UNIVERSITY OF THE WITWATERSRAND
JOHANNESBURG

FACULTY OF SCIENCE
School of Computer Science

A PERFORMANCE EVALUATION OF PEER-TO-PEER STORAGE SYSTEMS

Deya Mwembya Mutombo

A Dissertation submitted to the Faculty of Science, University of the Witwatersrand, in fulfilment of the requirements for degree of Master of Science

Supervisor: Professor Ekow Otoo

Johannesburg, October 30, 2012

Declaration

I declare that this dissertation is my own, unaided work.
It is being submitted for Degree of Master of Science as applicable at the University of the Witwatersrand Johannesburg.
It has not been submitted before for any degree or examination at any other University.

Johannesburg, October 30, 2012 _____

Deya Mwembya Mutombo

Abstract

This work evaluates the performance of Peer-to-Peer storage systems in structured Peer-to-Peer (P2P) networks under the impacts of a continuous process of nodes joining and leaving the network (Churn). Based on the Distributed Hash Tables (DHT), the peer-to-peer systems provide the means to store data among a large and dynamic set of participating host nodes. We consider the fact that existing solutions do not tolerate a high Churn rate or are not really scalable in terms of number of stored data blocks. The considered performance metrics include number of data blocks lost, bandwidth consumption, latencies and distance of matched lookups. We have selected Pastry, Chord and Kademlia to evaluate the effect of inopportune connections/disconnections in Peer-to-Peer storage systems, because these selected P2P networks possess distinctive characteristics.

Chord is one of the first structured P2P networks that implements Distributed Hash Tables (DHTs). Similar to Chord, Pastry is based on a ring structure, with the identifier space forming the ring. However, Pastry uses a different algorithm than Chord to select the overlay neighbors of a peer. Kademlia is a more recent structured P2P network, with the XOR mechanism for improving distance calculation. DHT deployments are characterized by Churn. But if the frequency of Churn is too high, data blocks can be lost and lookup mechanism begin to incur delays. In architectures that employ DHTs, the choice of algorithm for data replication and maintenance can have a significant impact on the performance and reliability. PAST is a persistent Peer-to-Peer storage utility, which replicates complete files on multiple nodes, and uses Pastry for message routing and content location.

The hypothesis is that by enhancing the Churn tolerance through building a really efficient replication and maintenance mechanisms, it will:

i) Operate better than a peer-to-peer storage system such as PAST especially in replica placement strategy with a fewer data transfers.

ii) Resolve file lookups with a match that is closer to the source peer, thus conserving bandwidth.

Our research will involve a series of simulation studies using two network simulators OverSim and OMNeT++. The main results are:

- Our approach achieves a higher data availability in presence of Churn, than the original PAST replication strategy;
- For a Churn occuring every minute our strategy loses two times less blocks than PAST;
- Our replication strategy induces an average of twice less block transfers than PAST.

Dedication

First of all, I owe many thanks to my Lord Jesus Christ who made everything happen.

At the same time, I would like to thank my parents, for their support and encouragement throughout my years of studies. They have guided me both in study and in life, and I hope what I have done and what I will be doing will make the whole Mutombo family proud.

Last but not least, I am deeply indebt to my lovely wife Mutonji Mutombo and my first born Sharon Mutombo. They helped me through the entire year of my Master studies despite the short time I got for them. I must thank them for giving me care when I needed most.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Prof. Ekow Otoo, for his guidance and patience. His advices, insights and comments have helped me tremendously throughout my master year. Working under Prof. Ekow Otoo supervision enriched my experience greatly in being a researcher.

I am particularly grateful to Dr. Dmitry Shkatov, for his practical advices. This dissertation would be impossible without his guidance.

I highly appreciated the great working environment at our Computer Science Department and the University of the Witwatersrand, as a whole, which I found it a very pleasant place for my studies.

Contents

1 Introduction **1**

1.1 Introduction 1

1.2 Motivation and Contributions 2

1.3 Methodology 3

1.4 Organization of the Dissertation 3

2 Overview of some P2P networks **4**

2.1 Distributed Hash Tables (DHTs) 7

2.1.1 Iterative routing 8

2.1.2 Chord Protocol 9

2.1.3 Pastry Protocol 12

2.1.4 Kademlia Protocol 15

2.1.5 Replication techniques for implementing the DHT layer . . . 19

2.2 PAST 20

2.2.1 PAST Architecture 21

2.3 Replication in DHTs 23

2.4 Replica placement protocols 23

2.4.1 *Leafset-based* replication 23

2.4.2 Multiple key replication 24

2.5 Maintenance protocols 25

2.6 Impact of the Churn on the DHT performance 26

3 Free_DHT **27**

3.1 Placement Strategy of Free_DHT 27

3.1.1 Maintenance protocol 28

3.1.2 Maintenance message treatment 31

3.1.3 End of a lease treatment 33

4 Implementation and Simulation **34**

4.1 Simulation evironment 34

4.1.1 OMNeT++ Discrete Event Simulator 34

4.1.2 INET Framework for OMNeT++ 34

4.1.3 OverSim 35

4.2 Distribution used in the simulations 37

4.3 The OverSim Simulator Classes 39

5 Performance Evaluation **41**

5.1 Simulation setup 41

5.2 Simulation Result 44

5.2.1 Losses and stabilization time after one hour Churn 46

5.2.2 Continuous Churn 47

5.2.3 Maintenance protocol cost 49

5.2.4 Side effects and limitations 51

5.2.5 Data blocks distribution 51

5.2.6 Possible impact of *Bloom Filters* 52

5.2.7 Lookups processes assessment 52

6 Conclusion and Future Work **56**

6.1 Validation 56

6.2 Future Work 57

A Appendix: OverSim parameter files **64**

A.1 Default.ini 64

A.2 Omnetpp.ini 73

B Appendix: Moditions to OverSim code **78**

C Appendix: MyClass **79**

C.1 MyClass.h 79

C.2 MyClass.cc 80

D Appendix: Free_DHT Data Storage **86**

D.1 Free_DHTDataStorage.h 86

D.2 Free_DHTDataStorage.cc 89

List of Figures

2.1 Overaly Network 5
2.2 Hashing peers and items into the identifier space 6
2.3 Example of Chord with an identifier space 10
2.4 Chord's Lookup process, Node 17 looking for Key 13 11
2.5 Pastry overlay structure and routing principle 15
2.6 Kademlia's Subtree model 16
2.7 Kademlia's Lookup process 18
2.8 Structure of a DHT based system 20
2.9 Leaf set based contiguous replication 24
2.10 Multiple key based replication 25

3.1 Data structures managed on each peer 29
3.2 Message composed of x AddData elements and y NewRoot elements 32

4.1 OverSim architechture 35
4.2 Screen shot for OverSim 37
4.3 Churn model repreeenting the ON or OFF behaviour of peer i . . . 38
4.4 OverSim libraries and modified classes 40

5.1 Components of a Pastry simple network simulation model. 42
5.2 Modules used in a Pastry simulation on an Ipv4 network. 43
5.3 Simulation of a Pastry DHT on an Ipv4 network with 100 peers . . . 44
5.4 Number of data block lost 45
5.5 Number of exchanged data blocks to restore a stable state 46

5.6 Recovery time: time for retreiving all the copies of every remaining data block. 47

5.7 Number of data blocks losses while the system is under continuous Churn, varying inter-perturbation delay 48

5.8 Number of data blocks transfers required while the system is under continuous Churn, varying inter-perturbation delay 49

5.9 A plot of average hop count under the effects of Churn 53

5.10 A plot of the average bandwidth under the effects of Churn 54

5.11 A plot of average latency under the effects of Churn 55

5.12 A plot of average delivery ratio for the iterative and recursive routing 55

List of Tables

2.1 Pastry Routing table 14
2.2 XOR table 16
2.3 XOR metric 16
2.4 Description of different nodes 19

List of Algorithms

1 Maintenance message construction 30
2 Maintenance message reception 32

Chapter 1

Introduction

1.1 Introduction

In the last couple of years peer-to-peer applications have attracted a lot of public attention. In this dissertation, we study and evaluate the performance of replication strategies and lookup mechanism in Peer-to-Peer storage system. Firstly, we implement a DHT layer relying on an existing structured overlays; Pastry and Chord, and then simulate those protocols using Omnet++ and OverSim simulator.

Distributed Hash Tables (or DHTs) (Stoica et al. (2003), Zhao et al. (2003), Rowstron & Druschel (2001*a*), Ratnasamy, Francis, Handley, Karp & Shenker (2001)), are distributed systems that allow efficient lookup of identifiers by routing to the corresponding nodes. They achieve this by imposing on the routing tables of nodes a rigid structure that guarantees quick convergence to a target.

Also, Distributed Hash Tables (Rowstron & Druschel (2001*b*), Stoica et al. (2003)), are distributed storage services that use a structured overlay based on key-based routing (KBR) protocols (Dabek et al. (2003)). DHTs provide the system designer with a powerful abstraction for wide area persistent storage, hiding the complexity of network routing, replication, and fault-tolerance. Therefore, DHTs are increasingly use for dependable and secure applications like backup systems (Landers et al. (2004)), content distribution systems (Jernberg et al. (2006)), and distributed file systems (Busca et al. (2005), Dabek et al. (2001)).

A major problem in any peer-to-peer (P2P) application is that peers are free to join and temporarily leave (for long or short times) the system at any time. Some peers can fail due to software or hardware problems thus they permanently leave the system. This leave/join phenomenon is referred to as "Churn". In general, joining the system has no remarkable impact on the system. It can add some delay in routing results until the system detects the new nodes that have joined and updates the pointers of data location toward the right nodes. However, the departure and

failure events have an important negative impact because they may cause, at worse, data losses and degradation in performance of file lookups (Akavipat et al. (2010)), due to the reorganization of the set of replicas of the affected data, that consumes bandwidth and CPU cycles.

Churn resilience is a problem that has been mostly addressed at the P2P routing level of existing DHT implementations to ensure the reachability of peers by maintaining the consistency of the logical neighborhood, i.e., the *Leafset* of a peer (Rhea, Geels, Roscoe, & Kubiatowicz (2004), Castro et al. (2004), Legtchenko et al. (2009)). Reconfiguring the peers at storage level while avoiding data migration is a way to optimize bandwidth consumption. *Leafset* of a node is a peer's logical neighbourhood in the logical ring.

1.2 Motivation and Contributions

In DHT, each data block is associated with a root peer whose identifier is the (numerically) closest to its key, and Churn can limit the availability of a given object (Yang et al. (2010)). In order to minimize the cost of object location. Most DHTs store replicas at the set of nodes which are closest to the object key. These are called the replica set (Dabek et al. (2003)). The traditional replication scheme relies on using the subset of the root *Leafset* containing the closest logical peers to store the copies of a data block (Rowstron & Druschel (2001*b*)). Therefore, if a peer joins or leaves the *Leafset*, the DHT enforces the placement constraint on the closest peers and may migrate many data blocks. In fact, it has been shown that the cost of these migrations can be high in terms of bandwidth consumption (Landers et al. (2004)). A solution to this problem relies on creating multiple keys for a single data block (Ratnasamy, Francis, Handley, Karp, & Schenker (2001), Zhao et al. (2003)). Therefore, only a peer maintaining a key can be affected by a reconfiguration. However, each peer maintaining a data block has to periodically check the state of all the peers possessing a replica. The number of peers to check can be huge, since copies are randomly spread on the overlay.

This work investigates the effect of Churn and evaluates the performance of Lookup Mechanism in Pastry, Chord, and Kadmelia. It also provides a variant of the *Leafset* replication approach in which we avoid data block migrations when the desired number of replicas is still available in the DHT. We release the logically closest placement constraint on block copies and allow a peer to be inserted in the *Leafset* without forcing migration. Then, to reliably locate the block copies, the root peer of a block maintains replicated localized metadata. Metadata management is integrated to the existing *Leafset* management protocol and does not incur additional overhead in practice. This *Leafset* replication strategy is usefull to tolerate a high Churn rate and can adapt itself to offer best performance across a wide range of operating conditions with bounded bandwidth consumption.

1.3 Methodology

The research methods used in this work are those that cover the techniques essential for P2P overlay network simulations. We try to keep the level of details reasonable without compromising the intelligibility of the study. To investigate the effects of Churn and evaluate the performance of replication strategy and lookup mechanism in Pastry, Chord, and Kademlia, we have performed simulations using OverSim ((Team 2011)). OverSim is an overlay simulator that is integrated with the OMNeT++ network simulator ((Omnet++ 2011*b*)), and the INET framework ((Omnet++ 2011*a*)). The INET framework is an open-source communication networks simulation package, written for the OMNEST/OMNeT++ simulation system. The INET framework contains models for several Internet protocols.

The actual research is carried out by simulating our solution with the OverSim overlay network simulator. Our approach called Free_DHT requires us to program a model with the desired features. Using Free_DHT simulator we prepare and evaluate various simulation scenarios against PAST. We simulate three DHT algorithms Pastry, Chord and Kademlia in both iterative and recursive routing mode. The DHT algorithms are tested as they are configured by default in OverSim and they are not tuned or developed in any way.

1.4 Organization of the Dissertation

The Dissertation is organized as follows. In Chapter 2 we first present a survey of P2P networks, together with their classification and characteristics, while giving examples of the most relevant existing overlays to our research. We then describe distributed hast table caractecteristics. In Chapter 3 we proceed with elaborating on our solutions for designing efficient DHT which we name Free_DHT. In Chapter 4 we present the simulation environment, and the implentation Free_DHT which release placement constraint. In Chapter 5 we conduct our performance evaluation, and then present the simulation results. We present our conclusion in the last chapter.

Chapter 2

Overview of some P2P networks

In this chapter we present the classification of Peer-to-Peer distributed systems and the replication techniques that are used for implementing the DHT layer. A peer-to-peer system is a distributed system where peers (computers) communicate in a decentralized way. Peer-to-peer systems are mostly used for information transmission, such as file sharing, content distribution, storage, video or audio transmission.

These systems can be categorized into structured and unstructured networks. They are built on top of a network (or another system) that is used as support for communication. Peer-to-peer systems appeared from the need to decentralize. There is no central entity to act as the application server, or to coordinate or perform management tasks. The advantages are the removal of the single point of failure (i.e., the system continues to perform even if a resource fails, using other resources instead), and the distribution of the tasks between the peers. Each peer acts both as a server and a client, serving and being served by the system, respectively. To enable communication, peers collaborate in a self-organizing manner.

See Figure 2.1 on page 5, a peer-to-peer system connects participating peers in an overlay. Peers are connected through logical links (or simply links) that represent routing paths in the underlying network. The most used underlay network is the Internet or IP networks in general. In order to communicate, each peer knows a subset of the participating peers, which are called neighbors.

There are three communication operations that are application independent and that any peer needs to know how to perform;

- Receive a message from another peer;
- Send a message to another peer;
- Forward a message.

Besides the above, there are specific application operations, such as:

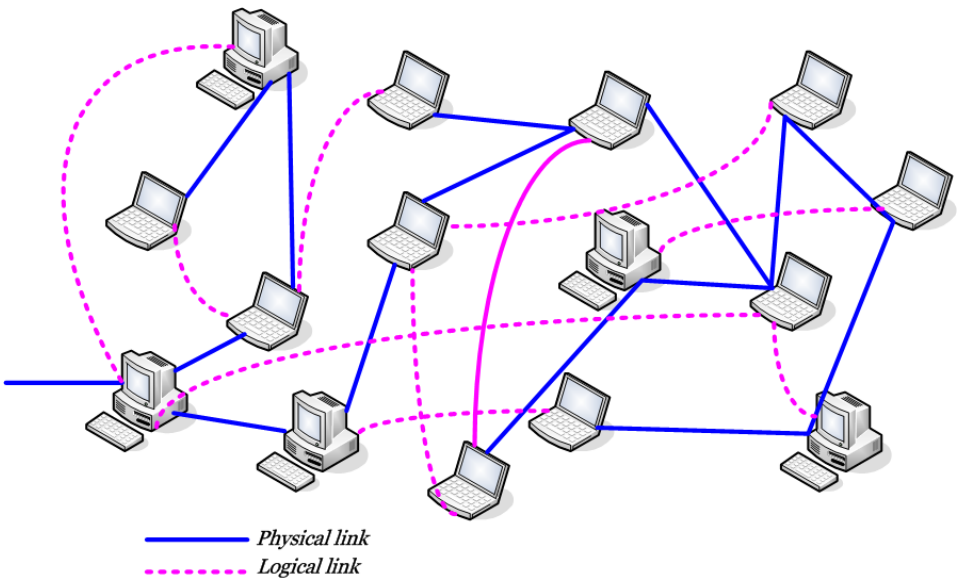

Figure 2.1: Overaly Network

- Finding the right neighbor to send or forward a message to;
- Initiating some requests in the overlay;
- Replying to incoming requests.

The process of becoming part of an overlay is called "joining". Depending on the overlay, in order to connect to other participating peers, a new peer p may receive a list of other peers (called bootstrap peers) or it has to find some peers by itself. The peer p thus becomes part of the overlay by connecting to a set of peers through logical links. However, this is not enough. Peer p needs to be known also by the other peers. For this reason, some overlays require additional messages, while others rely on the communication in the overlay in order to gradually create these links.

The life time of a participating peer ends with its departure. A peer can either fail (i.e., silently leave) or it can announce its departure. Different methods are applied in each case and some overlays put in place mechanisms for fault tolerance in order to support these departures. In the case of failures, the overlay has to reorganize (i.e., recreate links) over time. A neighbor is detected as failed when it fails to receive a message during periodical checks. Then, it is not considered a neighbor anymore. For overlays that need a specific number of neighbors, other suitable peers need to be found. The rate of the arrivals and departures is referred to as "Churn". Often, the performace of a newly proposed overlay is analyzed and evaluated under Churn.

The creation and maintenance of the logical links determine the class of the peer-to-peer system. A common nomenclature differentiates between unstructured and structured peer-to-peer systems. A system without any constraints on its logical links, having usually arbitrary links between the peers, is commonly called unstructured system. Gnutella (Gnutella (2005)) and Freenet (Freenet (2005)) are examples of unstructured systems. The management of this kind of systems is rather basic. A peer keeps some links towards some other peers and requests are sent without any sense of direction using partial or complete flooding (i.e., sent to a subset or to all neighbors). This method not only charges the system with a high traffic, it also does not assure that any message will reach its destination.

In contrast, a structured system assures a high reachability of the destination peer and does not generate much traffic. However, their management is more complex. Peers have identifiers and they maintain rather strict connections between them: specific constraints common for all peers are imposed on the neighbors, e.g., neighbors having to match certain patterns for their identifiers. Additionally, they define specific rules for message transmission, as we will discuss in the following paragraph.

The mostly spread branch of structured peer-to-peer systems are unquestionably the Distributed Hash Tables (DHTs). A DHT is a system where objects are assigned to different peers, based on a hash function, as follows. In addition to peers having identifiers, each object (information) also has an identifier, which is obtained by applying a hash function on some attribute of the object, such as its name.

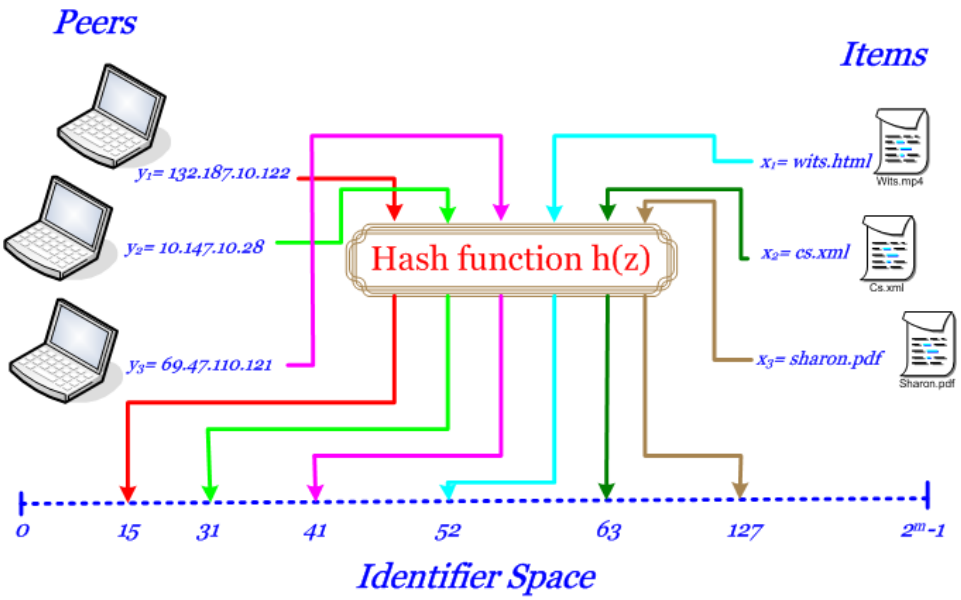

Figure 2.2: Hashing peers and items into the identifier space

The objects are distributed on the peers in the system according to their identifiers, e.g., an object is assigned to the peer with the closest identifier to the object's identifier. The main purpose of DHTs is object lookup: a peer issues a request for an object and the request is forwarded in the system until the destination peer is found (or some failure occurs), The forwarding process gives a sense of direction: each peer in the path chooses the following peer based on its neighbors identifiers and the object identifier. When the destination peer is found, it can directly communicate with the inquiring peer.

In decentralized systems, all peer-to-peer systems rely on communication in order to accomplish their tasks, either for maintenance or for the end-user inquiries. The ability of the overlay to act in a distributed, and a decentralized manner by having all peers collaborate to produce emergent behavior, is called "self-organization".

In the following sections of this chapter, we first present DHTs, describing its most well-known systems such as Chord, Pastry and Kademlia.

2.1 Distributed Hash Tables (DHTs)

A DHT is a peer-to-peer system that uses a hash function to distribute objects (information) over the peers. DHTs are mostly used for information lookup: a peer inquires the system for some information to which other peer(s) from the system replies. If not found in the original target node, the request travels in the system from peer to peer. In the past decade, several DHTs have been proposed. Basically, these DHT approaches differ in the hash space they consider. The rules for associating objects to peers and the routing strategies are discussed in the rest of this section.

The hash space (henceforth called identifier space), can be represented as a unidimensional or multidimensional space (2D, 3D, etc.). It may be a ring, Euclidean space, hypercube, or any other type of graph. In this space, peers are less formally called nodes, while objects are referred to as keys. The objects are assigned to peers based on peer and object identifiers. Usually, a key is mapped to the closest or to the following node in the identifier space (according to a predefined order), but other methods can be employed as well. The node identifiers and their logical links represent the structure of the overlay, thus the identifier space has to be chosen accordingly.

Since DHTs deal with node failures, we use the following terminology. A node is live (or alive) if it actively participates in the lookup protocol, i.e., it can reply and forward requests. Conversely, a node is dead (or, it failed) if it cannot be contacted anymore and, as a consequence, it cannot be used to forward requests. Each peer may issues a request for an object.

The process of forwarding the request from peer to peer, from source until destination, is called "routing". Each peer keeps its neighbors in a routing table,

which has a fixed number of entries. Thus, when a peer fails, it has to be replaced by a new peer. Usually, there are strict rules for the peers at specific entries (as we will see in the DHT examples the following subsections), thus, finding another suitable peer, when it exists, requires additional messages. These operations represent the maintenance costs of the routing tables. The choice of the neighbor to forward a request to is given by the routing strategy. DHTs have various routing strategies that depend on the overlay structure.

The DHT structures lies in the node degree, i.e., the number of neighbors with which a node maintains continuous contact for supporting the routing mechanism. A constant node degree, which is usually a small number, assures low maintenance costs for operations such as maintaining routing tables or exchanging control information (e.g. state of neighbors). Other DHTs use a logarithmic node degree. Examples of such structures include Chord (Stoica et al. (2003)), Pastry (Rowstron & Druschel (2001*a*)), Tapestry (Zhao et al. (2004)) or Kademlia (Maymounkov & Mazieres (2002)).

While these systems induce high costs for maintaining the multiple entries in the routing tables, they allow for defining routing strategies that exploit alternative paths, using alternative routing table entries for routing a request. For instance, such paths can be used in case of a node failure, as in (Serbu et al. (2007)). Alternative paths have not only the advantage of providing better fault tolerance, but they also offer support for multiple routing strategies.

2.1.1 Iterative routing

In iterative routing, an intermediate node receiving a query message, returns a response message that includes the information about the next hop node. The responsibility of actually contacting the next hop remains with the initiating node. In iterative routing, nodes are arranged in a ring shape. The initiating node monitors the routing process and decides how long it wants to wait for a response message before it considers the queried node absent. For a lookup path of N nodes $(2N-1)$ messages are needed for reaching the target node (Kunzmann (2005)).

Recursive routing: In the recursive routing, the intermediate nodes simply forward the query to the next hop node without informing the initiating node. This means that the initiating node has no control over the routing after it has sent the first query message. There are different versions of recursive routing, in which the routing of the response message from the destination node to the initiating node varies. These include symmetric recursive, forward-only and direct response. For a lookup path of N nodes recursive routing needs N messages to reach the target node (Kunzmann (2005)).

Symmetric recursive: In symmetric recursive routing the response message visits the same nodes as the request message did only in a reversed order. This

requires the nodes to remember from which node they received the request message. A via list storing information about the visited nodes can be included in the query message. This allows the response message to follow the same path in reverse.

Forward-only: If the forward-only recursive routing is used, the response message is routed as a new message initiated by the recipient of the query. The route of the response message is thereby independent from the route of the query message.

Direct response: The response message can also be sent directly to the initiating node. The initiator needs to encode its contact address in the query message so that the recipient node knows where to send the response. This is the optimal routing technique but it can rarely be used due to connectivity issues like NATs.

For DHTs, we present Chord (Stoica et al. (2003)) and Pastry (Rowstron & Druschel (2001*a*)) as typical examples of systems with a ring structure that employ greedy-routing (more specifically prefix-routing for Pastry), and Kademlia (Maymounkov & Mazieres (2002)) as typical example of overlay with some new features to improve efficiecy of file lookup.

2.1.2 Chord Protocol

In Chord (Stoica et al. (2003)), each node and each key has a m-bit identifier on a 2^{m-1} identifier space designed as a ring, that is derived by respectively hashing the IP address and the name. Each key is mapped to the first node (starting at the key identifier) that follows clockwise on the ring. For routing purposes, each node has a routing table with m entries, each entry i pointing towards the first node on the ring at a distance of at least 2^i, where $i = 0 \ldots \ m-1$. The node at entry i is also called "*finger* i". The corresponding links are referred to as incoming links on the node they point to.

A graphical representation of a Chord ring is shown in Figure 2.3, as an example of the identifier space of $2^{m=6} = 64$ addresses with 15 nodes. Figure 2.3 (a) shows the outgoing and incoming links of node 22, with solid and dashed lines, respectively. *fingers* do not point to nodes at a distance exactly equal to a power of 2. The same applies also to the incoming links: they do not come always from distances equal to a power of 2. For example, node 5 comes from a distance of $2^4 + 1$. Chord uses greedy routing, a routing strategy that sends the requests as close as possible to the destination. The average path length is in the order of O(log N). The path is clockwise on the ring, as in the example from Figure 2.3 (b): $61 \rightarrow 15 \rightarrow 19 \rightarrow 22$.

In order to assure connectivity and to facilitate the join and departure mechanisms, each node also keeps track of its predecessor. Note that any node knows

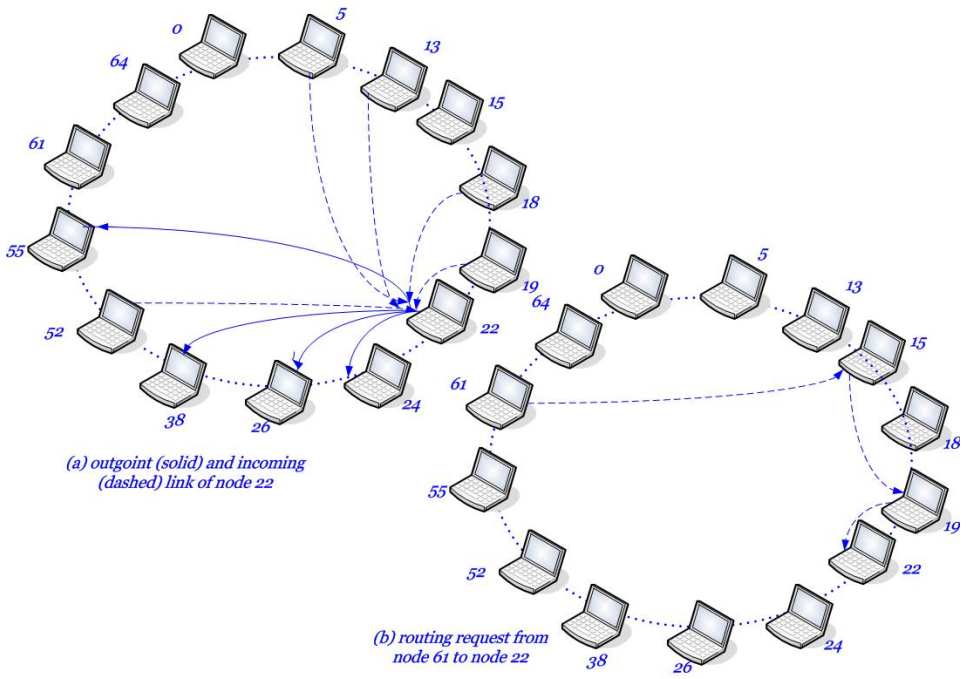

Figure 2.3: Example of Chord with an identifier space

its successor, which is its first *finger*. When a new node n_j joins the system, links are created/updated and the responsibility of the keys are reconsidered, as follows. Node n_j creates links towards its predecessor and successor and requests them to consider n_j as their new successor and predecessor, respectively. Node n_j builds its own routing table and the other nodes from the system update their routing tables in order to reflect its arrival. With the links being created, the node n_j obtains from its successor the objects for which it is now responsible for.

A stabilisation mechanism is used for the nodes that fail or leave voluntarily: periodically, each node n_s checks its successor, whether it is still alive, it is the right node and if it has n_s as its predecessor. Moreover, and also periodically, each node n_s refreshes its routing table by finding the right nodes for each entry. A node that leaves voluntarily gives the responsibility of its objects to its successor in order to preserve the rule of key mapping to nodes.

To deal with failures, each node maintains a successor-list of r nodes (i.e., its r nearest successors on the ring). Whenever a request needs to be sent to a *finger* that is not reachable anymore, a lower *finger* is used instead, and if necessary, the nodes in the successor-list can also be used as alternatives. Objects can also be replicated at several successors. Chord has thus the advantage of simplicity and has been proved (Stoica et al. (2003)) to scale well with the number of nodes and to recover from high rates of Churn.

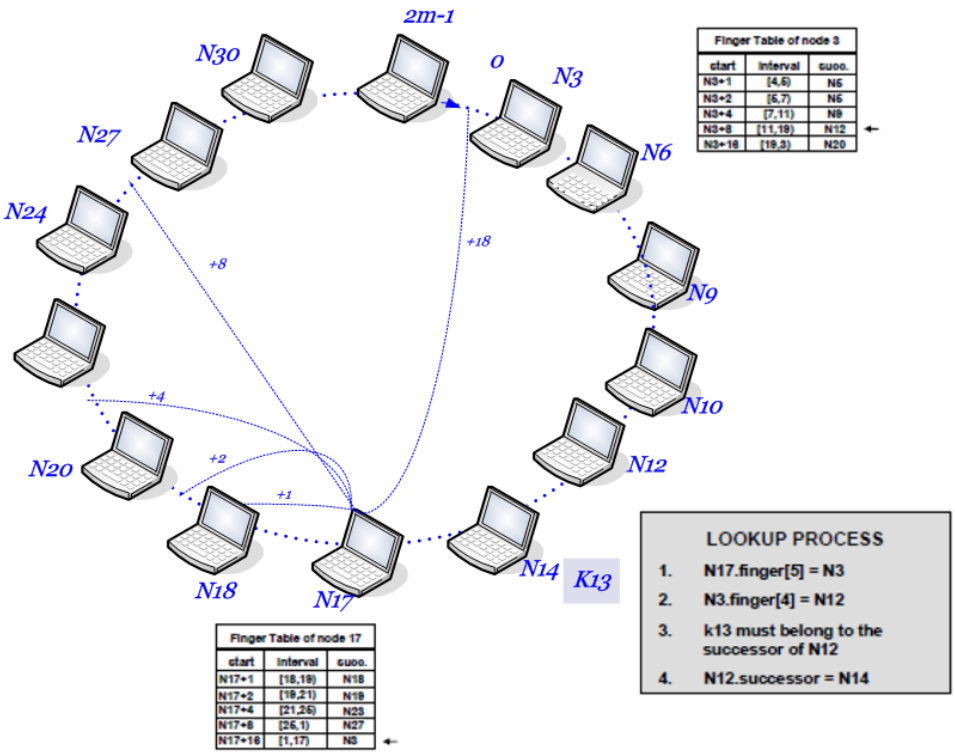

Figure 2.4: Chord's Lookup process, Node 17 looking for Key 13

Lookup process

In the beginning of a key lookup process the initiating node checks if it has the node responsible for the requested key in its successor list. If there is no match in the node's successor list, it uses its *finger* table to find a *finger*[i] which most immediately precedes the requested key. The initiating node then contacts this node and the contacted node repeats the same lookup procedure. The lookup process continues until the node that immediately precedes the desired identifier is found. This node then returns its successor as the lookup value. The lookup process is illustrated in Figure 2.4. Depending on the routing style the contacted node either gives the initiating node the address of the next node to contact (iterative style) or it contacts the next node itself (recursive style). These two routing modes have different constraints about the connections between the network nodes. Which style is preferred, depends on the type of network Chord is used in. Figure 2.4 shows an illustration of node with ID 17 performing a lookup for a key with ID 13. Node 17 looks for the closest node to ID 13 from its *finger* table and notices that 13 belongs to 5th *finger* interval [1, 17). The corresponding entry in its *finger* table is node 3 which is the first node in this interval. Node 17 will then ask node 3 to find the successor of ID 13. Node 3 then checks its *finger* table and finds out that ID 13 belongs to the 4th interval [11, 19) where the first node is 12. Finally node 12 knows that ID 13 must belong to its successor node. Node 12 now replies that the node responsible for ID 13 is node 14.

Performance

Chord does not waste nodes' capacity but uses it effectively. For efficient routing, each node needs to maintain information for only $O(logN)$ other nodes. The lookups are also resolved in an efficient manner. Each node can carry out a lookup to any given node via $O(logN)$ messages to other nodes (Stoica et al. (2003)).

2.1.3 Pastry Protocol

Like Chord, Pastry (Rowstron & Druschel (2001*a*)) has also a ring structure, where a node identifier is the result of hashing its IP address or its public key, and the key of an object is determined by hashing its name. In Pastry, each node and key has an identifier with a sequence of digits in base 2^b (where b has a typical value of 4) that determines its position on the ring. Each node is responsible for the closest keys, towards its predecessor or successor on the ring.

Pastry routes requests to the node that is numerically closest to the destination key. Routing takes less than $\log_{2^b} N$ steps, in a network of N nodes. At each step, the request is sent to a neighbor that shares with the key, a prefix that is at least one digit longer than the common prefix of the key with the local node. If no

such neighbor exists, the request is sent to a neighbor that shares exactly the same prefix as the local node but numerically closer to the key. In order to support this routing procedure, each Pastry node maintains a routing table and a *Leafset*. When receiving a request for an object O, the request is forwarded to the node from the routing table or from the *Leafset* whose identifier has the longest common prefix with O. The routing table is composed of $\log_{2^b} N$ rows with $2^b - 1$ entries each. The i^{th} entry in the routing table of node n_j maps to $2^b - 1$ nodes that share a common prefix of exactly i digits with node n_j. If no node is found suitable for an entry, that entry is left empty. However, this is unlikely to happen due to the uniform distribution of the nodes in the identifier space. The *Leafset* (denoted L) contains the numerically closest $|L|$ neighbors on the ring: the numerically closest $|L|/2$ nodes among its successors and the numerically closest $|L|/2$ nodes among its predecessors.

Besides the routing table and the *Leafset*, a Pastry node also has a neighborhood set M , which contains nodes that are closest to the local node, according to a proximity metric. The neighborhood set is mostly used to maintain locality properties and it is used for routing only when neither the routing table nor the *Leafset* contain a node that can forward the request. The metric is usually the number of IP routing hops or the geographic distance between two nodes. Typical values for $|L|$ and $|M|$ are 2^b or 2×2^b.

When joining the system, a node creates its routing table as follows. A newly arrived node X sends a request towards its own identifier through a known node A such as a node that is close according to the proximity metric. Node A is considered not to share any prefix with X, thus the first row in the routing table of A would perfectly fit as the first row in the routing table of X. All the nodes on the path send part of their routing tables to X: the first node B in the path (after A) shares a prefix of length 1 with X, and so B's first row is appropriate for X's first row, and so on with the other nodes on the path.

An example of routing table is shown in Table 2.1, where we have:

1. L nodes in *Leafset*;
2. $(\log_{2^b} N)$ Rows, actually $(\log_{2^b} 2^{128} = 128/b)$;
3. 2^b columns;
4. L network neighbors.

In a Pastry system where all nodes have an identifier as a sequence of digits in base b. For $b = 2$, $2^{b=2} = 4$ and the figure shows the *Leafset*, routing table and neighborhood set of node 10233102. The *Leafset* contains $|L| = 8$ nodes, half with smaller and the other half with larger identifiers than 10233102. The routing table has 8 rows, starting with row 0. Each row i contains 3 nodes, each with a different digit after the common prefix of length i with 10233102. Row 0 contains nodes

NodeID 10233102

Leaf set

Smaller		*Larger*	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Table 2.1: Pastry Routing table

that do not share any prefix with 10233102, so their first digit is either 0, 2 or 3 (different from 1, which is the first digit of 10233102). The nodes at row 1 share the first digit with the local node so their first two digits are 11, 12 and 13, and so on with the rest of the rows. The shaded cells do not contain any node. They only show the corresponding digit of the local node. The neighborhood set contains $|M| = 8$ nodes that have been chosen according to a predefined proximity metric. The corresponding IP addresses are not shown. Figure 2.5 shows another example of Pastry overlay, for routing, where node N_4 sends a lookup request for key K_{24}. The Pastry parameters are $b = 1$, $|L| = 2$ and identifiers on 5 digits. From N_4 , the request is sent to node N_{28} (the node numerically closest to K_{24} in the routing table of N_4) and then forwarded to N_{24}. The figure also shows the *Leafset*s and routing tables of nodes N_4 and N_{28}. Note that some rows are empty in the routing tables, since there are no nodes in the system to fit them.

Also as an observation in a single hop, a request does not necessarily go to one digit closer to the destination. For example, from N_4, which did not share any prefix with K_{24}, the request is sent to N_{28}, which shares the first 2 digits with K_{24}. Pastry uses thus prefix routing and it has the particularity of allowing a large choice for the nodes in the routing table (especially in the first rows). We use Pastry in our research as a DHT system model to elaborate on routing tables that take into consideration releasing the placement constraint.

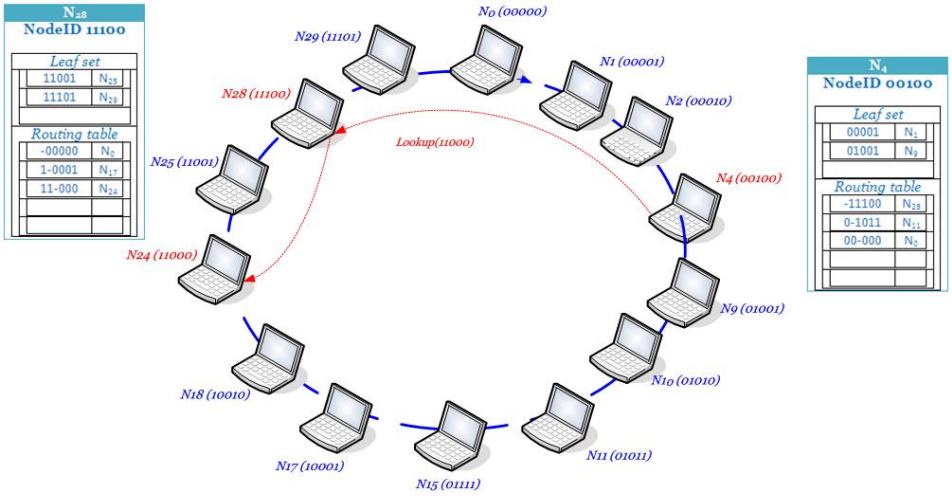

Figure 2.5: Pastry overlay structure and routing principle

2.1.4 Kademlia Protocol

Kademlia is a DHT algorithm that has already been used in various peer-to-peer applications. It has been utilized by a number of very popular applications like eMule and BitTorrent. Kadelmia takes advantage of every message it sends by using them to keep the routing tables up to date. (Pita & Riesco (2012)) This way the need for explicit update messages is minimized.

Node identifiers and keys have 160 bits. Kademlia uses exclusive or (XOR)-metric to define logical distances between two nodes in the network. The distance between identifiers a and b is an integer interpretation of their bitwise XOR. Nodes are considered as leaves of a binary tree. The XOR topology is symmetric unlike the ring topology used in Chord. The symmetry feature means that the distance from node x to node y equals the distance from y to x. Table 2.2 and Table 2.3 depicts the XOR metric.

Nodes are treated as leaves of a binary tree where each node gets its location according to the shortest unique prefix of the node's identifier. Every node sees the network as a group of subtrees that the node itself doesn't belong to. The highest subtree consists of the other half of the binary tree not containing the node itself.

The next subtree includes the half of the remaining binary tree that is not part of the highest subtree. The subtree model is illustrated in Figure 2.6.

XOR	0	1
0	0	1
1	1	0

Table 2.2: XOR table

	00110101
XOR	11100011
	11010110

Table 2.3: XOR metric

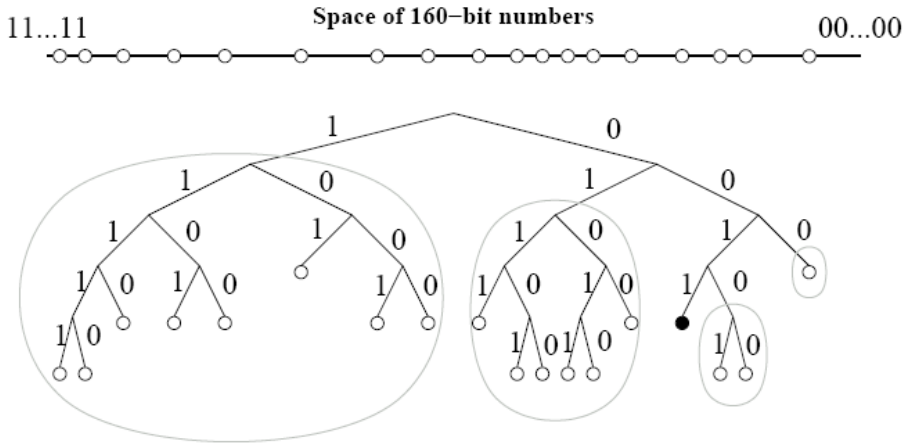

Figure 2.6: Kademlia's Subtree model

K-bucket concept

Every node stores contact information about other nodes in lists called k-buckets. Each node keeps a k-bucket for nodes that lie from a distance between $2i$ and $2i+1$, and itself. As $0 \leq i < 160$ this results in 160 k-buckets for every node. K-bucket entries are sorted by the time that has passed since the entries have been seen. The least recently seen node is the first node on the list and the most recently seen node is the last. Parameter k stands for the size of k-buckets. Each k-bucket can store at most k entries.

Routing table

Kademlia nodes use their k-buckets to form routing tables. Entries in each k-bucket have a common prefix of their IDs. The k-buckets are arranged in a binary tree and their positions are determined by the prefix. This binary tree with k-buckets as its leaves is the node's routing table. The routing table covers the whole ID range. As all k-buckets have equal size, each node knows more about the ID ranges near its own ID than those that are further away.

Routing table updating

Kademlia minimizes the need of updated messages by including pieces of information about the overlay in every lookup message sent. As a node receives any message it updates the appropriate k-bucket. If the k-bucket already contains the sender's node ID, the sending node is moved at the end of the list. If the sending node doesn't exist in the appropriate k-bucket and if the k-bucket is not full, the sending node is inserted at the tail of the list. In the case that the k-bucket is already full, the receiving node pings the least recently seen node in the k-bucket to know whether that node is still alive. If the ping is responded, the least recently seen node is moved at the end of the list and the new node is not added to the k-bucket. To keep the k-buckets stable and resilient against lost messages, Kademlia doesn't evict a node from a k-bucket until it has failed to respond to five consecutive messages. An optimization has also been made to the process of pinging of the least recently seen node depicted above. The original algorithm results in a large number of ping messages and the optimized algorithm delays contacting the least recently seen node until there is a useful message it can send to this node. Kademlia still keeps a timer for updating the k-buckets. If there are no lookups for a particular ID range, the corresponding bucket could end up out of date. To prevent this from happening, each node performs a refresh procedure to every bucket that hasn't been a target of a lookup in the past hour. This is done by performing a node search for a random ID in the k-bucket's ID range.

Lookup process

Kademlia's has a node lookup procedure in which the parameter determines the number of parallel requests sent. The initiator picks closest nodes to the requested node ID from its k-buckets and sends the lookup requests to these nodes. The recipients then return k closest nodes to the requested ID they know of (a single k-bucket if the bucket is full). As the initiator gets the responses, it picks the next nodes that will be the new recipients of a request. This process goes on until the initiator has queried all the k closest nodes it has learned during the lookup process and also gotten a response from all of them. The lookup process is illustrated in Figure 2.7 on page 18.

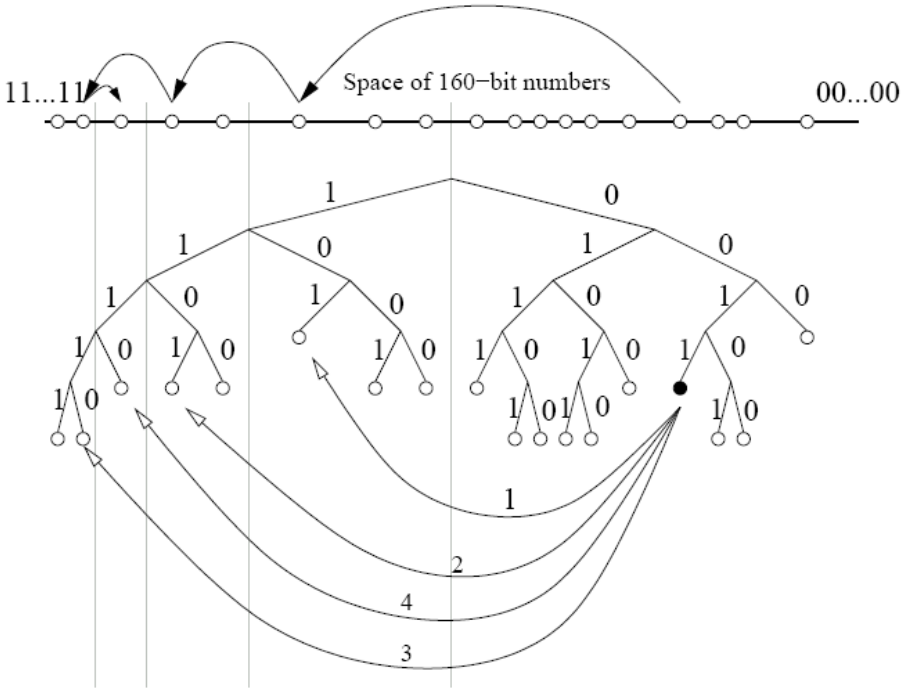

Figure 2.7: Kademlia's Lookup process

Performance

Each Kademlia node keeps information about $(B.Log_B N + B)$ other nodes in its routing table. Here the letter B stands for the number of different node IDs in the network. Kademlia uses 160 bit node IDs which gives B a value of 2^{160} (Maymounkov & Mazieres (2002). Node lookups are performed in $(O(log_B N) + c)$ hops where c indicates a small constant.

Summarized in Table 2.4 are the terminologies we use in naming various nodes and messages in Chord, Pastry and Kademlia.

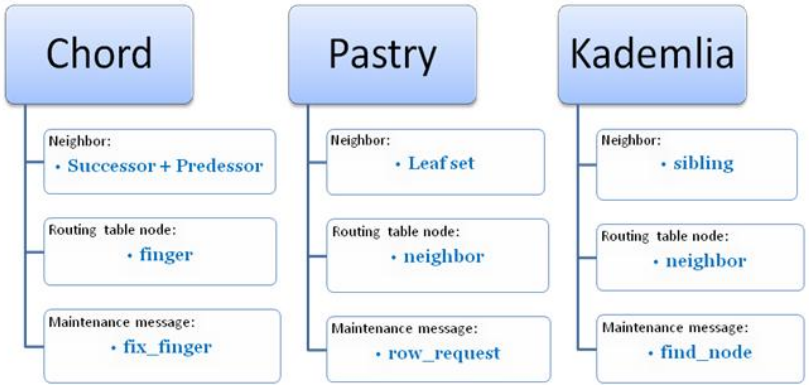

Table 2.4: Description of different nodes

2.1.5 Replication techniques for implementing the DHT layer

DHT based P2P systems are usually structured in three layers:

1. Routing layer;
2. The DHT itself;
3. The application that uses the DHT.

The routing layer is based on keys for identifying peers and is, therefore commonly qualified as Key-Based Routing (KBR). Such KBR layer hides the complexity of scalable routing, fault tolerance, and self-organizing overlays to the upper layers. In recent years, many research efforts have been made to improve the resilience of the KBR layer to a high Churn rate (Rhea, Geels, Roscoe, & Kubiatowicz (2004)). The main examples of KBR layers are: Pastry (Rowstron & Druschel (2001*a*)), Chord (Stoica et al. (2003)), Tapestry (Zhao et al. (2004)), and Kademlia (Maymounkov & Mazieres (2002)).

The DHT layer is responsible for storing data blocks. It implements a distributed storage service that provides persistence and fault tolerance, and can scale up to a large number of peers. DHTs provide simple get and put abstractions, that greatly simplifies the task of building large-scale distributed applications. PAST (Rowstron & Druschel (2001*b*)) and DHash (Dabek et al. (2004)) are DHTs

respectively built on top of Pastry (Rowstron & Druschel (2001*a*)) and Chord (Stoica et al. (2003)).

Finally, the application layer is a composition of any distributed application that may take advantage of a DHT. Representative examples are:

- the Cooperative File System (CFS) (Dabek et al. (2001)), which implements a read-only distributed system,
- the PeerStore backup system (Landers et al. (2004)), and
- PAST (Druschel & Rowstron (2001)).

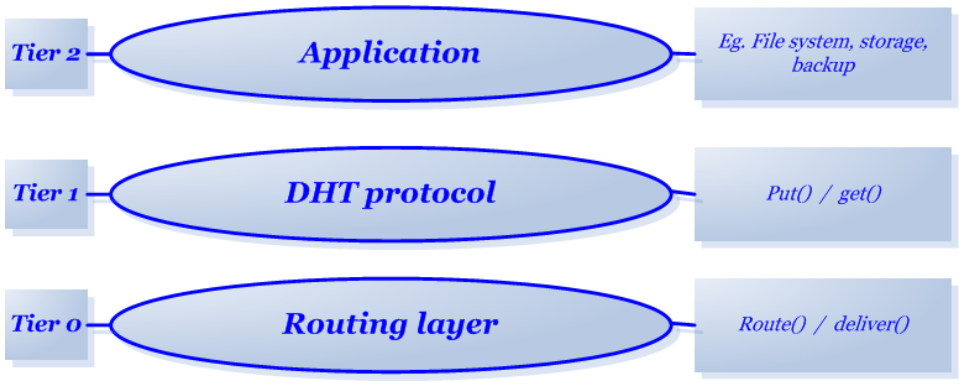

Figure 2.8: Structure of a DHT based system

2.2 PAST

PAST (Druschel & Rowstron (2001) is a persistent peer-to-peer storage utility, which replicates complete files on multiple nodes. The PAST system is composed of nodes connected to the Internet, where each node is capable of initiating and routing client requests to insert or retrieve files. Optionally, nodes may also contribute storage to the system. The PAST nodes form a self-organizing overlay network.

An efficient routing scheme called Pastry is used for message routing and content location. Pastry, ensures that client requests are reliably routed to the appropriate nodes. Inserted files are replicated across multiple nodes for availability. With high probability, the set of nodes over which a file is replicated is diverse in terms of geographic location, ownership, administration, network connectivity, rule of law, etc. Client requests to retrieve a file are routed, with high probability, to a node that is close in the network to the client that issued the request1, among the live nodes that store the requested file. The number of PAST nodes traversed, as well as

the number of messages exchanged while routing a client request, is logarithmic in the total number of PAST nodes in the system under normal operation. A storage utility like PAST is attractive for several reasons. First, it exploits the multitude and diversity (in geography, ownership, administration, jurisdiction, etc.) of nodes in the Internet to achieve strong persistence and high availability. This obviates the need for physical transport of storage media to protect backup and archival data; likewise, it obviates the need for explicit mirroring to ensure high availability and throughput for shared data. A global storage utility also facilitates the sharing of storage and bandwidth, thus permitting a group of nodes to jointly store or publish content that would exceed the capacity or bandwidth of any individual node (Ballani et al. (2011)).

A 128-bit node identifier is assigned to each PAST node, called a *idNode*. PAST stores a content item on the node whose node identifier, *idNode*, is closest to a quasi-unique file identifier *idFile* which is generated at the time of the files insertion into PAST. Files can be shared at the owners discretion by distributing the *idFile* (potentially anonymously) and, if necessary, a decryption key. To retrieve a file in PAST, a client must know its *idFile* and, if necessary, its decryption key. PAST does not provide facilities for searching, directory lookup, or key distribution.

Routing a message to the closest node is done by choosing the next hop node whose *idNode* shares with the *idFile* a prefix that is at least one digit longer than the prefix that the *idFile* shares with the present node's *idNode*. The *idFile* is generated by hashing the filename and the *idNode* is assigned randomly when a node joins the network. The routing path length scales logarithmically in terms of the overall number of nodes in the network.

For each file an individual replication factor k can be chosen and replicas are stored on the k nodes that are closest to the *idFile*. Maintaining the k replicas in the case of a node failure is detected by the Pastry background process of exchanging messages with neighbors. When a node detects a neighbor node's failure, the replica is automatically replaced on another neighbor. Free storage space is used to cache files along the routing path, while approaching the closest node during the publish or retrieval process. This can only be done if the file data is routed along the reverse query path. Thus there is no direct peer-to-peer file transfer. Similar to the Freenet (Freenet (2005)) system, nodes with low bandwidth may become a bottleneck.

2.2.1 PAST Architecture

The collection of PAST nodes forms an overlay network in the Internet. Minimally, a PAST node acts as an access point for a user. Optionally, a PAST node may also contribute storage to PAST and participate in the routing of requests within the PAST network. Any host connected to the Internet can act as a PAST node by installing the appropriate software.

The PAST system exports the following set of operations to its clients:

- *idFile* = Insert(name, owner-credentials, k, file) stores a file at a user-specified number k of diverse nodes within the PAST network. The operation produces a 160-bit identifier (*idFile*) that can be used subsequently to identify the file. The *idFile* is computed as the secure hash (*SHA-1*) of the files name, the owners public key, and a randomly chosen salt. This choice ensures (with very high probability) that *idFiles* are unique. Rare *idFile* collisions are detected and lead to the rejection of the later inserted file.

- file = Lookup(*idFile*) reliably retrieves a copy of the file identified by *idFile* if it exists in PAST and if one of the k nodes that store the file is reachable via the Internet. The file is normally retrieved from a live node near the PAST node issuing the lookup (in terms of the proximity metric), among the nodes that store the file.

- Reclaim(*idFile*, owner-credentials) reclaims the storage occupied by the k copies of the file identified by *idFile*. Once the operation completes, PAST no longer guarantees that a lookup operation will produce the file. Unlike a delete operation, reclaim does not guarantee that the file is no longer available after it was reclaimed. These weaker semantics avoid complex agreement protocols among the nodes storing the file.

A 128-bit node identifier is assigned to each PAST node. The *idNode* indicates a nodes position in a circular namespace, which ranges from 0 to 2^{1281}. The *idNode* assignment is quasi-random (e.g., *SHA-1* hash of the nodes public key) and cannot be biased by a malicious node operator. This process ensures that there is no correlation between the value of the *idNode* and the nodes geographic location, network connectivity, ownership, or jurisdiction. It follows then that a set of nodes with adjacent *idNodes* are highly likely to be diverse in all these aspects. Such a set is therefore an excellent candidate for storing the replicas of a file, as the nodes in the set are unlikely to conspire or be subject to correlated failures or threats. During an insert operation, PAST stores the file on the k PAST nodes whose *idNodes* are numerically closest to the 128 most significant bits (msb) of the files *idFile*. This invariant is maintained over the lifetime of a file, despite the arrival, failure and recovery of PAST nodes. For the reasons outlined above, with high probability, the k replicas are stored on a diverse set of PAST nodes. Another invariant is that both the set of existing *idNode* values as well as the set of existing *idFile* values are uniformly distributed in their respective domains. This property follows from the quasi-random assignment of *idNode*s and *idFiles*; it ensures that the number of files stored by each PAST node is roughly balanced. This fact provides only an initial approximation to balancing the storage utilization among the PAST nodes. Since files differ in size and PAST nodes differ in the amount of storage they provide, additional, explicit means of load balancing are required.

The number k is chosen to meet the availability needs of a file, relative to the expected failure rates of individual nodes (Yang et al. (2010)). However, popular files may need to be maintained at many more nodes in order to meet and balance the query load for the file and to minimize latency and network traffic. PAST adapts to query load by caching additional copies of files in the unused portions of PAST nodes local disks. Unlike the k primary replicas of a file, such cached copies may be discarded by a node at any time.

2.3 Replication in DHTs

In a DHT, each peer and each data block is assigned an identifier (i.e., a key). A data block's key is usually the result of a hash function performed on the block. The peer whose identifier is the closest to the block's key is called the block's root. All the identifiers are arranged in a logical structure as used in Chord (Stoica et al. (2003)) and Pastry (Rowstron & Druschel (2001*a*)) or a d-dimensional torus as implemented in CAN (Ratnasamy, Francis, Handley, Karp, & Schenker (2001)) and Tapestry (Zhao et al. (2003)).

A peer possesses a restricted local knowledge of the P2P network, i.e., the *Leafset*, which amounts to a list of its neighbors in the ring. For instance, in Pastry the *Leafset* contains the addresses of the $L/2$ closest neighbors in the clockwise direction of the ring, and the $L/2$ closest neighbors counter-clockwise.

Each peer monitors its *Leafset*, removing peers which have disconnected from the overlay and adding new neighbor peers as they join the ring. In order to tolerate failures, each data block is replicated on k peers which compose the Replica-set of a data block. Two protocols are in charge of the replica management, the initial placement protocol and the maintenance protocol. We now describe existing solutions for implementing these two protocols.

2.4 Replica placement protocols

There are two main basic replica placement strategies:

- *Leafset-based* replication;
- Multiple key replication.

2.4.1 *Leafset-based* replication

The data block's root is responsible for storing one copy of the block. The block is also replicated on the root's closest neighbors in a subset of the *Leafset*. The

neighbors storing a copy of the data block may be either successors of the root in the ring, predecessors or both. Therefore, the different copies of a block are stored contiguously in the ring. This strategy has been implemented in PAST (Rowstron & Druschel (2001*b*)) and DHash (Dabek et al. (2004)). Successor replication is a variant of *Leafset* based replication where replica peers are only the immediate successors of the root peer instead of being the closest peers (Ktari et al. (2007)).

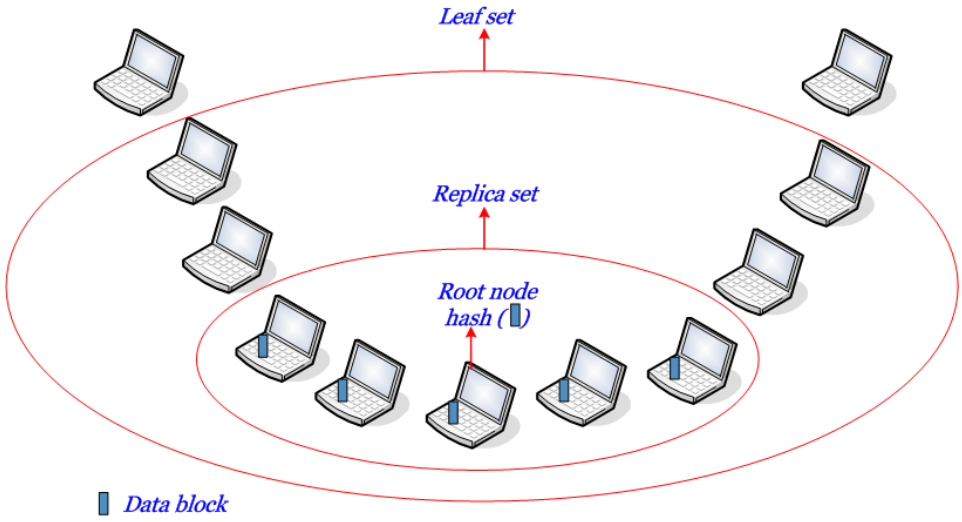

Figure 2.9: Leaf set based contiguous replication

2.4.2 Multiple key replication

This approach relies on computing k different storage keys corresponding to different root peers for each data block. Data blocks are then replicated on the k root peers. This solution has been implemented by Content Addressable Network (CAN) (Ratnasamy, Francis, Handley, Karp, & Schenker (2001)) and Tapestry (Zhao et al. (2003)). Google File System (GFS) (Ghemawat et al. (2003)) uses a variant based on random placement to improve data repair performance. Path and symmetric replication are variants of multiple key based replication (Ghodsi et al. (2005), Ktari et al. (2007)). Path replication stores data blocks along a routing path, using the path to attribute the keys, then each peer on the path is responsible for monitoring its successor (Ktari et al. (2007)). Symmetric replication is a particular kind of multiple key based replication (Ghodsi et al. (2005)) where an identifier of a block is statically associated with $f - 1$ other identifiers. Harvesf and Blough (Harvesf & Blough (2006)) (Harvesf & Blough (2011)) propose a random placement scheme focusing on producing disjoint routes for each replica set.

Lian et al. (Lian et al. (2005)) propose an hybrid stripe replication scheme where small objects are grouped in blocks and then randomly placed. They show

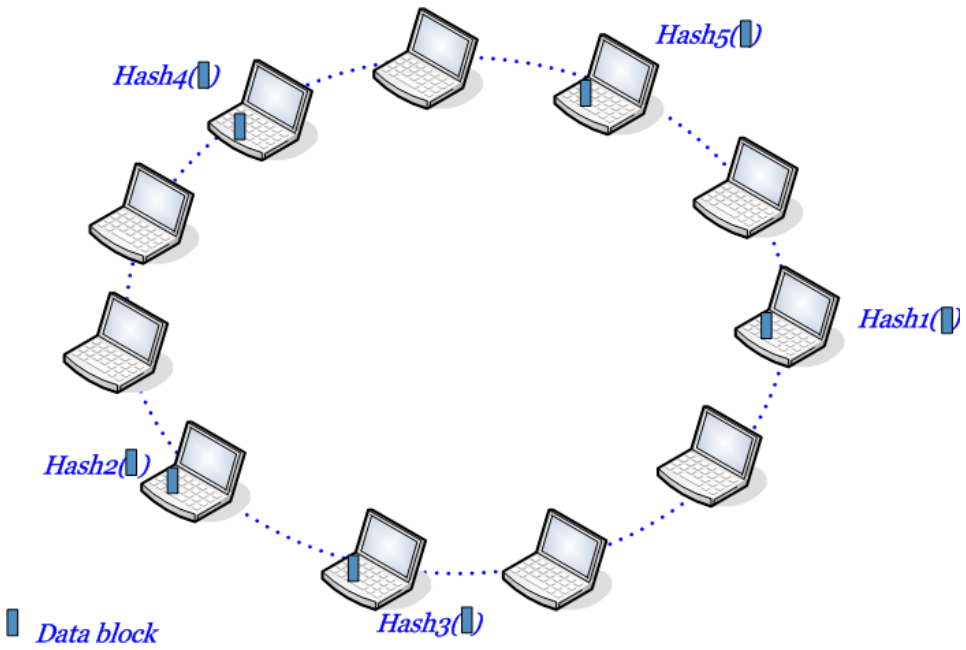

Figure 2.10: Multiple key based replication

using an analytical framework that their scheme achieves near-optimal reliability. Finally, several works have focused on the placement strategies based on availability of nodes. Van Renesse (van Renesse (2004)) proposes a replica placement algorithm on DHT by considering the reliability of nodes and placing copies on nodes until the desired availability was achieved. To this end, he proposes to track the reliability of each node such that each node knows the reliability information about each peer. In FARSITE (Adya et al. (2002)), dynamic placement strategies improve the availability of files. Files are swapped between servers according to the current availability of these latter. With theses approaches, the number of copies can be reduced. However, the cost to track reliability of nodes can be high. Furthermore, such approaches may lead to a high unbalanced distribution whereby highly available nodes contain most of the replicas and can become overloaded.

2.5 Maintenance protocols

The maintenance protocols have to maintain k copies of each data block without violating the initial placement strategy. This means that the k copies of each data block have to be stored on the root peer contiguous neighbors in the case of the *Leafset-based* replication scheme and on the root peers in the multiple key based replication scheme. The *Leafset-based* maintenance mechanism is based on periodic information

exchanges within the *Leafsets*. For instance, in (Rowstron & Druschel (2001*b*)) the fully decentralized PAST maintenance protocol, each peer sends a bloom filter (For short, the sent bloom filter is a compact and approximative view of the list of blocks stored by a peer) of the blocks it stores to its *Leafset*. When a *Leafset* peer receives such a request, it uses the bloom filter to determine whether it stores one or more blocks that the requester should also store. It then answers with the list of the keys of such blocks. The requesting peer can then fetch the missing blocks listed in all the answers it receives. In the case of the multiple key replication strategies, the maintenance has to be done on a per data block basis. For each data block stored in the system, it is necessary to periodically check if the different root peers are still alive and are still storing a copy of the data block.

2.6 Impact of the Churn on the DHT performance

A high Churn rate induces a lot of changes in the P2P network, and the maintenance protocol must frequently adapt to the new structure by migrating data blocks. While some migrations are mandatory to restore k copies, some others are necessary only for enforcing placement invariants. A first example arises at the root peer level which may change if a new peer with a closer identifier joins the system. In this situation, the data block will be migrated on the new peer. A second example occurs in *Leafset* based replication, if a peer possesses an identifier that places it within a replica set. Therefore, data blocks have to be migrated by the DHT to enforce replicas to maintain the 'closest peers from the root' property. It should be noted that the larger the replica set, the higher the probability for a new peer to induce migrations. Kim and Park (Kim & Park (2006)) try to limit this problem by allowing data blocks to interleave in *Leafsets*. However, they have to maintain a global knowledge of the complete *Leafset*; each peer has to know the content of all the peers in its *Leafset*. Unfortunately, the maintenance algorithm is not described in detail and its real cost is unknown. In the case of the multiple key replication strategy, a new peer may be inserted between two replicas without requiring migrating data blocks, as long as the new peer identifier does not make it one of the data block roots. However, this replication method has the drawback that maintenance has to be done on a per-data block basis; therefore it does not scale up with the number of blocks managed by a peer. For backup and file systems that may store up to thousands of data blocks per peer, this is a severe limitation.

Chapter 3

Free_DHT

In this chapter we present our replication approach in which we avoid data block migrations when the desired number of replicas is still available in the Distributed Hash Table.

3.1 Placement Strategy of Free_DHT

Our intent is to design a DHT, we named as Free_DHT, that tolerates a high rate of CHURN without degradation of performance in this project. For this, we avoid to copy in data blocks when this is not mandatory for restoring a missing replica. We introduce a *Leafset* based replication that releases the placement constraints in the *Leafset*.

Our solution is built on top of a KBR layer such as Chord or Pastry. Our design decisions are to use replica localized meta-data and separate them from data block storage. We keep the notion of a root peer for each data block. However, the root peer no longer stores a copy of the blocks for which it is the root. It only maintains metadata describing the Replica-set and periodically sends messages to the Replica-set peers to ensure that they keep storing their copy. A new peer may join a *Leafset* without necessarily inducing data blocks migrations. Using localized medata allows a data block replica to be anywhere in the *Leafset*.

For two reasons we choose to restrain the localized of replicas within the root's *Leafset*.

1. To remain scalable, the number of messages of our protocol does not depend on the number of data blocks managed by a peer, but only on the *Leafset* size;

2. The routing layer already induces many exchanges within *Leafsets*, the local

view of the *Leafset* at the DHT-layer can be used as a failure detector.

We use the put(k,b) operation to store a data block in the system. Put(k,b) produces an "insert message" which is sent to the root peer. When the root receives the "insert message," it randomly chooses a Replica-set of k peers around the center of the *Leafset*. The probability that a chosen peer quickly becomes out of the *Leafset* due to the arrival of new peers is reduced that way. After that, a "Add Data message" is sent to the Replica-set peers by the root. This "Add Data message" contains the following elements:

1. The data block;
2. The the metadata which is the identity of the peers in the Replica-set;
3. The identity of the root.

In the system, each peer can be root for several data blocks and part of the Replica-set of other data blocks. For this reason, the peer stores two list:

- A list rootVector of data block identifiers with their associated Replica-set peer list for blocks for which it is the root;
- A list replicaVector of data blocks for which it is part of the Replica-set. It also contains: the identifier of the data block, the associated Replica-set peer-list, and the identity of the root peer.

Each stored data block is associated whith a lease counter set to a value L. At each KBR-layer maintenance, this counter is decremented. Then maintenance protocol, described below, is responsible to periodically reset this counter to L.

3.1.1 Maintenance protocol

The root is the peer that the closest identifier from the data block's one. This periodic protocol help to ensure the existence of a root peer for each data block. It also ensure that each data block is replicated on k peers located in the data block root's *Leafset*.

A peer p executes Algorithm-1 after each period T, so as to send maintenance messages to the other peers of the *Leafset*. It is important to note that, because of the inter-maintenance time of the KBR layer is much smaller than the DHT layer's one, the Algorithm-1 uses the *Leafset* knowledge maintained by the KBR layer which is relatively accurate.

A set of the following two elements is contained in the messages constructed by Algorithm-1:

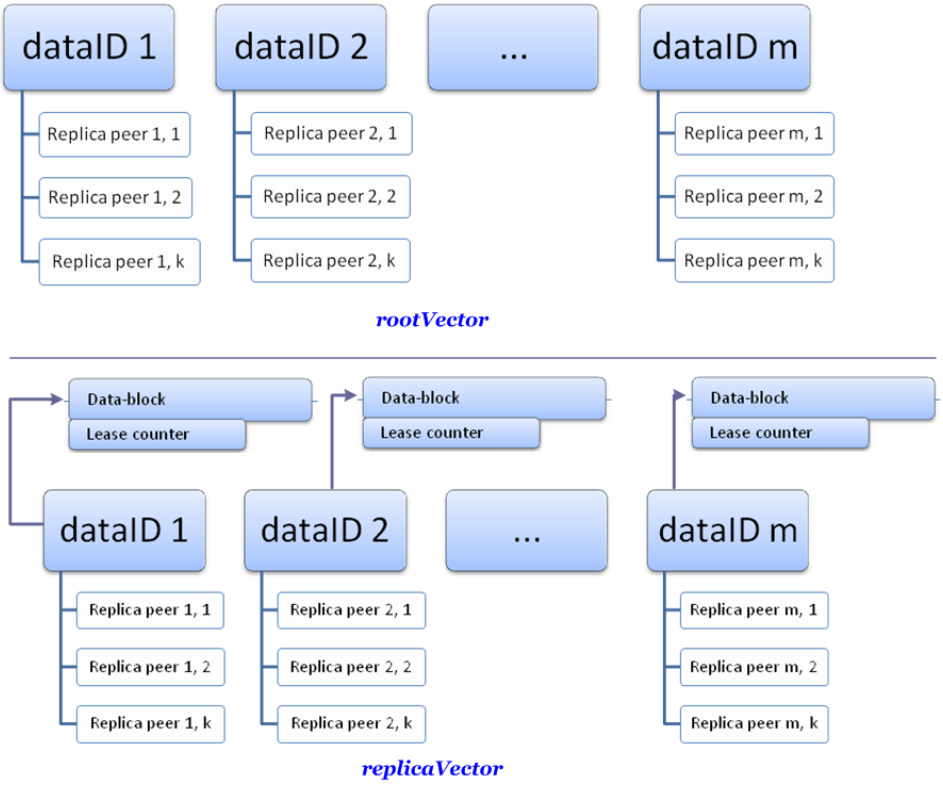

Figure 3.1: Data structures managed on each peer

Algorithm 1 Maintenance message construction

```
1: for all key ∈ rootVector do
2:     for all replica ∈ key.replicaSet do
3:         if NOT isInCenter (replica, leafset) then
4:             newPeer = choosePeer(replica, leafset)
5:             replace(key.replicaSet, replica, newPeer)
6:         end if
7:     end for
8:     for all replica ∈ key.replicaSet do
9:         add(messag[replica], < AddData, key.blockID, key.replicaSet >)
10:    end for
11:    for key ∈ replicaVector do
12:        if NOTcheckRoot(key.rootPeer, leafset) then
13:            newRoot = getRoot(key.rootPeer, leafset)
14:            add(messag[newRoot], < NewRoot, key.blockID, key.replicaSet >
    )
15:        end if
16:    end for
17:    for all p ∈ leafset do
18:        if NOTempty(messag[p]) then
19:            send(messag[p], p)
20:        end if
21:    end for
22: end for
```

1. *AddData* element for asking a replica node to keep storing a specific data block;

2. *NewRoot* element for notifying a node that it has become the new root of a data block.

These message elements contain both a data block identifier and the associated Replica-set peer-list. In order to remain scalable in terms of the number of data blocks Algorithm-1 sends at most one single message to each *Leafset* member.

We can subdivide the Algorithm-1 into three phases:

1. The first phase computes *AddData* elements using the rootVector structure. (from the first lines 1 to the 10^{th} line);

2. the second phase computes *NewRoot* elements using the replicaVector structure (from the 11^{th} line to the 16^{th} line);

3. the last phase sends messages to the destination peers in the *Leafset* (from the 17^{th} line to the end).

Message elements computed in the two first phases are added in msgs[]. The msgs[q] is a message like the one presented by Figure 3.2 containing all the elements to send to node q at the last phase.

Therefore, each peer periodically sends a maximum of *Leafset*-size maintenance messages to its neighbors. In the first phase, for each block for which the peer is the root, it checks if every replica is still in the center of its *Leafset* (the 3^{rd} line) using its local view provided by the KBR layer. If a replica node is outside, the peer replaces it by randomly choosing a new peer in the center of the *Leafset* and it then updates the Replica-set of the block (the 4^{th} and 5^{th} lines). Finally, the peer adds a *AddData* element in each replica set peers messages (the 8^{th} and 9^{th} lines). In the second phase, for each block stored by the peer (i.e., the peer is part of the block's Replica-set), it checks if the root node did not change. This verification is done by comparing the replicaVector metadata and the current *Leafset* state (the 12^{th} line). If the root has changed, the peer adds a *NewRoot* message element to announce to the future root peer that it is the root of the data block.

Finally, from the 17^{th} to 22^{nd} line, a loop sends the computed messages to each *Leafset* member.

3.1.2 Maintenance message treatment

Upon reception of a maintenance message, a peer executes Algorithm-2. For a *AddData* element (the 2^{nd} line), if the peer already stores a copy of the corresponding

Algorithm 2 Maintenance message reception

```
1: for elt ∈ message do
2:     if elt.type = AddData then
3:         if elt.data ∈ replicaVector then
4:             newLease(replicaVector, elt.data)
5:             updateRepSet(replicaVector, elt.data)
6:         else
7:             requestBlock(elt.data)
8:         end if
9:     else if elt.type = NewRoot then
10:        rootVector = rootVector ∪ elt.data
11:    end if
12: end for
```

data block, it resets the associated lease counter and updates the corresponding Replica-set if necessary (from the 3^{rd} line to the 6^{th} lines). If the peer does not store the associated data block (i.e., it is the first *AddData* message element for this data block received by this peer), it fetches it from one of the peers mentioned in the received Replica-set (the 7^{th} line).

For a NewRoot element a peer adds the data block-id and replicaset in the rootVector structure (the 9^{th} line).

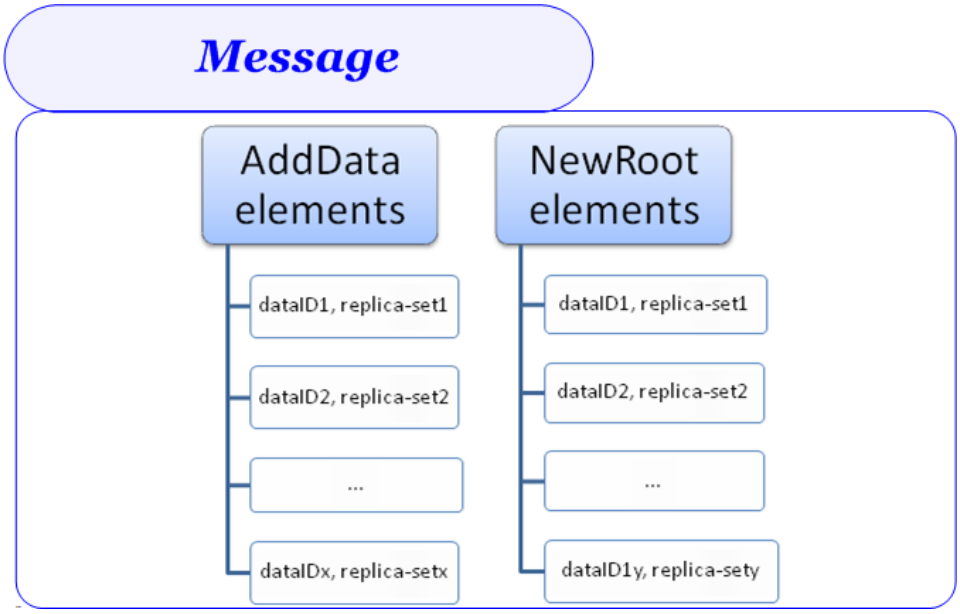

Figure 3.2: Message composed of x AddData elements and y NewRoot elements

3.1.3 End of a lease treatment

If a data block lease counter reaches 0, it means that no AddData element has been received for a long time. This can be the result of numerous insertions that have pushed the peer outside the center of the *Leafset* of the data block's root. The peer sends a message to the root peer of the data block to ask for the authorization to delete the block. Later, the peer will receive an answer from the root peer which either allows it to remove the data block or asks it to put the data block again in the DHT (in case the data block has been lost).

Chapter 4

Implementation and Simulation

This Chapter present the simulation environment used in our study and the implementation of our replication strategy.

4.1 Simulation evironment

The simulations were carried out by OverSim (Baumgart et al. (2007)) overlay network simulation framework. Oversim is designed to operate on top of another network simulator, OMNeT++ (Omnet++ (2011*b*)).

4.1.1 OMNeT++ Discrete Event Simulator

OMNeT++ (Omnet++ (2011*b*)) is a discrete event simulation tool built by Andras Varga at the Technical University of Budapest, Hungary. It contains a network editor (GNED) that allows for easy construction of various network topologies on a workspace, a graphical simulation execution environment (Tkenv) that allows for configuration and observation of a simulation run, event-by-event (if desired) and two statistical recording and analysis tools (plove and scalars) for visualizing and analyzing statistics generated during the course of the simulation run.

4.1.2 INET Framework for OMNeT++

The INET Framework (Omnet++ (2011*a*)) is an open-source set of models built in OMNeT++ and meant for the simulation of various network protocols and topologies, such as wired, wireless and ad-hoc networks. It incorporates various protocol suites, such as TCP-IP, PPP, IEEE 802.11, Ethernet, IPv4, IPv6, OSPF, etc. that

can be used in combination with models of network components such as hosts, routers and buses among others to build models of networks and test them.

4.1.3 OverSim

Oversin is the P2P Overlay Simulation Framework for OMNeT++. OverSim (Team (2011)) is developed under the scope of the ScaleNet (ScaleNet (2011)) project at the Institute of Telematics, Universitt Karlsruhe, Germany. OverSim allows the simulation of P2P overlay networks, using OMNeT++ and the INET Framework. Of particular interest that made it suitable for this project was the implementation of various DHT algorithms and network models built using the same, one of which was Chord.

OverSim consists of modules that are defined in a simple definition language NED. The modules are implemented in C++. Figure 4.1 illustrates OverSim's modular architecture.

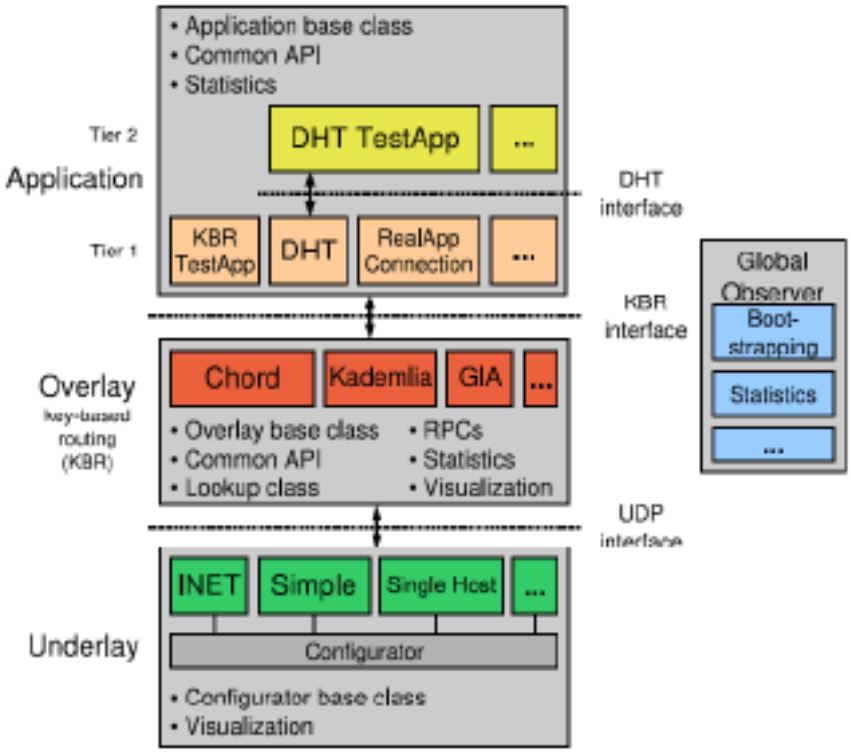

Figure 4.1: OverSim architechture

In this work the OverSim-20100526 release is used. We have chosen Oversim

because Oversim is an open-source overlay and peer-to-peer (P2P) network simulation framework having important features:

- Interactice GUI: Demonstrate network topology and message transfert in network.
- Scalability: Oversim can simulate networks of up to 100000 nodes.
- Fexibility: Oversim simulates both structured and unstructured P2P systems and overlay protocols.
- Routing models: All implemented protocols supports routing modes like iterative, exhaustive-iterative, semi-recursive, full-recursive, and source-routing -recursive.
- Underplaying Network: The underlying network can be easily configured or replaced using configuration file for network topology with realistic bandwidths, packet delays.
- Applications: Oversim default come with several applications like DHT, Internet Indirection Infrastructure (i3), P2PNS and some test applications as well.
- Oversim supports different Churn models like LifetimeChurn and ParetoChurn.
- Reusability: Different implementation of protocol is reusable for network applications.

OverSim supports different underlying network models. In the release we use, there are three models available: Simple underlay; Single host underlay and IPv4 underlay. In our study we use the Simple underlay which is illustrated in Figure 4.2 and is used for large networks because of its scalability. In this underlay model, packets are sent directly from one overlay node to another and have a constant delay. OverSim has three different Churn models to choose from: Lifetime Churn, Pareto Churn and Random Churn. In our study Pareto Churn is used.

OverSim implements DHT algorithms Chord, Pastry, and Kademlia that are all used in our simulations. OverSim also implements the desired routing modes and provides a generic lookup mechanism that can be used to test these different key based routing alternatives (Baumgart et al. (2007)).

OverSim has two configuration files that are used to specify all the relevant simulation parameters. A file called default.ini contains all those parameters and another file called omnetpp.ini contains simulation run specific parameter settings. The parameters in omnetpp.ini replace the values in default.ini if there is any overlapping between the two files. Omnetpp.ini is used for making different kinds of

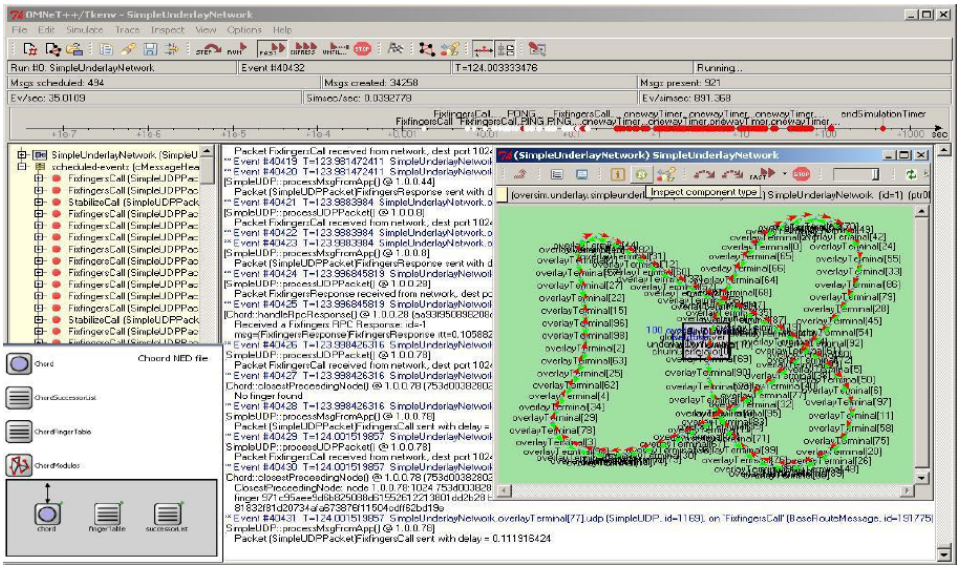

Figure 4.2: Screen shot for OverSim

simulation scenario settings. These scenarios can then be run without separately configuring the default.ini file. Both configuration files, Omnetpp.ini and default.ini, are presented in Appendix A.

4.2 Distribution used in the simulations

Churn affects many aspects of P2P computing, including lookup efficiency and network stabilization. It is crucial to capture these characteristics to increase the accuracy of our performance evaluation studies.

In a Weibull- or Pareto-based Churn model, distributions are generated by assuming that the durations of peer up times and down times follow these distributions. Some results indicate that durations of peer sessions in a typical P2P file sharing network follow a heavy tailed distribution (Gummadi et al. (2003)). For this reason, we have decided to base our models of churn on Pareto distributions.

The authors of "Modeling Heterogeneous User Churn and Local Resilience of Unstructured P2P Networs" (Yao et al. (2006)), have proposed a Pareto distribution churn model to capture the behavior of a single user. This model is based on the results of (Yao et al. (2006)). A user's behaviour is based on the specified about peer arrival, departure and selection of neighbours. Namely, it is assumed that the P2P network has a total of N peers, and each peer i can either be active (connected to the network) at time t or sleeping (disconnected from the network). An alternating renewal process $Zi(t)$ for each peer i modeles such a dynamic behaviour.

$$Z_i(t) = \begin{cases} 1 & \text{if user } i \text{ is alive at time } t, \\ 0 & \text{otherwise.} \quad , 1 \leq i \leq N \end{cases} \tag{4.1}$$

The subscript c stands for the cycle number of peer i. ON (active) and OFF (sleeping) periods are represented by random variables $A_{i,c} > 0$ and $S_{i,c} > 0$, respectively. The residual variable R_i, which is the duration of peer i remaining ON (active) period from time instant t is also shown. A possible dynamic behavior of an individual user (peer) is illustrated by Figure 4.3.

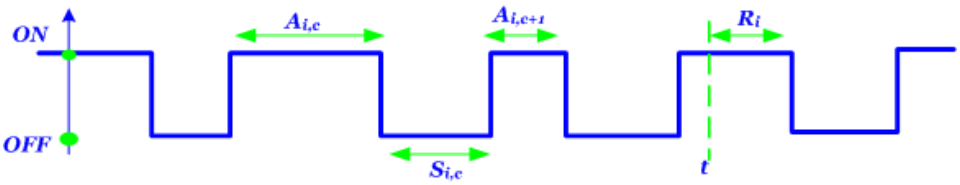

Figure 4.3: Churn model repreeenting the ON or OFF behaviour of peer i

It is assumed by the model that peers do not synchronize their arrival or departure times and have uncorrelated lifetime characteristics. It is also assumed that it will be uncommon for peers to be present in the system with multiple identities, or in other words, we assume that peers with such multiple identities do not have a significant impact on the dynamics of the network. It is also assumed that all $\{Z_i(t)\}$ have their ON durations $\{A_{i,c}\}_{c=1}^{\infty}$ governed by the same distribution, given by Equation (4.2). A similar assumption applies to OFF durations $\{S_{i,c}\}_{c=1}^{\infty}$ which are ruled by the distribution on Equation (4.3).

The assumption that both ON and OFF durations are governed by a shifted Pareto distribution means:

$$F_{Ai}(t_a) = 1 - (1 + \frac{t_a}{\beta_{A_i}})^{-\alpha_{A_i}} \quad \text{for } t_a > 0, \alpha_{A_i} > 1, \beta_{A_i} > 0, \tag{4.2}$$

$$F_{Si}(t_s) = 1 - (1 + \frac{t_s}{\beta_{S_i}})^{-\alpha_{S_i}} \quad \text{for } t_s > 0, \alpha_{S_i} > 1, \beta_{S_i} > 0, \tag{4.3}$$

Further, since we are assuming that they are identical, $\alpha_{A_i} = \alpha_{S_i} = \alpha_i$ and $\beta_{A_i} = \beta_{S_i} = \beta_i$.

Let the average activity time be $E[A_i]$, and the average sleeping duration be $E[S_i]$.

According to our assumptions,

$$E[A_i] = E[S_i] = \frac{\beta_i}{\alpha_i - 1} \tag{4.4}$$

The β_i value equals

$$\frac{1}{E[A_i](\alpha_i - 1)} \tag{4.5}$$

for activity time, and

$$\frac{1}{E[S_i](\alpha_i - 1)} \tag{4.6}$$

for sleeping time.

The peer's availability, i.e. probability that a given peer is in the network at a random instance t, $t >> 0$, is given by:

$$P_{A_i} = \lim_{t>>0} P(Z_i(t) = 1) = \frac{E[A_i]}{E[A_i] + E[S_i]}; \tag{4.7}$$

In simulations, during the network establishment phase, the network will populate itself to reach the maximum capacity, and each individual peer i will draw its active time from the distribution described by Equation (4.2) and its sleeping time duration from Equation (4.3). We can set the average peer session length in our simulations by adjusting the average active and sleeping parameters. Once the network is established, each individual peer i will stay connected in the network until its assigned lifetime, and disconnect from the network afterwards. Until the sleeping duration of peer i expires, the network will not add a new peer.

4.3 The OverSim Simulator Classes

DHTs are tested with OverSim's DHT test application which we have modified (as presented in Appendix B) in order to equate the behavior of Free_DHT protocol. Appendix D present some code source of Free_DHT protocol.

OverSim classes modified for the simulations that are run in this study and the relations between them are presented in Figure 4.4. DHTTestApp and GlobalDhtTestMap classes are located in the Tier2 library that communicates with the Applications library containing the DHT and DHTDataStorage classes. In the Overlay library there are Chord, Pastry, and Kademlia classes that define the logic that the DHT class executes. These two classes, illutrated in Appendixes B and C, are subclasses of the BaseOverlay class located in the Common library. SimpleNetConfigurator class in the Underlay library defines the parameters for the network running below the overlay.

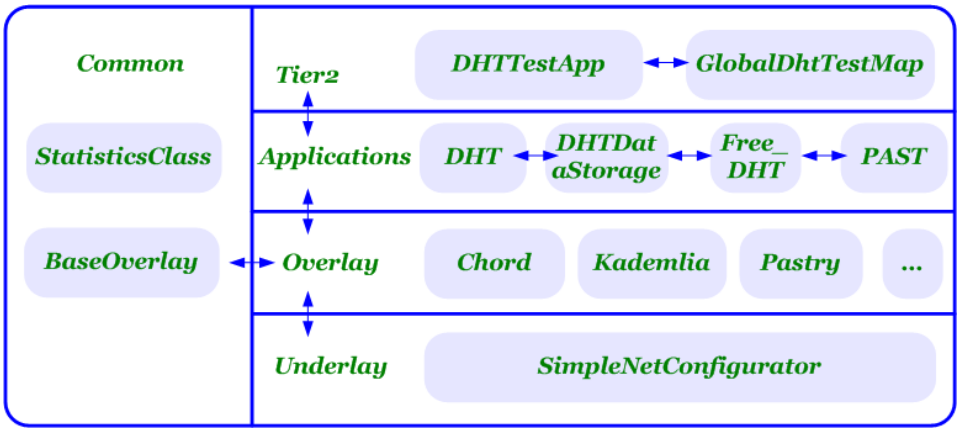

Figure 4.4: OverSim libraries and modified classes

The effect of four parameters is tested in our simulations and other parameters remained constant at their default value while these four were varied one at a time. These parameters include the number of nodes in the network, lifetime of the nodes, time between key updates and time between FETCH-messages sent by each node.

OverSims RECORD_STATS tool and our MyClass were used to collect statistics. The statistics collected with RECORD_STATS are automatically printed to a file called omnetpp.sca which is then processed with an OverSim script overStat.pl that prints the statistics in a more readable fashion. The file named Results.dat contains the statistics calculated in MyClass class.

Chapter 5

Performance Evaluation

This chapter provides a comparative evaluation of our replication strategy and PAST. It presents also the results of the performance evaluation study of Chord, Pastry, and Kademlia. The aim is to investigate the effects of Churn on the performance of these P2P networks. The results were obtained by means of stochastic discrete-event simulation, using OverSim as a simulator tool.

5.1 Simulation setup

To evaluate our solution, we have built a discrete event simulator using the Omnet++/OverSim. We have based our simulator on an already existing OverSim module simulating the Pastry KBR layer. We have also implemented a DHT relying on both the PAST strategy and our replication strategy on top of this module. It is important to note that all the different layers and all message exchanges are simulated. Our simulator also takes into account the network congestion: in our case, the network links may often be congested.

Figure 5.1 shows the component modules that would constitute a Pastry, Chord and Kademlia simulation model using a simple point-to-point protocol (PPP) for communication between peers. On the main simulation workspace are:

Global Observer: This is a module to set global parameters of the simulation, it provides the identifier of a bootstrap peer to a node joining the network, and collects global statistics.

Underlay Configurator: This is used to specify parameters of the underlying network.

Churn Generator: This is used to specify the type of churn (random churn, no churn, etc.), how often it occurs and other parameters related to nodes joining

and leaving the network.

Overlay Terminal: This is the model of the peer. This contains a UDP module that generates UDP messages for the peer, an Overlay module that contains the three essential modules of the Pastry DHT (Pastry algorithm, routing table and *Leafset*) or three essential modules of the Chord DHT (Chord algorithm, finger table and successor list) or three essential modules of the Kademlia DHT (Kademlia algorithm, routing table and sibling) and finally a test application module used to test the routing of messages to peers.

The above mentioned modules are common to all simulations.

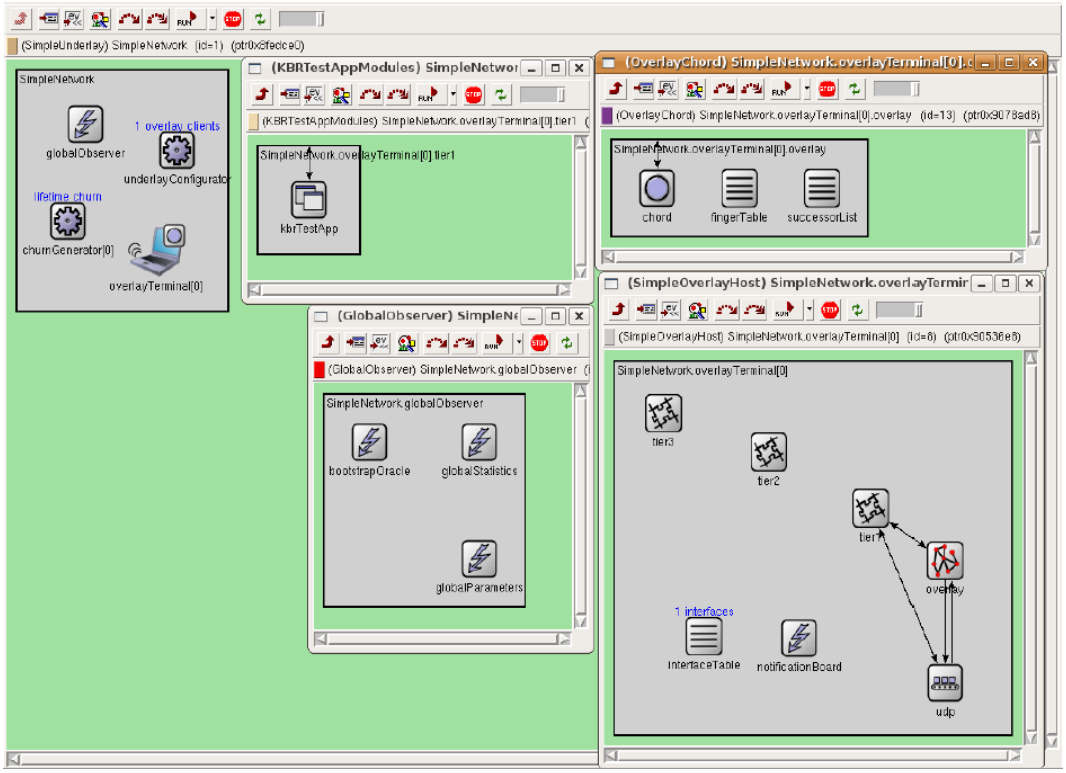

Figure 5.1: Components of a Pastry simple network simulation model.

The Figure 5.2 would constitute a Pastry (Chord or Kademlia) simulation model using a simple point-to-point protocol (PPP) for communication between peers.

The additional modules seen in this screen in the Figure 5.2 are those belonging to the access and backbone routers.

Figure 5.3 shows the main simulation workspace of figure 5.2 during simulation of a Pastry DHT on an Ipv4 network with 100 peers.

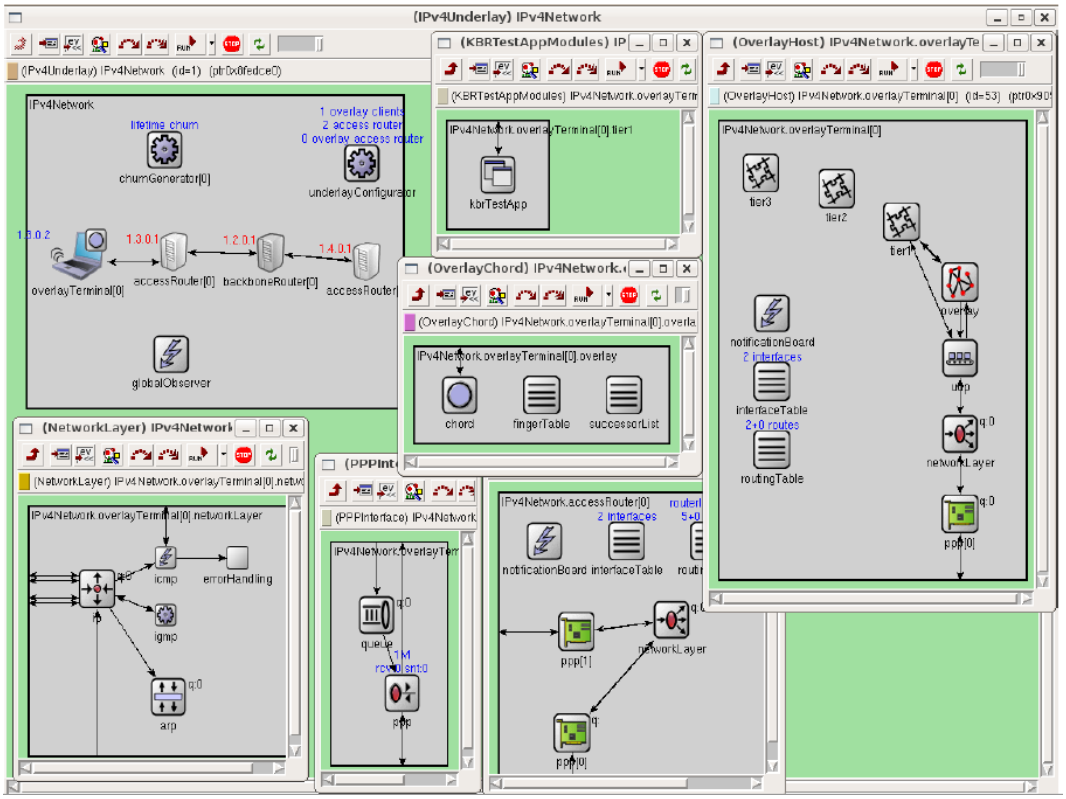

Figure 5.2: Modules used in a Pastry simulation on an Ipv4 network.

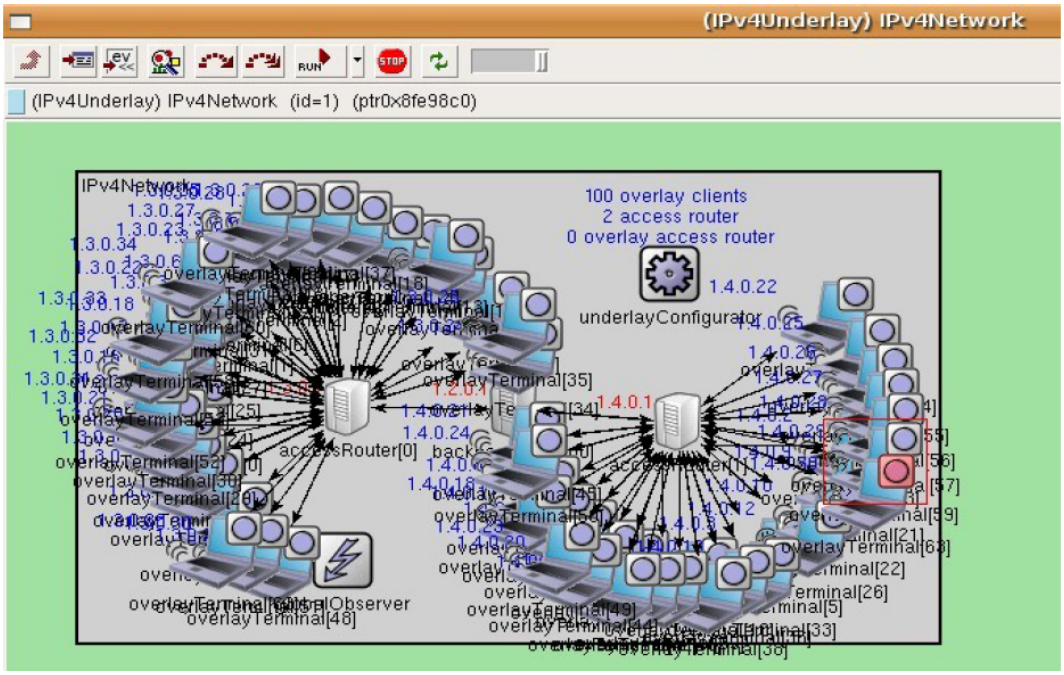

Figure 5.3: Simulation of a Pastry DHT on an Ipv4 network with 100 peers

For all the simulation results presented in this section, we used a 100 peers network with the following parameters (for both PAST and Free_DHT): a *Leafset* size of 24; an inter-maintenance duration of 10 minutes at the DHT level; an inter-maintenance duration of 1 minute at the KBR level; 10 000 data blocks of 10 000 KB replicated 3 times; network links of 1 $Mbits/s$ for upload and 10 $mbits/s$ for download with a delay uniformly chosen between 80 and 120 ms. A 100-peer network may seem a relatively small scale. However, for both replication strategies, PAST and our solution, the studied behavior is local, and contained within a *Leafset* (which size is bounded). It is however necessary to simulate a whole ring in order to take into account side effects induced by the neighbor *Leafsets*. Furthermore, a tradeoff has to be made between system accuracy and system size. In our case, it is important to simulate very precisely all peer communications. We have run several simulations with a larger scale (1000 peers and 100,000 data blocks) and have observed similar phenomenons.

5.2 Simulation Result

We have injected Churn following the three different scenario:

One hour Churn: One perturbation phase with Churn during one hour. This

phase is followed by another phase without connections/disconnections. In this case study, during the Churn phase each perturbation period is chosen randomly either as a new peer connection or a peer disconnection. This perturbation can occur anywhere in the ring (uniformly chosen). We have run numerous simulations varying the inter-perturbation delay.

Continuous Churn: For this set of simulations, we focus on phase one of the previous case. We study the system while varying the interperturbation delay. In this case, perturbation can be either a new peer connection or a disconnection. We also experiment a scenario for which only one peer gets disconnected. We then study the reaction of the system. The first set of experiments allows us to study:

1. How many data blocks are lost after a period of perturbation and;
2. How long it takes the system to return to a state where all remaining or non lost data blocks are replicated k times.

In real life systems there will be some period without Churn. The system has to take advantage of them to converge to a safer state. The second set of experiments zooms on the perturbation period and provides the ability to study how the system can resist when it has to repair lost copies in presence of Churn. Finally, the last set of simulations is done to measure the reparation of one single failure.

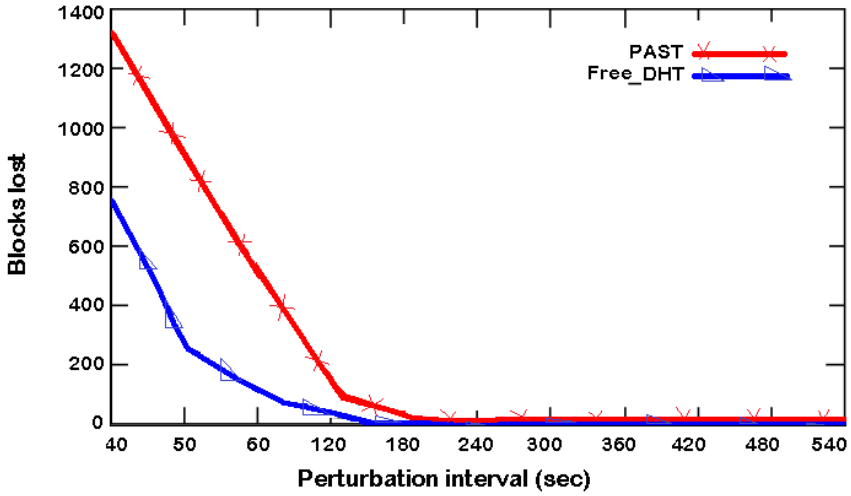

Figure 5.4: Number of data block lost

5.2.1 Losses and stabilization time after one hour Churn

We first study the number of lost data blocks (data block for which the 3 copies are lost) in PAST and in our solution under the same Churn conditions. Figure 5.4 shows the number of lost data blocks after a period of one hour of Churn. The inter-perturbation delay is increasing along the X axis. With our solution and our maintenance protocol, the number of lost data blocks is much lower than the PAST's. It reaches 50% for perturbations interval from lower than 50 seconds. The main reason of the result presented above is that, using PAST replication strategy, the peers have more data blocks to download. This implies that the mean download time of one data block is longer using PAST replication strategy. Indeed, the maintenance of the replication scheme location constraints generate a continuous network traffic that slows down critical traffic whose goal is to restore lost data block copies.

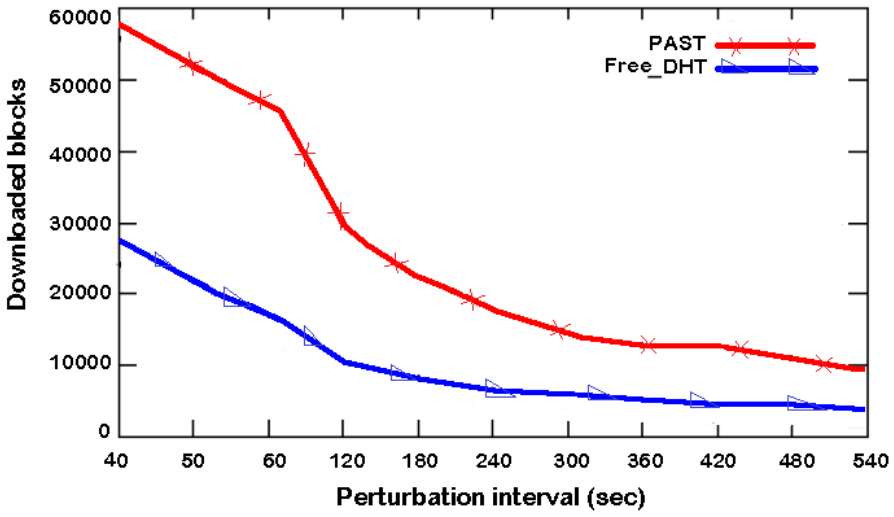

Figure 5.5: Number of exchanged data blocks to restore a stable state

Figure 5.5 shows the total number of blocks exchanged for both cases. There again, the X axis represents the inter-perturbation delay. The figure shows that with our replication strategy the number of exchanged blocks is always near 2 times smaller than in the PAST. This is mainly due to the fact that in the PAST case, many transfers (nearly half of them), are only done to preserve the replication scheme constraints. For instance, each time a new peer joins the DHT, it becomes root of some data blocks (because its identifier is closer than the current root-peer's one), or if it is inserted within Replica-sets that should remain contiguous. Using PAST replication strategy, a newly inserted peer may need to download data blocks during many hours, even if no failure/disconnection occurs. During all this time, its neighbors need to send it the required data blocks, using a large part of their upload

bandwidth. In our case, no or very few data blocks transfers are required when new peers join the system. It becomes mandatory only if some copies become too far from their root-peer in the logical ring. In this case, they have to be transferred closer to the root before their hosting peer leaves the root peer's *Leafset*. With a replication degree of 3 and a *Leafset* size of 24, many peers can join a *Leafset* before any data block transfer is required.

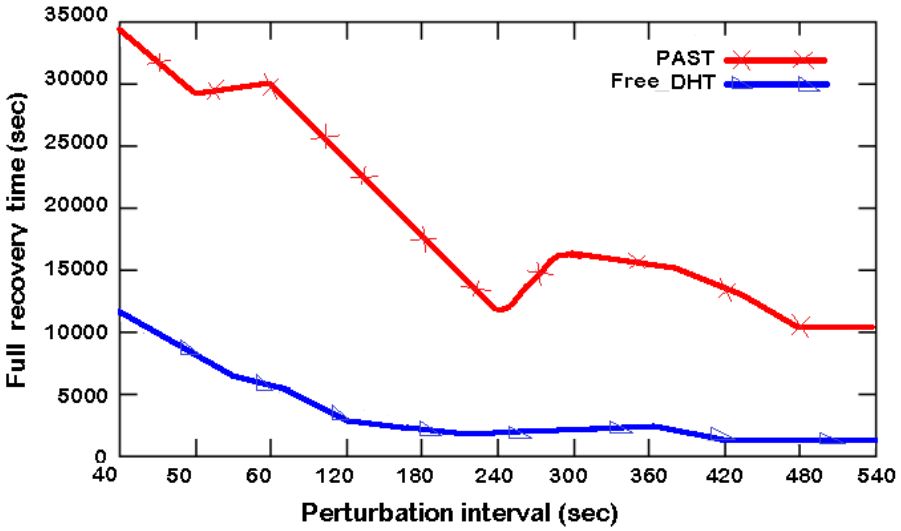

Figure 5.6: Recovery time: time for retreiving all the copies of every remaining data block.

Finally, we have measured the time the system takes to return in a normal state in which every remaining 3 data block is replicated k times. Figure 5.6 shows the results obtained while varying the delay between perturbations. We can observe that the recovery time is twice longer in the case where PAST is used compared to our solution. This result is mainly explained by the number of blocks to transfer which is much more lower in our case: our maintenance protocol transfers only very few blocks for location constraints compared to PAST's one. This last result shows that the DHT repairs damaged data blocks (data blocks for which some copies are lost), faster than PAST when using our replication strategy. It implies that it will recover very fast and be able to cope with a new Churn phase. The next section describes our simulations with continuous Churn.

5.2.2 Continuous Churn

Before presenting simulation results under continuous Churn, it is important to measure the impact of a single peer failure/disconnection. When a single peer fails,

stored data blocks have to be replicated on a new one. Those blocks are transferred to such a new peer in order to rebuild the initial replication degree k.

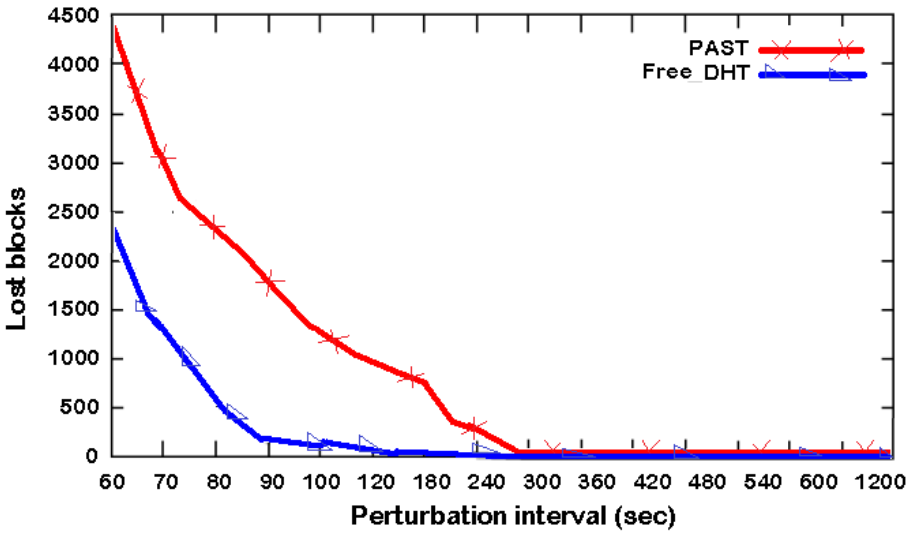

Figure 5.7: Number of data blocks losses while the system is under continuous Churn, varying inter-perturbation delay

In our simulations, with the parameters given above, it takes 4609 seconds for PAST to recover the failure: i.e., to create a new replica for each block stored on the faulty peer. While, with our solution, it takes only 1889 seconds. The number of peers involved in the recovery is much more important. This gain is due to the parallelization of the data blocks transfers:

- in PAST, the content of contiguous peers is really correlated. With a replication degree of 3, only peers located at one or two hops of the faulty peer in the ring may be used as sources or destinations for data transfers. In fact, only $(k+1)$ peers are involved in the recovery of one faulty peer, where k is the replication factor;
- in our solution, most of the peers contained in the faulty peer *Leafset* (the *Leafset* contains 24 peers in our simulations) may be involved in the transfers.

The above simulation results show that our replication stratagy:

1. Induce less data transfers, and
2. Remaining data transfers are more parallelized.

For these reasons even if the system remains under continuous Churn, our solution will provide a better Churn tolerance. Such results are illustrated in Figure 5.7. We can observe that, using the parameters described at the beginning of this section, PAST starts to lose data blocks when the inter-perturbation delay is around 7 minutes. This delay has to reach less than 4 minutes for data blocks to be lost using our replication strategy. If the inter-perturbation delay continues to decrease, the number of lost data blocks using our replication strategy remains near half the number of data blocks lost using PAST strategy.

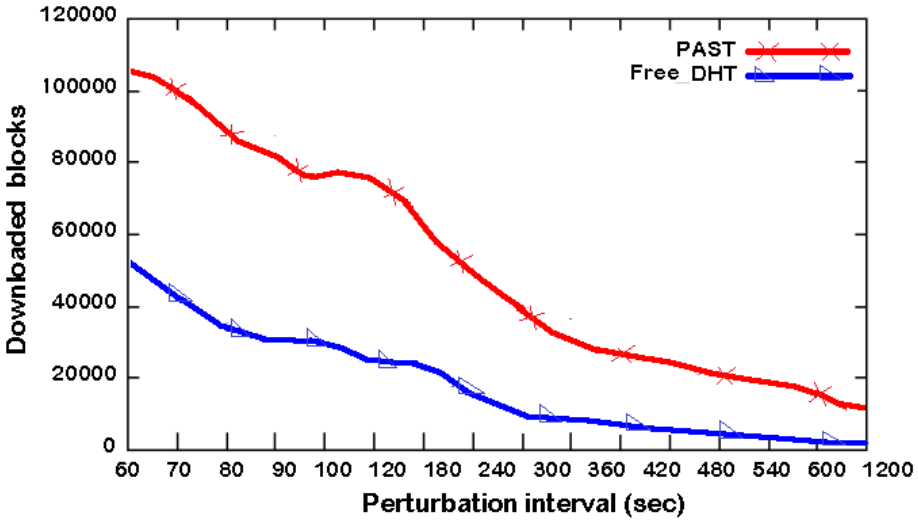

Figure 5.8: Number of data blocks transfers required while the system is under continuous Churn, varying inter-perturbation delay

Finally, Figure 5.8 confirms that even with a continuous Churn pattern, during a 5 hour run, the number of data transfers required by the proposed solution is much smaller (around half) than the number of data transfers induced by PAST's replication strategy.

5.2.3 Maintenance protocol cost

In the simulation results presented above, we have considered that maintenance protocol message size was negligible. Both PAST and Free_DHT maintenance protocols require x messages;

$$x = N * m \tag{5.1}$$

Where N is the number of peers in the whole system and m is the *Leafset* size.

We explain below that in absence of Churn, while using our solution, m can be reduced to the nodes in the center of the *Leafset* (smaller than *Leafset* size).

For PAST, each peer periodically sends a maintenance message to each node of its *Leafset*. This message has to contain the identifier of each stored block: an average of y identifiers:

$$y = \frac{M * k}{N} \tag{5.2}$$

Where M is the total number of blocks in the system and k the mean replication factor.

A peer stores data blocks for which it is the root, but also copies of data blocks for which its immediate $k - 1$ logical neighbors are root. Therefore each peer sends and receives z at each period (this can be lowered through the use of *Bloom Filters*).

$$z = \frac{M * k * Leafset\_size * Id\_size}{N} \tag{5.3}$$

For our solution, in absence of Churn the messages contain only *AddData* message elements. A peer is root of an average of v data blocks.

$$v = \frac{M}{N} \tag{5.4}$$

These v data blocks are replicated in average on k peers distributed in the center of the *Leafset*: the m inner peers. This implies that the average number (w) of *AddData* elements per message is:

$$w = \frac{M}{N} * \frac{k}{m} \tag{5.5}$$

w blocks for each of the peers in its *Leafset*. Furthermore, if a replica set has not changed since last maintenance, it is not necessary to send the replica set again to all of its members. Therefore, each maintenance message in absence of Churn has to contain identifiers of each block for which the source is the root and the destination is part of the replica set: an average of r identifiers, which is *Leafset* size times lower than in the PAST case.

$$r = \frac{M}{N} * \frac{k}{m} * m = \frac{M * k}{N} \tag{5.6}$$

PAST uses *Bloom Filters* to convey identifier lists. In absence of Churn, i.e., when the *Leafset* is equal to the one at the previous period, it is also possible to use *Bloom Filters* in our solution. In presence of Churn, however, it becomes difficult to use *Bloom Filters* with our solution because message elements have a structure (data block identifiers associated to peer identifiers). For each block identifier, it may be necessary to send the block identifier and the peer identifiers to the members of the

block's replica set (k peers in average). Thus, if we put aside the *Bloom Filters* optimization, in our case each peer sends/receives x_1 identifiers at each period;

$$x_1 = \frac{M * k^2}{N} \tag{5.7}$$

While peers using PAST send/receive x_2 identifiers at each period;

$$x_2 = \frac{M * k * Leafset\_size}{N} \tag{5.8}$$

k being usually an order of magnitude lower than *Leafset* size. This is mainly due to the fact that PAST peers send their content to all the members of their *Leafset* while our replication strategy peers use extra metadata to compute locally the information that needs to be transferred from one peer to another. A smart implementation of our replication strategy should try to use *Bloom Filters* whenever it is possible. To put it in a nutshell, the cost of our maintenance protocol is close to the cost of PAST maintenance protocol.

5.2.4 Side effects and limitations

Our replication strategy for peer-to-peer DHTs, is relaxing placement constraints of data block copies in *Leafsets*, which significantly reduces the number of data blocks to be transferred when peers join or leave the system. Thanks to this, we show in the next section that our maintenance mechanism allows us to better tolerate Churn, but it implies other effects. The two main effects concern the data block distribution on the peers and the lookup performance. While the changes in data blocks distribution can provide positive effects, the lookup performance can be damaged.

5.2.5 Data blocks distribution

While with usual replication strategies in peer-to-peer DHT's, the data blocks are distributed among peers according to some hash function. Therefore, if the number of data blocks is big enough, data blocks should be uniformly distributed among all the peers of the system. With both *Leafset-based* replication and multiple key based replication, this remains true even when peers leave or join the system, due to the maintenance algorithms. When using our solution, this remains true if there are no peer connections/disconnections. However, in presence of Churn, our maintenance mechanism does not transfer data blocks if is not necessary, new peers will store much less data blocks than peers involved for a longer time in the DHT. That way; more stable a peer is, more data blocks it will store.

5.2.6 Possible impact of *Bloom Filters*

A *Bloom Filters*, conceived by Burton Howard Bloom in 1970, (Knuth (1968)), is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. False positives are possible , but false negative are not; i.e. a query returns either "inside set (may be wrong)" or "definitely not in set". Elements can be added to the set, but not removed (though this can be addressed whith a counting filter). The more elements that are added to the set, the larger the probability of false positives.

The impact of *Bloom Filters* in presence of Churn by sending the block identifier and the peer identifiers to the members of block's replica set, so each peer could send/receive x_1 (as in Equation:(5.7)) identifiers at each period.

5.2.7 Lookups processes assessment

Our studies are intended for assessing the file lookup processes. We assume that once a match is found, the keyword from the file lookup is copied to the source node, and no content retrieval is performed.

- Assumption 1: Behaviour of a user is described by alternative renewal process given in Equation (4.1). Following our discussion in the previous Section, it means that each user spends a time interval of random length in activity state A_i, followed by a time interval of random length in sleeping state S_i. In all cases, the lengths of these time intervals are governed by Pareto distributions given by Equation (4.2) and Equation (4.3). This assumes that all users operate in the same way, so they are governed by the same Pareto distribution;
- Assumption 2: A simulated P2P network operates at its full capacity from the beginning of simulation. It means that in the initial state of simulation all peers are active;
- Assumption 3: A P2P network has N homogeneous peers. Unless otherwise stated, we first assume $N = 100$ and then $N = 1,000$.

Their homogeneity means that $\alpha_i = \alpha$ and $\beta_i = \beta$ for all i, given in Equation (4.2) and Equation (4.3).

Studies from (Rhea, Geels, Roscoe, & Kubiatowicz (2004)), (Herrera & Znati (2007)), (Li et al. (2004)) have suggested average activity periods ranging from few minutes to one hour. In (Ruiz & Znati (2005)) we observe that the results become constant beyond an average activity period of one hour. We have assumed the average activity periods of a peer to be 900, 1800, 2700 and 3600 seconds, respectively. This allows us to capture the dynamics of simulated processes related

with joining and leaving P2P networks by peers. Following (Herrera & Znati (2007)) and (Ruiz & Znati (2005)), we will consider the following performance metrics to represent the performance of file lookups processes:

- Hop count;
- Bandwidth consumption;
- Latency;
- Delivery ratio.

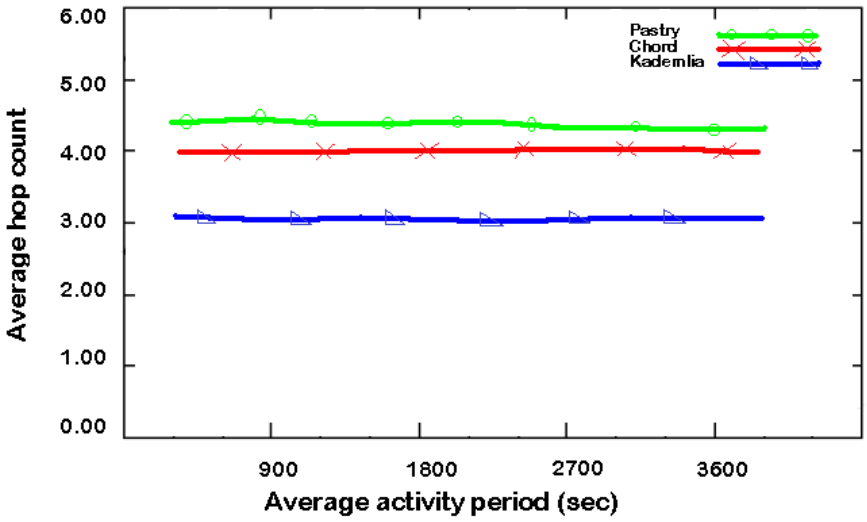

Figure 5.9: A plot of average hop count under the effects of Churn

Hop count: illustrated in Figure 5.9, Represents the distance between the source node who initiated the lookup and the destination node (the location of the searched value), measured by the number of intermediate nodes which needs to be traversed between the source and destination node.

Bandwidth consumption: illustrated in Figure 5.10.

Includes the amount of traffic generated by routing table updates and file lookups, measured by the number of messages generated during a fixed duration of time.

Latency: illustrated in Figure 5.11

The latencies measures the duration of the stabilization processes from sustaining Churn, as well as the duration of time needed for resolving file lookups from when it was initiated until it was responded to; measured in milliseconds.

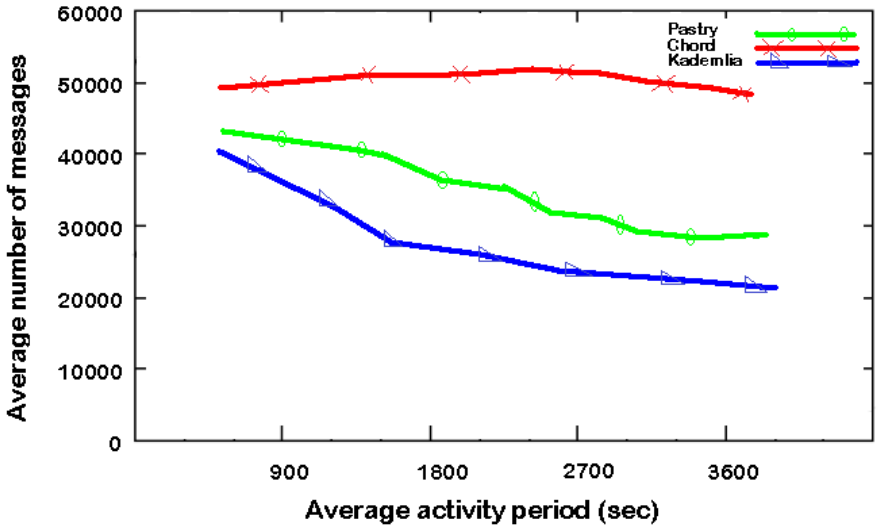

Figure 5.10: A plot of the average bandwidth under the effects of Churn

Delivery ratio: illustrated in Figure 5.12

The delivery ratio measures ratio of successful deliveries of file lookups to the destination node over a given time interval.

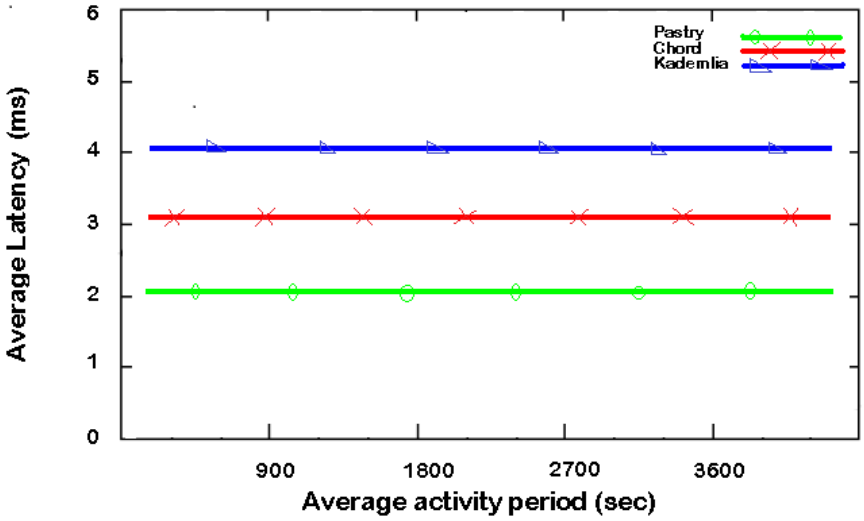

Figure 5.11: A plot of average latency under the effects of Churn

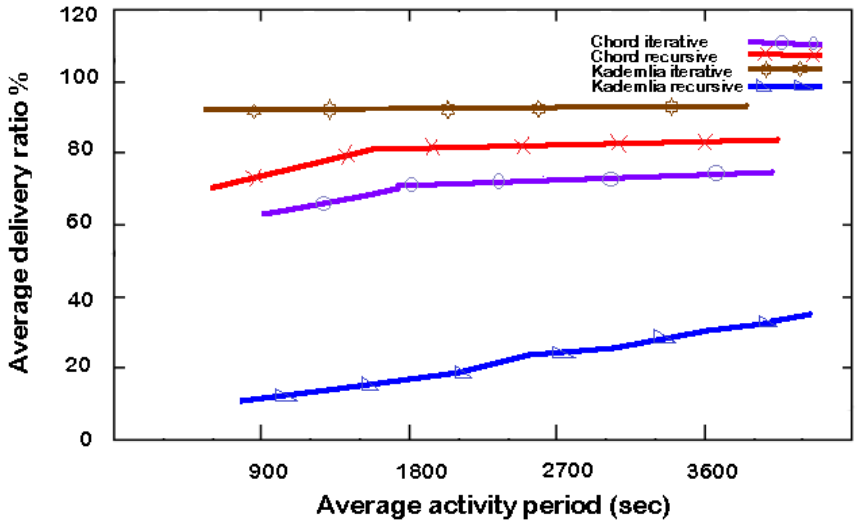

Figure 5.12: A plot of average delivery ratio for the iterative and recursive routing

Chapter 6

Conclusion and Future Work

Peer to peer distributed hash tables provide an efficient, scalable and easy to use storage system. However, existing solutions do not tolerate a high Churn rate or are not really scalable in terms of number of stored data blocks.

According to our findings, the existing solutions do not tolerate high Churn rate because of the strict placement constraints imposed on data placement, which induce unneceaasry data transfers.

6.1 Validation

We have proposed a new replication strategy, which is a solution that releases the placement constraints, and relies on metadata (replica-peers/data identifiers) to allow a more flexible location of data block copies within *Leafsets*. This significantely reduces the volume of data blocks to be transferred when peers join or leave the system. We have implemented both PAST and Free_DHT on top of OverSim.

The main results of our evaluations are:

1. Our approach achieves a higher data availability in presence of Churn, than the original PAST replication strategy;
2. For a connection/disconnection occuring every minute our strategy looses two times less blocks than PAST;
3. Our replication strategy induces an average of twice less block transfers than PAST.

Thus our replication strategy entails fewer data transfers than classical *Leafset-based* replication mechanisms. We have also demonstrated that the Churn resilience is obtained without adding a great maintenance overhead.

6.2 Future Work

Future work to be done include impact of *Bloom filter* in presence of Churn by sending the block identifier and the peer identifiers to the members of block's replica set, so each peer could send/receive x_1 (as in Equation:(5.7)) identifiers at each period.

Bibliography

Abu-Libdeh, H., Costa, P., Rowstron, A., OShea, G. & Donnelly, A. (2010), *Symbiotic routing in future data centers*, ACM SIGCOMM.

Adya, A., Bolosky, W., Castro, M., Chaiken, R., Cermak, G., Douceur, J., Howell, J., Lorch, J., Theimer, M., & Wattenhofer, R. (2002), *Farsite: Federated, available, and reliable storage for an incompletely trusted environment*, december edn, OSDI 02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation, Boston, MA, USA.

Akavipat, R., Dhadphale, A., Kapadia, A. & Wright, M. (2010), *ReDS: Reputation for directory services in P2P systems*, In Proceedings of The ACM Workshop on Insider Threats.

Alizadeh, M., Greenberg, A. G., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S. & Sridharan, M. (2010), *Data center TCP (DCTCP)*, In ACM SIGCOMM.

Ballani, H., Costa, P., Karagiannis, T. & Rowstron, A. (2011), *Towards Predictable Datacenter Networks*, In SIGCOMM.

Baumgart, I., Heep, B. & Krause, S. (2007), *OverSim: A Flexible Overlay Network Simulation Framework*, anchorage usa, may edn, in Proceedings of 10th IEEE Global Internet Symposium GI 07 in conjunction with IEEE INFOCOM 2007.

Beaver, D., Kumar, S., Li, H. C., Sobel, J. & Vajgel, P. (2010), *Finding a Needle in Haystack: Facebooks Photo Storage*, In Proc. of OSDI.

Bhagwan, R., Moore, D., Savage, S. & Voelker, G. (2003), *Replication strategies for highly available peer-to-peer storage*, vol. 2584 of lecture notes in computer science, 153158. springer edn, Future Directions in Distributed Computing.

Binzenh, A. & Leibnitz, K. (2007), *Estimating churn in structured p2p networks*, vol 4516 of lncs, 17 to 21 june edn, Proc of 20th International Teletraffic Congress, 630 to 641, Ottawa, Canada.

Blake, C. & Rodrigues, R. (2003), *High availability, scalable storage, dynamic peer networks:Pick two*, may edn, Proc. of HotOS IX, Lihue, Hawaii.

Branco, M. (2009), *Distributed Data Management for Large Scale Applications*, november edn, Faculty of Engineering, Science and MathematicsSchool of Electronics and Computer Science, University of Southampton.

Busca, J.-M., Picconi, F., & Sens, P. (2005), *Pastis: A highly scalable multiuser peer to peer file system*, august edn, Euro Par 05: Proceedings of European Conference on Parallel Computing.

Castro, M., Costa, M., & Rowstron, A. (2004), *Performance and dependability of structured peer-to-peer overlays*, washington, dc, usa: ieee computer society, june edn, DSN 04: Proceedings of the 2004 International Conference on Dependable Systems and Networks.

Dabek, F., Kaashoek, F. M., Karger, D., Morris, R., & Stoica, I. (2001), *Wide-area cooperative storage with CFS*, vol. 35, no. 5, december edn, SOSP 01: Proceedings of the 8th ACM symposium on Operating Systems Principles, New York, NY, USA: ACM Press.

Dabek, F., Li, J., Sit, E., Robertson, J., Kaashoek, F. F., & Morris, R. (2004), *Designing a DHT for low latency and high throughput*, NSDI 04: Proceedings of the 1st Symposium on Networked Systems Design and Implementation, San Francisco, CA, USA, March.

Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J., & Stoica, I. (2003), *Towards a common API for structured peer to peer overlays*, Proc. of IPTPS.

Druschel, P. & Rowstron, A. (2001), *PAST: A large-scale, persistent peer-to-peer storage utility*, may edn, Proc. HotOS VIII.

DUMINUCO, A. (2009), *Data Redundancy and Maintenance for Peer-to-Peer File Backup Systems*, august edn, cole Doctorale dInformatique, Tlcommunications et lectronique de Paris.

Foundation, P. (2009), *General Python FAQ*, june 27 edn, www.python.org\ doc \ faq \ general \.

Foundation, P. S. (2007), *What is Python? Executive Summary, Python documentation*, march 21 edn, www.python.org \ doc \ essays \ blurb \.

Freenet, T. (2005), *Freenet*, www.freenetproject.org.

Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003), *The google file system*, october edn, SOSP 03: Proceedings of the 9th ACM symposium on Operating systems principles. New York, NY, USA: ACM Press.

Ghodsi, A., Alima, L. O., & Haridi, S. (2005), *Symmetric replication for structured peer-to-peer systems*, information systems and peer-to-peer computing, trondheim, norway, august edn, DBISP2P 05: Proceedings of the 3rd International Workshop on Databases,.

Gnutella, T. (2005), *gnutella*, pre v0.4 edn, www.rfc-gnutella.sourceforge.net.

Gummadi, K., Dunn, R., Saroiu, S., Gribble, S., Levy, H., & Zahorjan, J. (2003), *Measurement, Modeling, and Analysis of P2P File-Sharing Workload*, acm edn, in Proceedings of Special Interest Groupon Operating Systems.

Harvesf, C. & Blough, D. M. (2006), *The effect of replica placement on routing robustness in distributed hash tables*, washington, dc, usa: ieee computer society, september edn, P2P 06: Proceedings of the 6th IEEE International Conference on Peer-to-Peer Computing.

Harvesf, C. & Blough, D. M. (2011), *Replica placement for route diversity in tree-based routing distributed hash tables*, 8–3 edn, IEEE Trans. Dependable Sec. Comput.

Herrera, O. & Znati, T. (2007), *Modeling Churn in P2P Networks*, ieee edn, in Proceedings of Annual Simulation Symposium.

Jernberg, J., Vlassov, V., Ghodsi, A., & Haridi, S. (2006), *Doh: A content delivery peer to peer network*, september edn, Euro-Par 06: Proceedings of European Conference on Parallel Computing, Dresden, Germany.

Kim, K. & Park, D. (2006), *Reducing data replication overhead in DHT based peer-to-peer system*, september edn, HPCC 06: Proceedings of the 2nd International Conference on High Performance Computing and Communications, Munich, Germany.

Knuth, D. (1968), *The Art of Computer Programming*, errata for volume3, 2nd edn, Addison-Wesley.

Ktari, S., Zoubert, M., Hecker, A., & Labiod, H. (2007), *Performance evaluation of replication strategies in DHTs under churn*, december edn, MUM 07: Proceedings of the 6th international conference on Mobile and ubiquitous multimedia. New York, NY, USA: ACM Press.

Kunzmann, G. (2005), *Recursive or iterative routing? Hybrid!*, In the proceedings of the Gesellschaft fr Informatik GI.

Landers, M., Zhang, H., & Tan, K. L. (2004), *Peerstore: Better performance by relaxing in peer to peer backup*, august edn, P2P 04: Proceedings of the 4th International Conference on Peer to Peer Computing. Washington, DC, USA: IEEE Computer Society.

Legtchenko, S., Monnet, S., Sens, P. & Muller, G. (2009), *Churn resilient replication strategy for peer to peer distributed hash tables*, vol 5873 edn, Lecture Notes in Computer Science.

Li, J., Stribling, J., Gil, T., Morris, R., & Kaashoek, M. (2004), *Comparing the Performance of Distributed Hash Tables Under Churn*, in Proceedings of International Workshop on Peer-To-Peer Systems.

Li, J., Stribling, J., Morris, R., & Kaashoek, M. F. (2007), *Bandwidth-efficient management of DHT routing tables*, MIT Computer Science and Artificial Intelligence Laboratory.

Lian, Q., Chen, W., & Zhang, Z. (2005), *On the impact of replica placement to the reliability of distributed brick storage systems*, washington, dc, usa: ieee computer society, june edn, ICDCS 05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems.

Mahajan, R., Castro, M. & Rowstron, A. (2006), *Controlling the Cost of Reliability in Peer to Peer Overlays*, University of Washington Seattle, WA and Microsoft Research Cambridge, UK.

Maymounkov, P. & Mazieres, D. (2002), *Kademlia: A peer-to-peer information system based on the xor metric*, march edn, IPTPS 02: Proceedings of the 1st International Workshop on Peer-to-Peer Systems, Cambridge, MA, USA.

Omnet++, C. (2011*a*), *INET Framework*, on july edn, Omnet++ Site: www.omnetpp.org/staticpages/index.php?page=20041019113420757.

Omnet++, C. (2011*b*), *OMNeT++*, on july edn, Omnet++ Site: www.omnetpp.org.

Pita, I. & Riesco, A. (2012), *Specifying and Analyzing the Kademlia Protocol in Maude*, wrla edn, 9th International Workshop on Rewriting Logic and its Applications.

PythonSoftwareFoundation (2008), *What is Python Good For, General Python FAQ*, september 05 edn, www.docs.python.org \ faq \ general.html.

Ramabhadran, S. & Pasquale, J. (2006), *Analysis of long-running replicated systems*, april edn, Proc. of IEEE Infocom, Barcelona, Spain.

Ratnasamy, S., Francis, P., Handley, M., Karp, R., & Schenker, S. (2001), *A scalable content-addressable network*, vol. 31, no. 4, october edn, SIGCOMM, ACM Press.

Ratnasamy, S., Francis, P., Handley, M., Karp, R. & Shenker, S. (2001), *A scalable content-addressable network*, august edn, Proc. ACM SIGCOMM 01, San Diego, California.

Rhea, S., Geels, D., Roscoe, T., & Kubiatowicz, J. (2004), *Handling churn in a DHT*, june edn, Proceedings of the 2004 USENIX Technical Conference, Boston, MA, USA.

Rhea, S., Geels, D., Roscoe, T. & Kubiatowicz, J. (2004), *Handling Churn in a DHT*, june edn, Proceedings of the USENIX Annual Technical Conference, University of California, Berkeley and Intel Research, Berkeley.

Rodrigues, R. & Blake, C. (2004), *When multi-hop peer to peer lookup matters*, february edn, IPTPS 04: Proceedings of the 3rd International Workshop on Peer to Peer Systems, San Diego, CA, USA.

Rodrigues, R. & Liskov, B. (2005), *High availability in dhts: Erasure coding vs. replication*, february edn, Peer-to-Peer Systems, IV 4th International Workshop IPTPS 2005, Ithaca, New York, USA.

Rowstron, A. & Druschel, P. (2001*a*), *Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems*, vol. 2218 edn, Lecture Notes in Computer Science.

Rowstron, A. I. T. & Druschel, P. (2001*b*), *Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility*, december edn, SOSP 01: Proceedings of the 8th ACM symposium on Operating Systems Principles.

Ruiz, O. H. & Znati, T. (2005), *Static Resiliency vs Churn-Resistance Capability of DHT-Protocols*, in Proceedings of International Society for Computers and their Applications: Parallel and Distributed Computing Systems.

ScaleNet, C. (2011), *The ScaleNet Project*, on july edn, The ScaleNet Project Site: www.scalenet.de.

Serbu, S., Kropf, P. & Felber, P. (2007), *Fault-Tolerant P2P Networks: How Dependable is Greedy Routing?*, Workshop on Dependable Application Support in Self-Organising Networks (DASSON).

Stoica, I., Morris, R., Nowell, D. L., Karger, D. R., Kaashoek, F. F., Dabek, F. & Balakrishnan, H. (2003), *Chord: a scalable peer to peer lookup protocol for internet applications*, vol 11, no 1, february edn, IEEE, ACM Trans. Netw.

Team, O. (2011), *Oversim: The Overlay Simulation Framework*, on july edn, www.oversim.org.

Utard, G. & Vernois, A. (2004), *Data durability in peer to peer storage systems*, 90 to 94, chicago, illinois, april edn, Proc. of IEEE, ACM CCGRID (GP2PC 2004).

van Renesse, R. (2004), *Efficient reliable internet storage*, october edn, WDDDM 04: Proceedings of the 2nd Workshop on Dependable Distributed Data Management, Glasgow, Scotland.

Wikipedia (2008), *Simpy*, www.en.wikipedia.org \ wiki \ SimPy.

Wikipedia (2010), *Python*, www.en.wikipedia.org \ wiki \ Python.

Wilson, C., Ballani, H., Karagiannis, T. & Rowstron, A. (2011), *Better never than late: Meeting deadlines in datacenter networks*, technical report msr-tr-2011-66 edn, Microsoft Research.

Wu, D., Tian, Y. & Ng, K.-W. (2009), *Analytical Study on Improving DHT Lookup Performance under Churn*, Department of Computer Science and Engineering, The Chinese University of Hong Kong Shatin, NT, Hong Kong.

Yang, Z., Tian, J. & Dai, Y. (2010), *Towards a More Accurate Availability Evaluation in Peer-to-Peer Storage Systems*, vol. 6 issue 3/4 edn, International Journal of High Performance Computing and Networking.

Yao, Z., Leonard, D., Wang, X., & Loguinov, D. (2006), *Modeling Heterogeneous User Churn and Local Resilience of Unstructured P2P Networks*, in Proceedings of International Conference Network Protocols.

Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D., & Kubiatowicz, J. D. (2003), *Tapestry: A global-scale overlay for rapid service deployment*, IEEE Journal on Selected Areas in Communications.

Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D., & Kubiatowicz, J. D. (2004), *Tapestry: A resilient global-scale overlay for service deployment*, vol. 22 edn, IEEE Journal on Selected Areas in Communications.

Appendix A

Appendix: OverSim parameter files

A.1 Default.ini

```
[General]
ned-path = ../../INET-OverSim-20101019/src;../src
# Use the following ned-path if you need ReaSE background traffic
#ned-path = ../../INET-OverSim-20101019/src;../src;../../ReaSE/src
tkenv-image-path = ../images


    # UdpOut does only work with SingleHostUnderlay!
# You have to select appropriate outDeviceType in SingleHost configuration
#scheduler-class = TunOutScheduler
#scheduler-class = UdpOutScheduler
# If a realworld connection is desired, debug-on-errors has to be disabled
#debug-on-errors=false
debug-on-errors = true
network = oversim.underlay.simpleunderlay.SimpleUnderlayNetwork


    # If an external app should be connected to the simulation, set app-port to
the appropriate TCP Port
# Has to be "0" if no external app is used
externalapp-app-port = 0
# If bigger than zero, accept only n simultaneous app connections
externalapp-connection-limit = 0
```

```
    # Change simtime scale (default is picosecond, which is not needed in most
overlay
# protocols and limits simulation time to  100 days)
# Nanoseconds are precise enough, and can run  300 years
simtime-scale=-9


    # — Application settings —


    # Here ** includes *.overlayTerminal, *.overlayBackboneRouter, *.singleHost


    # KBRTestApp settings
**.tier1*.kbrTestApp.kbrOneWayTest = true
**.tier1*.kbrTestApp.kbrRpcTest = false
**.tier1*.kbrTestApp.kbrLookupTest = false
**.tier1*.kbrTestApp.testMsgSize = 100B
**.tier1*.kbrTestApp.testMsgInterval = 60s
**.tier1*.kbrTestApp.msgHandleBufSize = 8
**.tier1*.kbrTestApp.lookupNodeIds = true
**.tier1*.kbrTestApp.failureLatency = 10s
**.tier1*.kbrTestApp.onlyLookupInoffensiveNodes = false


    #DHT settings
**.tier1*.dht.numReplica = 4
**.tier1*.dht.numGetRequests = 4
**.tier1*.dht.ratioIdentical = 0.5
**.tier1*.dht.secureMaintenance = false
**.tier1*.dht.invalidDataAttack = false
**.tier1*.dht.maintenanceAttack = false
**.tier1*.dht.numReplicaTeams = 3


    # DHTTestApp settings
**.tier2*.dhtTestApp.testInterval = 60s
**.tier2*.dhtTestApp.testTtl = 300
**.tier2*.dhtTestApp.p2pnsTraffic = false


    # ALMTest settings
**.tier*.almTest.messageLength = 100
**.tier*.almTest.joinGroups = true
**.tier*.almTest.sendMessages = true


    # P2PNS settings
```

```
**.tier2*.p2pns.twoStageResolution = false
**.tier2*.p2pns.keepaliveInterval = 10s
**.tier2*.p2pns.idCacheLifetime = 60s
**.tier2*.p2pns.registerName = ""


    # XmlRpcInterface settings
**.tier3*.xmlRpcInterface.limitAccess=false


    # generic app settings
**.tier*.*.debugOutput = true
**.tier*.*.activeNetwInitPhase = false


    # — Overlay settings —


    # Here ** includes *.overlayTerminal[], *.overlayBackboneRouter[], *overlay-
AccessRouter[]


    # Chord settings
**.overlay*.chord.joinRetry = 2
**.overlay*.chord.joinDelay = 10s
**.overlay*.chord.stabilizeRetry = 1
**.overlay*.chord.stabilizeDelay = 20s
**.overlay*.chord.fixfingersDelay = 120s
**.overlay*.chord.checkPredecessorDelay = 5s
**.overlay*.chord.routingType = "iterative"
**.overlay*.chord.successorListSize = 8
**.overlay*.chord.aggressiveJoinMode = true
**.overlay*.chord.extendedFingerTable = false
**.overlay*.chord.numFingerCandidates = 3
**.overlay*.chord.proximityRouting = false
**.overlay*.chord.memorizeFailedSuccessor = false
**.overlay*.chord.mergeOptimizationL1 = false
**.overlay*.chord.mergeOptimizationL2 = false
**.overlay*.chord.mergeOptimizationL3 = false
**.overlay*.chord.mergeOptimizationL4 = false


    # kademlia settings
**.overlay*.kademlia.lookupRedundantNodes = 8
**.overlay*.kademlia.lookupParallelPaths = 1
**.overlay*.kademlia.lookupParallelRpcs = 3
**.overlay*.kademlia.lookupMerge = true
```

```
**.overlay*.kademlia.routingType = "iterative"
**.overlay*.kademlia.secureMaintenance = false
**.overlay*.kademlia.minSiblingTableRefreshInterval = 1000s
**.overlay*.kademlia.minBucketRefreshInterval = 1000s
**.overlay*.kademlia.siblingPingInterval = 0s
**.overlay*.kademlia.maxStaleCount = 0
**.overlay*.kademlia.k = 8
**.overlay*.kademlia.s = 8
**.overlay*.kademlia.b = 1
**.overlay*.kademlia.exhaustiveRefresh = true
**.overlay*.kademlia.pingNewSiblings = false
**.overlay*.kademlia.enableReplacementCache = true
**.overlay*.kademlia.replacementCachePing = true
**.overlay*.kademlia.replacementCandidates = 8
**.overlay*.kademlia.siblingRefreshNodes = 0
**.overlay*.kademlia.bucketRefreshNodes = 0
**.overlay*.kademlia.newMaintenance = false


    # R/Kademlia
**.overlay*.kademlia.activePing = false
**.overlay*.kademlia.proximityRouting = false
**.overlay*.kademlia.proximityNeighborSelection = false
**.overlay*.kademlia.altRecMode = false


    # pastry settings
**.overlay*.pastry.bitsPerDigit = 4
**.overlay*.pastry.numberOfLeaves = 16
**.overlay*.pastry.numberOfNeighbors = 0
**.overlay*.pastry.joinTimeout = 20s
**.overlay*.pastry.readyWait = 5s
**.overlay*.pastry.secondStageWait = 2s
**.overlay*.pastry.repairTimeout = 60s
**.overlay*.pastry.enableNewLeafs = false
**.overlay*.pastry.optimizeLookup = false
**.overlay*.pastry.partialJoinPath = false
**.overlay*.pastry.useRegularNextHop = true
**.overlay*.pastry.alwaysSendUpdate = false
**.overlay*.pastry.useDiscovery = false
**.overlay*.pastry.pingBeforeSecondStage = true
**.overlay*.pastry.discoveryTimeoutAmount = 1s
**.overlay*.pastry.routingTableMaintenanceInterval = 0s
**.overlay*.pastry.sendStateAtLeafsetRepair = true
**.overlay*.pastry.overrideOldPastry = false
```

```
**.overlay*.pastry.overrideNewPastry = false
**.overlay*.pastry.routeMsgAcks = true
**.overlay*.pastry.routingType = "semi-recursive"
**.overlay*.pastry.minimalJoinState = false
**.overlay*.pastry.proximityNeighborSelection = true


      # NTree
**.overlay*.nTree.joinDelay = 0.1s
**.overlay*.nTree.pingInterval = 5s
**.overlay*.nTree.maxChildren = 25
**.overlay*.nTree.sendRpcResponseToLastHop = false


      # Generic settings
**.overlay*.*.nodeId = ""
**.overlay*.*.debugOutput = true
**.overlay*.*.hopCountMax = 50
**.overlay*.*.recNumRedundantNodes = 3
**.overlay*.*.collectPerHopDelay = false
SimpleUnderlayNetwork.*.overlay*.*.drawOverlayTopology = true
**.overlay*.*.drawOverlayTopology = false
**.overlay*.*.useCommonAPIforward = false
**.overlay*.*.routingType = "iterative" #"exhaustive-iterative semi-recursive
full-recursive source-routing-recursive"
**.overlay*.*.keyLength = 160
**.overlay*.*.joinOnApplicationRequest = false
**.overlay.*.localPort = 1024
**.overlay*.*.rejoinOnFailure = true
**.overlay*.*.sendRpcResponseToLastHop = true
**.overlay*.*.recordRoute = false
**.overlay*.*.measureAuthBlock = false
**.overlay*.*.dropFindNodeAttack = false
**.overlay*.*.isSiblingAttack = false
**.overlay*.*.invalidNodesAttack = false
**.overlay*.*.dropRouteMessageAttack = false
**.overlay*.*.restoreContext = false


      # SimpleMultiOverlayHost settings
**.numOverlayModulesPerNode = 10
**.overlay[0].*.localPort = 1024
**.overlay[1].*.localPort = 1025
**.overlay[2].*.localPort = 1026
**.overlay[3].*.localPort = 1027
**.overlay[4].*.localPort = 1028
```

```
**.overlay[5].*.localPort = 1029
**.overlay[6].*.localPort = 1030
**.overlay[7].*.localPort = 1031
**.overlay[8].*.localPort = 1032
**.overlay[9].*.localPort = 1033


    # general overlay lookup settings
**.overlay*.*.lookupRedundantNodes = 1
**.overlay*.*.lookupParallelPaths = 1
**.overlay*.*.lookupParallelRpcs = 1
**.overlay*.*.lookupVerifySiblings = false
**.overlay*.*.lookupMajoritySiblings = false
**.overlay*.*.lookupMerge = false
**.overlay*.*.lookupUseAllParallelResponses = false
**.overlay*.*.lookupStrictParallelRpcs = true
**.overlay*.*.lookupNewRpcOnEveryTimeout = false
**.overlay*.*.lookupNewRpcOnEveryResponse = false
**.overlay*.*.lookupFinishOnFirstUnchanged = false
**.overlay*.*.lookupVisitOnlyOnce = true
**.overlay*.*.lookupAcceptLateSiblings = true
**.overlay*.*.lookupFailedNodeRpcs = false
**.overlay*.*.routeMsgAcks = false


    # bootstrapList configuration
**.bootstrapList.debugOutput = true
**.bootstrapList.mergeOverlayPartitions = false
**.bootstrapList.maintainList = false


    # neighbor cache settings
**.neighborCache.enableNeighborCache = false
**.neighborCache.rttExpirationTime = 100s
**.neighborCache.maxSize = 400
**.neighborCache.rttHistory = 10
**.neighborCache.timeoutAccuracyLimit = 0.6
**.neighborCache.defaultQueryType = "exact"
**.neighborCache.defaultQueryTypeI = "available"
**.neighborCache.defaultQueryTypeQ = "exact"
**.neighborCache.doDiscovery = false
**.neighborCache.ncsType = "none" #"vivaldi","svivaldi","gnp","nps"
**.neighborCache.ncsSendBackOwnCoords = true


    # GNP settings
```

```
**.neighborCache.gnpDimensions = 2
**.neighborCache.gnpCoordCalcRuns = 50
**.neighborCache.gnpLandmarkTimeout = 2s


    # NPS settings
**.neighborCache.npsMaxLayer = 3


    # cryptoModule settings
SingleHostUnderlayNetwork.overlayTerminal[0].cryptoModule.keyFile = "key.bin"
**.cryptoModule.keyFile = ""


    # —- BaseRpc settings —-


    **.rpcUdpTimeout = 1.5s
**.rpcKeyTimeout = 10.0s
**.optimizeTimeouts = false
**.rpcExponentialBackoff = false


    # —- UnderlayConfigurator settings —-


    # UnderlayConfigurator module settings
*.underlayConfigurator.transitionTime = 0s
*.underlayConfigurator.measurementTime = -1s
*.underlayConfigurator.gracefulLeaveDelay = 15s
*.underlayConfigurator.gracefulLeaveProbability = 0.5
# disable churn for SingleHost networks
SingleHostUnderlayNetwork.underlayConfigurator.churnGeneratorTypes = ""
# any combination of "NoChurn", "LifetimeChurn", "ParetoChurn" and "Ran-
domChurn" separated by spaces
*.underlayConfigurator.churnGeneratorTypes = "oversim.common.NoChurn"


    # ChurnGenerator configuration
*.churnGenerator*.initPhaseCreationInterval = 1s
*.churnGenerator*.targetOverlayTerminalNum = 10
*.churnGenerator*.lifetimeMean = 10000.0s
*.churnGenerator*.deadtimeMean = 10000.0s
*.churnGenerator*.lifetimeDistName = "weibull"
*.churnGenerator*.lifetimeDistPar1 = 1.0
*.churnGenerator*.noChurnThreshold = 0s
```

```
        # RandomChurn (obsolete)
*.churnGenerator*.targetMobilityDelay = 300s
*.churnGenerator*.targetMobilityDelay2 = 20s
*.churnGenerator*.churnChangeInterval = 0s
*.churnGenerator*.creationProbability = 0.5
*.churnGenerator*.migrationProbability = 0.0
*.churnGenerator*.removalProbability = 0.5


        # use globalFunctions?
*.globalObserver.globalFunctions[*].functionType = ""
*.globalObserver.numGlobalFunctions = 0


        # global statistics
*.globalObserver.globalStatistics.outputMinMax = false
*.globalObserver.globalStatistics.outputStdDev = false
*.globalObserver.globalStatistics.globalStatTimerInterval = 0s
*.globalObserver.globalStatistics.measureNetwInitPhase = false
# GlobalNodeList settings
*.globalObserver.globalNodeList.maxNumberOfKeys = 100
*.globalObserver.globalNodeList.keyProbability = 0.1
*.globalObserver.globalNodeList.maliciousNodeProbability = 0.0
*.globalObserver.globalNodeList.maliciousNodeChange = false
*.globalObserver.globalNodeList.maliciousNodeChangeStartTime = 200s
*.globalObserver.globalNodeList.maliciousNodeChangeRate = 0.05
*.globalObserver.globalNodeList.maliciousNodeChangeInterval = 100s
*.globalObserver.globalNodeList.maliciousNodeChangeStartValue = 0
*.globalObserver.globalNodeList.maliciousNodeChangeStopValue = 0.5


        # GlobalObserver configuration
*.globalObserver.globalTraceManager.traceFile = ""
*.globalObserver.globalParameters.printStateToStdOut = false
*.globalObserver.globalParameters.topologyAdaptation = false


        # SimpleUnderlayNetwork configuration
SimpleUnderlayNetwork.overlayTerminal*.udp.constantDelay = 50ms
SimpleUnderlayNetwork.overlayTerminal*.tcp.constantDelay = 50ms
SimpleUnderlayNetwork.overlayTerminal*.udp.useCoordinateBasedDelay = true
SimpleUnderlayNetwork.overlayTerminal*.tcp.useCoordinateBasedDelay = true
SimpleUnderlayNetwork.overlayTerminal*.udp.jitter = 0.1
SimpleUnderlayNetwork.overlayTerminal*.tcp.jitter = 0.1
SimpleUnderlayNetwork.underlayConfigurator.terminalTypes = "over-
sim.underlay.simpleunderlay.SimpleOverlayHost"
```

SimpleUnderlayNetwork.underlayConfigurator.fieldSize = 150
SimpleUnderlayNetwork.underlayConfigurator.sendQueueLength = 1MB
SimpleUnderlayNetwork.underlayConfigurator.fixedNodePositions = false

# SingleHostUnderlay configuration
SingleHostUnderlayNetwork.underlayConfigurator.terminalTypes = "dummy"
SingleHostUnderlayNetwork.underlayConfigurator.nodeIP = ""
SingleHostUnderlayNetwork.underlayConfigurator.nodeInterface = ""
SingleHostUnderlayNetwork.underlayConfigurator.stunServer = ""
SingleHostUnderlayNetwork.underlayConfigurator.bootstrapIP = ""
SingleHostUnderlayNetwork.underlayConfigurator.bootstrapPort = 1024
SingleHostUnderlayNetwork.zeroconfConnector.enableZeroconf = false

InetUnderlayNetwork configuration
InetUnderlayNetwork.outRouter*.outDeviceType = "oversim.underlay.singlehostunderlay.TunOutDevice"
**.mtu = 65000
**.parser = "oversim.common.GenericPacketParser"
**.appParser = "oversim.common.GenericPacketParser"
**.gatewayIP = ""

# InetUnderlay IPv4 and IPv6 backbone configuration
InetUnderlayNetwork.underlayConfigurator.terminalTypes = "oversim.underlay.inetunderlay.InetOverlayHost"
InetUnderlayNetwork6.underlayConfigurator.terminalTypes = "oversim.underlay.inetunderlay.ipv6.InetOverlayHost6"

InetUnderlayNetwork*.churnGenerator*.channelTypes = "" # not used in InetUnderlay
InetUnderlayNetwork*.churnGenerator*.channelTypesRx = "" # not used in InetUnderlay
InetUnderlayNetwork*.backboneRouterNum = 1
InetUnderlayNetwork*.overlayBackboneRouterNum = 0
InetUnderlayNetwork*.accessRouterNum = 2
InetUnderlayNetwork*.overlayAccessRouterNum = 0
InetUnderlayNetwork*.connectivity = 0.8
InetUnderlayNetwork*.underlayConfigurator.startIPv4 = "1.1.0.1"
InetUnderlayNetwork*.underlayConfigurator.startIPv6 = "1::"
InetUnderlayNetwork*.outRouterNum = 0
InetUnderlayNetwork6.*Router[*].routingTable6.routingTableFile = xmldoc("dummy.xml")
InetUnderlayNetwork6.*overlayTerminal[*].routingTable6.routingTableFile = xml-

doc(”dummy.xml”)

TCP parameters
**.tcp.sendQueueClass = ”TCPMsgBasedSendQueue”
**.tcp.receiveQueueClass = ”TCPMsgBasedRcvQueue”

default overlay and application
Here ** includes *.globalObserver.globalTraceManager and *.churnGenerator*
**.overlayType = ”oversim.overlay.chord.ChordModules”
**.tier1Type = ”oversim.applications.kbrtestapp.KBRTestAppModules”
**.tier2Type = ”oversim.common.TierDummy”
**.tier3Type = ”oversim.common.TierDummy”
**.numTiers = 1

A.2 Omnetpp.ini

[Config ChordChurn]
repeat = 5
*.underlayConfigurator.churnGeneratorTypes = ”oversim.common.LifetimeChurn”
**.overlayType = ”oversim.overlay.chord.ChordModules”
**.tier1Type = ”oversim.applications.kbrtestapp.KBRTestAppModules”
**.targetOverlayTerminalNum = 100
**.initPhaseCreationInterval = 0.1s
**.measurementTime = 500s
**.transitionTime = 100s
**.overlay.chord.stabilizeDelay = $stab = 5,60s$
$**.lifetimeMean$ =lifetime=1000, 2000, 10000s

[Config Chord]
description = Chord (iterative, SimpleUnderlayNetwork)
*.underlayConfigurator.churnGeneratorTypes = ”oversim.common.LifetimeChurn”
**.overlayType = ”oversim.overlay.chord.ChordModules”
**.tier1Type = ”oversim.applications.kbrtestapp.KBRTestAppModules”

[Config ChordInet]
description = Chord (iterative, InetUnderlayNetwork)
network = oversim.underlay.inetunderlay.InetUnderlayNetwork
*.underlayConfigurator.churnGeneratorTypes = ”oversim.common.LifetimeChurn”
**.overlayType = ”oversim.overlay.chord.ChordModules”
**.tier1Type = ”oversim.applications.kbrtestapp.KBRTestAppModules”

InetUnderlayNetwork.backboneRouterNum = 1
InetUnderlayNetwork.overlayAccessRouterNum = 1
InetUnderlayNetwork.accessRouterNum = 1

[Config ChordInet6]
description = Chord (iterative, InetUnderlayNetwork6)
network = oversim.underlay.inetunderlay.InetUnderlayNetwork6
*.underlayConfigurator.churnGeneratorTypes = "oversim.common.LifetimeChurn"
**.overlayType = "oversim.overlay.chord.ChordModules"
**.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
InetUnderlayNetwork.backboneRouterNum = 1
InetUnderlayNetwork.overlayAccessRouterNum = 1
InetUnderlayNetwork.accessRouterNum = 1

[Config ChordSimpleSemi]
description = Chord (semi-recursive, SimpleUnderlayNetwork)
*.underlayConfigurator.churnGeneratorTypes = "oversim.common.LifetimeChurn"
**.overlayType = "oversim.overlay.chord.ChordModules"
**.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
**.routingType = "semi-recursive"

[Config ChordFastStab]
description = Chord (semi-recursive, SimpleUnderlayNetwork, faster stabilize)
*.underlayConfigurator.churnGeneratorTypes = "oversim.common.LifetimeChurn"
**.overlayType = "oversim.overlay.chord.ChordModules"
**.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
**.routingType = "semi-recursive"
**.overlay.chord.stabilizeDelay = 5s
**.overlay.chord.fixfingersDelay = 60s

[Config ChordLarge]
description = Chord (semi-recursive, SimpleUnderlayNetwork, no churn, large-scale test -¿ run without GUI)
**.measurementTime = 500s
**.transitionTime = 100s
**.overlayType = "oversim.overlay.chord.ChordModules"
**.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
**.routingType = "semi-recursive"
**.churnGeneratorTypes = "oversim.common.NoChurn"
**.targetOverlayTerminalNum = 10000
**.initPhaseCreationInterval = 0.1s
**.debugOutput = false

[Config Kademlia]
description = Kademlia (SimpleUnderlayNetwork)
*.underlayConfigurator.churnGeneratorTypes = "oversim.common.LifetimeChurn"
**.overlayType = "oversim.overlay.kademlia.KademliaModules"
**.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"

[Config KademliaLarge]
description = Kademlia (SimpleUnderlayNetwork, no churn, large-scale test -¿ run without GUI)
**.measurementTime = 500s
**.transitionTime = 100s
**.overlayType = "oversim.overlay.kademlia.KademliaModules"
**.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
**.targetOverlayTerminalNum = 1000
**.initPhaseCreationInterval = 0.1s
**.overlay.kademlia.lookupRedundantNodes = 16
**.overlay.kademlia.s = 8
**.overlay.kademlia.k = 16
**.overlay.kademlia.lookupMerge = true
**.overlay.kademlia.lookupParallelPaths = 1
**.overlay.kademlia.lookupParallelRpcs = 1

[Config Pastry]
description = Pastry (SimpleUnderlayNetwork)
*.underlayConfigurator.churnGeneratorTypes = "oversim.common.LifetimeChurn"
**.overlayType = "oversim.overlay.pastry.PastryModules"
**.enableNewLeafs = false
**.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
**.targetOverlayTerminalNum = 30
**.measureNetwInitPhase = false
**.useCommonAPIforward = true
**.neighborCache.enableNeighborCache = true

[Config PastryLarge]
description = Pastry (SimpleUnderlayNetwork, no churn, large-scale test -¿ run without GUI)
**.churnGeneratorTypes = "oversim.common.NoChurn"
**.overlayType = "oversim.overlay.pastry.PastryModules"
**.enableNewLeafs = false
**.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
**.targetOverlayTerminalNum = 1000

**.neighborCache.enableNeighborCache = true

[Config ChordDht]
description = Chord DHT (SimpleUnderlayNetwork)
*.underlayConfigurator.churnGeneratorTypes = "oversim.common.LifetimeChurn"
**.lifetimeMean = 10000s
**.measurementTime = 1000s
**.transitionTime = 100s
**.overlayType = "oversim.overlay.chord.ChordModules"
**.numTiers = 2
**.tier1Type = "oversim.applications.dht.DHTModules"
**.tier2Type = "oversim.tier2.dhttestapp.DHTTestAppModules"
**.globalObserver.globalFunctions[0].functionType = "oversim.tier2.dhttestapp.GlobalDhtTestMap"
**.globalObserver.numGlobalFunctions = 1
**.targetOverlayTerminalNum = 100
**.initPhaseCreationInterval = 0.1s
**.debugOutput = false
**.drawOverlayTopology = true
**.tier1*.dht.numReplica = 4

[Config ChordDhtTrace]
description = Chord/DHT trace test (SimpleUnderlayNetwork)
**.overlayType = "oversim.overlay.chord.ChordModules"
**.numTiers = 2
**.tier1Type = "oversim.applications.dht.DHTModules"
**.tier2Type = "oversim.tier2.dhttestapp.DHTTestAppModules"
**.globalObserver.globalFunctions[0].functionType = "oversim.tier2.dhttestapp.GlobalDhtTestMap"
**.globalObserver.numGlobalFunctions = 1
**.tier2.dhtTestApp.testInterval = 0s
**.churnGeneratorTypes = "oversim.common.TraceChurn"
**.traceFile = "dht.trace"

[Config ChurnVisualization]
description = "Chord Churn Visualization"
**.overlayType = "oversim.overlay.chord.ChordModules"
**.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
*.underlayConfigurator.churnGeneratorTypes = "oversim.common.RandomChurn oversim.common.NoChurn"
**.overlay*.chord.routingType = "semi-recursive"
*.churnGenerator[0].targetMobilityDelay=300s
*.churnGenerator[0].targetMobilityDelay2=7s

```
*.churnGenerator[0].churnChangeInterval=7000s
**.globalObserver.globalStatistics.globalStatTimerInterval = 120s
*.churnGenerator[1].targetOverlayTerminalNum = 1


    [Config MultiOverlay]
description = Chord with MultiOverlayHosts (recursive, SimpleUnderlayNetwork)
**.churnGeneratorTypes = "oversim.common.RandomChurn oversim.common.RandomChurn"
**.terminalTypes = "oversim.underlay.simpleunderlay.SimpleOverlayHost oversim.underlay.simpleunderlay.SimpleMultiOverlayHost"
**.numOverlayModulesPerNode = 5
**.routingType = "semi-recursive"
**.churnGenerator[0].targetOverlayTerminalNum = 100
**-0[*].overlayType = "oversim.overlay.chord.ChordModules"
**-0[*].tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
**.churnGenerator[1].targetOverlayTerminalNum = 5
**-1[*].overlayType = "oversim.overlay.chord.ChordModules"
**-1[*].tier1Type = "oversim.common.TierDummy"


    [Config MyConfig]
description = MyApplication / MyOverlay (Example from the OverSim website)
**.overlayType = "oversim.overlay.myoverlay.MyOverlayModules"
**.tier1Type = "oversim.applications.myapplication.MyApplicationModules"
**.targetOverlayTerminalNum = 10
**.enableDrops = false
**.dropChance = 0
**.sendPeriod = 1s
**.numToSend = 1
**.largestKey = 10


    [Config TCPExampleApp]
description = TCPExampleApp (InetUnderlayNetwork)
*.underlayConfigurator.churnGeneratorTypes = "oversim.common.NoChurn"
**.overlayType = "oversim.applications.i3.OverlayDummyModules"
**.tier1Type = "oversim.applications.tcpexampleapp.TCPExampleAppModules"
**.targetOverlayTerminalNum = 2


    include ./default.ini
```

Appendix B

Appendix: Moditions to OverSim code

OverSim file: DHTTestApp.cc Function: handleTimerEvent(cMessage* msg) Added code: MyClass::addLookup(key, simulation.simTime());

OverSim file: DHT.cc Function: handleGetResponse(DHTGetResponse* dhtMsg) Added code: MyClass::removeLookup(key); Function: finishApp() Added code: MyClass::addKeys(dataStorage-¿getSize());

OverSim file: Kademlia.cc Function: findNode() Added code: MyClass::isValidLookup(key);

OverSim file: Chord.cc Function: findNode() Added code: MyClass::isValidLookup(key);

OverSim file: GlobalStatistics.cc Function: doFinish() Added code: MyClass::print(); MyClass::reset();

Appendix C

Appendix: MyClass

C.1 MyClass.h

```
// MyClass.h
\#include <string>
using std::string;
\#include<OverlayKey.h>
\#include<NodeHandle.h>
\#include <omnetpp.h>
\#ifndef MYCLASS_H
\#define MYCLASS_H
class MyClass {
public:
MyClass();
static void print();
static std::map<OverlayKey, int> lookupHops;
static std::map<OverlayKey, double> luDelay;
static void addMaintenance(int join, int notify, int stabilize,
int newSuc, int fixFing);
static void addHop(int hopCount);
static void addHop2(int hopCount);
static void addKeys(int keys);
static void reset();
static void printLookups();
static void addLookup(OverlayKey key, double simtime);
static void addDelay(double delay);
static void removeLookup(OverlayKey key);
static bool isValidLookup(OverlayKey key);
```

```
static void addHandle(IPvXAddress ip, NodeHandle handle);
static NodeHandle getHandle(IPvXAddress ip);
static double hops;
static bool iterative;
static double finished_lookups;
static double removed_lookups;
static double lookup_calls;
static int remaining_lookups ;
static double numberOfNodes;
static double numberOfKeys;
static double maxKeys;
static double minKeys;
static double stabilizeInterval;
static double fixFingersInterval;
static double time;
static string dht;
static double lookupFreq;
static double churnRate;
static int terminalsAdded;
static int terminalsRemoved;
static int networkSize;
static int routingTableSize;
static string routingModel;
static int aid;
// \#\#\#\# KADEMLIA \#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#
static int k;
static int s;
static int b;
static double hopsPerLookup;
static double lookupMessageOverhead;
static double lookupDelay;
static double keysPerNode;
};
\#endif
```

C.2 MyClass.cc

```
// MyClass.cc
\#include <iostream>
using std::cout;
using std::cin;
using std::ios;
```

```
using std::cerr;
using std::endl;
using namespace std;
\#include <fstream>
using std::ofstream;
\#include <cstdlib>
\#include <string>
using std::string;
\#include "MyClass.h"
typedef std::map<OverlayKey, int> lookupHops;
typedef std::map<OverlayKey, double> luDelays;
lookupHops lookups;
luDelays delays;
double MyClass::hops = 0;
double MyClass::finished_lookups = 0;
double MyClass::time = 0;
bool MyClass::iterative = true;
double MyClass::removed_lookups = 0; // used for calculating lookups
that take too long and are removed
double MyClass::lookup_calls = 0;
int MyClass::remaining_lookups = 0; // lookups that are in progress
when simulation stops
double MyClass::numberOfNodes = 0;
double MyClass::numberOfKeys = 0;
double MyClass::maxKeys = 0;
double MyClass::minKeys = 1000;
string MyClass::dht = "empty";
double MyClass::lookupFreq = 0;
double MyClass::churnRate = 0;
int MyClass::terminalsAdded = 0;
int MyClass::terminalsRemoved = 0;
int MyClass::networkSize = 0;
double MyClass::stabilizeInterval = 0;
double MyClass::fixFingersInterval = 0;
int MyClass::routingTableSize = 0;
string MyClass::routingModel = "empty";
int MyClass::aid = 0;
// \#\#\# KADEMLIA \#\#\#\#\#\#\#\#
int MyClass::k = 0;
int MyClass::s = 0;
int MyClass::b = 0;
double MyClass::hopsPerLookup = 0;
double MyClass::lookupMessageOverhead = 0;
double MyClass::lookupDelay = 0;
```

```
double MyClass::keysPerNode = 0;
MyClass::MyClass() {}
// calculates the hopcount
void MyClass::addHop(int hopCount)
{
hops = hops + hopCount;
finished_lookups++;
hopsPerLookup = hops / finished_lookups;
}
// calculates the number of keys in the network
// and the max/min number of keys in one node
void MyClass::addKeys(int keys)
{
numberOfNodes++;
numberOfKeys += keys;
keysPerNode = numberOfKeys / numberOfNodes;
if (keys < minKeys){
minKeys = keys;
}
if (keys > maxKeys){
maxKeys = keys;
}
}
// inserts lookups and simTimes to map containers
void MyClass::addLookup(OverlayKey key, double simtime)
{
lookups.insert(make_pair(key, 0));
delays.insert(make_pair(key, simtime));
lookup_calls++;
}
void MyClass::addDelay(double delay)
{
lookupDelay += delay;
}
// removes lookups from the map container
void MyClass::removeLookup(OverlayKey key)
{
bool skip = false;
// checks that the key exists in the lookups map
lookupHops::iterator it;
it=lookups.find(key);
if (it == lookups.end())
return;
// if delay > 5 seconds the lookup is removed from the map
```

```
double delaytmp = (simulation.simTime() - delays[key]);
if (delaytmp > 5) {
cout << "DELAY " << delaytmp << endl;
delays.erase(key);
lookups.erase(key);
skip = true;
++removed_lookups;
}
// removes all the lookups currently having a delay > 5 seconds from
    the
map
luDelays::iterator iter;
for (iter = delays.begin(); iter != delays.end(); iter++) {
double delaytmp =(simulation.simTime()-(iter->second));
if (delaytmp > 5) {
OverlayKey temp = iter->first;
delays.erase(temp);
lookups.erase(temp);
++removed_lookups;
}
}
if (skip == false) {
// calculates the number of hops in this
lookup
int hoptmp = (lookups[key] -1)*2;
addHop(hoptmp);
lookups.erase(key);
// calculates the delay of this lookup
double delaytmp = (simulation.simTime() - delays[key]);
addDelay(delaytmp);
delays.erase(key);
}
}
}
// checks that there is a ongoing lookup for a specific key
bool MyClass::isValidLookup(OverlayKey key)
{
map<OverlayKey, int, double>::iterator iter;
bool tmp = false;
for (iter = lookups.begin(); iter != lookups.end(); iter++) {
if (iter->first.compareTo(key) == 0 && time !=
simulation.simTime() ){
tmp = true;
iter->second++;
```

```
}
}
// simulation time stored to allow only one call of this function
at a time
time = simulation.simTime();
return tmp;
}
// resets the initial values
void MyClass::reset()
{
MyClass::hops = 0;
MyClass::finished_lookups = 0;
MyClass::removed_lookups = 0;
MyClass::lookup_calls = 0;
MyClass::remaining_lookups = 0;
MyClass::numberOfNodes = 0;
MyClass::numberOfKeys = 0;
MyClass::maxKeys = 0;
MyClass::minKeys = 1000;
MyClass::stabilizeInterval = 0;
MyClass::fixFingersInterval = 0;
MyClass::dht = "empty";
MyClass::lookupFreq = 0;
MyClass::churnRate = 0;
MyClass::networkSize = 0;
MyClass::routingTableSize = 0;
MyClass::routingModel = "empty";
MyClass::aid = 0;
MyClass::iterative = true;
MyClass::k = 0;
MyClass::s = 0;
MyClass::b = 0;
MyClass::hopsPerLookup = 0;
MyClass::lookupMessageOverhead = 0;
MyClass::lookupDelay = 0;
MyClass::keysPerNode = 0;
}
// prints the results to results.dat
void MyClass::print()
{
ofstream outResultFile( "results.dat", ios::app );
outResultFile << "DHTused " << dht << " variable" << " variable" <<
    ' ' <<
"RoutingModel " << routingModel << ' ' << "LookupCalls "
```

```
<< lookup_calls << ' ' <<
"FinishedLookups: " << finished_lookups << ' '
<< "removedLookups " << removed_lookups << ' ' << "SuccessRate " <<
finished_lookups/(lookup_calls - remaining_lookups) << ' ' << "Added
    " <<
terminalsAdded
<< ' ' << "Removed " << terminalsRemoved << ' ' <<
    "HopsPerFinishedLookup: "
<< hopsPerLookup << ' ' << "Lookup_delay: " <<
    lookupDelay/finished_lookups <<
' ' << "Network_Size: " << networkSize << ' ' <<
    "Stabilize_interval: " <<
stabilizeInterval << ' ' << "FixFingers_interval " <<
    fixFingersInterval << '
' << "Hops: " << hops << ' ' << "Keys_total " << numberOfKeys << ' '
    <<
"Max_keys " << maxKeys << ' ' << "Min_keys " <<
minKeys << ' ' << "Keys_per_node: " << keysPerNode << endl;
}
```

Appendix D

Appendix: Free_DHT Data Storage

D.1 Free_DHTDataStorage.h

```
\#ifndef \_\_Free\_DHTDataStorage\_H\_
\#define \_\_Free\_DHTDataStorage\_H\_

\#include <set>
\#include <vector>
\#include <map>
\#include <sstream>

\#include <omnetpp.h>

\#include <NodeHandle.h>
\#include <InitStages.h>
\#include <BinaryValue.h>
\#include <leafset.h>
\#include <CommonMessages\_m.h>

typedef std::map<BinaryValue, leafset> replica;

struct DhtDataEntry
{
    BinaryValue value;
    uint32\_t Peer;
    uint32\_t blockID;
```

```
    cMessage* messag;
    bool is\_modifiable;
    NodeHandle sourceNode;
    bool rootPeer; //is this node rootPeer for this key ?
    friend std::ostream& operator<<(std::ostream& Stream, const
       DhtDataEntry entry);
    replica replicaSet;
};

typedef std::vector<std::pair<OverlayKey, DhtDataEntry> >
   DhtDataVector;
typedef std::vector<DhtDumpEntry> DhtDumpVector;
typedef std::multimap<OverlayKey, DhtDataEntry> DhtrootVector;

class Free\_DHTDataStorage : public cSimpleModule
{
  public:

    virtual int numInitStages() const
    {
        return MAX\_STAGE\_APP + 1;
    }
    virtual void initialize(int stage);
    virtual void handleMessage(cMessage* msg);

       virtual uint32\_t getSize();
   DhtDataEntry* getDataEntry(const OverlayKey& key,
                              uint32\_t Peer, uint32\_t blockID);
    virtual DhtDataVector* getDataVector(const OverlayKey& key,
                                         uint32\_t Peer = 0,
                                         uint32\_t blockID = 0);
        virtual const NodeHandle& getSourceNode(const OverlayKey& key,
                                            uint32\_t Peer, uint32\_t
                                               blockID);
    virtual const bool isModifiable(const OverlayKey& key,
                                    uint32\_t Peer, uint32\_t blockID);
     virtual const DhtrootVector::iterator begin();
    /**
     * Returns an iterator to the end of the map
     *
     * @return An iterator
     */
    virtual const DhtrootVector::iterator end();
    /**
```

```
 * Store a new data item in the map
 *
 * @param key The key of the data item to be stored
 * @param Peer The Peer of the data item
 * @param blockID A random integer to identify multiple items
    with same key and Peer
 * @param value The value of the data item to be stored
 * @param messag The self-message sent for the p expiration
 * @param is\_modifiable Flag that tell if the data can be
    change by anyone, or just by the sourceNode
 * @param sourceNode Node which asked to store the value
 * @param rootPeer
 */
virtual DhtDataEntry* addData(const OverlayKey& key, uint32\_t
   Peer,
                              uint32\_t blockID,
                              BinaryValue value, cMessage* messag,
                              bool is\_modifiable=true,
                              NodeHandle
                                  sourceNode=NodeHandle::UNSPECIFIED\_NODE,
                              bool rootPeer=true);

/**
 * Removes a certain data item from the map
 *
 * @param key The key of the data item to be removed
 * @param Peer The Peer of the data item
 * @param blockID A random integer to identify multiple items
    with same key and Peer
 *
 */
virtual void removeData(const OverlayKey& key, uint32\_t Peer,
   uint32\_t blockID);
void display();
/**
 * Dump filtered local data records into a vector
 *
 * @param key The key of the data items to dump
 * @param Peer The Peer of the data items to dump
 * @param blockID The id of the data items to dump
 *
 * @return the vector containing all matching data items
 */
```

```
    DhtDumpVector* dumpDht(const OverlayKey& key =
        OverlayKey::UNSPECIFIED\_KEY,
                           uint32\_t Peer = 0, uint32\_t blockID = 0);
  protected:
    DhtrootVector rootVector; /**< internal representation of the
        data storage */

    /**
     * Displays the current number of successors in the list
     */
    void updateDisplayString();

    /**
     * Displays the first 4 successor nodes as tooltip.
     */
    void updateTooltip();
};
\#endif
```

D.2 Free_DHTDataStorage.cc

```
\#include <omnetpp.h>
\#include <hashWatch.h>
\#include "Free\_DHTDataStorage.h"
Define\_Module(Free\_DHTDataStorage);
using namespace std;

std::ostream& operator<<(std::ostream& os, const DhtDataEntry entry)
{
    os << "Value: " << key.value
       << " Peer: " << key.Peer
       << " blockID: " << key.blockID
       << " Endtime: " << key.messag->getArrivalTime()
       << " rootPeer: " << key.rootPeer
       << " SourceNode: " << key.sourceNode;

    if (key.replicaSet.size()) {
        os << " replicaSet:";

        for (replica::const\_iterator it = key.replicaSet.begin();
             it != key.replicaSet.end(); it++) {
```

```
            os << " " << it->first << " (" << it->second.size() <<
                ")";
        }
    }
    return os;
}


void Free\_DHTDataStorage::initialize(int stage)
{
    if (stage != MIN\_STAGE\_APP)
        return;

    WATCH\_MULTIMAP(rootVector);
}

void Free\_DHTDataStorage::handleMessage(cMessage* msg)
{
    error("This module doesn't handle messages!");
}

voblockID Free\_DHTDataStorage::clear()
{
    map<OverlayKey, DhtDataEntry>::iterator iter;

    for( iter = rootVector.begin(); iter != rootVector.end(); iter++
        ) {
        cancelAndDelete(iter->second.messag);
    }

    rootVector.clear();
}


uint32\_t Free\_DHTDataStorage::getSize()
{
    return rootVector.size();
}

DhtDataEntry* Free\_DHTDataStorage::getDataEntry(const OverlayKey&
    key,
                                        uint32\_t Peer, uint32\_t
                                            blockID)
{
```

```
    pair<DhtrootVector::iterator, DhtrootVector::iterator> pos =
        rootVector.equal\_range(key);

    while (pos.first != pos.second) {
        if ((pos.first->second.Peer == Peer) &&
                (pos.first->second.blockID == blockID)) {
            return &pos.first->second;
        }
        ++pos.first;
    }

    return NULL;
}



DhtDataVector* Free\_DHTDataStorage::getDataVector(const OverlayKey&
    key,
                                            uint32\_t Peer, uint32\_t
                                                blockID)
{
    DhtDataVector* vect = new DhtDataVector();
    DhtDataEntry key;

    pair<DhtrootVector::iterator, DhtrootVector::iterator> pos =
        rootVector.equal\_range(key);

    while (pos.first != pos.second) {
        key = pos.first->second;
        vect->push\_back(make\_pair(key, key));
        ++pos.first;
    }

    return vect;
}


const NodeHandle& Free\_DHTDataStorage::getSourceNode(const
    OverlayKey& key,
                                              uint32\_t Peer, uint32\_t
                                                  blockID)
{
    DhtDataEntry* key = getDataEntry(key, Peer, blockID);
```

```
    if (key == NULL)
        return NodeHandle::UNSPECIFIED\_NODE;
    else
        return key->sourceNode;
}

const bool Free\_DHTDataStorage::isModifiable(const OverlayKey& key,
                                    uint32\_t Peer, uint32\_t
                                        blockID)
{
    DhtDataEntry* key = getDataEntry(key, Peer, blockID);

    if (key == NULL)
        return true;
    else
        return key->is\_modifiable;
}


const DhtrootVector::iterator Free\_DHTDataStorage::begin()
{
    return rootVector.begin();
}

const DhtrootVector::iterator Free\_DHTDataStorage::end()
{
    return rootVector.end();
}

DhtDataEntry* Free\_DHTDataStorage::addData(const OverlayKey& key,
    uint32\_t Peer,
                                    uint32\_t blockID,
                                    BinaryValue value, cMessage*
                                        messag,
                                    bool is\_modifiable, NodeHandle
                                        sourceNode,
                                    bool rootPeer)
{
    DhtDataEntry key;
    key.Peer = Peer;
    key.blockID = blockID;
    key.value = value;
    key.messag = messag;
    key.sourceNode = sourceNode;
```

```
    key.is\_modifiable = is\_modifiable;
    key.rootPeer = rootPeer;

    if ((Peer == 0) || (blockID == 0)) {
        throw cRuntimeError("Free\_DHTDataStorage::addData(): "
                            "Not allowed to add data with Peer = 0 or
                               blockID = 0!");
    }

    pair<DhtrootVector::iterator, DhtrootVector::iterator> pos =
        rootVector.equal\_range(key);

    // insert new record in sorted multimap (order: key, Peer,
        blockID)
    while ((pos.first != pos.second) && (pos.first->second.Peer <
        Peer)) {
        ++pos.first;
    }

    while ((pos.first != pos.second) && (pos.first->second.Peer ==
        Peer)
            && (pos.first->second.blockID < blockID)) {
        ++pos.first;
    }

    return &(rootVector.insert(pos.first, make\_pair(key,
        key))->second);
}

void Free\_DHTDataStorage::removeData(const OverlayKey& key,
    uint32\_t Peer,
                                uint32\_t blockID)
{
    pair<DhtrootVector::iterator, DhtrootVector::iterator> pos =
        rootVector.equal\_range(key);

    while (pos.first != pos.second) {

        if (((Peer == 0) || (pos.first->second.Peer == Peer)) &&
                ((blockID == 0) || (pos.first->second.blockID ==
                    blockID))) {
            cancelAndDelete(pos.first->second.messag);
            rootVector.erase(pos.first++);
        } else {
```

```
            ++pos.first;
        }
    }
}

DhtDumpVector* Free\_DHTDataStorage::dumpDht(const OverlayKey& key,
    uint32\_t Peer,
                                    uint32\_t blockID)
{
    DhtDumpVector* vect = new DhtDumpVector();
    DhtDumpEntry key;

    DhtrootVector::iterator iter, end;

    if (key.isUnspecified()) {
        iter = rootVector.begin();
        end = rootVector.end();
    } else {
        iter = rootVector.lower\_bound(key);
        end = rootVector.upper\_bound(key);
    }

    for (; iter != end; iter++) {
        if (((Peer == 0) || (iter->second.Peer == Peer)) &&
                ((blockID == 0) || (iter->second.blockID ==
                    blockID))) {

            key.setKey(iter->first);
            key.setPeer(iter->second.Peer);
            key.setblockID(iter->second.blockID);
            key.setValue(iter->second.value);
            key.setp((int)SIMTIME\_DBL(
                        iter->second.messag->getArrivalTime() -
                            simTime()));
            key.setOwnerNode(iter->second.sourceNode);
            key.setIs\_modifiable(iter->second.is\_modifiable);
            key.setrootPeer(iter->second.rootPeer);
            vect->push\_back(key);
        }
    }

    return vect;
}
```

```
// TODO: not used ?
void Free\_DHTDataStorage::updateDisplayString()
{
    if (ev.isGUI()) {
        char buf[80];

        if (rootVector.size() == 1) {
            sprintf(buf, "1 data item");
        } else {
            sprintf(buf, "%zi data items", rootVector.size());
        }

        getDisplayString().setTagArg("t", 0, buf);
        getDisplayString().setTagArg("t", 2, "blue");
    }

}

// TODO: not used ?
void Free\_DHTDataStorage::updateTooltip()
{
    if (ev.isGUI()) {
        std::stringstream str;

        for (DhtrootVector::iterator it = rootVector.begin();
             it != rootVector.end(); it++) {
            str << it->second.value;
        }

        str << endl;

        char buf[1024];
        sprintf(buf, "%s", str.str().c\_str());
        getDisplayString().setTagArg("tt", 0, buf);
    }
}

// TODO: not used ?
void Free\_DHTDataStorage::display()
{
    cout << "Content of Free\_DHTDataStorage:" << endl;
    for (DhtrootVector::iterator it = rootVector.begin();
         it != rootVector.end(); it++) {
```

```
        cout << "Key: " << it->first << " Peer: " << it->second.Peer
             << " blockID: " << it->second.blockID << " Value: "
             << it->second.value << "End-time: "
             << it->second.messag->getArrivalTime() << endl;
    }
}
```