Chord and Symphony:
An Examination of Distributed Hash Tables and Extension of PlanetSim

Monica Haladyna Braunisch

A Thesis in the Field of Information Technology

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

June 2006

© 2006 Monica Haladyna Braunisch

All Rights Reserved

Abstract

In an expository approach, I analyze two peer-to-peer network overlay protocols, namely, Chord and Symphony. I have extended an existing network simulator, PlanetSim v.3.0, in the areas of secure hashing and under consideration of Internet Protocol Version 6 (IPv6). I use this extended PlanetSim code to simulate and compare both overlay protocols in a distributed hash table (DHT) application.

In the introduction, I provide general background information on the origins of DHTs and a brief overview of Chord and Symphony. I then proceed to an in-depth analysis of the two protocols, based on the most recent literature in this active field of research. In order to evaluate the Chord and Symphony protocols in their overall performance, I have selected PlanetSim, a new Java-based simulator designed by P. Garcia *et al.*, as the simulation platform. In a brief survey, I compare PlanetSim to other peer-to-peer (P2P) network simulators. Taking advantage of PlanetSim's extendibility and modularity, I proceed to implement and to test hashing with Chord and Symphony, using the 2nd generation Secure Hash Standard algorithms (SHA-2). I also demonstrate hashing of Internet Protocol Version 6 (IPv6) addresses, as stipulated in Request for Comments (RFC) 4291 of February 2006, by using SHA-2. The simulation parameters considered include network size, node distribution type, Chord ID bit length, and number of long distance links in Symphony.

Dedication

In memory of my mother Sylvia who always believed in the highest possible ideals: goodness, honesty, truth, humility, humanity, hope, and unconditional love.

Mama, I will love you forever, through all time, space and eternity.

To my loving and devoted husband Henning, for his support through these past months, and also for his kindness and love.

To my adorable and precious Isabel: You are the light of my life!

To my wonderful brother who always believes that there are no impossibilities, and to my Dad for his constant propagation of confidence and love.

Acknowledgments

I would like to thank my thesis director Scott Bradner for all his input, suggestions, answering all my questions and concerns and by shedding light on diverse subject matters when circumstances looked dim. Also thanks to my thesis advisor Professor Bill Robinson for taking the time to thoroughly read the thesis and make numerous and helpful suggestions to improve the thesis.

I would like to acknowledge and thank Professor Pedro García López from the Universitat Rovira i Virgili in Tarragona, Catalonia, Spain, for his kind invitation to participate with his team of scientists on the extension of the PlanetSim code (see planet.urv.es/planetsim/ on the Internet).

Also, special thanks to my husband Henning for helping explain various probability concepts, making suggestions that helped shape and improve the quality of the thesis material, as well as for testing code and helping find bugs when they were no longer visible. Special thanks to my father, Professor Ronald Haladyna, for taking the time to read the thesis and make suggestions to correct some grammatical errors and other important suggestions to improve the overall style.

Table of Contents

Dedication iv
Acknowledgments v
Table of Contents vi
List of Figures ix
List of Tables xii
Chapter 1 Introduction 1
1.1 Brief History of the Origins of P2Ps 3
1.2 Hash Tables 5
1.3 Collision Resolution Schemes 7
1.4 Distributed Hash Tables 9
1.5 From SHA-1 to SHA-2 14
1.6 Chord and Symphony Overview 15
Chapter 2 Chord 20
2.1 Overview of the Chord Protocol 20
2.2 System Model 21
2.3 Structure of the Chord Ring 24
2.4 Consistent Hashing 27
2.4.1 Creation of Node IDs 29
2.4.2 Creation of Key IDs 29
2.4.3 Mapping Node IDs and Key IDs to the Chord Ring 30
2.5 Finger Tables 31

2.6 Stabilization: Joins and Departures 33
2.6.1 Node Joins 33
2.6.2 Handling Node Failures and Departures with High Fault Tolerance 38
2.6.3 Lookups during Stabilization 40
2.7 Load Balance with High Probability 40
2.8 Effects of Churn: Rapid Node Join and Departures 42
2.9 Initial Conclusions 43
Chapter 3 Symphony 47
3.1 Models of the Small World Phenomenon 47
3.2 System Model 50
3.2.1 Load Balance 52
3.2.2 Low State Maintenance 53
3.2.3 Fault Tolerance 53
3.2.4 Flexibility 54
3.2.5 Latency vs. Outdegree 55
3.3 ID Distribution 56
3.3.1 Probability Density Function for Long-Distance Links 56
3.4 Regular Distribution of IDs 60
3.5 Random Distribution of IDs 65
3.5.1 Estimation Protocol 66
3.5.2 Link Establishment 67
3.5.3 Smooth Re-Linking Protocol 67
3.5.4 Graphical Examples 68
3.6 Balanced Distribution of IDs 71
3.7 Node Joins 72
3.8 Routing Table Structure 73

3.9 Node Departures 75

3.10 1-Lookahead and Greedy Routing 77

3.11 Initial Conclusions 79

Chapter 4 PlanetSim 81

4.1 Comparison of P2P Simulators 81

4.2 PlanetSim 85

4.3 SHA-1 & SHA-2 Families 87

4.4 Hashing with IPv6 90

Chapter 5 Results, Conclusions and Future Directions 96

5.1 Results Obtained with Simulations 96

5.2 Hashing Examples 107

5.3 Conclusions and Future Directions 112

Glossary 115

References 117

Appendix 1 Tabulated PlanetSim Results 123

Appendix 2 Application Code and Extensions 125

List of Figures

Figure 1.1 Distributed nodes on the Internet 10

Figure 1.2 Taxonomy of P2P systems showing Chord and Symphony............................ 16

Figure 1.3 Example of the Chord address space.. 18

Figure 1.4 Example of a Symphony network 19

Figure 2.1 Diagram of Chord nodes and keys on the ring 25

Figure 2.2 Creation and distribution of keys in Chord ring 30

Figure 2.3 Example of Chord ring and finger table 33

Figure 2.4 Node n_x wants to join the Chord ring 34

Figure 2.5 Node n_x = N25 finds its place in the Chord Ring 36

Figure 2.6 Node 40 leaves network abruptly 39

Figure 3.1 Simple example of a Symphony network with short reinforced links and one long distance link per node (original design by Manku *et al.*, 2003) 51

Figure 3.2 Harmonic probability density function showing 100 to 100,000 nodes.......... 57

Figure 3.3 Histogram for 1000 long-distance links in a network of 1000 nodes.............. 58

Figure 3.4 Symphony network with 10 nodes and a maximum of one long-distance link per node.. 62

Figure 3.5 Symphony network with 10 nodes and a maximum of two long-distance links per node.. 63

Figure 3.6 Network with 1,000 nodes with regular distribution and a maximum of one long-distance link per node 64

Figure 3.7 Symphony network with 1,000 nodes and a maximum of two long-distance links per node 65

Figure 3.8 Symphony's estimation protocol 66

Figure 3.9 Symphony network with 10 nodes randomly distributed and a maximum of one long-distance link per node 69

Figure 3.10 Ten node Symphony network with a maximum of two long-distance links per node 69

Figure 3.11 Symphony network with 1,000 nodes and a maximum of one long-distance link per node 70

Figure 3.12 Symphony network with 100 nodes and a maximum of two long-distance links per node 71

Figure 3.13 Example of a Symphony routing table 73

Figure 5.1 Chord and Symphony for increasing network size. Network creation with regular distribution of nodes. 97

Figure 5.2 Chord and Symphony for increasing network size. Key insertions with regular distribution of nodes. 98

Figure 5.3 Chord and Symphony for increasing network size. Key lookups with regular distribution of nodes. 98

Figure 5.4 Chord and Symphony for increasing network size. Network creation with random distribution of nodes. 99

Figure 5.5 Chord and Symphony for increasing network size. Key insertions with random distribution of nodes. 100

Figure 5.6 Chord and Symphony for increasing network size. Key lookups with random distribution of nodes. 100

Figure 5.7 Chord and Symphony with use of different hash algorithms. Network creation with regular distribution of 1,000 nodes. 102

Figure 5.8 Chord and Symphony with use of different hash algorithms. Key insertions with regular distribution of 1,000 nodes. 102

Figure 5.9 Chord and Symphony with use of different hash algorithms. Key lookups with regular distribution of 1,000 nodes. 103

Figure 5.10 Chord with varying bit length of IDs. Network creation with regular distribution of 1,000 nodes. 104

Figure 5.11 Chord with varying bit length of IDs. Key insertions with regular distribution of 1,000 nodes. 104

Figure 5.12 Chord with varying bit length of IDs. Key lookups with regular distribution of 1,000 nodes. 105

Figure 5.13 Symphony with varying maximum number of long distance links. Network creation with regular distribution of 8,000 nodes. 106

Figure 5.14 Symphony with varying maximum number of long distance links. Network creation with regular distribution of 8,000 nodes. 106

Figure 5.15 Symphony with varying maximum number of long distance links. Network creation with regular distribution of 8,000 nodes. 107

List of Tables

Table 1.1 Chaining example showing how both *j* and *k* hash to bucket 15. Subsequently *k* is moved to bucket number 16 with a reference in the linked list. 8

Table 1.2 Changes from hash tables to distributed hash tables 11

Table 1.3 Diverse topologies: Protocols using different routing mechanisms (Manku, 2004) 17

Table 2.1 Crucial differences between Chord and DNS.. 26

Table 2.2 Finger tables for node 20 and node 30... 37

Table 2.3 Addition of Node 25 to the Chord address space 37

Table 3.1 Components needed for routing to take place in Figure 3.13........................... 74

Table 4.1 Different SHA algorithms... 88

Table 4.2 IPv6 header and fields... 91

Table 4.3 IP address space from RFC 1884 showing the initial distribution of IP addresses 94

Table 5.1 Node “c” has a heavy load of 15 and a target of only 12 113

Table 5.2 Node “f” has a load equivalent to 5 and a target of 10 113

Table A1.1 Tabulated Chord simulation results obtained with PlanetSim..................... 113

Table A1.2 Tabulated Symphony simulation results obtained with PlanetSim............. 123

Chapter 1 Introduction

"Where a new invention promises
to be useful, it ought to be tried."
Thomas Jefferson, 1786

This chapter presents an overview of the thesis project including background information on the history of how peer-to-peer (P2P) systems originated, hash tables, distributed hash tables (DHTs), and the growth of P2P protocols, and a preliminary introduction to the Chord and Symphony routing protocols, which is at the core of this thesis.

Distributed hash tables are self-organizing, logical overlay networks that run on top of physical networks and offer services to map, lookup and delete hash keys in distributed P2P systems. The thesis focuses on a comparative study of Chord and Symphony and how they differ in their routing and lookup procedures. Even though they both resemble each other in their common ring infrastructure, they are dissimilar in their routing techniques (Stoica *et al.*, 2001; Manku, Bawa & Raghavan, 2003).

To better understand and analyze both protocols, simulations were performed using PlanetSim (Garcia *et al.*, 2005), a comprehensive Java-based simulator (see Chapter 4 for details).

Additionally, extensions to the PlanetSim simulation code (see Chapter 5 and Appendix 2) were developed to implement and test use of the SHA-2 family (SHA stands for secure hash standard algorithm), which without question produce longer message digests and also increase the computing time as compared to the conventional, but more insecure, SHA-1. Furthermore, a variation of hashing with Internet Protocol Version 6 (IPv6) was created to demonstrate how the transition from Internet Protocol Version 4 (IPv4) is possible (Hinden & Deering, 2006), a necessity in the near future.

The thesis is organized as follows:

1. The rest of this Chapter covers background information about the origins of hash tables and distributed hash tables. Additionally, other key concepts are examined such as the use of Secure Hash Algorithms and the role they partake in DHTs. Also, a brief introduction to Chord and Symphony is covered.

2. The subsequent parts of the thesis (Chapters 2 and 3) contain an analysis of Chord and Symphony.

    - Evaluating their scalability as well as assessing how fault tolerant they are in quickly shrinking and expanding networks.

    - Examining the resiliency of Chord and Symphony and showing how both protocols are able to continue operating efficiently under any amount of link failure or even massive link failures.

    - Examining how low state maintenance helps reduce the latency for both protocols and explaining why its beneficial to reduce the Round Trip Time.

- The way that dilation or stretch affect both protocols and why it is desirable for them to maintain this in a minimal state.
- Examining different forms of probability density functions that enhance load balance for both protocols and why both use different functions.
- Analyzing the infrastructure of both protocols and showing how both are able to adapt to any large-scale distributed network.

3. The last part tests Chord and Symphony with PlanetSim (Garcia *et al.*, 2005) and assesses the extended features that were implemented for the project.
    - The integration of SHA-512, among others, from the SHA-2 family, which produce longer message digests.
    - Hashing with IPv6. At some point in the near future, IPv6 needs to be integrated in real-world P2P scenarios. In this thesis, there are examples of how this can be implemented as of today (RFC 4291).
    - Plots of simulations using PlanetSim / Jacksum and tests for all the digest algorithms from the SHA-2 family.

1.1 Brief History of the Origins of P2Ps

The concept of Peer-to-Peer networks, known as P2P, is actually not novel, but has been around for a few decades. It began with the formation of the first wide area network (WAN) in 1965. Engineers established a connection by way of a phone line between the Massachusetts Institute of Technology (MIT) and the University of

California at Berkeley (ARPANET History, 2004). This propelled the creation of the ARPANET (Oram & O'Reilly & Associates, 2001; Wikipedia, 2004; Moore & Hebeler, 2002), which was sponsored by an agency of the Department of Defense and designed as a packet-switching network. The first interface message processor (IMP) messages to be sent through ARPANET originated from Stanford University to the University of California at Los Angeles (UCLA) on October 29, 1969. In less than two months, several academic institutions had joined the ARPANET: Stanford, UCLA, University of California at Santa Barbara and University of Utah. This junction formed the first-four node network in history, thus the first P2P network. The rationale behind this new structure was simple: To enable researchers to connect directly to each other, without a central authority, exchange information and share resources in a secure and distributed environment. This innovative development was followed by Usenet (Truscott & Ellis, 1979) and domain name service (DNS), which further helped characterize modern P2P structures. Unfortunately, the original Internet architecture did not survive as much as it might have and too often evolved into the client/server model, which has dominated the Internet until recent years.

It was not until late 1999 with the release of Napster (Fanning, 1999) that a relevant transformation took place in the structure of modern P2P networks. Napster, however, was not a fully decentralized P2P network because it utilized a central server to retain pointers and to resolve addresses. On the other hand, when Gnutella (Frankel & Pepper, 2000) was dispatched a significant change took place: Gnutella did not employ any central servers, but only directory servers that passed network addresses of peers to

other peers, and thus making it the first contemporary, decentralized and distributed network for file sharing (Wikipedia, 2004; Oram & O'Reilly & Associates, 2001).

Regardless of some limitations that Napster and Gnutella presented, these two technologies repeatedly demonstrated remarkable success by showing how they could scale to millions of users in a brief period of time. In the succeeding five years, numerous P2P networks and algorithms were designed and introduced to the Internet (Liben-Nowell, Balakrishnan & Karger, 2002b; Zhao *et al.*, 2001; Risson & Moors, 2004).

1.2 Hash Tables

A brief look at the evolution of hash tables is useful in order to understand how DHTs originated. Future generations of p2p networks may at some point depend on distributed hash tables as a directory service.

In early 1953, a mathematician, Hans P. Luhn, who at the time was working for International Business Machines (IBM) proposed in a memorandum to try a new mathematical function that would be able to fabricate uniform random variables. The notion behind this was to be able to retrieve data faster without having the computer read every line of text, but instead granting access to a subscript in a linear array, which later evolved into the notion of chaining. With this chaining, Luhn suggested using buckets or a virtual space in the array that held more than one element in order to perform external searches.

Although Luhn coined the term "hash," it was not documented until Arnold I. Dumey described the first primitive types of hashing techniques in his book, "Computers

and Automation" (Dumey, 1956). In 1956, Dumey took this a step further when he proposed that the key and value pairs should be divided by a prime number and the remainder be used as the hash address (Chern, 2005).

Through the course of the years, hashing became a standard tool used in dictionary searches, assembly programs, and for parsing, and was widely used for compiler languages (Allen, 1981). Hash tables have been used over the last couple of decades and evolved from simple heuristic methods into more elegant and complex data structures that aid in the rapid search, insertion and access to large amounts of information without excessive overhead (Wikipedia, 2005; Brookshear, 2000).

Hash tables are composed of associative arrays, which are known and referred to as look-up tables, abstract data types, maps or dictionaries. These tables are in turn used to relate a piece of information, known as a value, to a key. In order to be able to map (bind) a value (data or information) to a key and create the relation between them another element comes into place, called a hash function. A common or generic hash function takes the value and acquires the key with a modulo operation using the length of the table.

Once the key is produced, it is indexed or mapped as an entry in the array space, or bucket, where this input then contains documentation that is associated with the key. With a good hash algorithm, it can be assumed that the keys are distributed evenly and at random, although this is not always the case.

Taking the word "can", here represented by x and subsequently applying a generic hash function h to x, produces an integer $h(x)$, such that:

$$0 \le h(x) \le n-1$$

Here n is the number of available spaces in the hash table. Supposing n = 97 buckets, a sequence of steps explained below could follow.

Step 1. Convert the key field value “can” into a sequence of decimal values using an American Standard Code for Information Interchange (ASCII) table, resulting in:

99, 97, 110.

Step 2. Add the values, resulting in 306.

Step 3. Calculate 306 mod 97, giving the remainder 15.

Step 4. Assign “can” to bucket number 15.

The example shown above is truly a primitive hash table. It would tend to avoid collisions due to the fact that the table is of a fixed sized integer which is also prime (Krowne, 2005; Wikipedia, 2005). However, if the table is of an unknown size then it becomes harder to predict where the keys are deposited, if a good hash function is not used. In that case, clustering or collisions may occur.

1.3 Collision Resolution Schemes

Collisions take place when key-value pairs are assigned to the same hash bucket. There are several collision resolution schemes that can be applied to aid sorting the keys to distinct buckets. One is known as “chaining”, which employs linked lists, and the other is “open addressing”, which uses probing techniques.

Chaining is simple since it assigns a linked list to each bucket in the array (Wikipedia, 2005; Chern, 2001). The chaining algorithm performs its routings by searching through all the pointers until it finds the correct bucket. This makes it slower and more memory intensive than other collision resolution schemes. A visual example of chaining is shown in Table 1.1 where keys j and k hash to the same bucket, number 15, and due to the linked list approach k is then moved and referenced in bucket number 16.

Keys		Indexes	Key-Value Pairs
		0	
.		1	
.		2	
i		.	
		.	
.	$h(j)$ $h(k)$	15	j is inserted in bucket 15 and k is moved to bucket 16
.		16	k
j		.	
.		.	
.		23	
	$h(i)$	24	i
k		25	
.		.	
.		.	
		$n-1$	

Table 1.1 Chaining example showing how both *j* and *k* hash to bucket 15. Subsequently *k* is moved to bucket number 16 with a reference in the linked list.

Another collision resolution method is known as "open addressing" also known as "linear probing". Here the data are stored directly into the buckets and no longer as reference pointers. With this method, if a bucket is already full the algorithm simply

moves down the table until it finds an empty space to insert the key. It can also be set with predefined measurements such as jumping over a set number of buckets down the table and then inserting the key/pair in that slot. One of the disadvantages of open addressing is when the buckets are reaching maximum capacity, clustering begins dominating the hash table and the number of probes needed to assign or find a key drastically increases. When the hash table becomes entirely occupied, some of these probing methods no longer work and continue to run endlessly.

1.4 Distributed Hash Tables

At the core, DHTs are a derivative of hash tables. There have been notable shifts from the primitive hash table arrays as examined in Section 1.3 to the new and sophisticated DHT structures. Figure 1.1 displays geographically dispersed nodes.

The principal idea behind DHTs is the same as for hash tables: to be able to insert, and delete large amounts of information associated with keys. The concept of "bucket" is now replaced by a physical node, which could be located anywhere globally. Forming an overlay network using DHTs appears similar to the way standard hash tables are structured:

- Values are produced and associated with keys.
- Keys are hashed; in many cases with the node's port number and the node's IP address (Zhao *et al.*, 2003).

- The hashed keys are mapped (routed/bound) to the nodes in a balanced and procedural manner depending on the DHT algorithm (Chord, Symphony, Viceroy) that is exercised.
- The nodes hold the key and value pairs.

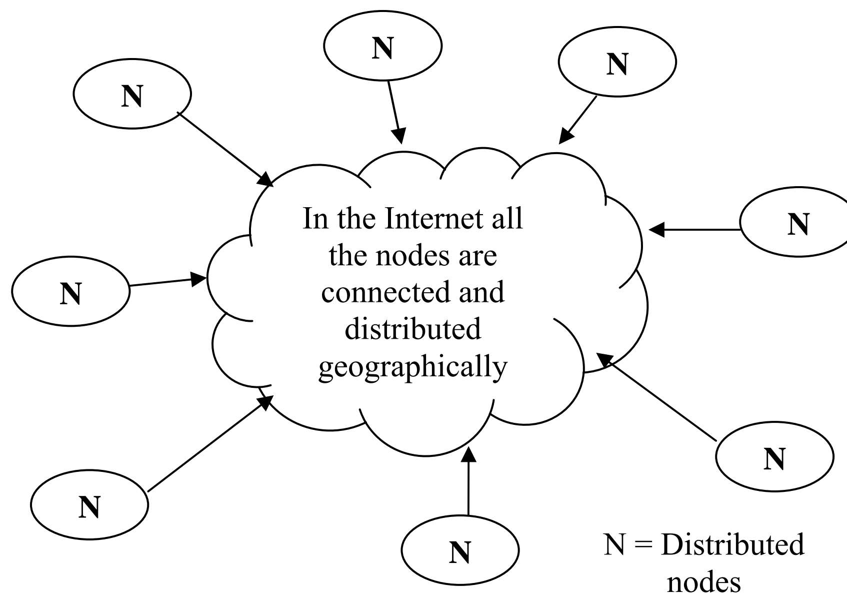


Figure 1.1 Distributed nodes on the Internet

One of the main problems with hash tables is that they are not flexible and cannot accommodate changes without disrupting the entire array. With these new network overlays composed of DHTs, hashing algorithms (for SHA-1 see Section 1.5), and routing protocols many advances came about. For example, hash tables have methods that enable distributing data with compressed digests (outputs with smaller ranges) but up until recently there existed no way to distribute in a balanced way the compressed digests.

Thus, collisions and overflows occur leading to loss of data. To better illustrate more of the changes from hash tables to DHTs, see Table 1.2.

Hash Tables		Distributed Hash Tables
Central server	→	No central server
Arrays representing hash tables	→	Distributed tables each
Constant time $O(1)$	→	Variations of logarithmic time
Buckets	→	Nodes
Collisions	→	Smart algorithms
Uneven distribution of key/value pairs	→	Balanced distribution of key/value pairs
Overflows and clustering	→	Balanced distribution, no clustering
Static	→	Dynamic, can scale to millions
Changes in hash table means rehashing entire table	→	Fault tolerant, can tolerate changes with continuous node arrival and departure
Linear	→	Distributes responsibilities to nodes

Table 1.2 Changes from hash tables to distributed hash tables

The concept of being able to distribute information throughout geographically distant systems and to assign nodes different responsibilities and obligations creates a distinct environment for global cooperation. By distributing and allocating resources in separate networks, it diminishes bottlenecks, eliminates lengthy searches through centralized indexes, and provides less overhead.

Structurally, DHTs consist mainly of two parts:

1. An overlay network which helps nodes in finding the key holders;
2. Keyspace partitioning; this divides and distributes rights over the keys (key ownership).

With keyspace partitioning, consistent hashing is typically employed to map the keys in the DHT. When a new node joins or departs, only the routing tables of the nodes that have links to each other are hashed. When primitive hash tables are used, any change requires the tables to be re-hashed and updated, taking up time and memory. Depending on the DHT topology, the keyspace is partitioned or divided into many sections and then keys are assigned to node managers in the divided space. These nodes are then addressed as endpoints. For instance, as shown in Chapter 2, the Chord protocol topology is a ring that advances with ID assignment in a clockwise direction. In consistent hashing, only a small fraction of the keys is reassigned to different locations, not affecting the entire system.

As for the term “overlay networks”, these are the logical networks that have nodes within an explicit topological space, and sustain links with other nodes in this same space.

The number of neighbors that each node has is important because in order to establish connections to other nodes in the network, it is desirable to keep the number of hops to the destination small. The number of neighbors per node also known as "degree" is favorable when low. The dilation, also known as stretch, is the maximum ratio of original distance for a pair of vertices and should be small.

Presently, DHTs are used in conjunction with new p2p technologies such as JXTA, Free Net, Chord, BitTorrent, and Oceanstore, to name a few. The progress established by these innovative distributed P2P networks is in the following areas:

- They are decentralized and autonomous, in that they do not rely or depend on any form of central servers or authority.
- They are self-organizing, because the search/lookup algorithms can re-structure the DHTs without the need of a central server.
- They have shifted from the semantic association in the indexing to a semantic-free index (Risson & Moors, 2004).
- They are robust, meaning that the P2P network can survive node failures and their constitution can recover without collapsing entire infrastructures.
- They are scalable.
- They are symmetrical with respect to distributed nodes by establishing a mode in which nodes are equivalent (Zhao *et al.*, 2003; Kubiatowicz, 2003; Manku, 2004).

- They are highly dynamic in that these networks can handle churn, constant node arrival and departure in a large-scale system.

1.5 From SHA-1 to SHA-2

Cryptographic hashes are used in DHTs in order to produce message digests that will associate data with a key. The first widely used cryptographic hash is SHA and first came out in 1993. Subsequently a stronger version, SHA-1, which is still broadly used today, was released two years later (SHA-1, 1995). The National Security Agency (NSA) created this standard as a secure hashing algorithm. SHA-1 is implemented in areas such as public key cryptography, digital signatures and hashing to protect confidential data, for authentication and to preserve integrity.

For example in networking, SHA-1 may take a node's IP address as a "message" and produce a 160-bit digest. The digest is similar to a checksum and equivalent to a fingerprint of the original message, making it hard to guess or deduct the message from the digest. In addition, the digest algorithms are designed to prevent collisions (Baldwin, 2005).

Recently, there has been concern with regard to security with SHA-1. According to experts at the Crypto conference in August 2005, a team of scientists, Wang, Yin, and Yu from the University of Shandong, China, stated (Schneier, 2005): *"We have developed a set of new techniques that are very effective for searching collisions in SHA1. Our analysis shows that collisions of SHA1 can be found with complexity less than* 2^{69} *hash operations. This is the first attack on the full 80-step SHA1 with complexity less than*

the 2^{80} theoretical bound. Based on our estimation, we expect that real collisions of SHA1 reduced to 70-steps can be found using today's supercomputers." (Wang, Yin & Yu, 2005)

Searches for more secure hashing structures are well on their way. Some experts (Crypto, 2005) are claiming that new hashing algorithms need to be designed or advise to begin using the more secure SHA-2 family. In this thesis project, SHA-2 has been introduced to the P2P simulator PlanetSim.

1.6 Chord and Symphony Overview

P2P systems are undergoing significant changes as the second generation P2Ps aim towards using structured algorithms. Now third generation P2Ps are beginning to emerge (see Figure 1.2), using new protocols and routing technologies (Risson & Moors, 2004; Wikipedia, 2004). Pre-2000 developments, such as skip lists, Plaxton trees, linear hashing, and other overlays are now considered outdated, since they were used in systems such as Napster and Gnutella and these first generation P2Ps used unstructured algorithms (Oram & O'Reilly & Associates, 2001; Risson & Moors, 2004). Two examples of this 3rd generation are Chord and Symphony.

The key view behind distributed peer-to-peer systems is that they are decentralized, autonomous, self-organizing, scalable, robust, and the nodes are symmetric in function (Zhao *et al.*, 2003; Kubiatowicz, 2003; Manku, 2004).

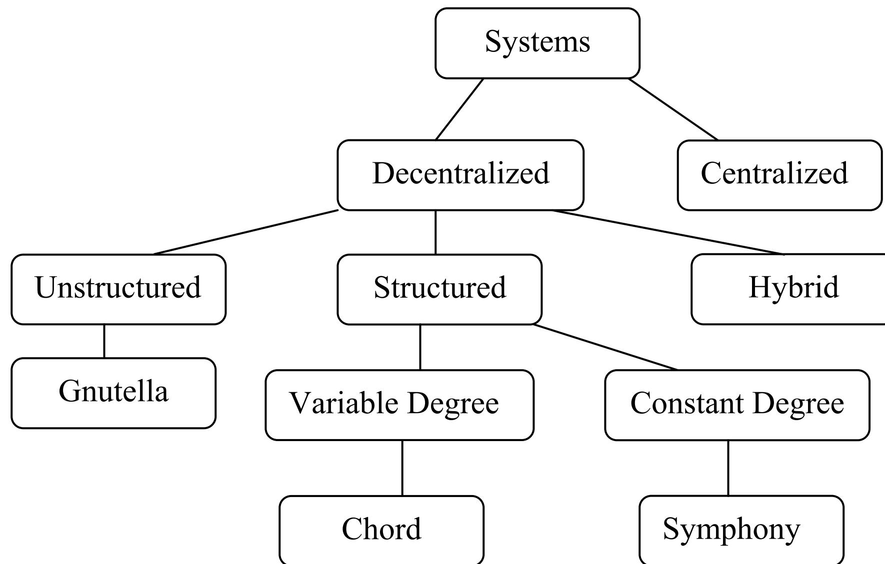


Figure 1.2 Taxonomy of P2P systems showing Chord and Symphony

The distribution of resources is an important goal since it enhances the performance of systems, and allows them to remain autonomous, as well as bridging geographical differences and lowering transaction costs (Liben-Nowell, Balakrishnan & Karger 2002a; Zhao *et al.*, 2001).

Due to the large number of P2P systems that have been developed in recent years, a wide variety of routing/lookup protocols have been produced such as Chord (Stoica *et al.*, 2001), Tapestry (Zhao *et al.*, 2004), Pastry (Rowstron & Druschel, 2001), Viceroy (Malkhi, Naor & Ratajczak, 2002), and Symphony (Manku, Bawa & Raghavan, 2003), to name a few (Risson, & Moors, 2004; Manku, 2004).

These new routing protocols are all innovative in the sense that they are self-governing, self-repairing, sustain low-maintenance states, and have all proved to scale relatively well while being robust.

Table 1.3 below is a chart that depicts several routing protocols and some of their properties such as the type of topology, the number of links to perform a lookup, or the latency.

Protocol	**Number of Links**	**Latency**	**Topology**
CAN	$O(\log n)$	$O(\log n)$	Deterministic
Chord	$O(\log n)$	$O(\log n)$	Deterministic
Pastry	$O(\log n)$	$O(\log n)$	Partially Randomized
Tapestry	$O(\log n)$	$O(\log n)$	Partially Randomized
Viceroy	$O(1)$	$O(\log n)$	Partially Randomized
Symphony	$2k+2$	$O((\log^2 n)/k)$	Randomized

Table 1.3 Diverse topologies: Protocols using different routing mechanisms (Manku, 2004)

The majority of this project focuses on the infrastructure of Chord (Stoica *et al.*, 2001) and Symphony (Manku, Bawa, & Raghavan, 2003). Even if both protocols resemble each other in their topological ring structure, they differ in their routing approach. Figures 1.3 and 1.4 below will help the reader obtain a clear idea of what the Chord address space looks like, as well as an example of a Symphony network.

Figure 1.3 is an example of a rudimentary Chord network that possesses eight nodes with the identifiers from zero to seven. Each node owns a 160-bit ID that was output when each node hashed its IP address and port number employing SHA-1.

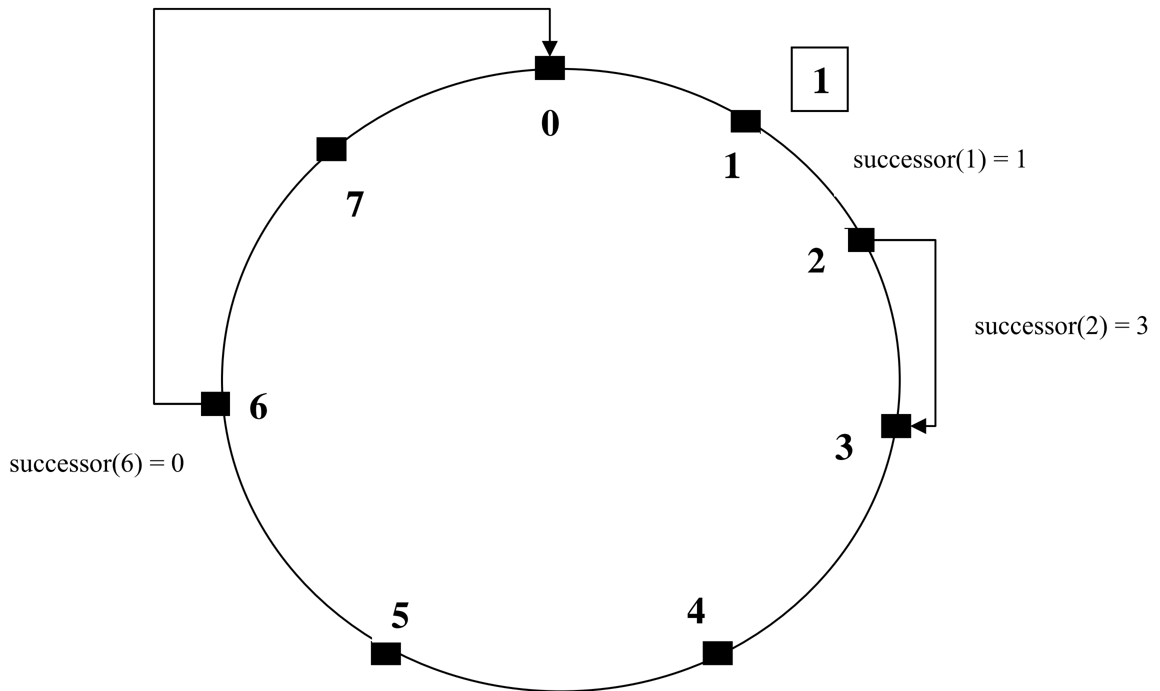


Figure 1.3 Example of the Chord address space

In a Symphony network (see Figure 1.4), the nodes are arranged in a ring with a perimeter of length equaling one. When a node arrives, in one Symphony variant, it chooses its position along the ring randomly and uniformly. Each node has one short link to the following node on the ring and *k* long links, which are randomly chosen long-distance links (Manku, 2003).

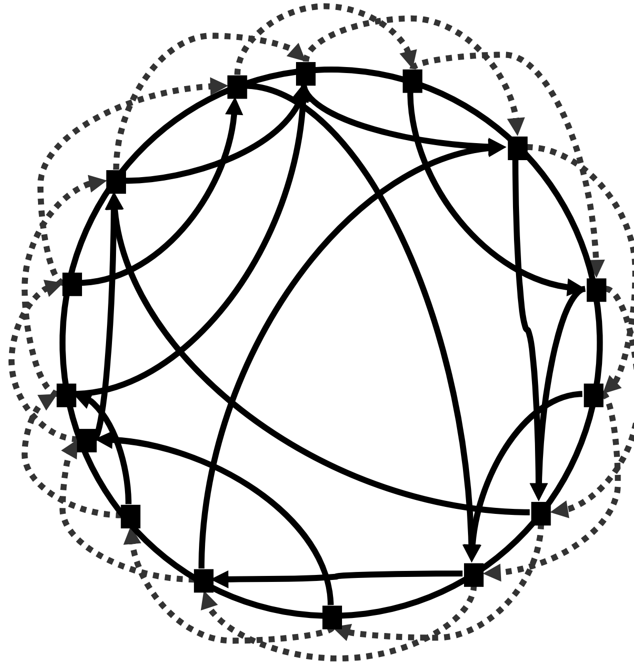


Figure 1.4 Example of a Symphony network

■	Nodes are arranged in a ring topology with the perimeter of length 1. The distribution of node joins in the ring space is uniform and random.
·······►	These links reinforce the short-distance links that each node has to the next node on the ring.
──►	Long-distance links. There are *k* of such links for each node.

Chapter 2 Chord

"Every action of our lives touches on some chord
that will vibrate in eternity."
Sean O'Casey, Irish dramatist (1880–1964)

This chapter describes the core of the Chord protocol starting with a brief history of its origins, and how a system prototype is represented with all of its attributes. Visual examples of the infrastructure of the Chord ring are given, how nodes join, stabilize, depart, and how the routing tables (finger tables) are updated. Chord uses a probability density function (PDF) that encompasses one central topic with its technique of load balancing with high probability that has been shown to work well under stresses such as churn. The last section describes some of the initial conclusions for this particular protocol as well as some of the drawbacks and other suggestions on how to improve the original version of Chord.

2.1 Overview of the Chord Protocol

The Chord protocol was designed as a DHT routing protocol by a group of students from the University of California at Berkeley and MIT (Stoica *et al.*, 2001). The designers of Chord proposed a very exciting new way to distribute data as well as perform lookups, assign node IDs and keys with many attractive features such as good

scalability, complete decentralization, efficient load balancing, and simplicity. At the core of the protocol, Chord maps (binds) a key to a particular node and with consistent hashing helps distribute keys to the managing nodes in a balanced and proficient mode.

2.2 System Model

The Chord protocol system model possesses certain traits that characterize it as a world-class DHT for distributed routing in large-scale P2P systems. These components distinguish Chord from past protocols (Stoica *et al.*, 2001; Dabek *et al.*, 2002). First of all, Chord is known for its simplicity in design and deployment. This point is crucial since Chord was developed in a way that can be easily understood and modified, and the code can be extended to accommodate a wide variety of P2P applications such as CFS, Cooperative File System (Dabek, 2001). There is a tendency that, when a protocol is overly focused on theoretical matter and has not been tested, it is not easily adopted by the general public and will consequently not be used (Manku, 2004). That is one of the reasons why Chord has been highly successful in this area, since its simple and elegant design complies with all the major structural aspirations of a wide area distributed P2P system, therefore attracting people from diverse networking areas.

Chord is also a protocol that meets the requirements for a P2P system in that it is geographically distributed in its entirety. This property manifests itself by having nodes spread out globally without the need of central servers, lists, or any type of authority that decides how the distributions of node IDs or routing schemes are handled. It is a completely self-governing, self-organizing entity (Liben-Nowell, Balakrishnan & Karger,

2002a). This complete decentralization is a powerful concept rendering each node in this conceptually colossal network an assignment to execute. This decentralization feature makes the network robust and resistant to massive failures (Stoica *et al.*, 2001) since, if a certain geographical area is suddenly jeopardized by any type of unexpected failure, the stretched out node distribution makes it possible for nodes to find successor nodes at different locations and update the finger tables in the Chord ring, thus keeping the system up and running. However, once the failed nodes (failed due to any number of reasons) decide that they want to rejoin the Chord ring, they will reinitiate the binding process and adhere themselves to the ring. By having all the nodes play independent roles and take management over a proportionate number of keys, an equalitarian system that is resilient and robust to node failures is created.

Load balance is another topic that is at the heart of routing protocols. Chord has been tested in various simulation experiments by Stoica *et al.*, and it has repeatedly demonstrated to distribute IDs to nodes effectively (Stoica *et al.*, 2001; Liben-Nowell, Balakrishnan & Karger, 2002b; Godfrey, Lakshminarayanan & Surana, 2004). Chord usually uses SHA-1 (Secure Hash Standard, 1993) as its principal hash function for hashing and subsequently mapping node IDs and keys to nodes thus helping load balancing. Combined with its probabilistic properties, it typically allocates a consistent number of keys to the managing nodes.

Another property of Chord is its exceptional scalability, which can be observed when running simulations. The first thing noted is that the lookup latency grows evenly with the number of nodes (Stoica *et al.*, 2001). Chord's excellent scalability is in part a

consequence of a couple of combined factors: for one with the PDF realized it gives nodes a certain guarantee that they are evenly distributed and acquire also a relatively proportionate number of keys to manage within the circumference of the ring. Another component is that each Chord node maintains a small routing table, known as the finger table (see Section 2.7), whose dilation is small, making it easy to conduct updates every 30 seconds with the stabilize() command and to keep all the pertinent routing information current. Due to these properties, Chord can perform accurate lookups with high probability. Also, by selecting a small number of neighbors leads to a small dilation and eliminates a large portion of overhead which makes this approach able to scale rapidly. Older systems such as Napster (Fanning, 1999) or Gnutella (Frankel & Pepper, 2000; Oram & O'Reilly & Associates, 2001) did not scale as well because of centralized servers, lengthy linked lists, or flooding entire infrastructures with queries.

Another key factor is the simplicity of the finger tables that make it possible to perform lookups quickly and to handle constant fluctuation, such as changes due to node departures or joining of nodes. It is shown in experimental settings (test beds), that it is nearly impossible to shut down an entire Chord system even when there are massive node failures due to, *e.g.*, an electrical outage in an entire geographical region (Manku, 2003). This is enabled by a specific property of Chord: when there is a timeout, a node realizes that the successor is no longer present, proceeds down the list, and continues to ping the next successor(s) until it reaches some node that is alive. In simulation tests, Chord does show the resiliency to endure major stresses. However, there is also a drawback: At times

the Chord ring separates into distinct entities and does not realize that a partitioning has occurred, continuing its routing to only the nodes that it believes have survived.

Flexible naming structure is the last of the system model components. Here it means that Chord uses a flat keyspace that can be anything one wants it to be. The only condition is that this naming structure is fixed in length. Typically, SHA-1 hashing is used to produce the keys. When using any P2P application in conjunction with Chord, the application manages the naming of the keys that are hashed and distributed (Stoica *et al.*, 2001; Liben-Nowell, Balakrishnan & Karger, 2001).

Overall, as examined, the Chord protocol does well under severe stresses, churn events (rapid and continuous node arrivals and departures), and even large-scale events such as power outages. This protocol is very powerful and has tremendous advantages over past protocols due to its simple yet dominant structure, with its exceptional scalability (no hotspots), load balancing, resistance to censorship, flexibility in its naming structure, and making itself readily available to receive a large number of nodes without compromising its infrastructure.

2.3 Structure of the Chord Ring

Figure 2.1 is a basic diagram of the Chord address space that displays how the successor nodes manage the keys in a clockwise mode. Since the number of nodes is smaller than the size of the keyspace, keys are always assigned to the next node on the ring.

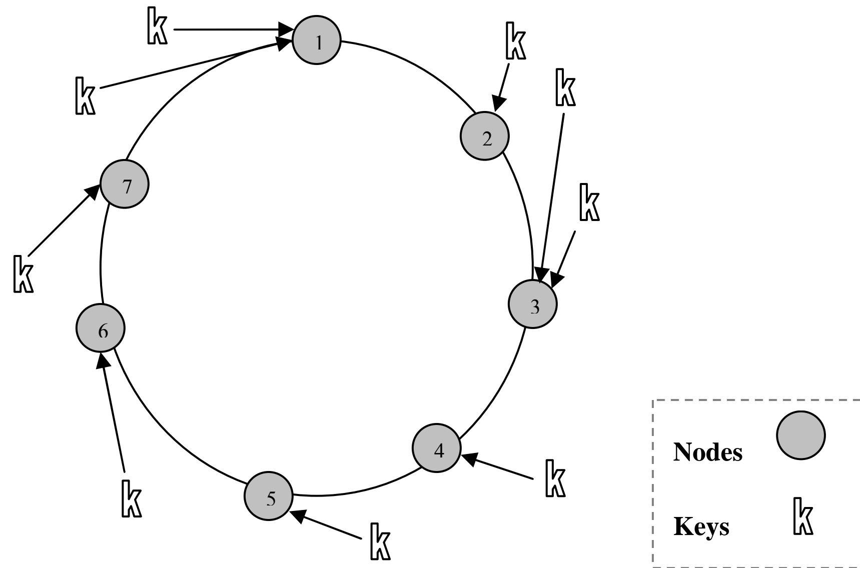


Figure 2.1 Diagram of Chord nodes and keys on the ring

Chord has features that set it aside from past protocols such as DNS, invented by Paul Mockapetris in 1983 (Wikipedia, 2005). For illustration, Table 2.1 shows the differences between Chord and DNS. As one of the grandfathers of the Internet, DNS has been around for over three decades and is still the main engine that enables lookups and the resolution of IP addresses and domain names. It was initially intended to map IP addresses to services or grant them values, based on a hierarchical design. DNS works by routing through a tree structure where the suffix such as ".org" or ".edu" is the point of origin to determine where to begin resolving the domain name of a particular site. With the growth of the Internet and deployment of IPv6, modifications to suffix based lookups are taking place as shown in RFC 1752 (Bradner & Mankin, 1995).

Chord	DNS
Could technically provide a lookup service by hashing into values all the names of the hosts	DNS is a lookup service that provides access to host names that are represented by keys and IP addresses are values
No servers, decentralized; nodes are self-governing	Numerous root servers
Self-organizing structure	Needs to be manually managed
Ring structure	Hierarchical in nature
Flat naming structure and can be named anything by application using Chord	Needs to resolve names
Chord will lookup data or services and it does not need to be tied to an explicit host	Needs to associate host names with services and afterwards return the results

Table 2.1 Crucial differences between Chord and DNS

As seen Chord can offer the same services as DNS, but instead of employing a hierarchy, Chord would hash all the IP addresses of the hosts promoting services into keys. This would provide a highly scalable system that would not utilize any root servers, as proposed by Dabek *et al.*, (2002).

Another system that is comparable to Chord but utilizes a diverse routing strategy is CAN or Content Addressable Network (Ratnasamy *et al.*, 2000). With CAN, routing is

based on a *d*-dimensional Cartesian graph that is used to distribute the keys to the nodes. Chord, on the other hand does not use the Cartesian graph method, instead it uses a one-dimensional space (a ring) to map nodes.

Although Chord has few drawbacks, there are a couple that should be addressed, such as the concern over anonymity. Chord does not provide a way for nodes to remain fully anonymous, since the nodes use their IP addresses along with their port numbers for node ID generation by hashing. Other routing structures do provide this feature at the expense of overhead or also have other types of routing methods such as logical addressing, lists, trees, etc., which usually means that they are not fully autonomous and self-governing. They may use root servers and their routing latency can be high (Stoica *et al.*, 2001; Dabek *et al.*, 2002; Manku, 2004).

Another disadvantage known about Chord is that sometimes queries do not reach the intended destination. This can happen for numerous reasons, but typically is related to a query taking place before the nodes have completed stabilization and the query leaving the boundaries of the Chord ring. This can cause the Chord ring space to break and form smaller disjoint rings that may not have knowledge of each other (Stoica *et al.*, 2001).

2.4 Consistent Hashing

In order to form a Chord ring, usually SHA-1 is used to hash and map the node IDs and also to hash the values and distribute the resulting keys to the nodes. The result appears to be a seemingly random and consistent distribution of keys and node IDs, making this scheme quite desirable to use. The idea of consistent hashing is that at each

node level a small routing table (see Section 2.5), also known as a finger table, is used. The finger tables will update themselves when nodes enter or leave the Chord ring, without needing to re-hash and re-map the entire ring structure.

These finger tables contain only the information of the successors of the node along with the node ID and an offset. The node only needs to know its adjacent successor and the finger tables have only a limited number of entries. Therefore, consistent hashing is the preferred choice of hash algorithm since it will help the nodes to update their routing tables with ease and updating will not affect other parts of the Chord system. If there is activity taking place in the system, such as node joins and departures, it is handled separately in diverse areas of the Chord ring.

SHA-1 is used in combination with numerous algorithms and applications that are cryptographic in nature, thus considered a preferred choice to map and distribute IDs. As mentioned in Section 1.5, it is known that SHA-1 can now be broken in much fewer cycles than originally conceived (Schneier, 2005; Wikipedia, 2005). According to experts, the use of SHA-1 in future secure applications will need to be avoided. Either the SHA-2 family is employed, or a different hashing algorithm is devised as base hash function. This will lead to the inevitable fact of having much lengthier message digests as well as more overhead in each message exchange. The routing tables will also need to be able to handle longer bit lengths in their message exchanges. However, one of the benefits of using the SHA-2 family is that to this date it has proved to be unbreakable and as a result is considered collision resistant.

2.4.1 Creation of Node IDs

In order to begin assigning IDs to the nodes, first the IP address of each node is hashed along with the port number of the node. This produces a message digest that has the fixed size of 160 bits using SHA-1. It does not matter where any of the nodes are located geographically; they are placed along the ring structure according to their 160-bit node ID (Dabek *et al.*, 2002; Horovitz, 2005; Wikipedia, 2005).

2.4.2 Creation of Key IDs

Now, by having formed the Chord ring with n equaling the number of nodes around the circumference, the keys need to be placed. This process, known as key ID assignment takes place by first finding the values that are to be hashed. This value can be anything and is typically assigned by a P2P application such as CFS. The value will then be hashed with SHA-1, resulting in a 160-bit message digest.

It can be shown theoretically that the resulting key IDs tend to be uniformly distributed and key clustering is minimized (Baldwin, R. G., 2005). This is important because when clustering takes place the key IDs will be mapped to a specific area in the ring and other areas will be left bare. This is illustrated in Figure 2.2 below, an example of a small Chord network.

The diagram shows that proximate nodes manage a more or less equal number of keys once they are distributed to the nodes. Some nodes, however, will manage more

keys than others will, but considering the overall scenario, they are fairly evenly distributed.

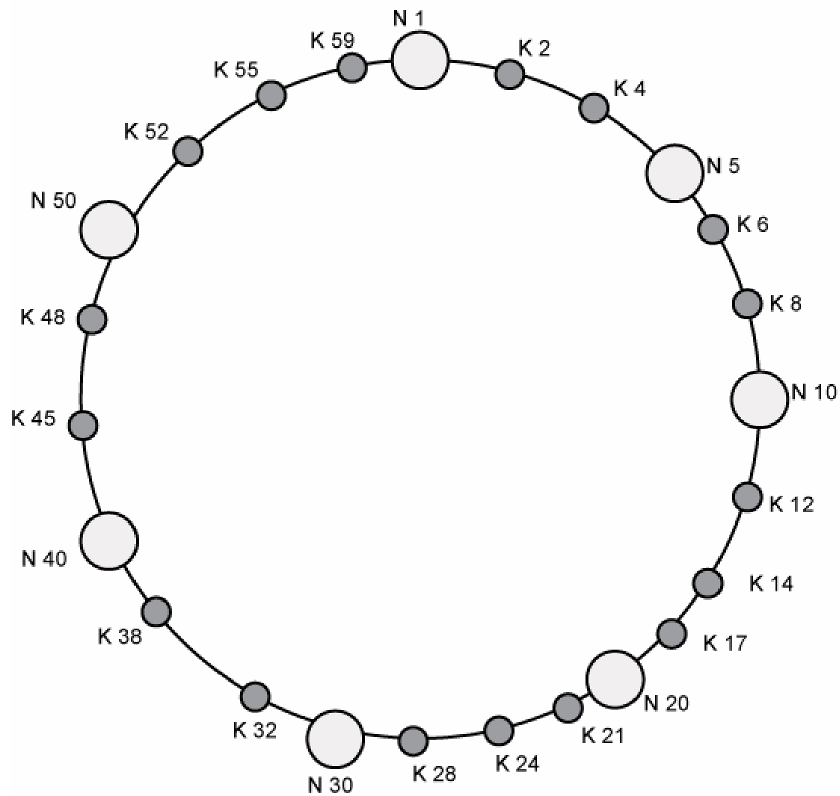


Figure 2.2 Creation and distribution of keys in Chord ring

2.4.3 Mapping Node IDs and Key IDs to the Chord Ring

Once the nodes have acquired their IDs as well as the values have been hashed and the keys produced, the ring address space can be formed.

First, the nodes are placed according to the produced identifiers, the 160-bit message digests modulo 2^m, where *m* determines the size of the Chord ring. Then each key *k* is mapped to a node that is equal to or follows *k* in the Chord ring. This node will then be known as the successor node of that particular *k*. Thus, the nodes form a circle and keys are inserted until all the keys have been distributed essentially equally and consistently throughout the circular space. The pseudo code for creating Chord ring ID *n* would resemble this (modified pseudo-code from Stoica *et al.*, 2001):

```
n.create();
predecessor = null;
successor = n;
```

Once a particular Chord ring has been formed a new node, *n*', that wants to join and participate in this system, will begin by running the Chord software and produce the ID for *n*'. Then *n*' will ask any node in the Chord ring to query its routing table and direct it to a place within the range of its closest successor. Then the new successor of *n′* starts the process of updating its own routing table and messages are exchanged with the old predecessor as well. The pseudo-code to join a Chord ring containing node *n* is:

```
n'.join(n);
predecessor = null;
successor = n.find_successor(n');
```

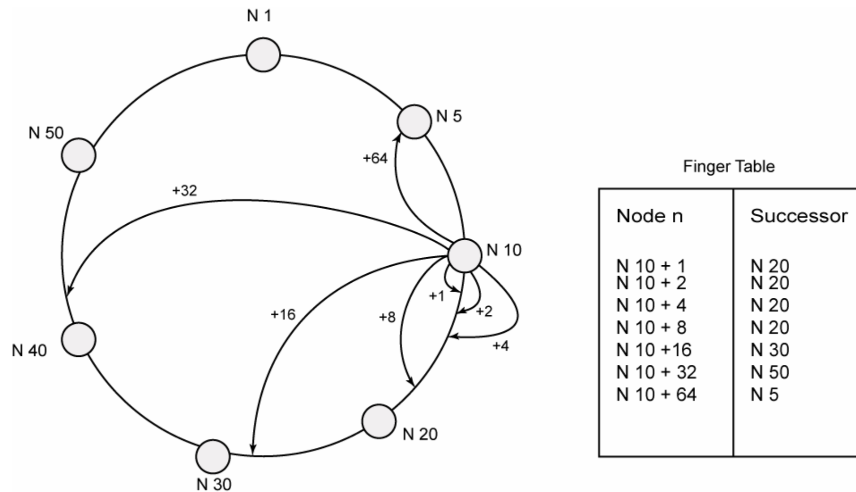
2.5 Finger Tables

Finger tables are used in each of the Chord nodes using *O*(log *N*) table size, as part of the Chord routing mechanism. These relatively small routing tables are utilized to collect information that helps each node perform lookups and route queries, efficiently

and without abundant overhead. They sustain information about the successor nodes, the number of "hops" (dilation) to the next node, as well as keep information about where the successor node is located. This enables new nodes to be inserted with ease into the Chord network since each node only knows $O(\log N)$ information about successor nodes. With consistent hashing (see Section 2.4), the finger tables can be updated with minimal effort since this will not change the entire infrastructure of the Chord ring. Only updating the tables as well as notifying the predecessor and the new successor is required.

Another property of finger tables is that they handle node failure well. If a node disappears, Chord will only need a small portion of correct data to be able to perform its lookups (Dabek *et al.*, 2002). In addition, these finger tables make it possible to lookup every other peer as well as the successor node in constant time.

The finger tables are structured as follows: Once the ring has been established, each finger table, located at each node n, will have its ith entry show the successor following or coinciding with the node ID equal to $n + 2^{i-1}$ with $1 \leq i \leq m$. Below in Figure 2.3 is a pictorial example of a Chord ring and finger table with $m = 7$.



Node n	Successor
N 10 + 1	N 20
N 10 + 2	N 20
N 10 + 4	N 20
N 10 + 8	N 20
N 10 +16	N 30
N 10 + 32	N 50
N 10 + 64	N 5

Figure 2.3 Example of Chord ring and finger table

2.6 Stabilization: Joins and Departures

The stabilization process is at the core of Chord. It is the process in which the finger tables are updated to ensure the integrity and correctness of the routing information.

2.6.1 Node Joins

The Chord ring needs to periodically run a method known as stabilize() on each of the finger tables. This occurs about every 30 seconds. This method will take care of updating the finger tables of the nodes that are joining or departing the network. This helps preserve the integrity of the Chord ring and keeps the successor links up to date.

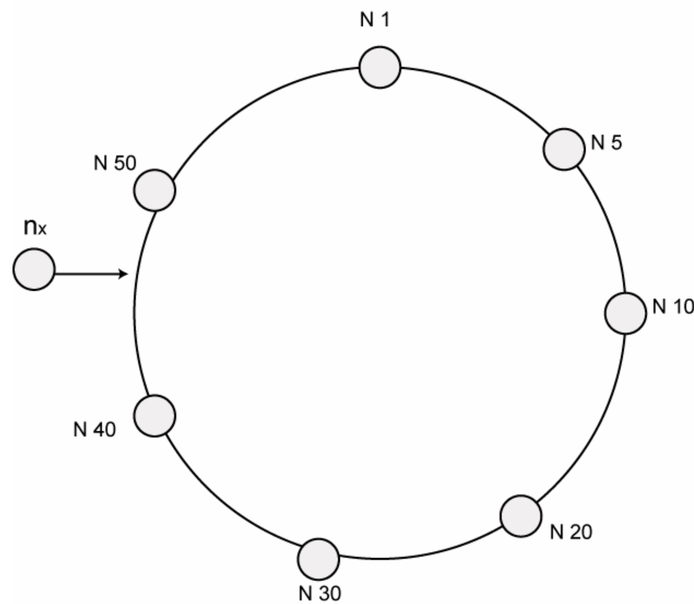


Figure 2.4 Node n_x wants to join the Chord ring

Therefore, when a new node n_x decides that it wants to join the Chord ring (Figure 2.4) and has already hashed its node ID, it begins the insertion process by first asking any node if it knows who its successor may be as seen in Figure 2.5. The node will check its finger table and forward the request to the node whose node ID falls between the successor of n_x and its predecessor. Once the new node n_x establishes its place in the Chord ring, it will update its finger table and insert node n_s (the successor node) as its immediate successor and will then ask node n_s for the predecessor of n_x. Then the finger tables are filled by asking n_s to lookup all successors of n_x for each entry. Also, n_s will then update its own finger table and will assign the newly inserted node n_x as its immediate predecessor. Then n_x will seize the previous predecessor of n_s, making it its

own new direct predecessor. Again then the entries in the finger tables of all nodes are updated.

Now supposing that n_x is actually node number 25 and first asked node 50 (Figure 2.4) for referral to where it is really supposed to be inserted in the Chord space. Node 50 will tell this new node with the ID of 25 that its immediate successor is node 30 and from this point on the following happens: Node 25 gets in touch with node 30 and tells this node that it wants to join the ring. The node 30 (the new immediate successor of node 25) updates its routing table and makes node 25 its new predecessor. The old predecessor of node 30, which is node 20, is also notified and makes node 25 its new immediate successor. Now node 30 will help node 25 fill its finger tables in order for node 25 to know where to route queries and be able to perform lookups. At this point, the keys that were managed by the initial successor (in this example node 30) are transferred to their new owner, in this case the newly joined node 25. Below is an example of how the finger tables of node 20 and node 30 would look before node 25 joins (Table 2.2). This refers to Figure 2.5 that exhibits where these nodes would reside in the Chord ring before joining of the new node is finalized. Table 2.3 shows the finger tables after joining of node 25.

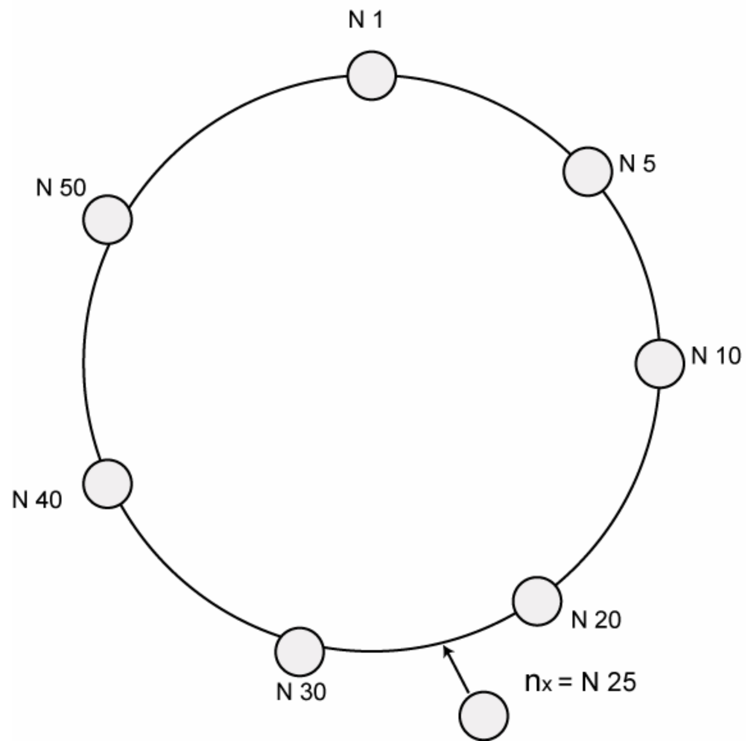


Figure 2.5 Node n_x = N25 finds its place in the Chord ring

Node 20		Node 30	
$O(\log N)$ hops to successors	Successor or i^{th} finger	$O(\log N)$ hops to successors	Successor or i^{th} finger
N20 + 1	N30	N30 + 1	N40
N20 + 2	N30	N30 + 2	N40
N20 + 4	N30	N30 + 4	N40
N20 + 8	N30	N30 + 8	N40
N20 + 16	N40	N30 + 16	N1
N20 + 32	N1	N30 + 32	N5
N20 + 64	N30	N30 + 64	N40

Table 2.2 Finger tables for node 20 and node 30

Node 20		Node 25		Node 30	
$O(\log N)$ hops to successors	Successor or i^{th} finger	$O(\log N)$ hops to successors	Successor or i^{th} finger	$O(\log N)$ hops to successors	Successor or i^{th} finger
N20 + 1	N25	N25 + 1	N30	N30 + 1	N40
N20 + 2	N25	N25 + 2	N30	N30 + 2	N40
N20 + 4	N25	N25 + 4	N30	N30 + 4	N40
N20 + 8	N30	N25 + 8	N40	N30 + 8	N40
N20 + 16	N40	N25 + 16	N50	N30 + 16	N1
N20 + 32	N1	N25 + 32	N1	N30 + 32	N5
N20 + 64	N25	N25 + 64	N25	N30 + 64	N40

Table 2.3 Addition of node 25 to the Chord address space

As observed, the finger table of node 30 does not change with this new addition since node 25 is the predecessor of node 30. This is a very small Chord ring, which cannot reflect all possible situations, but serves to demonstrate that the routing tables will typically be modified as node joins or departures take place.

The method stabilize() is then used at each node and it is efficient in that it will look for the immediate successor and check if it is still alive. If that node is gone, it will search for the next successor on the routing table and make that node the immediate successor. This helps to keep the Chord ring in a well-maintained state and makes all the nodes in the ring reachable (Dabek *et al.*, 2002; Liben-Nowell *et al.*, 2002a).

2.6.2 Handling Node Failures and Departures with High Fault Tolerance

Sometimes a node that is ready to depart the network voluntarily will ensure that the keys are transferred to the successor node and the pertaining information is kept intact. Nevertheless, in a realistic scenario many nodes will disappear or depart without announcement. At this point, the stabilize() method will repair the lost links. For example if node 40 would simply vanish unannounced from the network as illustrated in Figure 2.6, new steps will need to be implemented in order for the finger tables to be updated and contain correct information.

Once stabilize() is run and the adjacent nodes notice that node 40 has departed, they will self-repair by going down the successor list(s), as each node knows its immediate successors.

The swapping of stabilize messages between the nodes to correct and update the routing tables takes place at this time. Other techniques that rely on applications can be implemented as well, supposing that an application can create replicas of the data that is associated with the keys of the immediate successors. This feature of node failure and the

creation of replicas apply directly to how Chord possesses a high degree of fault tolerance and redundancy.

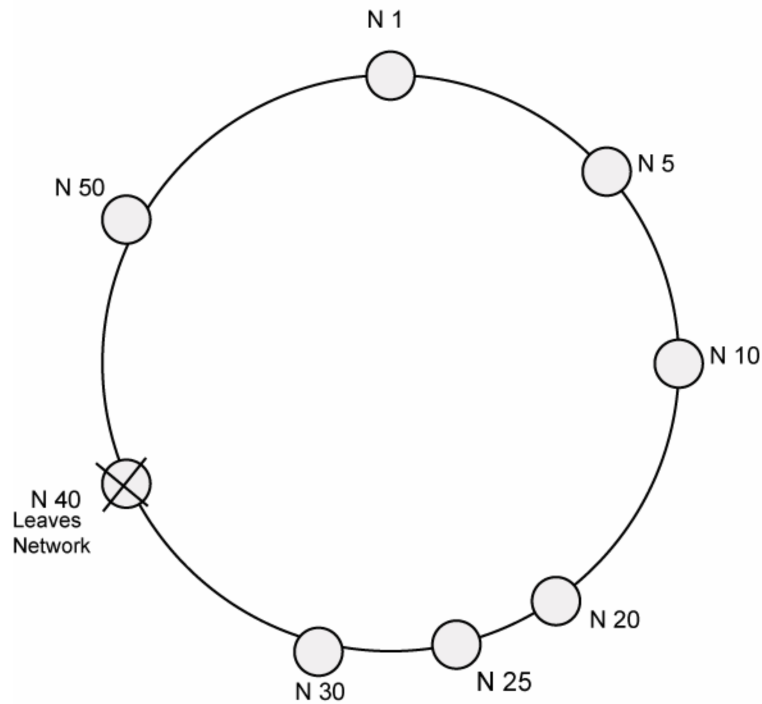


Figure 2.6 Node 40 leaves network abruptly

High fault tolerance is a property that distinguishes Chord from many past protocols and distributed systems where mass node failure would simply disintegrate the system. With Chord, it takes only one node down the successor list that still contains correct routing information to repair the entire ring.

2.6.3 Lookups during Stabilization

The Poisson process also known as "the law of small numbers," was discovered in the 1800s by Simeon Denis Poisson (Stark & Woods, 1994; Wikipedia, 2005). The Poisson process is generally useful for modeling of random event arrival times. It is another key element that serves to understand why churn can be handled in Chord.

To recapitulate, stabilize() is run periodically around every 30 seconds to update the finger tables. Dabek *et al.* (2002) tested key lookups using the Poisson process for node arrivals and departures. Their experimental results are consistent with theoretical expectations; Chord proved to be robust. With the Poisson process the stabilize() method is run at different intervals, not with the usual 30 seconds lapse, but within a range of 15 to 45 seconds. This creates a more uniform and reachable network,

If a node performs a lookup when the periodically run stabilize() is active and it has not finished its cycle, timeouts will take place. When the querying node cannot find the finger table, because it departed or joined, *e.g.*, 10 seconds early before stabilize() had completed its updates, loops can be created and this is serious because it can cause partitions and split the ring.

2.7 Load Balance with High Probability

The topic of load balance merits special examination. As previously seen, consistent hashing is a way to map nodes and keys to the ring topology, so that the nodes and keys that they manage are uniformly distributed. Therefore, in a large-scale network

the ideal notion would be to distribute all the keys in such a way that the number of keys per node is constant throughout the entire network (Dabek *et al.*, 2002).

The authors Dabek *et al.*, in their paper, "Building P2P Systems with Chord" (2002), demonstrate that in order to find the successor of an ID takes only $O(\log N)$ messages to be swapped between nodes. This happens with "high probability" where N is equivalent to the number of "servers", also known as "hosts", in the ring. Removing or disjoining a node also takes place with a high probability of $O(\log^2 N)$ messages to be exchanged.

It would appear at first glance that the load distribution is not guaranteed to be consistently even. So then, why such discrepancies? One of the reasons is that many nodes have not been placed in an evenly distributed Chord ring space, and will have longer paths to successors or predecessors (Godfrey, Lakshminarayanan & Surana, 2004).

Some possible problems include: The size of objects might not be the same when they are distributed to the nodes and some of the node IDs may not be chosen in a completely randomized way. Some claim that the insertion of virtual servers at each node would dramatically improve load balancing in Chord. Some of this future work suggests that perhaps further examination on how to predict volume such as with virtual nodes would be of great benefit. Since volume of data assigned to nodes is not really taken into account it would be good to assign the greater volume to already established nodes that carry less traffic. Bandwidth and storage also affect load balancing and in the future could be changed by establishing a measurement and considering the relationship among the load capability of the nodes.

2.8 Effects of Churn: Rapid Node Join and Departures

"Churn" is a term that is used to describe the presence of constant and continuous node arrivals and departures in a distributed network. Churn is critical to evaluate and validate any distributed P2P network since in real world models (not simulations) a DHT needs to prove to be capable of being submitted to severe stresses such as massive churn and being able to self-repair. The authors of "Handling Churn in a DHT" (Rhea *et al.*, 2004) examine various DHTs and submit them to various tests in order to verify that these DHTs can survive under heavy traffic.

There are three factors that Rhea *et al.* (2004) consider in order to evaluate churn in a DHT. The first factor that Rhea *et al.* examine is reactive vs. periodic recovery. The second one is the way that message timeouts occur and how this affects link recovery. The third is proximity neighbor selection and its effects on the effectiveness of DHTs.

Reactive recovery takes place when a node looks at its routing table and determines that a node has vanished and will try to replace the node right away. When testing Chord, most of the results in the lookups under churn turn up correct, but the lookup latency also increases. In this particular study, the authors compare with another DHT known as Bamboo, which shows for this reactive recovery to consume less than 1/3 of the bytes per second per node in contrast to Chord. This study demonstrates that Bamboo may have a lower latency since it performs its lookups in a recursive way whereas Chord uses iterative methods.

Chord uses periodic recovery the same as Bamboo (Rhea *et al.*, 2004). Reactive recovery can create a phenomenon known as "positive feedback cycles" that when a network is heavily loaded some of the lookups timeout and the neighbors believe that the node has failed. Then each neighbor will update its routing tables with wrong information, send messages to the predecessors and successors on the list, and increment at the same time the traffic load, and saturate the already overloaded link. Thus to ensure that these "positive feedback cycles" do not occur, periodic recovery will help prevent these cycles from occurring by sending only a periodic message to the neighbors with all the information.

Both reactive and periodic recoveries have proven during tests to return well-established, valid, and correct results. Nevertheless, in terms of bandwidth consumption reactive recovery is advisable to implement when there is a low amount of churn. However, when churn is heavy all the message exchanges become expensive, consume a lot of bandwidth, and make it costly and inefficient.

2.9 Initial Conclusions

Some of the initial conclusions regarding Chord can be observed from the paper "Observation on the Evolution of Peer-to-Peer Systems" (Liben-Nowell, Balakrishnan & Karger, 2002a) where the authors observe that Chord cannot be in an ideal state due to the continuously dynamic nature of P2Ps. Even when maintenance protocols are used in order to keep all the updates current in routing tables, it does not guarantee an ideal state model.

Chord has the ability to scale rapidly logarithmically and at the same time can handle simultaneous node failures, but if node failures are large-scale, there is also a risk of forming a “loopy” network that is “weakly but not strongly stable.” In order to prevent a “loopy” network the same authors, Liben-Nowell, Balakrishnan & Karger (2002b), suggest that this can be corrected by keeping an additional number of predecessors in the finger tables as a set of pointers before running a method known as idealize().

Other studies show that a Chord network can fail. The node can search itself and traverse the entire ring until its predecessor points to the inquiring node. This would cause unnecessary travel in a large network and would increase the cost of the lookup in the number of hops (dilation).

In other instances, regarding Chord, Dabek *et al.* (2002) indicate that there is still no anonymity or deniability with Chord. This is hard to implement because the Chord structure forms such a strong bond between the node and the information associated with it (the keys). The authors believe that in order to improve some factors they recommend that the formation of an overlay providing services such as anonymity would be useful. This overlay would run on top of the Chord ring transparently but would provide enhanced security.

In addition, there is the question of query latency with Chord and some believe that it would be beneficial to give servers in the vicinity some of the requests. This is difficult to implement realistically since node performance would first need to be tested. However, the goal of Chord is distribution and with this comes the distribution of responsibilities to the participating nodes.

The question of network proximity is not applicable to Chord; it does not support this notion. Forcing network proximity onto Chord defeats the purpose of having a fully independent and self-governing unit (Liben-Nowell, Balakrishnan & Karger, 2002a). Currently, there does not exist a way to test network proximity with Chord. Network proximity is supported by other protocols such as Pastry (Rowstron & Druschel, 2001) and CAN (Ratnasamy, 2001), utilizing a torus topology.

Other questions arise such as when malicious nodes are introduced into a Chord network. One way to check if this situation is true would be by having another node in the network look up the suspicious node. Then the node verifies that it is in fact the correct node and not a malicious participant. This would check for integrity and ensure a higher sense of authenticity. However, if the lookup turns up a different result than what the inquiring node expected, then the follow-up node knows that the ring space could have been compromised and tricked by a malicious participant.

There are in fact, countless issues with each routing P2P protocol, as pointed out by El-Ansary *et al.* (2003). Nevertheless, according to their analysis, Chord could also benefit during the stabilization of connections by implementing an “active correction”. This works by piggybacking the expired lists during lookups or joins. This decreases the chances for information not being recognized as expired and avoids the formation of a phenomenon known as the “expired lists syndrome.”

As noticed, the initial conclusions suggest there are many possibilities for Chord to be modified, such as introduction of virtual servers to improve load balance. In addition, expanding or keeping an extra link to an additional node in the finger tables

could improve efficiency. Anonymity could technically be applied to Chord either through an application or by creating an overlay. The Chord protocol is flexible, has low state maintenance and offers seemingly endless possibilities; it has also proven time and again to be extremely efficient (Stoica *et al.*, 2001; Dabek *et al.*, 2002).

Chapter 3 Symphony

"When we would prepare the mind by a forcible appeal, an opening quotation is a symphony preluding on the chords those tones we are about to harmonize."
Benjamin Disraeli, British politician (1804–1881)

This chapter covers first some theory on the small world phenomenon, which is the basis of how Symphony is structured. Then, it describes the important properties that Symphony possesses such as its flexible structure, how it manages load balancing, why it has such low latency and the way that it distributes and maps IDs to its circumference with three distinct variations: Regular, randomized, and balanced. Lastly, methods such as Neighbor of Neighbor (NoN) are discussed and how this routing mechanism works with the new property of 1-lookahead and why it is beneficial for this specific protocol.

3.1 Models of the Small World Phenomenon

The Symphony protocol created by Manku, Bawa & Raghavan (2003) is rooted in other existing algorithmic studies. The way that Symphony is structured has a well-studied and solid background that encompasses not only popular folklore and social network studies, but also proven algorithmic foundations.

It started in the 1960s when Stanley Milgram and his team devised an experiment based on how people have common acquaintances even among strangers (Milgram, 1967). They proved in this experiment that by giving a person a letter to "route" to a

perfect stranger it could arrive at the correct destination in an average of "six degrees of separation" (Travers & Milgram, 1969). If the perfect stranger were given a letter with basic information such as the recipient's address, name and profession, with the condition of giving the letter (routing it) to another person who may be closer to the object recipient, it would in fact arrive quickly in many cases.

Milgram's experiments showed some incredible results: that short paths to a destination are created and can efficiently route the information that needs to be transmitted. Through these results, it is then safe to conclude that there are in fact short links or paths of separation between acquaintances.

The basis of Kleinberg's theory (Kleinberg, 2000) is also a result of Milgram's social network theories, and also of the Watts and Strogatz (1998) prototype. In later years, Watts and Strogatz took into account Milgram's "small world phenomenon," and played an important role in proving how this phenomenon is useful in the computer world, relating to large-scale distributed networks and the World Wide Web.

The Watts-Strogatz prototype of the "small world phenomenon" was modeled as a computer network with a ring topology where the nodes are spread uniformly around the space. These nodes would then be joined to nearby neighbors, called the local contacts. Then long-distance edges were generated at random.

Although Kleinberg states that it is not provable that there could be a sole, decentralized algorithm to create short-distance paths, the Watts-Strogatz prototype demonstrates that even if it belongs to a large family of decentralized networks, it

provides a way for the decentralized algorithms to perform well, grant them freedom and be efficient in their routing methodology.

A few questions are addressed by Kleinberg (2000) concerning why the "Small World Phenomenon" occurs. The first question asked by Kleinberg is why there should be an occurrence of small chains or links of peers that are connected by pairs of unknown peers. When there are random networks, a phenomenon occurs that leads them to have a "low diameter", *i.e.*, short paths to destinations exist.

If a network, be it social or computer based, has a symmetric base and not a random one, it would be nearly impossible to prevent the occurrence of clustering. This clustering, as seen in Section 1.3, is a phenomenon that takes place frequently either with poorly designed hash tables or when poor hash functions are used to distribute and map nodes or keys in the chosen topology. With clustering of an area, or areas, the used topology is over-utilized whereas other areas are not used. This property is not desirable since the even, random distribution of all the elements renders low diameter and short paths to destinations.

The second question that Kleinberg poses with regard to the "Small World Phenomenon" is why then are these pairs of peers that do not know one another capable of discovering short chains of people that they know will eventually link them together? Milgram calls these "cues" within a social network meaning that people instinctively know how to pass information rapidly from beginning to the end. This is applied to computer networks as well by showing that if a routing table knows the recipient, then it is easy for the information to be passed along. If the routing table is unsure of that, it will

use a randomly generated link to try to come closer to the target by sending the information along a long-distance link.

Kleinberg's model establishes that each node should have many local links as well as some long-distance links. Kleinberg's prototype is not using a ring topology but a two-dimensional square grid.

Beginning (as shown in Figure 3.2) with two nodes in the network, *s* and *t*, to pass the information and only using local information to route the data, Kleinberg's algorithm displays a two-dimensional grid network that clearly shows the interrelationship between long-range links and discovering short paths through these long links. This enables routing to a recipient with low diameter.

3.2 System Model

Symphony (Manku, Bawa & Raghavan, 2003) emulates in many ways Kleinberg's small world algorithms, but differs in the prototype design since Symphony uses a ring topology, similar to the Watts-Strogatz model, yet differing in the principal routing methodology.

Symphony, like Chord, congregates all the components for a large-scale distributed network. It has a strong and powerful scaling technique, it is stable in that it can survive extreme stresses, yet structurally flexible and simple to implement.

Symphony proves that its performance not only meets all the fundamentals but also exceeds all the expectations of a solid and adjustable P2P overlay network that would be a superb addition to real world, large-scale distributed networks, if it were to be

implemented in the near future. Shown below in Figure 3.1 is a model of a Symphony network that displays its short-distance links, the reinforced short-distance links, as well as the long-distance links.

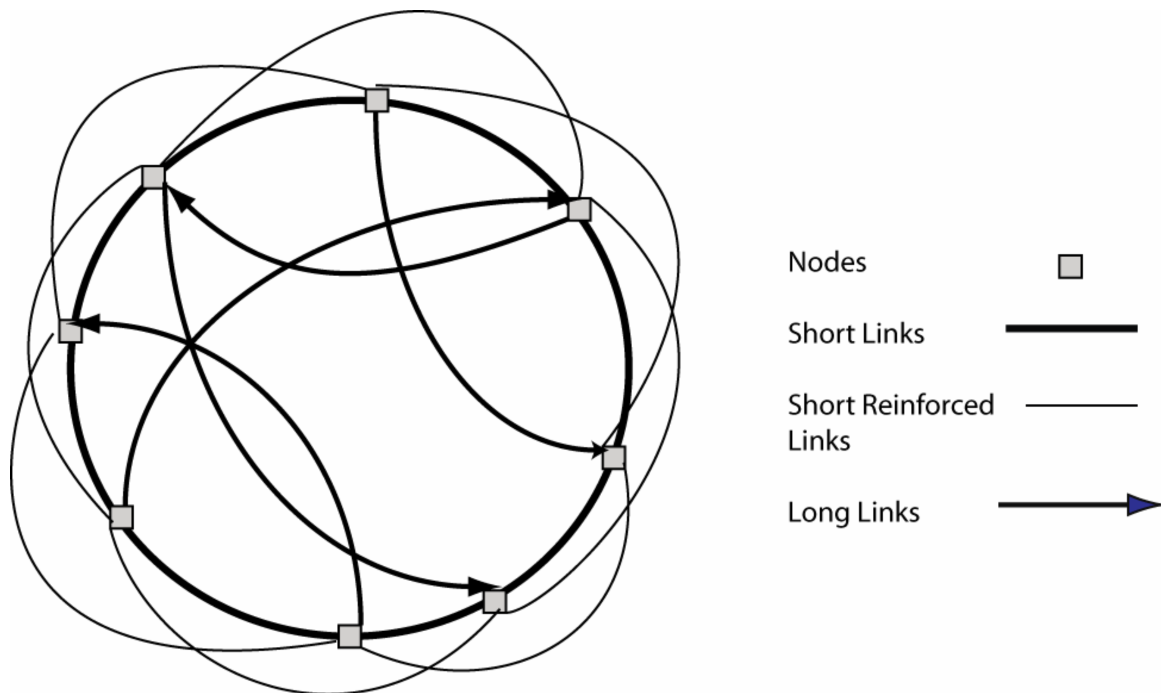


Figure 3.1 Simple example of a Symphony network with short reinforced links and one long distance link per node (original design by Manku *et al.*, 2003)

To recapitulate, the Symphony logical overlay handles a large load of dynamic hosts and provides the basic services of inserting, deleting, and looking up the hashed keys that store information at node level. As seen before, it is not advisable for a large-scale P2P to maintain a global hash table of thousands of nodes due to the difficulties of

keeping it up to date; it would not be practical or feasible to update large quantities of data in such a structure.

That is one of the reasons why Symphony, like many other P2P DHTs, breaks these large tables into more manageable blocks that can then be managed by local nodes. Information can be quickly accessed, deleted, and inserted when smaller portions are utilized. Then, as seen previously, Symphony also manages its routing information through routing tables (in Chord these are labeled as finger tables, see Section 2.5). There are differences concerning Symphony that must be addressed in order to explain why this protocol is unique.

3.2.1 Load Balance

One of the topics that is always a concern with DHTs is the way that load balance is handled. The amount of messages exchanged in a large-scale network can saturate a poorly designed system. When Symphony is used in simulations it consistently excels in performance, *e.g.*, in highly dynamic environments, and balances the distribution of IDs well. Each of the lookups begins at a randomly chosen node and the hash key is then elected at random in an even way with a range from zero to one. This was analyzed by Manku, Bawa & Raghavan (2003); simulations show that in a network with $N = 2^{15}$ nodes and $k = 4$ long-distance links per node, where each node performs a lookup of a random hash key, a well-balanced system results.

3.2.2 Low State Maintenance

Symphony has consistently been demonstrated through experiments to excel in providing a low state maintenance. This is an important consideration in a P2P because if there are frequent updates, as they occur in dynamic networks, the traffic management needs to be smooth. With Symphony, there are less frequent pings and keep-alive messages for nodes that are departing or inaccessible. This minimizes the traffic usually generated by message exchanges between the routing tables and creates room for faster operations. This typically refers to having a low degree in a DHT, since the maintenance protocol will not propagate huge amounts of overhead in the message exchanges. Moreover, when the node pairs are small in range, then the overhead will also be decremented in order to prevent high loads of traffic from traversing the system and causing unnecessary traffic.

Another property that can explain the low degree of maintenance is that when a node joins, the amount of time to adhere it to the hash table is extremely fast, with little time lost. When nodes depart the system, the hash tables will also recover quickly. Overall, this is a desirable property to possess because if there are less open connections to the hosts in the network then the network will not have so much traffic.

3.2.3 Fault Tolerance

Symphony has a high index of fault tolerance meaning that it can self-repair and survive with just about any amount of node failures, simultaneous node failures, or node joins. The Symphony model only backs up the short-range links from the predecessors

and to the successors in the ring topology but excludes any backups for the long-distance links.

The long-distance links seen in Milgram's theories are algorithmic approximations that are not easily explained by probability or statistics, since their nature is extremely complex. However, in this case, Symphony uses the long-distance links to create short paths to the desired target. This also diminishes the amount of traffic in the overlay and generates more space for the free flow of information.

3.2.4 Flexibility

This is another constituent that is addressed, given that Symphony is the only known large-scale P2P protocol that allows its nodes to own or manage different numbers of links. As examined with Chord, its finger tables are not flexible, but grow logarithmically with the number of nodes joining or leaving the system. In other words, Chord needs, like many other protocols, to adjust to node joins or departures and adjust the routing tables when necessary. Nevertheless, the big difference is that with Symphony the number of links is not an issue.

Symphony's structure is so versatile that it can adjust to any number of links at the node ends allowing for diversity to form part of the system. This clearly shows that with this characteristic in particular, it will make itself readily available to support mass numbers of node joins or departures with facility. As seen at the beginning of this thesis, some older schemes such as Gnutella or Napster (Section 1.1 and 1.6) are limited in their

availability or ability to make themselves able to host colossal amounts of nodes without creating more traffic than sustainable.

3.2.5 Latency vs. Outdegree

"Latency" (Wikipedia, 2005) is a metric that helps determine the round-trip time for a packet or request sent from a source to a target. It is a measurement that evaluates how long it takes to complete a transmission. It generally does not take into account the link's capacity to push through data, or even the size of the packets that are to be sent. However, it considers the smallest amount of postponements for a particular link. The amount of delay needs to be measured from source to destination in order to gather information about the performance of any network, be it a centralized one or a distributed P2P.

Symphony uses a special feature that can regulate the number of packets that are going to be sent out by the user(s). When the protocol begins running it is not set to any specific number but can be adjusted afterwards. This is the only P2P protocol that offers this feature. Since the outdegree represented by the number of packets that are going to be sent out from a specific node are not regulated at runtime, but can be afterwards, the latency naturally cannot be determined for each node.

One of the peak improvements in the Symphony structure is that it has the capacity to handle bi-directional routing. This has proven to minimize the "absolute" routing latency by 25 to 30% since it takes advantage of all the links that are originating from the node, and all the incoming links as well. Shortening the distance to the target

using neighbor of neighbor's tables (NoN) will make the transmissions from origin to destination more efficient.

Another element that has repeatedly shown to function with great efficiency is a function of Symphony known as 1-lookahead. This function, in conjunction with greedy routing, demonstrates how two nodes that are linked by a long link can routinely swap information piggybacked to their keep-alive messages. This will help to inform one another of other long-distance neighbors in the surrounding space. This has shown to reduce on average the latency from source node to destination node by around 40% (Manku, Bawa & Raghavan, 2003).

3.3 ID Distribution

ID distribution in Symphony varies since it can be done in three different ways: Regular distribution of IDs, random distribution, and balanced. All of the distribution schemes are performed over the regular ring space and every node is given an ID based on a calculation, which purports an approximate fraction of the entire ring.

3.3.1 Probability Density Function for Long-Distance Links

Symphony uses a special PDF that is defined by Manku, Bawa & Raghavan (2003) as part of the harmonic distribution function family as follows:

$$p_n(x) = \frac{1}{x \log n} \quad \text{for} \quad x \in [1/n, 1]$$

where n is the number of nodes in the network.

The harmonic PDF is used in Symphony in order to procure load balancing by creating long-distance links in a favorable configuration. To exemplify this distribution better, note Figure 3.2 below. As illustrated in the figure, the probability is high for long-distance links to be relatively short. On the other hand, the probability for long-distance links to be long remains appreciably large.

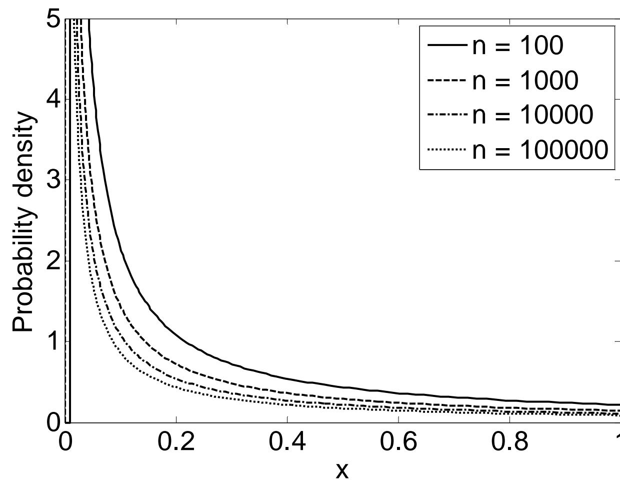


Figure 3.2 Harmonic probability density function showing 100 to 100,000 nodes

Figure 3.3 is an example histogram for 1,000 long-distance links in a network of 1,000 nodes, randomly drawn from the harmonic PDF $p_{1000}(x)$. More than 60% of the long-distance links fall into the range 0…0.1 (first two bins of the histogram). On the other hand, long-distance links in the range 0.1…1 also occur. For example, there are five long-distance links (0.5% of all links) in the range 0.95…1.

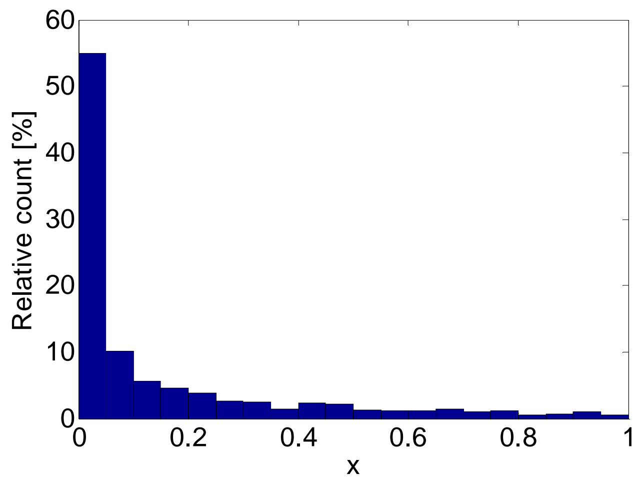


Figure 3.3 Histogram for 1,000 long-distance links in a network of 1,000 nodes

MATLAB was used to generate the graphics in Figure 3.2 and 3.3 above. The code is given here for reference:

MATLAB code “plotHarmonicPDF.m”:

```
%
% Investigation of the Harmonic Probability Density Function used
% in the Symphony protocol for distributed hash tables
%
% Plots the Harmonic Probability Density Function for various values of
% the parameter n that corresponds to the number of nodes in the
% overlay network. Generates long-distance link lengths randomly
% and plots an example histogram for the resulting link lengths.
%
% Uses "harmonicPDF.m".
%
% This is MATLAB code written by Monica H. Braunisch, 2006.
%
clear all;

% Plot the harmonic PDF
x = 10.^(-4:0.01:0)'; % x values from 0.0001 to 1
fh = figure(1);
clf reset
set(fh, 'color', 'w')
plot(x, harmonicPDF(100, x), 'k-', 'linewidth', 1.5)
```

```
hold on
plot(x, harmonicPDF(1000, x), 'k--', 'linewidth', 1.5)
plot(x, harmonicPDF(10000, x), 'k-.', 'linewidth', 1.5)
plot(x, harmonicPDF(100000, x), 'k:', 'linewidth', 1.5)
set(gca, 'fontsize', 20)
set(gca, 'ylim', [0 5])
xlabel('x')
ylabel('Probability density')
lh = legend('n = 100', 'n = 1000', 'n = 10000', 'n = 100000');
set(lh, 'fontsize', 18)
grid off

% Draw a few samples from the harmonic PDF and plot a histogram
n = 1000; % Number of participating nodes
m = n; % Number of samples
samples = exp( log(n)*(rand(m,1)-1.) ); % Ref.: Manku, 2004
edges = 0:0.05:1;
nh = histc(samples, edges);
fh = figure(2);
clf reset
set(fh, 'color', 'w')
bar(edges, nh/m*100, 'histc');
set(gca, 'fontsize', 20)
set(gca, 'xlim', [0 1])
xlabel('x')
ylabel('Relative count [%]')
```

MATLAB code “harmonicPDF.m”:

```
function pnx = harmonicPDF(n,x)
%
% Harmonic Probability Density Function used in Symphony
%
% Usage: pnx = harmonicPDF(n,x)
% Input: n: The number of nodes
%        x: Long-distance link length
% Output: pnx: The calculated probability density for x to occur
%
% Ref.: Gurmeet S. Manku, 2004.
% This is MATLAB code written by Monica H. Braunisch, 2006.
%
ind_in = find( x>=1./n & x<=1. );
pnx = zeros(size(x));
pnx(ind_in) = 1./( x(ind_in) * log(n) ); % LOG(X) is the natural log
```

3.4 Regular Distribution of IDs

The nodes in Symphony have two short links to their next neighbors, the successor and predecessor on the circumference of the ring. However, each node also maintains $k \geq 1$ long-range (long-distance) links. Long-distance links are formed by drawing a random number 0...1 from $p_n(x)$, subsequently mapped around the entire circumference of the circle.

A node initially makes a connection with the first node that lays clockwise (successor) from itself as a starting point and this will subsequently lead to the creation of a randomized network. Moreover, with the regular distribution of IDs we directed to a subject called greedy routing (Manku, S., 2004, Chapter 8). Symphony uses clockwise-greedy routing; when it implements unidirectional links, it will always route clockwise, to reduce the distance to the intended target.

Below are graphical representations of example Symphony networks simulated in PlanetSim, output in Graph Modeling Language (GML) format, and displayed in the yEd graph viewer and editor from yWorks GmbH (www.yworks.com). This uses the modified PlanetSim GML generator code shown in Appendix 2, resulting in the placement of nodes on the circle according to their actual ID. This is an extension of PlanetSim v.3.0 where nodes are always shown equally spaced even if their IDs are random. Long-distance links are shown as dashed lines, whereas the short-distance links are plotted as solid lines (details not visible for n = 1000 due to the large number of nodes). A gray disk

with unit perimeter is shown for reference in the cases $n = 10$. The variations among the figures are based on:

- Number of nodes, n
    - PlanetSim property FACTORIES_NETWORKSIZE in configuration file “symphony_dht2.properties”
- Maximum number of long-distance links, k_{max}
    - PlanetSim property SYMPHONY_MAX_LONG_DISTANCE in configuration file “symphony_dht2.properties”
- Distribution of IDs: Regular or random
    - PlanetSim property FACTORIES_NETWORKTOPOLOGY in configuration file “symphony_dht2.properties”

It is noted that short-distance links have been “fortified” by adding links to the successor of the successor of each node. In addition, the actual number of long-distance links in these PlanetSim simulations is in general less than the maximum number, since sometimes not all possible long-distance links have been established.

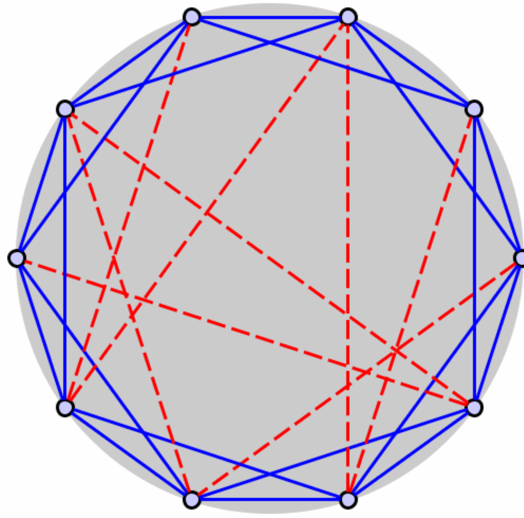


Figure 3.4 Symphony network with 10 nodes and a maximum of one long-distance link per node

Figure 3.4 above is an example of a Symphony network as simulated in PlanetSim with n = 10 nodes with regular distribution of IDs and a maximum of k_{max} = 1 long-distance link per node. It can be seen that this diagram is quite simple and compact since each node can have only one long-distance link. Links are in a unidirectional, clockwise mode.

The choice of links is explained above in Section 3.3.1 where it was discussed that the harmonic PDF will allow for a uniformly distributed number of links to be established in a way that routing is smooth and also minimizes required TCP connections.

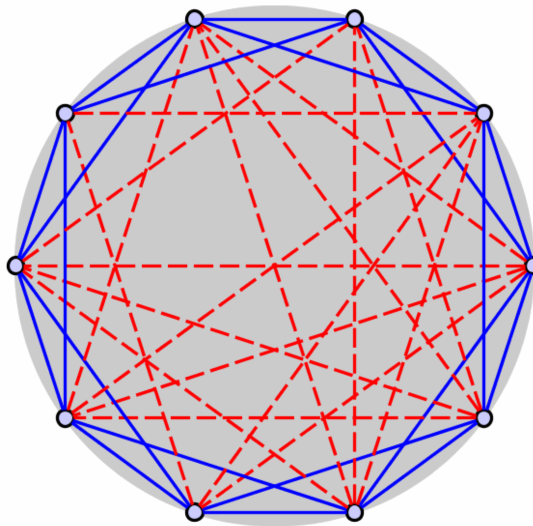


Figure 3.5 Symphony network with 10 nodes and a maximum of two long-distance links per node

Figure 3.5 is similar to Figure 3.4, but with $k_{max} = 2$: It is an example of a Symphony network as simulated in PlanetSim with $n = 10$ nodes with regular distribution of IDs and a maximum of $k_{max} = 2$ long-distance links per node. It can be seen that some of the nodes will possess either one long-distance link or two. Note that this example is only of a 10-node network with an evenly spaced distribution of nodes. Also, for this example the links are unidirectional and are used in the clockwise direction.

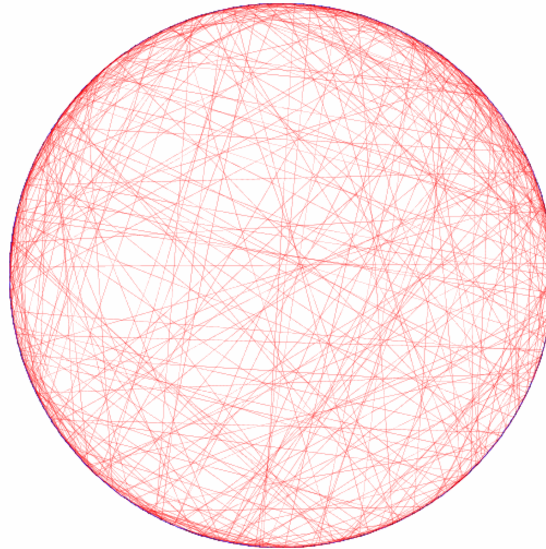


Figure 3.6 Network with 1,000 nodes with regular distribution and a maximum of one long-distance link per node

The Symphony network in Figure 3.6 resembles structurally the one in Figure 3.4, however, here the network size is comparably larger, consisting of $n = 1000$ nodes. This case was simulated using PlanetSim as well, and is another example of the regular distribution of IDs with a maximum of $k_{max} = 1$ long-distance link per node. Therefore, it can be assumed that in such a network with 1,000 nodes, each node will hold first of all its set of short-distance links, one for each of its adjoining neighbors, as well as the reinforced short-distance connection to its NoN, and one long-distance link that would fall, with about 60% probability, within the range 0 to 0.1, with the remaining 40% of nodes having a much longer distance range as shown in Figure 3.2.

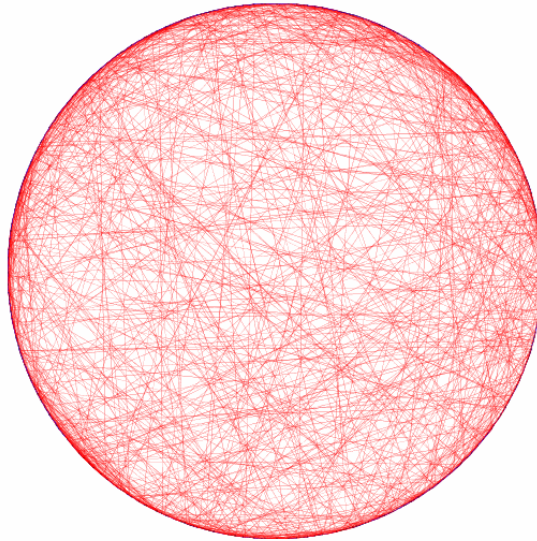


Figure 3.7 Symphony network with 1,000 nodes and a maximum of two long-distance links per node

The Symphony network in Figure 3.7 resembles Figure 3.6 in that it holds initially some of the same properties such as forming a network with 1,000 nodes. However, the principal difference here is that the maximum number of long-distance links per node is $k_{max} = 2$. As before, this graphic was produced with PlanetSim. Again, the way that nodes are distributed here is all uniform since this is how regular distribution of nodes is defined.

3.5 Random Distribution of IDs

With random distribution of IDs, every node needs first to try to acquire an approximation of the total number of nodes in the entire Symphony network. This is done by an estimation protocol (Section 3.5.1). Later, once the network size has been

determined by measuring three arcs and getting an approximation, then links can be established.

3.5.1 Estimation Protocol

In order to join a random Symphony network an arriving node needs to know one other node. Then it will hash and produce its own ID within the range of 0...1. Next, it will run the estimation protocol, which takes the estimate of three different arcs between nodes (Figure 3.8).

These arcs provide an estimate of the distance between the nodes. This, at no additional expense, will provide the new node a space in the ring that is at an approximate equal distance from the others. This estimation is performed by a node by using the length of an arc and that of its neighboring arcs. The new information is then piggybacked to any of the lookups or requests. This will enable the routing tables to update their information with little cost.

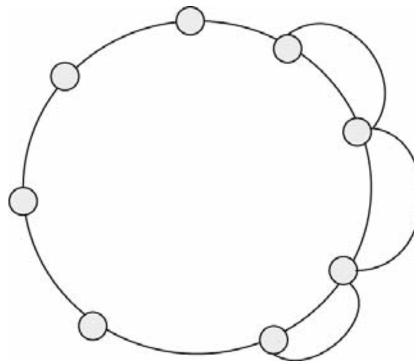


Figure 3.8 Symphony's estimation protocol

In summary, as shown in Figure 3.8, the estimation protocol will take the three arcs and estimate the approximate distance in the entire network in order to place new nodes.

3.5.2 Link Establishment

In order for a new node to be incorporated in the network, first the estimation protocol produces a result, which amounts to the estimation of n. The nodes all maintain two short links to their predecessor and successor nodes as well as each node having $k \geq 1$ long-distance links which can vary. Each link will start by getting a random number and subsequently contacting the corresponding key manager in a clockwise direction; it then sets up a connection with this managing node.

There is a limit in the number of in-degree links, also referred to as the incoming links, of $2k$ per node. Once this number is attained, then the next time that another node tries to link itself to this particular node it is rejected.

3.5.3 Smooth Re-Linking Protocol

A so-called re-linking protocol is used at every node's routing table, so that there exists a way to indicate an estimate of where each long-distance link made a connection. Once the estimation protocol is run, this information is piggybacked in order to update the routing tables. It is a good scheme, since if many nodes arrive periodically, then the nodes that need to re-link at any point would be one at a time.

3.5.4 Graphical Examples

In order to deepen the reader's understanding of how random distribution of IDs works in Symphony, a visual example will be helpful in order to distinguish it from how regular distribution of IDs appears, such as was shown in Section 3.4.

In these examples, again, the networks range in size from 10 nodes to 1,000 nodes, and the number of long-distance links is varied. For simplicity's sake, the number of long-range links was kept at a moderate size and varies from one to two links per node. These plots again were generated using the PlanetSim simulator with output in GML format and by displaying in the yEd graph viewer from yWorks GmbH (www.yworks.com).

Figure 3.9 plotted below is another example of a Symphony network that was simulated using PlanetSim and contains only 10 nodes with random distribution of the IDs. The long-range links for this example are determined to have a maximum of $k_{max} = 1$ long-distance link per node.

Figure 3.10 is similar to Figure 3.9, but with $k_{max} = 2$ long-distance links per node. This network also utilizes the random distribution of IDs and contains only 10 nodes. It is another example simulated using PlanetSim.

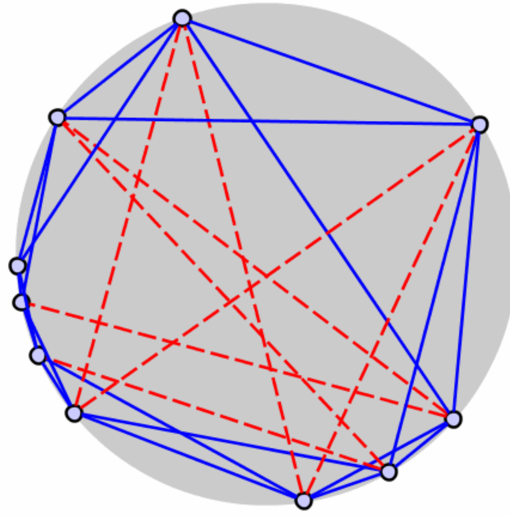


Figure 3.9 Symphony network with 10 nodes randomly distributed and a maximum of one long-distance link per node

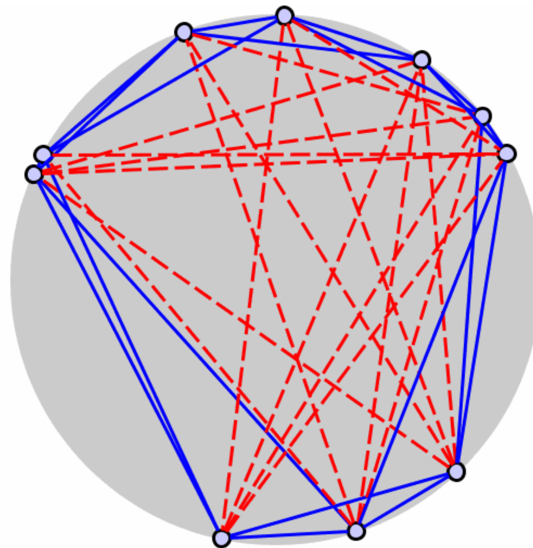


Figure 3.10 Ten-node Symphony network with a maximum of two long-distance links per node

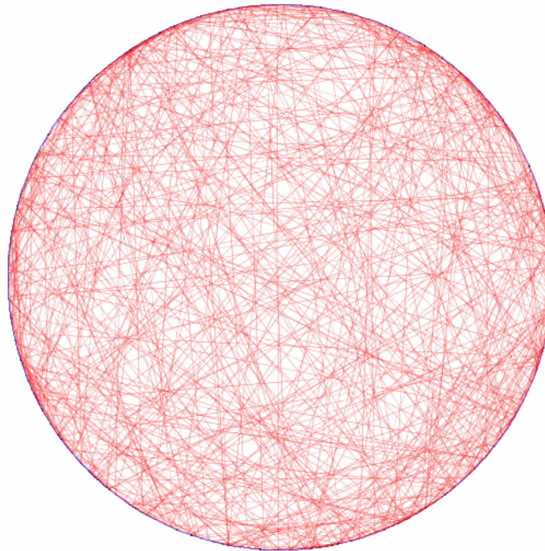


Figure 3.11 Symphony network with 1,000 nodes and a maximum of one long-distance link per node

Again, the Symphony network in Figure 3.11 looks much different since it contains 1,000 nodes. The network was simulated using PlanetSim with n = 1000 nodes, picked and distributed evenly and at random, with a maximum of only k_{max} = 1 long-distance link per node.

Figure 3.12 is again resembling Figure 3.11, however, the difference is that this Symphony network with 1,000 randomly distributed nodes holds a maximum of k_{max} = 2 long-distance links per node. Visually, this network looks quite different connection-wise. Since it is much more complex, it is hard to discern where any of the nodes are located exactly. They are nevertheless picked and distributed, as for the other random examples, within the range of 0 to 1 along the unit perimeter of the circle.

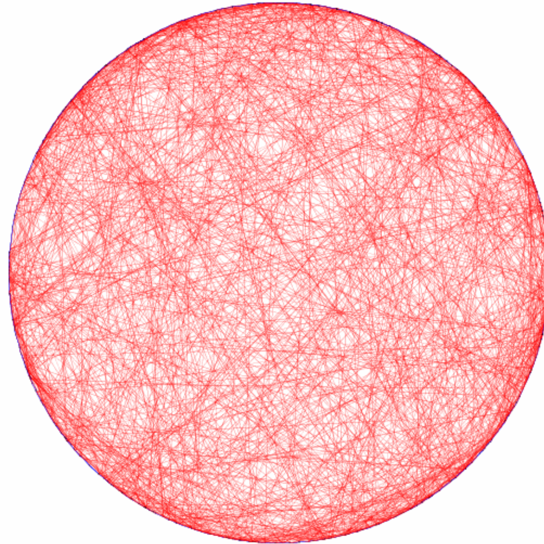


Figure 3.12 Symphony network with 100 nodes and a maximum of two long-distance links per node

3.6 Balanced Distribution of IDs

Balanced distribution of IDs is special as each node is associated with a leaf of a binary tree (Manku, 2004). The maximum depth of this tree is associated with the network size and leaves can be at three different levels around depth $[\log_2 n]$, where $[x]$ denotes the nearest integer approximation to x and n is the network size. In each of the leaf levels, nodes are spread out uniformly, or distributed regularly. Then, to map the nodes onto the unit perimeter circle, a binary node ID is obtained by traversing the tree and tracking 0s and 1s according to the branching encountered. This binary node ID is then normalized to a fractional value between 0 and 1, falling on the unit perimeter circle.

With balanced distribution of IDs, the number of long-range links can be fixed or made variable dependent on network size.

3.7 Node Joins

Node joins in Symphony are conceptually similar to Chord, as discussed in Section 2.6.1. However, it is important to remember that the Symphony ring is defined as a continuous interval ranging from 0 to 1. Node IDs are real numbers, not integers as in Chord. Also, the node IDs are usually not associated with node IP addresses and port numbers. Thus, the routing tables in Symphony have a fundamentally different data structure as compared to Chord and, clearly, this difference needs to be taken into account during node joins.

As for the routing tables themselves, two short-distance links (which are reinforced) have to be established with the direct neighbors: the successor and predecessor nodes. The long-distance links do not need to be reinforced. However, to establish the long-distance links it is necessary to generate a random number according to the harmonic PDF, using code similar to the one displayed in Section 3.3.1.

When a node joins, it initially contacts the manager that is next to it in the clockwise direction and sets up a link with the manager that has recognized the new node. This is consistent with the concept of clockwise-greedy and unidirectional routing.

3.8 Routing Table Structure

In order to understand how Symphony performs its lookups it is necessary to explain how its routing table is structured (Manku, Bawa & Raghavan, 2003). The number of long-distance links, an input parameter for the protocol, can vary. The short-distance links, however, always remain the same and have direct connections to any node's adjacent neighbors. As for the reinforced short-distance links, they will provide information regarding the neighbor's neighbor and thus improve routing efficacy by holding some additional amount of routing information. Figure 3.13 below shows a very simple example of a Symphony network with a total number of nodes of n = 8. Table 3.1 lists and categorizes the different kinds of links for clarity.

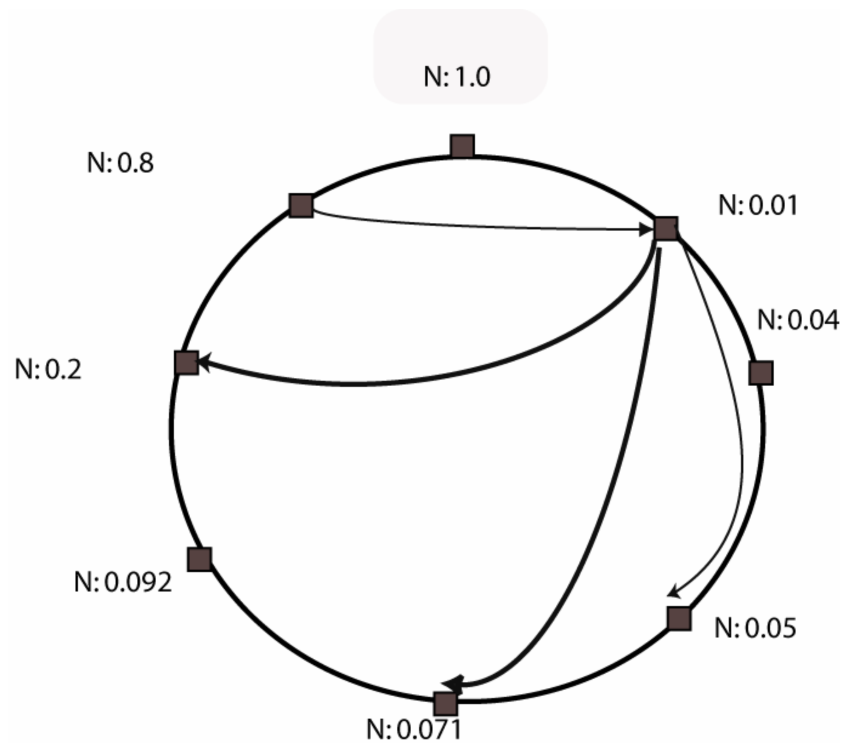


Figure 3.13 Example of a Symphony routing table

Starting point is at node N:0.01. Following is the distribution of long-distance links as well as the short-distance links and reinforced short-distance links.				
Starting Node	Short-Distance Links		Reinforced Short-Distance Links with NoN Going Clockwise	Long-Distance Links with Harmonic Distribution
	Predecessor	Successor		
N:0.01	N:1.0	N:0.04	N:0.05	N:0.071; N:0.2

Table 3.1 Components needed for routing to take place in Figure 3.13

As noted, Figure 3.13 above is a representation of how a small routing table would appear. This is an example of a regular, greedy-clockwise routing method with unidirectional links. Consider node N:0.01 as the starting point. The short-distance links are similar as in Chord but do not have any specific names. For the example, the associated nodes were called successor and predecessor for simplicity's sake. However, usually these nodes are referred to as immediate neighbors. As shown, the "predecessor" of this particular node N:0.01 is N:1.0. Then the "successor" node would be N:0.04, which is the immediate node in the clockwise direction.

Subsequently, there are also the short-distance reinforcing links, which utilize the previously mentioned NoN concept. Information is piggybacked in order to keep updates regarding node departures or joins accurate, taking advantage of keep-alive messages and pings to prevent additional overhead or updates from stalling the system's productivity. Here, the reinforced or bolstered short-distance links using NoN are also in a clockwise direction. In the example, for node N:0.1 the neighbor of its immediate neighbor would

be node N:0.5 that will also receive updates of routing table changes from N:0.01, again through keep-alive messages and pings.

Lastly, the long-distance links are produced by drawing from the harmonic PDF and fall within the distance range of 0 to 0.1 in circa 60% of all the long-distance connections, with the other 40% being longer distance links (Section 3.3.1). Considering that the long-distance links are not reinforced (backed up) and that they can range in number of connections to produce variety and heterogeneity, they can depart at any given moment. However, these long-distance links are very important and at the core of Symphony; they are the backbone of the protocol. Each node can possess a few long-distance links with the condition that their number is $k \geq 1$. This means that each node must have at least one long-distance link, with a length determined by the harmonic PDF. In this example, node N:0.01 has two long-distance links, to N:0.071 and N:0.2. The long-distance links enable routing of messages or requests to far away places on the ring based on the small world phenomenon.

3.9 Node Departures

One of the principal aspects of a well-designed and maintained P2P is to be able to handle node departures in a timely manner and not bring the system down. Most of the node departures can happen simultaneously and in large proportions, often unannounced. If the system is not equipped to manage such events, catastrophic failure will result.

With Symphony, however, with its random and decentralized methods, the protocol is simple to use and low cost, since the messages exchanged for the updating of the routing tables are piggybacked to keep-alives and pings.

Also, when nodes join the system or leave, the re-assignment of IDs is kept to a minimum. When a node decides to leave either voluntarily or abruptly, this departure will make only one other node update or change its ID. Manku (2004) shows that the cost for joins and leaves in the network due to the required number of message exchanges grows only logarithmically with network size.

When a certain node *n* departs the network, an exchange of IDs and reassignment takes place, ensuring that they are remapped to the corresponding node. All the links that are either coming to or leaving the departing node *n* are instantly severed from its long-range neighbors. Then the nodes that were at one point linked to the newly departed node take notice that their link to *n* is no longer functional. The nodes whose links have just been severed will take immediate action and reestablish a connection with the other nodes. The next-door neighbors or predecessor and successor of this departed node *n* will also evaluate the situation and begin to establish new links with the adjacent neighbors. The successor of departed *n* begins to run the estimation protocol over the next three neighboring links to update its own tables, as well as to determine an estimation of the new perimeter.

In typically dynamic large-scale P2P networks, it is a natural requirement for them to be able to scale quickly reaching up to millions of nodes in a short period of time. Since there are no central servers to regulate traffic flow, frequently hundreds of nodes

can simultaneously join a network as well as leave it abruptly without any notification. That is an important reason of why a well-designed and efficient maintenance protocol needs to form part of the existing DHT to enable updates to routing tables and to manage these large amounts of traffic. The constant flow of arrivals and departures in a large-scale distributed network was coined churn (Rhea, Geels, Kubiatowicz & Roscoe, 2004). These churn events can also be associated with flash crowding meaning that large numbers of nodes may leave and join at the same time. It is indispensable for P2P routing protocols to be devised in a way that will deal with churn and massive node failures and still keep the infrastructure of the system alive and updated. Both Chord and Symphony have demonstrated in being able to cope with such stresses in simulation experiments and can endure massive node failures, high degrees of churn, as well as accommodate properly newly adhered nodes.

3.10 1-Lookahead and Greedy Routing

The so-called 1-lookahead routing scheme is quite important for randomized networks. The theory is extensive and is addressed as NoN by Manku, Naor and Wieder (2004). They show that with the attribute of 1-lookahead the median number of hops drops significantly.

Greedy routing with 1-lookahead is a fundamental part of Symphony as well as of other randomized networks. The nodes that participate in randomized networks generate random bits, which are different from centralized deterministic networks that typically associate information with worldwide scale schemes such as lists and databases. The

centralized network protocol designers initially presumed that randomly generated bits would not be able to handle large-scale distributed elements because the associations based on nodes and keys would not be uniformly distributed or accessible with a small number of hops.

Later, with the introduction of these randomized schemes, in fact the opposite proved to be true. The nodes could be distributed uniformly and at random, in the case of Symphony, as well as other distributed networks, and the keys were too.

Latency times were compressed also. One of the experiments by Manku, Naor and Wieder (2004) shows that when NoN is implemented this can reduce the latency from source to destination consistently by around 40%.

These new decentralized lookup and routing methods actually permit the participating nodes to be enabled to route information without direct knowledge of all the other nodes in the system (Manku, Naor and Wieder, 2004).

Greedy Routing with 1-Lookahead is formally described as a way to take the routing tables from the neighbor's neighbors and with this routing information come up with improved routing schemes to minimize path length. Each of the nodes has a compressed list of the neighbor's neighbors so that information will be routed in a greedy manner to the closest neighbor of neighbor, requiring only two hops.

Another of the advantages with greedy routing is that fault tolerance to node failure is high and problems can be corrected and self-repaired rapidly. Security is bolstered as well, since it is hard to cause collisions when random numbers are generated,

even with brute force. Those are two powerful advantages over other distributed systems (Manku, Naor & Wieder, 2004).

3.11 Initial Conclusions

Some of the drawbacks that pertain to Symphony, *e.g.*, estimating the distance between nodes by sampling the ranges between a few nodes to determine the overall size of the Symphony network, at times leading to clustering, could be rectified by modifying the protocol as suggested in a similar DHT named Mercury (Bharambe, Agrawal & Seshan, 2004). In Mercury, they propose a novel idea based on histograms to be able to make load balancing more effective as well as to prevent clustering in prevalent areas of the ring. Bharambe, Agrawal & Seshan (2004) implement in Mercury a random histogram maintenance technique that takes traffic samples and then can determine new routes to keep the traffic load spread out more evenly. This is a problem in Symphony since it only takes the average range between selected nodes and sometimes the distribution of the nodes is not even. It would also be interesting to implement the Mercury technique in conjunction with Chord.

One of the notable differences between Chord and Symphony is that the long-distance links or short-distance links in Symphony are not set to a limit. They can fluctuate in number to adjust to variations in node joins or departures, thus producing less unnecessary traffic and increased “free-flow”.

Another distinction from Chord or other protocols is how Symphony does not “bolster” or backup its long-distance links. This property decreases traffic as well,

produces less pings and keep-alive messages, also promoting less TCP connections to nodes. This increments the capacity to handle even more node joins and departures, since as aforementioned there is more bandwidth for more important operations.

Symphony also holds the advantage of being able to route in two directions. This bi-directional routing has proven to reduce the overall latency by 25–30% (Manku, Bawa & Raghavan, 2003), by using both incoming and outgoing links and piggybacking some of the routing information in either direction to help update neighbor's tables. This cannot be done in Chord with the use of stabilize(), running every 30 seconds; in Symphony, however, updates can take place whenever needed.

As described by the authors of Symphony, it has a "tuning knob" (in the number of long-distance links) that can self-adjust to manage the load and number of links, thus rendering it with much more flexibility than other protocols that have "limited" or very structured tables that cannot adjust easily to changes in a dynamic system.

Chapter 4 PlanetSim

"No amount of experimentation can ever prove me right;
a single experiment can prove me wrong."
Albert Einstein

The importance of network simulations cannot be overstated; they are necessary in order to provide a quantitative metric-based evaluation and ensure efficient performance before deploying protocols in real world applications.

This Chapter covers a brief comparison of distinct network and overlay protocol simulators and why PlanetSim was chosen to be extended. PlanetSim is a powerful, modular, Java based P2P simulator that was designed to test both the Chord and Symphony protocols. During an extensive search to analyze the various properties of Chord and Symphony, PlanetSim came into light.

The following sections cover P2P simulators in general and themes that are touched by the PlanetSim code extensions: The SHA-2 family and hashing with IPv6 (Manku, Bawa & Raghavan, 2003; Hinden & Deering, 2006).

4.1 Comparison of P2P Simulators

Some experts in the P2P area claim that simulations and emulations will sacrifice the realistic scenario of true, large-scale systems (Ball, 1996; Darlagiannis, Liebau, Mauthe & Steinmetz, 2004). This allegation tries to declare that, since any type of

simulated environment cannot possibly reconstruct the randomness and improbability of the way that real-world traffic patterns are produced, the simulation results will only conduce to approximations. However, even if the simulation/emulation results are not absolutely precise and are a way to calculate through estimations the way that real-world distributed P2Ps behave, it is better to conduct any amount of testing rather than none at all.

It is very difficult to test new or modified protocols on a continuously running system, thus arises the need to create controlled environments to test them before implementing in a real world situation. Simulation of protocols or applications seems to be a mandatory call to test their behavior and observe if they will indeed succeed or fail.

There are two forms of simulators:

1. Network simulators that are usually narrow in scope in that they test at the packet level factors such as real number of TCP connections or queuing, but are expensive and cannot realistically scale well.
2. Overlay simulators whose purpose is to assess P2P protocols regarding the way that routing, mapping, hashing, and ID assignments take place and will generally not take into account the underlying packet level and physical layer.

Once such network simulator that tests packet-level metrics is PLP2P (He *et al.*, 2003). The PLP2P network simulator focuses on packet level traffic. It does not take into account the underlying physical layer. It offers a way to test different features in a large P2P such as running on top of a socket interface but will not consider the low-level

packet details. It can however be used in conjunction with other packet level simulators and this grants it some freedom. PLP2P has been used with the Gnutella framework to evaluate simulated performance of this protocol. It was devised to be extensible and be able to add other features depending on what needs to be tested. It has three separate levels that can be extended to test different properties: a PeerApp, a Socket Adaptation layer as well as a PeerAgent. Some of the published experiments test bandwidth at the link layer but do not consider packet level details. This shows that the simulator does not scale well but if it is run on distinct machines simultaneously then the results tend to improve (Sam, 2004).

The SimP2P simulator (Kant & Iver, 2003) helps analyze ad-hoc P2P networks. One of its properties is that its prototype design is similar to the way that Gnutella was constructed. Its graph format and random, not even, distribution of nodes make it very restricted in trying to analyze features such as node degrees and how to reach other nodes. In addition, it is not equipped to handle large degrees of node queries.

Peersim (Jelasity *et al.*, 2003) is another simulator that can test scalability up to millions of nodes that depart and join with churn. This simulator was designed to be able to handle highly dynamic networks and encompasses a wide variety of components that allow it to be flexible as well as extensible. It was developed for the BISON project and programmed in the Java language. Peersim is founded on a cycle engine that permits it to scale by not taking into account some of the details in the TCP or UDP layer. This is an event-generated engine that is authentic and can be used along with cycle-based protocols.

3LS (Ting & Deters, 2004) is yet another discrete event simulator that incorporates a time-stepped central clock. It is divided into three distinct sections: the network model, protocol prototype and the user model. The network model considers the distance in steps between the nodes by using a 2-D matrix. The protocol prototype incorporates the P2P protocol that is going to undergo testing and evaluation. The user model simulates the protocol prototype and outputs results. However, there are limitations in that it cannot handle large-scale simulations and is mostly employed to test smaller systems.

P2Psim (Gil *et al.*, 2004) from MIT is a simulator that has been designed to test various peer-to-peer protocols including Chord, Tapestry, Kademlia, but not Symphony. P2Psim is a discrete event simulator that is used to evaluate and better understand P2P protocols. It helps to examine, by comparing many protocols, how they scale, their latency in testing, and robustness. It is designed in such a way that it is easy to understand and implement testing of diverse protocols by using pseudo-code examples. There are drawbacks with the simulator even if it is easy to use, such that it has no interface readily available to incorporate other applications and also that it is hard to extend the code.

The Narses simulator (Baker & Giuli, 2002) does not take into consideration the packet level since it does not want to have the simulations bogged down by too much packet overhead. It is a flow-based simulator and possesses a solid reputation for enabling modeling of different protocols with precision. This simulator falls in the category of high-level packet simulators and ones that are more experimental. It is hard to employ due to its complexity.

4.2 PlanetSim

There are numerous other packet-level simulators as well as overlay simulators, but up to now, there has not been any that can be extended easily with a modular structure, written in Java, or that exclusively provides overlay simulations to test protocols.

PlanetSim is an innovative Java based P2P simulator that tests Chord and Symphony. Its mainframe is modular in structure and can easily be extended to include other tests without penalizing the entire infrastructure. This is the reason why, in this thesis project, PlanetSim was chosen to conduct all the simulations for Symphony and Chord as well as for testing new and novel ideas such as extending the hash values from the SHA-1 family to the SHA-2 family.

PlanetSim is well described in every aspect, which sets it aside from other simulators, because it explains in detail how to use it and is designed with well-studied devised features. It is structured to be extensible to test and develop other algorithms as well as to adhere to its structure other types of application. It also has a feature to facilitate the transition from simulation code to real-world code. One such place that is used by various institutions and individuals to test their protocols is PlanetLab (www.planet-lab.org). PlanetLab is a worldwide simulation test-bed for large-scale distributed structures, applications, and for theoretical implementations that need to be examined before deploying them.

One of the primordial highlights pertaining to PlanetSim is that it possesses a distinct modular design, which allows for the insertion of extensions and code expansion, which is easy to manage and well documented. With this foundation, it is doable to create new classes or modify any of the parameters that are already part of the PlanetSim core and test new theories or algorithms. Due to this modular design, it is a powerful simulation tool that renders flexibility to the coder and gives a way to test and evaluate protocols without the need to re-design the core.

The PlanetSim simulator is structured in three different parts: the network interface, the node level, also addressed as the overlay layer, and the application layer.

The network layer, the most important part of the simulator, is where all the routing takes place from origin to destination. At this layer, the nodes are created and connected using different types of topological structures. It is in these modules where the stabilization process is performed and where nodes join and depart.

In the overlay layer is where all the rest of the operations takes place. This is also divided into various modules that mediate the entire sequence of events. It works at the node interface as well as determines where the messages are routed and performs other operations concerning the IDs.

PlanetSim has a structure that introduces a common application programming interface (API) that allows for the extension by other overlay protocols. This common API (CAPI) allows for other applications such as scalable group multicast/anycast (CAST) or decentralized object location and routing (DOLR) in the application layer of

the PlanetSim structure to run. Using the CAPI makes it possible for these overlays to be incorporated and allows for different types of distributed algorithms to be tested.

The authors of PlanetSim, Garcia *et al.* (2005), made use of some of the features from Free Pastry's universal API structure. This renders flexibility by allowing many other algorithms a chance to be tested before deployment.

Overall, PlanetSim, due to its modular, well-documented design and structure, has endless possibilities and can be extended in many ways. For comprehensive information on the use of PlanetSim, please refer to the PlanetSim User and Developer tutorial (ants.etse.urv.es/planetsim/PlanetSim_tutorial.pdf).

4.3 SHA-1 & SHA-2 Families

The SHA families have been developed over several years starting with SHA-0 and MD5, now considered obsolete due to security concerns (Schneier, 2005). Most of cryptographic public keys as well as hash codes still utilize SHA-1. However, this will change in the future as explained in Section 1.5, as scientists from the University of Shandong in China recently proved that they could create collisions with a smaller effort than originally conceived.

With this new knowledge, it is safe to assume that many cryptographic hashes are going to be modified and will begin using a new family of hashes, mainly SHA-2. This family has not yet shown to be breakable in any way since the message digests are much longer, and it is extremely hard to reproduce any of the hashed message digests and thus collisions are not a concern.

The SHA-2 family consists of a number of different algorithms: SHA-224, SHA-256, SHA-384, and SHA-512. They all generate much longer message digests than with SHA-1 that produces a digest of 160 bits, which is broken into five blocks of 32 bits for the hashing scheme.

SHA-1 and SHA-2 are considered secure government standards as part of the Federal Information Processing Standards also known as FIPS. These different algorithms are all iterative in nature and are structured in a way that they take a message of variable length and output a compressed version of the message of fixed length known as the message digest. The various SHA-2 algorithms differ in the length of the message digest, *e.g.*, SHA-512 outputs a digest of 512 bits as can be noted in Table 4.1 below.

Algorithm	Message Size (bits)	Block Size (bits)	Word Size (bits)	Message Digest Size (bits)	Security (bits)
SHA-1	$< 2^{64}$	512	32	160	80
SHA-224	$< 2^{64}$	512	32	224	112
SHA-256	$< 2^{64}$	512	32	256	128
SHA-384	$< 2^{128}$	1024	64	384	192
SHA-512	$< 2^{128}$	1024	64	512	256

Table 4.1 Different SHA algorithms

The SHA algorithms are referred to as secure because it is highly improbable for two different messages to lead to the same message digest, especially for any in the SHA-2 family, thus preserving the integrity of the hash value and making it secure and unique in value.

All of the algorithms are divided into distinct steps:

1. Preprocessing takes the message that is going to be hashed and begins by padding and then dividing it into m-bit blocks.
2. Preprocessed m-bit message blocks are hashed. A "message schedule" is created and functions and operations are applied in an iterative way, finally outputting a hash value.
3. The obtained hash values are used to build the final message digest.

The algorithms that are used for hashing schemes are mainly different in the number of bits used. For example, SHA-1 delivers a message digest of 160 bits; the SHA-2 algorithms generate longer message digests, which provides enhanced security.

For all of these algorithms there are some rudimentary properties:

- Maximum message size in number of bits
- Block size in number of bits
- Word size in number of bits
- Message digest size in number of bits
- Security, improved with increased number of bits

As for message padding and parsing, the message will first be treated as a bit string, and the message length is the number of bits in the message. If the number of bits is a multiple of four, then the message could be shown directly in hex. The principal idea

of padding is to make the length of a padded message a multiple of 512 bits for SHA-224 and SHA-256 or 1024 bits for SHA-384 and SHA-512 (Table 4.1).

Utilizing PlanetSim, in Chapter 5 the results from having extended Chord and Symphony with SHA-2 are plotted and compared. With Symphony, hashed values are always mapped into the range of real numbers from 0 to 1 (represented as a “double” in Java). In Chord, hashed values are kept as integers but can be truncated according to the corresponding input parameter of the PlanetSim simulator.

4.4 Hashing with IPv6

The traditional Internet Protocol Version 4 (IPv4) is at some point running out of addresses even if they are partitioned using subnet masks. The majority of the IPv4 addresses are localized within the United States. The world is expanding and growing, thus providing new addresses is not only important but also inevitable. Since there were so many complexities, the Internet Engineering Task Force (IETF) started considering making changes to IPv4 and made the decision to solicit proposals on how to make improvements and modifications with a new protocol that could be devised for future implementation.

The decision to begin a shift from IPv4 to the IPv6 addressing scheme began taking place in the 1990s. The hope was that the new protocol could be formed before IP addresses would run out. One of the first requirements for this new version was the ability to support billions of hosts, also making the protocol much simpler by deleting some fields that were never properly used, somehow to find a way to decrement the size

of tables used in routing, and to enable the routers to process packets faster. These are just some of the rudiments that one needed to focus on in order to better the Internet Protocol. Other aspects relate to improvements by expanding the Quality of Service (QoS) and make it more applicable for real-time flows, and increasing security with the addition of optional extension headers. Finally, backward compatibility with IPv4 was an important consideration.

At this point in time, several working groups began to form, in order for each to take part in the transition and make this newly proposed protocol real. The IETF took action in following this recommendation and giving its approval for the transformation in 1994. One of the groups was called IPng WG or the IPng Working Group where "ng" stands for next generation. In addition, the Address Auto Configuration Working Group and an IPng Transition Working Group helped understand using Version 4 addresses in Version 6 environments. The Internet Engineering Steering Group (IESG) was to review and monitor all of the IETF activities with regards to the limitations and implications of IPng, by amending and revising old standards (Jain, 1997).

<table>
<tr><td>Version (4 bits)</td><td colspan="2">Priority (4 bits)</td><td colspan="2">Flow Label (24 bits)</td></tr>
<tr><td colspan="2">Payload Length (16 bits)</td><td colspan="2">Next Header (8 bits)</td><td>Hop Limit (8 bits)</td></tr>
<tr><td colspan="5">Source Address (16 bytes or 128 bits)</td></tr>
<tr><td colspan="5">Destination Address (16 bytes or 128 bits)</td></tr>
<tr><td colspan="5">Payload</td></tr>
</table>

Table 4.2 IPv6 header and fields

The architecture of the IPv6 structure is similar to IPv4 but has additions in security and can support anycast. In addition, the addresses are much longer.

The transition from IPv4 to IPv6 implies a quite drastic switch from 32-bit to 128-bit addresses, necessary to support world expansion and avoid future limitations. IPv6 has a fixed header length. Also, the IPv4 Don't Fragment fields were eliminated since IPv6 already predetermined that routers as well as hosts need to support packets that are 576 bytes long. In the case that a packet is too long then the router along the path sends a message back to the host and tells that host to fragment the packet. This is simply because the routers will no longer fragment, but will only provide the service to route packets. Consequently, the host needs to take care of fragmentation.

IPv6 addresses use fundamentally the same portrayal as version 4 except that the hexadecimal values have been extended to eight domains of 16 bits each, in contrast with containing four parts with 8 bits apiece for IPv4. The version 6 addresses are written as in the notation "x:x:x:x:x:x:x:x" with each x representing 16 bits. An example of an address would look something like this: 3198:FFA4:1893:1958:ABCF:E8B9:1332:F954, or others will have many zeros depending on the distribution scheme, for example: A984:0:0:0:4:A200:C45:1232 (leading zeros dropped). In this case, an address having zeros in consecutive fields can be condensed and the train of zeros is replaced by a simple double colon. For example, the address 984:0:0:0:4:200:C45:32 could be written more compactly as 984::4:200:C45:32.

If there is an IPv4 address that is still in use but is being actively incorporated to version 6, then the last two hexadecimal fields can be used for the "regular" version 4

address by separating its fields with dots instead of colons. An instance of this would look like this: 0:0:0:0:0:FFF:128.5.33.8 and compressed to ::FFF:128.5.33.8.

As of today, the allocation of certain addresses has already been predetermined and others are unassigned for future use as shown in Table 4.3 below. In the table, the entries of the third column add up to one as it should be. For an updated version of IPv6 refer to RFC 4291.

Allocation	**Prefix (binary)**	**Fraction of Address Space**
Reserved	0000 0000	1/256
Unassigned	0000 0001	1/256
Reserved for NSAP Allocation	0000 001	1/128
Reserved for IPX Allocation	0000 010	1/128
Unassigned	0000 011	1/128
Unassigned	0000 1	1/32
Unassigned	0001	1/16
Unassigned	001	1/8
Provider-Based Unicast Address	010	1/8
Unassigned	011	1/8

(continued on next page)
(continued from previous page)

Reserved for Geographic-Based Unicast Addresses	100	1/8
Unassigned	101	1/8
Unassigned	110	1/8
Unassigned	1110	1/16
Unassigned	1111 0	1/32
Unassigned	1111 10	1/64
Unassigned	1111 110	1/128
Unassigned	1111 1110 0	1/512
Link Local Use Addresses	1111 1110 10	1/1024
Site Local Use Addresses	1111 1110 11	1/1024
Multicast Addresses	1111 1111	1/256

Table 4.3 IP address space from RFC 1884 showing the initial distribution of IP addresses

As part of the project, one of the contributions is to create hash values from IPv6 addresses. In the near future, this will need to be done for systems such as Chord because

the hash values for the node IDs are exclusively produced by hashing the IP address along with the port number. This shows interesting results, in extension of the PlanetSim simulator code.

For more on all the new Requests for Comments (RFCs) that are being added to enhance IPv6 functionality either refer to the RFC Editor (www.rfc-editor.org) or to "IPv6, Internet Protocol Version 6" (www.networksorcery.com/enp/protocol/ipv6.htm) where they also have an extensive list of all the obsolete RFCs pertaining to IPv6.

Chapter 5 Results, Conclusions and Future Directions

This chapter shows the various simulation results that were reached extending the PlanetSim code. The final part will discuss further conclusions for both Chord and Symphony, and future directions for these protocols as well as a few themes that could be followed up on in the future.

5.1 Results Obtained with Simulations

Simulation results are generated using the extended version of PlanetSim v.3.0, as created in this thesis project. Run times are reported as obtained on a Pentium 4 system (3.04 GHz and 512 MB) under Windows XP SP2 running Sun Java v.1.5. Gathering of information beyond simulation time and simulation steps (see the tabulated results in Appendix 1), such as the average number of hops, is part of the future PlanetSim project and currently not available (Garcia *et al.*, 2005).

In each case, the following phases of an individual simulation are examined: Network creation consisting of node generation and initial stabilization of the network, insertion of 100 keys generated by hashing of the entries in the text file \bin\data\BluesMan.txt (part of the PlanetSim distribution) and looking up of the same. This utilizes the DHT test that can be launched by running \bin\dht2.DHTTest.bat (see Appendix 2 for details on how to do that).

The results in Figures 5.1 to 5.3 show the scaling of Chord and Symphony for increasing network size. Hash algorithm is SHA-1. Chord IDs are 32 bits long and the maximum number of long distance links for Symphony is two.

For both Chord and Symphony, the network creation phase (Figure 5.1) takes a long time, with simulation time approximately quadrupling for each doubling of the number of nodes. This limits the size of the network that can be simulated, especially for Chord. Key insertions and key lookups are quite fast, with simulation time about linear with number of nodes (simulation time doubles when the number of nodes doubles). In all cases, the Symphony simulations run significantly faster than with Chord.

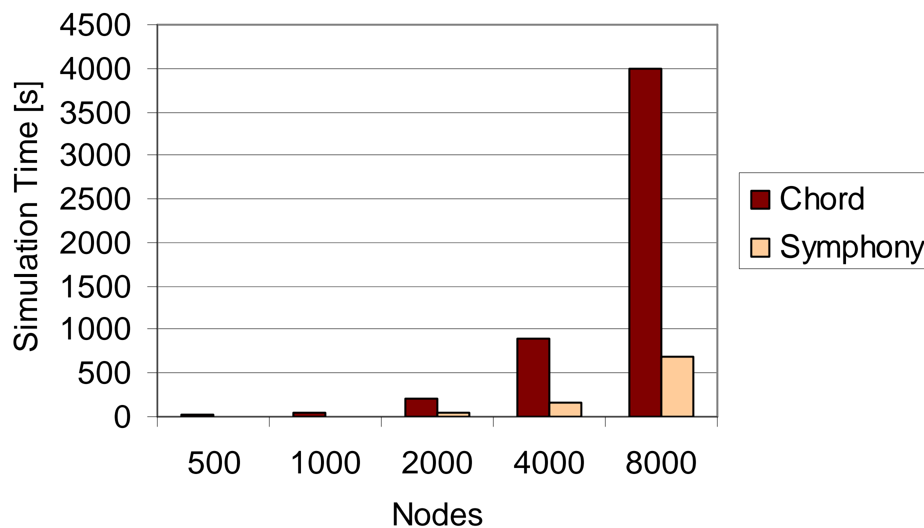


Figure 5.1 Chord and Symphony for increasing network size. Network creation with regular distribution of nodes.

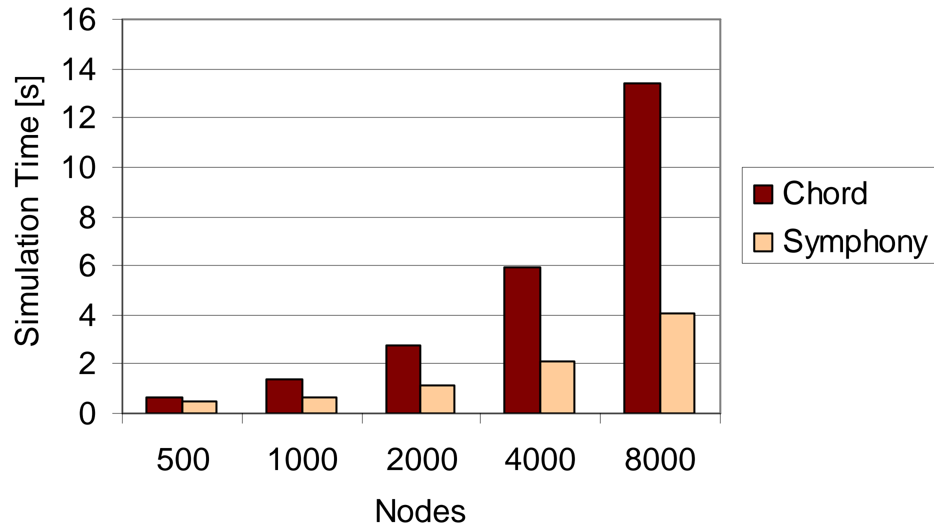


Figure 5.2 Chord and Symphony for increasing network size. Key insertions with regular distribution of nodes.

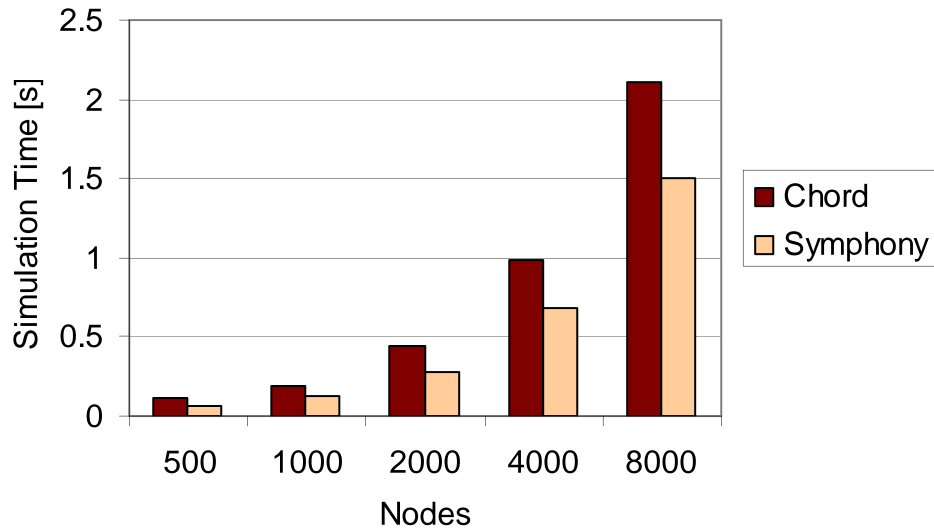


Figure 5.3 Chord and Symphony for increasing network size. Key lookups with regular distribution of nodes.

Figures 5.4 to 5.6 are similar to Figures 5.1 to 5.3, however, this time nodes generated are distributed randomly around the circle instead of being equally spaced as before.

Surprisingly, Figure 5.4 shows that this change reduces the time required for network generation (including stabilization) with Chord significantly, now faster than for the Symphony protocol. However, as depicted in Figures 5.5 and 5.6, key insertions and lookups with Symphony are again simulated quicker than for the Chord case.

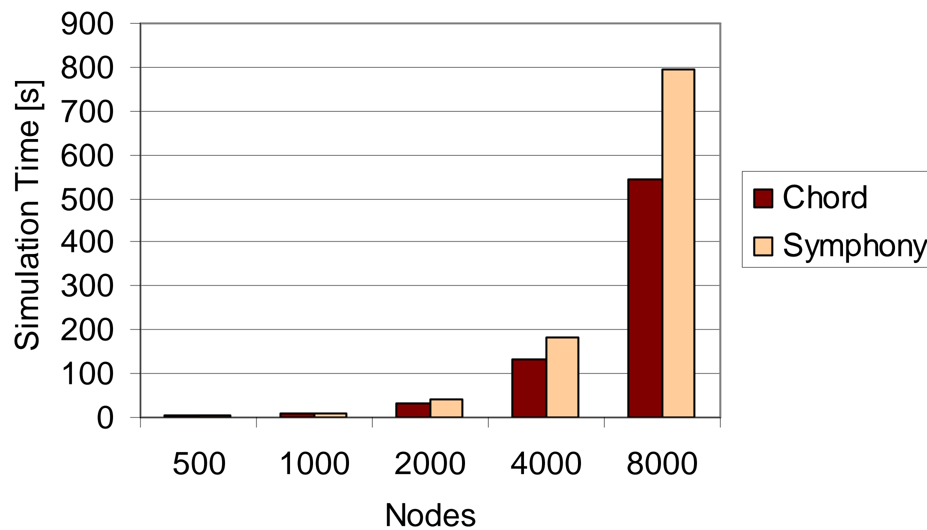


Figure 5.4 Chord and Symphony for increasing network size. Network creation with random distribution of nodes.

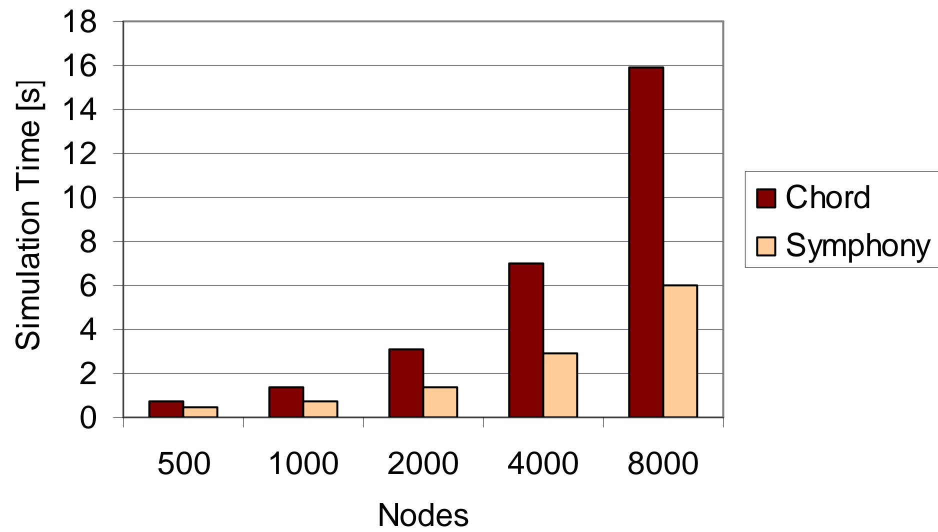


Figure 5.5 Chord and Symphony for increasing network size. Key insertions with random distribution of nodes.

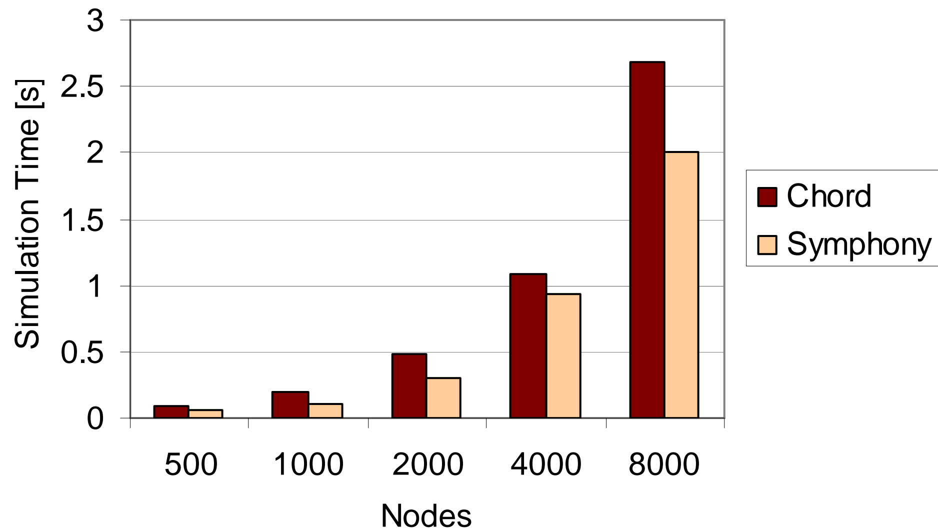


Figure 5.6 Chord and Symphony for increasing network size. Key lookups with random distribution of nodes.

In Figures 5.7 to 5.9, for a network with 1,000 equally spaced nodes, the influence of using different hash algorithms is examined. The maximum number of long distance links for Symphony is again two and Chord IDs are truncated to be 32 bits long.

Figure 5.7 is just a "sanity check": Since no hashing is applied when generating equally spaced nodes in PlanetSim, the time spent for network generation does not depend on the choice of hash algorithm.

Figure 5.8 and 5.9 show that performing the more secure hash algorithms generates minimal overhead for the key insertion and lookup tasks. However, note that Chord IDs and Symphony IDs are of fixed length in this simulation. In particular, keys for insertion into and lookup from the Chord ring are truncated to 32 bits (the chosen length of Chord IDs in this simulation), even if the hash algorithms generate outputs of 160 bits (for SHA-1) up to 512 bits (for SHA-512). Symphony IDs in PlanetSim are always represented as a "double" value in Java, corresponding to 8 bytes or 64 bits (Savitch, 1998), also leading to truncation of key IDs. Note that Symphony simulation runs are faster than with Chord, as in most cases before.

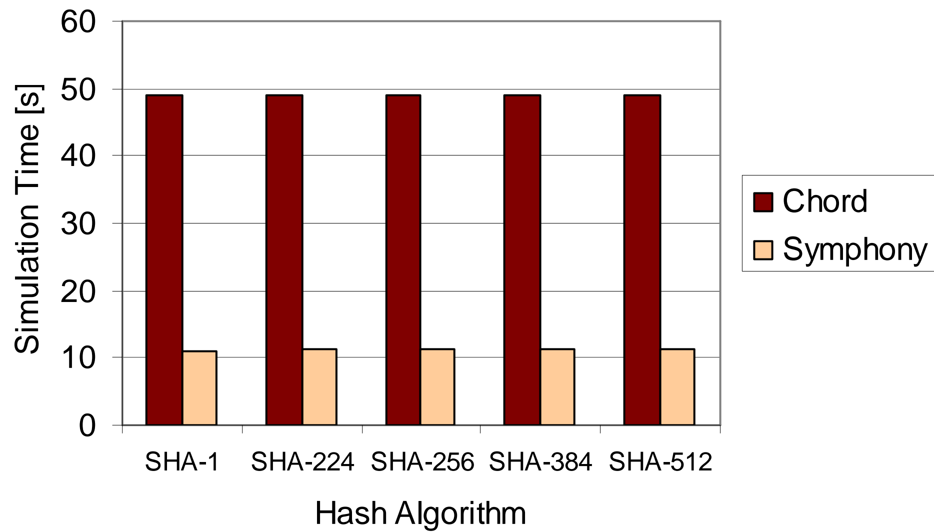


Figure 5.7 Chord and Symphony with use of different hash algorithms. Network creation with regular distribution of 1,000 nodes.

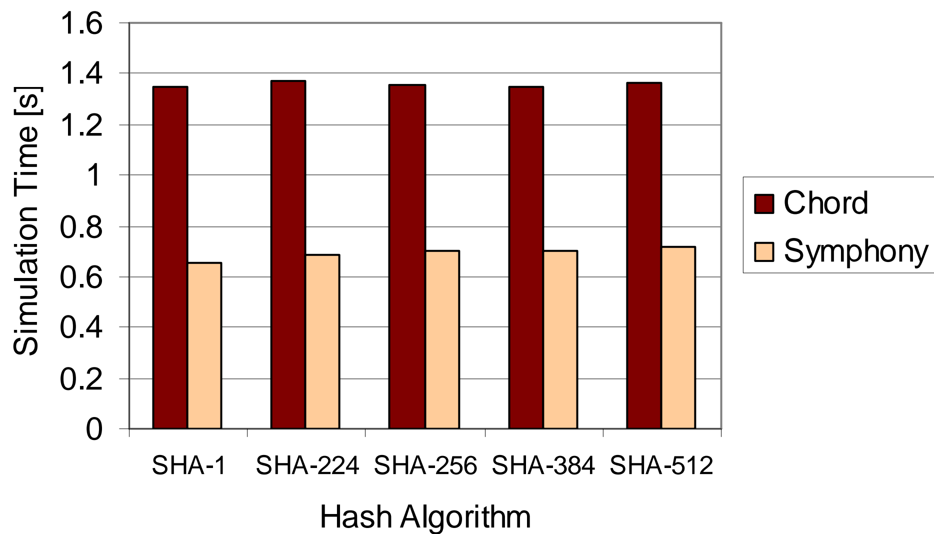


Figure 5.8 Chord and Symphony with use of different hash algorithms. Key insertions with regular distribution of 1,000 nodes.

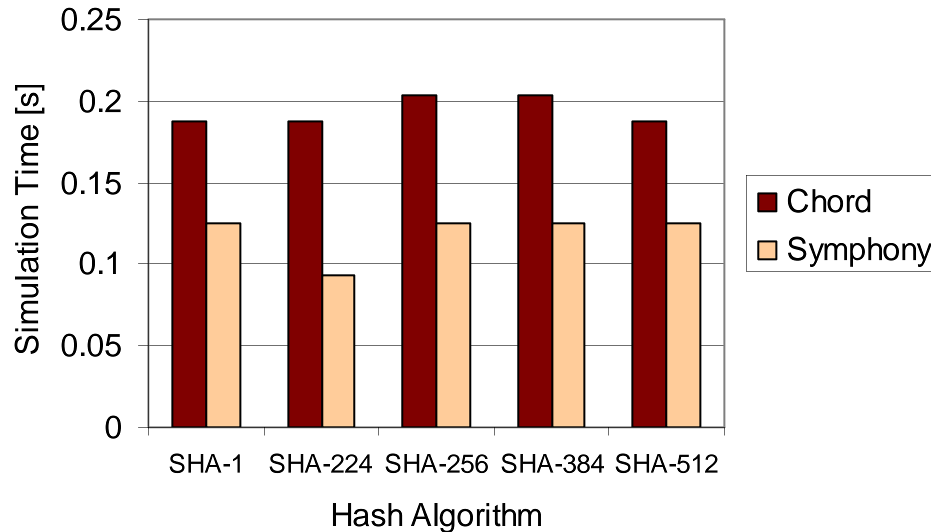


Figure 5.9 Chord and Symphony with use of different hash algorithms. Key lookups with regular distribution of 1,000 nodes.

For Figures 5.10 to 5.12 the length of Chord IDs in PlanetSim is varied, from 32 to 512 bits. The secure hashing algorithm SHA-512 is utilized in all cases. Thus, only in the last case of 512 bit long Chord IDs no truncation occurs. Network size is 1,000, with nodes uniformly distributed around the circle.

Longer IDs would be expected to lead to longer simulation times and the general trend is confirmed by the results in Figures 5.10 to 5.12. However, it is found that 64 bit long IDs give slightly faster run time compared to 32 bit long IDs. This is interesting since 32 is the default length of Chord IDs in the standard distribution of PlanetSim as of version 3.0. Note that the SHA-512 algorithm, or any other of the SHA-2 family, is not available in the standard PlanetSim; this capability was added during this thesis project.

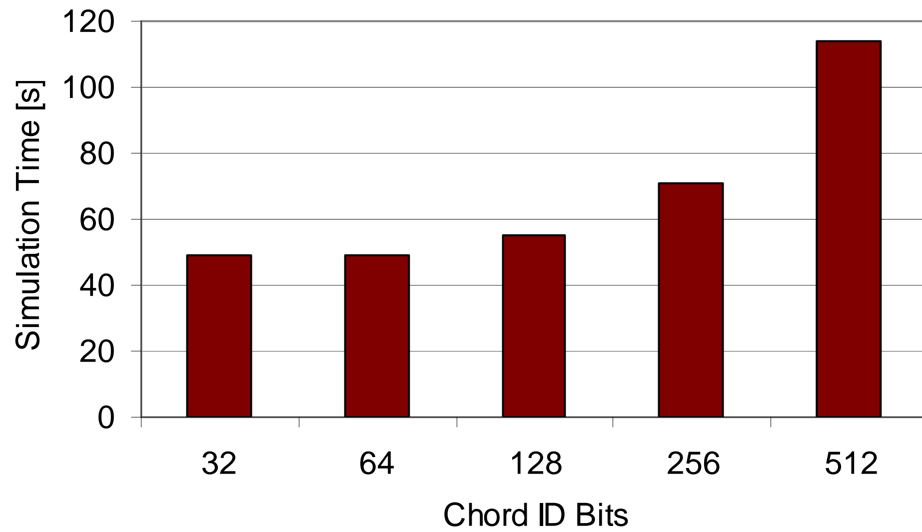


Figure 5.10 Chord with varying bit length of IDs. Network creation with regular distribution of 1,000 nodes.

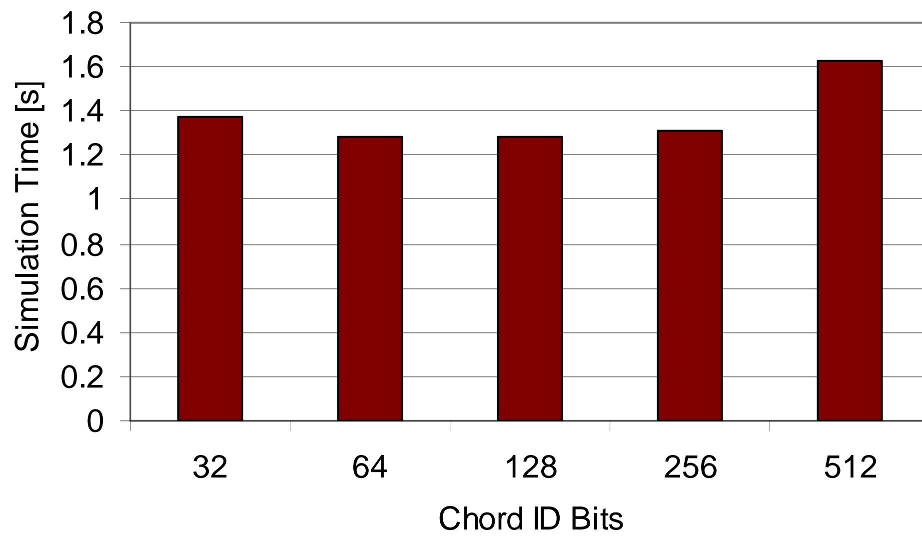


Figure 5.11 Chord with varying bit length of IDs. Key insertions with regular distribution of 1,000 nodes.

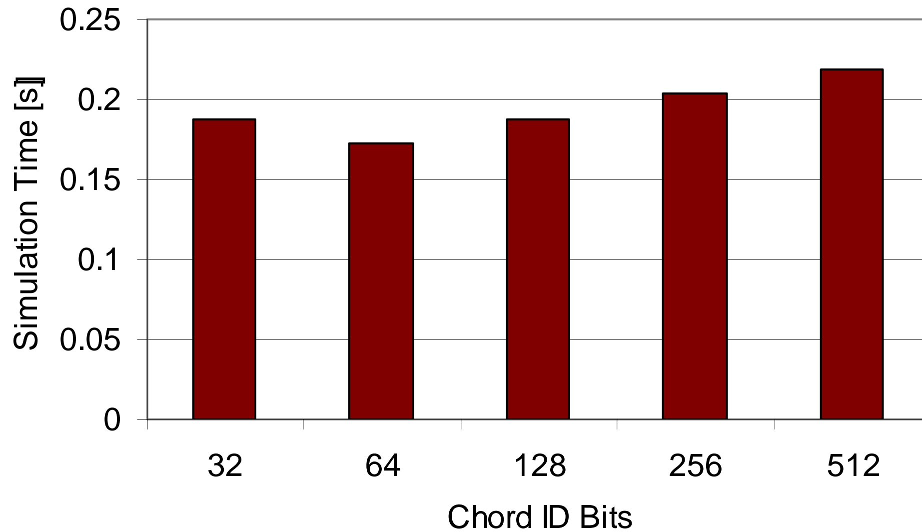


Figure 5.12 Chord with varying bit length of IDs. Key lookups with regular distribution of 1,000 nodes.

While for Chord a parameter to choose in PlanetSim is the number of bits per ID, for Symphony it is the maximum number of long distance links. Figures 5.13 to 5.15 show the results for simulation runs with an increasing maximum number of long distance links, doubling each time. Hashing is done using SHA-1. The size of the networks simulated is 8,000, with the nodes uniformly distributed around the circle.

Figures 5.13 and 5.14 show that the choice of maximum number of long distance links has little impact on network generation and key insertion simulation time. However, as can be seen from the results in Figure 5.15, the time spent on key lookups is reduced significantly for a larger number of long distance links, with eight being near the optimum. It is currently unclear why only key lookups and not insertions are sped up in this PlanetSim experiment (possible future work).

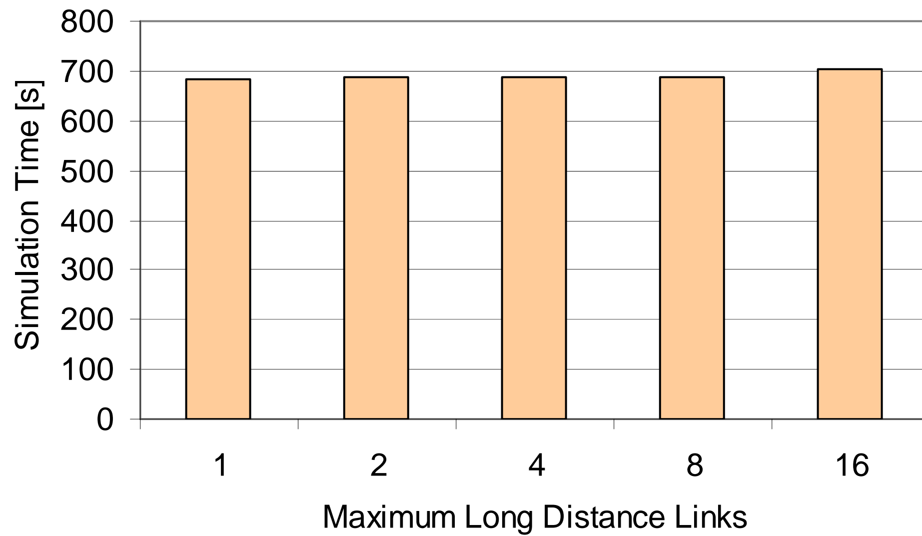


Figure 5.13 Symphony with varying maximum number of long distance links. Network creation with regular distribution of 8,000 nodes.

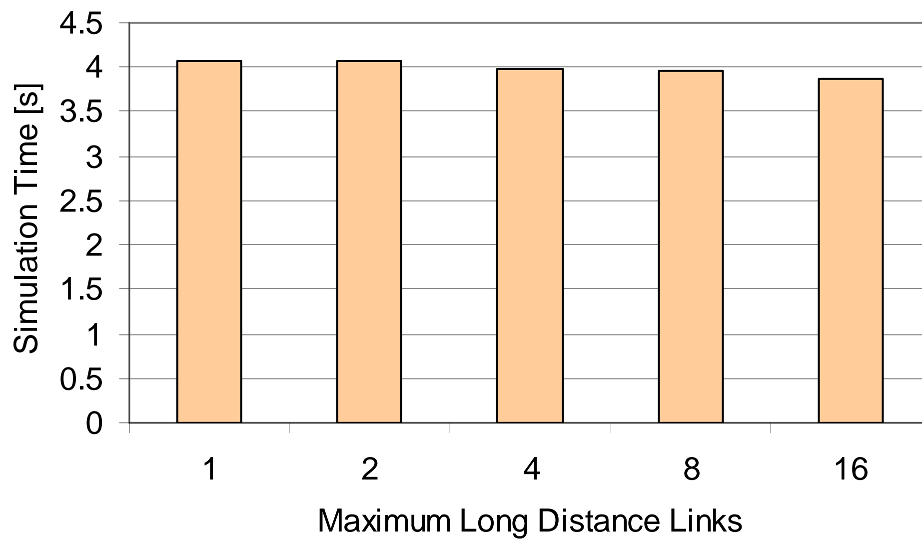


Figure 5.14 Symphony with varying maximum number of long distance links. Network creation with regular distribution of 8,000 nodes.

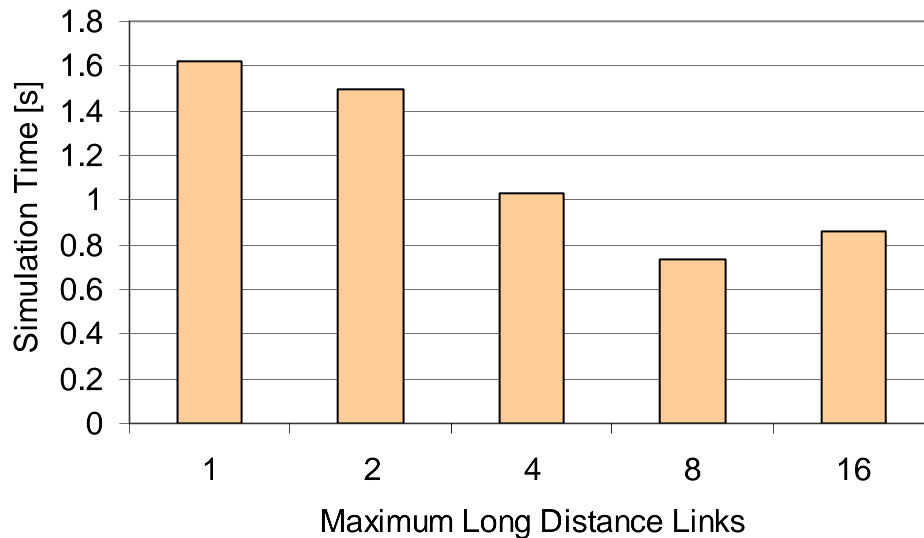


Figure 5.15 Symphony with varying maximum number of long distance links. Network creation with regular distribution of 8,000 nodes.

5.2 Hashing Examples

The original PlanetSim v.3.0 supports only SHA-1 hashing by making calls to the Java security package. In this thesis project, the PlanetSim code was extended to support many other algorithms, including all of the hash algorithms of the SHA-2 family: SHA-224, SHA-256, SHA-384, and SHA-512. This was done by adding a new input parameter to PlanetSim and integrating the freely available Jacksum package (www.jonelo.de/java/jacksum/). Making this work correctly required changes in many of the PlanetSim modules as shown in Appendix 2. At least as of Java v.1.5, the Java.security package does also support SHA-2, but not SHA-224, now available in PlanetSim through the addition of Jacksum.

The extended PlanetSim now supports a total of 44 different algorithms as of Jacksum v.1.6.1. Some of the alternative (now considered insecure) hash algorithms include CRC-32 (cyclic redundancy check; at 32 bits length convenient for quick testing) and the 128-bit algorithms MD2 and MD5. To make experimentation easy (without recompiling PlanetSim every time the user wants to use a different hash algorithm) a parameter FACTORIES_HASHALGORITHM was added to the simulator configuration (see Appendix 2).

As an example, consider the string “FEDC:BA98:7654:3210:3E9F:1089:FF8D:EE62”, the textual representation of some random IPv6 address.

CRC-32 will produce the following hexadecimal result:

968FAAB4

Written in decimal this is:

2525997748

Thus, the IPv6 address started out with falls onto the unit perimeter circle exactly at (calculated using the java.math.BigDecimal methods in Java):

0.588129681535065174102783203125

MD5 gives the following hexadecimal result:

8A32D6A5CE41976CAD2C8BAB87203AD7

Written in decimal this is:

183697431833380335436776137348754717399

Thus, the IPv6 address started out with falls onto the unit perimeter circle at:

0.53983823345175207287390575272075985835561730917773137588153053497539877451966044397103049179431621951152919791638851165771484375

SHA-1 produces the following hexadecimal result:

94D8289C92154120ADE0812949EF455F83091346

Written in decimal this is:

849751132814475020825707876972210082466358694726

Thus, the IPv6 address started out with falls onto the unit perimeter circle at:

0.58142331907773316290251530731518728556996856216182232078121433596058479927317861713213021399704004337767961401631613190671421520505646185483783483505249023437 5

SHA-224 leads to the following hexadecimal result:

EFBFCD05D5A65C68A1BB3141CBCF166923F3971D4662E5E4D41A4DFC

Written in decimal this is:

25248540010320515997968685886648407849683282049505812482027595386364

Thus, the IPv6 address started out with falls onto the unit perimeter circle at:

0.93652039902158307772436527627560692154505015077266381554772255337070234232611002657962091518420894219639778367267792189943151238404899 99

745333039089528462918020935203568773355420123624681671969938179245218634

60540771484375

SHA-256 gives the following hexadecimal result:

6FC773FD3993D62C2B926A4B5CDFFAF6525A477606C14A5E6BC1BC176E

14098C

Written in decimal this is:

505591292863934723745651234995784427489555830513409775829662544824

35909880204

Thus, the IPv6 address started out with falls onto the unit perimeter circle at:

0.4366371625161059335037940642489789203647808848761180540929910485

4957136457277311255398391541608741116106379064441146261717066575831731 66

4469211603466364109457273744249194650466102800294384554932822022111676 53

752976053338485940003010910004377365112304687 5

SHA-384 leads to the following hexadecimal result:

531AB7748460C56108FC0128797C217965A24C568416412F8FF1F8E57ADFE

6BFC28E7412B8839939BE4A0A3AE518866F

Written in decimal this is:

127909319481381079152413030146897830465039085243004647416738059808

888818412193861047808624058801741236109538760147 03

Thus, the IPv6 address started out with falls onto the unit perimeter circle at:

0.324626413294370660664924523772587472116721143121168197400938447694498253544759802144020647310745124115492383025616580791604876318886070596920559860762941709389042628590464472165661879854117495104929727478441572639780328998438066931162380954027940973684105610300927352444677527869900650234371231240301891528754587251603642706685895919979577771939671659851001095375977456569671630859375

Finally, **SHA-512** will generate the following hexadecimal result:

AE25434F7D6E459B68AADEC9B971725651E45CEB06A45788441E78F3B79B5A5A452C943C6AD9607766AC295FE29BA65DF4F7056129C77B63F91A219596B98DC4

Written in decimal this is:

91207429605424332765135447158149672697877340535749526706395619914646781911917921708708566440825070678471695634244073294855680182033903042902373571458206l2

Thus, the IPv6 address started out with falls onto the unit perimeter circle at:

0.6802560872142119020459495415950874763824349178077746315426627339971886872282301969457438895154988151517487785529967050817026466961123123883951188381146660606236515011502441054483678517708992202796566768441243296781478175297469014820610855580237609811061062489952731056803130454217012811937682338101359832250862796238275132786502852346802119145287794345558481894687107660894893029848045477090182489655052615953020122538325949

822855413698476024967682418830344273433377100604446674481096124509349465
37017822265625

From the growing length of the results listed above it should be clear that there may be concerns regarding overhead incurred due to use of the more secure hash algorithms. This is investigated to some extent in Section 5.1 (see also Appendix 1).

Note how for the different hash algorithms, the same node (with the specific given IP address) ends up at different locations around the circle, approximately at 0.59, 0.54, 0.58, 0.94, 0.44, 0.32, and 0.68.

While above the generation of a *node* ID from an IPv6 address was demonstrated, similar examples can be given for the generation of *key* IDs from arbitrary strings.

5.3 Conclusions and Future Directions

Some of the explored themes such as load balancing show that Chord can be less than ideal. Some newer proposals such as the one by Godfrey *et al.*, (2004) advocate the idea of inserting virtual servers where node utilization is at its maximum. Then this physical node would become a host to any number of virtual servers by logically transporting these virtual servers from where heavy traffic abides in the physical node level to less transited nodes. Some of this is already used in Chord with CFS (Dabek, 2001). This has been examined with the property of having the node IDs assigned randomly. It leads, according to Godfrey *et al.*, (2004), to a logarithmic imbalance that would be corrected with the insertion of the proposed virtual servers.

An example could be as shown in Tables 5.1 to 5.3. Suppose that the load in node "c" of Table 5.1 is equivalent to 15 and its target load is a mere 12. That would mean that the traffic is very heavy in node "c".

Node "c"	5	4	6

Table 5.1 Node "c" has a heavy load of 15 and a target of only 12

Thus the proposal is to redirect some of the heavy traffic to a less traversed or utilized node, such as node "f" shown in Table 5.2.

Node "f"	2	3

Table 5.2 Node "f" has a load equivalent to 5 and a target of 10

Therefore, if some of the traffic of node "c" can be transferred to the underutilized node "f"", equilibrium is reached as shown in Table 5.3. Since now the load of weight 4 from node "c" has been moved virtually to node "f", both nodes are rendered a light load.

Node "c"	5	~~4~~	6	Load = 11
		↓		
Node "f"	2	4	3	Load = 9

Table 5.3 The shift of 4 from node "c" to node "f" makes both nodes light

The advantage of this proposal is that it is flexible in moving loads from any place to any other location in the DHT. This technique would not impose too many rules or re-change the entire infrastructure of the DHT but only require updating the finger tables to let the nodes know where the transferred nodes now reside. With this, the light nodes would then pass the information to directories. The heavily loaded nodes will then request information from these directories on where and how to get in touch with the lighter loaded nodes and in this process pass the excess load to them. This has proved to show good results in heavily transited systems.

Another point is that in practice both Chord and Symphony never will reach a true state of stabilization since realistic large-scale systems are always in a state of continuous flux. Theoretical studies can model and simulate the protocols and consider them to achieve great results, but for real time networks they are both still under examination.

A general concern related to P2Ps is that not every user who wants to deploy a large-scale distributed network would use it for legitimate purposes, but some devote a great deal of time on malicious projects. Some will create modified protocols that can cause major disruptions such as can be seen with the so-called Sybill and Preimage attacks. The implantation of malicious nodes is already an issue now. This is unfortunate since the original historical basis of P2P networks (see Section 1.1) displayed an idyllic environment to develop research. The modern P2Ps could serve as a model for global cooperation, but sadly enough this may take a turn for the worse and this beautiful concept may have to reverse to or tend towards a more centralized system in order to monitor activity.

Glossary

ARPANET was the Advanced Research Projects Agency Network.

Bucket is a place where more than one piece of information can be stored.

Chord is an overlay protocol that has a ring structure and a deterministic topology.

Collision occurs when data are to be inserted in a bucket already in use.

DHT is a distributed hash table.

Distributed System resides on a collection of nodes, as opposed to one node.

Hashing is a method to enable insertion of values into a key-based table.

Hash Table is an array that stores hashed values for easy access and retrieval.

IMP stands for interface message processor, a protocol that allowed computers to send and receive messages on the early ARPANET.

Keyspace Partitioning assigns individual nodes as managers of entire sub-sections of a keyspace.

Load Factor represents a measure of what percentage of a hash table is filled up with values.

Logarithm is the power to which a base, such as 10, must be raised to produce a given number. If $n^x = a$, the logarithm of a, with n as the base, is x; symbolically, $\log_n a = x$. For example, $10^3 = 1{,}000$; therefore, $\log_{10} 1{,}000 = 3$. The kinds most often used are the common logarithm (base 10), the natural logarithm (base e), and the binary logarithm (base 2). Definition from: Logarithm (2003).

Overlay Network is a logical network that is run on top of a physical network in order to share resources or facilitate the execution of distributed applications.

Modulo Operation is used in calculating hash values by acquiring the remainder of an integer division.

P2P, Peer-to-Peer or sometimes referred to as Person-to-Person, is a way to communicate data between systems worldwide without a centralized directory or authority.

Small World Algorithm was proposed by John Kleinberg based on Milgram's theory that, if forwarding something at random, someone (or a node) is bound to know the destination.

Symphony is an advanced overlay network protocol that has a ring structure using k long-range links to hook up with its neighbor in a randomized fashion.

References

Allen, F. E., (1981). The History of Language Processor Technology in IBM. IBM J. Res. Development, vol. 25, no. 5, September 1981.
<www.research.ibm.com/journal/rd/255/ibmrd2505Q.pdf>

ARPANET History, (2004).
<http://www.jmusheneaux.com/21bb.htm>

Baldwin, R. G., (2005). Message Digests 101 using Java.
<http://www.developer.com/security/article.php/3487986>

Ball, P., (1996). Introduction to Discrete Event Simulation. Appears in Proc. 2nd DYCOMANS Workshop on "Management and Control: Tools in Action" in the Algarve, Portugal, May 15–17, 1996, pp. 367–376.
<http://www.dmem.strath.ac.uk/~pball/simulation/simulate.html>

Black, P. E., (2005). Hash Table. Dictionary of Algorithms and Data Structures, National Institute of Standards and Technology.
<http://www.nist.gov/dads/HTML/hashtab.html >

Bradner, S., and Mankin, A., (1995). RFC 1752: The Recommendation for the IP Next Generation Protocol. January 1995.
<http://www.ietf.org/rfc/rfc1752.txt?number=1752>

Brookshear, J. G., (2000). Computer Science: An Overview, Sixth Edition. Marquette University, Addison Wesley.

Chern, M.S, (2005). The Design and Analysis of Computer Algorithms.
<http://chern.ie.nthu.edu.tw/alg2003/ChaP2_alg.html>

Dabek, F., (2001). A Cooperative File System. Master's thesis, MIT, September 2001.
<pdos.csail.mit.edu/papers/cfs:sosp01/cfs_sosp.pdf >

Dabek, F., Brunskill, E., Kaashoek, M. F., Karger, D., Morris, R., Stoica, I., and Balakrishnan, H., (2002). Building Peer-to-Peer Systems with Chord, a Distributed

Lookup Service. Appears in Proc. 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII).
<http://www.pdos.lcs.mit.edu/papers/chord:hotos01/hotos8.pdf>

Darlagiannis, V., Liebau, N., Mauthe, A., and Steinmetz, R., (2004). An Adaptable, Role-Based Simulator for P2P Networks. Appears in Proc. 2004 International Conference on Modeling, Simulation and Visualization Methods, Las Vegas, NV, June 2004.
<http://www.kom.e-technik.tu-darmstadt.de/publications/abstracts/DMLS04-1.html>

Davis, D. R., and Lin, A. D., (1965). Secondary Key Retrieval Using an IBM 7090-1301 System.

Dumey A. I., (1956). Indexing for Rapid Random-Access Memory, Computers and Automation, vol. 5, no. 12, pp. 6–9, 1956.

El-Ansary, S., Onana Alima, L., Brand, P., and Haridi, S., (2003). Efficient Broadcast in Structured {P2P} Networks.
<citeseer.ist.psu.edu/article/alima03efficient.html>

Fanning, S., (1999). Wikipedia.
<http://en.wikipedia.org/wiki/Shawn_Fanning> and <http://www.snocap.com/>

García, P., Pairot, C., Mondéjar, R., Pujol, J., Tejedor, H., and Rallo, R., (2005). PlanetSim: A New Overlay Network Simulation Framework. Appears in Lecture Notes in Computer Science (LNCS), Software Engineering and Middleware, SEM 2004, Linz, Austria, Revised Selected Papers, vol. 3437, pp. 123–137, March 2005.
<http://ants.etse.urv.es/planetsim/>

Gil, T., Kaashoek, F., Li, J., Morris, R., and Stribling, J., (2004). P2PSim: A Simulator for P2P Network Protocols.
<http://pdos.csail.mit.edu/p2psim/>

Godfrey, B., Lakshminarayanan, K., and Surana, S., (2004). Load Balancing in Dynamic Structured P2P Systems. Appears in Proc. IEEE INFOCOM, March 2004.
<http://www.ieee-infocom.org/2004/Papers/46_4.PDF>

He, Q., Ammar, M., Riley G., Raj H., and Fujimoto, R., (2003). Mapping Peer Behavior to Packet-Level Details: A Framework for Packet-Level Simulation of Peer-to-Peer Systems. Appears in Proc. IEEE/ACM MASCOTS, pp. 71–78, October 2003.

Horovitz, S., (2005). Reliability of Distributed Systems, Lecture 12: Introduction to Peer-to-Peer Systems. January 2005.
<http://www.cs.huji.ac.il/course/2004/com1/Presentations/2004-5/Lessons/3_P2P/RDS2004_P2P.pdf >

Jacksum (2002–2006). Jacksum 1.6.1: A Free and Platform Independent Open Source Project.
<http://www.jonelo.de/java/jacksum/ >

Jain, R., (1997). IP Next Generation.
<http://dslab.cis.thu.edu.tw/course/87_2/ia/Ohio/f18_ip6.pdf>

Jelasity, M., Jesi, G. P., Montresor, A., and Voulgaris, S., (2004). Peersim: A Peer-to-Peer Simulator.
<http://peersim.sourceforge.net/#intro>

Joseph, S., (2003). An Extendible Open Source P2P Simulator. Appears in P2P Journal, University of Tokyo, Japan. November 2003.
<http://p2pjournal.com/issues/November03.pdf>

Kant, K., and Iyer, R., (2003). Modeling and Simulation of Ad-Hoc/P2P File-Sharing Networks.

Kleinberg, J., (2000). The Small-World Phenomenon: An Algorithmic Perspective. Appears in Proc. 32nd ACM Symposium on Theory of Computing, 2000.
<http://www.cs.cornell.edu/home/kleinber/swn.ps>

Krowne, A., (2005). Good Hash Table Primes.
<http://planetmath.org/encyclopedia/GoodHashTablePrimes.html>

Kubiatowicz, J., (2003). Extracting Guarantees from Chaos. Communications of the ACM, vol. 46, no 2, pp. 33–38, February 2003.
<http://oceanstore.cs.berkeley.edu/publications/papers/abstracts/CACM-kubiatowicz.html>

Lewis, H. R., and Papadimitriou, C.H., (1998). Elements in the Theory of Computation. Prentice Hall, Upper Saddle River, NJ.

Liben-Nowell, D., Balakrishnan, H., and Karger, D., (2002a). Analysis of the Evolution of Peer-to-Peer Systems. Appears in Proc. ACM Conf. on Principles of Distributed Computing (PODC), Monterey, CA.
<http://www.pdos.lcs.mit.edu/chord/papers/podc2002.pdf>

Liben-Nowell, D., Balakrishnan, H., and Karger, D., (2002b). Observations on the Dynamic Evolution of Peer-to-Peer Networks. Appears in Proc. First Int. Workshop on Peer-to-Peer Systems (IPTPS 2002), Cambridge, MA.
<http://www.pdos.lcs.mit.edu/chord/papers/iptps-evol.pdf>

Licker, M. D., (2003). Dictionary of Mathematics, Second Edition. McGraw-Hill.

Logarithm, (2003). The American Heritage® Dictionary of the English Language, Fourth Edition. Houghton Mifflin Company.
<http://www.thefreedictionary.com/logarithm>

Malkhi, D., Naor, M., and Ratajczak, D., (2002). Viceroy: A Scalable and Dynamic Emulation of the Butterfly. Appears in Proc. ACM Conf. on Principles of Distributed Computing (PODC), Monterey, CA.
<http://citeseer.ist.psu.edu/540694.html>

Manku, G. S., Bawa, M., and Raghavan, P., (2003a). Symphony: Distributed Hashing in a Small World. Appears in Proc. 4th USENIX Symposium on Internet Technologies and Systems (USITS), pp. 127–140, March 2003.
<http://www-db.stanford.edu/~manku/papers/03usits-symphony.pdf>

Manku, S. G., (2003b). Routing Networks for Distributed Hash Tables. Appears in Proc. ACM Conf. on Principles of Distributed Computing (PODC), 2003.
<http://www-db.stanford.edu/~manku/papers/03podc-dht.pdf>

Manku, S. G., Naor, M., and Wieder, U., (2004a). Know thy Neighbor's Neighbor: the Power of Lookahead in Randomized P2P Networks. Appears in Proc. 36th ACM Symposium on Theory of Computing (STOC), pp. 54–63, June 2004.
<http://www-db.stanford.edu/~manku/papers/04stoc-nn.pdf>

Manku, S. G., (2004b). Dipsea: A Modular Distributed Hash Table. PhD Dissertation, Department of Computer Science, Stanford University, CA, September 2004.
<http://www-db.stanford.edu/~manku/phd/index.html>

Milgram, S., (1967). The Small World Problem. Psychology Today, vol. 61, no. 1.

Morris, R. T., and Sit, E., (2002). Security Considerations for Peer-to-Peer Distributed Hash Tables. Appears in Proc. First Int. Workshop on NetworkWorldFusion, March 2002.
<http://www.nwfusion.com/>

Motowani, R., and Raghavan, P., (1999). Randomized Algorithms. Cambridge University Press, New York, NY.

Oceanstore Project. Providing Global-Scale Persistent Data. Computer Science Division, University of California at Berkeley.
<http://oceanstore.cs.berkeley.edu/>

Oram, A., and O'Reilly and Associates, (2001). Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology. O'Reilly and Associates, Sebastopol, CA.

Rhea, S., Geels, D., Roscoe, T., and Kubiatowicz, J., (2004). Handling Churn in a DHT. Appears in Proc. USENIX Annual Technical Conf., June 2004.
<oceanstore.cs.berkeley.edu/publications/papers/pdf/bamboo-usenix.pdf >

Risson, J., and Moors, T., (2004). Survey of Research towards Robust Peer-to-Peer Networks: Search Methods. University of South Wales, Wales, UK.
< http://uluru.ee.unsw.edu.au/~john/tr-unsw-ee-p2p-1-1.pdf >

Rowstron, A., and Druschel, P., (2001). Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems. Appears in IFIP/ACM Int. Conf. on Distributed Systems Platforms (Middleware), Heidelberg, Germany, pp. 329–350, November 2001.
<http://freepastry.rice.edu/pubs.htm>

Savitch, W., (1998). Java: An Introduction to Computer Science & Programming. Prentice Hall.

Schneier, B., (2005). Schneier on Security: SHA-1 Broken.
<http://www.schneier.com/blog/archives/2005/02/sha1_broken.html>

Secure Hash Standard, (1993). Announcing the Standard for Secure Hash Standard. Federal Information Processing Standards Publication (FIPS) 180-1, May 1993–April 1995.
<http://www.itl.nist.gov/fipspubs/fip180-1.htm>

Secure Hash Standard, (2001). Announcing the Secure Hash Standard. Federal Information Processing Standards Publication (FIPS) 180-2 Plus Change Notice to Include SHA-224, August 2001–February 2004.
<http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf >

Stark, H., and Woods, J.W., (1994). Probability, Random Processes, and Estimation Theory for Engineers. Prentice Hall, Englewood Cliffs, NJ.

Stoica, I., Balakrishnan, H., Kaashoek, F. M., Karger, D., and Morris, R., (2001). Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. Appears in Proc. ACM SIGCOMM, San Diego, CA, August 2001, pp. 149–160.
<http://www.pdos.lcs.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf>

Thomer, G. M., Kaashoek, F., Li, J., Morris, R., and Stribling, J., (2004). P2Psim: A Simulator for Peer-to-Peer Protocols. July 2004.
<http://pdos.csail.mit.edu/p2psim/>

Ting, N. S., Deters, R., (2003). 3LS – A Peer-to-Peer Network Simulator. Appears in Documents in Computing and Information Science, University of Saskatchewan, Canada.
<http://csdl2.computer.org/comp/proceedings/p2p/2003/2023/00/20230212.pdf>

Travers, J., and Milgram, S., (1969). An Experimental Study of the Small World Problem. Sociometry, vol. 32, pp. 425.

Truscott, T., and Ellis, J., (1979). USENET.
<http://www.usenet.com/articles/what_is_usenet.htm>

Wang, X., Yin, Y. L., and Yu, H., (2005). Collision Search Attacks on SHA1. Shandong University, China, February 2005.
<http://theory.csail.mit.edu/~yiqun/shanote.pdf>

Watts, D., and Strogatz, S., (1998). Collective Dynamics of ‘Small-World’ Networks. Nature, vol. 393, pp. 440–442, October 1998.

Wikipedia (2005). Peer-to-Peer.
<http://en.wikipedia.org/wiki/Peer-to-peer>

Zhao, B., Dabek, F., Druschel, P., Kubiatowicz, J., and Stoica, I., (2003). Towards a Common API for Structured Peer-to-Peer Networks. Appears in Proc. 2nd Int. Workshop on Peer-to-Peer Systems (IPTPS), February 2003.
<http://oceanstore.cs.berkeley.edu/publications/papers/pdf/iptps03-api.pdf>

Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D., and Kubiatowicz, J. D., (2004). Tapestry: A Resilient Global-Scale Overlay for Service Deployment. IEEE J. Selected Areas in Communications, vol. 22, no. 1, pp. 41–53, January 2004.

Appendix 1 Tabulated PlanetSim Results

For reference, the simulation results explained and charted in Chapter 5 are given here in tabulated form. Please see Chapter 5.1 for details. In addition, the simulation steps reported by PlanetSim are listed.

For Chord: In the table, "Bits" refers to the number of bits for Chord IDs in the simulation. Topology "Regular" means equally spaced node IDs around the circle. For "Random" the network nodes are distributed randomly around the circle.

Simulation Parameters				*Network Creation*		*Key Insertions (100)*		*Key Lookups (100)*	
Topology	**Hash Alg**	**Bits**	**Nodes**	**Sim Steps**	**Time [s]**	**Sim Steps**	**Time [s]**	**Sim Steps**	**Time [s]**
Regular	SHA-1	32	500	5797	11.688	6003	0.687	6032	0.109
			1000	10795	49.047	11002	1.344	11032	0.188
			2000	20784	202.265	20991	2.781	21022	0.438
			4000	40795	886.531	40999	5.953	41032	0.984
			8000	80795	4001.093	80999	13.421	81031	2.11
Regular	SHA-512	32	1000	10795	48.968	11001	1.375	11030	0.187
		64		11595	48.828	11802	1.281	11832	0.172
		128		13197	55.093	13404	1.281	13434	0.188
		256		16395	71.203	16601	1.312	16632	0.204
		512		22785	113.828	22989	1.625	23019	0.219
Regular	SHA-1	32	1000	10795	49.047	11002	1.344	11032	0.188
	SHA-224			10795	48.985	11002	1.375	11031	0.188
	SHA-256			10795	48.969	10999	1.359	11029	0.203
	SHA-384			10795	48.938	11001	1.344	11030	0.203
	SHA-512			10795	48.953	11000	1.36	11030	0.188
Random	SHA-1	32	500	2951	6.235	3156	0.719	3184	0.094
			1000	2819	10.125	3025	1.39	3054	0.203
			2000	4819	33.61	5021	3.047	5052	0.485
			4000	8810	131.438	9015	7.015	9046	1.078
			8000	16817	543.828	17025	15.907	17059	2.688

Table A1.1 Tabulated Chord simulation results obtained with PlanetSim

For Symphony: In the table, “kmax” refers to the maximum number of long distance links in the simulation using PlanetSim. The other columns are as for the tabulated Chord results above.

Simulation Parameters				*Network Creation*		*Key Insertions (100)*		*Key Lookups (100)*	
Topology	**Hash Alg**	**kmax**	**Nodes**	**Sim Steps**	**Time [s]**	**Sim Steps**	**Time [s]**	**Sim Steps**	**Time [s]**
Regular	SHA-1	1	500	1420	3.375	1644	0.484	1715	0.094
			1000	2770	11.313	2984	0.671	3045	0.172
			2000	5493	41.922	5709	1.141	5775	0.328
			4000	10878	166.359	11110	2.094	11185	0.704
			8000	21838	684.453	22090	4.062	22177	1.625
Regular	SHA-1	2	500	1407	3.328	1609	0.469	1645	0.063
			1000	2743	11.063	2953	0.656	2998	0.125
			2000	5432	42.468	5645	1.156	5695	0.282
			4000	10979	164.234	11195	2.094	11268	0.688
			8000	21930	688.422	22167	4.063	22247	1.5
Regular	SHA-1	1	8000	21838	684.453	22090	4.062	22177	1.625
		2		21930	688.422	22167	4.063	22247	1.5
		4		21930	687.516	22162	3.984	22216	1.031
		8		21863	688.359	22075	3.968	22113	0.735
		16		21952	705	22159	3.875	22205	0.859
Regular	SHA-1	2	1000	2743	11.063	2953	0.656	2998	0.125
	SHA-224			2770	11.187	2984	0.687	3017	0.093
	SHA-256			2756	11.14	2963	0.703	3011	0.125
	SHA-384			2776	11.391	2992	0.703	3035	0.125
	SHA-512			2764	11.235	2985	0.719	3027	0.125
Random	SHA-1	2	500	1359	3.125	1566	0.453	1601	0.063
			1000	2731	11.172	2952	0.734	2988	0.109
			2000	5318	43.218	5537	1.375	5580	0.296
			4000	10831	183.344	11050	2.922	11121	0.938
			8000	21605	795.078	21843	5.984	21913	2

Table A1.2 Tabulated Symphony simulation results obtained with PlanetSim

Appendix 2 Application Code and Extensions

This appendix summarizes the PlanetSim source code modifications and extensions. The Java code developed is based on PlanetSim v.3.0 (Garcia *et al.*, 2005). Unmodified PlanetSim Java packages, classes, and methods are omitted. Omissions within classes are indicated by "`[...]`". Please consult the PlanetSim source code for those omitted portions of the simulator code. Also, a marker "`### mhb ###`" is used to flag additions and modifications visibly. Java classes are listed alphabetically including full package name.

The Java classes are followed by the modified portions of the PlanetSim configuration files that are located in the "conf" directory of the PlanetSim distribution. These configuration file snippets show the parameters that need to be set in order to generate the results in this thesis.

Finally, there is a batch file that can be used to run the simulator on a Windows platform.

Java class "planet.chord.ChordId":

The simple change here is needed to enable Chord with IDs that exceed 160 bits in length, such as all of the SHA-2 hashing algorithms.

```
package planet.chord;
import java.math.BigInteger;
import java.util.Random;

import planet.commonapi.Id;
```

```
import planet.util.Properties;
import planet.util.Utilities;

/**
 * Spedific Chord Id implementation.
 * @author
 * <a href="mailto: jordi.pujol@estudiants.urv.es">Jordi Pujol</a>
 * 25/02/2005
 */
public class ChordId extends Id implements java.io.Serializable {

      /* ****************** CHORD SPECIFIC  CONSTANTS **************/
      /**
       * Chord specific constant: Maximum number of bits for any Id.
       */
      public static final int MAX_BITS = 512;
/* ### mhb ### public static final int MAX_BITS = 160; */

[...]

}
```

Java class “planet.chord.ChordProperties”:

The changes here also relate to enabling Chord with IDs that exceed 160 bits in length.

```
package planet.chord;

import planet.commonapi.exception.InitializationException;
import planet.util.OverlayProperties;
import planet.util.PropertiesWrapper;

/**
 * This class includes the initialization and the values for all
 * configuration properties of the Chord overlay.
 * @author
 * <a href="mailto: jordi.pujol@estudiants.urv.es">Jordi Pujol</a>
 * Date: 05/07/2004
 */
public class ChordProperties implements OverlayProperties {

[...]

      /**
       * Initialize all configuration properties of the Chord overlay.
       * @see [...]
       * @param properties Properties with all (key,value) pairs.
       * @throws InitializationException
       */
      public void init(PropertiesWrapper properties)
      throws InitializationException {
```

```
[...]

        bitsPerKey = properties.getPropertyAsInt(CHORD_BITS_PER_KEY);
        if (!isValidValue(bitsPerKey))
            throw new InitializationException("Property '" +
            CHORD_BITS_PER_KEY +
            "' is not a valid value. Must be multiple of 32 " +
            "in range of [32..512].");
/* ### mhb ### throw new InitializationException("Property
'"+CHORD_BITS_PER_KEY+"' is not a valid value. Must be multiple of 32
in range of [32..160]."); */
      }

[...]

    /**
     * Test if the <b>bitsPerKey</b> is multiple of 32 within the range
     * [32..512]. ### mhb ###
     * @param bitsPerKey Number of bits per key to be tested.
     * @return true if the preconditions are accomplished or
     * false in other case.
     */
    public static boolean isValidValue(int bitsPerKey)
    {
        return bitsPerKey > 0 && bitsPerKey <= 512 && bitsPerKey%32==0;
/* ### mhb ### return bitsPerKey > 0 && bitsPerKey <= 160 &&
bitsPerKey%32==0; */
    }

[...]

}
```

Java class “planet.commonapi.Id”:

Calls to the java.security package are replaced by a call to the modified planet.util package to enable hashing using Jacksum.

```
package planet.commonapi;

import java.io.Serializable;
import java.math.BigInteger;
// ###mhb ### import java.security.MessageDigest;
// ###mhb ### import java.security.NoSuchAlgorithmException;
import java.util.Random;

import planet.commonapi.exception.InitializationException;
import planet.util.Utilities; /* ### mhb ### */
```

```
/**
 * This interface is an abstraction of an Id (or key)
 * from the CommonAPI paper.
 * <br><br>
 * Any class that extends this <b>Id</b> for a specific overlay,
 * must offer as least the no argument constructor.
 * <br><br>
 * The value for any built Id will be made with the related
 * <b>setValues()</b>
 * method. If any of them is nonapplicable to any Id implementation,
 * an InitializationException must be thrown.
 *
 * @author <a href="mailto: cpairot@etse.urv.es">Carles Pairot</a>
 * @author <a href="mailto: ruben.mondejar@estudiants.urv.es">Ruben
 * Mondejar</a>
 * @author
 * <a href="mailto: jordi.pujol@estudiants.urv.es">Jordi Pujol</a>
 *
 * @see planet.commonapi.exception.InitializationException
 */
public abstract class Id implements Comparable, Serializable {

[...]

    /**
     * Sets the new value for this Id, based on a hashed value. Finally,
     * it uses the setValues(byte[]) to set the correct value.
     * If this type of internal value is non applicable for the related
     * implementation, we recommend throws a NoSuchMethodError error.
     * @param material The input for the algorithm.
     * @param algorithm One-way hashing algorithm
     * such as "SHA" or "MD5".
     * @throws InitializationException if the <b>material</b>
     * is null or the <B>algorithm</b> is not found.
     * @return The Id itself
     * @see #setValues(byte[])
     */
    public Id setValues(String material, String algorithm)
    throws InitializationException {
        byte[] digest = null;
        if (material == null)
            throw new InitializationException(
                  "The arbitrary string is set to null");
        /** ### mhb ###
         * try {
         * MessageDigest md = MessageDigest.getInstance(algorithm);
         * md.update(material.getBytes());
         * digest = md.digest();
         * } catch (NoSuchAlgorithmException e) {
         * throw new InitializationException(
         *       "Algorithm not supported!", e);
         * }
         */
        digest = Utilities.generateByteHash(material,
```

```
                        algorithm); // ### mhb ###

        return setValues(digest);
    }
}
```

Java class “planet.generic.commonapi.factory.IdFactoryImpl”:

Implements the new network topology HASHED where the network is built by reading IP addresses (including IPv6 addresses) from a file and hashing the IP addresses. The IP addresses are actually first looked up on the Internet by using the java.net host lookup service. For IPv4 addresses, this may result in an “unknown host” error or of mapping to the standard IP address format. For IPv6, Java currently just checks the formatting of the address, without trying to look up the host on the Internet.

```
package planet.generic.commonapi.factory;

import java.io.IOException;
import java.io.BufferedReader; // ### mhb ###
import java.io.FileInputStream; // ### mhb ###
import java.io.InputStreamReader; // ### mhb ###
import java.io.FileNotFoundException; // ### mhb ###
import java.lang.reflect.Method;
import java.net.InetAddress; // ### mhb ###
import java.net.UnknownHostException; // ### mhb ###
import java.math.BigInteger;
import java.util.Random;

import planet.commonapi.Id;
import planet.commonapi.exception.InitializationException;
import planet.commonapi.factory.IdFactory;
import planet.util.Utilities;
import planet.util.Properties; // ### mhb ###

/**
 * This Factory generates unique Id from a material.
 * @author
 * <a href="mailto: jordi.pujol@estudiants.urv.es">Jordi Pujol</a>
 * 07-jul-2005
 */
public class IdFactoryImpl implements IdFactory {
```

```
[...]

    /** ### mhb ###
     * Attributes for input from file for HASHED topology
     */
    protected FileInputStream fileInput = null;
    protected BufferedReader fileBuffer = null;
    protected String readString = "NotNull";

[...]

    /**
     * Sets the specified initial values.
     * @param idClass Class reference for the current Id implementation.
     * @param topology Desired network topology.
     * @param networkSize Desired network size.
     * @return The same instance once it has been updated.
     * @throws InitializationException if any error occurs during the
     * initialization process.
     * @see
     * planet.commonapi.factory.IdFactory#setValues(java.lang.Class,
     * java.lang.String, int)
     */
    public IdFactory setValues(Class idClass, String topology,
    int networkSize)
    throws InitializationException {
        //get constructors for idClass
        this.idClass = idClass;

        try {
            Class types[]  = {int.class};
            idDivideMethod = idClass.getMethod("divide",types);
        } catch (Exception e) {
            throw new InitializationException(
                    "Cannot obtain divide(int) method for '"
                    + idClass.getName() +"'.", e);
        }

        //read defaultTopology of network
        this.topology = topology;
        if (!Topology.isValid(topology))
            throw new InitializationException("The topology '"
                  + topology
                  + "' is not a valid topology for building new Ids.");

        //initialize Random generator
        this.random = new Random();

        //read the defaultSize of network.
        this.networkSize = networkSize;
        if (this.networkSize < 0) {
            throw new InitializationException("The network size '"
                    + networkSize + "' is invalid.");
        }
```

```
        // ### mhb ### Open input file of list of
        // hostnames to be hashed
        if (this.topology.equalsIgnoreCase(Topology.HASHED)) {
            try {
                fileInput = new FileInputStream(
                        Properties.factoriesHostnames);
            } catch(FileNotFoundException e) {
                throw new InitializationException(
                        "Cannot find file with list of "
                        + "hostnames to be hashed.");
            }
            fileBuffer = new BufferedReader(
                        new InputStreamReader(fileInput));
        }

        initDistributedAttr();
        return this;
    }

[...]

    /**
     * Builds an Id with the actual configuration of
     * network topology and size.
     * Use the protected method buildRandomId() to build the Id if the
     * specified topology is random.
     * @return A new Id generated with the actual configuration.
     */
    public Id buildId() throws InitializationException {
        Id toReturn = null;
        if (this.topology.equalsIgnoreCase(Topology.RANDOM)) {
            toReturn = buildRandomId();
        } else if (this.topology.equalsIgnoreCase(Topology.CIRCULAR)) {
            if (this.actualBuiltIds >= this.networkSize ) {
                throw new InitializationException(
                        "Cannot build a new instance of ["
                        + idClass.getName()
                        + "]. The topology network is ["
                        + Topology.CIRCULAR
                        + "] and just generated all possible Ids ["
                        + this.networkSize+"].");
            }
            this.actualValue = this.actualValue.add(this.chunkValue);
            this.actualBuiltIds++;
            toReturn = actualValue;
        } else if (this.topology.equalsIgnoreCase(Topology.HASHED)) {
            /** ### mhb ###
             * Build Id for topology HASHED by reading
             * hostname from file,
             * converting it to an IP address (including IPv6 addresses)
             * and hashing the result. Because of the
             * PlanetSim architecture
             * this works for both Chord and Symphony alike,
```

```
 * without referring
 * to or testing for the specific protocol in the code here.
 */
if (this.actualBuiltIds >= this.networkSize ) {
    throw new InitializationException(
            "Cannot build a new instance of ["
            + idClass.getName()
            + "]. The topology network is ["
            + Topology.HASHED
            + "] and just generated all possible Ids ["
            + this.networkSize+"].");
}
// Read and hash next hostname from the input file,
// if any left
try {
    readString = fileBuffer.readLine();
} catch(IOException e) {}
if(readString != null){
    // Obtain IP address of host
    InetAddress hostIP = null;
    try {
        hostIP = InetAddress.getByName(readString);
    } catch (UnknownHostException e) {
        throw new InitializationException(
                "Cannot build a new instance of ["
                + idClass.getName()
                + "]. The topology network is ["
                + Topology.HASHED
                + "] and encountered unknown host ["
                + readString
                + "] in input.");
    }
    // Perform the hashing using IP address string
    // in textual presentation
    toReturn = buildId(hostIP.getHostAddress(),
            Properties.factoriesHashAlgorithm);
    // Increment Id count
    this.actualBuiltIds++;
    if ( this.actualBuiltIds == this.networkSize )
        // All Ids generated, so close file
        try {
            fileBuffer.close();
        } catch(IOException e) {}
} else {
    // No input left, so close file and throw exception
    try {
        fileBuffer.close();
    } catch(IOException e) {}
    throw new InitializationException(
            "Cannot build a new instance of ["
            + idClass.getName()
            + "]. The topology network is ["
            + Topology.HASHED
            + "] and ran out of input file "
```

```
                        + "entries to generate "
                        + " all requested Ids ["
                        + this.networkSize+"].");
            }
            /** ### mhb ###
             * End of HASHED code
             */
        } else {
            throw new InitializationException(
                    "Cannot build a new Id under the "
                    + "current network topology ["
                    + this.topology
                    + "].");
        }
        System.out.println("### mhb IdFactoryImpl.buildId ### Id = "
                    + toReturn);
        return toReturn;
    }

[...]

}
```

Java class “planet.generic.commonapi.factory.Topology”:

Here the new network topology HASHED is introduced as a valid input parameter in the corresponding configuration files.

```
package planet.generic.commonapi.factory;

/**
 * This class allows the programmer to specify
 * all the topologies of the
 * networks. Currently there exist four topologies:
 * <ul>
 * <li><b>HASHED ("Random")</b>:
 * Where the node IDs are hashed hostnames.</li> ### mhb ###
 * <li><b>RANDOM ("Random")</b>:
 * Where the Id of nodes are distributed randomly.</li>
 * <li><b>CIRCULAR ("Circular")</b>:
 * Where the Id of nodes are distributed uniformly.</li>
 * <li><b>SERIALIZED ("Serialized")</b>:
 * Where the entire ring are restored from serialized
 * state file.</li>
 * </ul>
 * @author Jordi Pujol
 * @see planet.commonapi.Id Id
 */
public class Topology {
      /** ### mhb ###
```

```
 * This topology specifies that the node IDs
 * are obtained by hashing
 * hostnames (Internet addresses) that are read from a file.
 */
public static final String HASHED = "Hashed";

/**
 * This topology specify that the Id of nodes
 * are distributed randomly.
 */
public static final String RANDOM = "Random";

/**
 * This topology specify that the Id of nodes
 * are distributed uniformly
 * in the ring.
 */
public static final String CIRCULAR = "Circular";

/**
 * This topology specify that the entire ring
 * are restored from serialized
 * state file.
 */
public static final String SERIALIZED = "Serialized";

/**
 * Identify if the <b>topology</b> specified is valid or not.
 * @param topology Topology to test if is valid.
 * @return true if <b>topology</b> is valid.
 * False in another case.
 */
public static boolean isValid(String topology) {
      return
      topology != null && (
            topology.equalsIgnoreCase(HASHED)   || // ### mhb ###
            topology.equalsIgnoreCase(RANDOM)   ||
            topology.equalsIgnoreCase(CIRCULAR) ||
            topology.equalsIgnoreCase(SERIALIZED));
}

/**
 * Inform when the specified <b>topology</b> is valid to
 * build <b>new instances with sentence <i>new</i></b>.
 * Actually, only the SERIALIZED topology is not valid.
 * @param topology Topology to test.
 * @return true if the topology is RANDOM or CIRCULAR. false
 * in other case.
 */
public static boolean isValidForNew(String topology) {
      return
            topology.equalsIgnoreCase(HASHED) || // ###  mhb ###
            topology.equalsIgnoreCase(RANDOM) ||
            topology.equalsIgnoreCase(CIRCULAR);
```

```
        }
}
```

Java class “planet.symphony.results.ResultsGMLGenerator”:

To customize Symphony graphics, the new package planet.symphony.results was added and the generic class ResultsGMLGenerator from planet.generic.commonapi.results copied here. Then the edges for long distance links were changed into dashed lines for better readability when printed. Node placement was changed so that the location on the circle corresponds to the node ID that is in the range from 0 to 1. A dummy node was added to the graph that shows the circle with unit perimeter on which the nodes are placed. Example graphics generated with this are displayed in Sections 3.4 and 3.5.

```
package planet.symphony.results; // ### mhb ###

import java.io.BufferedWriter;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;

import planet.commonapi.Network;
import planet.commonapi.Node;
import planet.commonapi.exception.InitializationException;
import planet.commonapi.results.ResultsConstraint;
import planet.commonapi.results.ResultsGenerator;
// ### mhb ### start
import planet.generic.commonapi.results.ResultsEdgeImpl;
import planet.generic.commonapi.results.ResultsGMLProperties;
import planet.generic.commonapi.results.ResultsNames;
import planet.symphony.SymphonyId;
// ### mhb ### end
import planet.util.Properties;

/**
 * @author
 * <a href="mailto: jordi.pujol@estudiants.urv.es">Jordi Pujol</a>
 * 15/02/2005
 */
public class ResultsGMLGenerator implements ResultsGenerator {

[...]

    /**
```

```
 * This method generates GML information into <b>out</b> file
 * for a stable Overlay ring using GML format.
 * @param network Network.
 * @param out Path of the file to write it out.
 * @param constraint Constraint used to select
 * edges for resulting Overlay graph
 * @param wholeNetworkLayout This boolean indicates
 * if we want to shown
 * all the nodes of the network.
 * @see
 * planet.commonapi.results.ResultsGenerator#generateResults(
 *      planet.commonapi.Network,
 *      java.lang.String,
 *      planet.commonapi.results.ResultsConstraint, boolean)
 */
public void generateResults(Network network, String out,
        ResultsConstraint constraint, boolean wholeNetworkLayout) {
    StringBuffer buffer = new StringBuffer(
            "graph [\n    " +
            "\tid 0\n" +
            "\tdirected 0\n" +
            "\thierarchic 1\n" +
            "\tlabel      \"\"\n");

    java.util.Iterator it = network.iterator();

    ResultsGMLProperties gmlProps = null;
    try {
        gmlProps = (ResultsGMLProperties)
              Properties.getResultsPropertiesInstance(
                    ResultsNames.GML);
    } catch (InitializationException e) {
        e.printStackTrace();
        System.exit(-1);
    }

    double radius = (network.size() * gmlProps.minimalNodeDistance)
                         / (2.0 * Math.PI);
    // ### mhb ### double da = (2.0 * Math.PI) / network.size();
    // x cartesian coordinate of the node
    double x;
    // y cartesian coordinate of the node
    double y;
    // Initial angle of the circular Identifier Space
    double a = 0.0;

    // Draw the circle with perimeter 1 as dummy node ### mhb ###
    buffer.append("\tnode [\n" +
            "\t\tid \"Circle\"\n" +
            "\t\tlabel \"Circle\"\n" +
            "\t\tgraphics\n" +
            "\t\t[\n" +
            "\t\t\tx  " + radius + "\n" +
            "\t\t\ty  " + radius + "\n" +
```

```
            "\t\t\tw  " + 2*radius + "\n" +
            "\t\t\th  " + 2*radius + "\n" +
            "\t\t\ttype  \"ellipse\"\n" +
            "\t\t\tfill  \"#CCCCCC\"\n" +
            "\t\t\toutline  \"#CCCCCC\"\n" +
            "\t\t]\n" +
            "\t\tLabelGraphics\n" +
            "\t\t[\n" +
            "\t\t\ttext  \"\"\n" +
            "\t\t\tfontSize     " + gmlProps.fontSize + "\n" +
            "\t\t\tfontName     \"" + gmlProps.fontName + "\"\n" +
            "\t\t\tmodel  \"null\"\n" +
            "\t\t\tanchor \"null\"\n" +
            "\t\t]\n" +
            "\t]\n");

    java.util.Collection E = new java.util.HashSet();
    //  node GML clause
    while (it.hasNext()) {
        Node node = (Node) it.next();
        // Calculate location on circle (angle)
        // from node ID ### mhb ###
        a = 2. * Math.PI *
              ((SymphonyId)node.getId()).getDoubleValue();
        x = radius * (1 + Math.cos(a));
        y = radius * (1 + Math.sin(a));

        /////////// GML Template header \\\\\\\\\\\\\
        String header = "\tnode [\n\t\tid \"" +
                node.getId() + "\"\n" +
                "\t\tlabel \"" + node.getId() + "\"\n" +
                "\t\tgraphics\n" +
                "\t\t[\n" +
                "\t\t\tx  " + x + "\n" +
                "\t\t\ty  " + y + "\n" +
                "\t\t\tw  " + gmlProps.width + "\n" +
                "\t\t\th  " + gmlProps.height + "\n" +
                "\t\t\ttype     \"" + gmlProps.shape + "\"\n";

        ////////// GML Template footer \\\\\\\\\\\\\\
        String footer = "\t\t\toutline      \"" +
                gmlProps.outline + "\"\n" +
                "\t\t]\n" +
                "\t\tLabelGraphics\n" +
                "\t\t[\n" +
                "\t\t\ttext     \"" + node.getId() + "\"\n" +
                "\t\t\tfontSize " + gmlProps.fontSize + "\n" +
                "\t\t\tfontName \"" + gmlProps.fontName + "\"\n" +
                "\t\t\tmodel    \"null\"\n" +
                "\t\t\tanchor   \"null\"\n" +
                "\t\t]\n" +
                "\t]\n";

        if (!wholeNetworkLayout) {
```

```
                if (constraint.isACompliantNode(node.getId()))
                    buffer.append(header + "\t\t\tfill      \""
                            + gmlProps.fill
                            + "\"\n" + footer);
            } else {
                if (constraint.isACompliantNode(node.getId())) {
                    buffer.append(header + "\t\t\tfill      \""
                            + gmlProps.fill
                            + "\"\n" + footer);
                } else {
                    buffer.append(header + "\t\t\tfill      \""
                            + gmlProps.alternativeFill + "\"\n"
                            + footer);
                }
            }
            // ### mhb ### a = a + da;
        }

        it = network.iterator();
        while (it.hasNext()) {
            Node node = (Node) it.next();
            node.buildEdges(ResultsNames.GML,E, constraint);
        }

        it = E.iterator();
        while (it.hasNext()) {
            ResultsEdgeImpl e = (ResultsEdgeImpl) it.next();
/* ### mhb ### buffer.append("\tedge [\n\t\tsource \"" +
                    e.getSource() + "\"\n" +
                    "\t\ttarget \"" + e.getTarget() + "\"\n" +
                    "\t\tlabel      \"\"\n" +
                    "\t\tgraphics\n" +
                    "\t\t[\n" +
                    "\t\t\tfill     \"" +  e.getFill() + "\"\n" +
                    "\t\t]\n" +
                    "\t\tLabelGraphics\n" +
                    "\t\t[\n" +
                    "\t\t]\n" +
                    "\t]\n"); */
            // Make red lines (long-distance links) dashed,
            // others solid ### mhb ###
            String lineFill = e.getFill();
            String lineStyle = "solid";
            if( lineFill.equalsIgnoreCase("#FF0000") )
                    lineStyle = "dashed";
            buffer.append("\tedge [\n\t\tsource \"" + e.getSource() +
                    "\"\n" +
                    "\t\ttarget \"" + e.getTarget() + "\"\n" +
                    "\t\tlabel      \"\"\n" +
                    "\t\tgraphics\n" +
                    "\t\t[\n" +
                    "\t\t\tfill     \"" +  lineFill + "\"\n" +
                    "\t\t\tstyle \"" +  lineStyle + "\"\n" +
                    "\t\t]\n" +
```

```
                    "\t\tLabelGraphics\n" +
                    "\t\t[\n" +
                    "\t\t]\n" +
                    "\t]\n");
        }
        buffer.append("]");
        try {
            FileOutputStream F = new FileOutputStream(out);
            BufferedWriter BF = new BufferedWriter(
                                    new OutputStreamWriter(F));
            BF.write(buffer.toString());
            BF.close();
        } catch (IOException e) {
        }

    }

}
```

Java class “planet.test.dht2.DHTApplication”:

Previously hard coded specification of SHA-1 hashing is changed to make use of the new FACTORIES_HASHALGORITHM configuration parameter. This allows use of all the hash algorithms in the added Jacksum package for this DHT test.

```
package planet.test.dht2;

import java.util.Hashtable;
import java.util.Iterator;
import java.util.Set;
import java.util.Vector;

import planet.commonapi.Application;
import planet.commonapi.EndPoint;
import planet.commonapi.Id;
import planet.commonapi.NodeHandle;
import planet.commonapi.exception.InitializationException;
import planet.generic.commonapi.factory.GenericFactory;
import planet.util.Properties; // ### mhb ###

/**
 * Application that contains all key/values pairs,
 * required for owner Node Id.
 *
 * @author Carles Pairot <cpairot@etse.urv.es>
 * @author Jordi Pujol <jordi.pujol@estudiants.urv.es>
```

```
 * @author Marc Sanchez <marc.sanchez@estudiants.usrv.es>
 */
public class DHTApplication implements Application {
    public static final String APPLICATION_ID =
            "SymphonyDHTApplication";

    private String appId = APPLICATION_ID;
    private Id id;
    private Hashtable data = new Hashtable();
    private EndPoint endPoint = null;

[...]

    /**
     * This method retrieves value from a key
     * from the dsitributed repository build on top of
     * Symphony Lookup protocol.
     * @param msg The message exchanged between applications.
     */
    public void retrieve(DHTMessage msg) {
        String key = msg.getKey();
        if (msg.getVectorValue() == null) {
            DHTMessage reply = new DHTMessage(this.id,
            DHTMessage.LOOKUP, key, (Vector) data.get(key));
            endPoint.route(msg.getSource(), reply, null);
        } else {
            try {
                Id target = GenericFactory.buildId(key,
                        Properties.factoriesHashAlgorithm);
                /* ### mhb ### Id target =
                        GenericFactory.buildId(key, "SHA"); */
                System.out.println("\n\tINFO: '"
                        + msg.getVectorValue()
                        + "' Bound to '" + target + "'\t....OK\n");
            } catch (InitializationException e) {
                e.printStackTrace();
            }
        }
    }

[...]

    /**
     * Owner SymphonyDHTApplication method
     * which permits to send a Message with a
     * key/value pair, for INSERT.
     * IMPORTANT: the <b>key </b> is in hexadecimal format.
     *
     * @param key
     *            Text to be used as material for
     *            construct Message Id, in
     *            hexadecimal format.
     * @param value
     *            Related value to that key.
```

```
     */
    public void insert(String key, String value) {
        try {
            DHTMessage msg = new DHTMessage(this.id,
                                    DHTMessage.INSERT, key, value);
            Id target = GenericFactory.buildId(key,
                                  Properties.factoriesHashAlgorithm);
            /* ### mhb ### Id target =
                  GenericFactory.buildId(key, "SHA"); */
            System.out.println(" INSERT '" + target
                  + "'\t Tied To '" + value + "'");
            endPoint.route(target, msg, null);
        } catch (InitializationException e) {
            e.printStackTrace();
        }
    }
    /**
     * Owner SymphonyDHTApplication method
     * which permits to send a Message with a
     * key/value pair, for LOOKUP.
     * IMPORTANT: the <b>key </b> is in hexadecimal format.
     *
     * @param key
     *            Text to be used as material for construct
     *            Message Id, in hexadecimal format.
     */
    public void lookup(String key) {
        try {
            DHTMessage msg = new DHTMessage(this.id,
                  DHTMessage.LOOKUP, key, (Vector) null);
            Id target = GenericFactory.buildId(key,
                  Properties.factoriesHashAlgorithm);
            /* ### mhb ### Id target =
                  GenericFactory.buildId(key, "SHA"); */
            System.out.println(" LOOKUP Of Target '" + target + "'");
            endPoint.route(target, msg, null);
        } catch (InitializationException e) {
            e.printStackTrace();
        }
    }

[...]

}
```

Java class “planet.test.dht2.DHTTest”:

This is the executable main program used for generating the results reported in this thesis.

It is part of the PlanetSim distribution, but needed to be modified as shown to generate a

graphical representation of the network in GML format (as described in the User and Developer Tutorial distributed with PlanetSim). The generated ".gml" file is always placed in folder "out". The resulting graph can be displayed by double clicking on "yed.jar" in folder "lib" and opening the ".gml" file.

```
package planet.test.dht2;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;

import planet.commonapi.Network;
import planet.commonapi.exception.InitializationException;
import planet.generic.commonapi.GenericApp;
import planet.generic.commonapi.factory.GenericFactory;
import planet.generic.commonapi.results.ResultsNames; // ### mhb ###
import planet.simulate.Results;
import planet.test.TestNames;
import planet.util.Properties;

/**
 * Main application that tests to inserts a wide number of key/value
 * pairs at the ring and looks up a concrete existing key.
 * @author Carles Pairot   <cpairot@etse.urv.es>
 * @author Jordi Pujol     <jordi.pujol@estudiants.urv.es>
 * @author Marc Sanchez    <marc.sanchez@estudiants.urv.es>
 * @version 1.0
 */
public class DHTTest extends GenericApp {

    /**
     * Constructor that initialize a network with MAX_NODES,
     * and register over each node
     * an instance of SymphonyDHTApplication.
     */
    public DHTTest() throws InitializationException {
        // arguments: properties file, application level,
        // events, results, serialization
        /** ### mhb ###
         * This for turning GML output on:
         */
         super("../conf/master.properties",TestNames.DHT2_DHTTEST,
                 true,false,true,true);
         /* super("../conf/master.properties",TestNames.DHT2_DHTTEST,
                true,false,false,true); */

        try {
```

```
            System.out.println("Starting creation of "
                  + Properties.factoriesNetworkSize + " nodes...");

            long t1 = System.currentTimeMillis();
            // build network but not is stabilize
            Network network = GenericFactory.buildNetwork();
            // stabilize network
            int steps = network.stabilize();
            // register application
            network.registerApplicationAll();
            long t2 = System.currentTimeMillis();
            System.out.println(Properties.factoriesNetworkSize
                  + " nodes created OK with [" + steps
                  + "] steps and ["
                  + GenericApp.timeElapsedInSeconds(t1,t2)
                  + "] seconds.\n");
            System.out.println("RouteMessages: Created["
                  + GenericFactory.getBuiltRouteMessages()
                  + "], reused["
                  + GenericFactory.getReusedRouteMessages()
                  + "], free["
                  + GenericFactory.getFreeRouteMessages() + "]");
            network.printNodes();
            Results.resetInserts();
            /** ### mhb ###
             * Generate graphical representation of network
             * in GML format
             */
            GenericFactory.generateResults(ResultsNames.GML, network,
                    "../out/network.gml",
                    GenericFactory.buildConstraint(ResultsNames.GML),
                    true);

[...]

}
```

Java class “planet.util.Properties”:

Here it is shown how the new configuration parameters FACTORIES_HOSTNAMES (for building networks from hostnames in a file, including IPv6 addresses) and FACTORIES_HASHALGORITHM (the hash algorithm to be used in all hashing operations) are added to PlanetSim as valid input parameters.

```
package planet.util;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.StringTokenizer;
import java.util.TreeMap;
import java.util.Vector;

import planet.commonapi.exception.InitializationException;

/**
 * This class loads all mandatory attributes
 * of the specified properties file.
 * The loading process follows these steps:

[...]

 * @author <a href="jordi.pujol@estudiants.urv.es">Jordi Pujol</a>
 * @author <a href="marc.sanchez@urv.net">Marc Sanchez</a>
 * 30-jun-2005
 */
public class Properties {
    /* ************* FACTORIES PROPERTIES NAMES *****************/

    /* REQUIRED ATTRIBUTES */
    /* Factories: */

[...]

    /**
     * Factories property: Default key specified
     * in the properties file that
     * identifies the network topology.
     */
    public static final String FACTORIES_NETWORKTOPOLOGY
            = "FACTORIES_NETWORKTOPOLOGY";

    /** ### mhb ###
     * Factories property: Default key specified
     * in the properties file that
     * identifies the file containing network
     * hostnames for topology HASHED.
     */
    public static final String FACTORIES_HOSTNAMES
            = "FACTORIES_HOSTNAMES";

    /** ### mhb ###
     * Factories property: Default key specified
     * in the properties file that
     * identifies the hash algorithm to be used for key
     * and id generation.
     */
    public static final String FACTORIES_HASHALGORITHM
```

```
            = "FACTORIES_HASHALGORITHM";

    /**
     * Factories property: Default key specified
     * in the properties file that
     * identifies the network size.
     */
    public static final String FACTORIES_NETWORKSIZE
            = "FACTORIES_NETWORKSIZE";

 [...]

}
```

Java class “planet.util.Utilities”:

Here is the core of the addition of Jacksum into PlanetSim, for making available a wide variety of hashing algorithms in PlanetSim. There is test code surrounding the calls to the jonelo.jacksum package to output and check the hashing result in various formats (hexadecimal, decimal, and mapped to the unit perimeter circle, for example). A Jacksum example snippet from the Jacksum Web site (http://www.jonelo.de/java/jacksum/) was adapted. The java.math.BigDecimal class is used to map to the unit circle at full precision (mostly for Symphony testing).

```
package planet.util;

// ### mhb ### import java.security.*;

// ### mhb ### added imports start
import java.security.NoSuchAlgorithmException;
import java.math.BigInteger;
import java.math.BigDecimal;

import jonelo.jacksum.*;
import jonelo.jacksum.algorithm.*;

import planet.commonapi.exception.InitializationException;
import planet.util.Properties;
### mhb ### added imports end

/**
 * Offers different utilities.
 * @author Jordi Pujol
```

```
 * @author Pedro Garcia
 */
public class Utilities {

[...]

    /**
     * Generate a hash code from specified data.
     * @param data String to obtain its hash code.
     * @return null if no hash result could be obtained,
     * or result of hash function in
     * int array format.
     */
    public static int[] generateIntHash(String data)
    throws InitializationException {
        // ### mhb ### MessageDigest md = null;
        byte[] digest = null;

        /* ### mhb ###
        try {
            md = MessageDigest.getInstance("SHA");
            md.update(data.getBytes());
            digest = md.digest();
        } catch (NoSuchAlgorithmException e) {
            System.err.println("No SHA support!");
        }*/
        // ### mhb ###
        try {
            digest = generateByteHash(data,
                    Properties.factoriesHashAlgorithm);
        } catch (InitializationException e) {
            throw e;
        }

        return toIntArray(digest);
    }

    /**
     * Generate a hash code from specified data.
     * Hash function is as given in
     * configuration file ### mhb ###.
     * @param data String to obtain its hash code.
     * @return null if no hash result could be obtained,
     * or result of hash function
     * in byte array format.
     */
    public static byte[] generateByteHash(String data)
    throws InitializationException {
        // ### mhb ### MessageDigest md = null;
        byte[] digest = null;

        /* ### mhb ###
         try {
            md = MessageDigest.getInstance("SHA");
```

```
            md.update(data.getBytes());
            digest = md.digest();
        } catch (NoSuchAlgorithmException e) {
            System.err.println("No SHA support!");
        } */
        // ### mhb ###
        try {
            digest = generateByteHash(data,
                  Properties.factoriesHashAlgorithm);
        } catch (InitializationException e) {
            throw e;
        }

        return digest;
    }

    /** ### mhb ###
     * Generate a hash code from specified data, using the specified
     * hash algorithm.
     * Uses Jacksum <http://www.jonelo.de/java/jacksum/> to perform
     * hashing. Jacksum uses Java's MessageDigest when possible, but
     * has implementations of additional algorithms. For example,
     * Sun Java 1.5 does not support SHA-224, but with this extension
     * SHA-224 is available. See Jacksum documentation for all
     * algorithms available.
     *
     * @param material String for which to obtain its hash code.
     * @param algorithm String indicating the desired hash algorithm.
     * @return null if no hash result could be obtained, or result of
     * hash function in byte array format.
     */
    public static byte[] generateByteHash(String material,
    String algorithm)
    throws InitializationException {
        byte[] digest = null;

        AbstractChecksum checksum = null;
        try {
            System.out.println(
                    "### mhb Utilities.generateByteHash ### "
                    + "Input: " + material + ", " + algorithm);
            // Select an algorithm
            checksum =
                  JacksumAPI.getChecksumInstance(
                        algorithm.toLowerCase());
            // On some systems you get a better
            // performance for particular
            // algorithms if you select an alternate algorithm
            // (see also Jacksum option -A)
            // checksum = JacksumAPI.getChecksumInstance(
            //          algorithm, true);
            checksum.update(material.getBytes());
            digest = checksum.getByteArray();
        } catch (NoSuchAlgorithmException e) {
```

```
            // Algorithm doesn't exist
            System.out.println(
                    "### mhb Utilities.generateByteHash ### "
                    + "No such algorithm exception...");
            throw new InitializationException(
                  "Algorithm not supported!", e);
        }
        int digestLengthBits = 8*digest.length;
        System.out.println("### mhb Utilities.generateByteHash ### "
                + "Digest length: " + digestLengthBits + " bits");
        String digestHex = Utilities.byteToHex(digest);
        System.out.println("### mhb Utilities.generateByteHash ### "
                + "Digest (hex): " + digestHex);
        BigInteger digestDec = new BigInteger(digestHex ,16);
        System.out.println("### mhb Utilities.generateByteHash ### "
                + "Digest (dec): " + digestDec);
        // For Chord, PlanetSim interprets the digest
        // backwards in blocks of 32 bits = 8 hex digits, so do this:
        String digestReverseHex32Bits =
                Utilities.reverseHex32Bits(digestHex);
        System.out.println("### mhb Utilities.generateByteHash ### "
                + "Digest (hex in 32-bit reverse): "
                + digestReverseHex32Bits);
        BigInteger digestDecFromReverseHex =
                new BigInteger(digestReverseHex32Bits ,16);
        System.out.println(
                "### mhb Utilities.generateByteHash ### "
                + "Digest (dec from hex in 32-bit reverse): "
                + digestDecFromReverseHex);
        // In Symphony, all IDs are on a unit circle with values 0...1,
        // so calculate and output this ID value from the digest
        // for double checking (at full precision if desired), by
        // dividing the hash value by 2^H where H is the length of
        // the hash (for example, H = 512 for SHA-512).
        BigDecimal digestDecUnitCircle = new BigDecimal(digestDec);
        BigInteger digestDenumerator = new BigInteger("2");
        digestDenumerator = digestDenumerator.pow(digestLengthBits);
        digestDecUnitCircle = digestDecUnitCircle.divide(
                new BigDecimal(
                digestDenumerator), 600,
                BigDecimal.ROUND_HALF_UP); // Second arg
        // determines number of digits; 600 digits will give
        // full precision for 512-bit hash.
        System.out.println("### mhb Utilities.generateByteHash ### "
                + "Digest (dec on unit circle): "
                + digestDecUnitCircle);

        return digest;
    }

[...]

    /** ### mhb ###
     * Converts array of bytes into string of hex.
```

```
     * This method is similar to
     * "byteArrayToHexStr(byte[])" in Listing 24 of
     * "Java Programming, Note #729: Message Digests 101 using Java"
     * by R. G. Baldwin (2005) available on-line.
     *
     * @param byteArray The array of bytes
     * @return The string of hex
     */
    public static String byteToHex(byte[] byteArray) {
        String hexString = "";
        String tmpString = "";
        for( int i=0; i<byteArray.length; i++ ){
            tmpString = java.lang.Integer.toHexString(byteArray[i]
                & 0XFF);
            if( tmpString.length()==1 )
                hexString += "0" + tmpString;
            else
                hexString += tmpString;
        }
        return hexString.toUpperCase();
    }

    /** ### mhb ###
     * Reverse string (of hex digits) in blocks of 8 characters
     * (8 hex digits = 32 bits)
     * @param in The string to be reversed
     * @return The reversed string
     */
    public static String reverseHex32Bits(String in) {
        String out = "";
        if( in.length()%8 != 0){
            throw new Error("String length not divisible by 8.");
        }
        for(int i=0; i<in.length()/8; i++){
            out += in.substring( in.length()-8*i-8, in.length()-8*i );
        }
        return out;
    }
}
```

Configuration file “master.properties”:

It is here where the choice “Chord” or “Symphony” is made by the user. Just one line needs to be changed before running the simulator to pick one of the two protocols. To run the simulator, see below for the description of the corresponding Windows batch file.

```
################################################################
# Main configuration file:
#
# -----------------------------
#
[...]
#
# Made by:
#
#  Jordi Pujol Ahullo (jordi.pujol@estudiants.urv.es)
#
# Under:
#
#  Planet Project: http://ants.etse.urv.es/planet
#
#  PlanetSim:      htpp://ants.etse.urv.es/planetsim
#
################################################################

[...]

################################################################
DHT2_DHTTEST = ../conf/chord_dht2.properties
### mhb ### DHT2_DHTTEST = ../conf/symphony_dht2.properties

[...]
```

Configuration file “chord_dht2.properties”:

In this file, parameters for Chord simulations are specified.

```
################################################################
# Chord configuration file:
#
# -----------------------------
#
[...]
#
# Made by:
#
#  Jordi Pujol Ahullo (jordi.pujol@estudiants.urv.es)
#
# Under:
#
#  Planet Project: http://ants.etse.urv.es/planet
#
#  PlanetSim:      htpp://ants.etse.urv.es/planetsim
#
################################################################
```

```
#################################################################
# FACTORIES PART
#
#################################################################

########## MANDATORY ATTRIBUTES

[...]

# The default network topology.
# Default possible values: RANDOM, CIRCULAR,
# HASHED ### mhb ###, and SERIALIZED
FACTORIES_NETWORKTOPOLOGY = HASHED
### mhb ### FACTORIES_NETWORKTOPOLOGY = CIRCULAR
### mhb ### FACTORIES_NETWORKTOPOLOGY = RANDOM

### mhb ###
# The default filename for hostnames to be read
# for network topology HASHED
FACTORIES_HOSTNAMES = data/Hostnames.txt

### mhb ###
# The default hashing algorithm for keys and node IDs
# Possible values are:
#       SHA (defaults to SHA-1), SHA-1, SHA-224, SHA-256,
#       SHA-384, SHA-512,
#       MD2, MD5 plus many others (see Jacksum documentation)
# SHA-224 is not supported by Sun Java 1.5 but provided
# by Jacksum since Jacksum 1.6.1 <http://www.jonelo.de/java/jacksum/>
### mhb ### FACTORIES_HASHALGORITHM = SHA-1
FACTORIES_HASHALGORITHM = SHA-512

# The default initial network size
### mhb ### FACTORIES_NETWORKSIZE = 1000
FACTORIES_NETWORKSIZE = 100

[...]

#################################################################
# CHORD SPECIFIC PART
#
#################################################################

########## MANDATORY ATTRIBUTES

[...]

# The default number of bits for ChordIds
CHORD_BITS_PER_KEY = 32
### mhb ### CHORD_BITS_PER_KEY = 512

[...]
```

Configuration file “symphony_dht2.properties”:

Here the user specifies parameters for Symphony simulation runs.

```
#################################################################
# Symphony configuration file:
#
# -----------------------------
#
[...]
#
# Made by:
#
#  Jordi Pujol Ahullo (jordi.pujol@estudiants.urv.es)
#
# Under:
#
#  Planet Project: http://ants.etse.urv.es/planet
#
#  PlanetSim:      htpp://ants.etse.urv.es/planetsim
#
#################################################################

#################################################################
# FACTORIES PART
#
#################################################################

########## MANDATORY ATTRIBUTES

[...]

# The default network topology.
# Default possible values:
#     RANDOM |CIRCULAR | HASHED ### mhb ### | SERIALIZED
FACTORIES_NETWORKTOPOLOGY = HASHED
### mhb ### FACTORIES_NETWORKTOPOLOGY = CIRCULAR
### mhb ### FACTORIES_NETWORKTOPOLOGY = RANDOM

### mhb ###
# The default filename for hostnames to be read
# for network topology HASHED
FACTORIES_HOSTNAMES = data/Hostnames.txt

### mhb ###
# The default hashing algorithm for keys and node IDs
# Possible values are:
#      SHA (defaults to SHA-1), SHA-1,
#      SHA-224, SHA-256, SHA-384, SHA-512,
#      MD2, MD5 plus many others (see Jacksum documentation)
# SHA-224 is not supported by Sun Java 1.5 but provided
# by Jacksum since Jacksum 1.6.1 <http://www.jonelo.de/java/jacksum/>
### mhb ### FACTORIES_HASHALGORITHM = SHA-1
```

```
FACTORIES_HASHALGORITHM = SHA-224

# The default initial network size
FACTORIES_NETWORKSIZE = 1000
### mhb ### FACTORIES_NETWORKSIZE = 8000

[...]

#################################################################
# SYMPHONY SPECIFIC PART
#
#################################################################

########## MANDATORY ATTRIBUTES

# The default number of long distance connections
# Default value: 2
### mhb ### SYMPHONY_MAX_LONG_DISTANCE = 16
SYMPHONY_MAX_LONG_DISTANCE = 2

[...]

#################################################################
# RESULTS PART
#
#################################################################

[...]

########## OPTIONAL ATTRIBUTES: Test dependant

[...]

# The default ResultsGenerator class
### mhb ### RESULTS_GENERATOR =
planet.generic.commonapi.results.ResultsGMLGenerator, \
### mhb ### planet.generic.commonapi.results.ResultsPajekGenerator
RESULTS_GENERATOR = planet.symphony.results.ResultsGMLGenerator, \
      planet.generic.commonapi.results.ResultsPajekGenerator

[...]

#################################################################
# GML SPECIFIC RESULTS PART
#
#################################################################

########## OPTIONAL ATTRIBUTES: Test dependant

# The default width of the virtual bounding box
RESULTS_PROPERTIES_GML_WIDTH = 5.0f
### mhb ### RESULTS_PROPERTIES_GML_WIDTH = 20.0f

# The default height of the virtual bounding box
```

```
RESULTS_PROPERTIES_GML_HEIGHT = 5.0f
### mhb ### RESULTS_PROPERTIES_GML_HEIGHT = 20.0f

[...]

# The default font size of the node Id level
RESULTS_PROPERTIES_GML_FONT_SIZE = 8
### mhb ### RESULTS_PROPERTIES_GML_FONT_SIZE = 12

[...]
```

Windows batch file “dht2.DHTTest.bat”:

To run the simulator on a machine with the Windows operating system and where a Java runtime environment is properly installed, one should be able to open a command window, change to the “bin” directory of the extended PlanetSim, and type

```
dht2.DHTTest.bat > ..\out\dht2_out.txt
```

After the simulation finishes one can examine the results in the “out” folder using any text editor. The batch file itself contains the following as a single line:

```
@java -Xms256M -Xmx1024M
-cp ../lib/planetsim_3.0_mhb.jar;../lib/jacksum.jar
planet.test.dht2.DHTTest
```