ROYAL INSTITUTE
OF TECHNOLOGY

MyriadStore: A Peer-to-Peer Backup System

Birgir Stefansson and Antonios Thodis

Master of Science Thesis
Stockholm, Sweden, June 2006

Examiner: Seif Haridi

Supervisor: Ali Ghodsi

Abstract

Traditional backup methods are error-prone, cumbersome, and expensive. Distributed backup applications have emerged as promising tools able to avoid these disadvantages, by exploiting unused disk space of remote computers. In this thesis we propose MyriadStore, a peer-to-peer backup system in which nodes store their backed up data by using other nodes' storage capacities.

MyriadStore makes use of a trading scheme that ensures that a user has as much available storage space in the system as he/she contributes to others. In order to verify that nodes are storing data they have been entrusted with, a challenge mechanism is in place. A node can challenge the nodes that are responsible for holding its data by requesting from them parts of the data. In case they fail to respond correctly to these challenges, they are punished according to a reputation-based punishment model. With this model, the nodes that fail challenges are punished according to the reputation they have in the system.

MyriadStore minimizes bandwidth requirements and migration costs by treating separately the storage of the system's meta-data and the storage of the backed up data. Meta-data is stored on a distributed hash table (DHT) which is provided by the DKS system, while data is stored outside the DHT. This approach also offers flexibility on the placement of the backed up data, a property that facilitates the deployment of the trading scheme.

Acknowledgments

This project was carried out at the Swedish Institute of Computer Science, SICS, in co-operation with Acreo between October 2005 and June 2006. This project was financed by IRECO.

We thank our examiner, professor Seif Haridi, for providing us with the opportunity to work on this project. We thank our supervisor, Ali Ghodsi, for his unending support and guidance that has helped us during the development of this project. Lastly, we would like to thank the following people for their support, suggestions and constructive comments: Cosmin Arad, Per Brand, Sameh El-Ansary, Fredrik Holmgren, Janusz Launberg and Roland Yap.

Contents

1 Introduction **1**

1.1 Traditional Backup Methods 2

1.2 Peer-to-Peer Backup 2

1.3 MyriadStore: A Peer-to-Peer Backup Application 3

1.4 Outline 5

2 Background **7**

2.1 Peer-to-Peer Networks 7

2.1.1 Structured and Unstructured Peer-to-peer Networks . . . 8

2.2 Distributed Hash Tables 8

2.2.1 The Overlay Network 8

2.2.2 The DHT 9

2.2.3 Insertion and Retrieval from the DHT 9

2.2.4 DHT Properties 10

2.3 The DKS Structured Peer-to-Peer Network 11

3 Design of MyriadStore **13**

3.1 Basic Features 13

3.1.1 Backup 13

3.1.2 Restore 14

3.1.3 Backup Sets 14

3.1.4 Symmetric Trading 14

3.1.5 Security and Availability 15

3.2 Symmetric Trading 15

3.2.1 Trading Scheme 15

3.2.2 Reputation 16

3.2.3 Extending the Trading Scheme 16

3.2.4 Receipts 17

3.2.5 Challenges 17

3.2.6 Trading in a fully utilized network 18

3.3 Data Organization 20
3.3.1 Separate Management of Meta-data and Files 20
3.3.2 Organization of Backup Data 20
3.3.3 Organization of Meta-data 22
3.4 Security 26
3.5 Availability 26
3.6 The Backup Process 28
3.7 The Restoration Process 31
3.8 Failures 36
3.8.1 Introspection 36
3.8.2 Inspection 36

4 Implementation **39**
4.1 Architecture of MyriadStore 39
4.1.1 Implementation Goals 39
4.1.2 Software layers 39
4.1.3 Layers of Communication between Nodes 40
4.1.4 Layer Reusability 41
4.2 Software Packages 42
4.2.1 The *ui* Package 42
4.2.2 The *backup* Package 43
4.2.3 The *storage* Package 43
4.2.4 The *util* Package 44

5 Related Work **47**
5.1 Distributed Backup versus Distributed File System Applications 47
5.2 Distributed Backup Applications 48
5.2.1 pStore 48
5.2.2 PeerStore 51
5.2.3 Pastiche 54
5.2.4 Cooperative Internet Backup Scheme 57
5.3 Distributed File System Applications 58
5.3.1 OceanStore 58
5.3.2 The Cooperative File System 60
5.3.3 PAST 61
5.4 Comparison of Distributed Backup/File System Applications . . 63
5.4.1 Distributed Hash Table Usage 63
5.4.2 Fairness and Efficiency in Storing Data 64
5.4.3 Data Availability and Mutability 65
5.5 Other Distributed Backup/File System Applications 66

6 Conclusion **69**
6.1 Future Work 70

List of Figures

2.1 An example of a ring topology 9
2.2 Distribution of a hash table's entries among the network nodes . 10
2.3 The DKS architecture layers 11

3.1 The process of gaining storage rights in a full network 19
3.2 Overview of data organization of MyriadStore 24
3.3 Splitting meta-data results in smaller migration costs 25
3.4 The structure of the meta-blocks 27
3.5 Protocols between MyriadStore's nodes 30
3.6 Algorithm for a node that initiates a trading session 31
3.7 Algorithm for a node that receives a request for trading 32
3.8 Algorithm for a node that sends a request for storing a file block 33
3.9 Algorithm for a node that receives a request for storing a file block 34
3.10 Algorithm for a node that sends a challenge to a partner 34
3.11 Algorithm for a node that receives a challenge from a partner . . 35
3.12 Algorithm for a node that sends a request for retrieving a file block 35
3.13 Algorithm for a node that receives a request for retrieving a file block 35

4.1 MyriadStore's software layers 41
4.2 The communication between two nodes in MyriadStore 42

5.1 File blocks and file block lists in pStore 50
5.2 An overview of how PeerStore works 52
5.3 Encryption and naming of Pastiche chunks 56

List of Tables

4.1 The software packages of MyriadStore 45

5.1 Comparison between distributed backup applications 66
5.2 Comparison between distributed file system applications 67

Chapter 1

Introduction

Rapid developments in computer hardware continuously make newer and increasingly more sophisticated devices available. New microprocessors perform tasks even faster and new storage devices can store increasingly greater amounts of data. Today's ordinary workstations have therefore become quite powerful computers with excess computing resources, including processor power and storage space. This, in conjunction with broadband internet becoming more common, has paved the way for peer-to-peer computing and the development of peer-to-peer applications.

Peer-to-peer computing has gained recognition in recent years and has become a popular approach to computing. In peer-to-peer networks a large number of participating computers, owned by individuals or organizations, share resources for computing and data storage tasks without the need for a centralized controlling server. This decentralization property aims to prevent bottlenecks and single points of failure in the network.

An interesting application for peer-to-peer systems that has recently received considerable attention is that of backing up data. The process of backing up data involves copying and maintaining copies of important data so that in case the original data is damaged or lost, it can be restored from a backup copy. In Section 1.1 we describe the methods that are traditionally used for backing up data and Section 1.2 discusses how these methods can be improved with the use of a peer-to-peer system. Section 1.3 introduces MyriadStore, the solution we propose in this thesis for backing up data. Finally, Section 1.4 gives an outline of the remainder of this report.

1.1 Traditional Backup Methods

Backing up data is a well-known and commonly used process as the safekeeping of data is an important task. Most companies, organizations, as well as individuals, regularly backup their data, as a scenario of losing them due to some physical disaster of some other reason would result in a, most probably, unaffordable cost. The backup methods that are traditionally used involve copying data to removable media such as tapes, CDs, and DVDs, and then transferring it somewhere off-site as it needs to be recoverable in case of an on-site disaster. These actions need to be done frequently enough to ensure that data can be recovered when needed. In these procedures there are several drawbacks and inefficiencies that can be pointed out:

- Removable media such as tapes, CDs and DVDs do not make for long-term reliable media as they deteriorate over time and are susceptible to damage and loss.
- The backup process takes a long time to complete. Tapes, which are the most common media to be used when taking backup of large amounts of data, have very slow read and write speeds. This means that both backing up data and retrieving it from tape is very time consuming.
- Considerable operational costs are involved, such as hiring personnel for moving the backed up data to a remote site, as well as making sure that it is safe.
- The media holding the backup data needs to be well organized to prevent accidental loss of it as well as to allow easy recovery of the data.

The previous description shows that traditional backup methods require users to perform many tasks. The user needs to manually write the backup data to removable media, transfer it to a remote location and take all the necessary measures to ensure that it is safely stored there. Undoubtedly, this is a costly and cumbersome process.

1.2 Peer-to-Peer Backup

Ideally, the backup process should be as transparent as possible to the user without involving him/her with details of how data is backed up and how it is stored safely. The user should only be required to perform a minimum number of tasks for the backup process and still the backup data should be kept private and secure, and of course it should always be available so that the user's requests for backup data are always satisfied.

Peer-to-peer backup applications have emerged as an approach that promises to offer a backup process that, compared to the traditional backup methods, is closer to the ideal description we gave previously. These applications utilize free hard disk space of computers in a network to store data that a user desires to backup. This is a transparent process to the user, as the user does not have any knowledge of how the backup data is stored to the remote computer's hard disks and on which computers the backup data resides. Those are details that the user does not need to worry about. From the user's point of view, the important requirements that need to be satisfied are the availability and security of the backed up data. Peer-to-peer backup applications aim to meet these requirements, while keeping performance at satisfactory levels.

Peer-to-peer backup applications provide a ubiquitous and easy access to backed up data from anywhere at any time. A user only needs to have access to a computer connected to the network and the necessary authentication information in order to access the data he/she has backed up. Another advantage of this class of applications is that they do not require any special hardware. Furthermore, expendable media and special personnel for safekeeping the backup data, which are essential resources for the traditional backup methods, are not needed. This makes it clear that peer-to-peer backup applications impose little or no administrative costs, a fact that makes them a much cheaper and more efficient solution when compared to the traditional backup methods.

1.3 MyriadStore: A Peer-to-Peer Backup Application

In this report we present MyriadStore, a peer-to-peer backup application that targets at realizing the desirable properties of the class of peer-to-peer backup applications we explained previously. Therefore, MyriadStore offers a backup process that is:

- *Transparent to the user.* Using MyriadStore's client a user only needs to specify which data should be backed up, optionally specify additional settings (such as degree of compression and security for the backed up data as well as scheduling options of the backup), and then choose to proceed with the backup process simply with a click of a button. The user can then be sure that the specified data is backed up without having to know how this backup process took place and where the data was placed. The user can be sure that the backed up data can be retrieved when needed.

- *Cheaper than traditional non peer-to-peer backup.* With MyriadStore there

is no need for hiring personnel for copying the backed up data to removable media, transferring it off-site and ensure its safekeeping. Furthermore, there is no need for using any expendable media for storing backed up data.

While achieving these desirable characteristics, MyriadStore manages to keep the backed up data:

- *Easily accessible.* A user can access it from anywhere at any time by specifying all the required authentication information.

- *Well-organized.* At any moment the user can easily browse through all his backed up data and can easily locate items that need to be recovered.

- *Secure.* Mechanisms are in place for encrypting the backed up data and ensuring its safekeeping.

- *Highly available.* MyriadStore adapts techniques for storing backed up data that ensure that it can still be retrieved even if some of the hosts accommodating it are unavailable.

For realizing the desirable features and properties that were described previously certain design decision were made. A major design decision that brings the performance of MyriadStore to desired high levels regards the approach used for storing backed up data and meta-data of the system. In chapter 3 we will see that there are different approaches used for storing these two types of data and we will describe in detail the benefits of this design decision. Regarding fairness, MyriadStore makes use of a symmetric trading protocol that ensures that each node contributes to the system as many resources as it consumes from it. To address the issue of safekeeping, MyriadStore uses a punishment model for punishing nodes that are not storing data they have been entrusted with. This punishment model takes into consideration the nodes' reputation in the system so that it is possible to differentiate between nodes that behave maliciously and nodes that might be temporarily unavailable. Regarding data privacy, a public key infrastructure is used for encrypting the remotely stored data and preventing unauthenticated access to it. Lastly, the availability issue is handled by replicating data to multiple nodes so that if one of them becomes unavailable then another replica of the desired data will still be available in the system, stored at another node.

1.4 Outline

The remainder of this report is structured as follows: Chapter 2 gives the reader some background on peer-to-peer systems. Chapter 3 explains in detail the design decisions that MyriadStore is based on and gives detailed descriptions of the functionalities it provides and the procedures that actually take place during MyriadStore's operations. Chapter 4 gives details about MyriadStore's implementation. Chapter 5 surveys work in the fields of distributed backup and distributed file systems. Finally, Chapter 6 concludes the report.

Chapter 2

Background

This chapter gives the reader some background information on basic concepts that are used throughout this report. Section 2.1 explains the notion of peer-to-peer networks and gives their main characteristics and categories. Section 2.2 gives the definition and properties of a Distributed Hash Table. Finally, Section 2.3 gives the definition and the main characteristics of the Distributed K-ary System (DKS) [1].

2.1 Peer-to-Peer Networks

A peer-to-peer network is a network in which all participating computers, or the so-called nodes, behave as equal peers. Thus, a peer-to-peer network is decentralized without having dedicated servers responsible for serving clients' requests. Every node in a peer-to-peer network has a dual role as a client and as a server and is therefore able to both serve requests as well as make requests. The main advantage of such a network design over the classic client-server design is that there are no centralized servers that might become bottlenecks for the whole network. Furthermore, in the client-server design, centralized servers are single points of failure whereas in peer-to-peer networks such an issue is avoided.

One main characteristic of peer-to-peer networks is the potential heterogeneity of the nodes of which it consists of, as there is no restriction on their characteristics. For instance, the available bandwidth, the processor speed, the memory and the disk space capacity are all aspects that might differ from node to node in a peer-to-peer network. Another characteristic of peer-to-peer networks is that their participating nodes do not necessarily have to be trusted. Nodes may be malicious or simply not available, and they are free to arbitrarily join or leave the network. Therefore, it is clear that a peer-to-peer network is a very dynamic environment that continuously changes.

2.1.1 Structured and Unstructured Peer-to-peer Networks

Peer-to-peer networks can be categorized into two classes: *unstructured* and *structured*. The logical topology of an unstructured peer-to-peer network is random. Whenever a node makes a request for some data, this request is propagated on a hop-by-hop basis flooding the network until the node that is holding the requested data is reached, or until the request's given time-out expires. In an unstructured peer-to-peer network it can be the case that some data is not found even if it exists in the network. Another major shortcoming of this approach is its efficiency, as it doesn't scale very well as the number of nodes in the network increases. Gnutella [15] is a typical example of an unstructured peer-to-peer network.

In structured peer-to-peer networks the logical topology has some kind of structure. For instance, the nodes of such a network may form a ring or a mesh topology. Ideally, in a structured peer-to-peer network a search of some data is guaranteed to return after some deterministic number of steps and, thus, data location in such networks is more efficient than in the unstructured peer-to-peer networks. DKS [1], Chord [16], Pastry [17], CAN [30], Tapestry [34], Koorde [32], Skipnet [33], EpiChord [35], and OpenDHT [36] are some examples of structured peer-to-peer networks.

2.2 Distributed Hash Tables

Due to the efficient lookup mechanisms that structured peer-to-peer networks provide, they can be used as a suitable infrastructure for the development of peer-to-peer applications. A basic mechanism that has been developed on top of structured peer-to-peer networks is that of a Distributed Hash Table (DHT). A DHT is a service that allows maintaining and using a hash table structure, the entries of which are dispersed among the nodes of a structured peer-to-peer network. The next subsections explain DHTs on top of structured peer-to-peer networks that use ring topology and give an overview of their features and functionalities.

2.2.1 The Overlay Network

A ring topology in a structured peer-to-peer network is realized by assigning identifiers to the nodes of the network. Each node is assigned a unique identifier that is picked from some identifier space. With this assignment, nodes form a virtual ring according to their identifiers (Figure 2.1). As mentioned before, this ring topology facilitates the basic mechanisms for routing and looking up

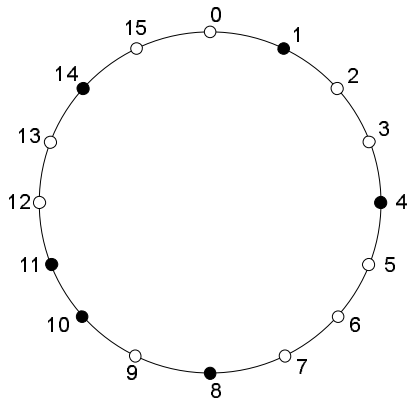

Figure 2.1: An example of a ring topology with a node identifier space of size 16. The network contains 6 nodes that have been assigned identifiers 1, 4, 8, 10, 11 and 14 (black circles). Any node that joins this network will be assigned a number from 0 to 15 that is not already used. The white circles solely indicate the potential identifiers that can be assigned and have not yet been assigned.

between the nodes. This virtual ring network topology is often called the *overlay network*.

2.2.2 The DHT

The DHT provides a hash table abstraction that allows inserting data items into the system and retrieving them from it again. The DHT can be considered as a usual hash table data structure, entries of which are distributed among the nodes of the overlay network. Using the DHT, data items can be inserted into the overlay network by specifying a key for each one of them just as if they were inserted into an ordinary hash table structure. Keys are chosen from the same identifier space as identifiers for the nodes are chosen from. Similarly to a usual hash table, in order to retrieve a data item that was inserted into the DHT, the key with which it was inserted needs to be specified.

2.2.3 Insertion and Retrieval from the DHT

Whenever a data item is inserted into the DHT, the node that will be responsible for storing it is determined by the key that the data item is associated with. This technique of routing is called key-based routing. Different DHT implementations employ different techniques for assigning data items to nodes that will be responsible for them. In Chord [16] for example, a data item will be stored on the node that has the smallest possible identifier that is greater or equal to its key. This node is called the successor of the data item's identifier (Figure 2.2). When asked to store a data item under some key, the DHT uses

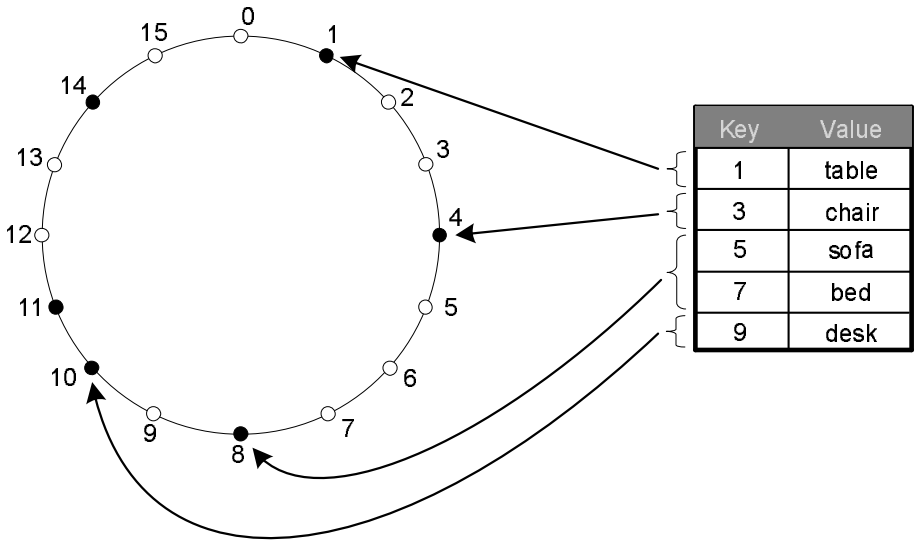

Figure 2.2: Distribution of a hash table's entries among the network nodes. The DHT can be considered as an ordinary hash table structure distributed among the nodes of the overlay network. According to the key-based routing used in Chord, a data item is stored on the successor of its identifier, that is the node that has the smallest possible identifier that is greater or equal to the data item's key.

the key-based routing to determine which node should be responsible for holding the particular data item. Once the address of the node that is the successor of the data item's identifier is known, the DHT can contact the node and ask it to store the item.

For retrieving a data item, the DHT uses the key-based routing to determine who should be holding the particular data item, that is, the successor of the data item's identifier. After having determined the data item's holder, the DHT can contact this node to retrieve it.

2.2.4 DHT Properties

It should be mentioned that a major responsibility of the DHT is to maintain data items inserted to it. The DHT achieves this by moving the data items accordingly as nodes join and leave the system. Furthermore, node failures is also an issue that a DHT needs to confront. The solution to failing nodes is replication of the data items to several nodes. When a node fails, the data it is holding can be retrieved from other nodes that hold replicas of this data. Moreover, the replicas that the failed nodes are holding can be reproduced and stored in other alive nodes, keeping in this way the replication factor to the desired levels.

Generally, the desired properties that a DHT is aiming for are:

- *Decentralization.* The data items are dispersed among the nodes of the peer-to-peer network and there is no any central coordination between the network's nodes.
- *Scalability.* The DHT should be able to operate efficiently even when the number of nodes becomes large.
- *Fault tolerance.* The DHT should be able to tolerate node failures.

2.3 The DKS Structured Peer-to-Peer Network

MyriadStore is built on top of the DKS [1, 2, 3] (Distributed K-ary System) which is a middleware that constructs a structured peer-to-peer system. It provides services for distributed hash tables and group communication. The nodes participating in the DKS peer-to-peer network are forming a ring topology and all the properties and features we described in the previous subsection hold in this network as well. The DHT of DKS plays a fundamental role in the design of MyriadStore as we will see in chapter 3. Figure 2.3 shows the DKS architecture layers.

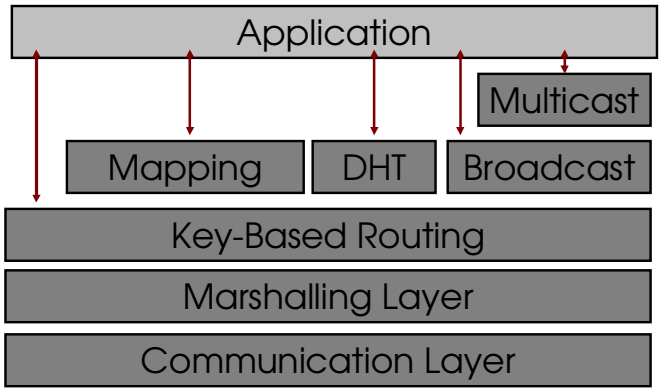

Figure 2.3: The DKS architecture layers. MyriadStore resides on the Application Layer.

DKS is different from other DHTs, such as Chord, as it gives lookup consistency guarantees, uses a different replication scheme, and provides efficient group communication abilities. The lookup consistency is achieved by using a protocol called *local atomic actions*, which ensures that the responsible node for any identifier is exactly one node in presence of nodes joining and leaving. This is not the case with Chord, which at times can have more than one node

as responsible for an identifier. For MyriadStore, this is of high importance as we do not want the joining and leaving of nodes to affect access to backups. DKS also uses a replication scheme called *symmetric replication*, which has a cost of $O(1)$ for restoring the replication degree when nodes are joining and leaving. This is in contrast to systems, such as Chord and Pastry, which need to shuffle files on $O(n)$ nodes, where n is the replication degree. Furthermore, the scheme is suitable for erasure codes, as encoded blocks can be used instead of replicas. Symmetric Replication then allows to do efficient parallel lookup to any k replicas. Last, DKS provides *bulk lookup*, which provides the ability to efficiently do parallel lookups to thousands of identifiers, while guaranteeing that each node will send maximum $\log_2(n)$ messages, where n is the number of nodes in the system. This is important for MyriadStore, as sometimes a file can consist of thousands of blocks. Fetching them with individual lookups would incur an overhead of marshaling and sending thousands of messages, while this guarantees that in the worst case a node sends a message to each neighbor.

Chapter 3

Design of MyriadStore

This chapter delves into the details of the design of MyriadStore. Section 3.1 gives an overview of the functionalities provided by MyriadStore and explains what operations and features it provides to its users. Section 3.2 gives a description of the symmetric scheme with which storage rights are exchanged between users in MyriadStore. Section 3.3 explains how data is organized in the system and discusses why some design decisions were taken over than others. Sections 3.4 and 3.5 discuss how the security and availability issues of the backed up data are handled in MyriadStore. Sections 3.6 and 3.7 give a description of the activities that take place in the system when backup and restore operations are performed. Finally, Section 3.8 explains some assumptions of MyriadStore regarding failures, and some ways for being able to overcome some faulty situations that might occur.

Throughout this chapter, as well as in the remainder of this report, unless stated otherwise, the term *node* will refer to a computer that participates in the peer-to-peer network and runs MyriadStore. A *MyriadStore client* is the front end part of MyriadStore that a *user* uses to interact with the system.

3.1 Basic Features

3.1.1 Backup

Through a MyriadStore client, a user can select some data to be backed up. To be able to perform a backup of the specified data, a node needs to find remotely available disk space. This is done by the node trading some amount of its local disk space with other nodes in the peer-to-peer network. The amount of local disk space that a node is willing to trade with other nodes is limited by the allocated local disk space specified by the user. When enough remotely

available disk space is available a backup can be performed. Files that are to be backed up are encrypted and partitioned into chunks and sent out to peers that have made their local disk space available. The backup process is discussed in detail in Section 3.6.

3.1.2 Restore

After a user has performed a backup, he/she can review and verify that it was completed successfully. A user can browse files stored in the system by date or browse different versions of an individual file. Similarly, a user can restore backed up files from a specific date, either individually or as a whole, as well as restore a specific version of a specific file. When retrieval is performed, all the chunks of a file need to be located and retrieved from other peers before a file can be restored by reassembling the chunks and decrypting the reassembled chunks. The restoration process is discussed in detail in Section 3.7.

3.1.3 Backup Sets

A user can define multiple sets of files to be backed up. One set could for instance contain a user's digital photographs, whereas another set could contain a user's financial records. Backup sets should be helpful to the user as, by using this feature, he/she is able to organize and have a better overview of what is backed up. Files can be added to multiple backup sets.

Each backup set can be configured to have different properties, such as replication degree, backup frequency and compression degree. Stronger encryption could be specified for backup sets that contain sensitive data. A higher replication degree could be specified for backup sets that should have higher availability. A higher degree of compression could be specified if a user chooses to utilize the computing power of his/her computer to minimize the amount of data he stores in the system. As we will see shortly, minimizing the amount of data one stores in the system also results in minimizing the amount of local disk space needed to store data on behalf of other users.

3.1.4 Symmetric Trading

To give the users incentives to contribute storage space in the system, MyriadStore adapts a symmetric way of exchanging storage rights between the users. MyriadStore guarantees that a user has as much space available in the network as the one the user contributes to the network. Users must be frequently present in the system to be able to perform backups and to make their local disk space available to other users. If a user is not present for longer periods of time, other

users may choose to punish him/her by gradually dropping data he/she has stored on their local disk space. A mechanism is in place for nodes to verify that their users' data is actively being stored in the system. If they discover that a node is not fulfilling its obligations they may choose to punish it by dropping data they are holding for it. This punishment model takes into consideration the reputation of the users in system, as we will see afterwards. With such a punishment model MyriadStore gives incentives to the users to behave nicely. The trading scheme is discussed in detail in section 3.2.

3.1.5 Security and Availability

In MyriadStore, the data that a user backs up is stored to remote computers and, therefore, needs to be encrypted in a way that only its owner should be able to access them. Furthermore, MyriadStore needs to maximize the availability of the backed up data. To solve this issue, the case in which nodes holding some backed up data are offline should be handled. MyriadStore's approach regarding the issues of security and availability are discussed in sections 3.4 and 3.5.

3.2 Symmetric Trading

3.2.1 Trading Scheme

To be able to store something in the system, a node needs to have some remote disk space available. Users exchange storage rights with each other in a symmetric fashion. This means that a user reserves the same amount of storage space on his/her local disk space for others to store their backup data as they reserve for his/her backup data. In other words, if a user offers to other users an amount A of storage space for storing their backup data, then this user will have the same amount A of storage space available to him/her in the network for storing his/her backup data.

Users specify how much of their local disk space they are willing to share with others and nodes trade disk space with each other according to a trading protocol. Each trading session that takes place according to this protocol involves two nodes of the peer-to-peer network. Upon the completion of a trading session, trading nodes have allocated disk space to each other according to a *contract* which is signed by both parties. A contract indicates how much disk space each party is willing to share with the other. There is no limit in the number of contracts that two peers can establish with each other.

According to the logic of contracts, after a node A has established a contract c with another node B, then A is entitled to utilize the amount of data space it

has reserved on B in terms of the contract c. Thus, A can send some data d to B specifying that c is the contract that is being utilized. In this way, the data d will be associated with the contract c.

Contracts have a fixed expiration date but contracts can be renewed as needed. After the expiration date has passed, the contract is no longer valid and nodes are no longer responsible for storing data items associated with the contract. Two nodes that have agreed upon one or more contracts are called *partners*.

3.2.2 Reputation

In order to predict reliability of nodes, a reputation statistic is maintained for every node in the system. A correlation exists between a node's reputation and its grace period, that is, the time it is given to recover its data in case of a crash.

For every partner a node has, it maintains a numeric value that represents the opinion it has of the partner. A node's opinions are determined from interactions with its partners. The reputation of a specific node is determined by taking the average of opinions other nodes have of the specified node. Reputation of the other nodes is taken into account and their opinions are weighted by their own reputation. Opinions are maintained in a global entity that any node can access. Nodes digitally sign their opinions before publishing them for other nodes. This is done for being able to verify the authenticity and origin of the opinions.

When two nodes agree to exchange disk space they give each other an initial opinion value. This value gives each of the nodes some initial grace period.

3.2.3 Extending the Trading Scheme

The quality of the backup service that a node provides to others does not only depend on the amount of storage space it is offering. Nodes are located in geographically dispersed locations and are connected to the internet in different ways. Some nodes might have a permanent, high speed connection whereas others might be on a dial-up modem connection. Therefore, this symmetric scheme could be extended in the way that other properties of nodes, other than disk space, could be considered when making contracts. Instead of considering only the amount of disk space exchanged, other properties such as, uptime of users' nodes, reputation, bandwidth and backup/restore rates would be considered. This would enable users to find partners according to their requirements and needs. When making requests for trades, users could set a minimum requirement for availability of disk space, reputation, how frequently backup data items could be retrieved and bandwidth on trading nodes. If a trading node is not able to fulfill all the requirements of a specific trade request it could for

instance offer more disk space to make up for its limitations. These additional properties would be embedded in the contracts.

3.2.4 Receipts

When a contract has been made, partners can start to utilize each others' disk space as specified in the contract. Whenever sending a request for storing data to a partner a reference to a contract previously established is included. This enables the node receiving the request to verify if the claimed partner has indeed storage rights on it. For each data item stored, a node issues a receipt where it acknowledges receiving the data item correctly and that it will actively store it. The receipt is then sent back to the owner of the data item. All receipts are signed by the issuer so they can later be verified for authenticity. Any node holding a receipt can be sure that the issuer has indeed received a particular data item and that the issuer takes responsibility for actively storing it while the referred contract is valid.

3.2.5 Challenges

In order to verify that partners are actively storing data items they are responsible for, a challenge mechanism is in place. A node randomly picks a data item stored with a partner and challenges the partner to provide some part of the data item. Challenges are done periodically. A receipt for the data item in question is attached to the challenge so that the node being challenged can verify the authenticity of the challenge.

Before a challenge can be sent a node needs to prepare the challenge by determining which data it should ask for. It would not work to ask for just any arbitrary data as then the node would have nothing to compare a challenge response against, and a challenged node could reply with any arbitrary answer. There are two ways for a node to prepare challenge data. One way would be to generate a set of challenge data for the data item before sending it to a partner. When the node would want to send a challenge it could pick an item from the set of challenge data and use that to compare against a challenge response from its partner. As data items continue to exist locally on a node, another way would be to generate challenge data from the local data every time a challenge should be prepared.

For successful challenges, nodes are awarded points which are added to the challenging node's opinion of them. Similarly, in case of a failed challenge, points are deducted from the challenging node's opinion. When a node's reputation rating, which is a weighted average of their partners' opinions (as explained in Subsection 3.2.2), reaches zero points they will be punished by the challenging

node which will delete some of the data items of the node that failed the challenge. By adapting a similar technique as the one proposed in Samsara [10], the probability with which data items are dropped could have a small initial value but which would increase exponentially with the number of failed challenges.

By maintaining a reputation statistic, nodes that behave nicely accumulate good-will and will in turn have a longer grace period to recover their data items from their partners in case of a disaster. They could also be absent for some period of time before they would be punished for it. Nodes are fundamentally selfish in the way that they are only interested in keeping their remotely stored data safe. This is an incentive to honor contracts with partners and to be present in the system as then they can monitor their partners and make interventions if needed. If a partner is found to be failing challenges a node could try to replace the partner with a new one, and migrate all data items to the new partner. This could even be done prematurely or before the expiration of a contract.

After experiencing a disaster where data is lost, a node could attempt to recover all data items it was storing for others after it has restored its backed up data from its partners. To determine which data items it should recover it could ask all partners to provide receipts for data items it was responsible for. By inspecting the receipts it could verify that it was indeed storing them and could then recover the items from identical replicas stored on other nodes.

3.2.6 Trading in a fully utilized network

MyriadStore adapts a technique inspired by [9] for handling the case where a new node joins a network in which all the nodes have used up all the space they are willing to offer for storing other nodes' backup data. When all available storage space on all nodes has been exhausted and a new node enters the system, nodes can free up local disk space by forwarding data items to the new node. By accommodating this data, the new node gains storage rights on the nodes that forwarded their data to it. This process is illustrated in Figure 3.1.

Nodes allow this process to take place with some probability. Before it is allowed, the requesting node should have already tried making trades with several nodes. When a trading session has been completed the accepting node forwards data items stored locally to the requesting node. An accepting node cannot offer more space to a requesting node than the amount of local disk space used to store data items for nodes other than the requesting node.

A node that forwards data items to another node is still responsible for those data items, even though they no longer reside on it. This node will still be receiving challenges for the data items it has forwarded. In this case the challenges should be forwarded to the node where the data items now reside.

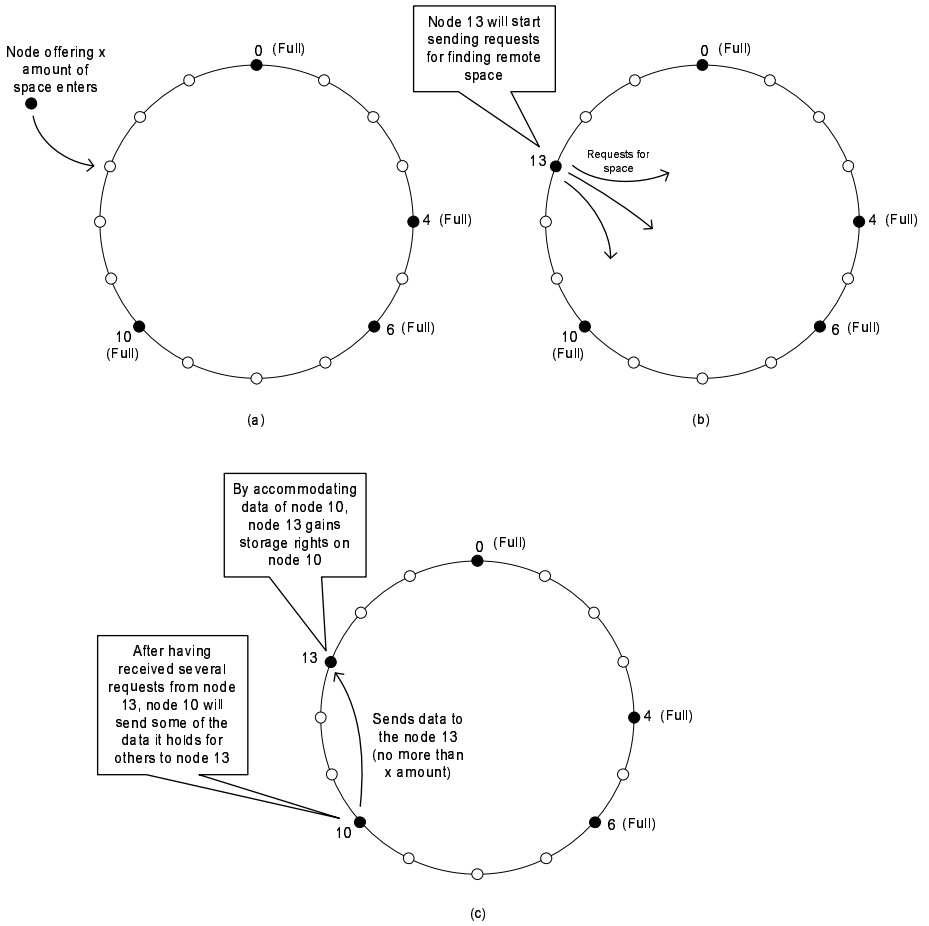

Figure 3.1: The process of gaining storage rights in a full network. (a) shows a network where every node has exhausted the space they offer to others for storing their backup data. Node 13 joins this network and is willing to offer an x amount of disk space to others. Node 13 will start sending requests for finding space but, since no node has some free space to offer, the requests will be initially ignored (Figure (b)). However, after some node (in this case node 10) has received many requests for space by node 13, eventually it will send some of its data to node 13. By accommodating data of node 10, node 13 gains storage rights on node 10 (Figure (c)).

3.3 Data Organization

3.3.1 Separate Management of Meta-data and Files

The data used in MyriadStore are of two types: the actual data that a user backs up and the meta-data needed for being able to access the actual data as well as serving several other purposes, as we will discuss later. Both actual data and meta-data are stored remotely. There would be no point in designing a distributed backup system if we were to store this data on the local host. However, in MyriadStore there is a significant difference on how these two types of data are stored in the network, as meta-data storage is decoupled from actual data storage. Meta-data is stored in the DKS [1] Distributed Hash Table (DHT). Actual data is stored directly on the DKS nodes' local file systems.

One reason for decoupling the meta-data storage from the actual data storage is because such an approach minimizes network traffic and migration costs. If the DHT was used to store actual data then the data items of literally hundreds of megabytes would have to move frequently from one node to another as nodes join and leave the system ([37]). Since actual data items are quite large, moving them would cause considerable network traffic and migration costs. It would also imply that a join or leave operation would take hours to complete. Storing these data items directly to the DKS nodes' local file systems avoids these costs. On the other hand, meta-data is usually small and that gives the freedom to use the DHT for holding it, since moving them will not impose too much migration cost. Furthermore, the availability of meta-data at all times is a feature that the system needs. Meta-data holds information about all the settings a user has and all his/her backup sets. This information should be accessible to the user at all times. Since meta-data is saved into the DHT, it is the DHT's responsibility to ensure the meta-data's availability as nodes join, leave or fail. The DHT undertakes the task of replicating and moving the meta-data accordingly so that they will be always available upon request.

Additionally, not using the DHT for storing actual data items offers greater flexibility on managing where the data will be placed. Such a flexibility gives the freedom to apply fair strategies on storage space usage.

3.3.2 Organization of Backup Data

A desired property of a distributed backup system is decentralization on the way that backed up data is stored. This implies that data to be backed up should be dispersed among nodes participating in the system. The main advantage gained by storing the backed up data using this method, is that efficient techniques for resolving the issue of the availability of the backed up data can be applied. For

instance, a suitable replication scheme could be used for the dispersed pieces of data so that if a node known to hold some data is not reachable, then another node that is holding a replica of this data could be contacted. Such a replication scheme would not be efficient if the dispersed pieces of data were too large.

Another advantage gained by dispersing the backed up data into the network is that the disk space capacity that the nodes are offering for accommodating other nodes' backed up data is better utilized. Consider, for example, the scenario where a node has to backup a file of size of 5Mb and there are 20 other nodes in the system, each one of them offering 500Kb for other nodes to store their backed up data. If the 5Mb file was not to be split into smaller pieces then there would not be enough space in the system to store it, even though the overall capacity of the system is enough for storing 5Mb of data. Breaking the 5Mb file into pieces of 500Kb would solve this problem.

Storing the data in the network in small pieces also has the advantage that both their sending to remote nodes and their retrieval from the nodes where they reside can be done in parallel. Therefore, the backup and retrieval process can be done more efficiently in comparison with the case where data were stored remotely as a whole.

Lastly, by partitioning the backed up data into small pieces, there is some probability that identical such pieces will be created. The smaller these pieces are, the greater the probability that some identical ones will be created. Thus, another advantage of partitioning the backed up data in pieces is that if identical pieces were created by this partitioning then only one of them would need to be stored remotely. There would be no point in storing remotely data pieces that are identical. Efficiency could be even more improved if these data pieces where encrypted using *convergent encryption* [29]. In this case if several nodes needed to store the same data piece then this would only need to be saved once and then the owners of this piece could share it.

MyriadStore disperses the backup data into the network as described before and, thus, exploits the advantages that were explained previously. The use of a structured peer-to-peer overlay network such as DKS facilitates the application of this technique. To be able to disperse a file's data in the network, MyriadStore first splits it into relatively small chunks of data called *file blocks*. Every file block has a fixed maximum size z (currently 400Kb). MyriadStore's meta-data records the required information for finding the file blocks of the backed up files in the network and for reassembling the backed up files from them.

3.3.3 Organization of Meta-data

MyriadStore needs to maintain several kinds of meta-data. Information about which files a user is backing up, together with the various settings that have been selected for them, the associations between the backed up files and their file blocks and the location of these file blocks in the network, the partners that a node has found and the deals that were agreed during the trading sessions, is all meta-data that a MyriadStore client needs to have access to. As explained previously, MyriadStore's meta-data reside on the DHT of DKS. This subsection discusses about how the meta-data information is structured and how it is stored on the DHT.

Structuring the meta-data

An interesting design issue that arises is how the meta-data can be structured so that the process of retrieving it from the DHT is done efficiently. To motivate the meta-data structure used in MyriadStore, we will present the advantages and disadvantages of alternatives that can be used.

Consider the scenario where all the meta-data is handled as a single unit. This would imply that both saving and retrieving all the meta-data from the network would occur in one single step. Furthermore, let's suppose that many files have been backed up and each file has more than one backed up versions. It's clear that in this case the size of the meta-data will be large, as it needs to hold information for each of the backed up files. If at some point a file that has been backed up needs to be retrieved, the meta-data will need to be consulted to determine the location of the data of the backed up file in the network. With this approach, even though the user is only interested in a small part of the meta-data, he/she will have to retrieve all of it. It is obvious that this approach for storing the meta-data is not efficient enough.

A way to optimize the previous approach would be to structure the meta-data in levels. At a first lookup for meta-data, the first level would be retrieved, and having this in place a second level could be accessed and so on. This approach would prevent accessing meta-data that will not need to be used. This might sound as a better approach but again, if many levels were introduced in such a structure then the number of lookups on the DHT would increase, and this would significantly increase the retrieval time of the desired meta-data. Another problem that comes up by introducing several meta-data levels is that whenever an update needs to be done to some level, then the data of all the other levels associated with this change would need to be updated as well, to keep the meta-data structure consistent. It is clear that as the number of levels increases, this problem becomes more severe.

In an attempt to minimize the number of lookups on the DHT, MyriadStore has its meta-data organized in two levels. The first level keeps three different types of information: the *backup meta-data*, the *storage meta-data* and the *contract meta-data*. The second level accommodates the *file block lists* of the backed up files. Instances of these types of meta-data are treated as separate *meta-data units* when they are saved and retrieved from the DHT. Figure 3.2 gives an overview of this meta-data structure. The next paragraphs give an overview of the information that each of these types of meta-data holds:

File Block List. A file block list contains information about file blocks and how they can be assembled to retrieve a backed up file. Each backed up file is associated with its own file block list. It might be the case that a file has several versions that have been backed up. In this case, each version will be associated with its own file block list.

Backup meta-data. The backup meta-data hold information about all the settings and selections that the user has specified. For instance, the backup sets that have been created, the files that each backup set contains, the various setting for encryption and compression associated with each backup set and the scheduling settings for each backup set are information contained in this type of meta-data.

Storage meta-data. The storage meta-data maintain an entry for each one of the files that is being backed up and this entry holds the required information for locating the backed up file's file block list.

Contract meta-data. The contract meta-data contain information about contracts between partners. For each contract entry there is information about which partner is associated with the specific contract, the storage space exchange agreement that was made with this partner, as well as the file blocks, if any, that have been saved in terms of the particular contract. Combining the information of the contract meta-data and the file block lists it can be determined which partners hold the file blocks of each backed up file.

Storing the meta-data

The DHT ensures the availability of the meta-data items by moving them accordingly as nodes join and leave the system. This scheme is based on the assumption that the meta-data items are of small size, so that moving them will not cause serious network traffic. However, previously we explained that meta-data may be large in some cases. Saving this meta-data into the DHT as

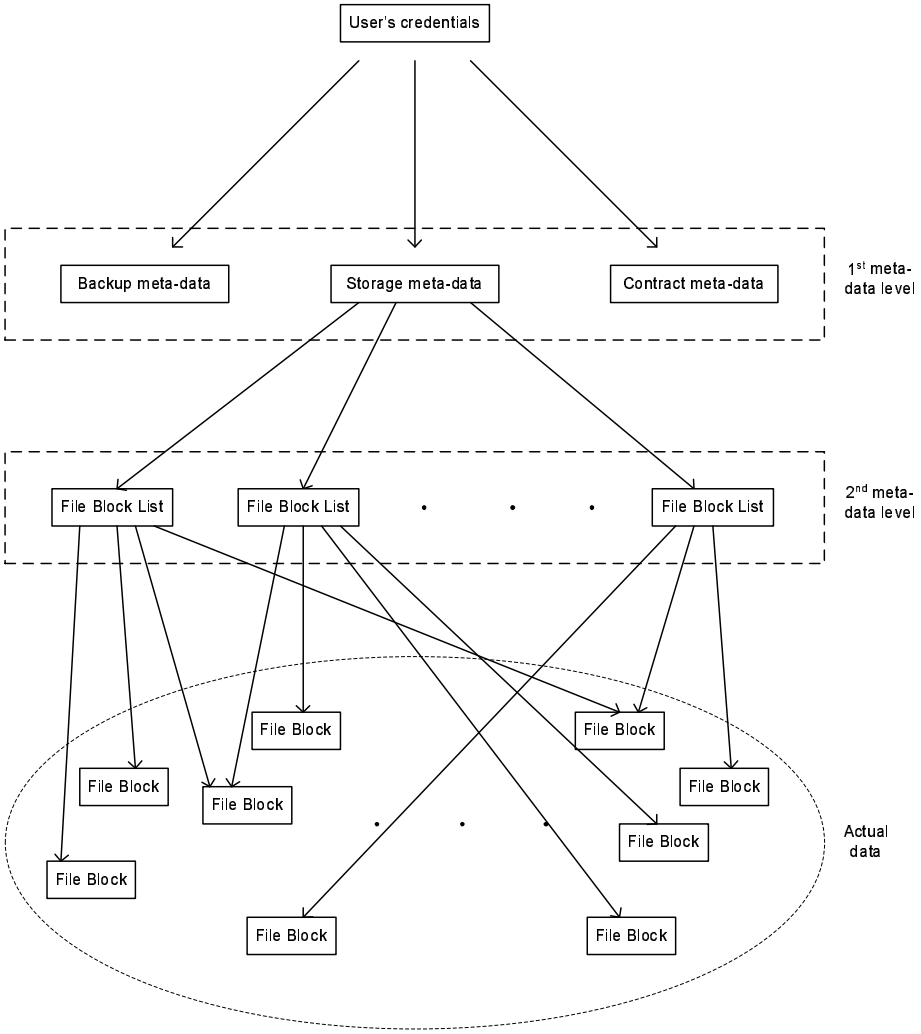

Figure 3.2: Overview of data organization of MyriadStore. The first level of the meta-data structure can be accessed with the appropriate user's credentials. Having the first level of meta-data in place, the file block list of any backed up file can be retrieved from the DHT, by using the information contained in the storage meta-data. The file block list can then lead to the file blocks of the backed up file, providing information about how to reassemble the file blocks together to get the original file. Note that a file block may be referenced by more than one file block lists.

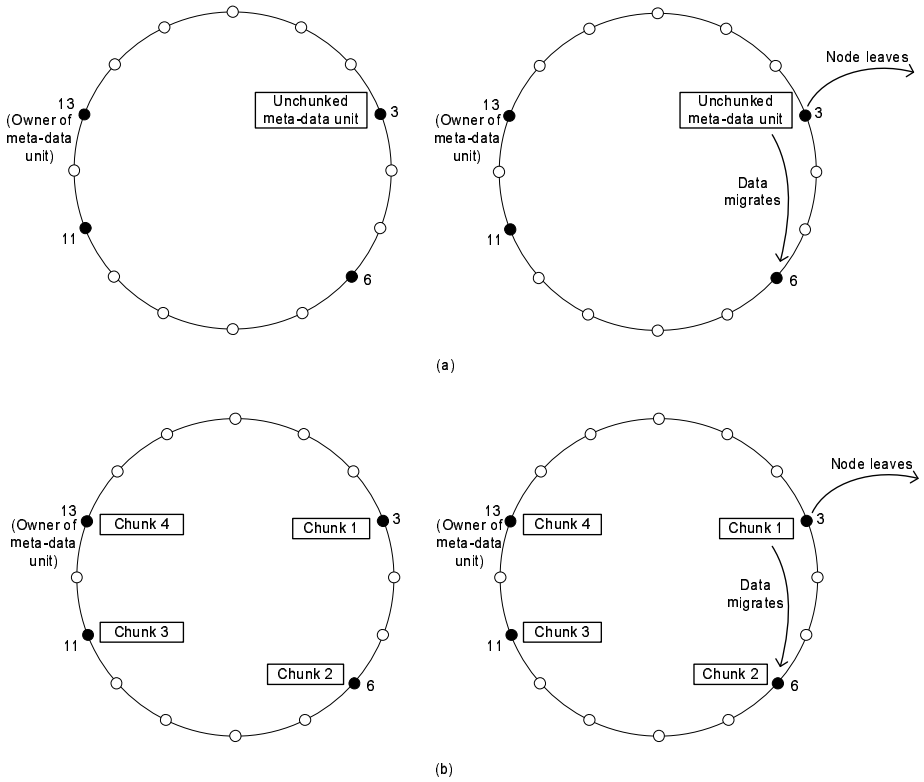

Figure 3.3: By splitting the meta-data into smaller pieces, migrations costs are minimized. Figure (a) shows the case where a meta-data unit is saved as a whole. When the node holding this data leaves the system, the meta-data unit will need to migrate to the successor of the leaving node. On the other hand, when data is split into smaller chunks (Figure (b)), the cost for the migration of data is lower, as the data that migrates is of smaller size.

a single unit would be inefficient. In these cases, meta-data should be split into smaller parts in order to be saved into the DHT (Figure 3.3).

Another advantage that is gained by partitioning the meta-data into smaller parts is that it allows parallel saving and retrieval of the meta-data from the network. Therefore, the processes of saving and retrieving the meta-data can be done more efficiently, as in the case of the file blocks. Moreover, partitioning the meta-data and dispersing it to the network results in load balancing among the nodes.

The splitting of the meta-data will result in the creation of the so called *meta-blocks*. As with file blocks, every meta-block has a fixed maximum size w (currently 4Kb). This size, however, is much smaller than the maximum size of file blocks. File blocks are accommodating actual data that can be very large whereas meta-blocks will not need to carry that much information.

In our approach, a *root meta-block* is associated with each meta-data unit (each instance of the meta-data types described earlier) that needs to be saved into the DHT. If this unit's data is small enough to fit into a single meta-block then the root meta-block holds its data. Otherwise, a sufficient number of meta-blocks will be created to hold this data. In this case the root meta-block contains pointers to these meta-blocks and it might as well contain some data of the meta-data unit itself (Figure 3.4).

3.4 Security

As data items (file blocks and meta-blocks) are stored remotely on nodes that are not trusted, security is a crucial issue that needs to be considered. Data items reside on nodes that do not own them. It is therefore important that these nodes cannot have unauthorized access to this data and that they cannot modify the data. Thus, data items need to be encrypted in a way that only their owner is able to access the data they contain.

Every user has a pair of public and private keys. A user's private key is kept secret but the public key is accessable to anyone. Data that is encrypted with a user's private key can be decrypted using his/her public key and items encrypted using a user's public key can only be decrypted using his/her private key. This encryption scheme is referred to as Public Key Cryptography. Since a user's public key is generally quite large it is not efficient to use the public key to encrypt large blocks of data. Therefore, a symmetric key should be used to encrypt data items.

Since only the owner of a data item should be able to access the data it contains, every data item is therefore encrypted using a symmetric key which is the content hash of the data it contains. The content hash is then encrypted using the user's public key and stored as part of the user's meta-data.

Users' private keys can be used to digitally sign information that should be published and communicated to other nodes on behalf of a user. Other nodes can then verify the origin of the information by using a user's public key. This scheme is generally referred to as Public Key Infrastructure (PKI).

3.5 Availability

As explained in Subsection 3.3.1, meta-data is stored in the DHT of DKS. Therefore, the DHT is responsible for keeping the meta-data available at all times by moving it accordingly as nodes join and leave, as well as replicating it to several nodes.

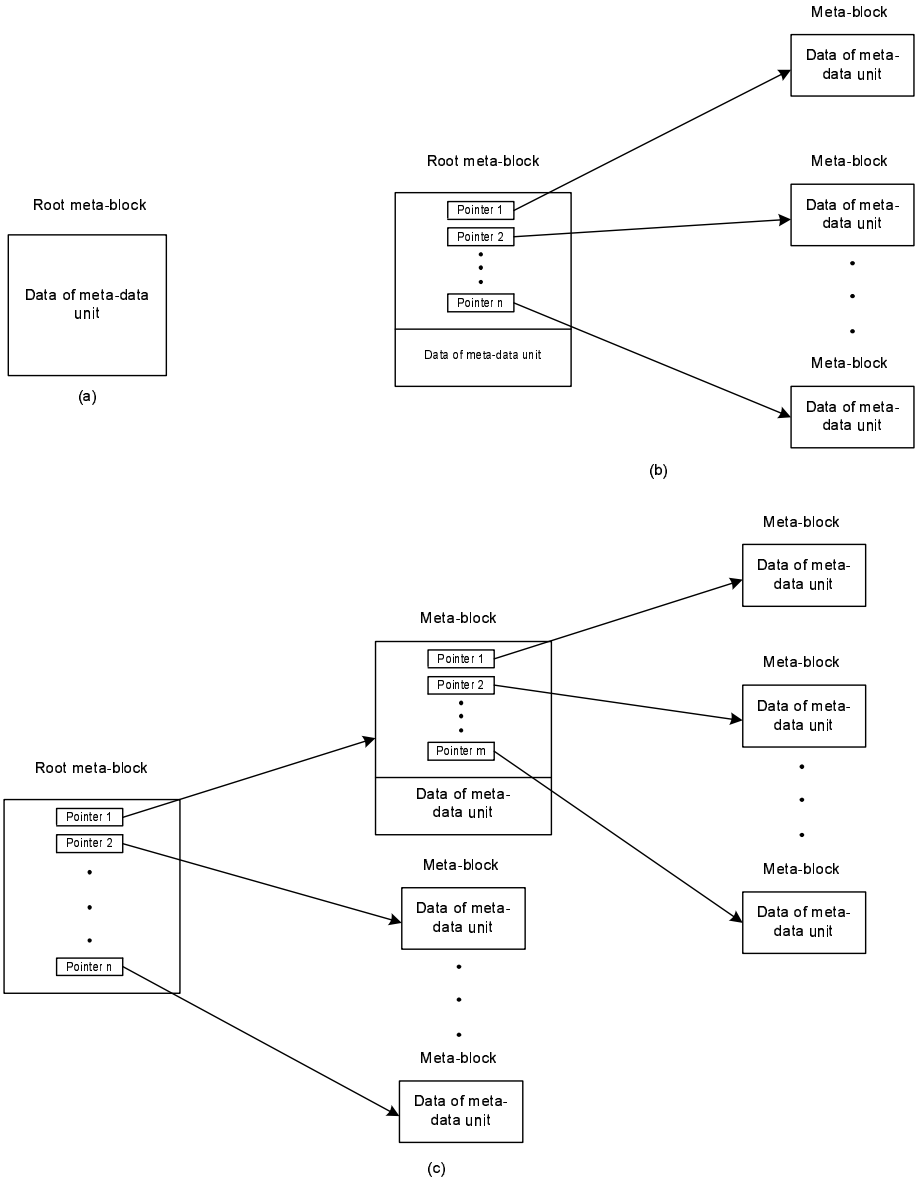

Figure 3.4: The structure of the meta-blocks. (a) shows the case where all the data of a meta-data unit fits into a single meta-block. In this case the root meta-block will hold all this data. (b) and (c) illustrate cases where there is not enough space into a single meta-block to keep all the data of a meta-data unit. In (b), the root meta-block can hold pointers to all the meta-blocks that contain the meta-data. If there is sufficient space, then it holds some meta-data itself. In (c), the root meta-block does not have enough space to hold all the pointers to all the meta-blocks that contain meta-data. The pointers that cannot fit into the root meta-block are moved to other meta-blocks, which are referenced by the root meta-block. This technique is applied recursively if needed.

On the other hand, since file blocks are not stored into the DHT, it is MyriadStore's responsibility to maximize their availability. To deal with this issue, MyriadStore replicates file blocks to several nodes in the network.

If only one copy of each file block existed in the network then in order to be able to retrieve a file it would be necessary that all nodes that keep file blocks of this file are online. If at least one of them was offline it would be impossible to retrieve the file. Using replication, if a node holding a file block is offline then it will be still possible to retrieve this file block from another node. Replication increases the availability of file blocks but naturally induces higher storage/bandwidth overhead.

3.6 The Backup Process

When a user starts a MyriadStore client, he/she should be able to see all the settings for every backup set he/she has created, together with the list of files associated with each backup set. As described earlier such information is not stored locally. It is meta-data that resides on the DHT of DKS. Therefore, when the client is started this information needs to be retrieved. Having it in place allows the user to view all the setting he/she has previously set. The user can then decide whether he/she wants to retrieve some of the backed up files or if he/she wants to backup some others.

When it is time for a file to be backed up, the first thing that needs to be checked is whether the file has been changed since the last backup. To do this check, the content hash[1] (SHA-1) of the file to be backed up is checked against the content hash of the last backed up version of this file (which is obtained from the retrieved meta-data). If the content hashes match then there is no need to perform a backup of the file. Otherwise, a backup is performed.

When the backup operation of some data starts, the first thing to be done is to partition the data into file blocks. As described before, this is done on a per-file basis: each file is taken one by one and it is split into file blocks. Initially the file blocks are stored locally. File blocks are stored with a file name that is derived by their content hash. The next step is to determine where the file blocks should be stored in the network. To do this, some partners need to be found that are willing to store file blocks in their local file systems.

As mentioned in Section 3.2, MyriadStore makes use of a symmetric trading protocol for finding partners. According to this protocol the node that wants to store some file blocks remotely will send requests to randomly selected nodes

[1]The content hash of some data is the result of applying a hash function to this data. This application of the hash function results in some form of a small digital fingerprint of this data. It is unlikely that two different data pieces will have the same content hashes. On the contrary, identical data pieces always have the same content hash.

indicating how much disk space it needs. If a node receives such a request and it is interested in trading then it will reply with an offer of some amount of disk space that is less or equal to the amount requested. If the node that receives the request is not interested in trading, it will simply ignore the message. When the requesting node receives an offer it may choose to accept or reject it. If it chooses to accept, then a contract is established between the two parties and each party allocates to the other an amount of disk space equal to the disk space of the offer. The offering node will be notified of the creation of this contract when he receives the accept message from the requesting node. The requesting node will keep on sending requests and trying to establish contracts with other nodes until it finds all the space it needs for the backup to take place. The trading protocol described is illustrated in Figure 3.5(a). Figures 3.6 and 3.7 give the algorithms for each of the nodes of a trading session.

When the trading session has ended the node making the backup will have allocated as much remote space as needed and this allocation will have been done in terms of contracts. Now the node can start utilizing its remotely allocated space by storing its file blocks on it. For each file block to be stored remotely, an appropriate contract will be found that offers enough space to accommodate it. Having found such a contract, the node that is going to store the file block will have to be notified that a file block transfer is about to take place (Figure 3.5(b)). To do that, an appropriate message is sent including, among others, the contract id of the contract that is being utilized. The receiver of this message can then verify whether the contract that is claimed to be utilized exists and has enough unallocated space left. If these checks are passed successfully, then an accept message will be sent allowing the receiver of it to start the transfer of the file block. After the transfer has been completed a receipt of having accepted the file block is sent to the sender (as explained in Subsection 3.2.4) This process is done for all the file blocks of the data that is being backed up and after it finishes the backup of this data will have taken place successfully. Figures 3.8 and 3.9 give the algorithms for storing and receiving a file block.

As can be seen in figure 3.5, file block transfers do not go through the overlay network. Instead, they are sent directly from node to node using the underlying network. If the overlay network was used for these transfers then data might need to go through several hops in the overlay network in order to reach their destination. Using the underlying network for the data transfers results in more efficient communication as data is sent directly to its destination.

The previous descriptions made it clear that the symmetric trading protocol guarantees that a node should provide as much space as it offers. However, we should say that deviations from strict symmetry may be allowed. Nodes can decide if they want to tolerate some difference between the amount of the disk

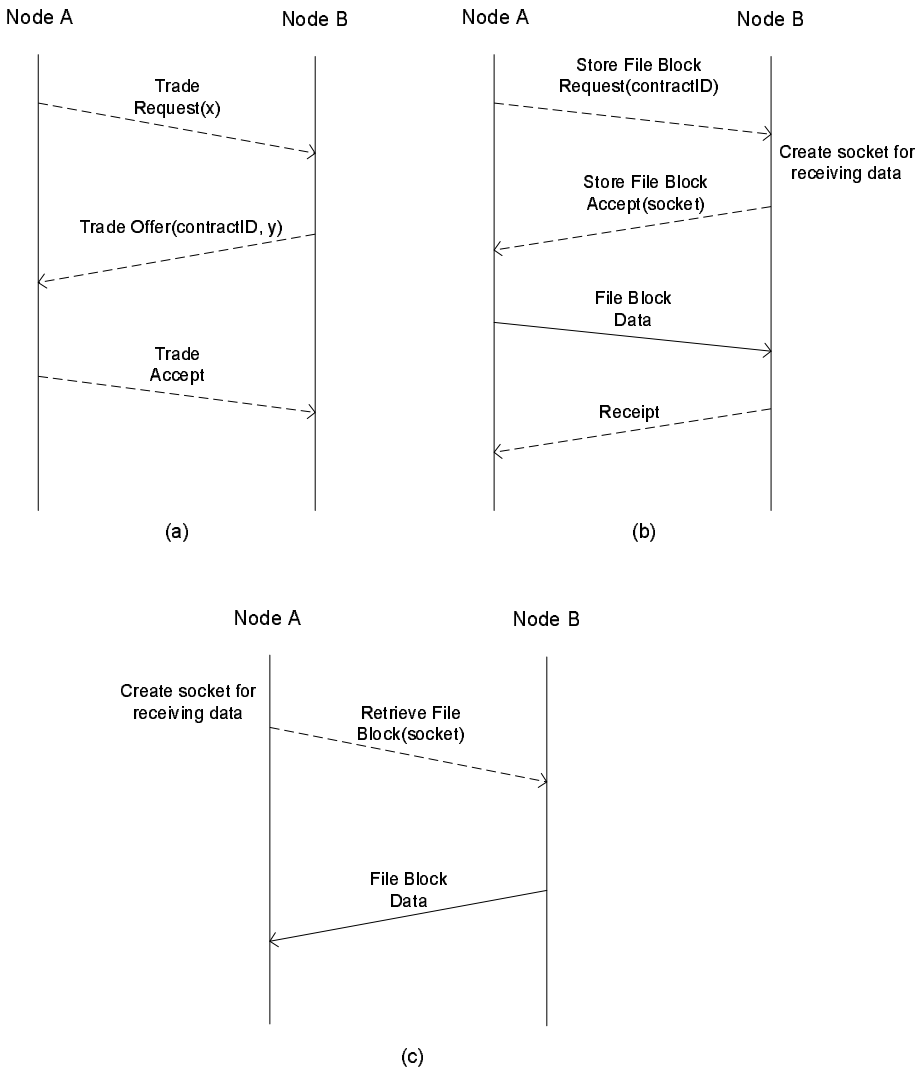

Figure 3.5: Protocols between MyriadStore's nodes. Nodes make use of a symmetric trading protocol to allocate remote space on other peers (a). The offering node is replying to a request specifying some amount of space y it offers, together with a contract ID. Having acquired the remote space they need they can utilize it by sending file blocks (b). The node that wants to send a file block sends the ID of the contract it is utilizing. Finally, remotely stored file blocks can be retrieved to reassemble the backed up data (c). Dashed lines represent communication through the DKS overlay network whereas solid lines indicate direct node-to-node communication without using the overlay network.

```
spaceNeeded;


while(spaceNeeded>0) {
      node = pickRandomNode();
      send(node, tradeRequest(spaceNeeded));
}

                          ||

receive(node, tradeOffer(contract(spaceOffered)));
addPartner(node, contract);
spaceNeeded -= spaceOffered;
send(node, tradeAccept(contract));
```

Figure 3.6: Algorithm in pseudocode for a node that initiates a trading session for finding some remotely available space. The vertical lines that separate the two pseudocodes mean that these two codes run concurrently.
Until a node has located storage space for all the data it wants to store it keeps sending trade requests to random nodes in the system. When it receives a trade offer it records information about the new partner as well as information about the offered contract before sending a trade accept message to the partner.

space they provide and the amount of disk space they acquire.

As discussed in Subsection 3.2.5 nodes can challenge their partners by asking them for parts of file blocks they have been entrusted with. Figures 3.10 and 3.11 give the algorithms for making challenges and responding to challenges.

3.7 The Restoration Process

To be able to recover some data from the system the file block lists that are associated with this data needs to be retrieved from the DHT. Once the file block lists are fetched, the file blocks that constitute the desired data are known. Subsequently, by consulting the contract meta-data, the partners that hold these file blocks can be determined. When the locations of the file blocks have been determined, they can be retrieved, decrypted and reassembled as specified in the file block lists.

To retrieve a file block a node sends a request to a node that is storing the file block, asking the node to send it. Upon receiving a request for a file block the storing node locates the data of the file block in it is local file system and sends it to the requesting node (Figure 3.5(c)). Figures 3.12 and 3.13 give the algorithms for retrieving and sending of a file block.

When all file blocks have been received they can be reassembled and the data recovered. Before file blocks can be reassembled they need to be decrypted as specified in the meta-data for each file block.

```
spaceAllocated; // The size of the over-all space allocated for the storage
spaceAvailable; // The portion of spaceAllocated that has not been traded
localSpaceUsed; // The total space used by local fileblocks

receive(node, tradeRequest(spaceRequested));
if(spaceAvailable > 0) {
       if(requestedSpace <= spaceAvailable) {
              spaceOffered = spaceRequested;
       } else {
              spaceOffered = spaceAvailable;
       }
       proposedContract = createContract(spaceOffered);
       spaceAvailable -= spaceOffered; // Reserve space
       send(node, tradeOffer(proposedContract, spaceOffered));
       if(receive(node, tradeAccept(proposedContract))) {
              addPartner(node, proposedContract);
       } else {
              discard(proposedContract);
              spaceAvailable += spaceRequested; // Unreserve space
       }
} else {
       // If some local fbs exist then with some probability, p, allow trade
       if(pickRandomNumberIn(0,1) <= p && getLocalFileblocks().size > 0) {
              if(spaceRequested <= localSpaceUsed) {
                     spaceOffered = spaceRequested;
              } else {
                     spaceOffered = localSpaceUsed;
              }
              proposedContract = createContract(spaceOffered);
              send(node, to(proposedContract, spaceOffered));
              if(receive(node, tradeAccept(proposedContract))) {
                     addPartner(node, proposedContract);
                     forwardFBs = random local fileblocks to forward;
                     foreach(lfb in forwardFBs) {
                            sendFileblock(node, lfb);
                            localSpaceUsed -= lfb.size;
                     }
              } else {
                     discard(proposedContract);
              }
       }
}
```

Figure 3.7: Algorithm in pseudocode for a node that receives a request for trading.

Upon receiving a trade request, a node checks if some of the allocated space has not been traded. The maximum space it can trade is the size of its untraded space. If the trade request is for more space than the untraded space only the untraded space is offered. When the amount of space to be traded has been determined it is reserved and a proposed contract is sent to the requesting node. If no reply is received the proposed contract is discarded and the space unreserved. In case no space is left untraded, a node will with some probability offer some space to a requesting node. It will free up space for the requesting node by forwarding file blocks stored locally to the partner. Therefore, it cannot forward file blocks it has been entrusted with that are owned by the partner (in case they have previously established contracts). Before being able to receive any data from the partner it needs to forward enough file blocks to make room for the partner's file blocks.

```
partner = partner to which to send;
contract = the contract that will be utilized;
fb = fileblock to send;

while(true) {
       contractId = contract.getId();
       contentHash = contentHash(fb);
       size = fb.getSize();
       send(partner, storeFBRequest(contractId, contentHash, size));
       if(receive(partner, StoreFBAccept(receipt)) {
              send (partner, fb);
              terminate();
       } else if (receive(partner, StoreFBExists(receipt)) {
              terminate();
       } else if (receive(partner, StoreFBDeny()) || timeout)  {
              if(inspect(partner) == OK) {
                     punish(partner);
              }
              contract = findAnotherContract();
              partner = contract.getPartner();
              continue; // restart loop
       }
}
```

Figure 3.8: Algorithm in pseudocode for a node that sends a request for storing a file block.

For storing a file block a node sends a request to its partner containing a reference to a previously established contract between the two as well as a content hash of the file block and the size of it. If it receives an accept reply it proceeds to send the file block to the partner. In case the partner is already storing the file block it will inform the node by replying with en exists message indicating that there is no need to send the file block again. If the partner refuses to store the file block the node can inspect the partner and verify that the file blocks it is storing for the node are indeed correct. Inspection is discussed further in Section 3.8. If the node is unable to store a file block with a partner it will try to find another partner to store the file block with.

```
partner = requesting partner;

receive(partner, storeFBRequest(contractId, contentHash, dataAmount);
contract = getContract(contractId);
if(establishedContracts.contains(contract) &&
                          contract.getFreeSpace() >= dataAmount) {
       if(getLocalFileblocks.contains(getLocalFileBlock(contentHash))){
              send(partner, StoreFBExists(receipt));
       } else {
              send(partner, StoreFBAccept(receipt));
              receive(partner, fb);
              storeLocalFileblock(fb);
       }
} else {
       send(partner, StoreFBDeny());
}
```

Figure 3.9: Algorithm in pseudocode for a node that receives a request for storing a file block.

Upon receiving a request for storing a file block, a node verifies if there exists a contract between itself and the requesting node. It makes sure that there exists enough unused space to accommodate the file block before replying with an accept message. If the node is already storing the same file block it replies with an exists message. If the contract the requesting node refers to is invalid or if it contains no free space the node replies with a deny message. Signed receipts are sent for all file blocks that a node accepts to store.

```
while(true) {
       rfb = random remote fileblock;
       offset = random offset in rfb;
       length = random length between 0 and length of rfb;
       receipt = receipt from partner responsible for rfb;
       expectedResult = getData(rfb, offset, length);
       contentHash = contentHash(rfb);
       partner = rfb.getPartner();
       send(partner, challenge(contentHash, offset, length, receipt));
       if(receive(partner, challengeResponse(data)) &&
          challengeResponse(data) == expectedResult) {
              increaseOpinion(partner);
       } else {
              decreaseOpinion(partner);
              if(getReputation(partner) <= 0) {
                     punish(partner);
              }
       }
       sleep(random);
}
```

Figure 3.10: Algorithm in pseudocode for a node that sends a challenge to a partner.

Periodically a node send challenges to its partners where it asks a partner to produce some parts of a file block it has been entrusted with. If the partner's response matches the expected result the node increases its opinion of the partner which has the effect that the partner's reputation increases. However, if the partner's response does not match the expected result or if the partner does not reply at all, the node decreases its opinion of the partner which has the effect that the partner's reputation decreases. If the partner's reputation crosses a specific threshold for a value of reputation, zero for example, the node will punish the partner by dropping some of the data it is storing for the partner. The challenge mechanism is discussed in detail in Subsection 3.2.5.

```
partner = a node that owns some local fileblocks;
contentHash = content hash of a local fileblock;
offset = random offset in local fileblock;
length = random length between 0 and length of local fileblock;
receipt = receipt for storing of local fileblock;

receive(partner, challenge(contentHash, offset, length, receipt));
if(verify(receipt)) {
       lfb = locateLocalFileblock(contentHash);
       data = getData(lfb, offset, length);
       send(partner, challengeResponse(data));
}
```

Figure 3.11: Algorithm in pseudocode for a node that receives a challenge from a partner.
Upon receiving a challenge for producing some part of a file block, a node verifies that the challenge is valid by inspecting an attached receipt. If the challenge is valid it locates the referred file block and extracts the requested data from it before proceeding with sending the data in a challenge response to the challenging node. In case the referred file block has been forwarded to another partner the challenge is forwarded to that partner.

```
rfb = remote fileblock to be retrieved;
partner = rfb.getPartner();
contentHash = contentHash(rfb);

send(partner, retrieveFBRequest(contentHash));
if(receive(partner, fb)) {
       fb.getData();
} else {
       punish(partner);
}
```

Figure 3.12: Algorithm in pseudocode for a node that sends a request for retrieving a file block.
In order to retrieve a specific file block a node sends a request for retrieval to its partner that has been entrusted with the file block. If the partner does not respond with the data of the file block it is punished as discussed in Subsection 3.2.5.

```
partner = requesting partner;
contentHash = content hash of requested fb;
lfb = local fileblock to be retrieved;

receive(partner, retrieveFBRequest(contentHash));
lfb = locateLocalFileblock(contentHash);
send (partner, lfb);
```

Figure 3.13: Algorithm in pseudocode for a node that receives a request for retrieving a file block.
Upon receiving a request for a file block a node locates the file block locally and sends the file block data to the requesting node. In case the file block has been forwarded to another node as discussed in Subsection 3.2.6 the node needs to contact the partner that is storing the file block.

3.8 Failures

As previously stated any data that is put into the DHT is assumed to be kept there and never lost. Reliable communication between nodes is also assumed.

As was described in Subsection 3.3.3, meta-data is partitioned into small blocks of data (meta-blocks) which have references to each other before storing them in the DHT. In case of a crash occurring in the middle of storing meta-data to the DHT some of the blocks would not be added to the DHT. This would result in incomplete meta-data as some references would point to empty or non-existing blocks.

To avoid this problem meta-data is stored in reverse order so that all blocks that are referenced by any others are stored first. Therefore, the root meta-block is the last block stored in the process of storing a meta-data unit.

Most failures result from protocols not being successfully completed on both sides for two parties communicating. This results in meta-data for the two parties not being consistent with each other. To correct the failures and to synchronize the meta-data a node can perform *inspection* on its partners.

In case of a permanent failure, such as when a node's hard disk crashes and all data previously kept on the disk is lost, all local copies of the node's partners' file blocks are also lost. After a user has recovered from the hard disk crash, reinstalled and restarted the MyriadStore client, the node can recover by performing *introspection* where it determines which file blocks it should be storing locally. Descriptions of inspection and introspection follow.

3.8.1 Introspection

Every time a node is started it performs self-examination to see if the status of its local file blocks is as expected. In order to do the introspection a node has to have loaded its meta-data to know which local file blocks it should be storing. It then proceeds with locating the local file blocks on its filesystem and verifying them by matching their content hash with the content hash recorded in the meta-data.

3.8.2 Inspection

Meta-data should always reflect exactly which file blocks are currently stored in the system and where. Failures may cause meta-data not to be in accordance with what file blocks are actually stored. Therefore, inspection is a way to synchronize meta-data with the actual status of the system.

When a node encounters an unexpected behavior in communication with a partner, for instance, if the partner refuses to store a file block on account of a

contract being fully utilized, the node can perform inspection on the partner. A node performs inspection by requesting a report from a partner that gives an overview of all local file blocks the partner is storing for the node and all remote file blocks the node should be storing for the partner.

If some of the partner's local file blocks are unknown to the node it can request the partner to remove them and therefore freeing up space for other file blocks that it wants to store with the partner.

If some of the partner's remote file blocks that the node should be responsible for are unknown to the node it can validate the partner's claims by verifying the attached receipts for the remote file blocks. If some of those receipts are indeed valid the node requests the partner to send the corresponding file blocks.

Chapter 4

Implementation

In chapter 3 we examined MyriadStore from a high level perspective, giving an overview of the functionalities it provides to the user, giving arguments about why some design decisions were preferred than others and algorithmically describing how MyriadStore's operations work. This chapter examines MyriadStore from a lower-level point of view. It explains MyriadStore's software architecture, describing the software entities that MyriadStore consists of and the responsibilities of each one of them.

4.1 Architecture of MyriadStore

4.1.1 Implementation Goals

The main goals that guided the process of developing MyriadStore were reusability, maintainability and efficiency. The ideas and algorithms on which MyriadStore is based needed to be implemented in an efficient way without overusing the available computing and memory resources. Furthermore, MyriadStore, as any software system, needs to be maintainable so that the process of adding new features to it, as well as the process of improving the features it already includes, can be done easily enough. Lastly, MyriadStore was developed in a way that would make it easy for other applications to reuse some of the useful subsystems it consists of. Having this in mind, the subsystems that could be reused in other applications should be clearly separated from the other parts of the system.

4.1.2 Software layers

In accordance with the three goals that were mentioned previously, MyriadStore consists of three software layers:

- The *User Interface* layer.
- The *Backup* layer.
- The *Storage* layer.

The User Interface layer is the one responsible for the interaction between the user and MyriadStore. This software layer makes accessible to the user all the features MyriadStore provides to him/her. These features need to be presented to the user in a user-friendly way, so that it is clear to him/her how they can be used.

The layer that the User Interface layer directly communicates with is the Backup layer. All the interaction of the user with the User Interface layer is translated into appropriate calls in the Backup layer. Thus, the Backup layer contains all the software entities that are able to handle and realize the user's requests that are handed in by the User Interface layer. For instance, the Backup layer provides operations for creating and removing backup sets, for adding and removing files from a backup set, for scheduling the backup of the files of a backup set or performing it now, for retrieving backed up files etc.

The Storage layer of MyriadStore is responsible for handling the data of the system. This layer undertakes the task of saving and retrieving the backed up data whenever it is requested, as well as challenging the nodes that should be storing the backed up data and punishing them in the case they fail to respond correctly to these challenges. Moreover, the trading protocol is implemented on this layer. Thus, every message that needs to be exchanged during a trading session or during the operations of saving data, retrieving data, and challenging nodes responsible for keeping specific data, is sent and received by this layer of the participating nodes. This layer as well saves into the DHT of DKS the necessary meta-data for being able to perform these tasks. All the tasks that the Storage layer is asked to perform come from the Backup layer, which, in turn, as we explained previously, takes its orders from the User Interface layer.

Figure 4.1 shows the software layers of MyriadStore that we described.

4.1.3 Layers of Communication between Nodes

A point that should be emphasized regards the message and data transfers between MyriadStore nodes. As we saw in MyriadStore's design (Chapter 3) communications between nodes take place according to specific protocols that involve the exchange of specific types of messages. For instance, a node that initiates a trading session sends a specific type of message for requesting trade, a node that receives such a request and is interested in trading its storage space will send a type of message for making an offer etc. In the MyriadStore's

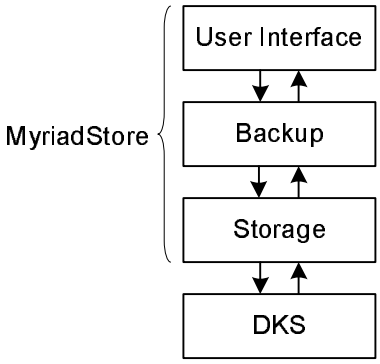

Figure 4.1: MyriadStore's software layers. As can be seen, MyriadStore is built on top of the distributed k-ary system (DKS), and the Storage layer of MyriadStore is the one that directly communicates with DKS.

implementation these messages are sent through the overlay network. In other words, the messages are associated with a recipient and they are handed to DKS. DKS is then responsible for delivering them to the specified recipient by routing them accordingly inside the ring.

On the other hand, the actual data of the backed up files are transferred between nodes through TCP/IP sockets. Since this type of data is usually quite large, using the overlay network for its transfer would be inefficient as, in this case, every data item would need to be routed according to the DKS routing algorithms. Such routing would result in a data transfer of multiple hops in the overlay network. Instead, having a direct TCP/IP link between the nodes will result in a quicker transfer. Figure 4.2 shows the two kinds of transfers we explained.

4.1.4 Layer Reusability

As we explained before, one of the goals of MyriadStore's implementation was to clearly separate from each other entities that could be useful in another context. The separation of the Backup layer from the Storage layer was done according to this criterion. The Storage layer is a software entity that is given some data, partitions this data into smaller pieces and spreads them into the network. This layer has access to all the information about where these pieces are located and how to reassemble them to get the original data. Clearly, the Storage layer is a software entity that could be used in a peer-to-peer distributed file system application as well, since this type of application also requires data dispersal into peer-to-peer network. Thus, the Storage layer of MyriadStore can be the storage infrastructure of a distributed peer-to-peer file system application. Using this infrastructure, the only additional component that is required for building such

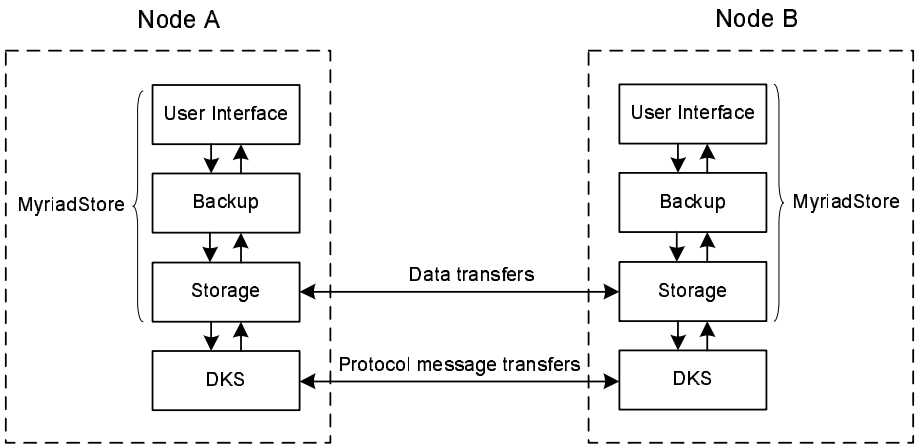

Figure 4.2: The communication between two nodes in MyriadStore. All protocol messages are sent using the DKS infrastructure (overlay network). On the other hand, data are sent directly from node to node without using the overlay network.

a distributed peer-to-peer application is a file system client that will present in a user-friendly way all the necessary information about the files that have been stored in the distributed file system.

4.2 Software Packages

MyriadStore was implemented using Java 1.5. In this section we will present the packages that implemented the layers of MyriadStore's architecture, by listing the main classes that they consist of and describing their main responsibilities.

4.2.1 The *ui* Package

The *ui* package of MyriadStore implements the User Interface layer. It consists of two classes: the *LoginDialog* and the *BackupClient*. The *LoginDialog* is the entity that handles the user authentication process. The *LoginDialog* calls the *BackupClient* by specifying some user identification information and the *BackupClient*, which is the MyriadStore client that the user interacts with, fetches and displays the appropriate meta-data of the user. After the authentication process has been completed, the user can use the *BackupClient* to interact with MyriadStore: set the desired settings, backup files, restore files etc.

4.2.2 The *backup* Package

The *backup* package implements the Backup layer of MyriadStore's architecture. The main entity of this package is the *BackupManager*. The *BackupManager* keeps track of the meta-data that record the user interaction with MyriadStore: which backup sets the user has created and which files each one of them contains, information about the backups that have already been performed, the scheduling setting of the backup sets etc. The *BackupManager* calls the appropriate objects in the Storage layer for saving and retrieving from the DHT the appropriate meta-data. The *Repository* class of the *backup* package is the one that implements the notion of the backup set. The *BackupManager* is the entity that creates *Repository* objects, according to the user's requests. Each *Repository* is associated with a *BackupScheduler*, an entity that makes sure that the schedule associated with its corresponding *Repository* is followed.

4.2.3 The *storage* Package

The Storage layer of MyriadStore is implemented by the *storage* package. One of the basic entities of this package is the *StorageManager*. The *StorageManager* is orchestrating many of the operations that take place in this layer. The *StorageManager* accepts from the Backup layer request for storing some files remotely. The *StorageManager* partitions these files' data into file blocks and hands in to the *StorageDaemon* a request for storing these file blocks. The *StorageDaemon* will try to find space in the already established contracts that can accommodate these file blocks. If it fails, then it will request from the *TradeDaemon* to find some available remote space by establishing contracts with other nodes, using the trading scheme we described in Section 3.6.

The *StorageManager* is also the object that is responsible for saving and retrieving from the DHT the meta-data that references the file block lists of the files that are being backed up. Furthermore, the entity that is assigned with the task of saving and retrieving from the DHT the meta-data regarding the contracts is the *ContractManager*. This meta-data contains all the necessary information about the contracts that have been established with partners. As explained in chapter 3, this meta-data together with the file block lists contain the required information for locating the backed up files' data in the peer-to-peer network.

The task of challenging partners that are supposed to keep specific data is done by the *ChallengeDaemon*. This object makes challenges periodically and punishes the challenged partners if they fail to reply correctly.

Lastly, the *DKS* class and the *MessageDaemon* class are two important entities of the storage package that should be explained. The *DKS* is the class

that acts as an intermediary between MyriadStore and DKS. This class makes the appropriate calls of the DKS middleware so that a node can join or leave a DKS ring. Furthermore, all calls for saving and retrieving data from the DHT are passed to the DKS through this class. The *MessageDaemon* is the entity from which all protocol messages between nodes are sent. The *MessageDaemon* calls the appropriate methods of the *DKS* class, which in turn calls the corresponding DKS methods.

4.2.4 The *util* Package

The *util* package contains some utility classes that play the role of useful tools that the MyriadStore's entities we described before make use of. For instance, the *Hasher* class of this package provides methods for deriving the content hash both in a string encoded format and in the normal byte format. Furthermore, in this class there are different methods provided for different kinds of forms of the data (such as file, input stream, array of bytes, string) of which the content hash needs to be computed.

Table 4.1 summarizes the main classes of the software packages of MyriadStore.

Package	Main classes and their responsibilities
ui	*LoginDialog*: manages User Authentication *BackupClient*: the MyriadStore Client
backup	*BackupManager*: orchestrates the operations and manages the meta-data of the *Backup* layer *Repository:* implements the notion of the backup set *BackupScheduler*: manages the schedule of backups
storage	*StorageManager*: orchestrates the operations and manages the meta-data of the *Storage* layer *StorageDaemon*: manages the data transfers *TradeDaemon*: establishes contracts with other nodes *MessageDaemon*: sends protocol messages *ChallengeDaemon*: challenges partners *ContractManager*: manages the contract meta-data *DKS*: hands in requests to the DKS middleware
util	*Hasher*: computes the content hashes of data

Table 4.1: The software packages of MyriadStore and the main classes they contain.

Chapter 5

Related Work

Peer-to-peer networks, and particularly the structured ones, offer a nice infrastructure on which distributed applications can be based. The efficiency in locating data and the ability to adapt to a constantly changing environment are certainly desired properties that distributed applications could take advantage of. Distributed backup and distributed file system applications are two categories of such applications. Several systems have been proposed for either of these categories. Section 5.1 explains the differences between the two types of applications. Sections 5.2 and 5.3 give an overview of some of the most representative systems for each category. Section 5.4 compares the systems described in Sections 5.2 and 5.3. Finally, Section 5.5 mentions some additional noteworthy approaches in distributed backup and distributed file systems.

5.1 Distributed Backup versus Distributed File System Applications

Distributed backup applications are similar to the distributed file system applications in the sense that they both have to do with data storage. They are both applications responsible for storing a user's data remotely and providing the option of retrieving this data whenever it is needed. However, there are significant differences between them that should be pointed out.

One main difference between the two classes of applications regards the number of users that are entitled to access the stored data. A backup is a process that involves storing data for safe-keeping so that they can be retrieved in case they cannot be accessed from the location they normally reside. By this description it is clear that providing concurrent access to the backed up data to multiple users is not a needed feature. Only one user should have the right to

read or write a particular data item that is stored remotely. On the other hand, this is not the case for distributed file system applications, a class of applications the main responsibility of which is to provide a common file system to a group of users so that they can work simultaneously with it. Concurrent reads and writes of the stored data is an issue that distributed file system applications should be able to deal with. This makes these applications more complex to build, in comparison with the backup applications.

The fact that in distributed file system applications data is accessed by many users imposes security issues that add even more complexity to them. The users might have different rights regarding the reading, writing, appending and creating data, and there might be users that have no rights in accessing specific data. All these security issues regarding the sharing of data between the distributed file system's users need to be handled by the application.

Another issue that differs in the two classes of applications is the frequency of accessing the stored data. A backup, by definition, is not expected to be accessed very frequently. The backup of some data will only be accessed in the exceptional case that the data itself becomes unaccessible. On contrary, the data of distributed file system is expected to be accessed much more often. Therefore, the performance requirements are more strict in the latter case.

5.2 Distributed Backup Applications

The following subsections discuss the approaches of four representative distributed backup systems: pStore [6], PeerStore [4], Pastiche [5] and the Cooperative Internet Backup Scheme [9].

5.2.1 pStore

Overview

pStore [6] is a secure distributed backup system that brings together developments in peer-to-peer storage with those in the domain of data backup and versioning. pStores uses a distributed hash table (DHT) to store all the backed up data, together with all the meta-data that are needed to locate the backup data inside the peer-to-peer network.

The three primary design goals of pStore are: reliability, security and resource efficiency. Reliability is provided through replication of data where copies are available on several different nodes in different locations, in case some of these nodes become unavailable or are malicious. Security is ensured by using encryption and content hashes. Private data is therefore only readable by its owner, who is also the only one that can delete the data remotely. Furthermore,

the owner can easily detect if his/her data was tampered with. As backups can be frequent and large, pStore aims to reduce resource-usage by sharing stored data and exchanging data only when necessary.

The next subsection gives a brief description of how these three primary design goals are achieved.

The backup process in pStore works in the following way: When a user wants to insert a file, pStore computes an identifier that is specific for that file and does not conflict with other files or other users files. The file is encrypted and broken into blocks that are digitally signed, and signed meta-data is assembled which indicates how to reassemble the blocks. Both the meta-data and the blocks are inserted into a peer-to-peer network. To retrieve a file, a user specifies which file and which version of the file, or browses the pStore directory hierarchy. Once the meta-data has been retrieved, all the blocks belonging to the file can be retrieved and assembled. Their signatures can be used to ensure integrity.

Data Structures

A pStore file is represented by a file block list (FBL) and file blocks (FB). Each FB contains a portion of the file's data. The FBL maintains four pieces of information for each FB: a file block identifier used to uniquely identify each FB, a content hash of the unencrypted FB, the length of the FB in bytes, and the FB data's offset in the original file. The FBL contains an ordered list of all the FBs in the pStore file. Different versions of a file are maintained by additional ordered lists of FBs in the FBL for each file version. Unchanged FBs between versions are simply referenced in later versions therefore saving storage space and network traffic (Figure 5.1). New or changed parts of the file are broken into FBs of appropriate size, referenced in the FBL, and inserted into the network. To determine which FBs have changed between the versions and which not, an adaptation of the rsync algorithm [12] is used. Using this algorithm there is no need to retrieve the actual FBs from the network. Instead, it uses the content hash, the length and the offset information of the FBs to determine which have changed and which not. The advantages of this versioning scheme is that now whichever version of a file can be reconstructed from a pStore file with no duplicate FBs, and a corruption of a FB leaves versions that do not include that FB intact. Using this versioning scheme, pStore maintains snapshots for each file allowing a user to restore any snapshot at a later date. File attributes such as permissions and date of creation can be stored in the FBL. pStore also allows files to be grouped into directories. A pStore directory is simply a text file containing a list of the names of files and subdirectories contained within the directory. The same versioning scheme is used for pStore directories as for

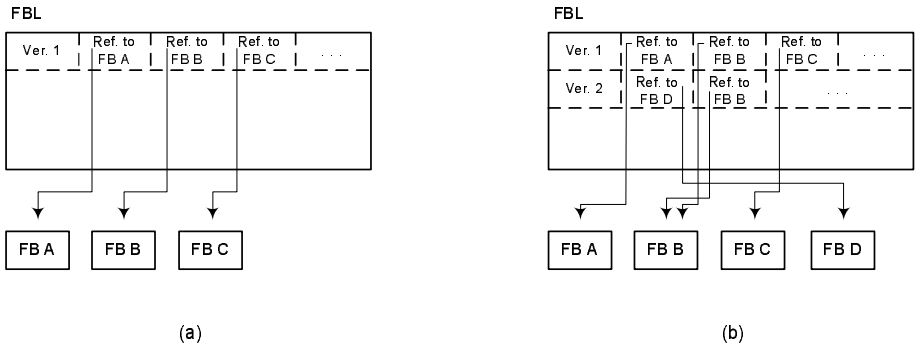

Figure 5.1: (a) shows a file block list which associates a version of a pStore file with its file blocks. The file block list of the version contains references to the file blocks in the network. In (b) the file block list holds references to the file blocks of the new version. Any previous file blocks that did not change in the new version (file block B) are simply referenced in the file block list of the new version. Thus, there is no need to be saved to the network again.

normal files.

FBLs and FB are encrypted with symmetric keys to preserve privacy. FBs are encrypted using convergent encryption which means that a hash of the unencrypted FB contents is used as the symmetric key when encrypting the FB. To make more clear how this works, let's consider for example how the encryption procedure would take place for a block b. First, the hash of the actual content of the block should be derived; let's call this hash C. Then the block is encrypted by using C as the symmetric key. Then the hash of the encrypted block should be computed; let's call it D. The encrypted block should be then stored into the peer-to-peer system DHT under the key D. By using this method, two identical blocks will have the same id and, therefore, will only need to be stored once (D of course, together with C, needs to be stored in the file block list). This makes block sharing between different users feasible as all users with the same FB will use the same key for encryption. All FBLs for a given user are encrypted with the same symmetric key derived from a user's private key, simplifying key management considerably.

FBs and FBLs are treated the same in insertion and retrieval from the peer-to-peer network. Both can be viewed as a data chunk containing an identifier which can be used to retrieve it from the network, public meta-data which is signed with the owner's private key, a digital signature and the actual data. The owner's public key is included with every data chunk which allows anyone to verify and view public meta-data, but only the owner can change it. For the FB identifier pStore uses a hash of the encrypted FB contents. This makes it easy for one to compare a rehash of the data chunk to the identifier to verify

the data has not been tampered with. The public meta-data for a FB chunk contains this identifier and is used for authentication when deleting FB chunks.

To eliminate unwanted FBL chunk identifier collisions between different users, each pStore user has a private namespace into which all files are inserted and retrieved. The private namespace is created by a private/public key pair. Users can have multiple namespaces and users can share the same namespace.

Replication

pStore uses exact-copy chunk replication to increase the reliability of a backup. Chunks are stored on several different peers in different locations and if one fails then the chunk can be retrieved from any of the remaining peers. The chunk replicas are randomly distributed throughout the identifier space and replicas are sent directly to the target nodes.

Deletion

A user can request to remove a file from pStore using an explicit delete operation. pStore uses the public meta-data associated with each FB to authorize an owner to delete a chunk. Each storage node keeps an ownership tag list (OTL) for each chunk. An FB chunk may have many ownership tags but each FBL chunk should only have one ownership tag. When a user wants to remove a chunk from pStore he inserts an empty chunk with the same identifier as the chunk he wants to delete. pStore compares the public keys between the ownership tags in the OTL and removes the appropriate ownership tag from the OTL. When the OTL becomes empty pStore removes the chunk.

5.2.2 PeerStore

Overview

PeerStore [4] is another peer-to-peer backup system. PeerStore is different from its preceding backup systems as it decouples the meta-data management from the actual backup data storage. This strategy results in several advantages:

- Different strategies can be employed so that the different needs of the two subsystems are met.
- The storage layer can be implemented in a way that achieves fairness.
- The meta-data layer can perform quick searches for replicas of a block of data in the network.

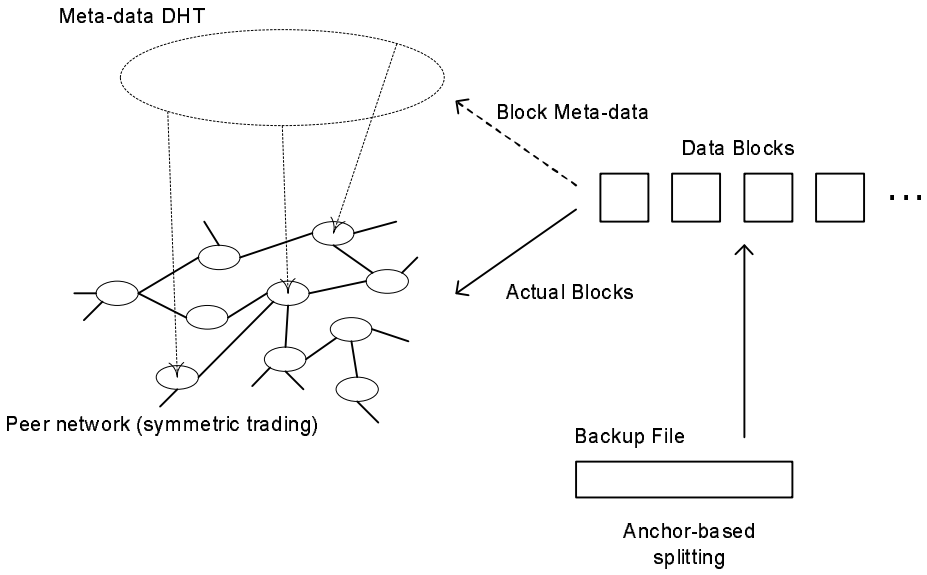

Figure 5.2: An overview of how PeerStore works. A file to be backed up is first split into blocks of actual data. Together with these, blocks of meta-data are created to facilitate finding the actual data blocks inside the network. The blocks of meta-data are stored into the meta-data DHT whereas the blocks of actual data are stored directly into the network's peers.

– The meta-data is relatively small, so it can be maintained aggressively to keep the information up-to-date.

The meta-data layer is based on a DHT since it should be able to provide fast searching, an important feature for duplicate detection. Adapting this design, no real data needs to be migrated when nodes join and leave the network. Only the information of the meta-data records needs to be transferred and updated. This saves the maintenance cost in a high degree. On the other hand, the storage layer is based on a symmetric trading scheme. That means that a peer that wants to backup its data must also be willing to store some data from each of its trading partners. This approach offers flexibility, avoids the high maintenance cost of a DHT and eases the addition of a fairness mechanism.

Using these two layers, the files that are to be backed up will be split into blocks and subsequently they will be encrypted and stored at partners using the symmetric trading scheme. A meta-data record maintained for each block will facilitate their later retrieval. These meta-data records are stored in the *Metadata DHT*, which offers efficient duplicate checking and fast searching of any required data block (Figure 5.2).

Design

There are four major design aspects in PeerStore: making the backup, restoring the backup, fairness and short-term versus long-term availability.

The backup procedure starts by splitting the backup files into data blocks, each having a unique block identifier. The data is then encrypted. Splitting and encrypting the blocks is done in a way that enables peers to share identical data in backups. With this technique each file is represented as a list of unique block identifiers. To avoid wasting storage space, blocks with a sufficient number of replicas in the network are eliminated (with the help of the meta-data DHT). Finally, trading partners are found and data is backed up with an efficient number of replicas.

To restore a backed up file, a peer will first need to obtain the list of block identifiers for each version of the file and then determine the keepers of each block (with the help of the meta-data layer). Subsequently, it can download the blocks from the keepers, decrypt them and get the original file.

There two aspects regarding fairness in a peer-to-peer data storage network: safekeeping and fair contribution. PeerStore is concerned with the former as the symmetric trading scheme takes care of the latter. The idea proposed in PeerStore to ensure safekeeping is to regularly challenge every trading partner to ensure that it is still storing the blocks entrusted to it. If a partner fails to answer a challenge then he is punished by removing some data blocks of his backup.

Finally, PeerStore mainly focuses on providing high long-term availability instead of short-time availability because the latter requires much more resources from the system. However, this lazy strategy could result in possibly long waiting times for a restore operation to take place. Another problem is that there is no way to decide whether a partner is temporarily off-line or has permanently left the network.

To conclude, the primary goal of PeerStore is to provide peer-to-peer backup with low maintenance cost while supporting heterogeneity. All the preceding related works suffered from a high maintenance cost in unstable networks. This is because whenever a node was joining or leaving the network the data stored on affected nodes needed to migrate to make sure that they are stored on the correct responsible node. This resulted in great migration cost. PeerStore is trying to reduce this cost by relaxing the strict guarantees regarding data placement that a DHT imposes. This seems a great improvement as according to simulations analyzed in [4] data migration cost dominates total maintenance cost. Another simulation given in [4] comparing the performance of pStore [6] and PeerStore shows that while the two approaches result in similar backup traffic,

their difference regarding the maintenance cost they require is quite big. This is because of the much lower migration cost imposed by PeerStore, as we explained earlier.

5.2.3 Pastiche

Pastiche [5] is a backup system that is based on three enabling technologies to achieve its goals:

1. *Pastry* [17], a peer-to-peer network that provides scalable, self-administered routing and node location.

2. *Content-based indexing*, a mechanism for finding common data among different files.

3. *Convergent encryption*, which is a type of encryption that allows hosts to use the same encrypted representation for common data without sharing keys.

According to Pastiche, each node that wants to make a backup of its data will first have to find a set of appropriate *buddies*, that is, other nodes that share with it a significant amount of data. As a rule of thumb, each Pastiche node should maintain five buddies. According to Pastiche's design, a node should be able to restore its data from *any* of its buddies whenever it wants.

Pastry

Pastry is an overlay peer-to-peer network that provides efficient routing and object location techniques within its nodes. The main objectives that Pastry aims at satisfying include scalability, self-organizing and robustness.

Pastry virtually lays out its nodes in a ring topology. It uses an identifier space for assigning identifiers to its nodes, in the same way as we described in Subsection 2.2 about the distributed hash tables. In this way, every Pastry node is assigned a *nodeId* and the nodeIds are uniformly distributed over the nodeId space. Every Pastry node is aware of its *proximity* to every other node of the Pastry network, a metric that typically depends on the network costs of communication between two nodes of the Pastry network.

Each node n in the Pastry network maintains a *leaf set*, a *neighborhood set* and a *routing table*. The leaf set consists of L nodes. $L/2$ of these nodes have nodeIds that are the numerically closest smaller nodeIds of n's nodeId and the other $L/2$ of these nodes have nodeIds that are the numerically closest larger

nodeIds of n's nodeId. The neighborhood set of n consists of those M nodes that are closest to n according to the proximity metric[1].

The routing table of node n contains as many rows as the number of digits that a nodeId can have, and each row holds as many entries as the number of possible values that a digit can have. The first row of the routing table contains the IP addresses of the nodes whose nodeId differ from n's nodeId in the first digit. The second row contains the addresses of those nodes whose nodeId match with n's nodeId in the first digit and they differ on the second. Generally, the i^{th} row will contain the addresses of the nodes whose nodeIds differ from n's nodeId in the first $i-1$ digits and they match on the i^{th} one. It may be the case that more than one nodes have the appropriate prefix in their nodeId to fill a specific position of n's routing table. In this case the node to fill this position is chosen to be the one which is closer to n according to the proximity metric.

Data items that are inserted to the Pastry network are associated with a key which is taken from the nodeId space. In Pastry the node responsible for holding a particular data item is the one whose nodeId is the numerically closest to the key of the data item. The routing technique that is used to accomplish this is called *prefix routing*. According to prefix routing, when a node n needs to route a data item, it forwards it to the node whose nodeId has a matching prefix with the data item's key that is at least one digit longer than node's n matching prefix with the data item's key. If no such node is known, the data item is forwarded to a node with an identical to node's n prefix but that is numerically closer to the key of the data item. If the key of the data item falls within the range of nodeIds covered by the leaf set nodes then it is delivered to the node of the leaf set that has a nodeId numerically closer to the key of the data item. Using this prefix routing scheme, the expected number of routing steps is $\log N$, with N being the number of nodes in the Pastry network.

Storage and Encryption of Data

Backed up files in Pastiche are stored on disks as chunks. How these chunks are created is determined by content-based indexing. Content-based indexing identifies boundary regions called *anchors* [20] inside the files to be backed up, and anchors divide the files into *chunks*. Anchors are determined using Rabin fingerprints [21].

Chunks are encrypted using convergent encryption. Using this, when different nodes have identical chunks to backup, they do not need to store them separately. Instead, chunks can be stored only once and nodes can then share them. Each chunk is hashed and the result of this hashing is called the chunk's

[1]Although the neighborhood set was introduced in Pastry's initial design, this notion was removed for simplicity reasons in later work of the Pastry group.

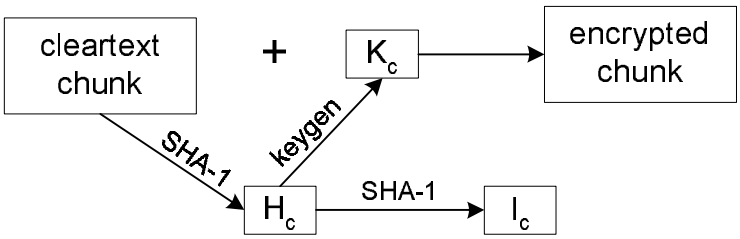

Figure 5.3: Encryption and naming of Pastiche chunks. The handle H_c of a chunk is produced by hashing it. The handle then produces the symmetric encryption key K_c. Also, the handle is hashed once again to generate the chunkId, I_c, of the chunk. Storing of the chunk takes place after it has been encrypted with K_c, and the stored chunk is named by I_c.

handle, H_c. For each chunk, a symmetric encryption key, K_c, is generated by using the chunk's handle. The handle is then hashed once again and this results in the public chunkId, I_c, of the chunk. Each chunk is encrypted using K_c and is named by I_c before stored on disk (Figure 5.3).

A file's meta-data contains a list of handles of the chunks of which this file consists and information about ownership, permissions, creation, modification dates etc. By the handle list, the decryption key and the chunkId of each chunk can be derived. Meta-data is not chunked because it is typically small and the likelihood of being shared is low. Meta-data is written to the disk as actual data are.

Finding buddies

Pastry is the system used for finding a node's buddies. The notions of a *signature* and an *abstract* are essential for this procedure. Signature is defined as the list of chunk IDs that describes a node's current file system. The idea behind this is that a node can send its signature to others in order to find appropriate buddies. However, signatures can become quite large, making their usage inefficient. To solve this problem an abstract is sent instead. A node's abstract is a random subset of its signature.

There are two mechanisms that facilitate buddy discovery: *lighthouse sweep* and *coverage-rate*. The former suggests that a node that wants to find a set of suitable buddies will have to route its abstract to another node whose ID was randomly generated. Each node encountered on this route will compute its *coverage* - the fraction of chunks in the abstract that this node stores locally - and return it. If this initial probe does not produce a sufficient candidate set of buddies then it will be repeated until it does. However, in every subsequent probe the first digit of the original node's id, that is the node to which the

abstract is routed, will vary in its first digit. This assures that the sets of nodes visited in every probe will be disjoint[2]. If the mechanism of lighthouse sweep fails to find an appropriate set of buddies for a node, then the coverage-rate overlay will have to find one. This mechanism uses file system overlap as the distance metric rather than network hops (that were used during the lighthouse sweep). With this technique a node will choose a buddy from its Pastry neighborhood set that has the best coverage available.

Unsolved problems - Evaluation

Two main problems that need to be efficiently solved are: (a) Detecting malicious buddies unwilling to safely keep a node's backup and (b) Coping with greedy nodes that aggressively consume space by using storage on many hosts without ever retiring any of them. Regarding the detection of malicious buddies, Pastiche uses a probabilistic mechanism according to which whenever a node is making a backup of its filesystem, it asks its buddies for a random subset of chunks they should be storing. If a buddy fails to reply with the requested data then the node will remove it from its buddy list and it will initiate a search for finding another one. On the other hand, coping with greedy nodes is the most challenging problem faced by Pastiche. A distributed quota enforcement mechanism is needed that will allow a node to occupy only as much space as it contributes. The solutions suggested so far for this problem are not completely satisfactory.

Finally, simulations on Pastiche (that are given on [5]) have shown that the burden it imposes to the filesystem is not too big. Furthermore, they confirm the effectiveness of node discovery, and analysis shows that detecting malicious nodes requires only few resources.

5.2.4 Cooperative Internet Backup Scheme

The Cooperative Internet Backup Scheme [9] proposes an Internet-based backup technique using a decentralized peer-to-peer scheme that stores backup data on participating computers' hard drives. To ensure high reliability, participating computers pair up with partners multiple times to swap equal amounts of disk space. Partners are preferably in different geographical locations. As a backup needs change, partners may be added or removed. Partners can be changed if they are found to be absent in the system (e.g., due to excessive downtime) or if a new computer needs partners. Partners agree to an amount of storage space that will be swapped and a level of uptime they have to fulfill. Different pairs of partners can make different agreements. A maintainer of a computer

[2]Due to Pastry's prefix property.

is responsible for maintaining a list of partners for that computer that will be used in case of a disaster.

A centralized matchmaker keeps track of the computers in the system and is used as a resource for finding partners. Computers inform the matchmaker about the partners they need and have, including information about uptime and storage-swapping levels. When a computer needs a new partner it queries the matchmaker for possible matches and then proceeds to contact resulting candidate partners to see if they are still available. When there are no other computers looking for partners a computer looking for a partner has to step between two existing partners that have an agreement similar to the one it desires.

A highly-reliable logical disk is constructed from a large number of partners, using Reed-Solomon erasure-correcting codes [18]. Each computer can decide for itself how to tradeoff reliability against overhead of using the erasure-codes. A logical disk contains enough space to hold two full snapshots so that a crash while writing a new snapshot still leaves the system with a viable backup. Logical disks are treated as a circular tape where each snapshot is written right after the next wrapping around at the end of the disk. The format of each snapshot, which is similar to archival formats such as tar, makes it possible to start reading a snapshot in the middle and still extract all the files that come after that point. In case of a crash, when a previous full snapshot has been partly overwritten by the next snapshot, it is still possible to read and recover all the complete files in both partial snapshots.

Finally, for handling the issue of trust between partners, each computer periodically challenges each of their partners to make sure that they are holding the data they agreed to hold. Partners that are down too often or that do not answer or fail when challenged are replaced by new partners. A grace period of two weeks is given before abandoning a partner. A quota is imposed on how many reads or writes a partner can do per day.

5.3 Distributed File System Applications

The subsections that follow give an overview of three important distributed file system applications: OceanStore [7], the Cooperative File System [19], and PAST [8].

5.3.1 OceanStore

An interesting approach oriented towards persistent storage in general is OceanStore [7]. OceanStore is a global persistent storage utility that guarantees a number of

desirable properties, namely, fault-tolerance, availability, consistency and durability. Users of this storage service are expected to pay a monthly fee in exchange for access to the persistent storage.

The fundamental units in OceanStore are *persistent objects* that are named by globally unique identifiers and can only be modified through updates. Objects are not bound to a particular machine but can move freely from machines to machine . This concept of OceanStore is called *nomadic data*. Objects are replicated and dispersed to multiple servers around the world to provide both durability and availability in presence of network partitions, failures or attacks. A responsible party is appointed which is financially responsible to keep replicas consistent by carrying out management protocols on users' behalf.

Oceanstore provides *deep archival storage* which assures that objects survive in the system despite numerous failures. This is achieved through using *erasure codes*. With this approach, an input file is treated as a series of fragments which are transformed into a greater number of fragments where the original file can be reconstructed using any sufficiently large subset of fragments. This is due to redundant information being added to each fragment. Fragments are distributed over multiple servers ensuring that a file can be reconstructed in presence of regional disasters.

Regarding security, it is assumed that the infrastructure of OceanStore is fundamentally untrusted and therefore all data that enters the infrastructure should be encrypted. Furthermore, OceanStore tolerates Byzantine failures by placing a small set of replicas in charge of updates to an file. Primary replicas run a Byzantine agreement protocol and distribute the results to all other replicas in the system. Unauthorized writers are prevented similarly.

To locate objects, OceanStore uses a probabilistic algorithm that is backed up by a slower deterministic algorithm. The probabilistic algorithm locates entities that are in the local range, whereas a distributed data structure (similar to the one introduced in [28]), which stores the location of each OceanStore object, is used to locate entities that can not be found locally.

OceanStore is designed to provide service on a global scale to roughly 10^{10} users, and to support over 10^{14} files. This requires the system to be highly scalable. Oceanstore uses *promiscuous caching* for achieving locality of data, which means that data can be cached anywhere, anytime. Although promiscuous caching complicates data coherence and location, it provides great flexibility to optimize locality and to trade off consistency for availability. Discovery of relationships between objects through *introspective* monitoring is used for locality management. That way, data can seamlessly be pro-actively migrated to wherever it is needed to enhance performance and load balance the system. This freedom of data provides maximum flexibility in selecting policies for replication,

availability, caching and migration.

Lastly, OceanStore employs an update model based on conflict resolution which supports a range of consistency semantics - up to and including ACID semantics favored in databases. Changes to objects are made through updates, which are lists of predicates associated with actions. An update is applicable if any of the predicates evaluates to true. As noted before, a primary set of replicas are responsible for choosing the order for updates. They cooperate with one another in a Byzantine agreement protocol and communicate the result to all other replicas.

5.3.2 The Cooperative File System

The Cooperative File System (CFS) [19] is a peer-to-peer read-only storage system. The main software layers of each CFS client are:

- *FS*. This is the layer responsible for interpreting data blocks as files. In this way, it presents a file system interface to any higher level applications.
- *DHash*. This is a distributed hash table implementation. With this layer data blocks are stored in the various CFS servers.
- *Chord* ([16]). This is the low level layer that DHash uses in order to determine where the data blocks should be stored and from where they should be retrieved afterwards. In other words, Chord is responsible for the routing of the data in the network[3].

A CFS server includes the last two layers, without including the first one. The above layered structure implies that a CFS file system is spread over available CFS servers in the form of data blocks. The DHash uses Chord to determine where the data blocks should be stored and from where they should be retrieved, and the FS layer of the CFS client is responsible for interpreting and presenting these data blocks as file system data.

Some characteristics of the design of CFS are: a) it uses public keys or content hashes to authenticate data b) it doesn't provide any search mechanism c) it doesn't provide anonymity. Furthermore, CFS is a read-only storage system but the publishers of data maintain the right to overwrite data they had inserted.

Generally, CFS uses Chord as it was initially implemented. However, CFS introduced some optimization to Chord's functionality regarding which CFS servers will be contacted. Contacting servers that are nearby in the underlying

[3]To relate this with the discussion about distributed hash tables of subsection 2.2, DHash uses Chord in the same way as a distributed hash table uses its underlying structured peer-to-peer network.

network is preferred over contacting those that are located further away. In this way the latency of a lookup is reduced.

The main issues that the DHash layer of CFS needs to deal with are the following:

1. *Replication.* DHash deals with replication by placing a block's replicas at the k servers that immediately follow after the block's successor in the identifier space. With this placement of the replicas, even if a block's successor fails, the block will be still available at the successor of the failed node. A block's replica will only be unavailable if all the k servers that store it are unavailable, a case which is highly unlikely.

2. *Caching.* A block is cached in every server that was contacted during this block's lookup.

3. *Load Balance.* Blocks are spread uniformly in the identifier space since the content hash function used to identify each block uniformly distribute the block IDs in the ID space. However, this is undesirable as different servers have different storage capabilities. To overcome this problem DHash introduces the notion of a *virtual server*. In this way each server is associated with a number of virtual servers. The more storage capacity a server has, the more virtual servers are associated with it.

4. *Quotas.* For preventing malicious users from exhausting the servers storage capacity with large amounts of data, every CFS server associates quotas for every publisher, that is, a maximum amount of storage that the publisher can use on this server.

5. *Updates and Deletion.* As explained before, updates of data are only permitted for the publisher of this data. In addition to this, CFS does not include an explicit delete operation. However, such operation functions implicitly because a CFS server may delete data that has not been refreshed recently. Thus, a publisher that wants its published data to remain in the system must refresh it periodically.

5.3.3 PAST

PAST [8] is a peer-to-peer storage utility that aims to provide strong persistence, high availability, scalability and security. The system is composed of nodes that are capable of initiating and routing client requests to insert or retrieve files. For routing, PAST uses Pastry [17] (as Pastiche [5] does) for accomplishing efficient data routing. Nodes can optionally contribute storage to the system. Inserted files are replicated on multiple nodes to ensure persistence and availability. Files

are replicated over a set of nodes that have high probability of being diverse in terms of geographic location, ownership, administration, network connectivity, etc.

Each PAST node and each user of the system holds a smartcard. Each such card is associated with a public/private keypair. To ensure balance between storage supply and demand, PAST includes a secure quota system. In simple cases users are assigned fixed quotas, or they are allowed to use as much storage as they contribute. Storage quotas are maintained by the user's smartcard. Optionally, a user could acquire a smartcard from an organization, called a *broker*, which controls how much storage must be contributed and/or may be used. A broker is not directly involved in the operation of a PAST network and only knows about the number of smartcards it has circulated, their quotas and expiration dates.

Files stored in PAST are immutable as an identifier for a file is derived from its filename and files cannot be inserted multiple times with the same identifier. An inserted file can be shared by the owner by distributing the file identifier. To avoid agreement protocols between nodes no delete operation is provided, however, an owner of a file can reclaim the storage associated with a file which does not guarantee that the file is no longer available.

When a file is inserted a *file certificate* is generated. The certificate contains some meta-data and is signed by a user's smartcard. The certificate allows storing nodes to verify that the owner of the file has credentials to insert a file, to ensure that the contents of the file have not been corrupted by faulty or malicious intermediate nodes and to ensure that the identifier for the inserted file is correct. After a file has been inserted, each storage node that successfully stored a copy of the file issues and returns a *store receipt* to the client which allows him to verify that all requested copies were successfully stored. When a file certificate is issued an amount corresponding to the file size times the replication factor is debited against the quota left on the user's smartcard. On retrieval of a file the certificate of the file is returned as well so a client can easily verify that the content is authentic.

When reclaiming files a *reclaim certificate* is issued for a file by the user. This certificate can be inspected and verified by the node receiving a reclaim request. This prevents users other than the owner of the file from reclaiming the file's storage. If the reclaim operation is accepted the storage node issues a *reclaim receipt* and returns it to the client. When a client presents an appropriate reclaim receipt issued by a storage node, the amount reclaimed is credited against the client's quota.

Storage nodes are randomly audited to see if they can produce files they are supposed to store, thus exposing nodes that are contributing less storage than

claimed on their smartcards.

A user can request that a file should be replicated k times. This replication factor can be different between different files.

Privacy and anonymity of users of PAST are ensured in the way that each user holds an unlinkable identity in the form of a public key. They never need to reveal their identity to others when retrieving or inserting. A user's smartcard signature is the only information associating a stored file or a request with the responsible user. Relationship between a smartcard and a user's identify is only known to the user unless he voluntarily releases this information.

Load balancing among storage nodes is maintained by uniform distribution of file identifiers and node identifiers. This balances the number of files stored at each node. For non-uniform popularity of files PAST uses a caching scheme to minimize fetch distance and to balance the query load.

5.4 Comparison of Distributed Backup/File System Applications

The previous sections gave an overview of some of the most representative distributed backup and distributed file system applications. pStore, PeerStore, Pastiche and the Cooperative Internet Backup Scheme belong to the first category, whereas OceanStore, the Cooperative File System and PAST belong to the second one. Section 5.1 gave a brief explanation of the similarities and differences between the two classes of applications. In this section we will compare the systems we have seen so far, having as points of reference some important issues that regard their design: (a) whether they use a distributed hash table and, if they do so, how they use it, (b) how they achieve fairness and efficiency regarding the ways they use for distributing the network's resources to the nodes, and (c) how they address the issue of the availability of the data.

5.4.1 Distributed Hash Table Usage

A distributed hash table structure provides efficient data location within a structured peer-to-peer network. This is the reason it is used by many distributed backup and file system applications. From the applications we have seen so far, MyriadStore, pStore, PeerStore, OceanStore and the Cooperative File System are the ones that make use of a distributed hash table structure, although each one of them is using it in different ways.

pStore and the Cooperative File System both use a distributed hash table for holding their data, as well as the meta-data that should be recorded. In these systems, the data and meta-data blocks that should be inserted into the

network are assigned an identifier that is derived by hashing their data. This identifier is used as the key with which they are inserted into the distributed hash table structure. In PeerStore and MyriadStore, the DHT is only used for keeping the meta-data, whereas the location of the actual data of the files that are backed up is determined by a symmetric trading protocol. This approach results in fairness regarding the distribution of the network's storage capacity between the nodes, as with this scheme a node consumes as much storage space as it contributes to the system. Furthermore, using the DHT for storing only the meta-data (which are expected to be relatively small), greatly avoids the migration costs that would be imposed in the case that the DHT was holding the actual data of the backed up files.

Finally, it should be pointed out that even though Pastiche and PAST do not use a distributed hash table for storing their data or meta-data, they are using the routing properties of the Pastry structured peer-to-peer network for accomplishing their goals. A Pastiche node uses Pastry's routing properties in order to establish a set a suitable buddy nodes. PAST uses Pastry to determine where data should be placed. Each file in PAST is associated with an identifier that corresponds to the cryptographic hash of the file's name, the public key of file's owner and a random salt[4]. Pastry routes a file to the node responsible for holding it, according to the file's identifier and Pastry's routing properties. This description shows that determining the location where a file should be placed in PAST is similar to determining the location of data inserted into a DHT structure.

5.4.2 Fairness and Efficiency in Storing Data

To compare the applications on how they distribute the network's storage capacity among its participating nodes, we will examine their approaches with respect to symmetry of storage space usage between the nodes and the ways they use for ensuring the safekeeping of the data.

PeerStore uses a symmetric trading protocol which ensures that nodes are consuming as much storage space of the network as the one they contribute to it. However, a node may be willing to accept some imbalance in the trade with another peer. In PeerStore every node keeps a *trade ratio* that indicates how much such imbalance is willing to accept. MyriadStore extends this symmetric trading scheme, so that nodes might not only trade for storage space but for other factors as well that the quality of a backup service might depend on (as explained in Section 3.2). In the Cooperative Internet Backup Scheme usage of

[4]In cryptography the term salt refers to a random number that is added to a password or an encryption key to protect them from being disclosed.

storage space is also symmetric. The other systems we have seen so far do not use the storage space in this symmetric fashion.

The systems we have described so far adapt several techniques for ensuring that nodes are indeed storing the data they are supposed to be storing. A PeerStore node randomly challenges the nodes that hold its backed up data. If they fail to respond then the node that made the challenge can punish the challenged node by dropping some of its data. This punishment could follow the *probabilistic punishment* model proposed in [10]. If a node does not respond to a challenge for a long period of time it should then be replaced with another node. Pastiche and the Cooperative Internet Backup Scheme also use challenges for ensuring data safekeeping. If challenged nodes are failing to reply, they are replaced by other nodes. Finally, in OceanStore safekeeping is achieved financially, as if a server is not holding the data it should be holding then the provider of this server is responsible for compensating for this loss.

Finally, regarding the efficiency of storing data, we will consider the use of the method of convergent encryption. This method was explained in Subsection 5.2.1. It provides a way of sharing a data blocks between several nodes that are entitled to access it. From the systems we have seen the ones that make use of this efficient way of encrypting and storing data blocks are pStore, PeerStore and Pastiche.

5.4.3 Data Availability and Mutability

Replication and erasure codes are the most widely used techniques that address the issue of availability of data. pStore, PeerStore, Pastiche, the Cooperative File System and PAST replicate the data among the nodes of the peer-to-peer network. With this technique, if a node holding some data is not available, then the data can still be retrieved by another node that is holding a replica of the desired data. On the other hand, the Cooperative Internet Backup Scheme uses Reed-Solomon erasure-correcting codes [18]. Erasure codes address the issue of availability of some data D in the following way: D is first split to produce a set of chunks; let's call this set C. Erasure codes encode this set of chunks to produce a larger set of chunks; let's call this set L. The property of L is that the original data D can be restored by *any* subset of chunks of L which is sufficiently large.

OceanStore makes use of both replication and erasure codes. Replicas are used for OceanStore's objects that are in *active* form, that is, the objects that exist in the system in their latest version. For the earlier versions of OceanStore's objects, which form OceanStore's deep archival storage, erasure codes are used.

Finally, OceanStore is the only system from the ones we examined that

	MyriadStore	pStore	PeerStore	Pastiche	Cooperative Internet Backup Scheme
Use of a DHT	Yes	Yes	Yes	No	No
Storing data on a DHT	No	Yes	No	No	No
Storing meta-data on a DHT	Yes	Yes	Yes	No	No
Symmetric storage space usage	Yes	No	Yes	No	Yes
Way of ensuring data safekeeping	Periodic challenges with reputation-based punishment	-	Random challenges with probabilistic punishment model	Challenges	Random challenges
Convergent encryption	Yes	Yes	Yes	Yes	No
Replication or erasure codes	Replication (can be extended to erasure codes)	Replication	Replication	Replication	Erasure codes
Mutable data	No	No	No	No	No

Table 5.1: Comparison between the distributed backup applications we have seen so far.

allows updates. In OceanStore these updates are allowed for the system's active objects. The deep archival storage maintains immutable data objects. In all the other systems we saw, the data residing in them are immutable.

Tables 5.1 and 5.2 summarize the comparison we gave between the distributed backup and distributed file system applications we have seen so far.

5.5 Other Distributed Backup/File System Applications

Apart from the systems we saw in the previous sections, there has been a number of other remarkable approaches in the field of distributed backup and distributed file systems.

BAR-B [11] is a first attempt of making a distributed backup system able to tolerate users with Byzantine behavior while ensuring that an unbounded number of "rational" users can be supported. [11] defines rational nodes as nodes that are self-interested and may deviate from the suggested protocol if doing so would be beneficial for them.

	OceanStore	Cooperative File System	PAST
Use of a DHT	Yes	Yes	No
Storing data on a DHT	No	Yes	No
Storing meta-data on a DHT	Yes	Yes	No
Symmetric storage space usage	No	No	No
Way of ensuring data safekeeping	Financial	-	Random challenges
Convergent encryption	No	No	No
Replication or erasure codes	Both	Replication	Replication
Mutable data	Yes (for active objects only)	No	No

Table 5.2: Comparison between the distributed file system applications we have seen so far.

Venti-DHash [13] is another distributed backup system that uses a distributed hash table for storing the backed up data. This system makes use of erasure codes for increasing its reliability. It uses Rabin's Information Dispersal Algorithm (IDA) [22] with $p = 65537$ for the erasure code it is using. According to this algorithm, for a given m, any number of fragments can be generated, and with probability $1 - \frac{1}{p} = \frac{65536}{65537}$, the original block can be reconstructed by any subset m of these fragments.

Farsite [23] is a another distributed file system. File availability and reliability in Farsite is provided with replication. The secrecy of file contents is ensured with cryptographic techniques and the integrity of the data of files and directories is ensured with a Byzantine-fault-tolerant protocol. Farsite uses local caching of data and lazy propagates of file updates to keep its performance to a sufficient level.

SFSRO [27] is a distributed read-only file system. It provides secure access to data with the use of a replicated database.

Finally, Ivy [26] and Pastis [25] are two other distributed read-write peer-to-peer file systems that make use of a distributed hash table. Data in Ivy reside in the DHash distributed hash table (the one that the Cooperative File System also uses) in the form of *logs*. Pastis uses the Past distributed hash table [24] for storing all its data. Experimental results showed that Pastis is between 1.4 to 1.8 times slower than NFS. This result shows that Pastis is relatively fast, since Ivy and OceanStore are between two to three times slower than NFS.

Chapter 6

Conclusion

MyriadStore is a peer-to-peer backup system that offers a more efficient solution for making backup when compared to the solutions that are traditionally used. MyriadStore offers a process of making backup that is more transparent to the user and that is cheaper than the one that traditional backup methods provide. These highly desirable characteristics are achieved while keeping the performance at satisfactory levels and while keeping the backed up data easily accessible, well-organized, secure and highly available.

MyriadStore uses a symmetric trading scheme that ensures that nodes get to use the same amount of storage space in the peer-to-peer network as the one they contribute to it. A challenge mechanism is in place in order to verify that nodes are storing the data they are entrusted with. Using this mechanism, a node can challenge its partners by requesting from them some small amounts of the data they should be holding. In the case that partners fail to respond correctly to these challenges, they are punished according to a reputation-based punishment model. By making use of this model, partners that fail challenges are punished according to the reputation they have in the system. The symmetric trading scheme along with the challenge and punishment mechanism works like a tit-for-tat mechanism which increases incentives to behave.

Similarly to many other storage systems, MyriadStore makes extensive use of a distributed hash table (DHT). However, MyriadStore does not make use of content hashes for the storage of the backed up data, as such an approach would not provide the flexibility of controlling the placement of data. This is highly undesirable as nodes have different capacities and share different amounts of storage space. Furthermore, storing the backed up data in the DHT would require that nodes shuffle huge amounts of data to ensure the correctness of the DHT. Under high churn, this would impose great network traffic and would have a great impact on the system's performance. MyriadStore solves these

problems by separating the storage of files and the storage of meta-data from each other.

Nodes in MyriadStore perform introspection to ensure that they indeed storing the file blocks they should be storing. If during introspection it is found that some of this data is missing then the node should contact the owner of this data in order to fetch it. Using introspection nodes can avoid being punished by challenging nodes. Moreover, nodes regularly perform inspection to ensure that the meta-data and the file blocks are in accordance to each other.

In order to keep the backed up data secure, MyriadStore uses public-key cryptography for encrypting every data piece that is stored remotely and, thus, preventing any unauthorized access to it. For keeping the backed up data available, MyriadStore replicates it to several nodes of the peer-to-peer network so that a replica of some desired data will most likely be available.

6.1 Future Work

MyriadStore is a system that has to address many issues and some of them are not yet implemented. In MyriadStore's current implementation the punishment mechanism works in a basic way that does not take into consideration the nodes' reputations in the system. This reputation system needs to be built in the future and included in the punishment mechanism as proposed in chapter 3.

Another issue that is worth investigating is a switch from the use of replication to the use of erasure codes. As briefly mentioned in chapter 5, with erasure codes some data is partitioned into data pieces which are then encoded to produce a greater number of fragments. The original data can then be can be reconstructed using *any* sufficiently large subset of these fragments. [14] gives a quantitative comparison between erasure codes and replication and shows that erasure codes offer a more efficient approach.

MyriadStore's implementation has not yet been integrated with security and this an essential feature that should be added in the future. As mentioned in chapter 3, public-key cryptography could efficiently solve the security issue. Moreover, to increase efficiency, *convergent encryption* could be used, so that different nodes can share the same data in the network, given that they have the rights to access it.

Lastly, the operations of *introspection* and *inspection* described in chapter 3 need to be included in MyriadStore's implementation. As we saw in chapter 3, these operations help MyriadStore recover from some faulty situations and, thus, they make the system more robust.

Bibliography

[1] L. O. Alima, S. El-Ansary, P. Brand and S. Haridi, DKS(N, k, f) A family of Low-Communication, Scalable and Fault-tolerant Infrastructures for P2P applications, *The 3rd International workshop on Global and P2P Computing on Large Scale Distributed Systems (CCGRID 2003)*, Tokyo, Japan, May 2003.

[2] A. Ghodsi, L. O. Alima, and S. Haridi. Symmetric Replication for Structured Peer-to-Peer Systems, DBISP2P2005, The 3rd International Workshop on Databases, Information Systems and Peer-to-Peer Computing, August 28-29, 2005, Trondheim, Norway.

[3] A. Ghodsi, L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi. Self-Correcting Broadcast in Distributed Hash Tables. In Series on Parallel and Distributed Computing and Systems (PDCS'2003), ACTA Press, Calgary, 2003.

[4] M. Landers, H. Zhang, K-L. Tan. PeerStore: Better Performance by Relaxing in Peer-to-Peer Backup. p2p, pp. 72-79, Fourth International Conference on Peer-to-Peer Computing (P2P'04), 2004.

[5] L. P. Cox and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of Fifth ACM/USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[6] C. Batten, K. Barr, A. Saraf, and S. Treptin. pStore: A secure peer-to-peer backup system. Technical Memo MIT-LCS-TM-632, MIT Laboratory for Computer Science, December 2001.

[7] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX).* ACM Press, November 2000.

[8] P. Druschel, A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS VIII)*, pages 75-80, IEEE Computer Society Press, May 2001.

[9] M. Lillibridge, S. Elnikety, A. Birrel, M. Burrows, and M. Isard. A Cooperative Internet Backup Scheme. In *Proceedings of the 2003 Usenix Annual Technical Conference*, pages 29-41, 2003.

[10] L. P. Cox and B. D. Noble. Samsara: Honor Among Thieves in Peer-to-Peer Storage. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 120-132, Bolton Landing, NY, USA, October 2003.

[11] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J-P. Martin, and C. Porth. BAR Fault Tolerance for Cooperative Services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, Brighton, United Kingdom, October 2005.

[12] A. Tridgell, P. Mackerras. The rsync algorithm. Technical report, TR-CS-96-05, Australian National University, June 1996.

[13] E. Sit, J. Cates, and R. Cox. A DHT-based backup system, August 2003.

[14] H. Weatherspoon and J. D. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, Cambridge, Massachusetts, March 2002.

[15] T. Klingberg and R. Manfredi, draft of the "The Gnutella 0.6" rfc, http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html, 2002.

[16] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM*, San Diego, CA, USA, August 2001.

[17] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329-350, Heidelberg, Germany, November 2001.

[18] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software, Practice and Experience*, 27(9):995-1012, September 1997.

[19] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[20] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Conference*, pages 1-10, San Fransisco, CA, January 1994.

[21] M. O. Rabin. Fingerprinting by random polynomials. Technical report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[22] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335-348, April 1989.

[23] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.

[24] A. Rowstron, and P. Druschel. Storage management and caching in Past, a large-scale, persistent, peer-to-peer storage utility. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP 2001)*, October 2001.

[25] J-M. Busca, F. Picconi, and P. Sens. Pastis: A Highly-Scalable Multi-User Peer-to-Peer File System. In *Euro-Par 2005*, Lisboa, Portugal.

[26] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A Read/Write Peer-to-Peer File System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, December 2002.

[27] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 181-196, October 2000.

[28] C. Plaxton, R. Rajaraman, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of ACM SPAA*, pages 311-320, Newport, Rhode Island, June 1997.

[29] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop

PCs. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 3443, Santa Clara, CA, June 2000.

[30] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of the ACM SIGCOMM 2001 Symposium on Communication, Architecture, and Protocols*, pages 161-172, San Diego, CA, U.S.A., August 2001. ACM SIGCOMM.

[31] K. Aberer, P. Cudrè-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: a self-organizing structured P2P system. *SIGMOD Record*, 32(3):29-33, 2003.

[32] M. F. Kaashoek and D. R. Karger. Koorde: A Simple Degree-optimal Distributed Hash Table. In *The 2nd Interational Workshop on Peer-to- Peer Systems (IPTPS'03)*, volume 2735 of *Lecture Notes in Computer Science*, Berkeley, CA, USA, 2003. Springer-Verlag.

[33] N. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, March 2003.

[34] B. Y. Zhao, L. Huang, S. C. Rhea, J. Stribling, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Global-scale Overlay for Rapid Service Deployment. *IEEE J-SAC*, 22(1):41-53, January 2004.

[35] B. Leong, B. Liskov, and E. Demaine. EpiChord: Parallelizing the Chord Lookup Algorithm with Reactive Routing State Management. In *12th International Conference on Networks (ICON'04)*, Singapore, November 2004.

[36] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: a public DHT service and its uses. In *Proceedings of the ACM SIGCOMM 2005 Symposium on Communication, Architecture, and Protocols*, pages 73-84, New York, NY, USA, 2005. ACM Press.

[37] C. Blake and R. Rodrigues. High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two. In *9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*.