



Hierarchical peer-to-peer look-up service

Prototype implementation

(Master Thesis)

Francisco Javier Garcia Romero

Tutor in Institut Eurecom:
Prof. Dr. Ernst Biersack

March 28, 2003

Acknowledges

I first of all wish to thank Mr. Dr. Ernst Biersack, my master thesis supervisor, for supporting and advising me and for doing this work possible. I also want to thank Mr. Luis Garcés who has been my principal advisor during this master thesis and my personal friend during my stay at Institut Eurecom.

I specially thank Mr. Dr. Javier Aracil for his comprehension and for doing possible this project at Institut Eurecom. I also want to thank Mr. Dr. Pascal Felber for his help and very interesting contributions to this work.

Last but not least, I would like to thank Christian Schleippmann, my office colleague for all the joyful moments that we have passed together.

Abstract

This work describes the design and implementation of a prototype of a distributed storage system over a hierarchical organized peer-to-peer network. This system uses empty disk space on Internet hosts for reliable storage of data objects. To increase the reliability of the system, the objects are replicated and distributed to peer-nodes of a two-level hierarchical peer-to-peer system. The peers are organized into groups. The groups are organized in a top-level Chord overlay network that is spanned over the participating groups. The Chord top-level overlay network provides a robust and well scaling binding of objects to groups. For organizing the peers inside each group, a CARP lower-level overlay network is used. This distributed storage system provides high reliable content location in an environment of unreliable hosts when nodes and groups join or leave the system. It is robust against host failures and the binding is resolved by a two-step look-up operation. The downloading of contents has been improved by a parallel data access. The major part of the prototype implemented in Java is versatile object oriented framework architecture. A hierarchy of framework layers provides generalized solutions to problems in peer-to-peer networking. The file storage application itself is a thin layer on top of this framework.

Table of Contents

1. Introduction	4
1.1. A reliable storage network over a hierarchical peer-to-peer system	6
1.2. Organization of this work	7
2. Related Work.....	8
2.1. Usenet.....	8
2.2. DNS	8
2.3. Some recent peer-to-peer applications.	9
2.3.1. First generation P2P routing and location schemes	9
2.3.1.1. Napster	9
2.3.1.2. Gnutella	10
2.3.1.3. FastTrack.....	10
2.3.2. Second generation P2P systems	11
2.3.2.1. Pastry.....	11
2.3.2.2. Tapestry.....	12
2.3.2.3. CAN	12
2.3.2.5. CARP	12
2.3.3. Distributed Storage Applications	12
2.3.3.1. Freenet.....	13
2.3.3.2. PAST	13
2.3.3.3. Cooperative File System (CFS)	14
3. Problem analysis.....	15
3.1. Overlay Routing Networks	15
3.1.1. Chord.....	16
3.1.2. CARP	20
3.2. A hierarchical look-up service	20
3.3. Proposed solution	21
3.3.1. A two-step look-up service	22
3.3.2. Metadata structure	23
3.3.3. Intended use of this system	23
4. Overview of the framework layer	24
4.1 Object oriented programming languages concepts	24
4.1.1. Classes and objects.....	24
4.1.2. Inheritance.....	24
4.1.3. Polymorphism	25
4.1.4. Java.....	25
4.2. Framework layer design	26
4.2.1. Message layer.....	27
4.2.2. Look-up and Routing layer	27
4.2.3. Block Storage layer	28
4.2.4. Application layer.....	29

5. Message Layer Implementation.....	31
5.1. A pool of threads for managing messages reception	31
5.2. Node-to-node communication method.....	31
6. Look-up and Routing layer implementation	33
6.1. Node identifiers	33
6.2. Top-level overlay network	34
6.2.1. Successors and predecessors	34
6.2.2. Fingers	35
6.2.3. Inter-group look-up	36
6.2.4. Join and leave of groups.....	36
6.2.4.1. Stabilization process.....	37
6.2.4.2. Notification	38
6.2.4.3. Group join	38
6.3. Lower-level overlay network	39
6.3.1. Node group information	39
6.3.2. A node joins the group	40
6.3.3. A node leaves the group.....	41
6.4. Hierarchical look-up service	42
7. Block storage layer implementation	44
7.1. Basic Elements	44
7.1.1. Metadata Hash.....	44
7.1.2. Paraloader.....	45
7.2. Storing a block data.....	46
7.3. Fetching a block	47
7.4. Reorganization of data block due a overlay network changes	48
7.4.1. A node leaves the group	48
7.4.2. A node joins the group	49
8. Application layer implementation	50
8.1. Batch nodes file.....	50
8.2. Graphical user interface (GUI).....	51
8.3. Storing and fetching a file	52
8.4. Block event methods	53
9. Conclusion and Future Work	54
10. Bibliography	54

List of Figures

<i>Figure 1.1- A client/server architecture</i>	<i>5</i>
<i>Figure 1.2- A peer-to-peer architecture.....</i>	<i>6</i>
<i>Figure 3.1- A key is stored at its successor: node with next higher ID.....</i>	<i>17</i>
<i>Figure 3.2- The basic look-up</i>	<i>18</i>
<i>Figure 3.3- An example of a finger interval with the finger pointer</i>	<i>18</i>
<i>Figure 3.4- Chord routing look-up example.....</i>	<i>19</i>
<i>Figure 3.5- Implemented system overview</i>	<i>21</i>
<i>Figure 3.6- Two-step look-up service.....</i>	<i>22</i>
<i>Figure 4.1- Framework Layer design and classes hierarchy.....</i>	<i>26</i>
<i>Figure 4.2- Interaction of Look-up and routing layer with the other layers.....</i>	<i>28</i>
<i>Figure 4.3- Communication between the Storage layer and the others layers</i>	<i>29</i>
<i>Figure 4.4- Communication between the Application layer and the Storages layer.....</i>	<i>30</i>
<i>Figure 5.1- Basic communication method among nodes.....</i>	<i>32</i>
<i>Figure 6.1- An example of the top-level Chord overlay network</i>	<i>35</i>
<i>Figure 6.2- An inter-group look-up example.....</i>	<i>37</i>
<i>Figure 6.3- A CARP group example.....</i>	<i>40</i>
<i>Figure 7.1- Parallel download process</i>	<i>46</i>
<i>Figure 7.2- Storing a data block.....</i>	<i>47</i>
<i>Figure 7.3- Fetching a data block.....</i>	<i>48</i>
<i>Figure 8.1- A batch nodes file example</i>	<i>50</i>
<i>Figure 8.2- Graphical user interface (GUI) appearance.....</i>	<i>52</i>
<i>Figure 8.3- Files Splitting into Blocks.....</i>	<i>53</i>

1. Introduction

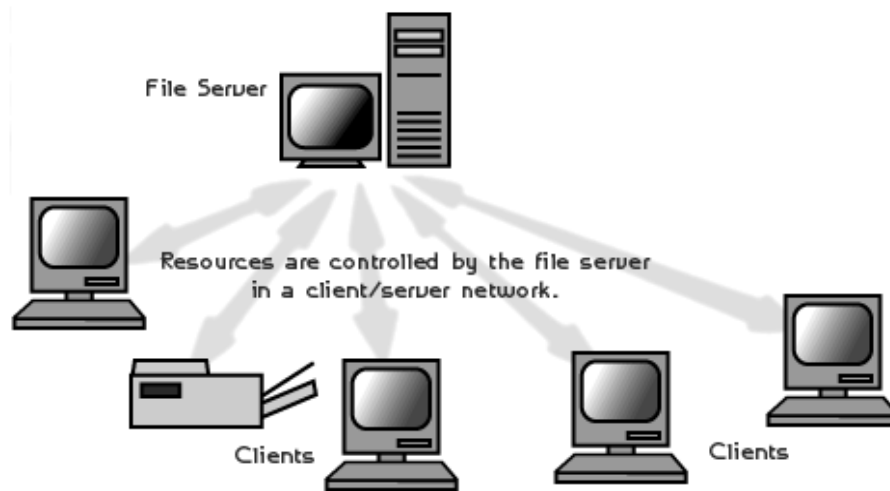
Peer-to-peer computing can be described as a class of applications that takes advantage of resources available at the edges of the Internet. Peer-to-peer (P2P) systems are distributed systems without any centralized control or hierarchical organization, where the software running at each node is equivalent in functionality. These systems have recently received significant attention in both, academic and industry environment for a number of reasons. The lack of a central server means that individuals can cooperate to form a P2P network without any investment in additional high-performance hardware to coordinate it. Furthermore, P2P networks suggest a way to aggregate and make use of the tremendous computation and storage resources that remain unused on idle individual computers. Finally, the decentralized, distributed nature of P2P systems makes them robust against certain kinds of faults, making them potentially well-suited for long-term storage or lengthy computations.

In the last years, new peer-to-peer applications have been introduced and they have become a lot of public attention. A new type of peer-to-peer based file sharing applications, like Napster[1] or GNutella[2], have re-invented the history of the music industry and they became a strong polemic a cause of legal issues concerning the content downloading. The peer-to-peer paradigm seems to be something really new. This is not truth. In fact, the early Internet was designed like this. We can take the Usenet, which appeared in 1979 and which is still used, like a clear example of this. The properties that characterize P2P applications are[3]:

- They are distributed systems, so they improve the scalability.
- They take advantage of distributed, shared resources such as storage, CPU cycles, and content on peer-nodes
- Peer-nodes have identical capabilities and responsibilities
- Symmetrical communication between peer-nodes
- Significant autonomy from central servers for fault tolerance
- Operate in dynamic environment where frequent join and leave is the norm

The reason why most people consider peer-to-peer applications as something radical new is that in the last decade, the paradigm of Internet applications changed from decentralized applications, like the Usenet, to the server centric World Wide Web. In fact, the Internet as originally conceived in the late 1960s was a peer-to-peer system. The goal of the original ARPANET was to share computing resources around the U.S. The first few hosts on the ARPANET, UCLA, SRI, UCSB, and the University of Utah, were already independent computing sites with equal status. The ARPANET connected them together not in a master/slave or client/server relationship, but rather as equal computing peers. For example, the early "killer apps" of the Internet, FTP and Telnet, were themselves client/server applications. A Telnet client logged into a compute server, and an FTP client sent and received files from a file server. But while a single application was client/server, the usage patterns as a whole were symmetric. Every host on the Net could FTP or Telnet to any other host, and in the early days of minicomputers and mainframes, the servers usually acted as clients as well. Between 1995 and 1999 the Internet became a mass medium driven by the "killer" applications World Wide Web (WWW). The World Wide Web is a typical client/server application. The change of the Internet to a mass cultural phenomenon caused by the WWW,

has had a far-reaching impact on the network architecture, an impact that directly affects our ability to create peer-to-peer applications in today's Internet, because the networks switched to the client/server paradigm. These changes are seen in the way we use the network and are the origin of the breakdown of cooperation on the Net and of the increasing deployment of firewalls on it. Because of the asymmetry in the WWW service, page requests are much smaller than the page reply, the dial-up technologies were developed considering that asymmetry. ADSL and V.90 modems have three to eight time higher downstream bandwidths than upstream bandwidth. This growth of asymmetric networks caused a lot of problems to peer-to-peer applications because the P2P applications have symmetrical bandwidth characteristics.



-Figure 1.1- A client/server architecture

The P2P paradigm has many advantages, among which we can enumerate:

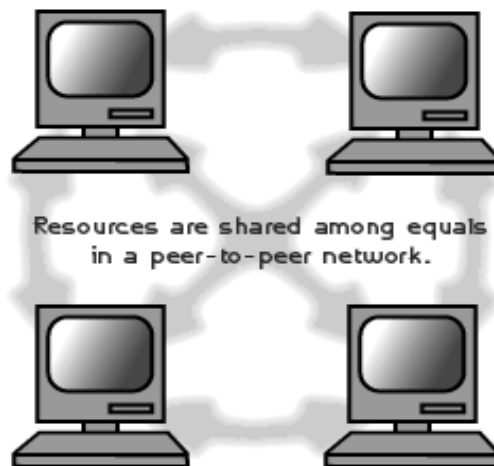
- Good harnesses of client resources.
- Provides robustness under failures.
- Redundancy and high fault-tolerance.
- Immunity to DoS attacks.
- Load balance among the clients.

But not everything is advantages. Among the disadvantages, we can enumerate:

- A tough design problem.
- How do you handle a dynamic network (nodes join and leave frequently)
- A number of constrains and uncontrolled variables:
 - No central servers
 - Clients are unreliable
 - Client vary widely in the resources they provide
 - Heterogeneous network (different platforms)

In resume, we can say that the Internet started out as a fully symmetric, peer-to-peer network of cooperating users. As the Net has grown to accommodate the millions of people flocking online, technologies have been put in place that have split the Net up into a system

with relatively few servers and many clients. At the same time, some of the basic expectations of cooperation are showing the risk of breaking down, threatening the structure of the Net. These phenomena pose challenges and obstacles to peer-to-peer applications: both the network and the applications have to be designed together to work in tandem. Application authors must design robust applications that can function in the complex Internet environment, and network designers must build in capabilities to handle new peer-to-peer applications. Fortunately, many of these issues are familiar from the experience of the early Internet; the lessons learned there can be brought forward to design tomorrow's systems.



-Figure 1.2- A peer-to-peer network

1.1. A reliable storage network over a hierarchical peer-to-peer system

The Java based prototype presented in this work uses empty disk space available on Internet hosts to build a reliable storage system. By April 2002, a typical workstation PC is shipped with a hard disk of about 60 GB storage capacities. After the operating system and some other applications are installed, most of the capacity on the hard disk is still unused. For example, the software takes 10 GB and the remaining free disk space is 50GB. For an organization with 100 such workstations, the total amount of unused disk space is 5 TB. Nowadays, most workstations in an organization are connected together with a local area network (LAN) that uses the Internet protocol (IP). This work describes the design of such a system and explains the implementation of a simple file storage application, which uses an underlying framework architecture developed as a mayor part of this work.

The goal is to achieve a maximum of reliability and fault tolerance for the storage service built out of Internet hosts, called *nodes* in this context. These nodes are organized in a hierarchical peer-to-peer system. Nodes are organized into groups. We use overlay networks for organizing both nodes into groups (in a lower-level overlay network) and groups (in a top-level overlay network). Each group has one or more superpeers for inter-group communication. These overlay networks use distribute hash functions to map keys identifiers of the stored contents to specific nodes.

The storage network must be reliable while the nodes themselves are not. Unlike dedicated file servers, the nodes are workstations generally not shipped with redundant power supplies or RAID (Redundant Array of Inexpensive Disks) systems. Since the workstations are under control of their users, their system availability is not predictable. Users may shut down their workstations or a network link may temporarily fail.

Assuming a heterogeneous and dynamic environment of hosts connected together by a high bandwidth and low latency IP network this work has focuses on:

- reliability of data storage.
- scalability in terms of the number of hosts and content requests.
- efficient usage of the available resources

Further, it is assumed that there are no restrictions concerning firewalls and Network Address Translation (NAT) issues. All hosts are willing to cooperate by relaying messages and they store data if their storage quotas have not been exceeded.

1.2. Organization of this work

This work is organized as follows: in the chapter "Related Work", existing applications with their solutions to P2P specific problems are analyzed. Immediately after, the chapter "Problem analysis" introduces the problem and presents the general solution proposed. Immediately after, the solution design is presented. The solution proposed a framework layer design, so the more important implementation aspects of every layer are presented in separated chapters. This work closes with the chapter "Conclusion and Future Work", where the results of this work are summarized and an outlook for future improvements are given. In this document important terms are emphasized with *italic* font, when first introduced. Class and method names are always emphasized with *italic*.

2. Related Work

The Internet as originally conceived in the late 1960s as a peer-to-peer system. The early Internet was also much more open and free than today's network. For example, firewalls were unknown until the late 1980s. Generally, any two machines on the Internet could send packets to each other. The Net was the playground of cooperative researchers who generally did not need protection from each other. The protocols and systems were obscure and specialized enough that security break-ins were rare and generally harmless. Let's look first at two long-established fixtures of computer networking that include important peer-to-peer components: Usenet and DNS. Immediately after, we are going to present some new peer-to-peer applications more recently appeared.

2.1. Usenet

Usenet news implements a decentralized model of control that in some ways is the grandfather of today's new peer-to-peer applications such as Napster, Gnutella and Freenet[4]. Fundamentally, Usenet is a system that, using no central control, copies files between computers. Since Usenet has been around since 1979, it offers a number of lessons and is worth considering for contemporary file-sharing applications.

The Usenet system was originally based on a facility called the Unix-to-Unix-copy protocol, or UUCP. UUCP was a mechanism by which one UNIX machine would automatically dial another, exchange files with it, and disconnect. This mechanism allowed UNIX sites to exchange email, files, system patches, or other messages. The Usenet used UUCP to exchange messages within a set of topics, so that students at the University of North Carolina and Duke University could each "post" messages to a topic, read messages from others on the same topic, and trade messages between the two schools. The Usenet grew from these original two hosts to hundreds of thousands of sites. As the network grew, so did the number and structure of the topics in which a message could be posted. Usenet today uses a TCP/IP-based protocol known as the Network News Transport Protocol (NNTP), which allows two machines on the Usenet network to discover new newsgroups efficiently and exchange new messages in each group.

Usenet has been enormously successful as a system in the sense that it has survived since 1979 and continues to be home to thriving communities of experts. Usenet has evolved some of the best examples of decentralized control structures on the Net. There is no central authority that controls the news system. The addition of new newsgroups to the main topic hierarchy is controlled by a rigorous democratic process. Still, Usenet's systems for decentralized control, its methods of avoiding a network flood, and other characteristics make it an excellent object lesson for designers of peer-to-peer systems.

2.2. DNS

The Domain Name System (DNS) is an example of a system that blends peer-to-peer networking with a hierarchical model of information ownership. DNS was established as a

solution to a file-sharing problem. In the early days of the Internet, the way to map a human-friendly name like *bbn* to an IP address like *4.2.49.2* was through a single flat file, *hosts.txt*, which was copied around the Internet periodically. As the Net grew to thousands of hosts and managing that file became impossible, DNS was developed as a way to distribute the data sharing across the peer-to-peer Internet.

The remarkable thing about DNS is how well it has scaled, from the few thousand hosts it was originally designed to support in 1983 to the hundreds of millions of hosts currently on the Internet. The lessons from DNS are directly applicable to contemporary peer-to-peer data sharing and distribute storage applications.

The namespace of DNS names is naturally hierarchical. This built-in hierarchy yields a simple, natural way to delegate responsibility for serving part of the DNS database. Each domain has an *authority*, the name server of record for hosts in that domain. When a host on the Internet wants to know the address of a given name, it queries its nearest name server to ask for the address. If that server does not know the name, it delegates the query to the authority for that namespace. That query, in turn, may be delegated to a higher authority, all the way up to the root name servers for the Internet as a whole. As the answer propagates back down to the requestor, the result is cached along the way to the name servers so the next fetch can be more efficient. Name servers operate both as clients and as servers.

2.3. Some recent peer-to-peer applications.

So from its earliest stages, the Internet was built out of peer-to-peer communication patterns. One advantage of this history is that we have experience to draw from in how to design new peer-to-peer systems. The problems faced today by new peer-to-peer applications systems such as file sharing are quite similar to the problems that Usenet and DNS addressed 10 or 15 years ago. This applications are presented because they deal with the kind of problems with are we going to deal in this work.

2.3.1. First generation P2P routing and location schemes

The first generation of new P2P applications, like Napster or Gnutella, were specially intended for large scale sharing of data files. Because of their big success, the peer-to-peer paradigm has become very popular. Such applications are very valid to solve the objectives for which they were designed. However, there were a set of issues for which they didn't give a nice solution, like for example self-organization of the peers that make up the system or the scalability of the P2P system with the number of peers. Moreover, a reliable content location was not guaranteed.

2.3.1.1. Napster

Napster is a file sharing application. Napster not follows a true P2P concept. Napster can be characterized as a client/server system for centralized content metadata look-up combined with direct client-to-client connections for content delivery. At startup the Napster client software connects to a central Napster server, authenticates itself with login and

password and registers its shared content's metadata to a central index database. A content query is sent to the central index server, which processes the query by an index database look-up, and returns to the client a list of matching content metadata records containing the network location of the client sharing the content item, its exact filename and some bandwidth and latency information. From this list the user has to choose a client from whom to download the content file. The download reliability is low, because only a single unreliable source is used and a broken download is not automatically continued from a different source. After connecting to one of the central servers, the client stays connected to its server for the whole session. Since each server maintains its own index database, a user will only see a restricted view of the total content available. The handicap of Napster is the centralized index, which simplifies the system but results in a single point of failure and a performance bottleneck.

2.3.1.2. Gnutella

Gnutella is another file sharing application. To avoid the disadvantages of Napster, the Gnutella network is decentralized. The only central component is the host cache service, which is used by the *servants*, a Gnutella specific term of combined client and server, to find a bootstrap node. The Gnutella protocol uses a time-to-live (TTL) scoped flooding for servant and content discovery. A servant is permanently connected to a small number of neighbors. When a servant receives a request from one of his neighbors, it decreases the TTL counter of the request and forwards it to all its neighbors if the TTL is greater than zero. The reply is routed back along the reverse path. There are two important request/reply pairs. A *Ping* request for discovering new servants is replied with a *Pong* response message containing the IP address, TCP port and some status information. The other pair is the *Query* request which contains query keywords and is answered with a *QueryHit* if the Query matches some files shared by the servant. The *QueryHit* is routed back along the reverse path to the servant that initiated the Query and contains the necessary information to start a direct download of the file, which is done similar to the HTTP get command.

The main disadvantage of Gnutella is the distributed search based on scoped flooding, which does not scale in terms of the number of servants[5], because the number of messages grows exponentially and uses much of the servant's bandwidth. To reduce the number of servants the next generation of the Gnutella protocol will introduce *supernodes* which will act as a message routing proxies for clients with limited bandwidth. These clients, called *shielded nodes*, have only a single connection to one supernode, which shields them from routing Gnutella messages. The supernode concept is a result of the nodes heterogeneity observed in the real world. Not all nodes are really equal concerning their resources and by far not all users want to share them.

2.3.1.3. FastTrack

The FastTrack protocol, used in the KaZaa[6] and Morpheus[7] application, is a hybrid and two layered architecture of peers connect to supernodes, which themselves are connected together. A supernode acts like a local search hub that maintains the index of the media files being shared by each peer connected to it and proxies search requests on behalf of its local

peers. FastTrack elects peers with sufficient bandwidth and processing power to become a supernode if its user has allowed it in the configuration. A search results in FastTrack contains a list of files that match the search criteria. FastTrack uses parallel download and client side caching for file transfers. A file is logically split into segments and these segments are downloaded from other peers that share the same file or, in the case of client side caching, do download this file and share the segments downloaded so far until the download is completed. This can increase the download speed significantly, especially for asymmetric dial-up connections, because the limited upstream bandwidths add up together. As FastTrack is a proprietary protocol, it is so far difficult to evaluate which scaling properties the supernode network has.

2.3.2. Second generation P2P systems

This second generation of P2P applications, like Pastry[8], Tapestry[9], CAN[10] or Chord[11], were intended to resolve the problems than the first generation of P2P applications doesn't give a solution. They form a self-organizing overlay network. In addition, they provide a load balanced, fault-tolerant distributed hash table, in which items can be inserted and looked up in a bounded number of forwarding hops, so they guarantee a definite answer to a query in a bounded number of network hops. They use hash identifiers for both, nodes and contents. In fact, all of these systems provide the same unique service: for a key k which identifies any content, any node A can determine the current live node B responsible for the key k . These systems could be used in a variety of distributed applications, including distributed stores, event notification, and content distribution. It is critical for overlay routing to be aware of the network topology. Otherwise, each routing hop takes a message to a node with a random location in the Internet, which results in high look-up delays and unnecessary wide-area network. While there are algorithmic similarities among each of these systems, an important distinction lies in the approach they take to topology-aware routing. We present a brief comparison of the different approaches that propose each system.

2.3.2.1. Pastry

In Pastry, *keys* and *idNodes* are 128 bits in length and can be thought of as a sequence of digits in base 16. A node's routing table has about $\log(16N)$ rows and 16 columns (N is the number of nodes in the overlay). The entries in row n of the routing table refer to nodes whose *idNodes* share the *rst n* digits with the present node's *idNode*. The $(n + 1)$ th *idNodes* digit of a node in column m of row n equals m . The column in row n corresponding to the value of the $(n+1)$ th digit of the local node's *idNodes* remains empty. At each routing step in Pastry, a node normally forwards the message to a node whose *idNodes* shares with the key a *prex* that is at least one digit longer than the *prex* that the key shares with the present node's id. If no such node is known, the message is forwarded to a node whose *idNode* shares a *prex* with the key as long as the current node but is numerically closer to the key than the present node's id. Each Pastry node maintains a set of neighboring nodes in the *idNode* space (leaf set), both to locate the destination in the internal routing hop, and to store replicas of data items for fault tolerance. The expected number of routing hops is less than $\log(16N)$.

2.3.2.2. Tapestry

Tapestry is very similar to Pastry but differs in its approach to mapping keys to nodes in the sparsely populated id space, and in how it manages replication. In Tapestry, there is no leaf set and neighboring nodes in the namespace are not aware of each other. When a node's routing table does not have an entry for a node that matches a key's n th digit, the message is forwarded to the node in the routing table with the next higher value in the n th digit *modulo*($2b$). This procedure, called *surrogate routing*, maps keys to a unique live node if the node routing tables are consistent. For fault tolerance, Tapestry inserts replicas of data items using different keys. The expected number of routing hops is $\log(16N)$.

2.3.2.3. CAN

CAN routes messages in a d -dimensional space, where each node maintains a routing table with $O(d)$ entries and any node can be reached in $O(dN \ 1=d)$ routing hops. The entries in a node's routing table refer to its neighbors in the d -dimensional space. Unlike Pastry, Tapestry and Chord, CAN's routing table does not grow with the network size but the number of routing hops grows faster than $\log N$ in this case, namely $O(dN \ 1=d)$.

2.3.2.4. Chord

Chord uses a circular 160 bit id space. Unlike Pastry, Chord forwards messages only clockwise in the circular id space. Instead of the prex-based routing table in Pastry, Chord nodes maintain an n *finger table*, consisting of *idNodes* and IP addresses of up to 160 other live nodes. The i th entry in the *finger table* of the node with *idNode* n refers to the live node with the smallest *idNode* clockwise from $n^{2^{i+1}}$. Each node also maintains pointers to its predecessor and to its k successors in the id space (the *successor list*). Similar to Pastry's leaf set, this *successor list* is used to replicate objects for fault tolerance. The expected number of routing hops in Chord is $\log(N)$, where N is the number of nodes of the system.

2.3.2.5. CARP

Another example of overlay networks is CARP[12]. When using CARP, each peer has a list with the identities pi of all other peers in the network. In order to find a key k , a peer performs a hash function hi , $h(pi; k)$ over all peers pi . The peer pi with the highest hi is the one that stores key k . Thus, a peer in a CARP-organized network is able to find the peer where a key is stored with exactly one hop. However, each peer must know the identities and IP addresses of all the others.

2.3.3. Distributed Storage Applications

Because in this work we are going to implement the prototype of such an application, we are going to present some applications existing nowadays. Distributed storage applications must have an active replication strategy to increase reliability as compared to file sharing and content distribution application, which more rely the fact that with a large number of users sharing content, the probability of content being available can be quite high, however without

determination. An important difference to file sharing applications is that distributed storage applications in general have a *publish* process, which adds content items to the system. The location of the content items is not predefined like it is for file sharing applications.

2.3.3.1. Freenet

Freenet is a distributed publishing system, which provides anonymity to publishers and consumers of content. An adaptive network is used to locate content by forwarding requests to nodes that are closer to the key that identifies a content item. On each hop information whether the item was found on this path or not travels in backward direction and is temporarily stored on the nodes. The next request for the same key takes advantage of this information and gets routed directly to the content source. When the query reaches the content source, the content is propagated along the query's reverse path and cached in the intermediate nodes. Freenet uses an intelligent flooding search, where the routing information and cached copies are stored along the path. The more requests for a content item, the more cached copies and routing information are available. If there has been no request in a period of time for a content item, the nodes discard the content items, because all routing information about this item on the other nodes has already timed out and the item is not referenced anymore. As a consequence, published content is stored persistent only as long as there is enough demand for keeping routing information alive. The content objects are floating around in the network and there is only temporal and local knowledge about where the content is actually located. To provide anonymity to the publishers and consumers, there is no direct peer-to-peer data transfer. Instead, the content data is routed through the network. Nodes with low bandwidth may become a bottleneck to the system and the flooding based content look-up causes scales badly.

2.3.3.2. PAST

PAST [13] is a persistent peer-to-peer storage utility, which replicates complete files on multiple nodes. Pastry is used for message routing and content location. PAST stores a content item on the node whose node identifier *idNode* is closest to the file identifier *idFile*. Routing a message to the closest node is done by choosing the next hop node whose *idNode* shares with the *idFile* a prefix that is at least one digit longer than the prefix that the *idFile* shares with the present node's *idNode*. The *idFile* is generated by hashing the filename and the *idNode* is assigned randomly when a node joins the network. The routing path length scales logarithmic in terms of the overall number of nodes in the network. For each file an individual replication factor k can be chosen and replicas are stored on the k nodes that are closest to the *idFile*. Maintaining the k replicas in the case of a node failure is detected by the Pastry background process of exchanging messages with neighbors. When a node detects a neighbor node's failure, the replica is automatically replaced on another neighbor. Free storage space is used to cache files along the routing path, while approaching the closest node during the publish or retrieval process. This can only be done if the file data is routed along the reverse query path. Thus there is no direct peer-to-peer file transfer. Similar to the Freenet system, nodes with low bandwidth may become a bottleneck.

2.3.3.3. Cooperative File System (CFS)

CFS[14] is a read only file system built on top of Chord, which is used for content location. Chord belongs to the same family of second generation peer-to-peer resource location services like Pastry and Tapestry, which use routing for content location. On each hop, the closest routing alternative is chosen to approach the closest node defined by a metric of the identifiers generated by a hash function. Since Chord was used for the inter-groups look-up in this work, it is described in detail in Section 3.1.1. Right now, it's enough to know that the hashed identifiers are interpreted as *n-bit* numbers, which are arranged in a circle by the natural integer order. A file is split into blocks identified by *idBlocks*. Similar to PAST and Silverback, cache replicas are stored along the reverse look-up path when the requested block data is returned to reduce latency and to balance the load.

3. Problem analysis

Peer-to-Peer applications aim to take advantage of shared resources in a dynamic environment where fluctuating participants are the norm. In this work, the resource of interest is free disk space available on Internet hosts. Each host is identified by a unique Internet Protocols (IP) host address used for packet routing to this host. To establish a communication to a host, an additional port number is necessary to identify the software that handles the communication process on that host. The IP address together with the port number is the *network location* which is necessary to communicate with the software on a host managing its storage resources. Moreover, a content item needs a *content name* that identifies it among all the other content items.

In this work, we proposed a storage file system application. In this kind of application, the content is already stored on hosts and there must be mechanisms to discover the content stored on the hosts. The system itself decides where the content items are stored during the publish process and therefore, an addressing scheme based on an *overlay network* can be used, which resolves bindings by routing. The addressing scheme maps content items to nodes and this mapping is used to store and retrieve the content items.

3.1. Overlay Routing Networks

Every application in a P2P environment must handle an important problem: the *look-up* problem. For example, for the special case of a P2P storage file system, this problem consists in how do we find the data or how do we store the data. There are a lot of approximations to resolve this problem. For example, a centralized look-up scheme can be used as Napster does or a flooded query look-up scheme can be used as Gnutella does. In the case where the peer-to-peer system has n nodes, the centralized look-up scheme uses a table of a size $O(n)$ and it achieves the look-up in $O(1)$ hops. This method has scalability problems in number of nodes (because of the centralized approximation) and has also a great dependency of a central point. The flooded queries method uses a table size of $O(1)$ and achieves the look-up in $O(n)$ hops. This method has also problems of scalability when the number of nodes goes up (considerably increase of number of messages to be sent). Moreover, these two methods are not absolutely reliable in terms of existing data discover.

In order to improve the scalability of the system with the number of nodes, overlay networks were introduced. The look-ups are done by routing messages through the overlay network. To achieve a look-up cost of $O(\log(\#nodes))$, we should do a binary search, so we need to have an ordered array. With the objective of ordering both, the nodes and the data items, *hash functions* are used. Using a hash function like SHA-1, each object (nodes, data items) are identified by a unique *id* which is a fixed length bit string. In this way, all the objects are ordered and look-ups can be done in $O(\log(\#nodes))$ hops.

An overlay routing network is built of nodes connected together by a network of distinct topology. This network is a logical overlay network because the nodes communicate over an underlying communication network. But the logical network topology determines how the routing is done. To resolve a binding for a content name, a message is routed to a node which is "closest" to the content name according to a metric of the node identifiers and the content names. The network location of this "closest" node is the result of the look-up operation. The routing algorithm on each node exploits local routing knowledge to route the message to the "closest" local routing alternative.

To define the closeness, there must be a metric that applies to both, the node identifier and content name space. A hash function is used to map node identifiers and content names into a hash space and a metric is chosen to define the closeness. Pastry, Tapestry and Chord are based on this idea and therefore, share the same average look-up path length of $O(\log(\#nodes))$ hops, but they use different metrics and overlay network topologies. All of them use flat overlay networks. In this work we have interest in Chord, which we introduce it next.

3.1.1. Chord

Chord is a distributed look-up and routing overlay network using consistent hashing[15], originally introduced for distributed caching. Chord resolves the look-up problem of the peer-to-peer systems providing just one operation:

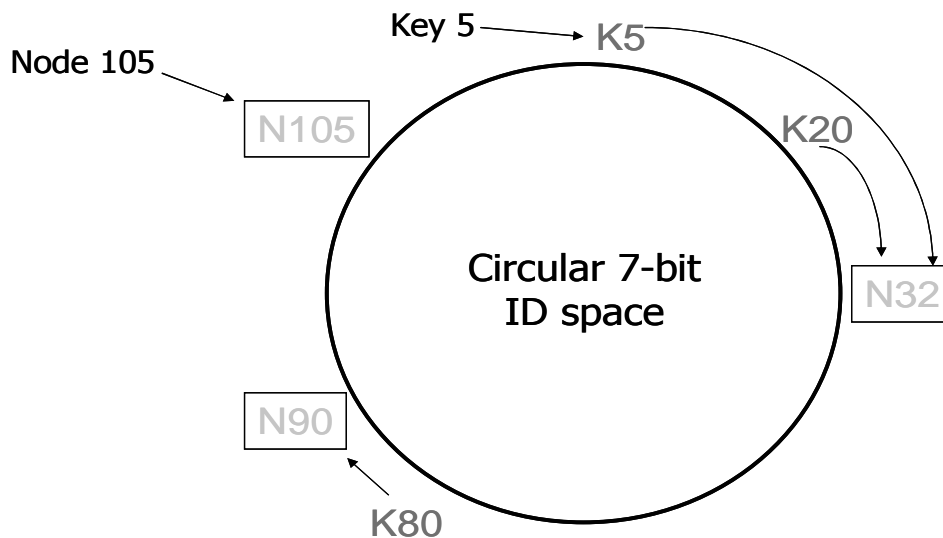
- Look-up(key) \rightarrow IP address.

where *key* is the identifier of a content item and the IP address is the one of the peer responsible to store it. Chord doesn't stores the data items. Chord *ids* are calculated with a hash function:

- Key identifier = SHA-1(key).
- Node identifier = SHA-1(IP address).

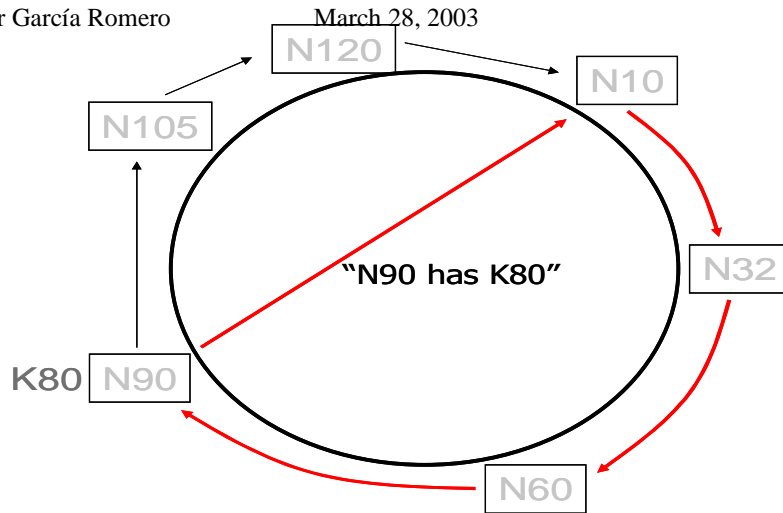
Both identifiers are uniformly distributed and both exist in the same *id* space. Nodes are organized in a circular topology by using *m-bit* node *ids* interpreted as nonnegative integer numbers wrapped around at zero. The total ordering of integer numbers assigns each node a *predecessor* and *successor* node. A node *id* is generated by applying a hash function to the node's host IP address. Therefore, the overlay network becomes a deterministic function of the host address. In other words, the host IP address determines the position in the circle. This makes the overlay network topology completely unaware of the underlying network layer topology, which has some positive and negative effects. Routing a message to the successor neighbor in the overlay network could result in routing to the other side of the world in the underlying IP network, causing high latency. On the other hand, IP network failures in a region do not map to a region of the logical overlay network region, often used for placing redundant replicas, like in CFS or PAST.

A content item is also identified by an m -bit content key, and the binding from keys (hashed content name) to node ids (hashed host address) is defined by the *successor* function. A key k is located at the node n with the same or the next higher id than the key k written as $n=successor(k)$ (see figure 3.1). The content item associated with the key k is not stored in Chord itself. Chord just assigns a responsible node whose network location is used to access the content item.



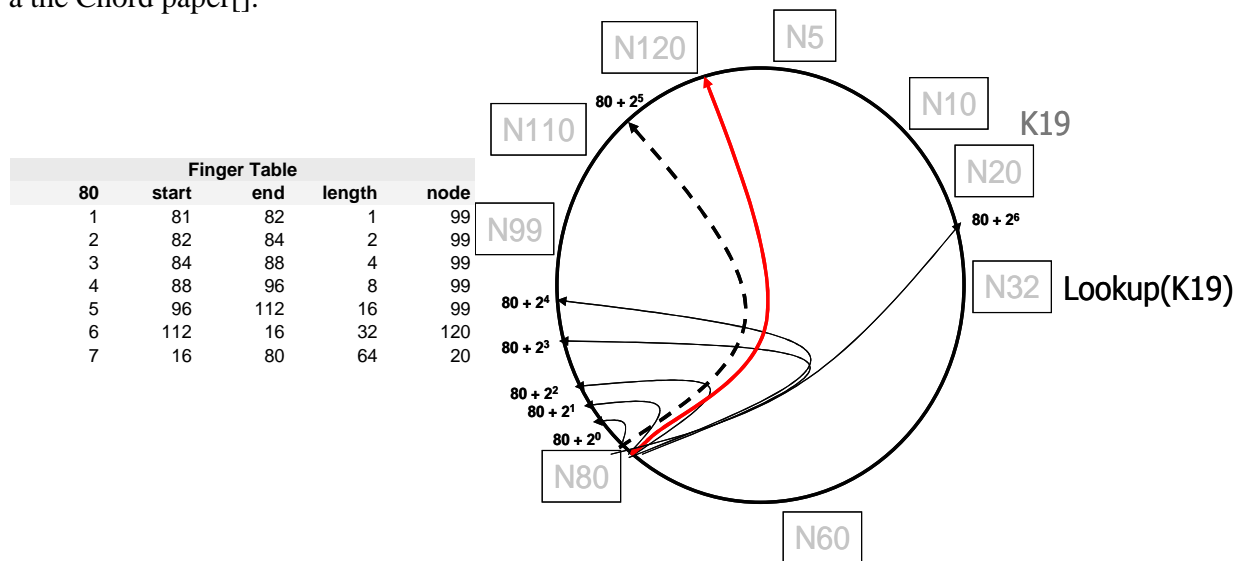
-Figure 3.1- A key is stored at its successor: node with next higher ID

Using the same identifiers for nodes and keys leads to a combined look-up and routing. A look-up is resolved by routing a message to the node that is the *successor* of the key. Every node knows at least two other nodes, its successor and its predecessor. The “*simple look-up*” algorithm routes a messages around the circle by following the successor pointers until a node with the same id or the next higher id than the key is found (see figure 3.2). As one node is only aware of its successor and its predecessor as available routing alternatives, a look-up message is always traversing the circle in direction of the successor pointers, because only this reduces the distance. In the worst case, a message has to complete a full circle turn, before the node that is successor to the key is found. Resolving a look-up with the simple algorithm takes $O(\#nodes)$ hops. As long as every node has a working pointer to the immediate successor in the circle, a successful successor look-up is guaranteed. In a real application with frequent node join and leave, a single successor pointer is not sufficient enough to guarantee a successful look-up. A single node failure would break the circle and result in look-up failures. Therefore, redundant successor pointers are used. As long as one working successor pointer is found, the look-up routing can proceed and a successful look-up is guaranteed.



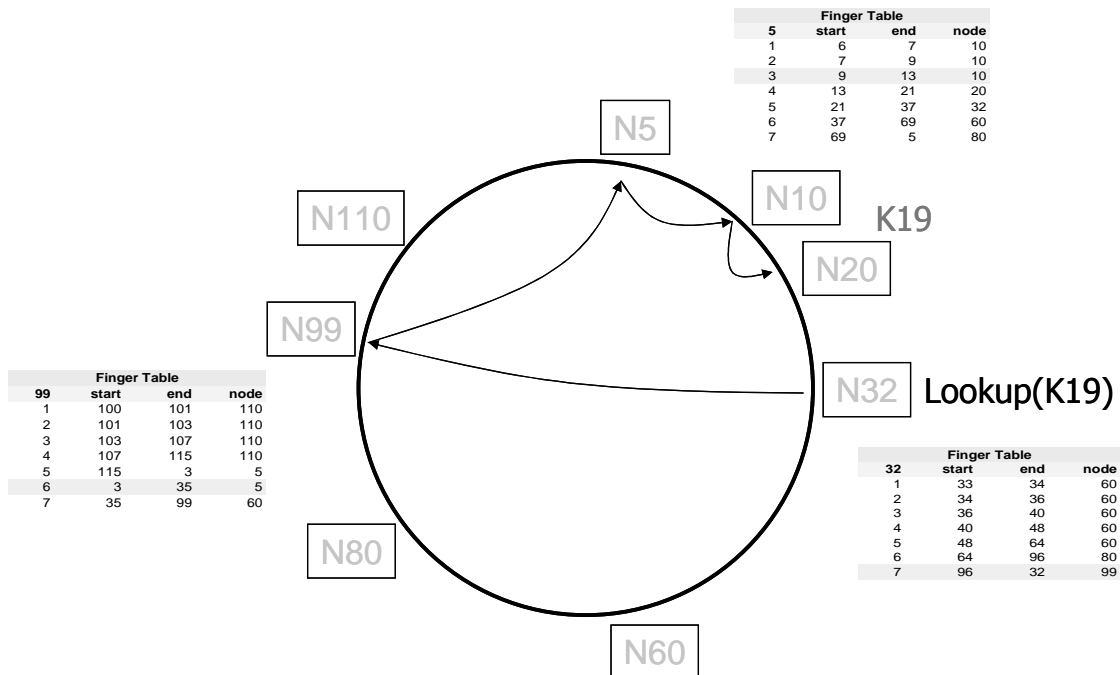
-Figure 3.2- The basic look-up

To reduce the average look-up path length to a practical number, a *finger table* with additional routing information is introduced (see figure 3.3). *Fingers* are like shortcuts, used instead of going around the circle from node to node following the successor pointers. Every node divides the circle into m finger intervals with exponentially growing size in power of 2. A finger points to the successor of the interval start, which could result in finger pointers being outside their corresponding finger interval. The finger nodes are resolved by the Chord look-up function, which returns the successor node of the interval start id . Using the finger table, adds $O(m)$ additional routing alternatives and the one is chosen that leads closest to the successor of the key. The higher the finger index, the farther away the finger points. Therefore, the finger table is searched in reverse order, starting at the $finger[m]$. If a finger i points to a node preceding the key, this hop reduces the distance to the key by 2^{i-1} . With a few hops, the distance to the key is quickly reduced, which results in an average look-up path length of $O(\log(\#nodes))$. This bound was proven theoretically and verified by controlled experiments in the Chord paper[.]



-Figure 3.3- An example of a finger interval with the finger pointer

Figure 3.4 shows a detailed Chord routing example using finger tables in a $m=7$ bit circular hash space. Starting at node $N32$, which wants to resolve the successor of the key $K19$, $N32$ looks in its finger routing table for the node that closest precedes $K19$. The finger table is searched in reverse order, starting at the finger with higher index. This finger matches the criteria and therefore, the look-up continues at $N99$. On $N99$, the finger table is searched again. The 7th finger $N60$ does not precede $K19$, therefore the 6th finger is tested. This one, pointing to $N5$, precedes $K19$, hence the look-up continues on $N5$. $N5$ finds $N10$ as its closest preceding finger. $N10$ now terminates the look-up, because it can make out that its successor $N20$ is the successor node of $K19$.



-Figure 3.4- Chord routing look-up example

Chord provides a peer-to-peer hash look-up service. Chord offers a scalable, robust, and balanced mapping of hashed content names to host network locations (IP address and a node D) with a minimal join/leave disruption. The Chord overlay network delegates responsibility for content items to nodes. Chord does not store the data itself. The participating hosts are organized in an overlay network that establishes neighbor relations among participating hosts based on numerical proximity of their *ids* in a virtual coordinate space. The look-up time of Chord in terms of the number of peer hops is $O(\log N)$, where N is the number of peers in the overlay network.

In spite of good characteristics that Chord presents, Chord has also some problems. The main problem in Chord is that the overlay network doesn't care about the underlying Internet. The look-up time can be high because two *neighbor* peers in the overlay network can be physically very far in terms of communication latency. Another thing that Chord doesn't consider is the fact that there are hosts which have much better characteristics than others (transmission bandwidth, CPU power or up-time). It seems a good idea to take advantage of these node with good characteristics in order to give them more responsibilities so that the

reliability of the peer-to-peer system is improved. It's also demonstrated that in a Chord system, look-ups can be achieved with a cost of $O(\log N)$ hops only when the overlay network is stable, when there aren't a lot of node joins and leaves. These problems can be solved introducing a hierarchy of nodes in the overlay network as is explained in [16].

3.1.2. CARP

When using CARP, each peer has a list with the identities P_i of all other peers in the group. In order to find a key k , a peer performs a hash function h , $h_i = h(P_i; k)$ over all peers P_i of the group. The peer P_i with the highest h_i is the one that stores the key k . Thus, every peer in a CARP-organized network is able to find the peer where a key is stored with exactly one hop. However, each peer must know the identities and IP addresses of all the others peers in the group. This does not scale well with the number of peers into a group.

3.2. A hierarchical look-up service

To solve the sort of problems than flats overlays show, the [16] paper propose to organize the participating peers into groups. The peers of each group form a *separate* overlay network (*lower-level overlay*). Another overlay network will be established among all *groups* (*top-level overlay*). The top-level overlay "interconnects" the lower-level overlays to ensure that there exists a routing path among peers in different groups. With this scheme:

- It's possible to implement a two-step hierarchical look-up P2P system. A look-up process requires finding first the group which a key is mapped to, and immediately after the peer in that group where the key is actually stored.
- We can improve the overall reliability and performance of the P2P system by choosing peers with the highest reliability and processing power inside each group to be part of the top-level overlay network. Those kind of peers are the super peers.
- We can put peers into the same group that are close in terms of communication latency across the Internet, thereby achieving shorter inter-peer delays and faster reaction to failures.
- We improve the scalability of existing solutions to overlay networks as each overlay only comprises a subset of all peers.

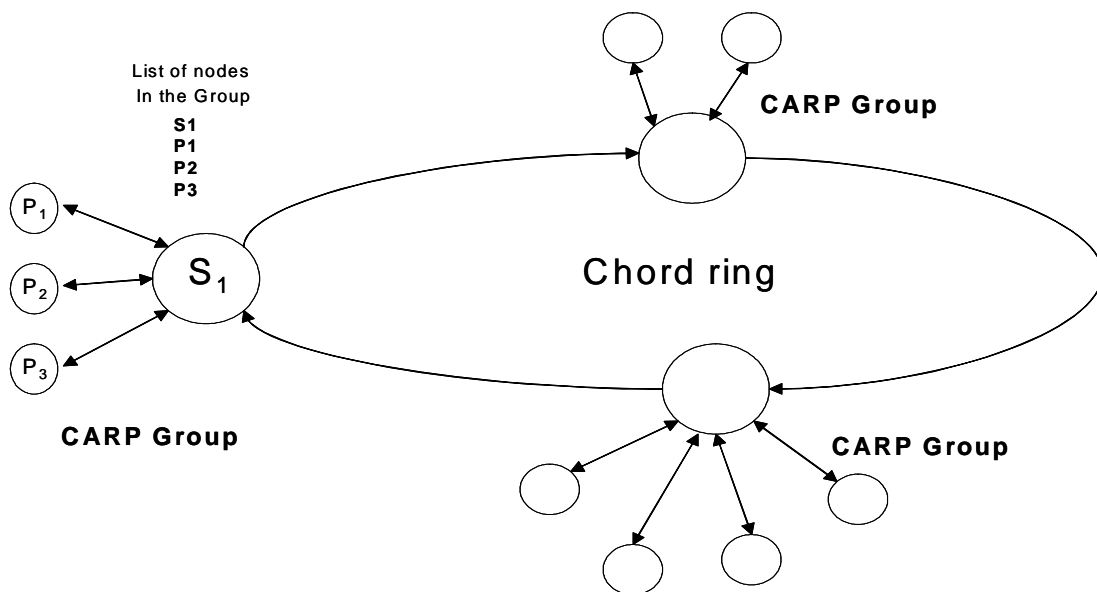
In this work the peers are organized into groups in a two level hierarchy. The groups are chosen so that the peers in the same group are "close" to each other in the sense that communication delays (latency and/or file transfer delays) between peers in the same group are relatively small. Each group is required to have one or more *super peers*. The super peers are gateways between the groups: they are used for inter-group query propagation. To this end, it's required that if S_i is a super peer in a group, then it musts know the identifier and the IP address of at least one super peer S_j of another group. With this knowledge, S_i can send query messages to S_j . On the other hand, if p is a "normal" peer in the group, then p must first send intra-group query messages to a super peer in its group, which can then forward the query message to another group. Within each group there is also an overlay network that is used for

query communication among the peers in the group. When a group is designed to be responsible of storing a key by the top-level overlay network, the key is stored within the group according to the organization of the group overlay network (lower-level overlay). Each of the groups operates autonomously from the other groups.

3.3. Proposed solution

In this work we are going to implement the prototype of a two-level hierarchical peer-to-peer look-up system. Peers are organized into groups. The groups are organized in an overlay network. For this top-level overlay network, we have chosen Chord. In fact, the system is a set of peer groups organized in a Chord ring (top-level overlay network). Inside groups, the peers are also organized in an overlay network. This system assumes that the number of peers into each group is small, in the order of hundred. CARP is used to organize the peers in a lower-level overlay network into the groups. Figure 3.5 shows an overview of the system.

At each group, there are a set of special peers, called super peers, which are the only peers that are able to execute inter-group look-ups. The rest of the peers of the group, called “normal” peers, are able to perform look-ups in other groups through any super peer of the group. Every peer can become a super peer at every moment. The super peers are intended to be the peers of the group with better characteristics and are chosen according a “merit” metric. Only the super peers have the routing information of the Chord ring: list of network locations of other super peers in successor groups, the super peers in the predecessor group, table of fingers. In the standard Chord system, at any “hop” of the Chord overlay network there is just a single node. In this scenario, at any “hop” of the Chord overlay network there is the set of super peers of a group.



-Figure 3.5- Implemented system overview

3.3.1. A two-step look-up service

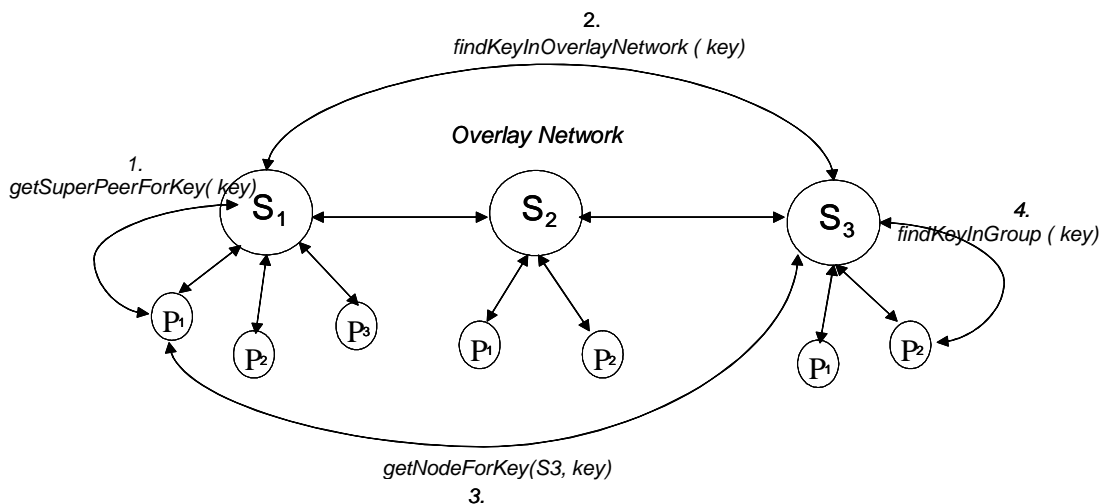
Our system provides a look-up service to find the peer that is responsible for a key. The key is under the responsibility of a peer, and CARP is used for assigning the responsibilities into the groups. The system first finds the group that is responsible for the key. This responsibility is assigned by the Chord top-level overlay network: the group responsible of storing a key k is the $idGroup = successor(k)$. Immediately after, the responsible peer within the responsible group is looked up. This responsibility is assigned by the CARP lower-level overlay network. Every peer at every group is able to perform a look-up in the system. Only the super nodes are able to perform an inter-level look-up. An intra-group look-up can be achieved by every node. We present the functions implemented that allow a node pi , integrated in the group Gi , to locate the peer pj , integrated in the group Gj , which is responsible for key k . The node pi has an IP address $@pi$, and the node pj has an IP address $@pj$. We assume that pi knows at least a super peer Si of this group and the IP address $@si$ of this super peer.

$@sj = getSuperpeerForKey(@si, k)$: The node pi calls this function to obtain from Si the IP address $@sj$ of the super node Sj managing the group responsible for key k .

$@sj = findKeyInOverlayNetwork(k)$: To answer the call $getSuperpeerForKey$, the super peer Si uses $findKeyInOverlayNetwork$ to obtain the IP address $@sj$ of the super node Sj of the group responsible for key k . This function implements the inter-group Chord look-up and only can be called by a super peer.

$@pj = getNodeForKey(@sj, k)$: A “normal” node pi calls this function to obtain from the super peer Sj of the group Gj responsible for key k , the IP address $@pj$ of the node pj responsible for k .

$@pj = findKeyInGroup(k)$: To answer the call $getNodeForKey$, a super peer Sj uses this function to obtain the IP address $@pj$ of the node pj responsible for k . This function implements the CARP look-up and can be called by any the node.



-Figure 3.6- Two-step look-up service

3.3.2. Metadata structure

In order to increase the reliability of the storing system, several replicas of every data blocks are stored in different nodes into the responsible group for the data unit and in successors groups of the responsible group. A metadata structure for a content item tracks pointers to nodes, where replicas for reliability are located. Every node which stores a data unit stores also this metadata structure. This metadata structure is then used for parallel download[18]. When a node requests a content item, it first looks-up a responsible node in each of the responsible groups for storing a data unit. It sends a message to a responsible node in each group to get the metadata pointers necessary for the parallel download. This parallel download scheme, accessing multiple replicas in parallel, will be used to increase download performance: instead of trying to select a fast node, connections to multiple nodes providing a replica of the desired content item are established and different blocks of the replica are transferred in parallel. In this case, the fastest node will be among the set of opened parallel connections. The aggregation of the individual connections bandwidths will potentially increase overall throughput, especially for asymmetric connections, because the limited upstream channels are aggregated to match with the higher bandwidth downstream channel of the receiving node. As an additional improvement, the intelligent paraloader performs a dynamic load balancing by choosing to download more block of a replica across the faster connections or drop connections that have become heavily congested during a download session. Due to paraloading, the download is more resilient to route or link failures and traffic fluctuations. More details about parallel access are found in the Section 7.1.2.

3.3.3. Intended use of this system

This system assumes that groups are quite stables. There are not frequent group joins and leaves. Chord is quite efficient in such high-availability environments. We also assume that the number of peers into each group is small, around a hundred. This is because in CARP, every peer knows the IP address and node identifier of all the others peers in the group, so this amount of information increases with the number of peers. Such a design has several advantages comparing to flat architecture designs:

- *Faster look-up time:* Because the number of groups will be typically orders of magnitude smaller than the total number of peers, queries travel over fewer hops. A cause that the number of groups is smaller than the number of peers, the number of hops in the Chord look-up algorithm is reduced. Into the groups, the look-up is done in only a hop. The groups could be formed by peers with small network latency communication.
- The top-overlay network is formed by the super-peers, which are the nodes with better reliability properties, so the reliability of the Chord ring is improved

4. Overview of the framework layer

A multi-layer design framework has been used to implement this system. Each layer provides some functionalities which can be used by the other layers. Obviously, the upper layers use the functionalities or services offered by the lower layers. The lower layers don't know anything about the upper layers. The communication between layers is achieved by two different means. The lower layers pass events to the upper layers when an one occurs. The upper layers use the methods or functions that the lower layers provide.

Object orientated (OO) programming languages and their concepts turned out to be very useful to design such a multi-layer framework. These concepts are used in this Java prototype, since Java is the model of an object orientated programming language. Before introducing the solution proposed to implement this prototype, it is desirable to introduce quickly the theoretical concepts of the OO programming languages.

4.1 Object oriented programming languages concepts

OO is organized around classes and objects, which are the instances of the classes. Thinking of analysis and design objects is very important because humans think in terms of objects[17].

4.1.1. Classes and objects

We can define a *class* like a blueprint, or prototype, which defines the variables and the methods common to all *objects* of a certain kind. An *object* is just a real instance of a specific class. The *class* is defined by the class's code in a programming language, while an *object* is the code's representation in the computer's physical memory. Different *objects* interact and communicate with each other by sending *messages* to each other. In OO programming a *message* is sent from one object to another by calling the others object's methods. The information is passed along with the message as *parameters*. *Messages* provide two important benefits:

- An object's behavior is expressed through its methods, so (aside from direct variable access) message passing supports all possible interactions between objects.
- Objects don't need to be in the same process or even on the same machine to send and receive messages back and forth to each other.

4.1.2. Inheritance

The concept of a class makes it possible to define *subclasses* that share some or all of the main class characteristics. This is called *inheritance*. A class inherits state and behavior from its *superclass*. *Inheritance* provides a powerful and natural mechanism for organizing and structuring software programs. However, *subclasses* are not limited to the state and behaviors provided to them by their *superclass*. *Subclasses* can add variables and methods to

the ones they inherit from the *superclass*. You are not limited to just one layer of *inheritance*. The *inheritance* tree, or class *hierarchy* can be as deep as needed. In general, the farther down in the *hierarchy* a class appears, the more specialized its behavior. *Inheritance* offers the following benefits:

- *Subclasses* provide specialized behaviors from the basis of common elements provided by the *superclass*. Through the use of *inheritance*, programmers can reuse the code in the *superclass* many times.
- Programmers can implement *superclasses* called *abstract classes* that define "generic" behaviors. The *abstract superclass* defines and may partially implement the behavior, but much of the class is undefined and unimplemented. Other programmers fill in the details with specialized *subclasses*.

4.1.3. Polymorphism

The polymorphism concept results of the combination of inheritance and object communication via messages. Any descendant class is allowed to overwrite any method inherited from its parent class with its own implementation. When another object calls such an inherited and overwritten method, the specific implementation is called. This allows defining a standard way to communicate with classes based on the same ancestor class without knowing about their specific internal details.

4.1.4. Java

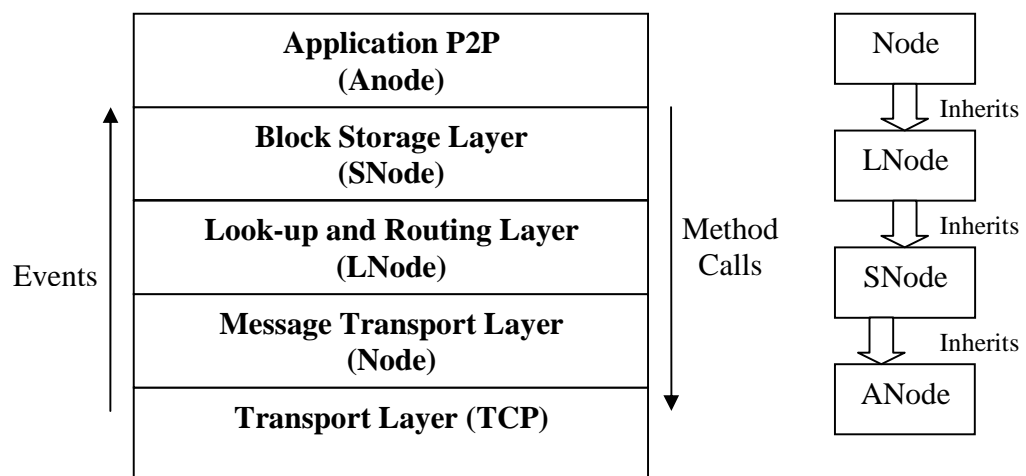
Java is a platform independent OO programming language with a C-like syntax and strict object orientation. Compared with C++ it lacks some advanced features like template classes and multiple class inheritance, but it is shipped with a fairly complete set of libraries covering most of the standard programming problems and data structures. The platform independence is achieved by using intermediate byte-code which is interpreted by a platform dependent Java Virtual Machines (JVM). For most systems a JVM implementation is available. Therefore Java is the optimal programming language for a distributed network operated in a heterogenic environment with multiple platforms. Developing and especially maintaining code versions for different platforms is hard. Either the code is separated from the beginning and therefore too many versions have to be synchronized, or compiling directives taking into account the individual platform's behaviors will result in fragmented, hardly understandable code. Java's "write once, run everywhere" paradigm helps out of this trouble. In real life, this is not always completely true, but in general the platform differences are minor. For applications with a graphical user interface (GUI), there is so far no real alternative to Java with its Swing library, which is available on all Java platforms. Swing offers a high level of abstraction and offers most standard widgets, elementary GUI items like lists, button and edit fields. Java offers a quite complete solution well adapted to distributed computing with many advantages to fast prototyping, but performance is still far away from native implementations and should be considered in the design concept.

4.2. Framework layer design

The solutions components are integrated into an object oriented framework architecture using Java's object oriented concepts. A *framework* in software design is a reusable class hierarchy that depends only on the abstract interfaces of various components[]. Another useful concept in software and network design is *layering*. Layers are defined by their services and interfaces to these services. Each layer uses services of the underlying layers to provide its own service and therefore hides underlying details. This super imposition leads to the desired abstraction necessary for a reusable framework design. Abstraction reduces the dependence from implementation details, because general mechanism are specified, not implementations.

The layer hierarchy is described by a hierarchy of classes. The service of the underlying layer is used by calling its methods. Sometimes it is necessary to inform a layer using a deeper layer's service about specific *events* in that layer. The events are signaled by polymorphic method calls. A layer signaling an event calls its abstract event method, which is overwritten by higher layers interested in processing that event. The polymorphic mechanism calls the top overwritten method first. This layer processes the event and calls the event method of the next deeper layer. The event soaks through the layers.

The notion of node is the basic element, out of which the storage network is built. In a pure P2P storage network, all nodes are equal and have the same functionality, even if in our case, all functionalities are not always used by all nodes at same moment. Hence, each node is an object instance of the same node class, implementing the node functionality. The functionality of the node is provided by the class hierarchy of superimposed layers. In order to collaborate with each other, the nodes need to communicate. The node to node message service is provided by the *Message Layer*, which itself uses a communication service of the underlying IP network layer. The overlay network which provides a look-up and routing service is maintained by the *Look-up and Routing Layer*. Reliable content block storage is handled by the *Block Storage Layer*. And finally, the *Application Layer* is storing files using the block storage service of the underlying layer.



-Figure 4.1- Framework Layer design and classes hierarchy

4.2.1. Message layer

The Message layer defines the basic behavior of a node, which is the capability of communicating with other nodes. The functionality of a node in this layer is provided by the *Node* class. Communication is based on two primitives, messages and responses. The communication initiator sends a message to another node, which receives the message, processes the message and returns a response according to the message. Each node has a unique address in the Message layer represented by a *Location* object which contains its IP address and a port number listening for incoming messages. The basic message object is implemented by the *Message* class and the basic response object is implemented by the *Response* class. By extending the *Message* base class, semantic meaning is added and specified by additional attributes. The base class only encapsulates the message originator's location. The *Response* class object has a boolean *success* field indicating if the processing was successful or not. If processing was not successful, the *reason* field should contain a string explaining the reason why the message processing failed.

For simple node-to-node communication the *sendMsg* method is used. It waits until the destination node has processed the message and returned a response. For recursive message processing, where several nodes are involved, the *sendRecursiveMsg* method must be called, with a *RecursiveMessage* object. This object extends the *Message* object with the *setResponse* method, which is called by the node that terminates the recursive message processing and supplies a processing response. The *sendRecursiveMsg* method waits until this method is called returning the response or a waiting response timeout has occurred. The processing itself is done by the layers on top of this one, which use the Message layer service. A layer using this service should overwrite the *processMsgEvent* method and handle processing of own messages by returning a response. Polymorphic method invocation results in calling the *processMsgEvent* method of the top-most layer class. Unknown messages, which cannot be processed are passed to the next deeper layer for further processing by calling the inherited *processMsgEvent* method of the deeper layer. If all layers above cannot process a message, it ends at the Message layer's *processMsgEvent* method, which provides a default message processing by setting the result to false.

4.2.2. Look-up and Routing layer

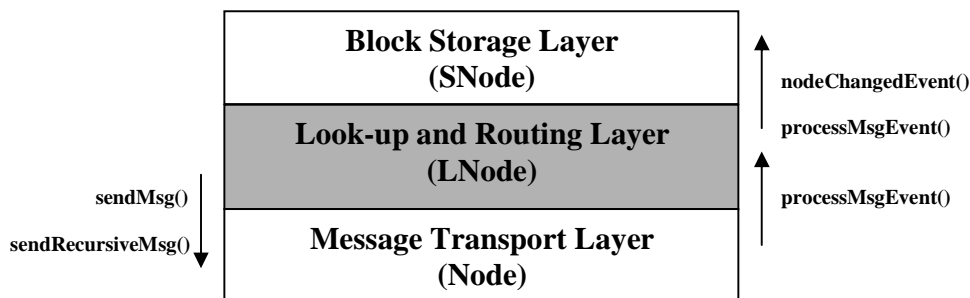
This layer provides the hierarchical look-up and routing service. The nodes are arranged in groups which are organized in a Chord ring. The functionality of a node in this layer is provided by the *LNode* class, which inherits directly from the *Node* class. Every node has two identifiers: an *idGroup* identifier, which is common to all the nodes in a group and an *idNode* identifier, which is unique for every node in a group. Since keys and node identifiers are generated by the same hash function, they are represented as instances of the *LHashID* class. Every node has in the Look-up and Routing layer a unique *LLocation* object, which extends the *Location* object of the Message layer with both node identifiers.

The look-up operation resolves the responsible node for a key. Firstly, for a given key, the look-up service find the responsible group to store the key using Chord rules: the responsible group is the $idGroup=successor(key)$. Immediately after, the responsible group for

a key has been found, the look-up process searches for the node in the group which is responsible for storing the key using CARP rules (see section 3.1.2).

There are some special nodes that are called *super peers*. There are a *NUMBER_SUPER_NODES* maximum number of super peers at each group. These nodes are chosen considering the bandwidth and CPU of the node. It is desirable to choose like super peers the nodes with better characteristics. These super peers are integrated in the top overlay network, that is to say, they build the Chord ring. The other nodes of the group don't know anything about the Chord ring. All nodes in a group know the list of super peers of the group. Every node can become a super peer at any moment. In fact, this is not a pure peer-to-peer system since there are peers that have more importance than the others. However, all the peers can perform both as super peer and "normal" peer.

The Look-up and Routing layer use the *sendMsg* and *sendRecursiveMsg* methods of the Message layer for node communication. The *processMsgEvent* method is overwritten and it is the way used by the deepest layer to pass events. The look-up method is called *findNodesResponsibles*. The look-up result is an array of *Successor* objects of each group. The *Successor* object contains the *idGroup* identifier of a group and a list of *LLocations* objects of the super peers in the group. In order to communicate node joins or leaves in a group to the upper layer, the Look-up and Routing layer provides the *nodeChangedEvent* method which must be overwritten by the upper layer.



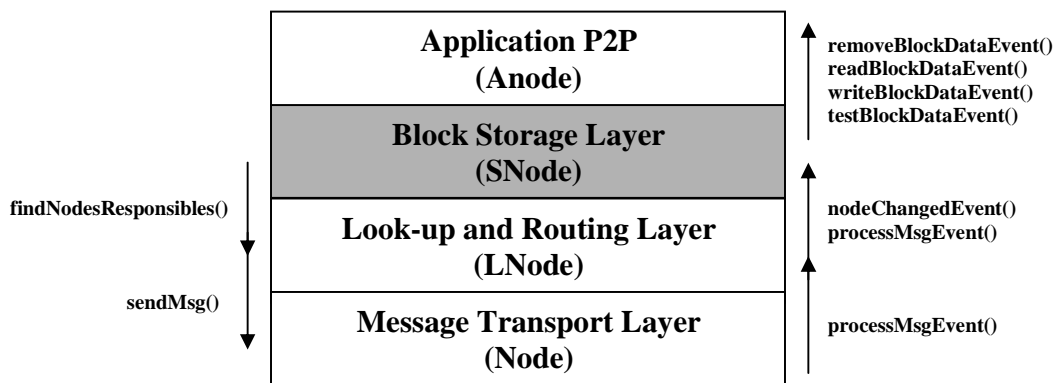
-Figure 4.2- Interaction of Look-up and routing layer with the other layers

4.2.3. Block Storage layer

The data storage is handled by the Block Storage layer. This layer offers a reliable block storage facility for storing, fetching and removing blocks using distributed redundant replicas for reliability. The basic storage unit is the data block, any kind of binary data represented by variable length byte sequence. Each block is identified by a unique *LHashID* object. The functionality of a node in this layer is implemented by the *SNode* class.

To achieve a more versatile approach, the layer itself does not read, write or delete block data. It logically manages the blocks and calls the *readBlockDataEvent*, *testBlockDataEvent*, *writeBlockDataEvent* and *removeBlockDataEvent* methods to let the

Application layer do the physical changes. It's up to the P2P application defined in the Application layer which storage devices are used for block storage and which block size is reasonable. The size should be chosen carefully considering that fine granularity helps to use the available storage resources more efficiently and balances the storage load among the nodes. On the other hand, the paraloader is less effective for smaller blocks due to the smaller number of block pieces downloaded in parallel. This layer uses the Look-up service offer by the Look-up and Routing layer to resolve the binding between keys and nodes and the Message layer for exchanging messages with other nodes. The Look-up and Routing layer report this layer about node leaves and joins into a group through the *nodeChangedEvent*. This kind of event causes a reorganization of the data blocks between the nodes that form the group. Each data block is stored in the node responsible to store the data unit in the group and in a set of other nodes.



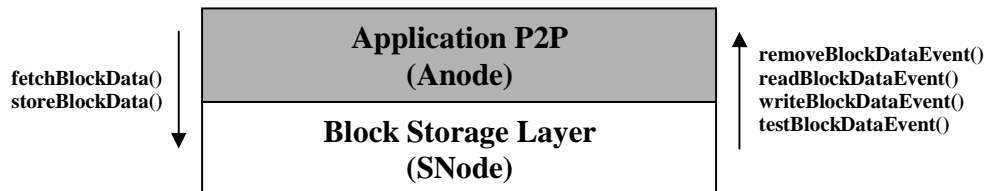
-Figure 4.3- Communication between the Storage layer and the others layers

4.2.4. Application layer

The Application layer is the topmost layer interfacing with the user. In this prototype implementation, the application offers a simple file storage interface to demonstrate the use of the underlying layers. The user can choose a file from his hard disk and file will be stored by splitting it into blocks and spreading them among the storage network's nodes. The file can be retrieved from the storage network and saved on the local hard disk by specifying a directory to write the file and the files original filename. The file's consistency is verified by comparing it with a stored hash fingerprint.

The framework is fully parameterizable by the Application Layer's *Parameter* class. In this class all parameters used for the different layers can be configured and adjusted to the needs of the application using the framework. The Application layer provides a simple graphic user interface (GUI) which is implemented by the *JFNodes* class. The *NodeTest* class implements the main method which runs the application. The user can create several nodes over the same machine. The nodes to be created are rode from a batch file which must be passed to the program as input.

The functionality of a node in this layer is implemented by the *ANode* class, which inherits directly from the *SNode* class. The *ANode* class uses the *storeBlockData* and *fetchBlockData* methods provided by the *SNode* class for storing and fetching a data block respectively. The *ANode* class overwrites the *readBlockDataEvent*, *testBlockDataEvent*, *writeBlockDataEvent* and *removeBlockDataEvent* methods. In this way, the Block Storage layer can pass events to the Application layer.



-Figure 4.4- Communication between the Application layer and the Storage layer

5. Message Layer Implementation

The Message Layer is designed as a general purpose node-to-node communication service based on *Message* and *Response* objects. Since this layer *Node* class is the base class for all other layers, it defines the basic properties and behavior of a node. The *Message* and *Response* objects are Java classes and the Java serialization is the mechanism used to send them between nodes. In this layer, a node is an instance of the *Node* class. A node is identified by the *Location* object's host IP address and the port number where the node is listening for possible incoming messages. After the node object is created, the *startup* method needs to be called for installing a specified number of pool threads waiting for incoming messages to process. The *shutdown* method should be called to properly shutdown a node by terminating the installed threads and remove their binding to the node's port. All classes based on the *Node* must call the *startup* and *shutdown* methods.

5.1. A pool of threads for managing messages reception

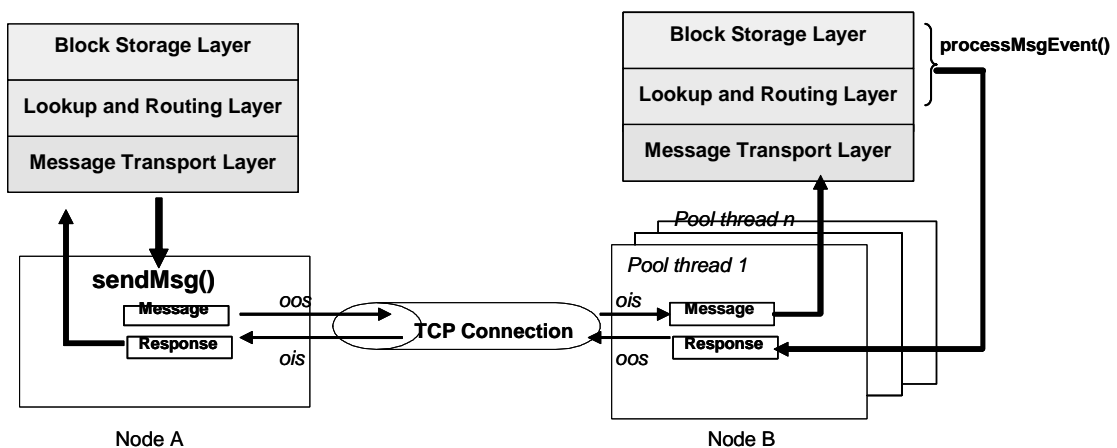
A node is listening for continuous incoming messages. To manage this, it would be possible to use the remote method invocation (RMI) to implement a node-to-node communication. But the standard remote method invocation frameworks like Java-RMI or CORBA create at least one thread per remote method call. For the Java-RMI, part of the standard Java distribution, it is reported that even three threads are created for each call[]. Since creating threads is an expensive operation for operating systems and a P2P application based on an overlay network will cause constant message traffic in order to maintain and stabilize its overlay network structure, using the standard solutions would cause a massive thread creation overhead. A Java specific mechanism, the garbage collector, makes things even worse. For convenience, Java objects do not need to be destroyed manually after usage. The garbage collector detects objects which are not referenced anymore and discards them. Therefore, they will stay in memory for an unspecified time until they are removed. This would also happen to the thread objects created for each incoming message call and occupy main memory. The key to avoid this overhead is thread re-use[]. A pool of threads is installed at startup, waiting for incoming messages. One of the waiting threads gets activated, handles the message processing and starts waiting again.

5.2. Node-to-node communication method

The node-to-node communication is done by using TCP connections over the underlying IP network. The *Message/Response* semantic is easier to implement with the bidirectional TCP protocol than using the unidirectional and unreliable UDP service, for which additional mechanisms for mapping responses to messages, fragmentation and error handling would be necessary to implement. It turned out that the Java socket implementation, when used together with multiple threads waiting at the same port, buffers incoming TCP

connections even if all pool threads are active. The next free thread handles the buffered connection, if a TCP connection timeout does not occur first. This buffering effect may smooth short load peaks. Figure 5.1 shows how the TCP-based Message/Response processing works. The *sendMsg* method tries to establish a TCP connection to the node's location. If a thread on node B is waiting, it gets activated and a connection is established. Otherwise a *LocationException* is thrown. After the connection is established, the corresponding *ObjectInputStreams (OIS)* and *ObjectOutputStreams (OOS)* get connected together by each of the unidirectional TCP data pipes. Node A sends a *Message* object serialized by the OOS and de-serialized by the OIS. Then Node B calls the processing method *processMsg* and returns the *Response* object in the same way. The default serialization methods of each individual *Message* derived class can be overwritten to pack the data more efficiently than the standard serialization implementation does.

The *sendRecursiveMsg* method is used to start a recursive message processing. The message object based on the *RecursiveMessage* class is sent to the node given by the destination location. On the other node, the implementation will recognize a recursive message by class introspection and return an acknowledgement *Response* object. Then, the *sendRecursiveMsg* starts waiting until the response for this message arrives or the timeout occurs. Each *RecursiveMessage* is identified by an eight byte message id randomly generated when its constructor is called. A node that terminates a recursive message processing calls the recursive message's *setResponse* method and sets the message's processing *response* attribute with a *Response* object. The *RecursiveMessage* object and the processing response are sent back to the message's originator node stored in its *org* attribute. The originator node receives the recursive message and recognizes from the non empty *response* attribute that a response to one of its recursive messages has arrived. Then it tries to identify the waiting thread from the unique message id and activates it. The thread calling the *sendRecursiveMsg* method is waiting at an *emphSemaphore*. A semaphore is a thread synchronization mechanism that was introduced by Dijkstra to solve the "lost signal" problem[1]. When a thread wants to enter a synchronized area it has to wait at the semaphore's *pass p()* method until another thread calls the *free v()* method signaling that the area is now free.



-Figure 5.1- Basic communication method among nodes

6. Look-up and Routing layer implementation

In this chapter we explain the implementation details of the Look-up and Routing layer. This layer provides the hierarchical look-up and routing service. The nodes are arranged in groups which are organized in a Chord ring. The functionality of a node in this layer is provided by the *LNode* class, which inherits directly from the *Node* class. All the nodes are instances of this class. The Look-up and Routing layer use the *sendMsg* and *sendRecursiveMsg* methods of the Message layer for node communication. The *processMsgEvent* method is overwritten and it is used by the deeper layer to pass events. In order to communicate node joins or leaves in a group to the upper layer, the Look-up and Routing layer provides the *nodeChangedEvent* method which must be overwritten by the upper layers.

There is a flag *isSuperNode* to distinguish a super peer from a “normal” peer. At every moment, any “normal” peer can become a super peer. In fact, this is not a pure peer-to-peer system since there are peers that have more importance like the others in a specific moment, but in fact, all the peers can perform as super peer or as “normal” peer at any moment. There are `NUMBER_SUPER_NODES` super peers at each group. Each node has two identifiers, the *idGroup* identifier, which is unique for all the nodes in the same group and the *idNode* identifier, which is unique for each node in the group. Both of these identifiers are represented by *LHashID* objects. Every node has in the Look-up and Routing layer a unique *LLocation* object, which extends the *Location* object of the Message layer with both node identifiers. We give more details about these identifiers below.

6.1. Node identifiers

Node identifiers are *m-bit* long and interpreted as non-negative integers. They are generated by a hash function. A hash function maps any variable length input data (an array of bytes) to a fixed *m-bit* length hash value. Hash functions are often used in cryptography, especially for integrity checking. The most used cryptographic hash functions are MD5 and SHA-1. The length of the MD5 hash values is 128 bit compared to 160 bit for SHA-1. Since the number of bits directly determines the number of addressable content items, the SHA-1 algorithm with its higher length of 160 bit and a resulting number of 2^{160} different possible content items was chosen for this implementation.

Using cryptographic functions in Java is standardized by the Java Cryptographic Extension (JCE) framework, which defines basic interfaces and classes implemented by a cryptographic provider, like the freely available *Cryptix* library used for this project. The identifiers are encapsulated in the *LHashID* class. The static *getLHashKey* method of the *LHashID* class, called with an input byte array as parameter, uses the SHA-1 function to generate the 20 byte (160 bit) hash value from the input data. From these 20 bytes, a positive *BigInteger* object is created and encapsulated in the *LHashID* object. For working with these

identifiers, a *LOpenHashIDIntervall* class is used for testing if an identifier is inside this open interval defined by two *LHashIDs*.

6.2. Top-level overlay network

The peers are organized in groups. These groups are organized in a top-level overlay network. In this work, we have chosen Chord for the implementation of this top-level network. There is a limited number `Parameter.NUMBER_SUPER_NODES` of super peers at each group. These special peers are supposed to be the best nodes of the group in accordance with a metric. This metric is called the *merit* factor. This merit factor must take account of properties of node (CPU power, bandwidth of the node and time-up). The merit of a node is obtained by the *getMerit* method. All the nodes in a group have at every moment the list of the super peers of the group. This list is called *superNodesList*. In this work, the implementation of the Chord system has been modified to support the hierarchical look-up service. Now, there is a Chord ring of groups instead of a Chord ring of nodes. Every group is represented by a *Successor* object. This object contains the *LHashID idGroup* parameter, which is the identifier of the group and the *LinkedList nodesGroup* parameter, which is the list of super peers of this group. Therefore, we have a Chord ring of *Successor* objects. All super peers are able to do look-ups through the Chord ring. This is achieved by the *findKeyOverlayNetwork* method. This method performs the Chord inter-group look-up. All the other “normal” nodes of the group can look-up in other groups through any of the super peers of the group. This is done through the method *getSuperpeerForKey*. Only the super peers are integrated in the Chord ring. For this purpose, every super peer has a list of the *Successor* objects of its next groups in the Chord ring. This list is named *successors* list. All super peers have another list with all the super peers in its predecessor group. This list is named *predecessors* list.

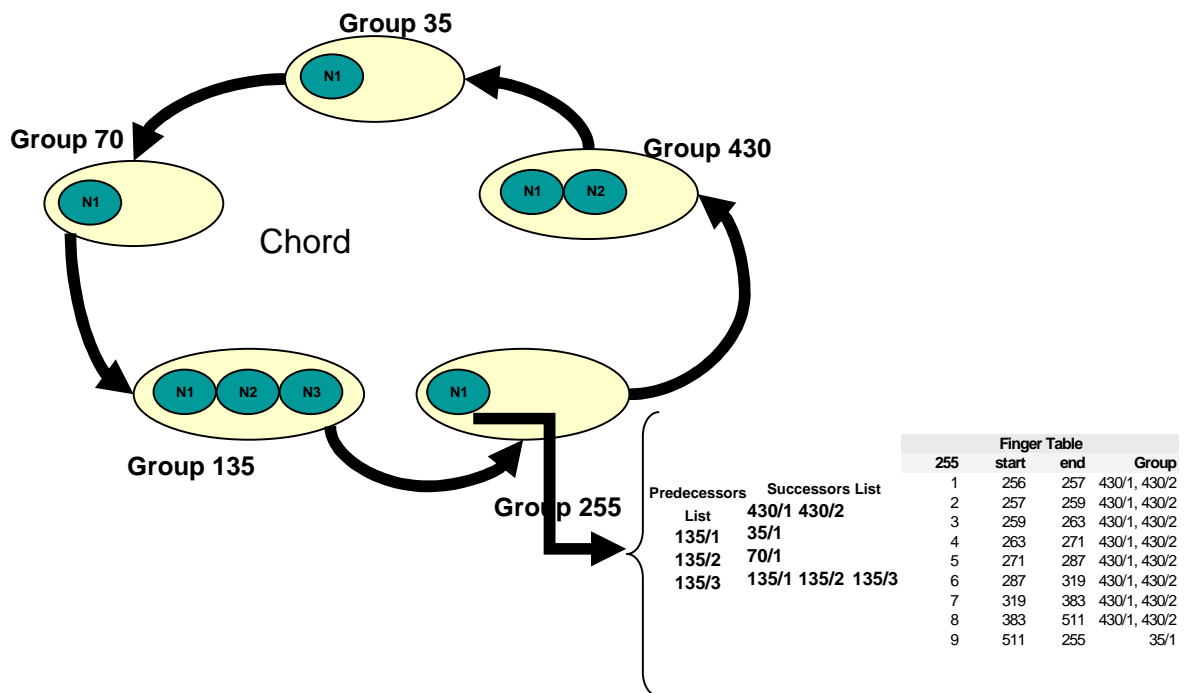
6.2.1. Successors and predecessors

Each super peer must have a pointer to its immediate successor group, that is, it must know the *Successor* object of the next group in the top-level overlay network. This is required for a successful look-up. A single successor pointer is not reliable in a dynamic scenario with frequent joining and leaving groups. If a group fails, the successor pointer is not working until the *stabilization* process has assigned the new successor group. In the worst case, this could cause a look-up to fail. Therefore, redundant successor pointers are used. They are implemented by a linked list of *Successor* objects, the *successors* list. The list is kept in ascending group *ids* order by inserting new groups at the position according to their group *ids*. The list has a maximal size specified by the *Parameter.SUCCESSOR_LIST_SIZE* constant, which should be set to $2\log_2(N)$, where N is the number of maximal expected groups. When the list has reached its maximal size, the entries at the end of the list are removed to get space for new entries with smaller *ids*. To find the successor, the list is searched from the beginning. We try to contact each of the super peers of this group until the first working super peer is found. If none of them answer, this group is not working and is removed from the list. The *successor* list is constantly filled when the *stabilization* process is executed. This is how new successor groups are discovered. More details about this method in subsection 6.2.4.1. The

access to the *successors* list is controlled by a flag *blockSuccessorsList* to avoid synchronization problems caused by of multiple threads accessing to the list at the same time. New *Successor* objects are introduced in the *successors* list through the *addSuccessor* or *addSuccessorGroup* methods and they are removed by the *removeSuccessor* method. Since predecessors are not necessary for the look-up, only the list of the immediate predecessor group is used.

6.2.2. Fingers

Correct finger table entries are necessary for a fast look-up while the successors are necessary for a reliable look-up. For the *m-bit* identifiers used in this implementation, the number of finger intervals is *m*. The *finger table* is implemented by an array of *m Finger* objects. A *Finger* object contains the finger interval start as *LHashID* object calculated when a node startups, because it does not change after the group *id* is set. This is done through the *initFingers* method. The finger pointers themselves do change due to joining and leaving groups or changing of super peers in the groups. Any entry in the *finger table* points to a *Successor* object of the group which has the next *idGroup* identifier bigger than the beginning of the interval of this table entry. The *finger table* entries are updated that is, they are periodically refreshed each *Parameter.FIXFINGER_INTERVAL_MS* (ms) by a thread which executes the method *refreshFingers* as it's described in the Chord paper.



-Figure 6.1- An example of the top-level Chord overlay network

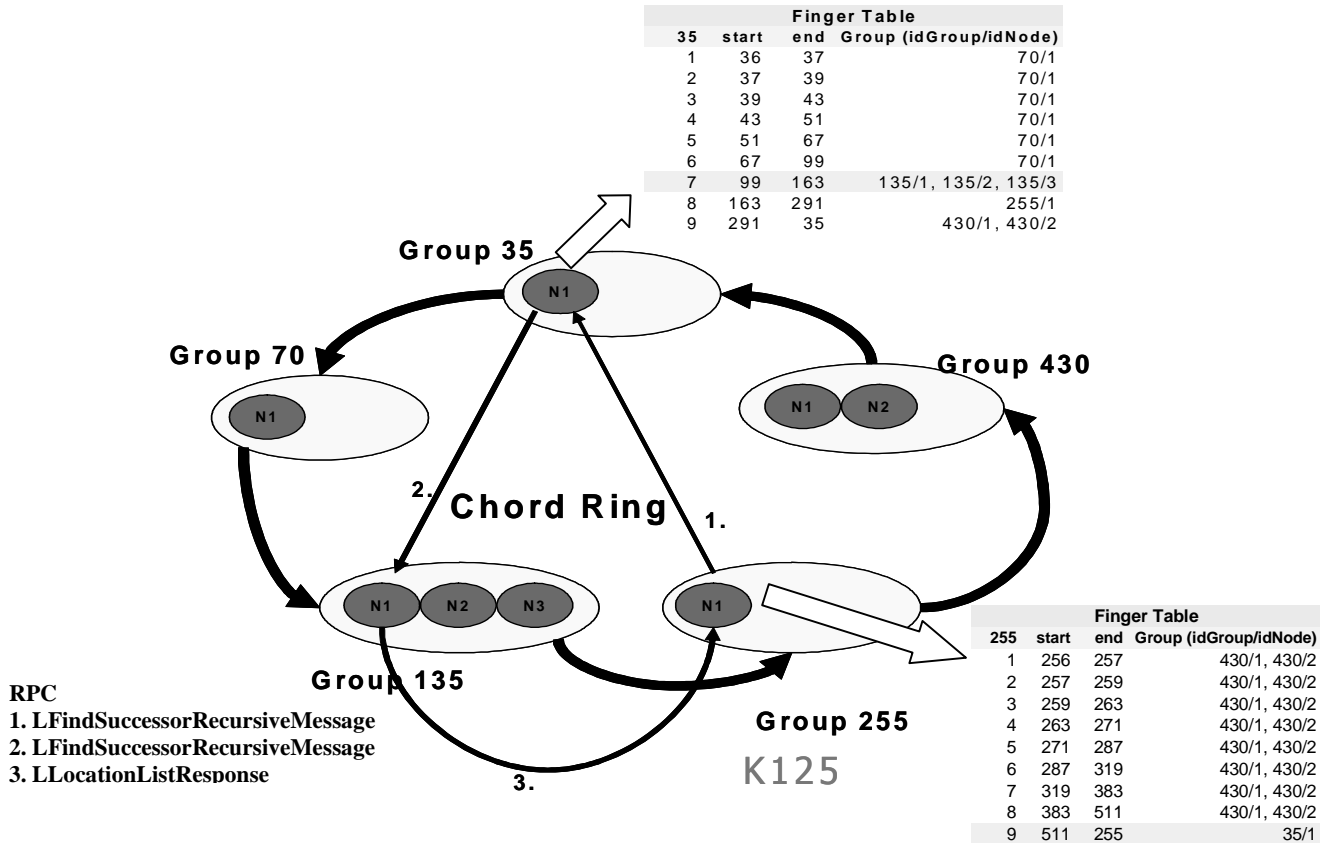
6.2.3. Inter-group look-up

The top-level overlay network as we have already said is a Chord ring. The inter-group look-up operation resolves the responsible group for a key: the responsible group is the group whose *idGroup* identifier is $idGroup = successor(k)$, where k is the searched key. The look-up is resolved using a recursive method. In the Chord paper is explained how to do an iterative look-up, but the recursive method is better because the number of messages sent is smaller and privacy of the sender is preserved. Only the super peers are able to achieve inter-group look-ups. This is done by the *findKeyOverlayNetwork* method. The node that wants to do the look-up, tries to contact a super peer of the first group in the look-up path. If this super peer doesn't answer, it tries to contact another in the same group. Each node receiving a query routes it to the next hop of the look-up path and the node that finally can resolve the successor of the look-up *id* has to return the *Successor* object of the responsible group to the node that initiated the look-up. Therefore the initiating node has to wait passively until it receives a result or a timeout occurs.

The recursive look-up is started by calling its *findKeyOverlayNetwork* method, which locally calls the *recursiveFindSuccessorList* method, which locally calls the *routeRecursiveFindSuccessor* method. This method routes the recursive look-up query by recursively calling the *routeRecursiveFindSuccessor* method on a super peer of the next closest preceding group until the successor group is resolved and the recursion stops. On the first local invocation of the *routeRecursiveFindSuccessor* method, an *LFindSuccessorRecursiveMessage* object is sent to a super peer on a preceding group with a *recursiveSendMsg* call. This super peer is determined by the *finger table*. If the *finger table* can't do the look-up, the *successors* list is used for routing this message to a super peer in a group "close" to the searched group. After sending the message to a super peer of the computed destination group and receiving an acknowledging *Response* object, the thread starts waiting until the result arrives or the timeout occurs. On the destination super peer, the *routeRecursiveFindSuccessor* method is called forwarding the message in the same fashion until the *setResponse* method is called and the result is returned to the super peer that initiated the look-up. The super peer that had initiated the look-up receives as response a *LlocationListResponse* with the *Successor* object of the responsible group for the searched key. The number of message pairs used for a recursive look-up is $l+1$, where l is the number of hops. At figure 6.2 we can observe an example of this look-up.

6.2.4. Join and leave of groups

In order to maintain the Chord circle topology new groups and new super peers must be integrated and groups and super peers that leave, fail or change must be detected and removed. In Chord there is no explicit mechanism for a node to leave. This case is treated like a node failure.



-Figure 6.2- An inter-group look-up example

6.2.4.1. Stabilization process

The *stabilization* process plays an important role in the maintenance of the Chord ring. The stabilization process is a distributed algorithm constantly performed by each super peer to detect failures and repair the top-level overlay network. In intervals defined by the *Parameter's* class *STABILIZE INTERVAL MS* constant, the *stabilize* method is executed and each super peer asks to a super peer of its immediate successor group for its *predecessors* list. For this purpose, a super peer sends an *LGetPredecessorMessage*. If a super peer in the next group doesn't respond, it is removed and another super peer in the same group is tried. If none one of the super peers of the next successor group responds, the group has disappeared and is removed from the *successors* list. This way, disappeared successor group are reported. The next group of the *successors* list is contacted. In a stable situation the successor's predecessor group is the same group. In the unstable situation, the super peer asked in the next successor group return another different group. The stabilization process tests if this new group is a closer successor group and updates the *successors* list if necessary. This way, new groups are discovered by the predecessors groups. In the stable case, the successor list is tried to be updated. For this purpose, the node tries to send an *LGetSuccessorListMessage* to a responding super peer of the *successors[1]* group. The *successors* list of this super peer is returned and is compared with our *successors* list in order to realize of changes discovered by successors groups.

6.2.4.2. Notification

A super peer of a predecessor group calls its successors *notify* function to inform a super peer in the successor group about its presence. The predecessor super peer sends an *LNotifyMessage* message to its successor super peer who tests if the node that sent the message was not stored in this *predecessors* list or if it belongs to a new group. If the *idGroup* of the sender node is “closer” (it’s between its *idGroup* and the *idGroup* of its old predecessor group) or the *predecessors* list is empty, the new predecessor super peer is stored in the *predecessors* list which is updated with the new group. This way, successor groups discover new predecessor groups. If the super peer belongs to the same group than the super peer stored in the *predecessors* list, is the case where a super peer in the predecessor group has changed. All of the old predecessor super peers are pinged by the *verifyPredecessors* method to see if they are still up. Not responding predecessor super peers are removed from the *predecessors*. The new *predecessors* list is sent to the other super peers of the group with an *LsendPredecessorsListMessage*. By this way, successors super peers discover super peer changes in its predecessor group.

6.2.4.3. Group join

Any node that wants to join an overlay network needs to know at least one bootstrap node which is already part of the system. This node could belong to any group and it may be any peer, since every “normal” node can do look-ups in other groups through the super peers of its group. This bootstrap node is given as an argument to the *join* function. In a first step, the new node sends the *idGroup* identifier of the group that it wants to join to the bootstrap node. To do so, the node sends an *LFindRemoteSuperNodeMessage*. The bootstrap node returns a *LlocationResponse* with an array of *Successor* objects. If the first *Successor* object of this array is not a *Successor* object representative of the group which the node wanted to join, the group doesn’t exist. This way, the new node knows that it must create a new group and it executes the *createGroup* method. The array returned by the bootstrap node is the new *successors* list for the new node. Therefore, the new node knows the position of its new group in the circle and informs a super peer in its immediate successor group about its presence by sending a *LNotifyMessage*. The super peer in the next group discovers the new predecessor group and reports to the other super peer of its group. By the execution of the stabilization process, the super peers in nearby groups discover the new group.

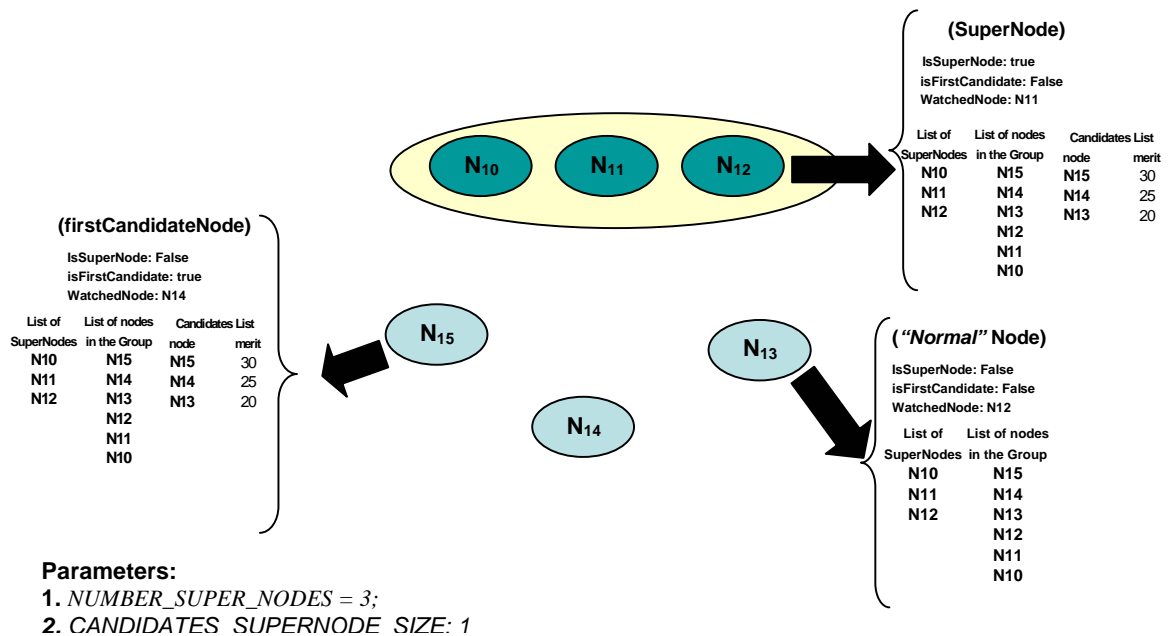
6.3. Lower-level overlay network

The nodes are organized in CARP groups. At each group there is a number of *Parameter.NUMBER_SUPER_NODES* super peers. The other peers are called “normal” peers. The super peers are gateways between the groups: they are used for inter-group query propagation. Only the super-peers are integrated in the Chord ring and therefore, only they are able to do look-ups in other groups. The super peers are intended to be the best nodes of the group in terms of bandwidth, CPU power or up-time. In this work, the merit factor is just the product of a bandwidth quantification factor with a CPU process quantification factor of the nodes. The method *getUpTime* returns the up-time of a node.

6.3.1. Node group information

In this section, we are going to enumerate the information that every node has about the group. There are two types of nodes: the super peers and the “normal” peers. All the nodes have a boolean flag, called *isSuperNode*. If true, this node is a super peer. Every node can become a super peer at every moment executing the *becomeSuperNode* method. This method sends an *LNotifyMessage* to a super peer in the next group to warn the super peers of the successor group about the new predecessor super peer. The best “normal” nodes of the group, the nodes which have the best merit factor to become a super peer, are called “*first candidates*” nodes. All the nodes have a boolean flag *isFirstCandidate*. If true, this node is a “first candidate” node. These nodes act like a normal peer but they have all the Chord ring information like a super peer, in order to become a super peer immediately after a super peer failure. The super peers inform to the normal nodes if they are a “first candidate” node by the *notifyFirstCandidates* method. A super peer sends an *LNotifyFirstCandidate* message to the “normal” nodes for this purpose.

All the nodes in the group know all the other nodes. Every node has a list, called *nodesList*, with all the other nodes of its group. This list stores *NodeGroup* objects which just store the *LLocation* identifier object of a node. All the nodes know which nodes are the current super peers of the group. For this purpose, all the nodes have a list, called *superNodesList*, which contains the *LLocation* identifier objects of the super peers of the group. This list is used for doing look-ups in others groups. Only the super peers and the “first candidate” nodes have another list, called *candidatesList*, where all the “normal” nodes of the group are ordered by its merit factor. This list contains *NodeCandidate* objects. These objects contain the *LLocation* identifier object of a node and its merit factor. In order to report to upper layers when a node has failed without sending a notify message, at section 6.3.3 there are explained the several surveillance mechanisms implemented. By one of these mechanisms, every *Parameter.WATCH_INTERVAL_MS* (ms), a node makes a ping to the node contained in this variable *watchedNode* to verify if it is still up. This variable stores the *LLocation* object of the current watched node. This watching node is always the node of the group with next lower identifier to a node. This node is obtained by the *getNextNode* method. At figure 6.3 we can observe a simple example of a group with the group information that has every type of node.



-Figure 6.3- A CARP group example

6.3.2. A node joins the group

A node, in order to join the system, must know the *LLocation* object of a current node in any group. The new node knows the *idGroup* identifier of the group that it wants to join. The new node also knows its *idNode* identifier in this group and the quantification factors of its bandwidth and CPU power. First, the new node sends an *LFindRemoteSuperNodeMessage* to the bootstrap node. This message contains the *idGroup* identifier of the group that it wants to join. The bootstrap node looks up in the system if this group already exists. If it's this case, it returns to the new node an *LFindRemoteSuperNodeResponse* with the *Successor* object of this group. The new node integrates an existing group through the *integrateGroup* method. If the group didn't exist, the new node must create a new group. In this case, the new node executes the *createGroup* method. When a node integrates a group, it knows the *Successor* object that identifies the group. This object contains the *LLocation* objects of all the current super peers of this group. It sends to the first responding super peer an *LRemotePresentationMessage* with its "credentials", CPU, bandwidth quantification factors. The super peer decides if the new node must be a super peer if there aren't enough super peers. The super peer returns to the new peer all the lists that it needs:

- If the new node is a new super peer, the bootstrap super peer gives to the new super peer the list of nodes of the group (*nodesList*), the list of super peers of the group (*superNodesList*), the list of the "normal" nodes of the group ordered by the merit factor (*candidatesList*), the list of *Successor* objects of the next groups in the top-overlay network (*successors*), and the list with the *LLocation* objects of the super peers of the predecessor top-level overlay network's group (*predecessors*).

- If the new node is a new “normal” peer, the bootstrap super peer gives back to the new “normal” peer just the list of nodes of the group (*nodesList*), the list of super peers of the group (*superNodesList*).

The new node fits its *isSuperNode* variable and its *watchedNode* variable. Immediately after, the new node sends an *LNotifyArrivalMessage* to all the other nodes of the group. When a node receives this message, it executes the *nodeChangedEvent* method to prevent the upper layers about the new node join in the group.

6.3.3. A node leaves the group

There are two possible situations: when a node leaves the group in a “friendly” way and when it fails suddenly. In both cases, when a node leaves the group, an *LNotifyLeaveMessage* must be sent to all the other nodes of the group. This message contains the *LLocation* identifier objects of the leave node. If the node leaves the group in a “friendly” way, the leaving node sends this message by the *leaveGroup* method. On the other hand, if a node falls down and it can’t send this warn message to the other nodes of the group, it is another node which sends this message. This *LNotifyLeaveMessage* message contains:

- The *LLocation* object of the failing node.
- If the failed node was a super peer, it sends the updated *superNodes* and *candidatesList*. In this case, the leaving node warns the new “first candidates” nodes too.

When a node receives this message, it updates their *nodesList* and *superNodesList*. It executes also the *nodeChangedEvent* method to warn the upper layers about the node leave. For looking after sudden node failures, there are a number of control mechanisms of surveillance among the nodes:

1. The super peers make a ping every *PINGSUPERNODES_INTERVAL_MS* (ms) to the other super peers of the group. This constant time is specified at the *Parameter* class. If a super peer discovers that another super peer is no more up, it sends an *LNotifyLeaveMessage* from the leaving node executing the *changeSuperNode* method.
2. The “first candidate” nodes make a ping every *PINGSUPERNODESFIRSTCANDIDATE_INTERVAL_MS* (ms) to all the super peers of the group. This constant time is specified in the *Parameter* class. Only if all the super peers have suddenly failed, the “first candidate” node sends an *LNotifyLeaveMessage* for all of them.
3. All the nodes watch another node of the group. Every *WATCH_INTERVAL_MS* (ms), every node makes a ping to its watched node to verify if it is still up. This constant time is specified at the *Parameter* class. If it has failed, it sends an *LNotifyLeaveMessage* for the failed node only if it wasn’t a failed super peer, since in this case another super peer will do it.

6.4. Hierarchical look-up service

The Look-up and Routing service offers to the upper layers just one service: it resolves the look-up problem of the peer-to-peer systems and provides just one operation:

- Look-up(key) \rightarrow IP address.

where *key* is the identifier of a content item and the IP address is that of the peer responsible of its storage. This layer doesn't store the data items. The *ids* are calculated with a hash function:

- Key identifier = SHA-1(key).
- Node identifier = SHA-1(IP address).

This look-up can be achieved by any node of the group. The node that executes the look-up can contact directly the responsible node for the searched *key* and can directly retrieve the data associated with this *key*.

As we have seen in this chapter, the peers are organized in a hierarchical two-level peer-to-peer system, so the look-ups are done in two steps. First, the responsible group for a key must be found. Immediately after, the responsible peer of storing this key in the group must be looked up. As we have also seen, the groups are organized in a Chord top-level overlay network and the peers are organized into the groups in CARP's lower-level overlay networks.

Only the super peers are able to perform a Chord look-up (an inter-group look-up). A Carp search (an intra-group look-up) can be done by every peer. We present the functions which allow a peer *pi*, to locate the peer *pj* which is responsible for key *k*. The peer *pi* is in the group *Gi* and has an IP address @*pi*, and the peer *pj* is in a different group *Gj* and has an IP address @*pj*.

@*sj* = *getSuperpeerForKey*(@*si*, *k*): The peer *pi* calls this function to obtain from any super peer of its group *Si* the IP address @*sj* of a super peer *Sj* in the responsible group for key *k*.

@*sj* = *findKeyInOverlayNetwork*(*k*): To answer the call *getSuperpeerForKey*, the super peer *Si* uses *findKeyInOverlayNetwork* to obtain the IP address @*sj* of the super peer *Sj* of the group responsible for key *k*. This function implements inter-group Chord look up.

@*pj* = *getNodeForKey*(@*sj*, *k*): A regular peer *pi* calls this function to obtain from the super peer *Sj* of the group *Gj* responsible for key *k* the IP address @*pj* of the peer *pj* responsible for *k*.

@*pj* = *findKeyInGroup*(*k*): To answer the call *getNodeForKey*, a super peer *Sj* uses this function to obtain the IP address @*pj* of the node *pj* responsible for *k*. This function implements the CARP intra-group look-up.

The two step look-up is observed at figure 3.6. In this work we are using this look-up service to implement a reliable block storage service over it. These functions have been modified a bit. The reliability of blocks storage is achieved storing each block in a set of nodes in the responsible group. In addition, a block can be stored in a number of successor groups of the responsible group. For this purpose, in order to give a more efficient service to the upper block storage layer, the *findNodesResponsibles* method has been implemented. This function returns a list of *StorageNode* objects for each responsible group storing a block identified by the key *k*. This *StorageNode* object contains the *LLocation* identifier object of a node and the CARP hash value of this node for the specific key *k*. This way, the upper layer makes a call to the Look-up and Routing layer through the *findNodesResponsibles* method to find the set of nodes where a given data block must be stored.

7. Block storage layer implementation

This layer aims to implement a reliable distribute storage functionality over the Look-up and Routing layer. The functionality implemented by the Block Storage layer consists in three main tasks: storing a data block, fetching a data block and the automatic reorganization of the data blocks between the peers due to overlay network changes. The specific behavior of a node in this layer is implemented by the *SNode* class. It uses the Message layer for communication among nodes and the Look-up and Routing layer to map responsibility for data blocks to nodes and resolve their network addresses. The Message layer communicates the new messages to the Storage layer through the *processMsgEvent* method and the Look-up and Routing layer, communicates the changes in the overlay networks through the *nodeChangedEvent*.

The basic storage unit is the data block, any kind of binary data represented by variable length byte sequence. Each block is identified by a unique *LHashID* object. This layer offers a reliable block storage service using distributed redundant replicas for reliability. Each data block is stored in the node responsible to store the data unit in the responsible group and in a set of $\text{Parameter.REPLICATION_FACTOR}-1$ other nodes in the responsible group. In order to increase the reliability, each data block could be stored in the same way by a number of $\text{Parameter.REPLICATION_GROUP_FACTOR}-1$ successor groups to the responsible group. This layer itself does not read, write or delete block data. It logically manages the blocks and calls the *readBlockDataEvent*, *testBlockDataEvent*, *writeBlockDataEvent* and *removeBlockDataEvent* methods to let the Application layer do the physical changes.

7.1. Basic Elements

Firstly, some basic elements of the Reliable Storage layer are introduced because of their importance in the implementation of this layer.

7.1.1. Metadata Hash

For any data block, there is a set of responsible nodes charged of storing it. A hash identifier is calculated for each data block. If key k is the hash value identifier for any data block, the responsible group of storing this data block is the group with the same or the next higher group hash identifier than the key k . In order to increase the reliability, the block data is also stored in the successors groups of the responsible group in number of $\text{Parameter.REPLICATION_GROUP_FACTOR}-1$. Into a group, there are $\text{Parameter.REPLICATION_FACTOR}$ node responsables of storing the data block. This set of nodes are chosen to be the nodes which maximize the value $h(k, P_i) \forall P_i \in P$, where $P = \{P_1, P_2, \dots, P_r\}$ is the set of nodes of the group and h is the hash function .

A node stores a *SKeyMetaData* object for each data block stored. This object contains a *LHashID* key identifier of the data block. The *keyDataBytes* attribute defines the number of

bytes of this data block. The *storageNodes* attribute is the list of nodes where this block data is stored in this group.

A stored file is composed by a set of block data everyone stored in their responsible nodes in the responsible groups. There is a special object called *FileMetaData* which contains the set of blocks identifiers of all the data blocks that make up the stored file. This *FileMetaData* is managed like another data block and its hash identifier is calculated from the file name.

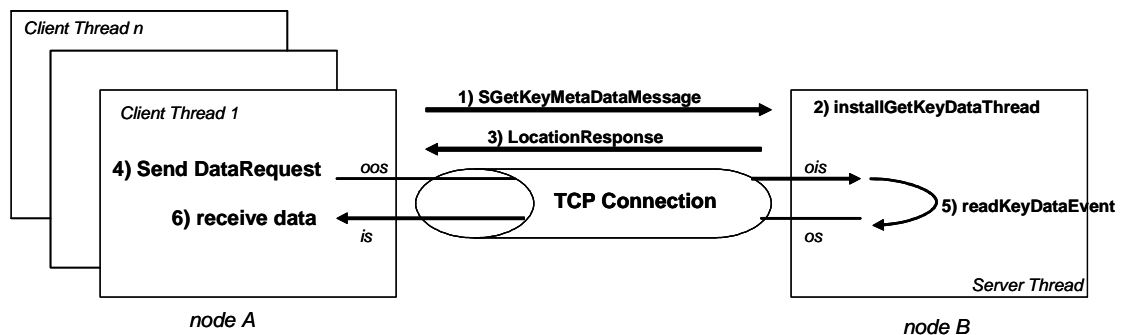
Of course, a node can be responsible for different data blocks. Each node has a hash data structure for uniquely mapping block data keys to *KeyMetaData* objects for which this node is responsible. This hash will be referred as *RNMetaData* and is implemented using the Java *HashMap* class which provides constant time performance for the basic operations *put* and *get*.

7.1.2. Paraloader

The parallel download mechanism, short paraloader[18], is implemented in the *SNode* class method *downloadBlockData*. This method takes an array of *SKeyMetaData* objects and performs a parallel download from the locations of the nodes contained in these objects. Since it is a parallel process, a multi-threaded implementation is necessary. We can observe this process at figure 7.1. For each download node location a client thread is created. First, a client thread sends a *SGetKeyDataMessage* to its download node. The receiver node checks if the desired data block is really available by a call to the *testBlockDataEvent* method. Since it is an event, the Application layer's implementation is called and it verifies if the data is physically available on the storage device. If the data is available, a server thread is installed waiting on a free TCP port between the *MIN DATA PORT* and *MAX DATA PORT* constants of the *Parameter* class. After the server thread was successfully installed, the port number and IP address is returned by a *LocationResponse* object. Otherwise the failure flag of the response object is set and the client thread gets to know that a download from this location is not possible. The client thread establishes a TCP connection to the given IP and port address and the two JVMs connect their streams. For the upstream direction, seen from the client thread, an *ObjectOutputStream* (oos) and an *ObjectInputStream* (ois) are connected together which allows to send serialized *DataRequest* objects to the server thread. This *DataRequest* object contains the offset position and the length of the block segment requested by the client thread. Serialization is used here, because it handles the difficulties with "big indian" and "little indian" integer conventions on different platforms. The server thread waits for an incoming *DataRequest* object, calls its abstract *readBlockDataEvent* method to get the requested data from Application layer and writes it to the binary *OutputStream* (os). Then it starts waiting for the next *DataRequest* object. On the client thread's side, the data is read from the *InputStream* and the segments are reassembled to reconstruct the block data.

The block to download is split into segments of length given by the *BLOCK SEGMENT SIZE* constant set in the *Parameter* class. All segments of a block that have to be downloaded are put into a *Workpile* object which represents a kind of job stack. A free client thread takes the next segment download job from the *Workpile* and tries to download it. If downloading fails because a connection cannot be established or the connection was closed by the server thread, the job is put back into *Workpile* and another client thread has to download

the segment. A client thread with a fast connection will download his first segment fast and take the next job from the *Workpile*. Therefore more segments will be downloaded from the fast connections and efficiency increases. It is important that there are be more segments than download connections, because otherwise the time until the slowest connection has finished is not used by the other client threads to download the remaining segments. The segment number should be a multiple of the number of download locations. For each segment that was successfully downloaded the synchronized counter *BlockDone* gets incremented by one. Synchronization is necessary to be sure that the number of downloaded segments is not counted incorrectly due to unlucky timing of two threads that want to increment the counter at the same time. Each client thread runs a main loop fetching download jobs from the *Workpile* until the *BlockDone* counter has reached the total number of segments of block. After terminating the main loop each thread waits at a *Barrier* class object. The *Barrier* object is necessary to synchronize the thread that called the paraloader and the parallel client threads. The calling thread must wait until all client threads have terminated because only then the block is completely downloaded. A *Barrier* object is initialized with an initial value i . This *Barrier* object blocks $i-1$ threads calling its *check* method and releases them when the i^{th} thread calls its *check* method. Here in this case, the *Barrier* object gets initialized with $i=l+1$ where l is the number of different node locations used for the download. The thread that calls the *downloadBlockData* method waits at the *check* method and is released when all l client threads have terminated.



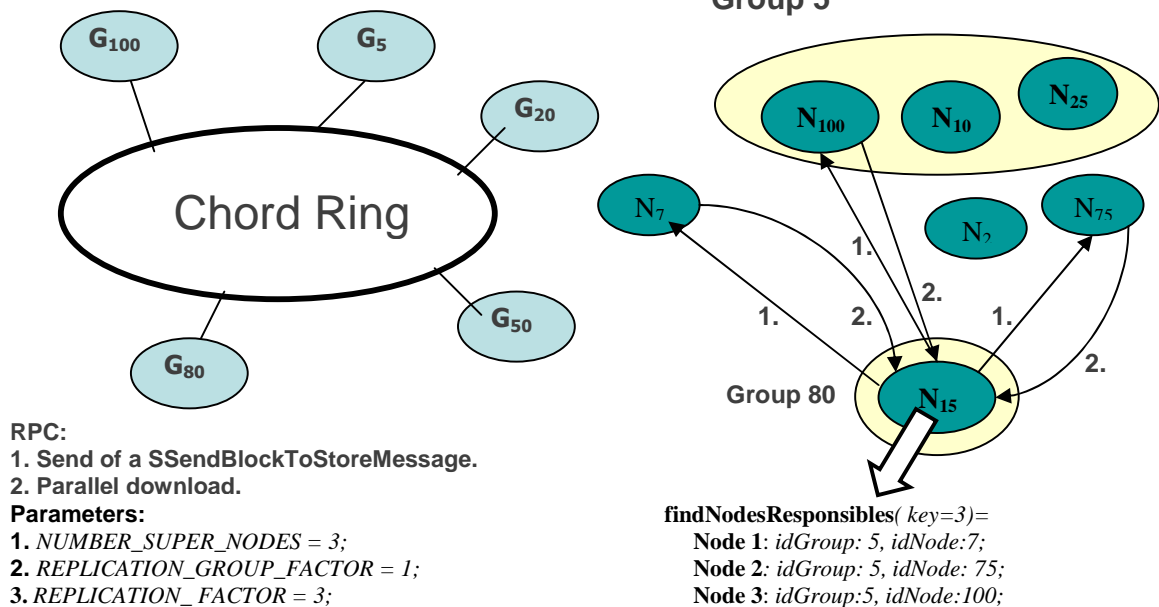
-Figure 7.1 – Parallel download process

7.2. Storing a block data

In this section we describe how a data block is stored. This functionality is implemented by the *storeBlockData* method of the Store Block layer. The Application layer gives to the Storage layer an array of bytes to store with the hash key identifier of this data block. According to this key, the set of nodes where the data block must be stored is searched by the *findNodesResponsibles* function provided by the Look-up and Routing layer. The *findNodesResponsibles* method of the Look-up and Routing layer, returns a number of $\text{REPLICATION_FACTOR} * \text{REPLICATION_GROUP_FACTOR}$ nodes to store every data block. An *SSendBlockToStoreMessage* message is sent to every node responsible and when every one of these nodes receives this message, they download the block data from the node

that is storing the block data. This download is done by the *downloadBlockData* method. The nodes use the *paraloader* to download the data block, even if in this case, the data block is downloaded from only a node. Figure 7.2 shows the steps to be performed to store a block in the Block Storage layer. The method *writeBlockDataEvent* physically stores the block on the storage device. The *SSendBlockToStoreMessage* also contains the rest of responsible nodes in the group to store the block data, so each responsible node creates a *SkeyMetaData* for the block and they add it to their *RNKeyMetaData*.

Example: N15 of G80 stores block data with id 3

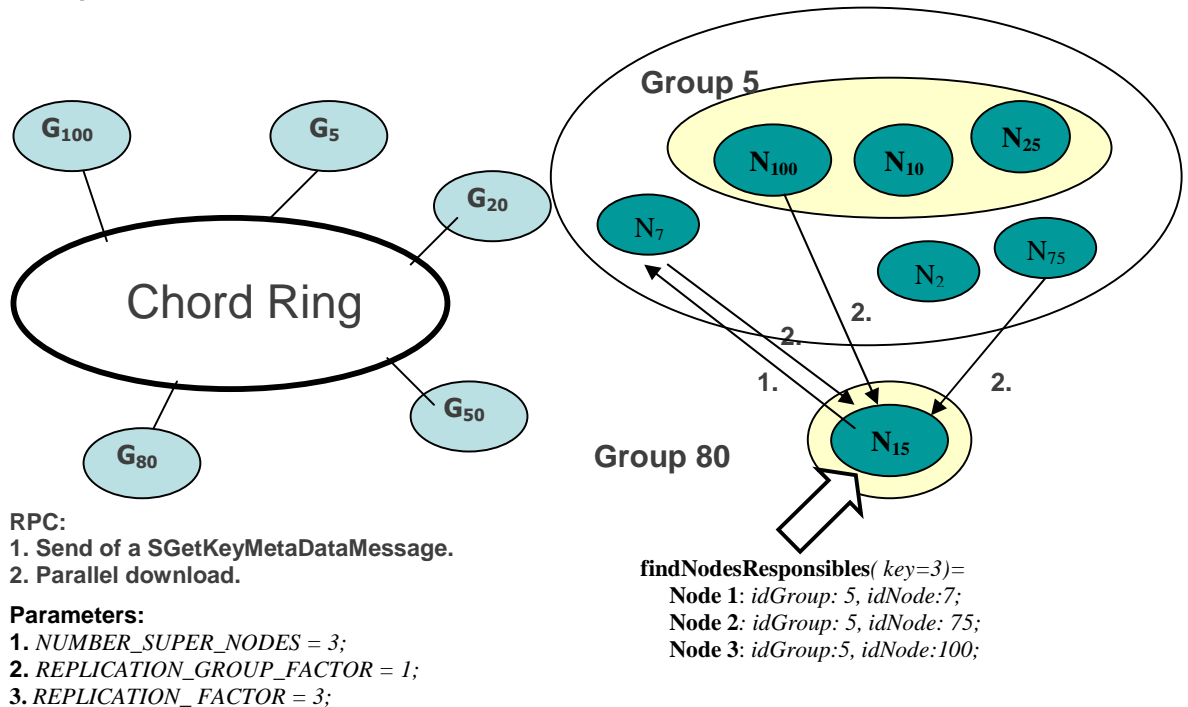


-Figure 7.2- Storing a data block

7.3. Fetching a block

Due to the separation of block data and metadata, a node *n* that wants to fetch a block from the network has to execute two steps: gets the *FileMetaData* object for this data block, containing the download locations of the responsible nodes that contains this data block and perform a parallel download from these locations. This functionality is implemented by the *fetchBlockData* function of the Storage Block layer. In order to find the responsible nodes where the data block is stored, the *findNodesResponsibles* function is used. Immediately after, the *fetchBlockData* function tries to send a *SGetKeyMetaDataMessage* to one responding node in each one of the different groups in order to recover the *SKeyMetaData* object which contains the *LLocation* objects of the responsible nodes in this group where the data block is stored. In this manner, we do a parallel download of the data block from all the responsible nodes in all responsible groups where block data is stored. A data block can be recovered always that it exists in at least one node. An example can be observed at figure 7.3.

Example: N15 of G80 fetches block data with id 3



-Figure 7.3- Fetching a data block

7.4. Reorganization of data block due a overlay network changes

When a node joins or leaves a group, all the others nodes of the group receive an `LNotifyArrivalMessage` or an `LNotifyLeaveMessage`. The Look-up and Routing layer notifies this event to the Storage Block layer through the `nodeChangedEvent` method. This method manages the changes. Always a data block must be replicate into a responsible group in a number of `Parameter.REPLICATION_FACTOR` responsible nodes. We have to distinguish between the cases when a new node leaves the group and when a node joins the group.

7.4.1. A node leaves the group

All the nodes have to check their `RNKeyMetaData` hash. This hash maps the `SKeyMetaData` objects to the keys for which this node is responsible. For every key this node is responsible for, this last has to check if the node which has left the group was a responsible node. If it was, the node eliminates the node which leaves the group from the `storageNodes` list attribute of the `SKeyMetaData` object for the data block and executes the `findKeyInGroup` method of the Look-up and Routing layer in order to find the new responsible node to store the data unit. When the new node is found, it sends to it an `SStoreBlockMessage`. The new

responsible node makes a parallel download from the rest of nodes that contains the data in the group. After, it creates a new *SKeyMetaData* object for this data block and stores it in its *RNKeyMetaData* hash structure.

7.4.2. A node joins the group

All the nodes have to calculate the hash value of the new node with every one of the keys for which it is a responsible node. This is done by the Block Storage layer method *getHashForThisKey*. If the new node has a hash value bigger than an old responsible node, it becomes responsible to store this key. Every responsible node updates their *SKeyMetaData* object for this data block and sends an *SStoreBlockMessage* to the new node. When the new node receives this message, it makes the parallel download of the block data from the others responsible nodes for the data block in the group and creates an *SKeyMetaData* object for this data block, which is stored it in its *RNKeyMetaData* hash structure. There is a node that is no more responsible for this data block. This node deletes this data block through *removeBlockDataEvent* method and deletes the *SKeyMetaData* object for this data unit from its *RNKeyMetaData* hash structure.

8. Application layer implementation

The Application layer is the topmost layer interfacing with the user. This layer implements the P2P application. A node in this layer offers specially two possibilities: store a file or retrieve a file. The functionalities of a node in this layer are implemented by the *ANode* class which inherits from the *SNode* class. The method that implements the *main* method is named *NodeTest*. The framework is fully parameterizable by the Application layer's *Parameter* class. In this class all parameters used for the different layers can be configured and adjusted to the needs of the application using the framework. This layer offers also a graphical user interface (GUI) that is implemented by a unique Java class *JFNodes*. The user can create several nodes over the same machine. This GUI gives the functionality of creating and deleting several nodes on the machine where it is running. The nodes are created from a text file that must be passed to the application as input. It gives also the functionality of storing and retrieving a file by the selected local node and it has to provide the overwritten implementations for the block events methods of the Storing Block layer.

8.1. Batch nodes file

The P2P storing application needs a text file with the list of nodes to be created as input. Each line of this file corresponds to a single node. An example of this file can be observed at figure 7.4. It is very important to separate each one of the parameters by a semi-colon character. The parameters of any of the lines of this file are:

- **IDGroup**: This is the identifier of the group the node belongs to.
- **IDNode**: This is the node identifier of this node into the group.
- **BW**: This is the bandwidth quantification factor of the node.
- **CPU**: This is the CPU process quantification factor of this node.

NodesFile

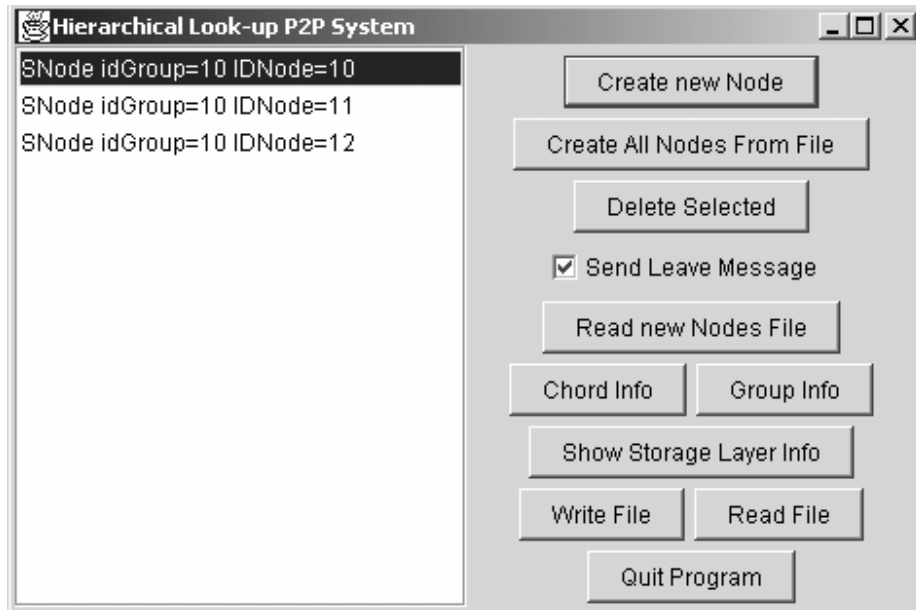
```
IDGroup=10;IDNode=10;BW=5;CPU=1;
IDGroup=10;IDNode=11;BW=5;CPU=2;
IDGroup=10;IDNode=12;BW=5;CPU=3;
IDGroup=10;IDNode=13;BW=5;CPU=4;
IDGroup=10;IDNode=14;BW=5;CPU=5;
IDGroup=10;IDNode=15;BW=5;CPU=6;
```

-Figure 8.1 – A batch nodes file example

8.2. Graphical user interface (GUI)

We can observe the GUI appearance at figure 7.5. We have to specify the batch nodes file at the command line. The user interface of this prototype consists on a list with all the nodes running on the local host and some buttons which perform operations on the selected virtual node when the application starts, the first node of the batch nodes file is created. The available options are:

- **Create new Node:** This button creates the next node of the batch nodes file. Every node needs know an existing node to join the system, so we have to select an existing node from the list as join system node.
- **Create All Nodes From File:** This button creates automatically all the nodes specified in the batch nodes file. The nodes are created every $2 * \text{Parameter. STABILIZE_INTERVAL_MS}$ (ms).
- **Delete Selected:** This button eliminates the selected node from the list of existing nodes.
- **Send Leave Message:** When this option is selected, the delete node sends an *LNotifyLeaveMessage* to all the other nodes of its group to warn them of its leave. If it is not selected, this message is not send. This situation recreates the sudden failure of a node.
- **Read new Nodes File:** This button permits to read another batch nodes file. All the current nodes are eliminated automatically. This button allows reset the program without stopping it.
- **Chord Info:** This button shows the Chord ring information of the selected node. Only the super peers have this information.
- **Group Info:** This button shows the intra-group information of the selected node.
- **Show Storage Layer Info:** This button shows the data blocks stored at this moment for the selected node.
- **Write File:** The user can select a file from a file dialog which gets stored on the network using the selected node.
- **Read File:** The user has to select a node which is used to retrieve the requested file and then an “open file” dialog appears where the directory and the requested filename can be specified.
- **Quit Program:** This button stops the application execution and removes from the hard disk all the directories created by the nodes.

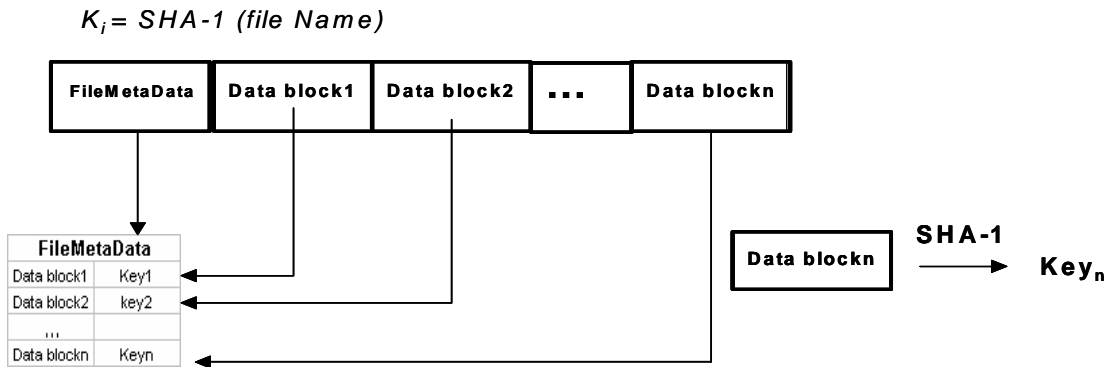


-Figure 8.2 – Graphical user interface (GUI) appearance

8.3. Storing and fetching a file

The Application layer uses the underlying Block Storage layer to store and retrieve files. The *ANode* class offers the *writeFile* which stores the selected file in the system. Files are split into blocks. The block size is defined by the *FILE BLOCK SIZE* constant of the *Parameter* class. First, a *FileMetaData* class object is created, shown in figure 7.6. This object contains the file's size, a fingerprint of the file content computed using the code snippet from Section 6.1 and an array of the block's *LHashID* keys generated by hashing the block binary data. Then each file block is stored with the *storeBlockData* method offers by the Block Storage layer. After all file blocks are successfully stored, the *FileMetaData* object itself is serialized and stored as a block with the hashed filename as block key.

A file is fetched by the inverse process implemented in the *readFile* method. First, the *FileMetaData* object is retrieved by hashing the filename provided by the user and the object is de-serialized. With the help of the *FileMetaData* object the blocks are fetched using the *fetchBlockData* method offers by the Block Storage layer and reassembled in the right order. Finally, a fingerprint of the reassembled file is compared to the fingerprint stored in the *FileMetaData* object. If they are not exactly the same, an exception is raised.



-Figure 8.3- Files Splitting into Blocks

8.4. Block event methods

The *ANode* class overwrites the *readBlockDataEvent*, *testBlockDataEvent*, *writeBlockDataEvent* and *removeBlockDataEvent* methods. These methods permit doing physical changes over the physical file system of the machine where the P2P application is running. The block event methods provide an interface to the physical storage device for the logical block management of the Block Storage layer. For this prototype, the hard disks on Internet hosts are used as storage devices. When a *ANode* class object is created, a disk quota *dataDirQuota* for that node and a data directory is specified. Each group creates its own directory in the data directory path named by its group *id* and each one of the nodes of this group, creates its own directory into the group directory named by its own node *id*. These directories are created under a temporal directory passed to the program as input at the command line. Blocks are stored in this directory named by their block id and the ".data" extension. In this way, nobody can know which files are stored at each physical machine. Every time a block is written or deleted the node increases or decreases its *dataDirUsed* variable by the block's size to keep track of the disk space used. If the block size plus the *dataDirUsed* exceeds the *dataDirQuota* limit, the *writeBlockEvent* method returns a failure.

9. Conclusion and Future Work

A prototype of a reliable storage system based on peer nodes has been designed and implemented in Java. The peers are organized in a two-level hierarchical peer-to-peer system. The peers are arranged in groups. The groups are organized in a top-level Chord overlay network. Inside each group, a CARP overlay network is used for arranging the peers in a lower-level overlay network. This peer-to-peer system provides a two-step look-up and routing service to the storage system. The goal of scalability in term of the number of nodes was achieved by implementing this hierarchical peer-to-peer system. Reliable storage is achieved by content replication managed by a Block Storage layer which uses the native events of the overlay network for the reattribution of dependencies among the peers. A parallel download content mechanism has been implemented in order to improve the download rate. The application itself is a thin layer on top of a powerful and versatile framework architecture. This framework could be the base for many other Java P2P applications, not only reliable distributed storage applications, because our framework provides solutions to many standard design problems in the field of P2P.

The framework nature of the implementation makes future improvements easy, because mechanisms and not implementations are specified by the framework layers and therefore implementations are exchangeable. It should be interesting to provide a Message layer's method to send messages by UDP to avoid the overhead of short lived TCP connections.

It should also be interesting that the Look-up and routing layer provides a method intended to be overwritten by the upper layers in order to pass them events about group leaves and joins. In this particular application, such a method could be useful to reorganize the data blocks among groups.

Caching mechanism inside the groups could be implemented. When a peer of the group recovers any block data, it could redistribute this block data among the group responsible peers to take advantage of low intra-group latency. Another improvement could be to consider up-time in the merit factor to promote a peer to super peer. The paraloader's efficiency could be further increased by dynamically adjusting the segment size to the block size and the number of download locations.

It could be very useful to implement an Application layer protocol to interact with the application through this protocol commands. Actually, it's just possible to interact with the program through the GUI. The last area of study could be the definition of a "physical proximity" metric among nodes in order to arrange them into groups. Security mechanisms to be aware from malicious participants have not been treated in this work.

10. Bibliography

- [1] Napster
<http://www.napster.com/>
- [2] Gnutella
<http://www.gnutella.com/>
- [3] C. Shirky, “What is P2P And What Isnt!”, Nov 2000.
- [4] Freenet
<http://freenetproject.org/cgi-bin/twiki/view/Main/WebHome>
- [5] J. Ritter, “Why Gnutella Can’t Scale”, unknown, Mar 2000.
- [6] Kazaa
<http://www.kazaa.com/us/index.php>
- [7] S. Han, B. Hore, I. Issenin, S. McCarthy, and S. Tauro, “HollyShare: Peer-to-Peer File Sharing Application”, , Information and Computer Science, University of California, Irvine, 2001.
- [8] A. Rowstron and P. Druschel, "*Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*". IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, pages 329-350, November, 2001.
- [9] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph, “Tapestry: An infrastructure for fault-tolerant wide-area location and routing,” Tech. Rep. UCB/CSD-01-1141, Computer Science Division, University of California, Berkeley, Apr 2001.
- [10] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, “A scalable content-addressable network,” in *Proc. ACM SIGCOMM*, 2001.
- [11] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” in *Proc. ACM SIGCOMM*, 2001.
- [12] K. W. Ross, “Hash-routing for collections of shared web caches,” *IEEE Network Magazine*, vol. 11, 7, pp. 37–44, Nov-Dec 1997.
- [13] P. Druschel and A. Rowstron, “PAST: A large-scale, persistent peer-to-peer storage utility”, In *Proc. HotOS VIII*, may 2001.
- [14] F. Dabek, “A Cooperative File System”, M.S. Thesis, Sep 2001.

[15] *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*, El Paso, Texas, May 1997, ACM Symposium on Theory of Computing.

[16] L. Garces-Erice, K. W. Ross, G. Urvoy-Keller, and E. W. Biersack, “Hierarchical P2P look-up services,” tech. rep., Institut Eurecom, July 2002.

[17] M. Prestwood, “An Introduction to Object Orientation”, Jan 2002.

[18] P. Rodriguez, A. Kirpal, and E. W. Biersack, “Parallel Access for Mirror Sites in the Internet”, Infocom. Tel-Aviv, Israel, March 2000.