relevant ones are explained in this chapter. Many of the projects located on code repositories were unfinished or abandoned, or simply they were not able to run on a local network between different machines. Others were projects developed by students for Distributed Courses that were not faithfully to the original algorithm. Most of these unuseful codes were tested but not listed here.

Chapter 4 describes the selected peer-to-peer implementations that meet all the required characteristics. In this section it is explained how to install them on simctl, and some examples are tested.

Chapter 5 presents the lab session developed by using simctl and the selected working P2P implementations.

Finally, Chapter 6 concludes this thesis summarizing the main findings and making suggestions for the future research.

Chapter 2

Background

Contents

2.1 Simctl 5

2.1.1 UML 5

2.1.2 VNUML 6

2.1.3 Simctl 6

2.2 Peer-to-Peer (P2P) Networks 7

2.2.1 Three Generations of P2P Networks 9

2.2.2 Identity Management in Existing P2P Overlays 11

2.1 Simctl

2.1.1 UML

User Mode Linux (UML) [54] was created as a kernel development tool to be able to boot a kernel in the user space of another kernel. It enables the creation of Linux virtual machines (guest) on Linux machines (host). The guests are created and managed as a process rather the system. Virtualization is made from a binary file of a Linux kernel and a file containing the file system, and the changes that can be made in both the kernel and the file system do not affect the host. So, if a developer messes with the code and the kernel is unstable, it is not necessary to reboot the host, just kill the kernel process.

2.1.2 VNUML

Virtual Network User Mode Linux (VNUML) [58] is an open-source general purpose virtualization tool designed to quickly define and test complex network simulation scenarios based on the User Mode Linux (UML) virtualization software. It can simulate general Linux based network scenarios using UML. VNUML was developed at the end of 2002 at the Department of Telematic Systems Engineering (DIT) of the Polytechnic University of Madrid (UPM).

VNUML tool is made of two main components:

- The VNUML language used for describing simulations in XML (Extensible Markup Language).
- The interpreter of the language (vnuml command), that builds and manages the scenario hiding all UML complex details to the user.

Using the VNUML language the user can write a simple text file describing the elements of the VNUML scenario such as virtual machines, virtual switches and the inter-connection topology. Then, the user can use the VNUML interpreter called vnuml-parser.pl to read the VNUML file and to run/manage the virtual network scenario.

2.1.3 Simctl

`simctl` is a virtualization tool based on VNUML, and developed with the aim of simplifying and extending the management capabilities of VNUML. It was developed at the Network Engineering department (ENTEL) of the Universitat Politècnica de Catalunya (UPC) by José Luis Muñoz, Gustau Pérez and Juanjo Alins. `simctl` is a VNUML wrapper written for Bash. The commands used throughout `simctl` are adapted so that the interpreter VNUML can deliver them correctly to UML to do the processing. It is, although the tool used by the user is `simctl`, basically what is being used is UML. Therefore, `simctl` makes use of VNUML, which in turn is using UML.

The script `simctl` allows the user to search for the different scenarios that he can run, start and stop a simulation, list the virtual machines that are part of a simulation, list the "labels" defined on each machine of a simulation, run defined labels, manage the consoles to access the virtual machines, view the network structure, and some more things.

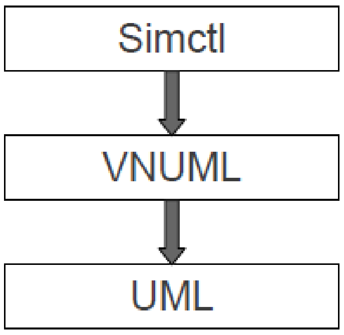


Figure 2.1: `simctl` operation.

2.2 Peer-to-Peer (P2P) Networks

Peer-to-Peer (P2P) networks are worldwide distributed systems where each node can be used both as a client and a server simultaneously. These networks emerged as an incipient paradigm of communications to share resources and services in a highly decentralized way. They have had to evolve over time giving rise to three different generations.

Among others, some factors which have contributed to the success of these networks are the availability of increasingly cheap bandwidth and the growing number of computers sharing services and resources over the years. P2P networks are being massively used and will remain so in the coming years since their performance is being improved over time and their application is increasingly widespread.

According to the annual Cisco Visual Networking Index (VNI) Complete Forecast for 2015 to 2020 [64], the P2P file sharing will decrease at a compound annual growth rate (CAGR) of 5 percent from 2015 to 2020. The growing use of smartphones and other devices makes that users migrate to other file transfer systems. And streaming systems are the responsible of a decrease in file transfer, since an important part of the P2P file sharing users was focused on downloading videos and music.

Although the use of P2P will decrease in file sharing, these systems will continue to play an important role in the future Internet for sure. Many types of services may be built based on P2P protocols such as:

- File sharing applications (BitTorrent [3]).
- Voice-over-IP (VoIP) services and instant messaging clients (Skype [53]).

- Video streaming applications (CoolStreaming [8]).
- Music sharing portals (Jamendo [21]).
- File synchronization tools (BitTorrent Sync [4]).
- Payment methods and digital currencies (Bitcoin [2]).
- Massively Multiplayer Online Games (MMOG) platforms (Badumna [72]).

But above all, video streaming applications are the ones that are experiencing a special growth. With video growth, Internet traffic is evolving from a relatively steady stream of traffic (characteristic of P2P traffic) to a more dynamic traffic pattern. The sum of all forms of IP video, which include Internet video, IP VoD, video files exchanged through file sharing, video-streamed gaming, and video conferencing, will continue to be in the range of 80 to 90 percent of total IP traffic [65]. Globally, IP video traffic will account for 82 percent of traffic by 2020 (Figure 2.2).

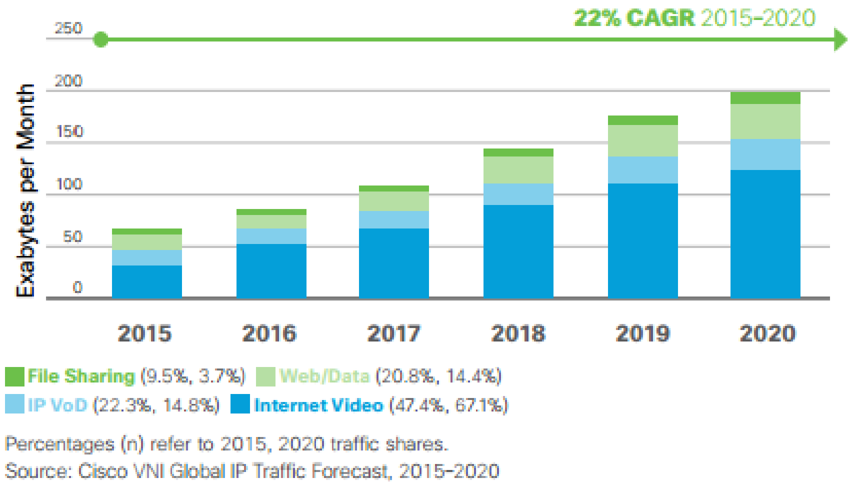


Figure 2.2: Global traffic by application category.

P2P is a low-cost content delivery system but unfortunately until now most content providers and distributors have opted for direct distribution, with the exception of applications such as PPStream [50] and PPLive [51] in China, which offer live video streaming through P2P and have had great success. Another application that is expected to succeed is BitTorrent Live, presented by BitTorrent Inc on May 17, 2016. BitTorrent Live is a streaming video platform that will opperate on TVs, smartphones and computers. Part of the reason that BitTorrent can offer free streaming is that it is using P2P technology similar to the one used by its file sharing client to distribute its streams. That way, the company does not have to pay for content delivery networks, and live streams are also less likely to fail under high demand. So it is expected that more content providers and distributors will adopt P2P as a distribution mechanism in a near future.

2.2.1 Three Generations of P2P Networks

The operation of the P2P networks has changed over the years, trying to adapt (and survive) to several different problems (even legal), generating up to three different generations.

First generation

First generation of P2P networks are known as *centralized P2P networks* because they depend on centralized servers to perform some functions, typically a centralized directory to find resources. This type of P2P networks suffer from a single point of failure, since the network may stop working if the central server goes down. The most widely known example of this kind of networks is the initial music sharing service Napster [36]. Users asked to a central server where to find one file. The server answered with some addresses of nodes that had the file. Finally, the user downloaded the file directly from the given node. The court ordered to close it for legal issues.

Second generation

The second generation of P2P networks appeared to avoid the above vulnerability, but they needed two phases to achieve the expected success. The first attempt was to design *pure decentralized P2P networks*, which use a P2P scheme in all their processes, and there is no central server at all. They are characterized by the arbitrariness of the links between nodes, and by the use of flooding of messages to search resources. Unfortunately, the use of broadcasting techniques limits their scalability, mainly because they generate congestion or infinite loops. Moreover, searching processes are not deterministic, and they do not guarantee that an unpopular file will be found. The typical example of such networks is Gnutella [18]. The second attempt to improve the P2P networks was to introduce some degree of centralization leading to the *hybrid P2P networks*, where some nodes (*supernodes*) manage certain extra functions. These networks do not suffer from a single point of failure, and searches perform better because they are partially centralized. However, they are still partially vulnerable if some of these supernodes are attacked (or closed), like happened some years ago with the file sharing network eDonkey2000 [12].

Third generation

Finally, the third generation of P2P networks emerged. They are totally decentralized networks, so there is not a point of failure, but with a certain structuring of resources. For this reason they are called *structured P2P networks* or also *structured P2P overlays*, as they create an *overlay*. An overlay is a logic high-level layer built over an existent network, which is used to structure nodes and connections between them. The objective of this structure is to speed up searches to obtain positive results in a limited number of hops independently of the popularity of the searched resource. Structured P2P networks, or P2P overlays from here on, can provide properties as scalability, fault-tolerance, self-organization, and low-latency. Unlike pure P2P networks, P2P overlays do not allow random connections between nodes. Instead, overlay nodes are connected via virtual links, which are known as *paths*. These paths can be constructed by using different physical links in the lower networks and the way that packets are routed on those underlying networks is not controlled by the overlay. A P2P overlay routing protocol uses the logical identifiers of the nodes (*nodeIDs*) to decide the routing, instead of using directly their IP addresses.

The most typical P2P overlay protocols (CAN [78], Chord [83], Pastry [80], Tapestry [86], Kademlia [75] or BitTorrent [66, 73], among others) are implemented using a DHT [77], which stores $\{key, value\}$ pairs together with the nodeIDs creating a virtual space. A *value* can be a certain resource (for instance, a file), or the way to reach this resource within the overlay (a pointer), and *keys* are used to locate these resources into the network (for instance, the hash of the file name). Moreover, DHTs are divided in subtables, which correspond to a zone of the virtual space, and these subtables are assigned to different overlay nodes. So each node is responsible for a zone, and hence it is responsible for the $\{key, value\}$ pairs contained in that zone (storing contents, or pointers to them, and routing messages). Usually, a zone is assigned to the node whose nodeID is numerically close to the keys stored in the corresponding subtable of the DHT. So if a node wants to download certain content from the overlay, it will send a query message towards the key in the hope that some node with information about that content will receive the message by proximity between the key and its nodeID. If so, the receiver will answer to this query by sending the corresponding file or the pointer to this file if resources are stored in the owner nodes. Therefore, it is obvious that the location of the nodes in the virtual space is directly related to their nodeIDs, and this deserves special attention.

The most widespread P2P overlay is the Kad network [82], a non-commercial file sharing service based on the Kademlia DHT routing protocol and implemented by the popular eMule [13] and aMule [1] clients (among others), and BitTorrent, another non-commercial file sharing service implemented by tens of open-source, freeware, adware or shareware applications [70].

2.2.2 Identity Management in Existing P2P Overlays

When learning P2P, it is typical to start by classifying the networks according to its architecture, distinguishing between centralized, pure (totally decentralized) and hybrid architectures. And then to deepen the study of P2P overlays, the structured pure P2P networks. In this Section we summarize the working of some typical P2P overlays: CAN, Chord, Pastry, Tapestry, Kademlia and BitTorrent. We analyze how these P2P overlays work, emphasizing in the aspects related to the identity management, access control and bootstrapping. In a P2P overlay, bootstrapping refers to the process by which one or more internal nodes (bootstrapping nodes) provide initial configuration information to newly joining nodes so that they may successfully join the network.

2.2.2.1 CAN (Content-Addressable Network)

CAN [78] uses a virtual d-dimensional Cartesian coordinate space on a *d-torus* to store $\{key, value\}$ pairs. To do so, a *key* is deterministically mapped onto a point p in the coordinate space using a uniform hash function. The corresponding $\{key, value\}$ pair is stored in the node that owns the zone within which the point p lies. Each node inside the overlay is responsible for a zone and keeps information on its immediate neighbors. Nodes in the CAN network do not have a nodeID. Instead, they are directly identified by their assigned zone within the virtual space.

Figure 2.3 shows the bootstrapping phase in CAN[1]. Let us consider that a newcomer node N wants to join the CAN network. To do so, it must contact with an internal node (bootstrapping node) B which will guide it until the bootstrapping phase finishes. Then, N randomly chooses a point p in the CAN virtual space, and sends a "join" request towards that point p using the node B as relay. Since B is an internal node, it can use the CAN routing mechanism to forward this "join" message until it reaches the node that manages the zone in which p lays (manager node M). Then, M splits its zone and assigns a half to N, transferring all the $\{key, value\}$ pairs located in that half to N. N also learns from M the IP addresses of its close neighbors, and with that information N can now generate its own routing and neighbor tables.

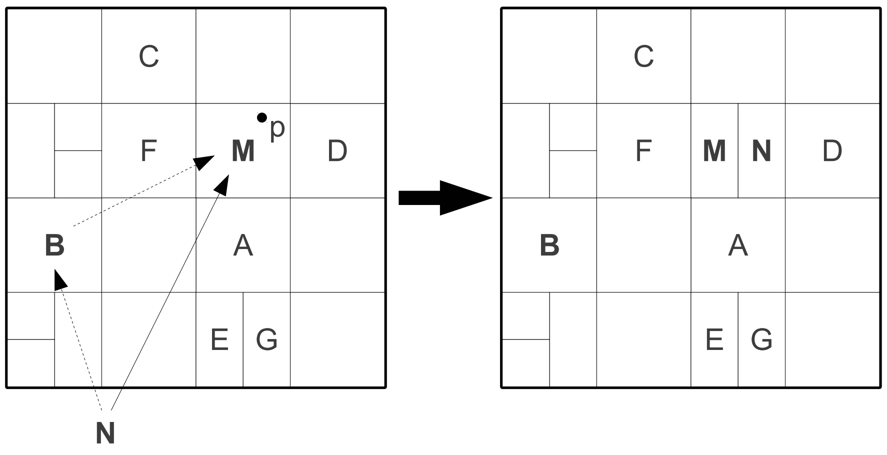


Figure 2.3: CAN bootstrapping phase.

Finally, we must mention that the authors of CAN do not define any mechanism to control who joins the network, so bootstrapping nodes allow any newcomer to join the network.

2.2.2.2 Chord

The virtual space of Chord [83] is circular. NodeIDs and resources (*keys*) are ordered according to a circular identifier that uses a modulo 2^m operation. Each $\{key, value\}$ pair is stored in the *successor* node of the *key* k (denoted as *successor(k)*), i.e., in the first node whose nodeID is equal to or follows the corresponding *key*. Values can be directly

[1]For simplicity, in this figure we draw the virtual space as a flat plane, but remember that the real space is a *d-torus*.

resources, like a file, and the *keys* of the resources are generated using a hash function over their names. NodeIDs are also constructed using a hash function, specifically over the users' IP addresses. The goal is to balance the load of the overlay between all the nodes. To ensure this, hash functions used must have the property that their outputs are equidistant between them with high probability.

In Figure 2.4 we can see an example of an identifier circle modulo 2^3 in which there are three nodes (nodeIDs 0, 1 and 3) and three resources (*keys* 1, 2 and 6). In this particular example, node 1 stores the $\{key, value\}$ pair corresponding to the *key* 1, node 3 stores the pair corresponding to the *key* 2, and node 0 stores the pair corresponding to the *key* 6.

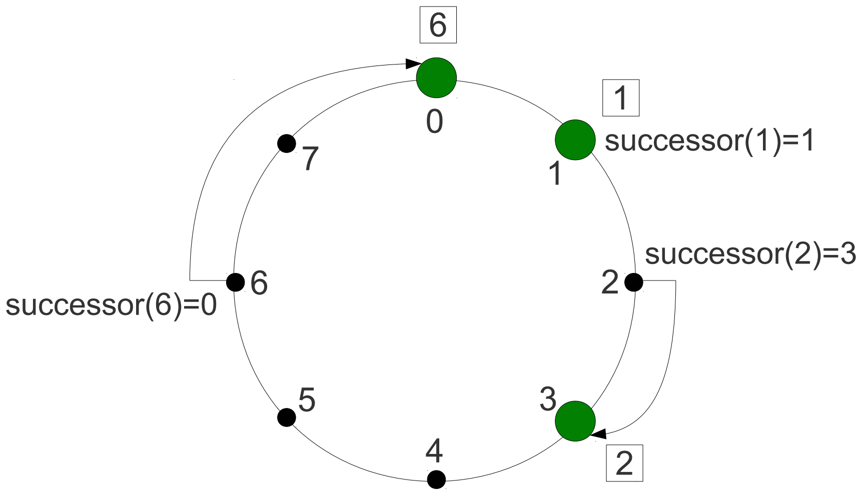


Figure 2.4: Example of a Chord identifier circle modulo 2^3.

A bootstrapping node B helps the newcomer N during the joining process, mainly to initialize its state and add itself to the existing Chord identifier circle. N contacts with the node that currently manages the zone that should be transferred to it (manager node M). This is possible since B is an internal node and it can use the routing mechanisms of the overlay to lookup M. Then, M transfers the corresponding $\{key, value\}$ pairs to N and also shares its routing information to help N to construct its routing table. Neighbors are also informed about the presence of N to properly re-establish the overall routing of the overlay.

Regarding the access control during the bootstrapping phase, the authors mention that this task could be delegated to an external server or even to an internal node, but they do not explicitly define any mechanism.

2.2.2.3 Pastry

In CAN or Chord, the overlay is responsible for storing resources, i.e., a creator node (a node that wants to introduce a resource) delivers resources to the overlay (in particular to a manager node) which is responsible for its storage and management in a structured way. Instead, Pastry [80] is only responsible for publishing the location of resources (in a structured way), but this resources continue being stored in the creator node. The virtual space of Pastry is also circular, i.e., an *identifier circle* modulo 2^m with $m = 128$ bits, and *keys* are computed as the digest of the name and the owner of the resources concatenated. The location of a resource is stored in a node if its nodeID is the numerically closest to the *key*. To lookup resources and route messages, each node not only manages a *routing table*, but also a *neighborhood set* and a *leaf set*. All these state tables are used to route messages in a small number of hops taking into account the geographic location of the nodes, unlike CAN or Chord.

The bootstrapping phase of Pastry is quite similar to that of Chord, and nodeIDs can be computed in two ways, using a hash function over the node's public key or the node's IP address. In most cases, this last option is the one used. Like in Chord, the use of a hash function assures that the computed nodeIDs will be uniformly distributed in the identifier circle, and with high probability, diverse in geography, ownership, jurisdiction, etc.

Regarding the user access control at bootstrapping phase, the authors do not also describe any mechanism to do that.

2.2.2.4 Tapestry

Tapestry [86] uses routing and location schemes similar to the ones presented by Plaxton et al. in [77]. When a creator node wants to publish a certain resource (for instance, a file) in the overlay, it sends a message towards the *key* of that resource, in the hope of reaching the responsible node of that *key*. Specifically, this node is responsible for storing the $\{key, value\}$ pair of that resource, and it is chosen because its nodeID is the numerically closest to the *key*. As Pastry does, Tapestry publishes the location of resources instead of storing the resources themselves, and the routing scheme also takes into account the location within the network to route messages. In Tapestry, multiple replicas of each $\{key, value\}$ pair are also created and stored in *replica*, or *surrogate* nodes. So, if a replica node is found prior to reach the original responsible node, this node will provide the location of the desired resource. This mechanism not only increases flexibility and saves bandwidth, but also alleviates the problem of a single point of failure.

Regarding bootstrapping phase, a newcomer N also contacts with a bootstrapping node B, which will start routing the "join" message towards the nodeID of N. Then, N informs the relevant nodes of its presence to update their neighbor maps. Once all potential neighbors are located, the relevant $\{key, value\}$ pairs are copied to N.

NodeIDs and *keys* may be distributed in the virtual space randomly using a hash function, but authors do not state which information should be used. The same happens with the user access control system; authors do not define any mechanism to carry on this task.

2.2.2.5 Kademlia

Kademlia [75] is similar to the other P2P overlays in the sense that $\{key, value\}$ pairs are stored in nodes with nodeIDs "close" to the *keys*. However, this closeness relies on a different notion of distance. Authors define the distance between two points in the key space as their bitwise exclusive-or (XOR) interpreted as an integer. NodeIDs and *keys* (generated using a hash function) have a length of $B = 160$ bits and *values* are pointers to the files. The $\{key, value\}$ pairs are replicated in several nodes.

Kademlia treats nodes as leaves in a binary tree, in which the position of each node is determined by the shortest unique prefix of its nodeID. For any given node, the binary tree is divided into a series of successively lower subtrees that do not contain the node itself. To organize this knowledge, nodes use *K-buckets*, which represent subtrees with a group of a maximum of K contacts, and divide them into more subtrees in the case that the *buckets* already contain K contacts. The *K-buckets* are organized by the distance between the node and its contacts. Specifically, for *bucket* j, where $0 <= j < B$, it is guaranteed that

$$2^j <= distance(node, contact) < 2^{j+1}.$$

This means that the *bucket* zero has only one possible member, the *key* which differs from the nodeID only in the low order bit. And on the other hand, if nodeIDs are uniformly distributed, it is very likely that half of all nodes will lie in the range of *bucket* $B - 1 = 159$.

An example of this distribution can be seen in Figure 2.5. This example shows the three smallest subtrees (*buckets*) constructed by the node with nodeID 111...110 (red point). These *buckets* have the prefixes 111...111, 111...10 and 111...0 respectively. The Kademlia protocol establishes that every node should know at least one node in each of these subtrees to locate any other node in a prefix basis. Otherwise, its location would not be guaranteed.

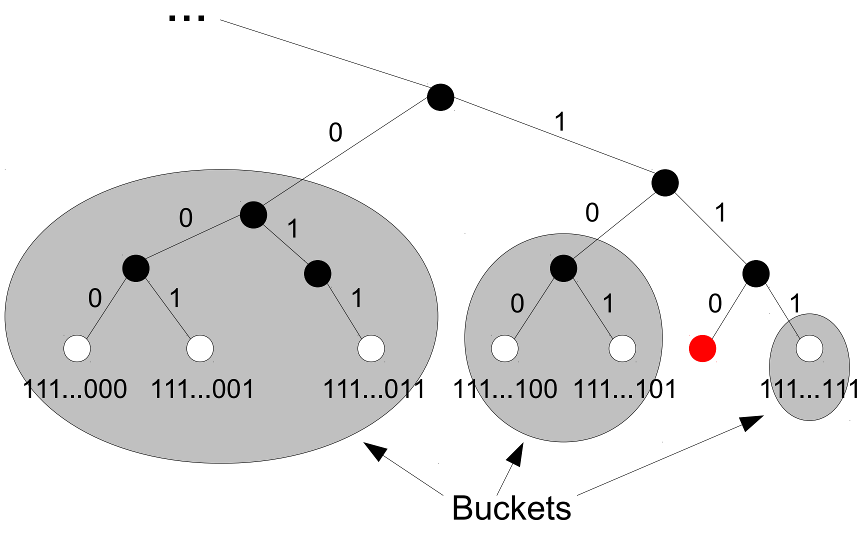


Figure 2.5: Generation of *buckets* in the node 111...110.

In Kademlia, every message transmitted by a node includes its nodeID, permitting the receiver to record the sender's existence if necessary. In this way, this topology has the property that every message exchanged conveys useful contact information. Using this information, a node can send parallel asynchronous query messages which tolerate node failures without imposing timeout delays on users. A nodeID-based routing algorithm lets anyone locate servers near a destination *key*.

Regarding the bootstrapping phase, first of all, a newcomer N must calculate its own nodeID in the 160-bit key space, generated by hashing its IP address as in Chord, and insert the nodeID of the bootstrapping node B into the appropriate *K-bucket*. Then, N must perform a node lookup for its own nodeID and refresh all *K-buckets* further away than its closest neighbor. During this last process, N populates its own *K-buckets* and inserts itself into other nodes' *K-buckets* as necessary. Unfortunately, authors neither include any mechanism for controlling the user access.

2.2.2.6 BitTorrent

BitTorrent [66] is a second generation P2P protocol for distributing files, where resource lookup is performed on a *web server* and resource dissemination is managed by a *tracker*. A tracker is a special type of server that assists in the communication between peers using the BitTorrent protocol Trackers neither store resources nor participate in their exchange, they only coordinate the set of *peers* that participate in the resource exchange (*swarm*) and keep track of the active peers of the set. Web servers index metadata files (*.torrent*) which describe the resource exchanged by a swarm. This includes information such as the address of its tracker and the names, sizes and checksums of all *chunks* in which the resource is split.

In the traditional BitTorrent network, whenever a user wants to download a resource, it contacts to a tracker to obtain the list of peers that make up the swarm. And then, it contacts to some peers of the list to download all chunks that make up the resource using the peer wire protocol, implemented over TCP. A peer communicates with the tracker regularly while it is part of the swarm to inform about the volume of bytes it has downloaded or uploaded, and the tracker responds with the list of active peers at each time.

Additionally, few years ago, Loewenstern and Norberg proposed to introduce a decentralized tracking system [73] to avoid single points of failure and improve the performance of the tracking process, where any peer can act as a tracker (trackerless torrent). This new tracking system was implemented using a DHT based on Kademlia (Mainline DHT, MDHT) to store and locate information about which peers hold what resources. Other DHT, also based on Kademlia, was also proposed but it is only implemented by one client software (Vuze [59], previously Azureus) and is not compatible with the MDHT, as they have non-trivial differences. For these reasons we only take into account the solution that uses the MDHT.

In the past, a BitTorrent client only included a peer, instance of the program to which other clients connect and transfer data. But now, most clients also include a *MDHT node*, another instance used to find peers in a decentralized way.

As Kademlia does, MDHT stores nodeIDs and *keys* (*infohashes* from here on) as leaves in a binary tree. Whenever a user wants to download a resource in DHT BitTorrent, firstly it must decide which nodes to contact to get the peers list to download from using her peer instance. For that, the node instance uses the XOR distance metric between the *infohash* of the resource and the nodeIDs of the nodes in her own routing table. Note that a node knows many nodes with nodeIDs "close" to its own, but it has few contacts with nodeIDs that are "far". Then, the original node contacts some nodes with nodeIDs closest to the *infohash*[1] and obtains information about peers currently uploading that resource. Obviously, if some contacted node does not know about peers for that resource it must respond with information of the nodes in its routing table that are closest to that *infohash*. This process finishes when the original node cannot find any closer nodes. Finally, the node stores in appropriate buckets the peers' information for a small number of the responding nodes with nodeIDs closest to the *infohash* of the resource. The protocol which allows peers within the same swarm to share their peer lists is the peer exchange (PEX) protocol, implemented over UDP.

Regarding the nodeID assignment and the user access control, unfortunately, at first, little attention was paid to security. In the BitTorrent context, nodeIDs were generated at random from the same 160-bit virtual space as *infohashes* and no access control mechanism was proposed.

The Bootstrapping phase is similar to that performed in Kademlia, with the difference that the nodeIDs are randomly selected. In addition, unlike Kademlia, the MDHT only stores each *infohash* on one node, resulting in no easy way to unambiguously determine which peers are responsible for a certain resource, complicating any replication or migration strategy.

2.2.2.7 JXTA

Juxtapose [25] is a set of open protocols that enable the creation and deployment of P2P networks. JXTA protocols enable users to discover and observe other nodes, to communicate among them, or to offer and localize resources within the network. In order to access those resources, JXTA completely relies in the usage of advertisements published by the resource owner. JXTA-Overlay [85] extends such protocols in a framework which increases the reliability of JXTA-based applications and supports group management

[1]An *infohash* is calculated as the hash of the "info" section of the original *.torrent* file.

and file sharing. This framework differs from above P2P protocols because it introduces the concept of peer group, one of its main features. The overlay network is divided into hierarchical groups of nodes, which offer a context for accessing services. Users are organized into different overlapping groups, so only members of the same group may interact between them. Peers must join the group that offers the services in which they are interested.

Brokers are special nodes which control access to the network, taking care of user authentication as well as helping client nodes interact between them by propagating their related information. Brokers are very important since they exchange information about all client nodes, maintaining a global index of available resources, which allows all nodes to find network services. Brokers also act as beacons used by client nodes which have recently gone on-line to join the network.

Unfortunately, the design focus on JXTA-Overlay was completely concerned with system performance, with the only exception of the user authentication. Users are authenticated using pairs username and password before they join the network, which are issued without any control.

Chapter 3

Tested solutions

Contents

3.1 CAN **22**

3.2 Chord **24**

3.2.1 MIT PDOS Chord implementation 24

3.2.2 Open Chord 26

3.2.3 Chordial 29

3.2.4 JChord 30

3.2.5 Chordjerl 30

3.2.6 jDHTUQ 30

3.3 Pastry **33**

3.3.1 FreePastry 34

3.3.2 PAST 36

3.4 Tapestry **39**

3.4.1 Tapestry project 39

3.4.2 Chimera 40

3.4.3 OceanStore 41

3.5 Kademlia **43**

3.5.1 OpenDHT 43

3.5.2 Pydht 45

3.5.3 TomP2P 45

3.5.4 Hive2Hive 46

3.5.5 MaidSafe 48

3.6 JXTA **51**

3.6.1 JXTA/JXSE 51
3.7 Edonkey 54
3.7.1 aMule 54
3.8 Routing algorithm simulators 58
3.8.1 Overlay Weaver 58
3.8.2 Entangled 61
3.8.3 OverSim 63
3.8.4 P2PSim 65
3.8.5 PeerSim 66
3.8.6 PeerFactSim.KOM 67

Since the main objective of this project is to run a P2P overlay over `simctl` for future work and lab sessions, I will look for the maximum available implementations that accomplish the requirements. I will try to find projects or ended software that are faithful to the real algorithms, being able to run in a local network between different computers and that can be executed by console, that is, with no graphical user interface (GUI).

In this chapter I will list some of the tested codes and software, explaining its advantages and issues, and checking if they meet all the requirements.

3.1 CAN

CAN is one of the original four distributed hash table proposals, introduced concurrently with Chord, Pastry and Tapestry. It was first proposed in 2001 at the University of California by Sylvia Ratnasamy, Paul Francis, Mark Handley,Richard Karp and Scott Shenker. But, although having a number of interesting properties, CAN is remarkably unpopular compared to Chord and Pastry, with respect to implementations.

Googling an existing implementation of CAN will most likely yield nothing. All few implementations are codes from Github and were developed by university students as projects for Distributed Systems courses. Therefore, they are not faithful to the original paper [78], since most of them are incomplete and limited to 3 or 4 peers. Moreover, these codes are designed to run different codes on different computers, it is, one machine creates the network and all the others work as clients. Therefore, if the bootstrap node dies the network breaks, and this is not the behaviour of the original

CAN. After testing most of those implementations, none of them meet the requirements to be used on `simctl`.

CAN was used in large scale storage management system such as Farsite, Publius or PIER project, but nowadays these projects are abandoned and their codes are not free source.

3.2 Chord

The Chord protocol dynamically constructs robust and scalable overlay networks that map a given key to an active node. The MIT PDOS Chord implementation [83] has served as a reference implementation of Chord, and over the years has accumulated many tweaks and improvements.

When looking for a Chord implementation, several codes appear on GitHub and other repositories, but since the original MIT PDOS Chord implementation code is available, it was the first option to test. In this section only some of the tested implementations are listed. I spent a long time trying to compile and test many other implementations from GitHub. They are not listed here since they were toy projects, not faithful to the Chord algorithm or were unfinished. Other implementations only simulated the ring on a single PC, or were projects from university students for some Distributed Systems courses that used some tricks to make its implementation easier. For example, some implementations made use of a common list where all the nodes registered before joining the ring, so that all nodes consulted which nodes were available.

Most of the implementations were not documented and it was hard to test them. Moreover, some of them were written in Erlang, Python or Golang programming language. As I have no knowledge of these languages, I tried to take a look at their codes and to install them, but I finally gave up and kept looking for another implementation that met the main requirements and work exactly as it is explained in the original paper.

3.2.1 MIT PDOS Chord implementation

This software [35] was developed at the MIT Laboratory for Computer Science in the Parallel and Distributed Operating Systems Group, and the authors may be contacted at chord@pdos.lcs.mit.edu. MIT PDOS Chord is implemented in C++ and works as described in the original publication [83]. MIT Chord is completely decentralized and symmetric, and can find data using only log(N) messages, where N is the number of nodes in the system. The source code provides a filestore tool called Cooperative File System (CFS) that can be used to store files in the Chord ring. The software breaks a file into blocks, names each block with different key and then distributes them.

The code was written in 2001 and was under active development until 2008. Shortly after it was abandoned. It was originally located at `http://pdos.lcs.mit.edu/`

chord/ but later in 2012 it migrated to Github. In 2013 the authors communicated that at that point no official release of Chord was available, although the complete development history could be found at the source repository.

There is a Chord HowTo [34] (last edited in 2012) that describes how to download and compile the software. At that time the HowTo was complemented by some snapshots, but nowadays all links are dead. To run the software some tools are required:

- SFSlite 0.8.16pre1 or newer.
- A recent GCC (4.0ish works well).
- Autoconf, automake, GNU make, etc.
- Berkeley DB v4.x.

As the source code was developed some years ago, it is strictly necessary to use the recommended versions. Some tools that are installed in Ubuntu 14.04 have to be downgraded, and some deprecated versions have to be manually installed from scratch. After testing different versions and getting lots of errors, the following tools are supported and installed without errors:

- Gmp-4.1.4/gmp-4.2.3.
- Flex and Bison.
- g++4.1.2 and gcc-4.1.2.
- Berkeley DB 4.5.20.
- SFSLite-0.8.16/SFSLite 0.8.17.

Once we have installed the prerequisites, we should proceed to install the Chord source code. It is available via Git [35] by running:

```
% git clone git://github.com/sit/dht chord-0.1
```

Once we have cloned the entire history we can select any version. In GitHub there is only the code of the master branch, so I try the proposed version in the HowTo section by using the command:

```
% git checkout 9ab9b473afd75ea254a9671df9dd38ee61a95262
```

When trying to run the software, the compilation breaks and shows some errors.After a long time looking for it on the Internet, I did not find anyone with the same problem using this code. The few webs that show their issues and errors related to the MIT Chord were dated on 2010 or fewer.

While searching the error, I found a project [61] from the Computer Engineering Department in Santa Clara University that tried to improve the code on August 2013. In the annex, they mention which tools they used to run the MIT Chord, and the version of the code, which is different from the GitHub's:

```
% git checkout ea40c6690b
```

After reinstalling all the tools that they used and selecting their MIT Chord version, another error appears. Although trying all the solutions proposed on Internet forums, the code does not compile.

During my research I found an email repository [60] with conversations between users and Dr. Emil Sit, one of the authors of the software. In one mail [81] from the author written in 2012, he says: "I last developed this in 2008, so if you install Ubuntu 8.04, you may have some success compiling and linking. There is no active development on this software and it is not supported in any way."(`sit@mit.edu`). Then I found one mail [74] from the user `Tri Nguyen Phi Minh` where he mentioned that he could compile the software successfully using Ubuntu 8.02.

Probably all that errors are due to the operating system, since I tried to install it on Ubuntu 14.04 (launched on 2014) and Ubuntu 12.04 (launched on 2012). I did not try Ubuntu 8.04 since it definitely died on 2013 and it has no support [57].

In a last attempt I contacted by mail with a user who had sent emails to the author in 2012, and with one GitHub user that contacted the author in GitHub in 2015. None of them could install the software and finally left. Therefore, this option is not feasible.

3.2.2 Open Chord

Open Chord [40] is an open source implementation of the Chord distributed hash table. It is available for free under GNU General Public License (GPL) and was developed by the Distributed and Mobile Systems Group of Bamberg University.

Open Chord provides the possibility to use the Chord distributed hash table within Java applications by providing an API to store all serializable Java objects within the distributed hash table. On the other hand, it provides a console-based experimentation environment to explore the functionality of a Chord network, allowing to create a Chord ring between different nodes and storing data between them.

The Open Chord console is intended for testing purposes in order to interact with the Chord network. Using the console, one peer can create a Chord network. Then, the other peers can join the network indicating the address and the port of a bootstrap node (any peer of the ring). Once the ring is running, any peer connected to it can store a key-value pair, where the key and the value are strings. The console provides other functions to each node such as to consult all the data stored in itself, to obtain the value corresponding to a specific key or to delete one pair of the network. Moreover, each node can consult its finger table to see its successors and predecessor peers, and can leave the network when it wants.

In the configuration files several parameters can be configured such as the length in bytes of displayed IDs or the number of successors of each node, it is, the number of replicas that are created from a data value. By default, this value is set to 2, which means that if one peer inserts some data it should be stored in 3 nodes (the responsible node and its two successors). If one of them fail, during the next stabilization interval the new responsible node that contains this data will replicate it, so that there are again 3 nodes that store the data.

There are two possibilities to create/join an Open Chord network:

- Local implementation (oclocal): the network is created in one Java Virtual Machine (JVM) using a single computer. Many local peers can be created with help of the console.
- Socked-based implementation (ocsocket): each peer is located on a different computer or runs a different console in the same computer. This implementation facilitates reliable communication between Open Chord peers based on TCP/IP sockets.

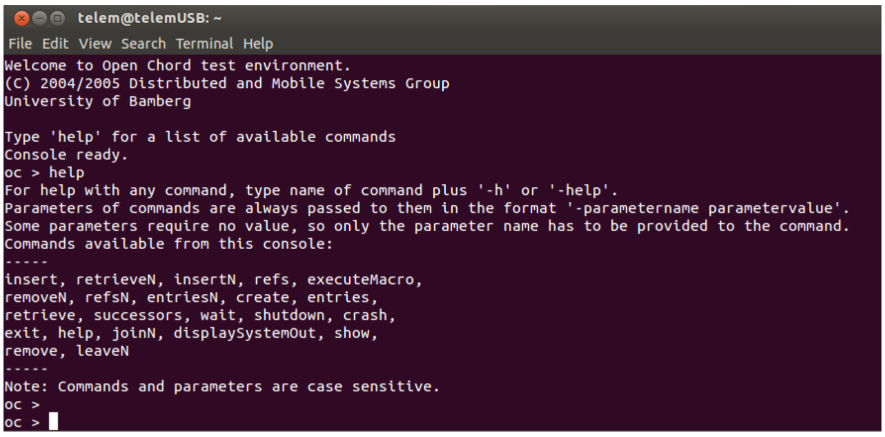
After viewing the Open Chord facilities, it looks interesting for the project. The socked-based implementation could be used in order to create a Chord ring between nodes located on different virtual machines connected by a virtual local area network.

The source code is located at [41]. In the webpage we can find 6 different releases, being the most current one Open Chord 1.0.5 (updated in 2008). The downloaded file provides a pdf document with a user's manual [67] that describes what is required to and how to install and run Open Chord.

After installing the prerequisites, building and compiling the software, I first test the local implementation which works in a single computer running in a JVM. All the options and functions work well, so then I move to the socked-based implementation.

From one node I create the Open Chord ring and try to connect to it from a peer located on another virtual machine of the same virtual LAN. When the second node tries to connect to the ring remotely, several errors appear telling that it is not possible to create the connection. After many frustrated tests, I decide to install the previous release Open Chord 1.0.4. The same happens, nodes located in other computers can not connect remotely to the ring.

After several tests, I execute two nodes on the main computer, but in different terminals. One node creates the ring and the other one can connect to it with no errors. After that, I execute another peer on the other machine and it can connect remotely to the ring without problems. It seems that one remote node needs at least two peers in the ring to connect successfully. Then, I execute more peers in different virtual computers to test the Open Chord network.



```
Welcome to Open Chord test environment.
(C) 2004/2005 Distributed and Mobile Systems Group
University of Bamberg

Type 'help' for a list of available commands
Console ready.
oc > help
For help with any command, type name of command plus '-h' or '-help'.
Parameters of commands are always passed to them in the format '-parametername parametervalue'.
Some parameters require no value, so only the parameter name has to be provided to the command.
Commands available from this console:
-----
insert, retrieveN, insertN, refs, executeMacro,
removeN, refsN, entriesN, create, entries,
retrieve, successors, wait, shutdown, crash,
exit, help, joinN, displaySystemOut, show,
remove, leaveN
-----
Note: Commands and parameters are case sensitive.
oc >
oc >
```

Figure 3.1: OpenChord interface with available commands.

Open Chord 1.0.5 has some problems when one of the remote peers try to store data in the ring, so during the rest of the tests I work with Open Chord 1.0.4. I test all the options and the software works correctly. Each node can insert, remove or consult the data stored on the network. Finger tables in each node are useful to understand the behaviour of the ring when a search is done. Each node can also consult its predecessor and successors, as well as its NodeID. It is also interesting to see how the data is replicated when one node is disconnected from the ring.

Despite these problems, the software can work with several nodes and is quite stable. Looking at the web, it seems that this software was also abandoned, since there is no activity since 2009, so no new release will be launched.

This software meets the main requirements proposed in my objectives: it can work with no GUI, implements the Chord algorithm faithfully to the original paper and works over a local network with several computers. Then, I will try to install and use it over `simctl` (Section 4.1).

3.2.3 Chordial

Chordial [6] is an open source implementation of MIT's Chord DHT algorithm in the Erlang programming language. It was last updated on 2009 by the GitHub user `mattwilliamson` but it is still under pre-alpha development, so it looks that this software was also abandoned.

Chordial allows key lookups over real networks of computers, which makes it interesting to test. The maximum number of nodes in a chord ring is 2^m where m is the number of bits in the hashing algorithm used. As Chordial uses sha1, which is 160 bits in length, it allows for 2^{160} nodes. The maximum number of keys is also 2^m. It seems that Chordial is quite faithful to the original Chord algorithm, watching for unresponsive/dead and new peers, rebalancing finger tables in an efficient way and replicating keys to predecessors when needed.

After downloading and compiling the code I try to execute some commands in the Erlang shell to stablish a Chord network between 3 nodes, each one running in a different machine. The software creates the network and it immediately shows the successor and predecessor in each node. It shows too much information and is not quite intuitive. Moreover, it only allows to look up keys, so it does not allow to insert data such as files or strings in a key-value mode. It is possibly due to the fact that the software was still under pre-alpha development, so it is unfinished. Then, it is not a good option to implement over `simctl`, since it is unuseful to study the behavior of Chord algorithm.

3.2.4 JChord

JChord [22]is a simple implementation of Chord protocol written in Java by the GitHub user `Scorpiovn` in 2008. JChord was developed for the purpose of understanding, teaching, simulating, and developing new algorithms over Chord.

This implementation is able to simulate a Chord ring with thousands of peers and draw an image of the ring with all peers, showing with lines in which nodes the information is stored. The drawback is that the original lookup and stabilization operations are implemented in a single VM and a single thread in a single PC, so it cannot run over several machines in a real network. Therefore, this implementation does not meet the main requirements to implement it over `simctl`.

3.2.5 Chordjerl

Chordjel [7] is an Erlang implementation of the Chord distributed hash lookup protocol. It was written in Erlang programming language by the GitHub user `jashmenn` and, despite being in Pre-Alpha status, it was last updated in 2009, so this project has been also abandoned. The goal of Chordjerl is to be a reusable Erlang implementation of the Chord distributed key lookup protocol.

This code has no documentation or explanation about how to use it, so I spent a long time trying to test it without success. Chordjerl provides support for just one operation: given a key, it maps the key onto a node. It seems that it does not have any function or console to interact with the Chord network, but it is a library that allows to reuse its functions to create other projects.

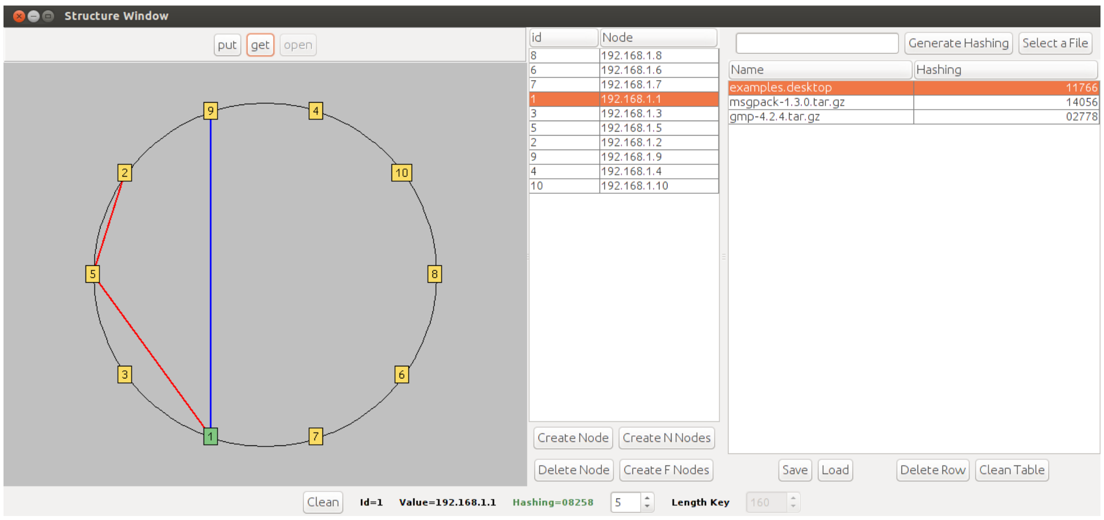
As I have no knowledge of Erlang programming language, I preferred to keep looking for another implementation instead of reusing or modifying this project.

3.2.6 jDHTUQ

jDHTUQ [23] is a peer-to-peer DHT system based in Chord algorithm, but built to generalize the implementation of P2P DHT system. Its two fundamental services are to put and get resources. The software of the project is available on SourceForge, and was last updated in 2014.

After downloading and decompressing the file, several directories are created. They provide all the libraries needed for well function of jDHTUQ, as well as some configuration files in XML format. jDHTUQ is also in JAR format, so the source code is not available. It is also provided an unfinished pdf document with some documentation of jDHTUQ. After taking a look at the documentation I download and run the software.

DHTUQ makes use of a GUI and cannot be used from a console, so it cannot be used in `simctl`. However, it could be a useful tool for learning, so I will test it.



id	Node
8	192.168.1.8
6	192.168.1.6
7	192.168.1.7
1	192.168.1.1
3	192.168.1.3
5	192.168.1.5
2	192.168.1.2
9	192.168.1.9
4	192.168.1.4
10	192.168.1.10

Name	Hashing
examples.desktop	11766
msgpack-1.3.0.tar.gz	14056
gmp-4.2.4.tar.gz	02778

Id=1 Value=192.168.1.1 Hashing=08258 5 Length Key 160

Figure 3.2: jDHTUQ in Structure mode.

Before executing the application, some parametres must be set in the configuration files. When running jDHTUQ, a window with two options is shown: Structure and Network. These are the two modes of executing the application, and they depend on which classes handle the communication.

- Data structure. Several nodes can be executed on a single host. The GUI is used to plot a circle and to represent each peer on the ring. It allows to insert (put) or retrieve (get) files from different nodes.
- Network. Using this mode, only one node can be executed in a host. It works over a real network using the UDP protocol for lookup services and TCP for storage services.

First I try the Structure mode. When clicking "Structure", a new window appears with an empty circle and some buttons. It allows the user to add nodes to the network one by one by giving a name for each one, to create a determinate number of nodes, or to create several nodes by reading them from an existing file. Once a name is given (a name, an IP address, etc) it obtains the hash of the name to create its identifier. So I add 10 nodes to the network, giving as a name an IP address. It immediately orders the nodes by its identifier. Then, from one node, I test the put option. It allows to upload files to the network, which will be later stored in its corresponding node. The hash of the file is calculated and then it is stored in the node which nodeID is equal or immediately greater than the hash of the file. The hash of any file can be generated by the user, so that he can see the identifier of each file and compare it with the nodeID of the node that is storing the file. When a file is uploaded, a new folder with the name of the storing node is created in jDHTUQ directory. All the uploaded files are stored in the folder of their responsible node. When the put or get options are used, some coloured lines are depicted to represent which nodes are consulted (finger tables) and which node will store the file.

After that, I try the Network node. Before running jDHTUQ, I edit the configuration files again to change the execution mode. I execute it in two different machines but they cannot fins each other. It seems that this second option does not work. Moreover, the documentation is very poor, and it is not explained what this function does. Therefore, this software is only useful to simulate several nodes on a single machine and observe how the Chord algorithm stores and retrieves data.

3.3 Pastry

Pastry, along with Chord, was one of the most popular routing technique for distributed hash tables among academics, since it appeared in a large amount of published papers. But if we mean popular among implementers, the picture changes. When P2P evolved to file sharing, other overlays such as Kademlia came out on top due to its much less complex design. Then, most of P2P applications where based on Kademlia, while Pastry was forgotten in the academic field. For this reason, there are a very few implementations of Pastry.

The most cited and used Pastry implementation is FreePastry, which is a free Pastry implementation developed by many members of different Universities and institutions, including the original Pastry's designers. This free Pastry implementation is placed on `www.freepastry.org`. Pastry's authors also designed different projects based on Pastry, such as SCRIBE, PAST, SQUIRREL, SplitStream, POST or Scrivener. Other universities also developed some projects based on Pastry such as PASTA, Herald, Pastiche or DPSR. But all the previous projects were developed in the early 2000, and most of them were abandoned and retired from their corresponding websites, so their source code and documentation is no more available. However, FreePastry is still available and well documented, and a free software release from 2009 can be downloaded.

Apart from FreePastry, there are very few Pastry implementations available on the network. Searching in GitHub and other repositories, some projects use FreePastry facilities to develop chat applications, and other implementations are unfinished or are not faithful to the real algoritm. Other projects are based on FreePastry and add some enhancements, or others such as BTPastry are a torrent search engine based on Pastry. And finally, there are a few Pastry implementations that look interesting but they are not documented at all and do not have any file with a description or a small explanation about how to use it.

What I am looking for is an implementation capable to run in a network between different machines and able to publish some data in the Pastry network, in order to analyse the sent packets and to learn about Pastry.

3.3.1 FreePastry

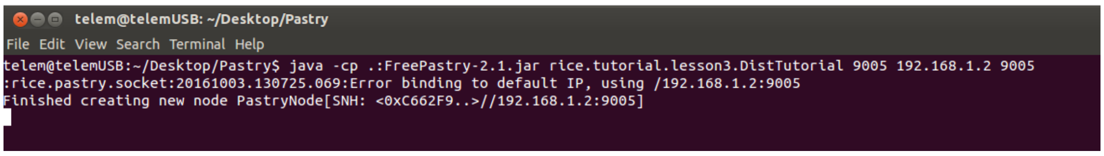
Pastry [80] is a generic, scalable and efficient substrate for peer-to-peer applications. Pastry nodes form a decentralized, self-organizing and fault-tolerant overlay network within Internet. This project [47] was developed in 2001 by several members from different institutions, such as MPI-SWS, Rice University, Purdue University, University of Washington and Microsoft Research. Then, from that project, two implementations of Pastry were born and are currently available for download: FreePastry from Rice University and SimPastry/VisPastry from Microsoft Research.

FreePastry [15] is an open-source implementation of Pastry intended for deployment in the Internet, and a generic, scalable and efficient substrate for peer-to-peer applications. Initially, it was intended as a tool that allows interested parties to evaluate Pastry, to perform further research and development in P2P substrates and as a platform for the development of applications. Later, it evolved to provide a robust and fully secure implementation that is suitable for a full-scale deployment in the Internet.

FreePastry is the most popular and used Pastry implementation, and it has been used for the development of several projects. FreePastry is really well documented, providing several tutorials [17] and some lessons to write the minimal code to create a pastry ring and start learning how to use it. The most important capability of FreePastry is that after compiling the code, we can either run multiple nodes on a single computer (but in separate processes) or if we have multiple computers, we can launch them on different machines as long as the computers can communicate with each other via IP. So it could be used in `simctl` to create a network between the different virtual machines.

After taking a look at the documentation, I download the latest available release FreePastry 2.1 that is dated on March 2009. When downloading FreePastry, two files are provided: the binary distribution and the source distribution. Since I do not need to modify the library, the binary distribution will be enough to test it. So I take a look at the tutorials to test the FreePastry-2.1.jar binary version. First of all, I follow a provided tutorial that shows how to create and run multiple FreePastry nodes within the same JVM. I run the given example code and it successfully creates a Pastry network with 10 nodes. FreePastry also provides a Discrete Event Simulator, and some tutorials that show the user how to simulate an application on a medium sized network (<100.000

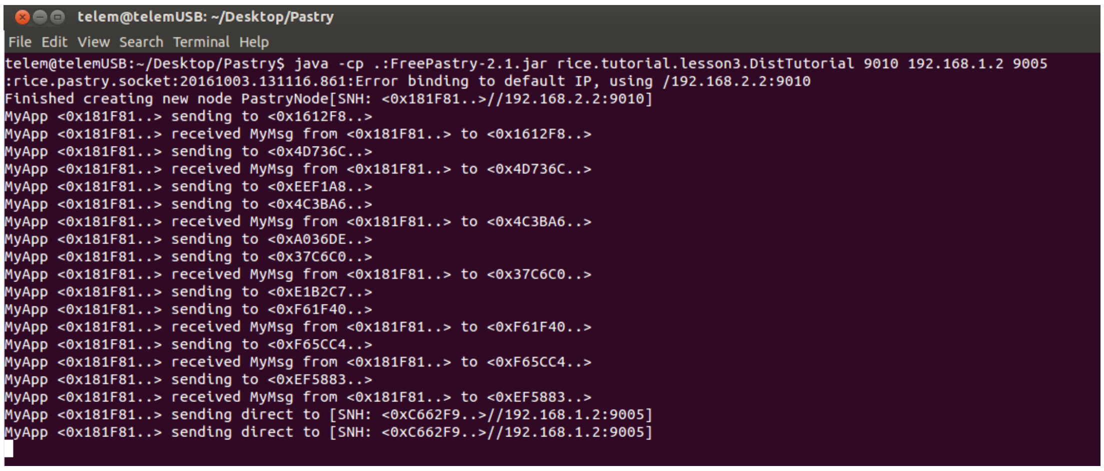
nodes) with a minimum of complications in the code. It also can use a latency matrix to more accurately simulate applications. But I am interested in running FreePastry in a local network between different machines, so I run another code in an Ubuntu 14.04 virtual machine. The code is provided in a tutorial that shows the user how to create and run a FreePastry application. It allows to send and receive messages through the network. When it is started, the node tries to connect to an existing network, but as it is the first peer, it creates the network.


telem@telemUSB: ~/Desktop/Pastry
File Edit View Search Terminal Help

```
telem@telemUSB:~/Desktop/Pastry$ java -cp .:FreePastry-2.1.jar rice.tutorial.lesson3.DistTutorial 9005 192.168.1.2 9005
:rice.pastry.socket:20161003.130725.069:Error binding to default IP, using /192.168.1.2:9005
Finished creating new node PastryNode[SNH: <0xC662F9..>//192.168.1.2:9005]
```

Figure 3.3: FreePastry capture. First node.

Then, from another Ubuntu 14.04 virtual machine I try to connect to the existing FreePastry network by using the first node as the bootstrap node. A new node is created in this second virtual machine and it sends some messages to the node created before.

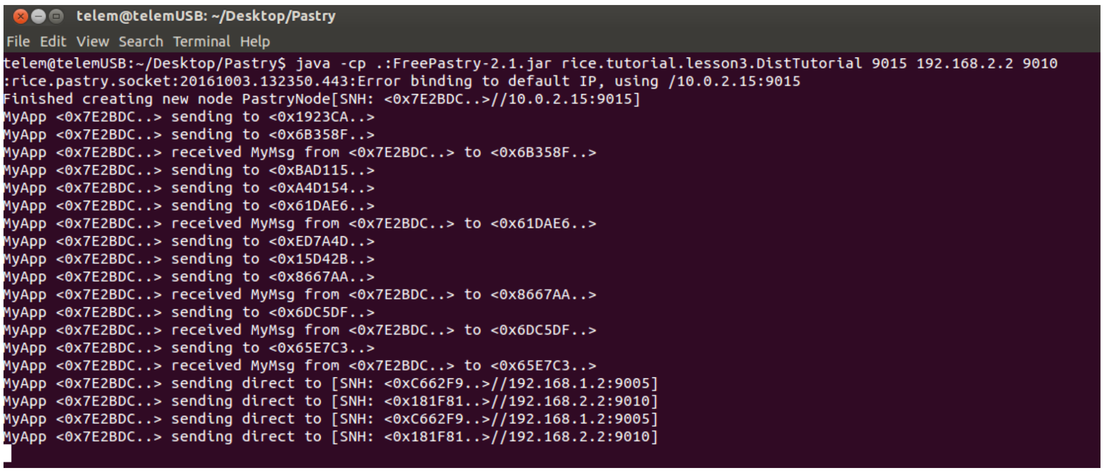

telem@telemUSB: ~/Desktop/Pastry
File Edit View Search Terminal Help

```
telem@telemUSB:~/Desktop/Pastry$ java -cp .:FreePastry-2.1.jar rice.tutorial.lesson3.DistTutorial 9010 192.168.1.2 9005
:rice.pastry.socket:20161003.131116.861:Error binding to default IP, using /192.168.2.2:9010
Finished creating new node PastryNode[SNH: <0x181F81..>//192.168.2.2:9010]
MyApp <0x181F81..> sending to <0x1612F8..>
MyApp <0x181F81..> received MyMsg from <0x181F81..> to <0x1612F8..>
MyApp <0x181F81..> sending to <0x4D736C..>
MyApp <0x181F81..> received MyMsg from <0x181F81..> to <0x4D736C..>
MyApp <0x181F81..> sending to <0xEEF1A8..>
MyApp <0x181F81..> sending to <0x4C3BA6..>
MyApp <0x181F81..> received MyMsg from <0x181F81..> to <0x4C3BA6..>
MyApp <0x181F81..> sending to <0xA036DE..>
MyApp <0x181F81..> sending to <0x37C6C0..>
MyApp <0x181F81..> received MyMsg from <0x181F81..> to <0x37C6C0..>
MyApp <0x181F81..> sending to <0xE1B2C7..>
MyApp <0x181F81..> sending to <0xF61F40..>
MyApp <0x181F81..> received MyMsg from <0x181F81..> to <0xF61F40..>
MyApp <0x181F81..> sending to <0xF65CC4..>
MyApp <0x181F81..> received MyMsg from <0x181F81..> to <0xF65CC4..>
MyApp <0x181F81..> sending to <0xEF5883..>
MyApp <0x181F81..> received MyMsg from <0x181F81..> to <0xEF5883..>
MyApp <0x181F81..> sending direct to [SNH: <0xC662F9..>//192.168.1.2:9005]
MyApp <0x181F81..> sending direct to [SNH: <0xC662F9..>//192.168.1.2:9005]
```

Figure 3.4: FreePastry capture. Second node.

Finally, from a third virtual machine, I create another node using the previous peer as the bootstrap node. This new node sends some messages to the two existing nodes of the FreePastry network.

So, after this simple test, it seems that FreePastry works right in a local network between three different virtual machines, and it runs in console-mode with no GUI. Then,


telem@telemUSB: ~/Desktop/Pastry
File Edit View Search Terminal Help

```
telem@telemUSB:~/Desktop/Pastry$ java -cp .:FreePastry-2.1.jar rice.tutorial.lesson3.DistTutorial 9015 192.168.2.2 9010
:rice.pastry.socket:20161003.132350.443:Error binding to default IP, using /10.0.2.15:9015
Finished creating new node PastryNode[SNH: <0x7E2BDC..>//10.0.2.15:9015]
MyApp <0x7E2BDC..> sending to <0x1923CA..>
MyApp <0x7E2BDC..> sending to <0x6B358F..>
MyApp <0x7E2BDC..> received MyMsg from <0x7E2BDC..> to <0x6B358F..>
MyApp <0x7E2BDC..> sending to <0xBAD115..>
MyApp <0x7E2BDC..> sending to <0xA4D154..>
MyApp <0x7E2BDC..> sending to <0x61DAE6..>
MyApp <0x7E2BDC..> received MyMsg from <0x7E2BDC..> to <0x61DAE6..>
MyApp <0x7E2BDC..> sending to <0xED7A4D..>
MyApp <0x7E2BDC..> sending to <0x15D42B..>
MyApp <0x7E2BDC..> sending to <0x8667AA..>
MyApp <0x7E2BDC..> received MyMsg from <0x7E2BDC..> to <0x8667AA..>
MyApp <0x7E2BDC..> sending to <0x6DC5DF..>
MyApp <0x7E2BDC..> received MyMsg from <0x7E2BDC..> to <0x6DC5DF..>
MyApp <0x7E2BDC..> sending to <0x65E7C3..>
MyApp <0x7E2BDC..> received MyMsg from <0x7E2BDC..> to <0x65E7C3..>
MyApp <0x7E2BDC..> sending direct to [SNH: <0xC662F9..>//192.168.1.2:9005]
MyApp <0x7E2BDC..> sending direct to [SNH: <0x181F81..>//192.168.2.2:9010]
MyApp <0x7E2BDC..> sending direct to [SNH: <0xC662F9..>//192.168.1.2:9005]
MyApp <0x7E2BDC..> sending direct to [SNH: <0x181F81..>//192.168.2.2:9010]
```

Figure 3.5: FreePastry capture. Third node.

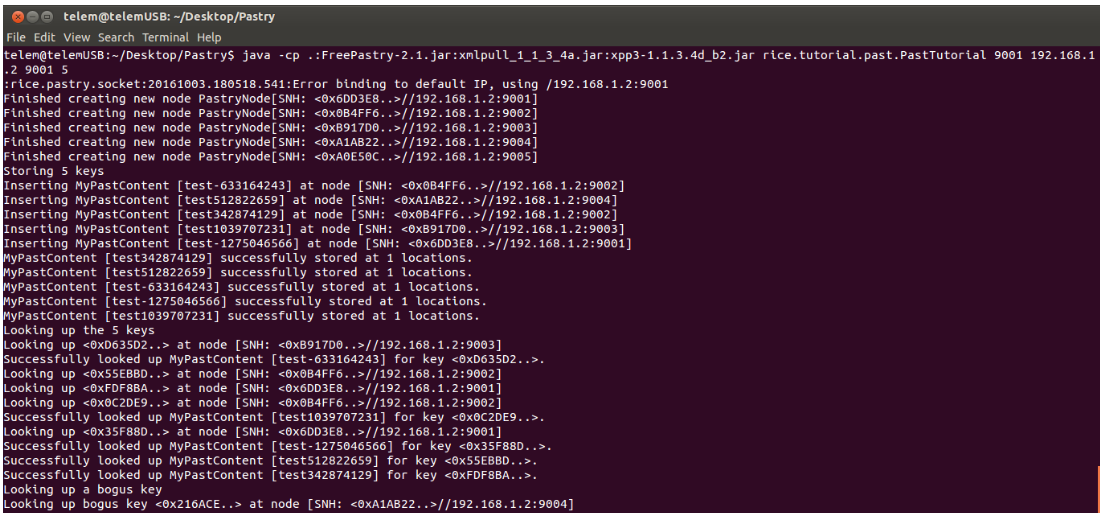
it can be a good implementation to use on `simctl`. As FreePastry is the most used Pastry implementation, there are some big projects or applications that use FreePastry libraries such as `https://las2peer.org/`, but they make us of a GUI and cannot be tested on `simctl`. However, it could be possible to create a simple code using the FreePastry library to create a network between different machines, and allowing each node to store data in the network.

3.3.2 PAST

PAST [68] is a large-scale peer-to-peer archival storage utility that provides scalability, availability, security and cooperative resource sharing. Files in PAST are immutable and can be shared at the discretion of their owner. PAST is built on top of Pastry, a generic, scalable and efficient substrate for peer-to-peer applications. It was designed by Peter Druschel (Rice University) and Antont Rowstron (Microsoft Research), the authors of the original Pastry paper [80].

The PAST system is composed of nodes connected to the Internet, where each node is capable of initiating routing client requests to insert or retrieve files. Optionally, nodes may also contribute storage to the system. The PAST nodes form a self-organizing overlay network. Inserted files are replicated on multiple nodes to ensure persistence and availability.

A free implementation of PAST was also developed at Rice University in the FreePastry project, which is called Past [45]. Past is a FreePastry's Distributed Hash Table (DHT), and it provides a tutorial [46] that shows the user how to get it up and running. To use Past, two additional jars binary files have to be included: *xmlpull_1_1_3_4a.jar* and *xpp3-1.1.3.4d_b2.jar*. These files are provided in the lib directory in the FreePastry source distribution. After getting these binary files, I take a look at the tutorial and run the example code. It creates 5 nodes on a single machine and each one stores and retrieves one string.


telem@telemUSB: ~/Desktop/Pastry
File Edit View Search Terminal Help

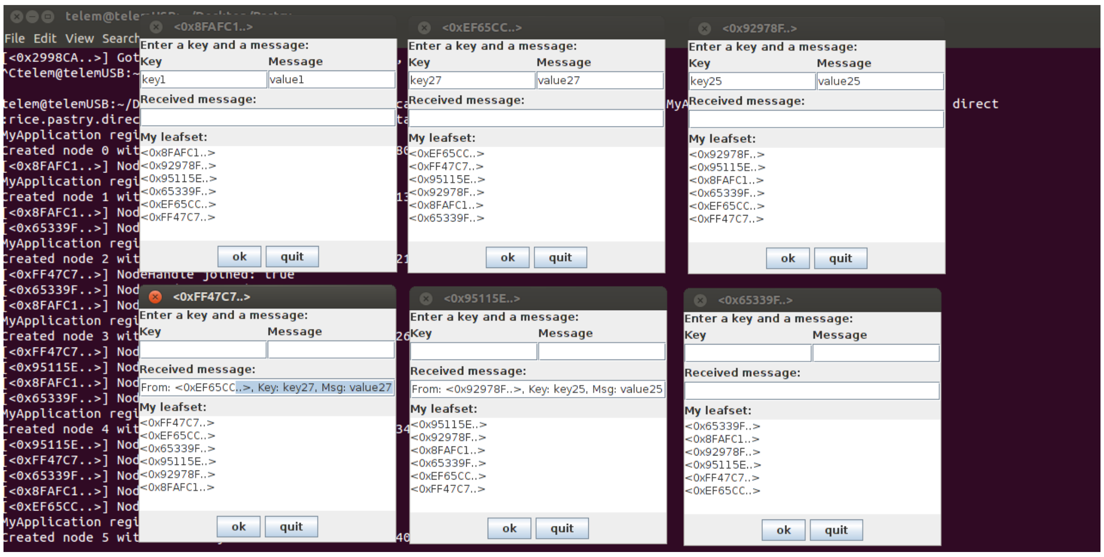
```
telem@telemUSB:~/Desktop/Pastry$ java -cp .:FreePastry-2.1.jar:xmlpull_1_1_3_4a.jar:xpp3-1.1.3.4d_b2.jar rice.tutorial.past.PastTutorial 9001 192.168.1
.2 9001 5
:rice.pastry.socket:20161003.180518.541:Error binding to default IP, using /192.168.1.2:9001
Finished creating new node PastryNode[SNH: <0x6DD3E8..>//192.168.1.2:9001]
Finished creating new node PastryNode[SNH: <0x0B4FF6..>//192.168.1.2:9002]
Finished creating new node PastryNode[SNH: <0xB917D0..>//192.168.1.2:9003]
Finished creating new node PastryNode[SNH: <0xA1AB22..>//192.168.1.2:9004]
Finished creating new node PastryNode[SNH: <0xA0E50C..>//192.168.1.2:9005]
Storing 5 keys
Inserting MyPastContent [test-633164243] at node [SNH: <0x0B4FF6..>//192.168.1.2:9002]
Inserting MyPastContent [test512822659] at node [SNH: <0xA1AB22..>//192.168.1.2:9004]
Inserting MyPastContent [test342874129] at node [SNH: <0x0B4FF6..>//192.168.1.2:9002]
Inserting MyPastContent [test1039707231] at node [SNH: <0xB917D0..>//192.168.1.2:9003]
Inserting MyPastContent [test-1275046566] at node [SNH: <0x6DD3E8..>//192.168.1.2:9001]
MyPastContent [test342874129] successfully stored at 1 locations.
MyPastContent [test512822659] successfully stored at 1 locations.
MyPastContent [test-633164243] successfully stored at 1 locations.
MyPastContent [test-1275046566] successfully stored at 1 locations.
MyPastContent [test1039707231] successfully stored at 1 locations.
Looking up the 5 keys
Looking up <0xD635D2..> at node [SNH: <0xB917D0..>//192.168.1.2:9003]
Successfully looked up MyPastContent [test-633164243] for key <0xD635D2..>.
Looking up <0x55EBBD..> at node [SNH: <0x0B4FF6..>//192.168.1.2:9002]
Looking up <0xFDF8BA..> at node [SNH: <0x6DD3E8..>//192.168.1.2:9001]
Looking up <0x0C2DE9..> at node [SNH: <0x0B4FF6..>//192.168.1.2:9002]
Successfully looked up MyPastContent [test1039707231] for key <0x0C2DE9..>.
Looking up <0x35F88D..> at node [SNH: <0x6DD3E8..>//192.168.1.2:9001]
Successfully looked up MyPastContent [test-1275046566] for key <0x35F88D..>.
Successfully looked up MyPastContent [test512822659] for key <0x55EBBD..>.
Successfully looked up MyPastContent [test342874129] for key <0xFDF8BA..>.
Looking up a bogus key
Looking up bogus key <0x216ACE..> at node [SNH: <0xA1AB22..>//192.168.1.2:9004]
```

Figure 3.6: Past capture. FreePastry's DHT with 5 nodes on a single machine.

Although the example executes various nodes on a single machine, it can also be used to store data between nodes located on different machines.

FreePastry Demo

FreePastry Demo [16] is a simple FreePastry-based application. It was developed at University of Zurich for a lecture on Distributed Hash Tables in a Communications and Distributed Systems course. The application uses the FreePastry 2.0 library and uses Past to store data in the network. It implements a GUI to interact with the network, allowing each node to insert data and see which data it stores. The application can run either with real TCP/UDP sockets or over an emulated network, and both source code and binary file are available to download.



```
telem@tele
File  Edit  View  Search
[<0x2998CA..>] Got
^Ctelem@telemUSB:~

telem@telemUSB:~/D
:rice.pastry.direc
MyApplication regi
Created node 0 wit
[<0x8FAFC1..>] Nod
MyApplication regi
Created node 1 wit
[<0x8FAFC1..>] Nod
[<0x65339F..>] Nod
MyApplication regi
Created node 2 wit
[<0xFF47C7..>] NodeHandle joined: true
[<0x65339F..>] Nod
[<0x8FAFC1..>] Nod
MyApplication regi
Created node 3 wit
[<0xFF47C7..>] Nod
[<0x95115E..>] Nod
[<0x8FAFC1..>] Nod
[<0x65339F..>] Nod
MyApplication regi
Created node 4 wit
[<0x95115E..>] Nod
[<0xFF47C7..>] Nod
[<0x65339F..>] Nod
[<0x8FAFC1..>] Nod
[<0xEF65CC..>] Nod
MyApplication regi
Created node 5 wit
```

```
<0x8FAFC1..>
Enter a key and a message:
Key                     Message
key1                    value1
Received message:

My leafset:
<0x8FAFC1..>
<0x92978F..>
<0x95115E..>
<0x65339F..>
<0xEF65CC..>
<0xFF47C7..>
ok   quit
```

```
<0xEF65CC..>
Enter a key and a message:
Key                     Message
key27                   value27
Received message:

My leafset:
<0xEF65CC..>
<0xFF47C7..>
<0x95115E..>
<0x92978F..>
<0x8FAFC1..>
<0x65339F..>
ok   quit
```

```
<0x92978F..>
Enter a key and a message:
Key                     Message
key25                   value25
Received message:

My leafset:
<0x92978F..>
<0x95115E..>
<0x8FAFC1..>
<0x65339F..>
<0xEF65CC..>
<0xFF47C7..>
ok   quit
```

```
<0xFF47C7..>
Enter a key and a message:
Key                     Message

Received message:
From: <0xEF65CC..>, Key: key27, Msg: value27
My leafset:
<0xFF47C7..>
<0xEF65CC..>
<0x65339F..>
<0x95115E..>
<0x92978F..>
<0x8FAFC1..>
ok   quit
```

```
<0x95115E..>
Enter a key and a message:
Key                     Message

Received message:
From: <0x92978F..>, Key: key25, Msg: value25
My leafset:
<0x95115E..>
<0x92978F..>
<0x8FAFC1..>
<0x65339F..>
<0xEF65CC..>
<0xFF47C7..>
ok   quit
```

```
<0x65339F..>
Enter a key and a message:
Key                     Message

Received message:

My leafset:
<0x65339F..>
<0x8FAFC1..>
<0x92978F..>
<0x95115E..>
<0xFF47C7..>
<0xEF65CC..>
ok   quit
```

Figure 3.7: FreePastry Demo capture. Network with 6 nodes running on a single machine.

This is an example of how we can create a simple application with few lines of code using a good implementation of Pasty. However, due to its GUI, this application is not useful for `simctl`.

3.4 Tapestry

Along with Chord, Pastry and CAN, Tapestry was one of the first structured overlay networks proposed in 2001. Since then, several structured protocols have been proposed, including Kademlia. On top of these overlays, researchers have proposed numerous distributed applications, including storage and backup systems, multicast systems, distributed spam filters, resilient routing networks, mobility support and anonymous routing networks. Although Tapestry had some advantages, it was not easy to implement, so developers preferred to implement other structured overlays. Apart from the original Tapestry project, there are no Tapestry implementations on GitHub or other repositories. If we look for Tapestry implementations in GitHub, we find hundreds of projects related to Apache Tapestry, which is an open-source framework to develop Java web applications, but not the Tapestry overlay.

Tapestry provides an overlay routing network that is stable under a variety of network conditions. This provides an ideal infrastructure for distributed applications and services. There are some applications based on Tapestry such as OceanStore (distributed storage utility on PlanetLab), Mnemosyne (steganographic file system), Bayeux (self-organizing multicasting application) and Spamwatch (decentralized spam filter).

3.4.1 Tapestry project

Tapestry [86] is an overlay network infrastructure designed to provide fault-resilient delivery of messages between overlay nodes and efficient location of objects given their IDs. The Tapestry project started in March 2000 and was developed at University of California at Berkeley Computer Science Division. In 2004 the Tapestry project was abandoned, and it was replaced by Chimera, the successor of Tapestry. However, Tapestry source code is available on Chimera project website [5]. There is an old message from the authors that says that the active development on the Tapestry protocol has stopped, and they are no longer supporting the software, since it was written for Java JDK 1.3. The latest release is Tapestry 2.0.1, released July 2004.

Although the project was abandoned twelve years ago, I download the compressed file with the last Tapestry version in order to try to run it. The file contains some folders with the source code, as well as some html files from the original Tapestry website. It provides some documentation and a basic guide with the installation steps. It also lists the requirements to run the code:

- Sun JDK 1.3 or a compatible Java Development and Runtime environment. The IBM Linux JDK runtime environment and the Jikes compiler are recommended.
- A version of the UNIX make program, used in compilation.
- The Cryptix JCE library.
- The Java interface libraries for the BerkeleyDB database.

After a few days trying to install the requirements and the Tapestry code on Ubuntu 14.04 and Ubuntu 12.04 without success, I give up. As it was abandoned a long time ago, it may be better to try to run Chimera project, the Tapestry's successor.

3.4.2 Chimera

Chimera [5] is a light-weight C implementation of a "next-generation" structured P2P overlay network that provides similar functionality as prefix-routing protocols Tapestry and Pastry. Chimera gains simplicity and robustness from its use of Pastry's leaf sets, and efficient routing from Tapestry's locality algorithms. Chimera also provides efficient detection of node and network failures, and reroutes messages around them to maintain connectivity and throughput. This project was developed at the CURRENT Lab at University of California at Santa Barbara.

The last release of Chimera is v1.20, last updated on February 2006, and it is available as a C library suitable for bundling with user applications. The compressed file provides some files with an extended documentation about Chimera, as well as Javadoc and a user's guide for developers. After taking a look at the documentation, the only requirements to install the library is autoconf tool and gcc. Although Chimera is quite old, the compilation was straightforward. Once the library has been successfully installed, I proceed to run the provided tests.

Chimera provides a folder with several tests ready to run, but any documentation about what they do or how to use them is provided. One of the provided tests is a chat based on Chimera, but it only works between several consoles on a single machine, not on a real network. After some time of testing, it is not possible to run Chimera tests between different machines, but it is supposed that Chimera can do it. Although it seems a good library to develop applications, it is not worth to write a code to use

Chimera between several machines on `simctl`. It is better to use a newer library with current support and that implements some of the main P2P overlays that are usually studied.

3.4.3 OceanStore

OceanStore [71] is a global persistent data store designed to scale to billions of users. It provides a consistent, highly-available, and durable storage utility above an infrastructure comprised of untrusted servers. The OceanStore project was developed at University of California at Berkeley Computer Science Division by the developers of the original Tapestry, and it uses the original Tapestry library. Tapestry provides the object location and routing functionality that OceanStore requires while meeting its demands for consistency and performance.

It exists an official website [38] which contains an overview and some information about the project. An OceanStore prototype called Pond [79] is available on SourceForge, and both the source code and binary releases are available. Pond prototype is a working subset of the vision presented in the original OceanStore paper [71]. Although many important challenges remain, it is sufficiently complete to support applications. The latest release was last modified on 2003, so it seems that this project was also abandoned. As it says on the website, Pond should run on any Linux operating system with a Java 1.4.1 jdk, so I download the file to try to install and run it on Ubuntu 14.04.

Some documentation and tutorials are provided in the SourceForge website [39]. After taking a look at the documentation, I proceed to test Pond. The main script used to start up OceanStore nodes is *run-experiment*, and it can be executed without a previous installation. This script reads a configuration file which describes what type of OceanStore nodes have to be created and where to run them. *run-experiment* can configure and run hundreds of virtual OceanStore nodes on dozens of remote machines. It coordinates sharing information across config files, pushing data to remote sites, monitoring remote processes, and cleaning up all sites afterward.

After some time working with the scripts any result is obtained. There is no option to execute each node in a different machine and to create a network between them. I thought that Pond would be like a client software that allows the user to join the network and share files between several machines using Tapestry as the routing algorithm. But, as it is a prototype, it was in a development status and developers were interested in evaluating its performance against a variety of benchmarks in order to validate the OceanStore design and compare its performance with more traditional approaches. So this option is ruled out to be used in `simctl`.

3.5 Kademlia

Kademlia is a good example of a basic DHT, because unlike Pastry or Chord algorithms, it is extremely simple. There are no explicit routing update messages, and the internal state it maintains is fairly straightforward and easy to understand. Lookups are also accomplished in a very efficient manner.

Kademlia has been used in many household software applications such as LimeWire, BitTorrent, Overnet, EDonkey2000, eMule and uTorrent, to name a few. Nowadays, most of the popular DHT networks are based on Kademlia, such as Mainline DHT (Bitorrent), and they are used for peer to peer file sharing. On these networks the key is the identifier of the torrent file and the values are the IP address of the clients sharing the torrent.

Due to its useful applications, most of the free implementations that are available on the Internet are libraries to facilitate the creation of new projects or clients for file sharing applications. It is difficult to find some implementation that provides a console-based demo to interact with the algorithm and made for learning and research purposes.

3.5.1 OpenDHT

OpenDHT [69] is a C++11 Kademlia distributed hash table implementation written by Savoir-faire Linux Inc., a Free Software consultant company based in Montreal. OpenDHT is a free and open library implementing a distributed hash table and incorporating several innovations. This software is actually under constant development and maintenance.

OpenDHT is simple to use, reducing the cost and difficulty of developing applications that benefit from it. With a few lines of code, it allows to start a new node and connect it to the network through a known bootstrap node. It also allows to store key-value data on the network, and then to retrieve the value of a given key.

OpenDHT also provides a tool to interact with the node without having to write a new code. *dhtnode* is a command-line utility included with OpenDHT to run, control and monitor an OpenDHT node:

```
dhtnode [-p local_port] [-b bootstrap_host:port] [-i] [-v]
```

Where:

- -b allows to specify a bootstrap node address.
- -p allows to specify the local UDP port to bind.
- -i will generate a DHT "identity" with the node (RSA key pair, and certificate published on the DHT). Required to perform cryptographic operations (encrypt/sign values).
- -v will enable debug logs for OpenDHT to standard output.

It also provides some commands to put or get values for a key, and to listen for value changes at a given key. This tool looks interesting for testing the software and the kademlia network. At this point, I try to test OpenDHT in one Ubuntu 14.04 virtual machine before implementing it on `simctl`. OpenDHT GitHub webpage provides a well-documented tutorial about how to build and use the library. It lists some requirements that have to be installed before OpenDHT:

- GnuTLS 3.1+, used for cryptographic operations.
- Nettle 2.4+, a GnuTLS dependency for crypto.
- Readline, an optional dependency for the DHT tools.
- Cython, an optional dependency for the Python bindings.
- msgpack-c 1.0+, used for data serialization.

After installing some requirements, I try to build and install msgpack-c 1.3. At first, when trying to compile, an error appears: *Could NOT find GnuTLS: Found unsuitable version "3.2.11", but required is at least "3.3"* while the tutorial said that GnuTLS 3.1+ was required. In the Ubuntu 14.04 repository, the last available version is GnuTLS 3.2.11, and GnuTLS 3.3.8 is not available until Ubuntu 15.04 [19]. So, instead of downloading a new Ubuntu 15.04 virtual image, I decide to edit the files *configure.ac* and *CMakeLists.txt* changing the requirements from GnuTLS 3.3 to GnuTLS 3.1. After that, the software compiles successfully and all the requirements have been installed. But then, when trying to compile OpenDHT, it breaks due to allocation memory problems. The virtual machine I was using had 1800MB of memory, so I stop it and expand the RAM to 3072MB. The same problem occurs. Then, I expand the memory to 4096MB of RAM, but the compilation breaks again. Since OpenDHT is in current development, maybe it is an internal problem of the software.

After that and before looking for another solution, I go deeper in the documentation and the code, and it seems that OpenDHT only acts as a client, connecting to an existing Kademlia network through a bootstrap node on the Internet. So OpenDHT cannot create a Kademlia network. Therefore, it is not possible to use it to create a local Kademlia network between several machines in a local area network, so it will not be useful for `simctl`.

3.5.2 Pydht

Pydht [52] is a python implementation of the Kademlia DHT data store written by the GitHub user `isaaczafuta`, and it was last updated on 2012. This implementation allows a node to create a kademlia network. Then the other peers can connect to the network by entering the address of one bootstrap node already connected.

After downloading the software, first I test it on a single PC and it works. It allows the nodes to insert data in the network in key-value mode. Once the data is stored on the network, the console returns the value of a given key. But nothing else, it does not allow the user to know which node is storing the data, or who the predecessor and successors of a peer are. Moreover, it does not work on a local network running each node in a different machine. Therefore, this software will not be useful for the project.

3.5.3 TomP2P

TomP2P is a Kademlia-based DHT implementation written in Java. The first TomP2P version was created in 2004 by Thomas Bocek and it has been improved since then. The code of this library is located at GitHub [56] and was last updated on May 2016, so it seems that it is in current development. At the official webpage [55] we can find all the documentation and releases of TomP2P.

TomP2P is a P2P library and a distributed hash table implementation which provides a decentralized key-value infrastructure for distributed applications. TomP2P stores key-value pairs in a distributed manner. To find the peers to store the data in the distributed hash table, TomP2P uses an evolved Kademlia iterative routing to find the closest peers. The documentation provides some code examples that show how to use the library in order to create a network and join different peers.

After taking a look at the documentation, it seems that TomP2P is a good implementation, used in several projects and can work on the Internet. Therefore, it can work on a local network between different computers.

I download the code from GitHub, which was modified some months ago, instead of the last release in the official webpage, TomP2P 5.0-Beta8, that was published on May 27, 2015. The TomP2P source code in the Git repository contains eclipse specific configuration files to test it on eclipse, but it can also be built by using Apache Maven Project. So I install Apache Maven Project and then I try to build the library on Ubuntu 14.04. The library compiles without problems but some of the example codes show some errors. Instead of writing some code to test the library, I decide to look if there is any project that uses the library to interact with the network.

After a long research I find Hive2Hive, an open-source library built on top of TomP2P, that provides a console-based tool for testing.

3.5.4 Hive2Hive

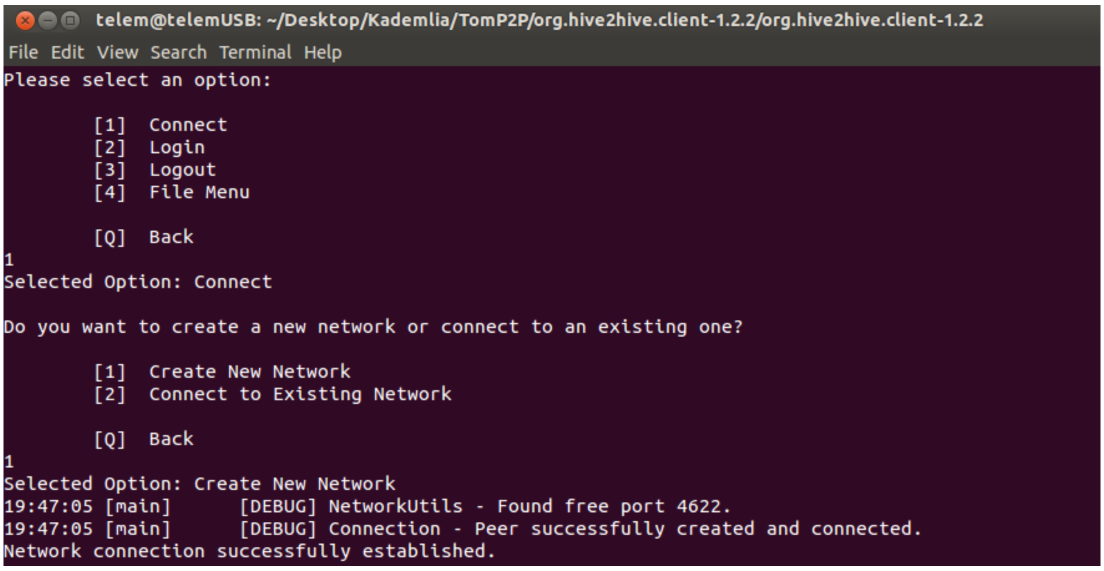
The Hive2Hive [20] library project is an open-source library written in Java that arose from a course challenge task project at the Communication Systems Group at the University of Zurich, Switzerland. It is built on top of TomP2P and is focused on secure, distributed P2P-based file synchronization and sharing.

Hive2Hive allows the users to store or share files on the network, so that these files are distributed among the other nodes. Hive2Hive offers the same basic functionality known from popular synchronization services such as Dropbox or Google Drive. In addition to providing the library, Hive2Hive provides a console-based tool to test the project. This looks very interesting to use in `simctl`, so I download the last release 1.2.2 updated on March 2015. The console-based tool is provided on a Java ARchive (JAR) file, so it can be executed by the command:

```
java -jar org.hive2hive.client-1.2.2.jar
```

After running the previous command, the console starts with some intuitive menus. It allows the user to create a new network or to connect to an existing network by introducing the IP address of the bootstrap node. Once the network is running, the user has to log in, since the central element of the user management in Hive2Hive is the user profile. This user profile contains all relevant information about a user in the network: the User ID, the User Encryption Key Pair, the User Authentication Key Pair and the File Tree. When the user has logged in, several options are showed. The user can add, update, download, move or delete a file stored at the network. It can also share a folder with another user by indicating its User ID.

I execute the command-based tool in 2 different Ubuntu 14.04 virtual machines. One of them creates the network and logs in with a User ID. The other one connects to the same network by using the IP address of the first virtual machine as a bootstrap node. After a successfully connection, it logs in with a different User ID. After that, I try to upload files to the network and it works well, but it is not able to share a folder with other users. There is a problem when giving reading and writing permissions to the other user in a shared folder. Then, I restart both consoles and try to use the same User ID in both peers. When one peer adds a file, the other peer with the same User ID downloads the file instantly. So it seems that the distributed storage, the network and the Kademlia iterative routing to find peers work well, although there is a problem in sharing files between users with a different User ID. So it can be useful for a user that stores files on the network and is able to get them in any computer by joining the network and login in with its User ID.


telem@telemUSB: ~/Desktop/Kademlia/TomP2P/org.hive2hive.client-1.2.2/org.hive2hive.client-1.2.2
File Edit View Search Terminal Help

```
Please select an option:

        [1]  Connect
        [2]  Login
        [3]  Logout
        [4]  File Menu

        [Q]  Back
1
Selected Option: Connect

Do you want to create a new network or connect to an existing one?

        [1]  Create New Network
        [2]  Connect to Existing Network

        [Q]  Back
1
Selected Option: Create New Network
19:47:05 [main]      [DEBUG] NetworkUtils - Found free port 4622.
19:47:05 [main]      [DEBUG] Connection - Peer successfully created and connected.
Network connection successfully established.
```

Figure 3.8: Hive2Hive demo main menu.

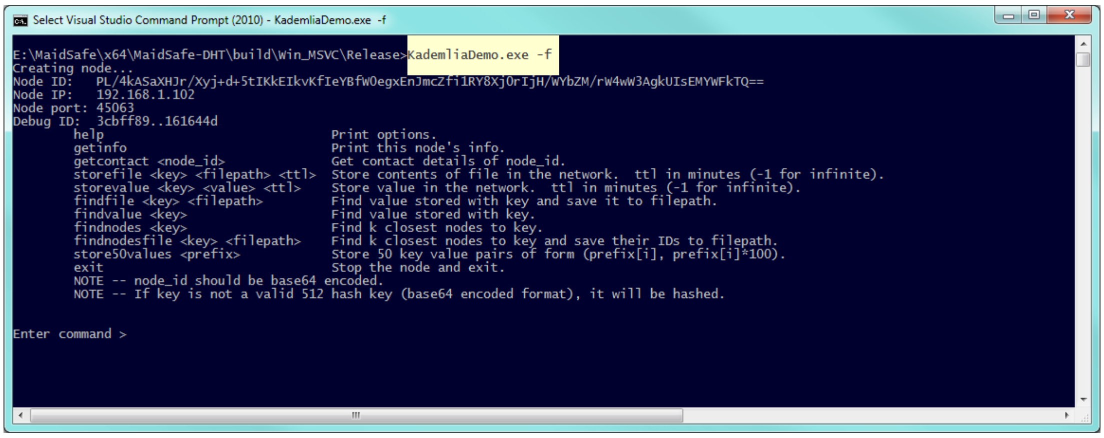
Therefore, it could be an option to implement on `simctl`. For example, in order to observe the Kademlia behaviour, a network with several nodes could be created. Then, from one peer, the user could log in and upload some files into the network. After that, from another node, the user logs in with the same User ID and all the files are downloaded. While this is done, the network can be analysed to observe which packets are send and which peers store the files.

3.5.5 MaidSafe

MaidSafe-DHT [30] is a Kademlia-like Distributed Hash Table (DHT) library written in C++ that utilizes Kademlia-like routing tables for scalability. This library was deprecated and replaced by MaidSafe-Routing, another library focused on routing. But, despite being deprecated, its source code is still available on GitHub.

All the MaidSafe libraries are part of a super-project called MaidSafe [32]. MaidSafe is a company that is designing and implementing what they call the SAFE (Secure Access For Everyone) Network. This network is a secure and decentralized data management service that is built by sharing the unused computer resources of the individual network participants.

This SAFE Network is built from the MaidSafe project and its libraries. The MaidSafe project provides some APIs and several libraries for routing, encrypting, etc. that can be used in any project, but I am only interested on the deprecated MaidSafe-DHT library, since it is based on Kademlia. The MaidSafe-DHT project also provides a KademliaDemo executable that allows to create a network with several nodes and share files between them using the kademlia algorithm. So I will try to install the library and test its KademliaDemo.


Select Visual Studio Command Prompt (2010) - KademliaDemo.exe -f

```
E:\MaidSafe\x64\MaidSafe-DHT\build\Win_MSVC\Release>KademliaDemo.exe -f
Creating node...
Node ID:   PL/4kASaXHJr/Xyj+d+5tIKkEIkvKfIeYBfW0egxEnJmcZfi1RY8Xj0rIjH/WYbZM/rW4wW3AgkUIsEMYWFkTQ==
Node IP:   192.168.1.102
Node port: 45063
Debug ID:  3cbff89..161644d
        help                            Print options.
        getinfo                         Print this node's info.
        getcontact <node_id>            Get contact details of node_id.
        storefile <key> <filepath> <ttl> Store contents of file in the network.  ttl in minutes (-1 for infinite).
        storevalue <key> <value> <ttl>  Store value in the network.  ttl in minutes (-1 for infinite).
        findfile <key> <filepath>       Find value stored with key and save it to filepath.
        findvalue <key>                 Find value stored with key.
        findnodes <key>                 Find k closest nodes to key.
        findnodesfile <key> <filepath>  Find k closest nodes to key and save their IDs to filepath.
        store50values <prefix>          Store 50 key value pairs of form (prefix[i], prefix[i]*100).
        exit                            Stop the node and exit.
        NOTE -- node_id should be base64 encoded.
        NOTE -- If key is not a valid 512 hash key (base64 encoded format), it will be hashed.


Enter command >
```

Figure 3.9: Kademlia demo main menu (screenshot from MaidSafe-DHT documentation).

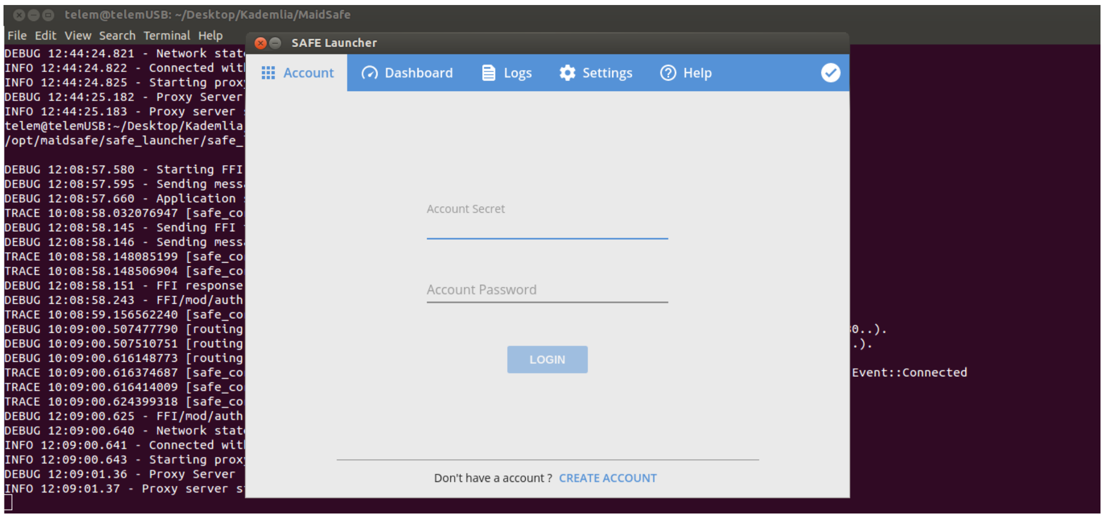
This kademlia demo looks interesting to implement over `simctl` in order to create a kademlia network between several nodes and analyze its behavior. To build and install MaidSafe-DHT some tools are required:

- gcc-4.8 or newer.
- CMake (minimum version 2.8.4) to build the project.
- MaidSafe-Common library [29].
- MaidSafe-Transport transport library [33].

After installing gcc and CMake, now we must download the source code of MaidSafe-Common, MaidSafe-Transport and MaidSafe-DHT libraries. Maidsafe-Common library provides many components that are widely used by other MaidSafe libraries, so it must be built and installed before anything else.

In the documentation of MaidSafe-Common it says that currently, this library should only be built as part of the MaidSafe super-project. The source code has been modified to use it with the super-project, so it cannot work with MaidSafe-DHT. The current source code dates on 2015, so I try to download a previous version of the code last edited on 2012 (v0.11.00), despite of being unfinished. Several errors appear during the compilation of MaidSafe-Common. Without this library it is not possible to install MaidSafe-DHT and, therefore, it is not possible to test the KademliaDemo.

As MaidSafe is based on Kademlia, it can be interesting to test the full project. In the official website [31] there is an alpha release of SAFE launcher. It is a gateway to the SAFE Network that enables users to connect to the network, authorize third party apps to connect on their behalf, and also provides access to SAFE websites. So I download and install it. When executing SAFE launcher, a new window with a graphical user interface starts. It is the client graphical application for joining the SAFE Network, and there is no version that can run in console mode with no GUI. Therefore, it cannot be used in `simctl`.


telem@telemUSB: ~/Desktop/Kademlia/MaidSafe
File Edit View Search Terminal Help
SAFE Launcher
Account
Dashboard
Logs
Settings
Help
Account Secret
Account Password
LOGIN
Don't have a account ? CREATE ACCOUNT

```
DEBUG 12:44:24.821 - Network stat
INFO 12:44:24.822 - Connected wit
INFO 12:44:24.825 - Starting prox
DEBUG 12:44:25.182 - Proxy Server
INFO 12:44:25.183 - Proxy server
telem@telemUSB:~/Desktop/Kademlia
/opt/maidsafe/safe_launcher/safe_

DEBUG 12:08:57.580 - Starting FFI
DEBUG 12:08:57.595 - Sending mess
DEBUG 12:08:57.660 - Application
TRACE 10:08:58.032076947 [safe_co
DEBUG 12:08:58.145 - Sending FFI
DEBUG 12:08:58.146 - Sending mess
TRACE 10:08:58.148085199 [safe_co
TRACE 10:08:58.148506904 [safe_co
DEBUG 12:08:58.151 - FFI response
DEBUG 12:08:58.243 - FFI/mod/auth
TRACE 10:08:59.156562240 [safe_co
DEBUG 10:09:00.507477790 [routing                                                  0..).
DEBUG 10:09:00.507510751 [routing                                                  .).
DEBUG 10:09:00.616148773 [routing
TRACE 10:09:00.616374687 [safe_co                                                  Event::Connected
TRACE 10:09:00.616414009 [safe_co
TRACE 10:09:00.624399318 [safe_co
DEBUG 12:09:00.625 - FFI/mod/auth
DEBUG 12:09:00.640 - Network stat
INFO 12:09:00.641 - Connected wit
INFO 12:09:00.643 - Starting prox
DEBUG 12:09:01.36 - Proxy Server
INFO 12:09:01.37 - Proxy server s
```

Figure 3.10: SAFE launcher capture.

3.6 JXTA

JXTA [25] is a programming language and platform independent Open Source protocol started by Sun Microsystems for peer-to-peer networking in 2001. The JXTA technology is a set of open protocols that enable any connected device on the network to communicate and collaborate in a P2P manner. This JXTA technology can be used to create P2P applications based on Java, although C and C++ versions are currently available. As it is an open community, developers can contribute to create extensions of JXTA protocols.

JXTA peers create a virtual network where any peer can interact with other peers and resources directly, even when some of the peers and resources are behind firewalls and network address translations (NATs) or on different network transports. For the communication between peers is used a group of asynchronous protocols based in the model request/reply. The different JXTA protocols standardize the manner in which peers:

- Discover each other.
- Communicate with each other.
- Cooperate with each other to form secure peer groups.
- Advertise network resources.

3.6.1 JXTA/JXSE

JXSE [24] is the name of the Java programming language implementation of JXTA. The JXSE website offers programmer's guides and tutorial codes. The last available release is JXSE 2.7 and was launched on 2011. It seems that the JXTA project has been progressively abandoned. In November 2010, Oracle officially announced its withdrawal from the JXTA projects, and the JXTA community forum has no activity since March 2014. Even so, I download the last JXSE version to test it.

The documentation of JXSE 2.7 contains some tutorials that provide links to download binary files and source codes for testing, but all links are dead. So any provided tutorial can be tested. The link of Javadoc provided in the official website is also dead. So, instead of writing some code to test JXTA, I will try to download any application developed with JXTA available on SourceForge or GitHub in order to see what type of applications can be built.

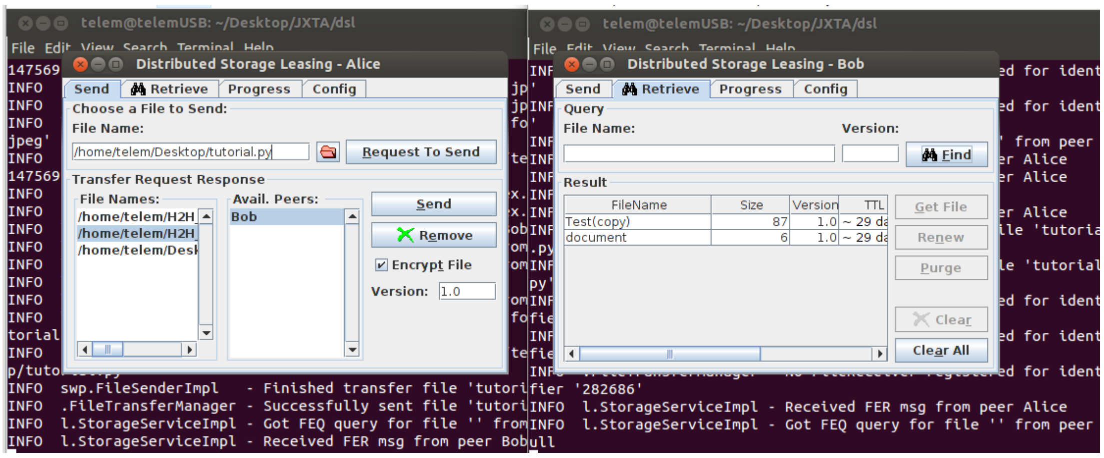
In SourceForge there are some applications developed with JXTA such as chat applications, frameworks to develop P2P applications or file sharing systems. After testing some unfinished projects, I find one working project.

Distributed Storage Leasing (DSL)

DSL [11] is a simple application built based on the JXTA project. Its purpose is to back up/replicate files from one machine to others, and later on retrieve those files. When a user uploads a file, he can select to which nodes he will send the file, so the files are stored on other nodes of the network. By this way, if one user uploads a file from one node and then switches off the machine, the files still remain on the network and can be downloaded from any machine by joining the network and logging in with the User ID and password. A node will not be able to read the files that it is storing if they are not from its property, since files can be encrypted before sending. So it can be seen as a distributed storage system such as Dropbox, but where the files are stored in the selected nodes instead of in Dropbox servers.

To test the application, at least two machines should be running within a local network. So I run the application in two Ubuntu 14.04 virtual machines. When we first run the application, a JXTA configuration window pops up. In this configuration window we must register each node with a username and a password, and then we must provide a port number. We can also decide if we want to be a rendezvous node or not, and we can add other rendezvous nodes by inserting its IP address and TCP port. JXTA applications usually connect to the Internet to download rendezvous lists, but if we work with an isolated local area network, we should add them manually.

Once the configuration has been done, the DSL GUI starts. The main tabs allow the user to send and to search and retrieve files. DSL is implemented as a Group Service, so when a user sends a file, DSL sends out a sending request to the group. After that, all the available nodes respond and are listed. The user can select the peers to which it wants to send the file. To retrieve a file, the user has to enter the file name and all the peers that have it will respond. The progress tab displays receiving and transferring progress, and the user can cancel any in-progress transfer. The configuration tab allows the user to specify the Maximum disk space storage for public file, the maximum time to live, and a few other options.



Distributed Storage Leasing - Alice

Choose a File to Send:
File Name: /home/telem/Desktop/tutorial.py

Transfer Request Response

File Names:
/home/telem/H2H
/home/telem/H2H
/home/telem/Desk

Avail. Peers:
Bob

☑ Encrypt File
Version: 1.0

Distributed Storage Leasing - Bob

Query
File Name:
Version:

Result

FileName	Size	Version	TTL
Test(copy)	87	1.0	~ 29 da
document	6	1.0	~ 29 da

```
INFO  swp.FileSenderImpl   - Finished transfer file 'tutori
INFO  .FileTransferManager - Successfully sent file 'tutori
INFO  l.StorageServiceImpl - Got FEQ query for file '' from
INFO  l.StorageServiceImpl - Received FER msg from peer Bob
```

```
fier '282686'
INFO  l.StorageServiceImpl - Received FER msg from peer Alice
INFO  l.StorageServiceImpl - Got FEQ query for file '' from peer
ull
```

Figure 3.11: Distributed Storage Leasing capture. Two peers in a local network.

The application works pretty well, and it is a secure way of storing files, being sure that any peer of the network that stores them will be able to open these files. But it is only an example of an application that can be built with JXTA. It is not very useful to understand how JXTA works. Moreover, as most of the JXTA applications, it makes use of a GUI, so it cannot be used in `simctl`.

After this experience, JXTA is ruled out to develop an application for a later use over `simctl`. Firstly, it is quite complex even for setting up simple P2P sockets. Then, documentation is really poor and none of the tutorials can be followed to learn how to use JXTA, since all links are dead and testing codes are not available. And finally, there is no more development on it and the JXTA project is abandoned.

3.7 Edonkey

The eDonkey Network, also known as the eDonkey2000 network or eD2k is a "semi-decentralized", mostly server-based, peer-to-peer file sharing network built to share big files among users, and to provide long term availability of files. It was developed in 2000 by US company MetaMachine.

eD2k is a semi-centralized network, as there is not any central hub for the network but hundreds of servers, and files are not stored on a central server but are exchanged directly between users based on the P2P principle. Unlike earlier P2P file-sharing programs such as Napster, eDonkey2000 featured "swarming" downloads, it is, clients could download different pieces of a single file from different peers utilizing the combined bandwidth of all of the peers instead of being limited to the bandwidth of a single peer.

The server part of the network is proprietary freeware. There are two families of server software for the eD2k network:

- The original one from MetaMachine, written in C++, closed-source and proprietary, and no longer maintained.
- eserver, written from scratch by a person Lugdunum [28] in pure C, also closed-source and proprietary. However, it was available free of charge and for several operating systems and computer architectures. Almost all existing eD2k servers ran this server software.

On September 2005 eDonkey officially closed its doors, but the eD2k network survived, since it was still used by other clients such as eMule [1] or Lphant [26] (officially dead). Currently, the eD2k network is not supported by any organization (in the past it was supported by the MetaMachine Inc.) and development and maintenance is being fully provided by its community and client developers. Although BitTorrent has overcome eDonkey network as the most widely used file sharing network on the Internet, eD2k is still used by some clients that have survived the passage of time such as eMule.

3.7.1 aMule

aMule [1] is a free peer-to-peer file sharing application that works with the eDonkey and Kad networks, offering similar features to eMule but supporting multiple platforms. It appeared in 2003 and comes from lMule, the first attempt to bring the eMule client to Linux.

aMule can be installed as a monolithic client, with the complete GUI and full options, or can be run in a modular way, where the core functionalities of the program are started using the aMule daemon and the software behaviour is controlled through one of the three different interfaces:

- aMuleCMD, the command-line modular client.
- aMuleGUI, the regular GUI of the software.
- aMuleWEB, the web interface provided by the aMule core built-in Webserver.

It can work with the eD2k and Kad networks on the Internet. Kad is a Kademlia network developed by the eMule Project in order to overcome the reliance on central servers. To connect to the semi centralized eDonkey network, aMule has a file called *server.met* that contains a list of servers. Each server provides information about files to the users connected to it. To connect the Kad serverless network, aMule has a file called *nodes.dat* which stores details about known Kad nodes, and this is used to bootstrap the Kad network when aMule starts. These files have to be uploaded periodically, since these nodes change with the time.

The problem is that aMule is rarely used nowadays, since a large amount of the eD2k servers closed due to legal problems and most of the clients migrated to BitTorrent-based networks. However, in May 2016 the official web [1] announced that after a long time, a new aMule release will be launched in June, 2016.

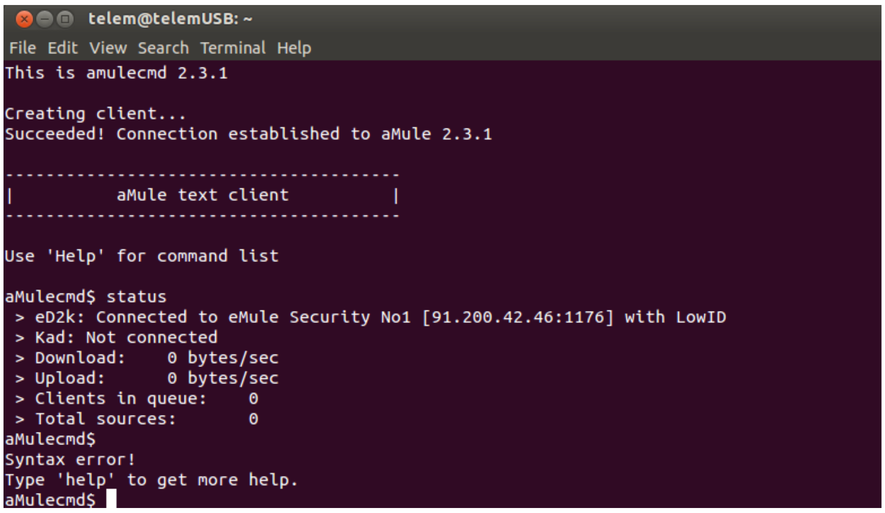
This software accomplishes the main requirements for this project, since it can run on Ubuntu 14.04 and can work without GUI by using the command-line client aMuleCMD. Moreover, it has a well-documented wiki in `http://wiki.amule.org` where the user can find manuals about how to download, install and use it.

For implementing this software over `simctl`, aMule should work in a local virtual network. As the simctl scenario does not have Internet acces, the Kad network cannot be accessed. So the solution should be to create a local eD2k server in one node and connect the other nodes to it. Then configure the server list of the clients with the address of the node that acts as a server.

Sources that contain IP addresses of private classes A (10.0.0.0 - 10.255.255.255), B (172.16.0.0 – 172.31.255.255), C (192.168.0.0 – 192.168.255.255), localhost (127.0.0.0 –

127.255.255.254) and all IPs beginning by 0 (0.x.x.x) are not valid on the Internet so, by default, aMule discards them. To run aMule on the local network this option must be changed in the configuration file *amule.conf* of all clients (FilterLanIPs=0). Otherwise, it would not work.

After installing aMule daemon in some virtual machines and modifying the IP LAN filter, the next step is to create a local eD2k server. At mid-2000s, when eDonkey2000 and Emule were popular, it was easy to find a software (Donkey Control and dserver) to build eD2k servers in the official web `www.edonkey2000.com` and `www.donkey-series.com`, but they dead long time ago. Fortunately, it exists an old web from Lugdunum [28] (the eserver developer) last updated in 2006 that was dedicated to Edonkey and eMule and has a repository [27] with some versions of eserver for different platforms. Eserver was a software that lets the user to create a local eD2k server, allowing to configure some basic parameters such as the maximum number of clients, address and default port of the server. It also allows to filter unwanted IPs.



```
This is amulecmd 2.3.1

Creating client...
Succeeded! Connection established to aMule 2.3.1

---------------------------------------
|          aMule text client          |
---------------------------------------

Use 'Help' for command list

aMulecmd$ status
 > eD2k: Connected to eMule Security No1 [91.200.42.46:1176] with LowID
 > Kad: Not connected
 > Download:     0 bytes/sec
 > Upload:       0 bytes/sec
 > Clients in queue:    0
 > Total sources:       0
aMulecmd$
Syntax error!
Type 'help' to get more help.
aMulecmd$
```

Figure 3.12: aMuleCMD client console.

I downloaded the latest available version eserver 17.14 for a high end linux ix86 machine with a “recent” 2.6 kernel and configured the required files. After that, I ran the eserver in one of the nodes. On the remaining nodes I ran aMule and added the address of the server node as a new server. After some tests, the remaining nodes connected successfully to the local server. aMule allows the clients to search some files in the local network and it finds those files that are located in the Incoming directory of other clients.

Therefore, this software can be a solution for the project. It is a P2P file sharing network that can be executed by console, with no GUI, and it runs on Ubuntu 14.04. Therefore, I will try to implement it on `simctl` (Section 4.2).

3.8 Routing algorithm simulators

So far, some algorithm implementations have been tested using a local network with a few peers. Due to the extremely large scale of P2P overlay networks, complexity has become a major issue. With the number of connected nodes reaching into the millions, a platform which can accurately simulate an overlay network is important. Many freely available network simulators exist, being able to simulate different algorithms ranging from modelling networks at the packet level to concentrating purely on the overlay network. In this section some of these simulators have been tested.

Most of these simulators must work with a graphical interface, so it is not possible to use them in `simctl`. But it is interesting to test them, since if any of the previous implementations can not be adapted to `simctl`, these graphical simulators could be an alternative for P2P studying.

3.8.1 Overlay Weaver

Overlay Weaver [42] was developed in 2007 in the Grid Technology Research Center of the National Institute of Advanced Industrial Science and Technology (AIST) in Tsukuba, Japan. It is an overlay construction toolkit that enables the user to implement a structured overlay algorithm only in hundreds of lines of code. For application developers, the toolkit provides a common API for higher-level services such as a distributed hash table (DHT) and multicast.

The toolkit provides multiple routing algorithms: Chord, Kademlia, Koorde, Pastry, Tapestry and FRT-Chord, but it also allows the implementation of a new algorithm, test it and compare it with other algorithms. Implemented algorithms can work on a real network in addition to the emulator, which can host hundreds of thousands of virtual nodes.

The toolkit provides several tools, and each of them is used for a different goal:

- Distributed Environment Emulator (owemu): There are two modes in which this option runs. In the normal mode, the whole emulator runs on a single computer, being able to host tens of thousands of nodes virtually. In the other mode, multiple computers from a real network form a single emulator in cooperation. The user starts a master Emulator and it invokes workers on remote computers via SSH.

The emulator reads the same scenario file in both cases and invokes and controls application instances according to the scenario. The scenario file is written by the user and contains the number of nodes, algorithm and some options and commands that will be simulated.

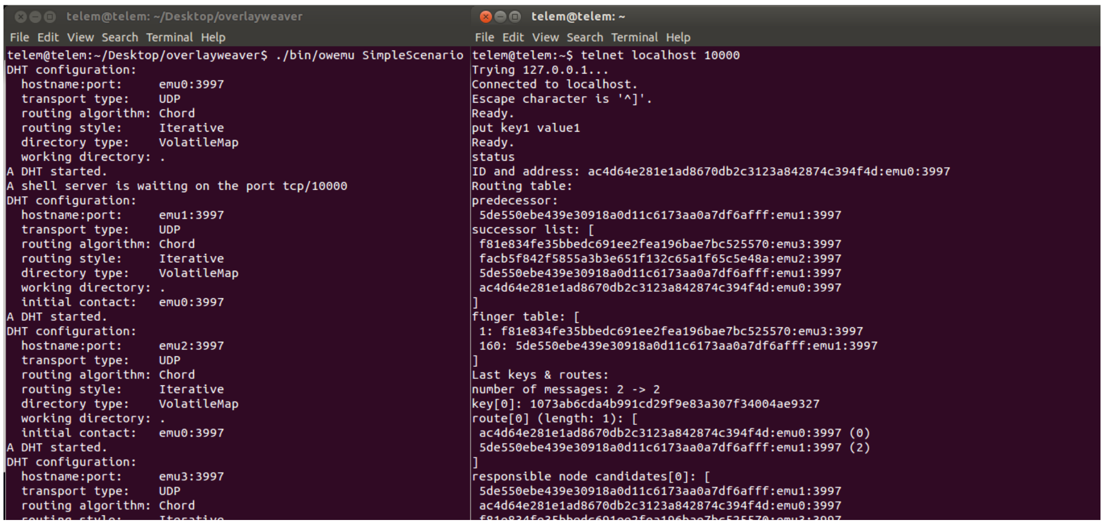
- Emulation Scenario Generator (owscenariogen): The toolkit provides a simple Emulation Scenario Generator, a tool that the user can use to generate the scenario file more easily.
- Overlay Visualizer (owviz): Overlay visualizer is a graphic tool which visualizes communication between nodes just in time. It collects communication statistics and represent it drawing the different nodes and linking them with lines, giving an intuitive understanding of the algorithm behaviour. It works both on an emulator and a real network.

In the project's website we can find barely documented information about Overlay Weaver and download the latest version of the software from November 2015. The web also contains a tutorial that explains how to build Overlay Weaver and how to use the different tools and shells. Moreover, there is a Developer's Guide that shows starting points to develop applications and routing algorithms which work with the toolkit.

Since we are looking for an implementation able to run over a local network between several machines and with no GUI, the Distributed Environment Emulator (owemu) tool in remote mode could be useful to run on `simctl`. But first I will try it on the four Ubuntu 14.04 virtual machines. Overlay Weaver requires the following software:

- Java Platform, standard Edition (Java SE) 5 or later.
- Apache Ant.

After installing the prerequisites, I download the newest version of Overlay Weaver, launched on November 2015. First of all, I try to emulate multiple nodes on a single computer in order to test if Overlay Weaver works fine. I create a simple scenario that invokes a node that waits a connection at one TCP port to read DHT shell commands. Then, the scenario invokes 3 nodes that contact the first node to join the Chord ring. From one terminal I start the emulator that creates the network and waits on the TCP port. Then, from another terminal, I telnet to the shell on the corresponding port. In this second terminal we can interact with the network putting and getting data or viewing status information such as the finger tables or all the value stored on the network. It is also possible to suspend and then resume any node.



```
telem@telem:~/Desktop/overlayweaver$ ./bin/owemu SimpleScenario
DHT configuration:
  hostname:port:     emu0:3997
  transport type:    UDP
  routing algorithm: Chord
  routing style:     Iterative
  directory type:    VolatileMap
  working directory: .
A DHT started.
A shell server is waiting on the port tcp/10000
DHT configuration:
  hostname:port:     emu1:3997
  transport type:    UDP
  routing algorithm: Chord
  routing style:     Iterative
  directory type:    VolatileMap
  working directory: .
  initial contact:   emu0:3997
A DHT started.
DHT configuration:
  hostname:port:     emu2:3997
  transport type:    UDP
  routing algorithm: Chord
  routing style:     Iterative
  directory type:    VolatileMap
  working directory: .
  initial contact:   emu0:3997
A DHT started.
DHT configuration:
  hostname:port:     emu3:3997
  transport type:    UDP
  routing algorithm: Chord
```

```
telem@telem:~$ telnet localhost 10000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Ready.
put key1 value1
Ready.
status
ID and address: ac4d64e281e1ad8670db2c3123a842874c394f4d:emu0:3997
Routing table:
predecessor:
 5de550ebe439e30918a0d11c6173aa0a7df6afff:emu1:3997
successor list: [
 f81e834fe35bbedc691ee2fea196bae7bc525570:emu3:3997
 facb5f842f5855a3b3e651f132c65a1f65c5e48a:emu2:3997
 5de550ebe439e30918a0d11c6173aa0a7df6afff:emu1:3997
 ac4d64e281e1ad8670db2c3123a842874c394f4d:emu0:3997
]
finger table: [
 1: f81e834fe35bbedc691ee2fea196bae7bc525570:emu3:3997
 160: 5de550ebe439e30918a0d11c6173aa0a7df6afff:emu1:3997
]
Last keys & routes:
number of messages: 2 -> 2
key[0]: 1073ab6cda4b991cd29f9e83a307f34004ae9327
route[0] (length: 1): [
 ac4d64e281e1ad8670db2c3123a842874c394f4d:emu0:3997 (0)
 5de550ebe439e30918a0d11c6173aa0a7df6afff:emu1:3997 (2)
]
responsible node candidates[0]: [
 5de550ebe439e30918a0d11c6173aa0a7df6afff:emu1:3997
 ac4d64e281e1ad8670db2c3123a842874c394f4d:emu0:3997
```

Figure 3.13: Simple test of Chord running in a single host.

After testing other scenarios using different routing algorithms and larger number of nodes, I try to use the emulator mode in which multiple computers form a single emulator in cooperation. In this Distributed Emulator, a user starts a master Emulator and it invokes workers on remote computers via SSH. But it also needs to read the scenario from a configuration file where the nodes are specified, so this is not what I expected. The idea was to run a network with any routing algorithm (Chord, Kademlia, Pastry, etc.) and create nodes in other computers that could join or leave the network. But using this emulator, if the master node falls all the network breaks, so it is not fault-tolerant like the real routing algorithms. So this software is not an option to implement on `simctl`.

Although being ruled out to implement it on `simctl`, Overlay Weaver is an interesting tool to learn and understand how several routing algorithms works. It can simulate a network with hundreds of nodes and represent the messages between nodes graphically. The messaging visualizer shows the communication between nodes and also draws spanning trees for multicasting with coloured lines.

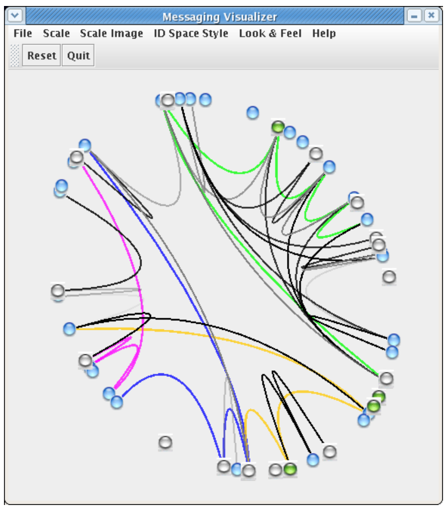


Figure 3.14: Messaging visualizer showing a Chord ring (extracted from the project's website).

3.8.2 Entangled

Entangled [14] is a distributed hash table (DHT) based on Kademlia, as well as a peer-to-peer tuple space implementation. It is written in Python, and makes use of the Twisted framework. Entangled can be used as a base for creating peer-to-peer network applications that require synchronization and event handling (such as distributed resource provisioning systems) as well as applications that do not (such as file sharing applications).

Entangled provides some examples to test the library and a basic interface to show graphically all the peers on the network and its behaviour when a node introduces some key-value pairs.

After downloading and installing Entangled, from one terminal I create a network with 15 virtual nodes on a single machine. From another terminal, I execute the graphical user interface to interact with the network. When the graphical user interface starts, a ring with several nodes appears. At the bottom there is a panel that allows to store key-value pairs, or to get or remove a value paired to a given a key. When some of these actions are run, several lines are drawn interconnecting the nodes that participate in each action. But this is too simple, and does not show anything to help to understand Kademlia.

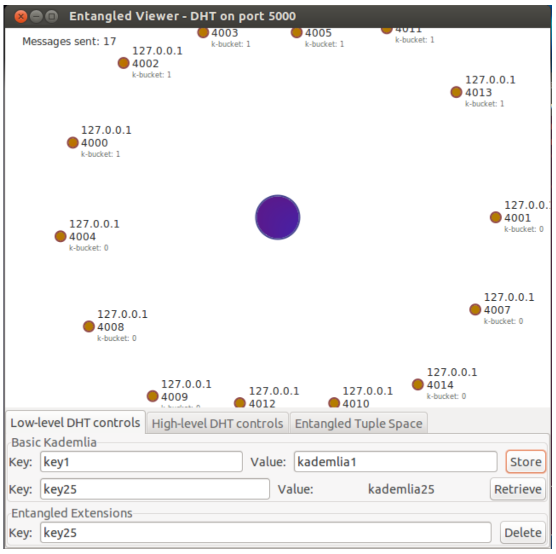


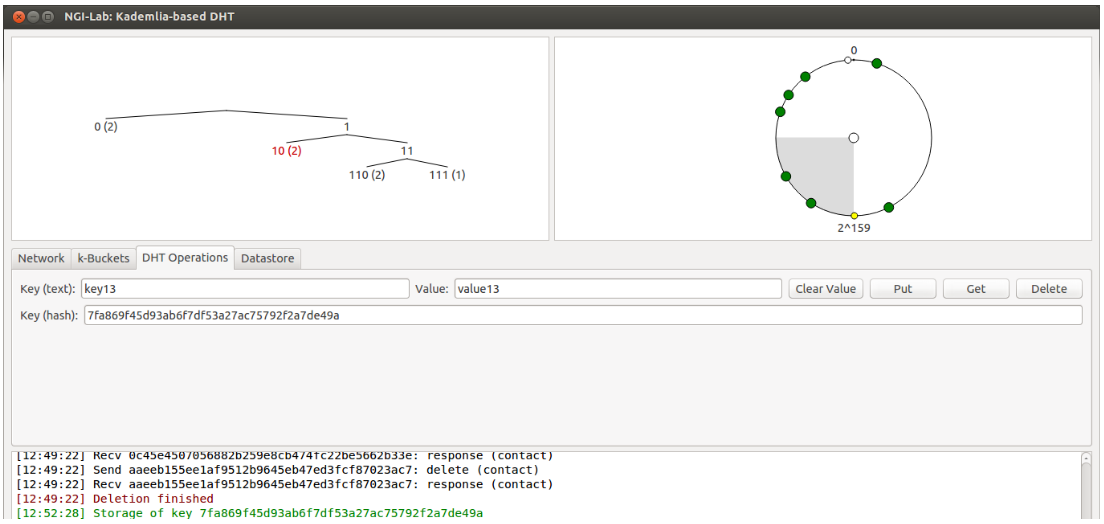
Figure 3.15: Entangled GUI.

NGI-Lab: Kademlia-based DHT

Looking for an application that uses Entangled, I found a Kademlia demo [37] for lab assignment, written in 2011 by Dr. Denis Martin, Institute of Telematics, KIT, Germanywritten. It is a demo application that visualizes some of Kademlia's interior for a simple DHT. It provides a simple put/get/delete interface of a key-value pair to the user, using Entangled as Kademlia implementation.

Unlike the Entangled GUI, this demo is quite complete and very interesting to understand Kademlia. First of all, it allows to choose an IP address and the port of our simulated node, and immediately it calculates the node ID. After that, we must choose the number of additional virtual nodes to be in the network and its ports. When these nodes are added to the network, the ring is updated and the binary tree of the routing table is represented.

It also allows to store, obtain or delete key-value pairs on the network. When writing a key, its ID is calculated on real time and it is represented on the ring, so the user can visualize where the data would be stored and which nodes are closer. Moreover, when an operation is made (put, get or delete), the corresponding nodes are interconnected by lines. Finally, k-buckets can be consulted in order to understand better how it works.



Key (text): key13 Value: value13

Key (hash): 7fa869f45d93ab6f7df53a27ac75792f2a7de49a

```
[12:49:22] Recv 0c45e4507056882b259e8cb474fc22be5662b33e: response (contact)
[12:49:22] Send aaeeb155ee1af9512b9645eb47ed3fcf87023ac7: delete (contact)
[12:49:22] Recv aaeeb155ee1af9512b9645eb47ed3fcf87023ac7: response (contact)
[12:49:22] Deletion finished
[12:52:28] Storage of key 7fa869f45d93ab6f7df53a27ac75792f2a7de49a
```

Figure 3.16: Kademlia demo GUI.

3.8.3 OverSim

OverSim [62] is an open-source overlay and peer-to-peer network simulation framework for the OMNeT++ simulation environment [84]. It was developed at the Institute of Telematics (research group Prof. Zitterbart), Karlsruhe Institute of Technology (KIT) within the scope of the ScaleNet project funded by the German Federal Ministry of Education and Research.

OverSim includes several structured P2P overlay protocols such as Chord, Kademlia and Pastry, and some unstructured protocols like GIA [63]. It also offers functions and tools to facilitate the implementation of new protocols. These protocol implementations can be used for both simulation as well as real networks. This framework allows to simulate complex heterogeneous underlay networks as well as simplified networks for largescale simulations with up to 100.000 nodes.

OverSim uses the GUI and visualization features from OMNeT++ to display networks topologies, nodes and messages. However, the GUI can be disabled for faster simulation, so it can run over console.

OverSim supports three different kinds of underlying models:

- Simple underlay: in this model data packets are sent directly from one overlay node to another by using a global routing table. It allows to simulate networks with a large number of nodes with a high level of accuracy.

- SingleHost underlay model: it allows to reuse protocol implementations in real networks. Each OverSim instance only emulates a single host, which can be connected to other instances over existing networks like the Internet.
- INET underlay model: it includes simulation models of all network layers from the MAC layer onwards. It is useful for simulations of complete backbone structures.

I will try to use it in order to test several routing algorithms running on a local network and disabling the GUI. In the official website [43] we can find well documented information about OverSim, as well as some tutorials about how to install and use it. The last OverSim release that can be downloaded at the website is dated on 2012 so, in order to avoid any problem, it is important to install OMNeT++4.2.2 and not the latest version.

After installing some of the requirements, several errors appear when trying to install OMNeT++4.2.2. After a long time looking for a solution, I decide to install it on Ubuntu 12.04, since Ubuntu 14.04 was not launched until 2014 and it may not be compatible with the old OMNeT version. Moreover, as the framework is outdated, it requires an old version of the INET framework that is also not supported in Ubuntu 14.04. Then, in the Ubuntu 12.04 virtual machine, all the requirements and OverSim can be installed without problems.

First of all, I try to simulate a Chord ring with several nodes in a single computer without GUI to test the simulator in console mode. This simulation mode only shows and collects various statistical data about the network such as sent, received, or forwarded network traffic per node, successful or unsuccessful packet delivery, and packet hop count. But it does not allow to interact with the network by storing data on the ring or using any node. Then, I try to use the simulator in SingleHost underlay mode in order to simulate one node per machine. But, every time this type of network is selected, the framework exits due to internal errors. Therefore, I cannot test it. But, anyway, this option seems to simulate the same parameters than in Simple Underlay mode, so it would not be useful for `simctl`.

Finally, although not using OverSim in `simctl`, I test the firmware with its GUI. The GUI is useful to visualize networks topologies, messages and node state variables like the routing tables. After selecting one routing protocol and configuring the number of nodes and other parameters of the network, OverSim shows step by step how the network is formed, drawing the nodes and all the messages that travel between them. This graphical simulator is very useful to understand the inner working of the routing algorithm.

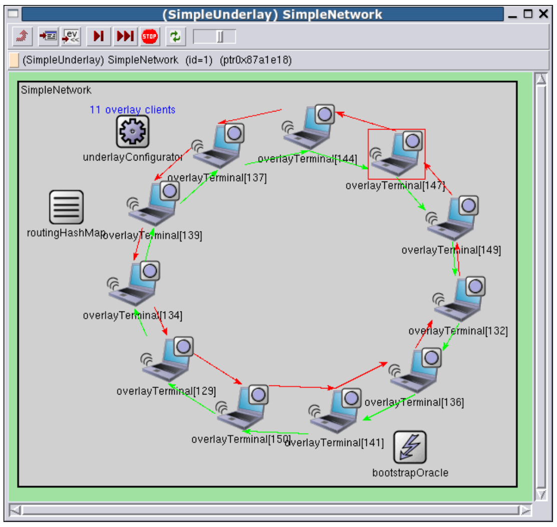


Figure 3.17: Chord ring in OverSim.

3.8.4 P2PSim

P2PSim [44] is a free, multi-thread, discrete event simulator to evaluate, investigate and explore P2P protocols, and is focussed on the underlying network simulation. P2PSim was developed at the MIT Computer Science and Artificial Intelligence Laboratory in the Parallel and Distributed Operating Systems Group in 2004.

P2PSim is written in C++ and because of using threads, implementations look like algorithm pseudo-code, which makes them easy to comprehend and allows developers to extend the simulator classes to implement new peer-to-peer protocols. Nevertheless, it provides multiple structured algorithm implementations such as Chord, Accordion, Koorde, Kelips, Tapestry and Kademlia. It also allows to make comparisons between different protocols.

P2PSim developers have been able to test the simulator with up to 3.000 nodes, but algorithm implementations do not work on a real network. P2PSim does not provide any simulation visualizations or a GUI but, however, P2PSim can be used in conjunction with a third party GTK (GIMP Toolkit, a cross-platform widget toolkit for creating graphical user interfaces) application to provide a GUI. Perl scripts are provided for the generation of graphs. The scalability of the simulator is in the order of 1000's of nodes.

The C++ documentation is very poor but it exists a How to with some instructions to install P2PSim. Event scripts can be used to control the simulation. The last version of the software is P2PSim 0.3, which was updated on April, 2005. So it is very important to use the version tools listed on the requirements:

- GCC 2.95.3.
- Openssl header files.
- libcrypto.
- libgmp.

As P2PSim does not work on a real network between several machines, I cannot use it on `simctl`, so I am interested on running P2PSim with the GTK GUI to represent the networks graphically. But the Ubuntu 14.04 image given in the Lab is too new and not compatible with P2PSim. So, if it is not possible to run it in Ubuntu 14.04, I will try to install it on Ubuntu 12.04, but not older Linux versions. Before trying to install the requirements, I take a few time googling to know which Linux versions used P2PSim developers. Most of them used Debian 3.0, Debian 5.0, etc. and the newest one was Ubuntu 10.04. But all the websites talking about P2PSim are dated several years ago, and none of these Linux versions have support.

After looking at the documentation it seems that this simulator only prints out some statistics such as the total number of bytes sent in the system, success/failure rates of key lookups, average latency, and average number of hops to locate keys. And any screenshot of the graphical simulation exists. So, as this simulator does not meet any of the requirements for the Lab session, I decide to not to test it and look for a better simulator.

3.8.5 PeerSim

Peersim [76] is a peer-to-peer simulator written in Java. It has been designed to be scalable and dynamic for simulating large P2P networks, and it can simulate both structured and unstructured overlays. PeerSim supports two models of simulation which can be attained by using two of its different simulation engines:

- Cycle-based models: nodes communicate with each other directly, and the nodes are given the control periodically in some sequential order when they can perform arbitrary actions. The cycle-based simulations can scale further with the number of nodes, but does not provide accurate results with larger models.
- Event-based models: based on scheduling set of messages in time and each node protocol is called upon according to the time message delivery order. The event-based engine provides results that are more accurate.

In the official website [49] there is some documentation with examples and tutorials provided in the form of Javadocs and online manuals. The cycle-based mode is well documented with examples, tutorials and class level documentation, while the event-driven mode is only documented at class level. The website contains additional packages developed by Peersim users for their research, and some of them are structured overlays such as Chord, Pastry and Kademlia.

PeerSim does not support distributed simulation, so it cannot be used on a real local network. Moreover, it does not allow to interact with the network by storing data or to manipulate any node. PeerSim provides neither a graphical user interface nor any debugging facilities, so it is not a useful simulator for learning about the simulated overlays.

PeerSim can simulate networks with millions of nodes that join and leave continuously, providing many statistical data. It offers class packages to perform common statistics calculation as well as additional user-defined data collection coding. Other packages perform statistical computation, lattice, random graphs, etc. This simulator can be interesting for investigation, but not for learning about a structured overlay.

The goal of this TFM is to create a Lab session to learn about some P2P overlay. We are interested on the overlay and its routing protocol, but not on the behavior of a network with millions of nodes. So I mention PeerSim because is a well-known simulator for P2P investigation, since it can simulate accurate statistics that can be similar to a real huge network.

3.8.6 PeerFactSim.KOM

PeerfactSim.KOM [48] is a simulator for large scale distributed/p2p systems, aiming at the thoroughly evaluation of interdependencies in multi-layered p2p systems. The main development of PeerfactSim.KOM started in 2005 at the Multimedia Communication Lab (KOM) at the Technische Universtät Darmstadt. It has been further extended and maintained at the University of Paderborn (UPB) and the University of Düsseldorf (HHU).

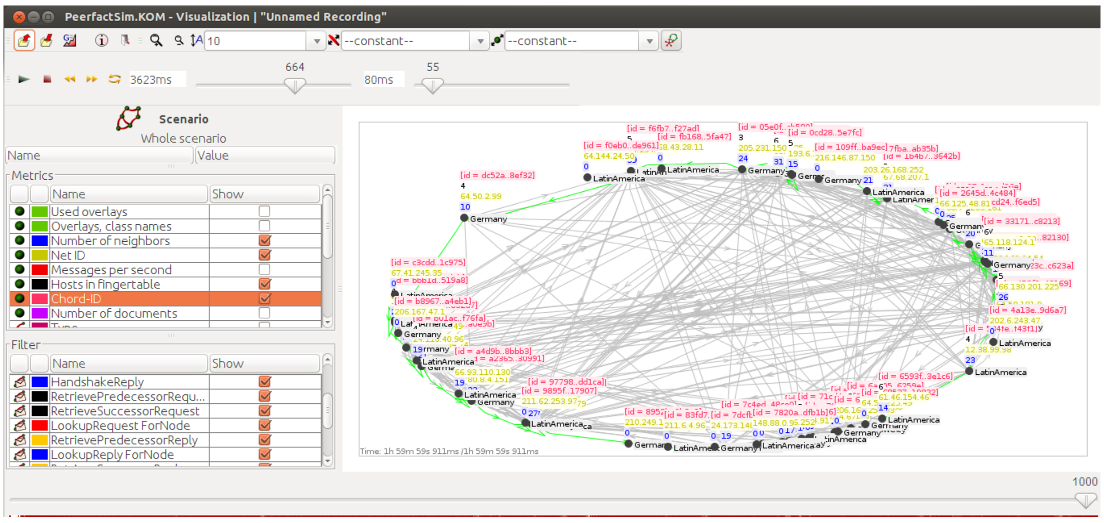
PeerfactSim.KOM is very well documented. In its website we can find several documents and video tutorials about how to install and use the simulator. The HowTos are intended for users that want to use the simulator just for simulations, while the common documentation highlights the general aspects of the complete architecture and gives some details about the implementation and the development of existing or further components. Moreover, every developer can contribute a documentation for the component that he implemented.

The simulator is written in java and the code can be found for free at the download section of its web. The p2p overlay layer covers structured p2p overlays such as Chord, Re-Chord, Kademlia, Pastry and Globase, and unstructured p2p overlays such as GIA, Gnutella and Napster. The simulator uses an XML-based configuration file where the user can specify which implementations and configurations on which layers are to be used. It also specifies an action file that contains operations that are to be started by specific peers at specific time intervals. Once the configuration is done, the user may start a GUI to choose the configuration file and to observe the simulation status. The simulator plots the network with its peers and also shows the traffic through the network and some statistical data. Each time PeerfactSim runs a simulation, it creates an output folder with more than 300 files with statistical data and plots.

The last available version of PeerfactSim.KOM is dated on June 2016, so this software is in continuous development. The source code is provided as an Eclipse project so that developers can modify or add new functionalities. But to use only the simulator with its GUI, an executable is provided and it is not necessary to install the Eclipse IDE. Moreover, it is also possible to run the simulator from the console.

I download the last version of the simulator updated on 2016 and test it on Ubuntu 14.04. The only requirement for running the executable is Java JDK 1.8. As Ubuntu 14.04 is earlier than Java 8, we must add webupd8team Java PPA repository in our system to install Oracle Java 8. Then, we can run the simulator successfully.

First of all, I test a Chord ring with the given configuration file from the console. This simulation mode does not show any type of data about the simulation, but it creates an output folder with all the plots and statistics. After that, I test the GUI of the simulator. When starting the visualization environment, the user must choose which P2P overlay want to run, and also has to choose the configuration file that specifies information about the nodes in the network. Once the simulation starts, the nodes begin to appear and different coloured lines are drawn between the peers representing messages sent between them and the connection between successors and predecessors. The distance between nodes also reflect the delay between them. The user can adjust the speed of the visualization and decide which information of the nodes is shown.



Name	Value

Metrics

		Name	Show
●	🟩	Used overlays	☐
●	🟩	Overlays, class names	☐
●	🟦	Number of neighbors	✓
●	🟨	Net ID	✓
●	🟥	Messages per second	☐
●	⬛	Hosts in fingertable	✓
●	🟥	Chord-ID	✓
●	🟪	Number of documents	☐

Filter

		Name	Show
	🟦	HandshakeReply	✓
	⬛	RetrievePredecessorRequ...	✓
	⬛	RetrieveSuccessorRequest	✓
	🟥	LookupRequest ForNode	✓
	🟨	RetrievePredecessorReply	✓
	🟦	LookupReply ForNode	✓

Time: 1h 59m 59s 911ms /1h 59m 59s 911ms

Figure 3.18: Visualization environment of PeerfactSim.

PeerfactSim.KOM is a user-friendly simulator both for users and developers. This simulator is a good tool for any researcher aiming at the simulation of multi-layered p2p systems in combination with realistic network models. It is also very interesting for learning about P2P overlays and to understand how they work.

Chapter 4

Implementation

4.1 OpenChord

Updating the file system

To install Open Chord and its requirements for a later use in `simctl`, we have to install it directly to the debian file system from which `simctl` runs the virtual machines. In the filesystems folder we can choose between debian5 and debian6. The debian5.fs image is a Debian 5 Lenny [9], that was abandoned in 2012. The debian6.fs image is a Debian 6 Squeeze [10], which had support until February 2016. First I updated the repositories of debian5 but, when trying to install Java, an internal error appeared:

```
E: Internal Error, Could not perform immediate configuration (2)
on initscripts
```

After some time looking for a solution, I decided to use debain6.fs instead of debian5.fs, since it is a newer version and have had support until some months ago. First of all, we have to open a new terminal and go to the filesystems directory. After that, with the `simtools-fsupdate` command we access to the Debian 6 filesystem:

```
cd /usr/share/vnuml/filesystems
sudo simtools-fsupdate debian6.fs
```

Once we are in the debian6 filesystem, we must update the repositories by adding some new links at the end of the file `/etc/apt/sources.list`. So we edit this file with any text editor such as VI:

```
vi /etc/apt/sources.list
```

and add the following lines:

```
deb http://archive.debian.org/debian/ squeeze main non-free contrib
deb-src http://archive.debian.org/debian/ squeeze main non-free
contrib
deb http://archive.debian.org/debian-security/ squeeze/updates
main non-free contrib
deb-src http://archive.debian.org/debian-security/ squeeze/updates
main non-free contrib
```

Once the repository has been modified, we can update debian 6 successfully.

```
apt-get update
```

Requirements

In order to be compiled, Open Chord requires:

- Java 2 Platform Standard Edition Development Kit 5.0.
- The Apache Ant3 build tool.
- A library of Apache log4j logging framework, that must be placed in the lib directory.

In order to be executed Open Chord just requires a Java 2 Platform Standard Edition Runtime Environment 5.0. The library Log4j is only required to compile Open Chord and does not need to be available at runtime. So we install versions of JRE and JDK coresponding to Debian 6 Squeeze by running the commands:

```
apt-get install default-jre
apt-get install default-jdk
```

After a successfully installation, now we must create the `JAVA_HOME` variable for all users, setting the path of JDK. We must edit `/etc/profile` with some text editor such as VI:

```
vi /etc/profile
```

and add the variables:

```
export JAVA_HOME=/usr/lib/jvm/java-6-openjdk
export PATH=$JAVA_HOME/bin:$PATH
```

Then we install Apache Ant by running:

```
apt-get install ant
```

Finally, we download the log4j library to place it in the `/lib` folder of Open Chord.

Installation

In order to install Open Chord 1.0.4, it has to be downloaded from https://sourceforge.net/projects/open-chord/files/Open%20Chord%201.0/1.0.4/. There, the sources and any other required files can be found in zip archive. When compiling, some errors appear since a few files contain non UTF-8 characters in the comments. Moreover, the log4j library must be placed to the `/lib` folder. So, before compiling the project on Debian 6, I edit the previous files and add the library to the required folder on Ubuntu 14.04. Then, I compress the full project so that I can download it in Debian 6 and compile it without problems. After that, in debian 6, we go to the `/home` directory, download and decompress the file with the modified OpenChord ready to compile:

```
cd home
wget https://dl.dropboxusercontent.com/u/48303732/open-chord_1.0.4.tar
tar -xvf open-chord_1.0.4.tar
```

At this point, the code is located at `/home/open-chord_1.0.4.tar`.

Compilation

Open Chord can be compiled with help of the Apache Ant build tool, that can be obtained from the Apache Software Foundation for free. For this purpose, Open Chord is distributed with an Ant build file (`build.xml`). To compile Open Chord we must go to the directory where the Open Chord build file is located. Then we use the `ant` command to compile the project successfully.

```
cd open-chord_1.0.4
ant -f build.xml
ant
exit
```

Once the compilation has finished, the code and the requirements are installed in the filesystem debian6.fs. We exit from Debian 6 OS with the `exit` command and go back to the Ubuntu 14.04 environment.

To start Open Chord, a long java instruction has to be typed. It contains the paths of the classes and libraries that have to be used. To make it easier, I copy it in a bash file that later will be copied in the Open Chord directory in `simctl`. By this way, users can run Open Chord in each node with a simpler command.

```
cd /usr/share/vnuml/scenarios/files
sudo mkdir /openchord
cd openchord
sudo vi openchord
```

The bash file looks like:

```
#!/bin/bash
# This file contains the command needed to execute Open Chord
# in simctl.

java -cp /home/open-chord_1.0.4/build/classes:/home/open-chord_1.0.4/config:
/home/open-chord_1.0.4/lib/log4j.jar de.uniba.wiai.lspi.chord.console.Main
```

Finally, we must give permissions to the file so that it can be executed in `simctl`.

```
sudo chmod 755 openchord
```

Running a test scenario

The software is installed and ready to run, so now we must create a scenario with several machines to use OpenChord in a network. Before creating the lab session, I create a scenario with 5 nodes to test OpenChord in `simctl` and make sure everything works.

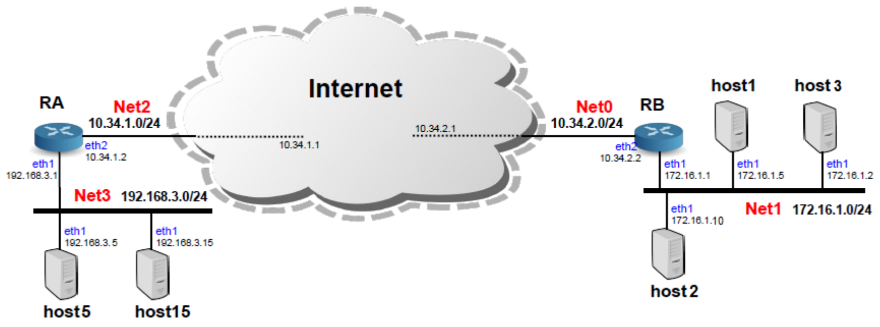


Figure 4.1: OpenChord test scenario.

Now, from a terminal, I start the test scenario and wait until it is created. Once it is running I access to the different nodes of the network to run OpenChord.

```
simctl test-OpenChord start
simctl test-OpenChord get host1
simctl test-OpenChord get host2
simctl test-OpenChord get host3
simctl test-OpenChord get host5
simctl test-OpenChord get host15
```

From any host console, to start the OpenChord software we must run the following command:

```
java -cp /home/open-chord_1.0.4/build/classes:/home/open-chord_1.0.4/config
/home/open-chord_1.0.4/lib/log4j.jar de.uniba.wiai.lspi.chord.console.Main
```

Or simply we can run the bash file created before starting the scenario:

```
./openchord
```

First of all, I try to create a Chord ring from host1 by:

```
@host1:#    joinN
```

The Chord ring has been successfully created. To add a new node to the ring, a bootstrap node address and port have to be indicated, as well as the TCP port which the new node will use to send messages. When a node creates a ring, 4242 is set as its default TCP port. Any other node that joins the network will be able to choose any available TCP port. So I add host2 to the Chord ring by using host1 as the bootstrap node. It is done by running:

```
@host2:#    joinN -port 8080 -bootstrap 172.16.1.5:4242
```

Then we can join the resting nodes to the network by adding the port that they will use and the address and port of any bootstrap node:

```
@host15:#   joinN -port 8015 -bootstrap 172.16.1.10:8082
@host5:#    joinN -port 8085 -bootstrap 172.16.1.5:4242
@host3:#    joinN -port 8083 -bootstrap 192.168.3.15:8015
```

Now the Chord ring is composed by 5 nodes. We can observe that any node can be used as a bootstrap node. From all the nodes we can test the main actions: insert or remove data from the network, get values of a given key, consult the finger table (refsN) with successors and predecessors, see all data stored in a node (entriesN), etc.

```
insertN -key key -value value
retrieveN -key key
removeN -key key
entriesN
refsN
```

The refsN command lets the user consult the finger table of each node. If we check the finger tables every time we add a node to the network, it is possible to see how the predecessor and successors of each node vary. Comparing finger tables with Figure 4.1 it is possible to see how the network evolves and which position is occupied by each node.

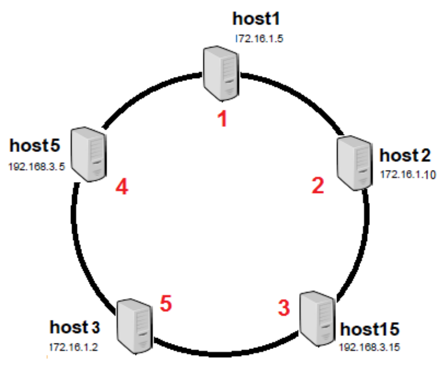


Figure 4.2: Chord ring evolution.

Figure 4.2 represents the position of the nodes in the ring, and the number in red is the order in which nodes joined the network. It is easy to observe that the ring changes dynamically when nodes are added or deleted. OpenChord works fine on simctl, so now we can create a new scenario and prepare the Lab session.

In section 3.2.2 some problems appeared when I tested Open Chord on Ubuntu 14.04. When one node created the network, a node from another machine was not able to join the ring, since several errors appeared. It was necessary to add one node in the same host from which the first node created the network, but from another terminal. But in simctl this problem does not occur. Another problem that appeared in Ubuntu 14.04 is that Open Chord detected 127.0.1.1 as the IP address of the host. Then, when obtaining the nodeID, it was calculated by sha1(ocsocket://127.0.1.1:port/).

So all nodes obtained their NodeID using the localhost IP address, so they were differentiated only by its TCP port. But again, this problem is not present in `simctl`.

4.2 aMule

Installation

To install aMule for a later use in SIMCTL, as before, we have to install it directly to the Debian file system from which SIMCTL runs the virtual machines. We will install it on debian6.fs, since we have installed OpenChord there, and the repositories are updated. First of all, we have to open a new terminal and go to the filesystems directory. After that, with the `simtools-fsupdate` command we access to the debian 6 filesystem.

```
cd /usr/share/vnuml/filesystems
sudo simtools-fsupdate debian6.fs
```

As I have said in 3.7.1, aMule can be installed as a monolithic client or as a daemon (without GUI). Once we are inside Debian 6, we only must install aMule-daemon. When the installation finishes we can exit from Debian 6 and go back to the Ubuntu 14.04 environment, since it does not require any additional tool.

```
apt-get install amule-daemon
exit
```

Configuration

aMule client

When amule-daemon is installed, a configuration file `amule.conf` is created. But instead of modifying this file in Debian 6, it is easier to install aMule on Ubuntu 14.04 and modify the configuration file there. Then this file will be copied to the aMule directories in Debian 6. By this way, any user will be able to modify the configuration in the graphical environment of Ubuntu 14.04 at any time before running `simctl`, so that users do not have to enter the Debian 6 file system with the risk of damaging it. So, in the Ubuntu 14.04 environment, we open a terminal and install aMule-daemon. When the installation finished, we run the aMule-daemon.

```
sudo apt-get install amule-daemon
amuled
```

The first time that the aMule-daemon runs, an aMule folder with all the required files is created. This directory is located at `~/.aMule`. And in this directory we can find the configuration file that we will modify and copy: `~/.aMule/amule.conf`. We must edit this file and modify two important parameters:

- AcceptExternalConnections=1 (in order to accept connections from other machines).
- IPLANfilter=0 (this enables the communication between machines from a local area network with private IP addresses).

The other parameters can be set as default. The aMule configuration file is done and ready to copy to the `simctl` scenario directory. So we create a new folder in `/usr/share/vnuml/scenarios/files` for aMule and copy the configuration file:

```
cd /usr/share/vnuml/scenarios/files
sudo mkdir /amule
cd amule
sudo mkdir config
cd config
sudo cp ~/.aMule/amule.conf ./
```

To run aMuleCMD (the aMule console client), first it is needed to start the daemon. To make it easier, I create a bash file that runs the daemon, waits some time until the daemon is completely running, and finally executes the aMule console. By this way, the user only has to run the bash file so that aMule is ready to use.

```
#!/bin/bash
# Executes amule-daemon and starts aMule console (amulecmd)
amuled $
sleep 2
amulecmd
```

Finally, I create one folder for each host. All the files stored in these files will be copied to the `/Incoming` folder in Debian 6 so that users can share them. By this way, before running the scenario, users can copy to these directories any file they want to share.

```
cd /usr/share/vnuml/scenarios/files/amule
mkdir host1
mkdir host2
mkdir host5
mkdir host15
```

eD2k server

Now we have to download and configure the files that will be used to create the eD2k server. We create a folder for storing server files in the `/amule` directory. Then, from the Lugdunum eserver repository [28], we download the last eserver version for Debian 6.

```
cd /usr/share/vnuml/scenarios/files/amule
sudo mkdir server
wget http://lugdunum.shortypower.org/files/eserver-17.14.x86_64-linux.nptl_ p.gz
```

The previous compressed file has to be decompressed and then we have to give execution permissions to it. For running eserver, a configuration file `donkey.ini` with some parameters has to be created. To make it easier, I create a bash file that creates a basic configuration file, decompresses eserver, gives permission and executes it. By this way, the user only has to execute the bash file and eserver starts.

```
#!/bin/bash
# Donkey.ini is the configuration file of the server. In this
# webpage http://mldonkey.sourceforge.net/Donkey.ini are listed
# all possible options. For our scenario, these few options
# are enough: Name, description, IP address and port of the
# server.
echo "[server]" >> donkey.ini
echo "name=aMule server 1" >> donkey.ini
echo "desc=aMule eD2k server test" >> donkey.ini
echo "thisIP=172.16.1.2 # The IP of my server" >> donkey.ini
echo "port=4232 # the TCP port (default is 4661, but it is wise
to choose another value)" >> donkey.ini

# Eserver17.15 is the last available version of eserver for i686
# linux in http://lugdunum.shortypower.org/files/
# This software let the user to create an eD2k server.
# These instructions unzip eserver, give execute permission and
# run it.
gzip -d eserver-17.15.i686-linux.nptl.gz
chmod 755 eserver-17.15.i686-linux.nptl
./eserver-17.15.i686-linux.nptl
```

Running a test scenario

The software is installed and ready to run, so now we must create a scenario with several machines to use aMule in a network. Before creating the lab session, I create a scenario with 1 server and 4 clients to test aMule in `simctl` and make sure everything works.

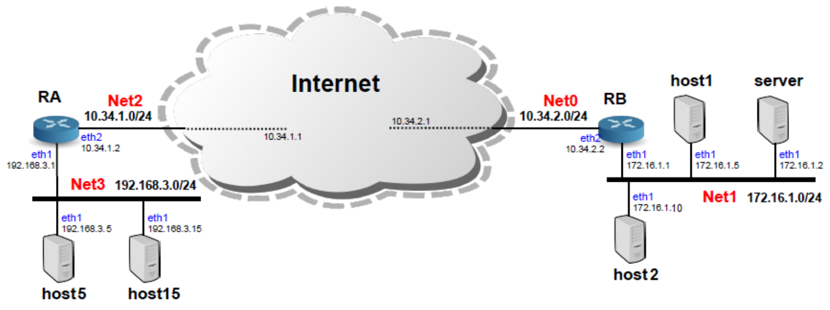


Figure 4.3: aMule test scenario.

Now, from a terminal, I start the test scenario and wait until it is created. Once it is running I access to the different nodes of the network to run aMule and eserver.

```
simctl test-aMule start
simctl test-aMule get host1
simctl test-aMule get host2
simctl test-aMule get host5
simctl test-aMule get host15
simctl test-aMule get server
```

From all the clients (host1, host2, host5 and host15), we run the aMule client console by running the bash file previously created:

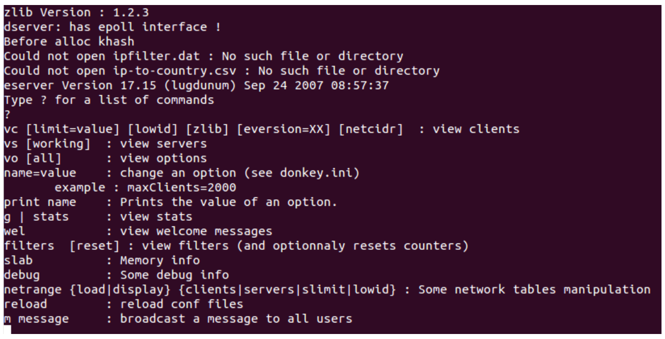
```
./amule
```

From the server node, we run the eD2k server by executing the bash file:

```
./eserver
```

eserver provides some commands (Figure 4.4) that can be showed by typing `?`. The most interesting command is `vc`, since it allows us to check who is connected to the server.

Now the server is running, but we have to connect all the clients to it. In each client, we can execute the `status` command to check if we are connected to any server or to



```
zlib Version : 1.2.3
dserver: has epoll interface !
Before alloc khash
Could not open ipfilter.dat : No such file or directory
Could not open ip-to-country.csv : No such file or directory
eserver Version 17.15 (lugdunum) Sep 24 2007 08:57:37
Type ? for a list of commands
?
vc [limit=value] [lowid] [zlib] [eversion=XX] [netcidr]  : view clients
vs [working]  : view servers
vo [all]      : view options
name=value    : change an option (see donkey.ini)
       example : maxClients=2000
print name    : Prints the value of an option.
g | stats     : view stats
wel           : view welcome messages
filters  [reset] : view filters (and optionnaly resets counters)
slab          : Memory info
debug         : Some debug info
netrange {load|display} {clients|servers|slimit|lowid} : Some network tables manipulation
reload        : reload conf files
m message     : broadcast a message to all users
```

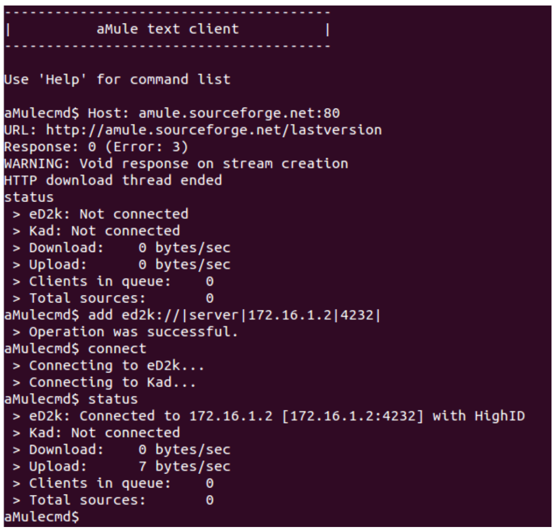
Figure 4.4: eserver commands.

the Kad network. But as we have no Internet connection, it cannot be connected to any network. So first of all we have to add the server to the server list in all the clients by:

```
add ed2k://|server|172.16.1.2|4232|
```

After that, when using the `connect` command, as it is the only server available, clients will connect to it. If we had several servers available, we could force the connection to one selected server by:

```
connect 172.16.1.2:4232
```



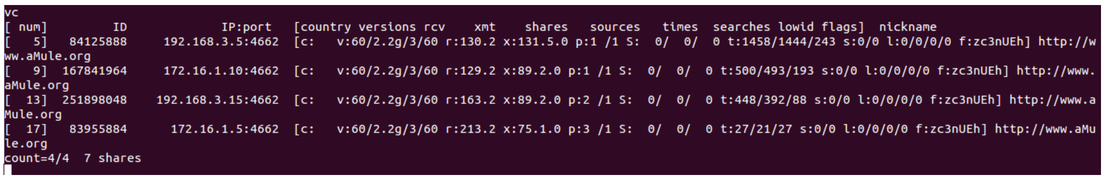
```
----------------------------------------
|          aMule text client           |
----------------------------------------

Use 'Help' for command list

aMulecmd$ Host: amule.sourceforge.net:80
URL: http://amule.sourceforge.net/lastversion
Response: 0 (Error: 3)
WARNING: Void response on stream creation
HTTP download thread ended
status
 > eD2k: Not connected
 > Kad: Not connected
 > Download:    0 bytes/sec
 > Upload:      0 bytes/sec
 > Clients in queue:    0
 > Total sources:       0
aMulecmd$ add ed2k://|server|172.16.1.2|4232|
 > Operation was successful.
aMulecmd$ connect
 > Connecting to eD2k...
 > Connecting to Kad...
aMulecmd$ status
 > eD2k: Connected to 172.16.1.2 [172.16.1.2:4232] with HighID
 > Kad: Not connected
 > Download:    0 bytes/sec
 > Upload:      7 bytes/sec
 > Clients in queue:    0
 > Total sources:       0
aMulecmd$
```

Figure 4.5: aMule client connection to eD2k server.

At this point all clients have connected to the server. From the server console, we can check it by using the `vc` command (Figure 4.6). It shows the IP address of all peers connected to it, and the total number of files ready to share.



```
vc
[ num]          ID              IP:port   [country versions rcv     xmt     shares   sources   times  searches lowid flags]  nickname
[   5]   84125888     192.168.3.5:4662  [c:   v:60/2.2g/3/60 r:130.2 x:131.5.0 p:1 /1 S:  0/  0/  0 t:1458/1444/243 s:0/0 l:0/0/0/0 f:zc3nUEh] http://w
ww.aMule.org
[   9]  167841964     172.16.1.10:4662  [c:   v:60/2.2g/3/60 r:129.2 x:89.2.0 p:1 /1 S:  0/  0/  0 t:500/493/193 s:0/0 l:0/0/0/0 f:zc3nUEh] http://www.
aMule.org
[  13]  251898048    192.168.3.15:4662  [c:   v:60/2.2g/3/60 r:163.2 x:89.2.0 p:2 /1 S:  0/  0/  0 t:448/392/88 s:0/0 l:0/0/0/0 f:zc3nUEh] http://www.a
Mule.org
[  17]   83955884      172.16.1.5:4662  [c:   v:60/2.2g/3/60 r:213.2 x:75.1.0 p:3 /1 S:  0/  0/  0 t:27/21/27 s:0/0 l:0/0/0/0 f:zc3nUEh] http://www.aMu
le.org
count=4/4  7 shares
```

Figure 4.6: eserver console showing clients connected to it.

Once all the clients are connected to the same server, we can search any file and download it. To search a file in the local network we must run:

```
search local filename
```

After that, an error will be shown:

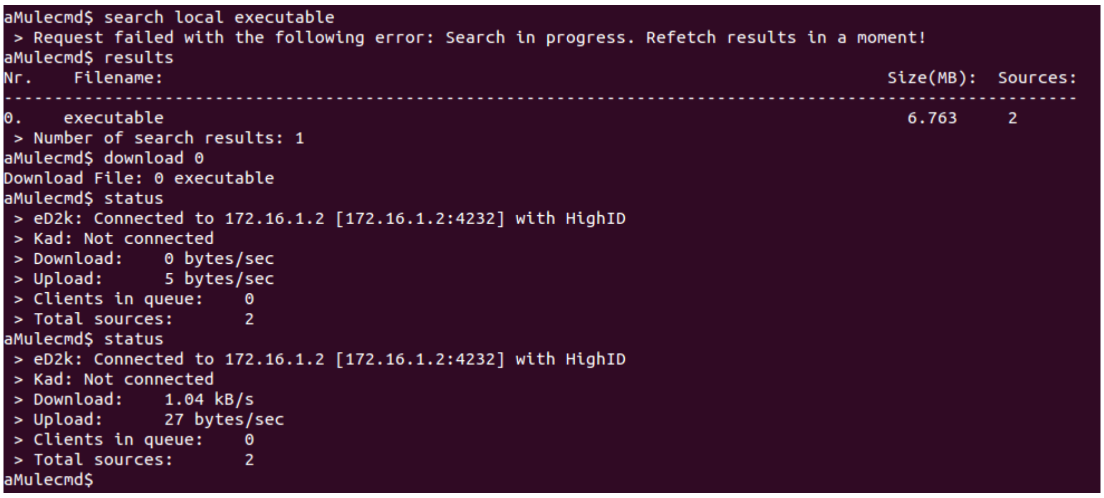
```
> Request failed with the following error: Search in progress.
  Refetch results in a moment!
```

But this is not important. The search has been successfully done and the results can be obtained by typing:

```
results
```

Then, a list with all results will be shown. It shows the different found files with the same name, specifying its size and how many nodes have the file. To download a file, it has to be used the command `"download #"` where "#" is the number of the file in the list. After that, the download starts automatically. By executing the `status` command, we can observe the progress of the download.

The network works fine. Now I must create a simpler scenario for the Lab session. The network of this test scenario is based on other scenarios focused on tunneling and multicast, so the network is a bit complex, with several subnetworks and routers. This makes that the scenario takes more than 2 minutes to start. As here the network is not important, a simpler scenario will reduce the execution time.



```
aMulecmd$ search local executable
 > Request failed with the following error: Search in progress. Refetch results in a moment!
aMulecmd$ results
Nr.    Filename:                                                                          Size(MB):  Sources:
---------------------------------------------------------------------------------------------------------------
0.    executable                                                                            6.763     2
 > Number of search results: 1
aMulecmd$ download 0
Download File: 0 executable
aMulecmd$ status
 > eD2k: Connected to 172.16.1.2 [172.16.1.2:4232] with HighID
 > Kad: Not connected
 > Download:    0 bytes/sec
 > Upload:      5 bytes/sec
 > Clients in queue:    0
 > Total sources:       2
aMulecmd$ status
 > eD2k: Connected to 172.16.1.2 [172.16.1.2:4232] with HighID
 > Kad: Not connected
 > Download:    1.04 kB/s
 > Upload:      27 bytes/sec
 > Clients in queue:    0
 > Total sources:       2
aMulecmd$
```

Figure 4.7: Search and download process.

Chapter 5

Lab session

Contents

0.1 Introduction 1
0.2 aMule 2
  0.2.1 The scenario of the lab session 2
  0.2.2 The branches 2
  0.2.3 Starting the scenario using simctl 3
  0.2.4 Creating the eD2k network 3
  0.2.5 Sharing files 4
0.3 Open Chord 6
  0.3.1 The scenario of the lab session 6
  0.3.2 The branches 6
  0.3.3 Starting the scenario using simctl 6
  0.3.4 Creating the Chord network 7
  0.3.5 Storing data 8
  0.3.6 New node 10
  0.3.7 Retrieving data 11
  0.3.8 Removing data 12
  0.3.9 Node failure 12

0.1 Introduction

Peer-to-Peer (P2P) networks are worldwide distributed systems where each node can be used both as a client and a server simultaneously. These networks emerged as an incipient paradigm of communications to share resources and services in a highly decentralized way. The operation of the P2P networks has changed over the years, trying to adapt (and survive) to several different problems, generating up to three different generations.

First generation of P2P networks are known as *centralized P2P networks* because they depend on centralized servers to perform some functions, typically a centralized directory to find resources. This type of P2P networks suffers from a single point of failure, since the network may stop working if the central server goes down.

The second generation of P2P networks appeared to avoid the above vulnerability, but they needed two phases to achieve the expected success. The first attempt was to design *pure decentralized P2P networks*, which use a P2P scheme in all their processes, and there is no central server at all. The second attempt to improve the P2P networks was to introduce some degree of centralization leading to the *hybrid P2P networks*, where some nodes (supernodes) manage certain extra functions.

Finally, the third generation of P2P networks emerged. They are totally decentralized networks, so there is not a point of failure, but with a certain structuring of resources. For this reason they are called *structured P2P networks* or also *structured P2P overlays*, as they create an *overlay*.

The aim of this lab session is to get in touch with some peer-to-peer network applications. We pretend to show a practical application of peer-to-peer networks, being able to interact with them by storing some data and observing its behaviour when a node joins or leaves the network. First we will run aMule, a P2P file sharing application that works with the eD2k network. eD2k can be considered from the second generation of P2P networks, since it is semi-centralized and there is not a single point of failure. Then we will run Open Chord, an implementation of the Chord P2P overlay. Chord is one of the four original structured networks (3rd generation) created in early 2000.

0.2 aMule

aMule is a free peer-to-peer file sharing application that works with the eDonkey and Kad networks, offering similar features to eMule but supporting multiple platforms. The eDonkey Network (eDonkey2000 network or eD2k) is a "semi-decentralized", mostly server-based, peer-to-peer file sharing network built to share big files among users, and to provide long term availability of files. It was developed in 2000 by US company MetaMachine in order to use the client eDonkey2000. In September 2006 eDonkey2000 closed its doors due to legal issues, but the eD2k network survived, since it was being used by other clients such as eMule or Lphant. As eD2k is server-based, the network would be affected if servers went down, so eMule Project developed a global Kademlia network (Kad) in order to overcome the reliance on central servers.

0.2.1 The scenario of the lab session

The aim of this exercise is to get in touch with the eD2k network. To do so, we are going to use two eD2k servers and four nodes that will run the aMule client. The scenario of this laboratory session is depicted in Figure 1.

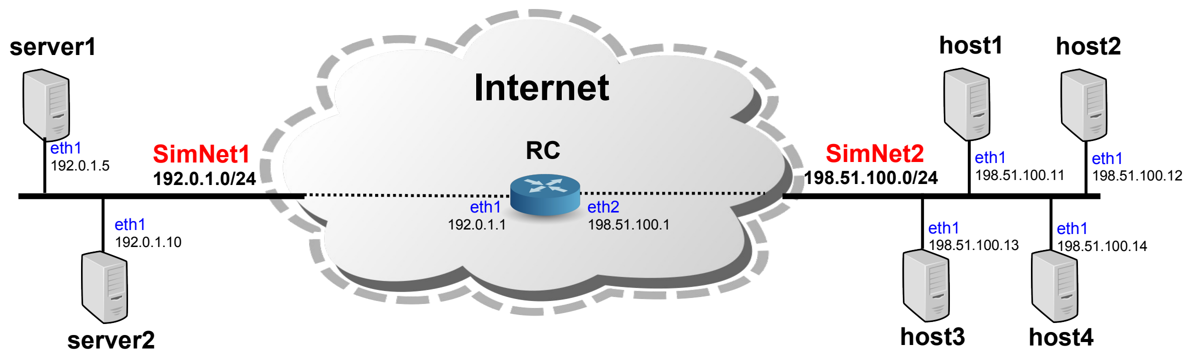


Figure 1: aMule scenario.

0.2.2 The branches

We will assign the following IP addresses to the different branches:

Branch	IP Network	Public or private
SimNet1	192.0.1.0/24	public addressing
SimNet2	198.51.100.0/24	public addressing

This network is not a real scenario, but our goal is to run P2P networks, so we are not focused on the architecture of the network. SimNet2 contains four hosts that will run the aMule client, while in SimNet1 there are two machines that will run an eD2k server.

0.2.3 Starting the scenario using simctl

First of all, we will start the virtual scenario with the command:

```
host$ simctl amule start
```

When starting this scenario, IP addresses in network interfaces and routing tables should be properly configured. If not, maybe there is a problem in the scenario or in `simctl`, so contact the professor.

0.2.4 Creating the eD2k network

Once the scenario is running we must run the aMule client in each host (**host1**, **host2**, **host3** and **host4**). The aMule software can be installed as a monolithic client, which has a graphical user interface (GUI) to search and download the files, and a console-based version that runs without GUI. This second version is what we will use. First we have to run a daemon called `amule-daemon`, and then we will be able to run `amulecmd`, that executes the aMule client environment. To make it easier, each host contains a bash file that executes the daemon, waits some time so that the daemon is ready, and finally executes `amulecmd`. So, from each host we must run:

```
host$ ./aMule
```

After that, the aMule console (`amulecmd`) will start. Now from any host with the aMule client running, we run the `help` command and all the available commands are shown. Then, run it on **host1** to see the main options provided by the aMule client.

```
host1:aMulecmd$ help
```

By typing the `status` command some information about the client is shown.

```
host1:aMulecmd$ status
```

To share and download files we must be connected to a network. In the shown information, we can observe that the client is not connected to any network. To connect to the eD2k network, aMule uses a file called `server.met` that contains a list of eD2k servers. To connect the Kad serverless network, aMule has a file called `nodes.dat` which stores details about known Kad nodes, and this is used to bootstrap the Kad network when aMule starts. Both files have to be uploaded periodically, since these nodes change with the time. But in this scenario we do not have Internet access, so we must create our own eD2k servers.

There are two families of server software for the eD2k network:

- The original one from MetaMachine, developed by eDonkey creators.
- `eserver`, written from scratch by a person (Lugdunum). Used by almost all eD2k servers.

We will use the `eserver` software to create eD2k servers on **server1** and **server2**. To run the software we must execute a bash file. This file creates the configuration file (`donkey.ini`) of the server and executes the software. Run the following command in **server1** and **server2**.

```
server1$ ./eserver
```

Now we have two eD2k servers running. Type `?` in the `eserver` console to see a list of the commands that it supports. With the `vc` command we can see all the clients connected to the server. At this point no client is connected, so now we go back to the hosts to connect them to the servers.

aMule client allows us to add an eD2k server manually by typing its address and port. The TCP port of the server is written in the configuration file of the server (`donkey.ini`). So now we will add **server1** and **server2** to the server list of **host1**.

```
host1:aMulecmd$ add ed2k://|server|192.0.1.5|4005|
host1:aMulecmd$ add ed2k://|server|192.0.1.10|4010|
```

After that, both servers have been added to the **host1**'s server list. Now, if we try to connect the client to any network, it will choose the first available server of the server list.

1. Open the `wireshark` network analyzers in the PHYSICAL HOST and capture traffic in **SimNet1**.
2. Use the `connect` command in **host1** to connect to a server.

   ```
   host1:aMulecmd$ connect
   ```

   (a) What type of messages do you see in `wireshark`?

   (b) Which nodes of the network are involved?

Now add **server1** and **server2** to the server list of the resting hosts. If we execute the `connect` command, all the hosts would connect to **server1**, since it has been added before **server2**. But we can also force the connection to a particular server. So we will connect **host2** and **host3** to **server1**, and **host4** to **server2**.

```
host2:aMulecmd$ connect
host3:aMulecmd$ connect
host4:aMulecmd$ connect 192.0.1.10:4010
```

From any host, we can verify that we are connected to the corresponding server by:

```
host4:aMulecmd$ status
```

And from each server, we can check which nodes are connected to it:

```
server1$ vc
```

0.2.5 Sharing files

At this point the eD2k network has been created. There are three aMule clients connected to one server and another host connected to the second server. Now we will make use of the file sharing system. To look for a file, we have to use the `search` command specifying the type of search: it can be global, local or kad. In our case the search can be local or global.

From **host1** we will try to search the file **host_2**, that is stored in the Incoming directory from **host2**.

```
host1:aMulecmd$ search local host_2
```

After that, an error will be shown:

```
> Request failed with the following error: Search in progress.
  Refetch results in a moment!
```

But this is not important. The search has been successfully done and the results can be obtained by typing:

```
host1:aMulecmd$ results
```

Then, a list with all the results is shown. It also informs about the size of the file and the number of nodes that have the file (sources). In our case, as this file is only located in **host2**, there is only one result. To download the file, we must run:

```
host1:aMulecmd$ download 0
```

where 0 is the identifier of the file in the results list. After that, we can run `status` and observe that the download speed is not 0 bytes/sec, since the file is being downloaded.

1. Open `wireshark` in the HOST and capture traffic in **SimNet1**.

2. Search the file **host_2** in **host3** and observe the results list:

```
host3:aMulecmd$ search local host_2
host3:aMulecmd$ results
```

   (a) Looking at the results list, which is the difference with the previous search?

   (b) Looking at `wireshark`, which nodes of the network are involved?

3. Restart the running live capture in `wireshark`. Now download the file:

```
host3:aMulecmd$ download 0
```

   (a) Looking at `wireshark`, which nodes of the network are involved? Why?

Now we will see the difference between a local and a global search. From **host1** we will try to search the file **host_10**.

1. Open the `wireshark` network analyzers in the PHYSICAL HOST and capture traffic in **SimNet1**.

2. Make a local search of the file **host_10** in **host1** and observe the results list:

```
host1:aMulecmd$ search local host_10
host1:aMulecmd$ results
```

   (a) Looking at the results list, is there any coincidence?

   (b) Looking at `wireshark`, which nodes of the network are involved?

3. Restart the running live capture in `wireshark`. Now we will make a global search:

```
host1:aMulecmd$ search global host_10
host1:aMulecmd$ results
```

   (a) Looking at the results list, is there any coincidence? Why it is different than before?

   (b) Looking at `wireshark`, which nodes of the network are involved? Why?

   (c) What is the difference between a local and a global search in the eD2k network?

As we are going to continue this lab session with another scenario, close this one:

```
host$ simctl amule stop
```

0.3 Open Chord

Open Chord is an open source implementation of the Chord P2P overlay. It is available for free under GNU General Public License (GPL) and was developed by the Distributed and Mobile Systems Group of Bamberg University.

Open Chord provides the possibility to use the Chord distributed hash table within Java applications by providing an API to store all serializable Java objects within the distributed hash table. On the other hand, it provides a console-based experimentation environment to explore the functionality of a Chord network, allowing to create a Chord ring between different nodes and storing data between them. So this tool is what we are going to use.

The console-based environment allows to create a virtual Chord network with several virtual nodes on a single host. But we are going to use it by creating a ring between several hosts on the same local network.

0.3.1 The scenario of the lab session

The aim of this exercise is to get in touch with the Chord network. To do so, we are going to use Open Chord to create a Chord ring with six nodes. The scenario of this laboratory session is depicted in Figure 2.

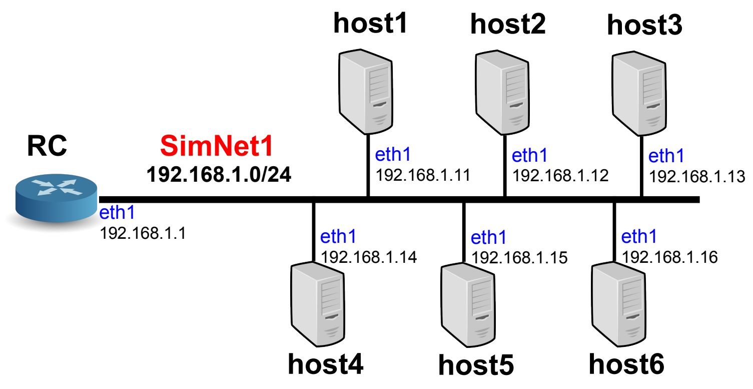


Figure 2: Open Chord scenario.

0.3.2 The branches

In this scenario we only have a branch. We will assign the following IP addresses to **SimNet1**:

Branch	IP Network	Public or private
SimNet1	192.168.1.0/24	private addressing

We have a single local network with one router and six hosts. The network is simplified since we are interested in working with Open Chord, and this exercise is not focused on the architecture of the network.

0.3.3 Starting the scenario using simctl

First of all, we will start the virtual scenario with the command:

```
host$ simctl openchord start
```

When starting this scenario, IP addresses in network interfaces and routing tables should be properly configured. If not, maybe there is a problem in the scenario or in `simctl`, so contact the professor.

0.3.4 Creating the Chord network

Once the scenario is running we must run Open Chord in all the hosts (**host1**, **host2**, **host3**, **host4**, **host5** and **host6**). To run Open Chord, a long java instruction has to be executed. It contains the paths of the classes and libraries in the project that have to be run. To make it easier, this instruction has been written in a bash file. So we execute the bash file in all the hosts.

```
host1$ ./openchord
host2$ ./openchord
host3$ ./openchord
host4$ ./openchord
host5$ ./openchord
host6$ ./openchord
```

When the console starts, we consult the available commands by using the `help` command. Then, a list with several commands is shown. We should realize that some commands end in N. This is because Open Chord provides two types of execution. The commands ending in N are used to create a ring in a real network between nodes in different machines, and the other commands are used to simulate several nodes on a single machine. Therefore, we will use commands ending in N.

At this point, Open Chord is running in all the hosts, but the network has not been created yet. We will create a Chord ring from **host1**.

```
host1:oc > joinN
```

Then, from **host2**, we add the node to the network by using **host1** as the bootstrap node. When joining the network, we have to indicate the TCP port that we will use for sending and receiving data, and then the IP address and the port of the bootstrap node. The TCP port of **host1** is 4242, since it is the default port of the node that creates the ring. So now we add **host2** to the network:

1. Open the `wireshark` network analyzer in the PHYSICAL HOST and capture traffic in **SimNet1**.

2. Add **host2** to the network:

   ```
   host2:oc > joinN -port 4202 -bootstrap 192.168.1.11:4242
   ```

   (a) Look at `wireshark`. What has happened?

3. Execute the following command in **host1** and **host2**:

   ```
   host1:oc > refsN
   host2:oc > refsN
   ```

   (a) What does this command show?

A Chord ring with only two nodes is not very useful, so now we are going to add three more nodes to the network. We will add **host3** by using **host1** as the bootstrap node, **host4** by using **host3** as the bootstrap node, and **host5** by using **host2** as the bootstrap node.

```
host3:oc > joinN -port 4203 -bootstrap 192.168.1.11:4242
host4:oc > joinN -port 4204 -bootstrap 192.168.1.13:4203
host5:oc > joinN -port 4205 -bootstrap 192.168.1.12:4202
```

1. Open the `wireshark` network analyzer in the PHYSICAL HOST and capture traffic in **SimNet1**. You can see hundreds of packets sent between all hosts. Open Chord establishes a TCP connection between nodes and lots of messages are sent periodically to inform about the status of each host. If you try to "follow TCP stream" you will see that no valuable information can be extracted. So from now, we will focus on the behaviour of the Chord ring, and will not enter in details in the sent packets.

2. Execute the `refsN` command in **host1**, **host2**, **host3**, **host4** and **host5**:

```
host1:oc > refsN
host2:oc > refsN
host3:oc > refsN
host4:oc > refsN
host5:oc > refsN
```

(a) Do you observe any changes in **host1** and **host2**?

(b) Using the information provided in the obtained tables, and looking at Figure 2, you will be able to fill the following figure:

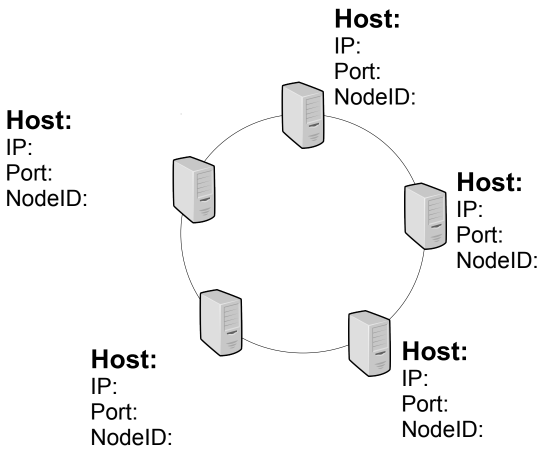


Figure 3: Chord ring with 5 nodes.

(c) After filling the Figure 3, look at the first 2 bytes of the nodeID of each node. How are the nodes ordered? (Translate the first 2 bytes from hexadecimal to decimal and you will see it clearer).

0.3.5 Storing data

Chord is a distributed hash table that stores key-value pairs by assigning keys to different nodes; a node will store the values for all the keys for which it is responsible. Nodes and keys are assigned an m-bit identifier using consistent hashing. The SHA-1 algorithm is the base hashing function for consistent hashing.

Open Chord also uses the SHA-1 algorithm. When a **key** is inserted, it applies the hash function to the key and then stores the 2 first bytes to set the key identifier (keyID). For example, if we insert the pair key=`key` and value=`value`, Open Chord will calculate `sha1(key)=a62f2225bf70bfaccbc7f1ef2a397836717377de`, and the keyID of `key` will be `A6`. This identifier will be later compared with the nodeIDs of the successors list to select where to store the value. We can check it manually by typing the following command in an Ubuntu terminal:

```
echo -n "key" |sha1sum
```

When a **node** joins the network, its identifier has to be calculated. To obtain the identifier of a node (nodeID), Chord calculates the hash function of the IP address and port of the node to ensure that there will not be two nodes with the same nodeID. Open Chord calculates the hash of the entire string `ocsocket://ipaddress:port/`. After that, to set the nodeID, it selects only the first 8 bytes of the result. For example, for **host3**, Open Chord will calculate:

sha1(ocsocket://192.168.1.13:4203/)=85eedc1d74391a29f4650312a197d3e2541a4a71 and its identifier will be 85 EE DC 1D. We can check it manually by typing the following command in an Ubuntu terminal:

```
echo -n "ocsocket://192.168.1.13:4203/" |sha1sum
```

In a Chord ring, nodes are ordered numerically depending on their nodeID. When a key-value pair is inserted into the network, the identifier of the key is obtained. Then, the value is stored at its key's successor node. The successor node of a key k is the first node whose ID equals to k or follows k in the identifier circle. Open Chord compares the first 2 bytes of the keyID and the nodeID to determine which is the key's immediate successor. Since the successor (or predecessor) of a node may disappear from the network (because of failure or departure), each node records a whole segment of the circle adjacent to it (its successors and predecessor).

At this point we know how values are stored in the corresponding nodes. So now we will insert some key-value pairs into the network to see a practical example.

1. From **host3**, insert the pair key=key0 value=value0.

   ```
   host3:oc > insertN -key key0 -value value0
   ```

2. In an Ubuntu terminal (NOT in the Open Chord console of any host), obtain the identifier of the key by:

   ```
   echo -n "key0" |sha1sum
   ```

   The obtained hash is sha1(key0)=adb1ef332d1f6e99e809fb9b00a08efcad930e82, so the key will be AD.

3. Now we want to know where this data is stored. From **host3**, consult the data that it is storing by typing:

   ```
   host3:oc > entriesN
   ```

   We can realize that **host3** is not storing the data that it has just inserted. Consult the stored data in the resting nodes:

   ```
   host1:oc > entriesN
   host2:oc > entriesN
   host4:oc > entriesN
   host5:oc > entriesN
   ```

   (a) Which nodes are storing the data?

4. From **host3**, insert another key-value pair: key=key1 value=value1.

   ```
   host3:oc > insertN -key key1 -value value1
   ```

5. Obtain the identifier of key1 and use the refsN command in all nodes.

   (a) Which nodes are storing the new data? Do you imagine why?

The previous values are stored in different nodes since the key identifiers are different. When a node inserts a key-value pair, it obtains the identifier of the key. Then, the value is stored in the key's immediate successor. Once it has been stored, the storing node replicates the value to its successors (the 2 following nodes). By this way, if the storing node dies, data will remain in the network.

As we have seen, the values will be stored in the key's successor node. So it does not matter from which node we insert the key-value pairs, since they will be stored in the same node.

1. Insert the following key-value pairs from any node. For example:

```
host1:oc > insertN -key key2 -value value2
host4:oc > insertN -key key3 -value value3
host3:oc > insertN -key key1 -value telem
host2:oc > insertN -key key1 -value UPC
```

2. Use the `refsN` command in all nodes.
   (a) What does it happen if different values have the same key?

0.3.6 New node

Now we will add a **host6** to the network and see what happens when it joins the ring.

1. From an Ubuntu terminal (NOT in the Open Chord console of any host) obtain the nodeID that will be assigned to **host6**.

```
echo -n "ocsocket://192.168.1.16:4206/" |sha1sum
```

   (a) Which will be its nodeID?
   (b) Where is it going to be located in the Chord ring? Fill Figure 4 with the new look it will have. Compare the 2 first bytes of all nodeIDs to know their position in the ring.

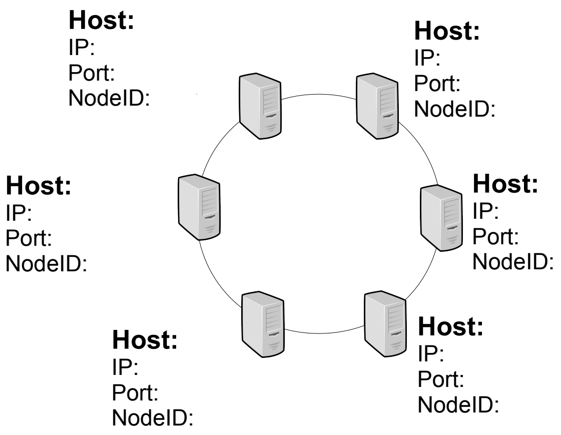


Figure 4: Chord ring with 6 nodes.

The following tasks are done for a newly joined node n:

- Initialize node n (the predecessor and the finger table).
- Notify other nodes to update their predecessors and finger tables.
- The new node takes over its responsible keys from its successor.

The predecessor of n can be easily obtained from the predecessor of successor(n) (in the previous circle). Then, it initializes the finger table from its immediate neighbours and make some updates. Finally, its successor transfers to node n all the data of which the n node will be the new responsible.

1. Before adding **host6** to the ring, execute the `entriesN` command in all the existing hosts (**host1, ..., host5**) to see which files values they are storing.
2. Join **host6** to the network by using **host4** as the bootstrap node.

```
host6:oc > joinN -port 4206 -bootstrap 192.168.1.14:4204
```

3. Use the `refsN` command and verify that Figure 4 is correctly filled. If the predecessor appears as null in **host6**, wait some time and execute the command again. It takes some time to initialize a new node completely.
4. Use the `entriesN` command in **host6**.
   (a) Which data is **host6** storing?
5. Execute the `entriesN` command again in all the resting hosts (**host1, ..., host5**) to see which values they are storing.
   (a) Has any node lost some data? Why?

0.3.7 Retrieving data

To get a value, the command `retrieveN` has to be used. We will ask for the key `key1`. The values associated to this key are stored on **host1**, **host6** and **host5**, it is, the three nodes located before **host4**.

1. Retrieve the values of `key1` from **host4**.

```
host4:oc > retrieveN -key key1
```

   (a) Which data are returned?

2. Execute `refsN` in **host4** to see its finger table. We obtain:

```
Finger table:
 85 ocsocket://192.168.1.13:4203/(0-155)      (host3)
 0F ocsocket://192.168.1.12:4202/ (156-159)   (host2)
```

The keyID of `key1` is `10`. Then, looking at the finger table, we realize that:

```
      0F      <  10   <      85        (hexadecimal)
      15      <  16   <      133       (decimal)
NodeID(Host2) < keyID < NodeID(Host3)
```

Between these two nodes, the host with nodeID=`85` would be the responsible of the key. Therefore, **host4** asks **host3** for retrieving the associated values. As **host3** is not storing these values, it looks at its finger table:

```
Finger table:
 0F ocsocket://192.168.1.12:4202/ (0-159)    (host2)
```

As **host3** has only one node in its finger table, it asks to **host2** if it has the values associated to `key1`. As **host2** is not storing these values, it looks at its finger table:

```
Finger table:
 2E ocsocket://192.168.1.11:4242/(0-156)     (host1)
 4A ocsocket://192.168.1.16:4206/(157)       (host6)
 55 , ocsocket://192.168.1.15:4205/ (158)    (host5)
```

Looking at the finger table we realize that:

```
 10   <      2E       <      4A       <      55        (hexadecimal)
 16   <      46       <      74       <      85        (decimal)
keyID < NodeID(Host1) < NodeID(Host6) < NodeID(Host5)
```

Between these three nodes, the host with nodeID=`2E` would be the responsible of the key. Therefore, **host2** asks **host1** for retrieving the associated values. As **host1** is storing the values associated to `key1`, it sends the data to **host4**.

(a) Retrieve the values of `key0` from **host5** and explain the followed process.

0.3.8 Removing data

To remove a key-value pair, the command `removeN` has to be used. The key and the value have to be specified, since there can be several values assigned to a single key.

1. From any node, remove one pair. For example:

```
host4:oc > removeN -key key1 -value value1
```

2. Execute the `entriesN` in all the hosts to ensure that it has been removed.

0.3.9 Node failure

Finally, we will check the fault tolerance skill of a Chord network. We will try to leave the network by killing the node without informing the other nodes, and then we will leave correctly.

1. Go to **host2** and abort the application by pressing `Ctrl + c`

```
host2:oc > ^C
```

The Open Chord console stops immediately.

2. Execute the `refsN` command in the resting nodes.
   (a) What has happened?
   (b) Which nodes have been affected?
3. Execute the `entriesN` command in the resting nodes.
   (a) What has happened?
4. Now execute the `leaveN` command in **host5**

```
host5:oc > leaveN
```

The node leaves the network informing to the other nodes of the ring. Then it returns to the Open Chord main menu, ready for creating or joining a new ring.

5. Execute the `refsN` and `entriesN` commands in the resting nodes.
    (a) What has happened?
    (b) Has any data been lost?

Realize that all the data has been redistributed between the surviving nodes, and all the finger tables and successors/predecessors have been updated. It does not matter if the nodes leave the network correctly, informing the resting nodes, or it stops suddenly because of a failure. The nodes are periodically sending messages to their predecessors/-successors to inform about their status. When a node shows no sign of life, the resting nodes understands that it has fallen, so they update the network.

Remember to close the scenario:

```
host$ simctl openchord stop
```

Chapter 6

Conclusions and Further Work

P2P systems became an interesting area since early 2000. Researchers conducted a large amount of research in some challenging areas such as security, reliability, flexibility, load balancing, etc. To validate their theories, they had to check the experiments of their research and reproduce their results. For that purpose, simulators and implementations of their networks appeared. As those initial systems are still taught in universities, some student's implementations exist on opensource repositories, as well as free implementations that were used for research and teaching. Other implementations that were available some years ago became a part of a commercial software, so they disappeared.

Other type of P2P networks became popular with the appearance of file sharing systems, and they have survived through the time. But the availability of increasingly cheap bandwidth and the appearance of streaming solutions have make that P2P networks passed into the background. It is possible that P2P networks resurface with the appearance of 5G technology, which is expected to be launched on 2020. The increase of the IoT (Internet of Things) and the use of M2M (machine to machine) networks could need the use of P2P technology.

The objective of this project was to search and find any P2P implementation able to run in a real network between different machines, and able to run in a console environment with no GUI. If possible, the idea was to find the original codes of structured P2P overlays developed by the authors of the original publications. After an extensive research, I realized that the original projects were abandoned several years ago. So I continued looking for other implementations available on the Internet. Most of the found implementations were unfinished or were abandoned several years ago. Moreover, most of them were rarely documented or were not faithful to the original publications. Other implementations that were important in their time are currently dead and all the files and documentation have been removed from the Internet. Nevertheless, I tested almost every code I found.

To test all the found implementations, I created a virtual network with four Ubuntu 14.04 virtual machines, making use of the Oracle VM VirtualBox software. The used virtual image is the one used in the lab of some subjects of network engineering. Therefore, this image contains the `simctl` platform. Before testing codes and applications on `simctl`, I tested them in the virtual network to see if they met the requirements. By this way, I could test if they were able to run between nodes located in different machines. When a software seemed interesting and useful for the objective of this master thesis, I installed it in the filesystem used by `simctl` so that I could run it on a simple scenario.

Two software were finally selected: aMule, a file sharing client that runs over the eD2k network, and Open Chord, a Chord implementation that provides a console-based mode to interact with the Chord network. Once the software was selected, I installed them in the filesystem used by `simctl` and developed different scenarios. For aMule, I modelled an existent scenario by adding some hosts that would run the aMule client, and another node that would run an eD2k server software. I tested all the available options and shared different files, and everything worked fine. Then I created an Open Chord scenario by modifying the aMule file. I ran the Open Chord console in several hosts and tested all the available actions: to create a Chord ring, to add or delete some nodes, to insert data, etc. And it also worked perfectly.

Finally, I created from scratch two different scenarios in order to reduce the execution time. As the lab session should be focused in peer-to-peer studying, a simple network would be enough. These new scenarios have more nodes than the previous tests, in order to simulate a more realistic scenario. After that, a lab session guide was created so that the student must test all the possible actions and must answer some questions that will help him to understand the behaviour of the network.

The created lab session is a first version, so it has to be revised or modified before creating a final lab session to be used in a network engineering subject. The professor could adapt it in relation to the topics explained in class. For a future work, as Open Chord is an open source software, and its source code is provided, it could be modified and expanded to apply security options to the software. It could be used for security research, trying to corrupt data or to add malicious nodes. And then, it could be used to teach some topics about security in peer-to-peer networks.

References

[1] aMule Home Page. http://www.amule.org. 11, 54, 55

[2] Bitcoin Home Page. https://bitcoin.org. 8

[3] BitTorrent Home Page. http://www.bittorrent.com. 7

[4] BitTorrent Sync Home Page. http://www.bittorrent.com/sync. 8

[5] Chimera. http://current.cs.ucsb.edu/projects/chimera/. 39, 40

[6] Chordial. https://github.com/mattwilliamson/chordial. 29

[7] Chordjerl. https://github.com/jashmenn/chordjerl. 30

[8] CoolStreaming Home Page. http://www.coolstreaming.us. 8

[9] Debian Lenny Home Page. https://wiki.debian.org/DebianLenny. 71

[10] Debian Squeeze Home Page. https://wiki.debian.org/DebianSqueeze. 71

[11] DSL. Distributed Storage Leasing. https://sourceforge.net/projects/dsl/. 52

[12] Edonkey Home Page (currently unavailable). http://www.edonkey.co.nr. 10

[13] eMule Project Home Page. http://www.emule-project.net. 11

[14] Entangled. http://entangled.sourceforge.net/. 61

[15] FreePastry. http://www.freepastry.org/FreePastry/. 34

[16] FreePastry Demo. http://www.csg.uzh.ch/teaching/fs09/p2p/lectures/pastrydemo. 37

[17] FreePastry Trac-Wiki. https://trac.freepastry.org. 34

[18] Gnutella Home Page. http://rfc-gnutella.sourceforge.net. 9

[19] GnuTLS versions. https://launchpad.net/gnutls/+packages. 44

[20] Hive2Hive. https://github.com/Hive2Hive/Hive2Hive. 46

[21] Jamendo Home Page. http://www.jamendo.com. 8

[22] JChord. https://github.com/scorpiovn/joonion-jchord/tree/master/JChord. 30

[23] jDHTUQ Home Page. https://sourceforge.net/projects/jdhtuq/. 30

[24] JXSE. The Java Implementation of the JXTA Protocols. https://jxse.kenai.com/. 51

[25] JXTA. The Language and Platform Independent Protocol for P2P Networking. https://jxta.kenai.com/. 18, 51

[26] Lphant Home Page (no longer available). http://es.lphant.com/. 54

[27] Lugdunum eserver repository. http://lugdunum.shortypower.org/files/. 56

[28] Lugdunum Home Page. http://lugdunum.shortypower.org/. 54, 56, 80

[29] MaidSafe-Common library. https://github.com/maidsafe-archive/MaidSafe-Common. 49

[30] MaidSafe-DHT library. https://github.com/maidsafe/MaidSafe-DHT. 48

[31] MaidSafe official website. https://maidsafe.net/. 49

[32] MaidSafe super-project. https://github.com/maidsafe-archive/MaidSafe. 48

[33] MaidSafe-Transport library. https://github.com/maidsafe/MaidSafe-Transport. 49

[34] MIT Chord/DHash HowTo. https://github.com/sit/dht/wiki/howto. 25

[35] MIT Chord/DHash repository. https://github.com/sit/dht. 24, 25

[36] Napster Home Page. http://www.napster.com. 9

[37] NGI-Lab: Kademlia-based DHT. http://www.delta-my.de/rande/kademlia_en.php. 62

[38] OceanStore official website. http://www.oceanstore.org/. 41

[39] OceanStore SourceForge. http://oceanstore.sourceforge.net/. 41

[40] Open Chord Home Page. http://open-chord.sourceforge.net. 26

[41] Open Chord repository. https://sourceforge.net/projects/open-chord/. 27

[42] Overlay Weaver. http://overlayweaver.sourceforge.net. 58

[43] OverSim official website. www.oversim.org. 64

[44] P2PSim official website. https://pdos.csail.mit.edu/archive/p2psim/. 65

[45] PAST. A large-scale, peer-to-peer archival storage facility. http://www.freepastry.org/PAST/default.htm. 37

[46] Past. FreePastry's DHT. https://trac.freepastry.org/wiki/tut_past. 37

[47] Pastry. A substrate for peer-to-peer applications. http://www.freepastry.org. 34

[48] PeerfactSim.KOM website. http://peerfact.com/. 67

[49] PeerSim: A Peer-to-Peer Simulator. http://peersim.sourceforge.net/. 67

[50] PPStream Home Page. http://pps.tv. 8

[51] PPTV Home Page. http://www.pptv.com/. 8

[52] Pydht, Python implementation of the Kademlia DHT data store. https://github.com/isaaczafuta/pydht. 45

[53] Skype Home Page. http://www.skype.com. 7

[54] The User-mode Linux Kernel Home Page. http://user-mode-linux.sourceforge.net/. 5

[55] TomP2P official webpage. http://tomp2p.net/. 45

[56] TomP2P repository. https://github.com/tomp2p/TomP2P. 45

[57] Ubuntu releases list. https://wiki.ubuntu.com/Releases. 26

[58] Virtual Network User-Mode-Linux (VNUML). http://web.dit.upm.es/vnumlwiki/index.php/Main_Page. 6

[59] Vuze Home Page. http://www.vuze.com (accessed February 26, 2014). 17

[60] Chord Mail repository, 2003-2013. https://pdos.csail.mit.edu/pipermail/chord/. 26

[61] Qiong Liu Ang Cheng Tiong. Peer to Peer Cloud File Storage - Optimization of Chord and DHash, 2013. http://www.cse.scu.edu/ mwang2/projects/P2p_chord_13s.pdf. 26

[62] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. OverSim: A flexible overlay network simulation framework. In *Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007, Anchorage, AK, USA*, pages 79–84, 2007. 63

[63] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, M. Kaashoek, and S. Shenker. Making gnutella-like p2p systems scalable. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures and Protocols for Computer Communication (SIGCOMM '03)*, pages 407–418, New York, NY, USA, 2003. 63

[64] Cisco Systems, Inc. Cisco Visual Networking Index (VNI) Complete Forecast for 2015 to 2020, 2016. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual -networking-index-vni/complete-white-paper-c11-481360.html. 7

[65] Cisco Systems, Inc. The Zettabyte Era - Trends and Analysis, 2016. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual -networking-index-vni/vni-hyperconnectivity-wp.pdf. 8

[66] Bram Cohen. The BitTorrent Protocol Specification, january 2008. http://www.bittorrent.org/beps/bep_0003.html. 10, 17

[67] Distributed and Mobile Systems Group Otto-Friedrich Universität Bamberg. Open Chord version 1.0.4 User's Manual, 2007. https://opus4.kobv.de/opus4-bamberg/files/113/Dokument_01.pdf. 28

[68] Peter Druschel and Antony Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *In HotOS VIII*, pages 75–80, 2001. 36

[69] Savoir faire Linux Inc. OpenDHT. https://github.com/savoirfairelinux/opendht. 43

[70] ipoque. Internet Study 2008/2009. http://www.ipoque.com/sites/default/files/mediafiles/documents/internet-study-2008-2009.pdf. 11

[71] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. *SIGPLAN Not.*, **35**(11):190–201, 2000. 41

[72] Santosh Kulkarni. Badumna Network Suite: A decentralized network engine for Massively Multiplayer Online applications. In *Proceedings of the 9th IEEE International Conference on Peer-to-Peer Computing*, P2P'09, pages 178–183, Seattle, WA, USA, september 2009. 8

[73] Andrew Loewenstern and Arvid Norberg. BitTorrent DHT Protocol Specification, january 2008. http://bittorrent.org/beps/bep_0005.html. 10, 17

[74] lordelf007@gmail.com (Tri Nguyen Phi Minh). Chord mail from Tri Nguyen Phi Minh, 2012. https://pdos.csail.mit.edu/pipermail/chord/2012-September/001023.html. 26

[75] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Proceedings of the 1st International Workshop on Peer-to Peer Systems*, IPTPS'02, pages 53–65, march 2002. 10, 15

[76] Alberto Montresor and Márk Jelasity. PeerSim: A scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA, September 2009. 66

[77] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. *Theory of Computing Systems*, **32**(3):241–280, 1999. 10, 14

[78] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures and Protocols for Computer Communication (SIGCOMM)*, pages 161–172, San Diego, CA, USA, 2001. 10, 11, 22

[79] Sean C. Rhea, Patrick R. Eaton, Dennis Geels, Hakim Weatherspoon, Ben Y. Zhao, and John Kubiatowicz. Pond: The oceanstore prototype. In Jeff Chase, editor, *FAST*. USENIX, 2003. 41

[80] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329–350, 2001. 10, 14, 34, 36

[81] sit@mit.edu (Dr. Emil Sit). Chord mail from Dr. Emil Sit, 2012. https://pdos.csail.mit.edu/pipermail/chord/2012-September/001023.html. 26

[82] Moritz Steiner, Taoufik En-Najjary, and Ernst W. Biersack. A Global View of KAD. In *Proceedings of the 7th ACM SIGCOMM Internet Measurement Conference*, IMC'07, pages 117–122, New York, NY, USA, 2007. ACM. 11

[83] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishman. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures and Protocols for Computer Communication (SIGCOMM)*, pages 149–160, San Diego, CA, USA, 2001. 10, 12, 24

[84] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, pages 60:1–60:10, 2008. 63

[85] Fatos Xhafa, Raul Fernandez, Thanasis Daradoumis, Leonard Barolli, and Santi Caballé. Improvement of jxta protocols for supporting reliable distributed applications in p2p systems. In *Network-Based Information*

Systems, **4658** of *LNCS*, pages 345–354. Springer-Verlag Berlin, Heidelberg, 2007. 18

[86] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C.Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, **22**(1):41–53, 2004. 10, 14, 39