Lehrstuhl für Netzarchitekturen und Netzdienste
Fakultät für Informatik
Technische Universität München

Redundancy and Access Permissions in Decentralized File Systems

Johanna Amann

Vollständiger Abdruck der von der Fakultät für Informatik
der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 20.06.2011 bei der Technischen Universität München
eingereicht und durch die Fakultät für Informatik
am 05.09.2011 angenommen.

Zusammenfassung

Verteilte Dateisysteme sind ein Thema mit dem sich die Informatik immer wieder aufs Neue auseinandersetzt. Die ersten verteilten Dateisysteme sind schon sehr kurz nach dem Aufkommen von Computernetzen entstanden.

Traditionell sind solche Systeme serverbasiert, d. h. es gibt eine strikte Unterscheidung zwischen den Systemen welche die Daten speichern und den Systemen welche die Daten abrufen. Diese Architektur hat Vorteile; es wird zum Beispiel angenommen, dass die Server vertrauenswürdig sind. Außerdem gibt es eine inhärente lineare Ordnung der Dateioperationen. Auf der anderen Seite sind solche zentralisierten Systeme schlecht skalierbar, haben einen hohen Administrationsaufwand, eine geringe Fehlertoleranz oder sind teuer im Betrieb.

Im Verlauf des letzten Jahrzehnts wurde eine neue Art von verteilten Dateisystemen wiederholt diskutiert. Diese werden dezentrale oder Peer-to-Peer Dateisysteme genannt. Bei diesen Systemen verschwimmt die strenge Trennung zwischen datenspeichernden und datenabrufenden Systemen. Die Daten werden auf allen am Dateisystem teilnehmenden Rechnern gespeichert. Eine solche Systemarchitektur kann viele der Problemstellen serverbasierter Systeme ausgleichen. Sie ist unter anderem leicht skalierbar, benötigt wegen der verteilten Architektur eine geringere Bandbreite und hat einen niedrigeren Administrationsaufwand. Sie bringt aber neue Herausforderungen mit sich: ihre interne Architektur ist weit komplexer, die benötigten Algorithmen werden noch immer erforscht oder verbessert. Die verschiedenen existierenden Prototypensysteme spielen in der Praxis keine Rolle, da in den bisherigen Betrachtungen einige für die Praxis wichtige Systembestandteile nicht einbezogen wurden.

Die vorliegende Arbeit präsentiert einige dieser fehlenden Systembestandteile. Um den Ausfall einzelner Rechner tolerieren zu können, müssen dezentrale Dateisysteme Daten redundant speichern. Diese Arbeit präsentiert ein ressourcensparendes Verfahren, welches eine kontinuierliche Verfügbarkeit der Daten sicherstellt und mit verschiedenen Redundanzalgorithmen verwendet werden kann. In Simulationen, die auf Verkehrsdaten echter Peer-to-Peer Systeme basieren, wird die Effektivität des Verfahrens demonstriert und die Tauglichkeit verschiedener Redundanzalgorithmen für dezentrale Dateispeicherung verglichen.

Eine weitere Problematik dezentraler Dateisysteme ist die Durchsetzung von Dateirechten. Da es keine vertrauenswürdigen Instanzen gibt, muss die Sicherung der Integrität und Vertraulichkeit durch kryptografische Verfahren erfolgen. Die existierenden Ansätze skalieren nicht, unterstützen keine Benutzergruppen oder benötigen zumindest teilweise vertrauenswürdige Rechner. Diese Arbeit präsentiert ein System, welches Dateirechte nur mittels Kryptografie durchsetzt und gleichzeitig alle diese Voraussetzungen erfüllt. Das

System wurde in einem Prototypen implementiert um die Funktionsfähigkeit des Ansatzes zu demonstrieren.

Die in dieser Arbeit entwickelten Algorithmen schließen zwei der wichtigsten Funktionalitätslücken der existierenden Systeme. Es können nun vollkommen dezentrale Dateisysteme, welche für den Nutzer in ihrer Funktionalität nicht von serverbasierenden Systemen unterscheidbar sind, entwickelt werden.

Abstract

Distributed file systems are a recurring topic in computer science. The first distributed file systems surfaced shortly after the inception of computer networking.

Traditionally these systems are server based; there is a strict distinction between systems that store data and systems that access data. This architecture has advantages; it is e.g. assumed that the servers are trusted and there is an implicit linear order in which the commands are executed. On the other hand, such systems do not scale well, have a high administrative cost, a low tolerance for failure or are expensive to operate.

In the last decade, a new kind of distributed file system has been frequently discussed. They are called decentralized or peer-to-peer file systems. In these systems, the strict separation between data storing and data accessing peers becomes blurred. The information stored in the system is spread among all peers that are part of the file system. Such system architectures are not impeded by many of the problems of server based systems. They are e.g. rapidly scalable, have lower bandwidth requirements because of their decentralized architecture and have a lower administrative overhead. However, these systems also pose new challenges: their internal architecture is much more complex and the required algorithms are still being researched or improved. The existing prototype systems are rarely used in practice because they are missing key features required in the real world.

This thesis presents some of these missing system components. To be able to tolerate node failures, decentralized file systems have to use data redundancy. This work shows a resource efficient method that guarantees the continuous availability of the data and can be used with different redundancy algorithms. In simulations that use real-world peer-to-peer network traces, the effectivity of the method is demonstrated and the suitability of the different redundancy algorithms for decentralized data storage is compared.

A further problem of decentralized file systems is the enforcement of file access permissions. Because there are no trusted nodes in the network, data integrity and confidentiality have to be enforced solely by cryptography. The existing approaches do not scale, do not support user groups or need at least partially trusted network nodes. This thesis presents a cryptographic approach that enforces file permissions and satisfies all these requirements. The system was implemented in a prototype to demonstrate the validity of the approach.

The algorithms developed in this thesis close two of the most important functionality gaps of existing systems. They can be used to implement decentralized file systems which are indistinguishable from server based systems from the user's perspective.

Pre-Publications

Parts of this dissertation have been pre-published in earlier versions:

- **Adding Cryptographically Enforced Permissions to Fully Decentralized File Systems**
  Johanna Amann and Thomas Fuhrmann
  Technical Report TUM-I1107, Technische Universität München, München, Germany, 6. April 2011
- **A Quantitative Analysis of Redundancy Schemes for Peer-to-Peer Storage Systems**
  Yaser Houri, Johanna Amann, and Thomas Fuhrmann
  Proceedings of the 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'10), New York City, USA, September 20-22, 2010
- **Cryptographically Enforced Permissions for Fully Decentralized File Systems**
  Johanna Amann and Thomas Fuhrmann
  Proceedings of the 10th IEEE International Conference on Peer-to-Peer Computing 2010 (P2P'10), Delft, the Netherlands, August 25-27, 2010
- **Unix-like Access Permissions in Fully Decentralized File Systems**
  Johanna Amann and Thomas Fuhrmann
  Poster Presentation at the 8th USENIX Conference on File and Storage Technologies (FAST'10), San Jose, California, USA, February 23-26, 2010
- **IgorFs: A Distributed P2P File System**
  Johanna Amann, Benedikt Elser, Yaser Houri and Thomas Fuhrmann
  Proceedings of the 8th IEEE International Conference on Peer-to-Peer Computing (P2P'08), Aachen, Germany, September 8-11, 2008

Danksagungen

Ich schulde vielen Menschen einen herzlichen Dank, die beim Schreiben dieser Arbeit auf verschiedenste Weise unterstützt haben.

Ich danke meinem Doktorvater Dr. Thomas Fuhrmann, ohne dessen Betreuung diese Dissertation nicht möglich gewesen wäre. Weiterhin danke ich den Mitgliedern der Prüfungskommission Prof. Dr.-Ing. Georg Carle, V.-Prof Dr. Kurt Tutschku sowie dem Vorsitzenden der Prüfungskommission Prof. Dr. Johann Schlichter, ohne deren schnelle Korrektur und zeitliche Flexibilität bei der Terminfindung der Antritt meiner Folgestelle nicht möglich gewesen wäre.

Vielen Dank auch an meine Kollegen Benedikt Elser, Björn Saballus und Thomas Nitsche, die sehr zu meiner langfristigen Motivation beigetragen haben.

Ich danke weiterhin Florian Sesser, Jean Ledwon, meinem Großvater Horst Rockinger und meinem Vater Anton Amann für die große Hilfe bei der Suche nach Grammatik und Rechtschreibfehlern und für die Kommentare zur Doktorarbeit.

Einen besonderen Dank möchte ich noch an Dr. Georg Groh für seine Unterstützung in der Endphase richten.

Vielen Dank auch an meine Familie und sowie alle anderen die mich in dieser Zeit unterstützt haben.

Contents

1. Introduction

Storing and retrieving information always has been of great importance to human culture. Mankind has always sought methods to make their knowledge and information available to other people and the following generations – examples for this are cave-paintings or (much) later hieroglyphs [Wuttig and Yamada, 2007]. Later, written notes, books, newspapers and the like became the main methods for the widespread transmission of information.

Electronic systems in general changed all this with inventions such as telephones, radio and television sets. What all these systems had in common was that it was very difficult for a single person to reach a larger audience. One possibility was to write letters to newspapers or periodicals in the hope that they would publish them in one of the following issues. However, distributing information on a global scale was nearly impossible.

Personal computers opened up completely new possibilities for the general public to handle information, which was no longer regulated by magazine editors or the like. Early systems for information exchange were, for example, the Bulletin Board Systems (BBS) [see e.g. Eichmann and Harris, 1981], FidoNet [Bush, 1993] or the Amateur Packet Radio Network (AMPR) [Leiner et al., 1985]. In the 90's, Internet access became easily available to the general public, making most of the mentioned systems obsolete[1].

The Internet is still evolving – the bandwidth available to general households is steadily increasing. The use cases of the Internet are also changing with the available bandwidth. 10 years ago, high quality video streaming was – more or less – science fiction. Today, the video platform [YouTube] is one of the most popular sites in the Internet.

The data that is stored in a computer system is usually stored in files. These files are made available to the user via the local computer file system.

When accessing information in the Internet, protocols like HTTP or SMTP are used to transfer information – often in the form of files. Emails, especially, are often "abused" as a means to share information because they are easy to use and widely available.

However, there are better solutions for the problem of making files available to other users.

When computer networks became available, one of the first things these systems were used for was the exchange of files between computer systems. To make this file sharing as transparent and easy for the user as possible – even at that time – different kinds of network file systems were devised. For the user, access to network file systems usually is transparent – files on a network file system are accessed exactly the same way as files on

[1]Many of them still are used on a small scale though. There are, for example, still active FidoNet nodes and the Amateur Packet Radio Network is also still operating – it is even carrying IP traffic.

a local file system. The only thing a user may notice is a longer delay when accessing files that are not stored on the local machine.

In today's environment, network file systems are ubiquitous. In corporate settings, especially, there are often huge company-wide accessible file systems, which are used to store all kinds of information that is important for the company.

Typically, such network file systems rely on centralized components and are administrated by a small group of people. This results in high costs for bandwidth, highly reliable components, and skilled personnel. These expenses and complications arise when a company grows – designing and operating a file system for a company that operates on a global scale is a very complex and costly task.

One approach solving many of the issues of such network file systems are *Distributed File Systems* (DFSs) – file systems where there is no single server. Instead, the data stored in the system is distributed among all computers that are active in the system. Wang and Anderson [1993] mention several key advantages of distributed file systems:

- Distributed file systems keep traffic in the local network where possible. Where this is not possible, the fastest node that can deliver the data is chosen. Thus, decentralized file systems save *bandwidth* and reduce *latency*.
- Distributed file systems have a higher *availability* because of their failure resistance. Depending on the exact settings even correlated failures of a great number of nodes will be tolerated without any data loss.
- Distributed file systems offer *gradual scalability*. When storage capacity needs to rise or the system bandwidth is no longer sufficient, new nodes can be added to the system.

Current events have shown how, especially, the problem of availability can be a greatly underestimated. Events like the Internet shutdown in Egypt [Betschon, 2011] or the many failures of deep-sea cables in the last months [Shaheen, 2011; Zhen, 2011] made the global internet unavailable or at least very slow in the affected areas. Global corporations with important servers in one of these regions would have had many problems. With distributed systems such failures do not affect the system as a whole.

Decentralized file systems are a special class of distributed file systems. In distributed file systems, there are still servers with dedicated functions – which can affect the general system functionality if they fail. Decentralized file systems drive the distributed design further – in a decentralized system all system nodes are egalitarian and share the complete functionality. There is no single point of failure anymore.

The goal of this work is to create an easy-to-use, fully decentralized distributed file system that provides transparent file access to its users. This thesis is based on structured peer-to-peer techniques because – when they are deployed correctly – P2P systems are extremely scalable and robust.

1.1. Contribution

In previous works our working group for self organizing systems at the chair of network architectures and services at the TUM built the fully decentralized peer-to-peer file system *IgorFs*, which can already remedy many of the problems that were discussed in the previous section. The system caches all information a local node has downloaded. When files stored in the network change, only the changed pieces of information have to be downloaded from the network. Data can be downloaded by all nodes that currently have a copy of the data block available. Proximity routing algorithms are used to retrieve this information from the node with the best connection.

Although current decentralized file systems are already usable for some settings, they are still lacking a few features that are very important in real-world environments.

In this work I present several more steps that go along the way to make fully decentralized file systems a viable alternative to the current server-based systems.

I present two different fields I have worked on, namely redundancy maintenance and permissions for fully decentralized file systems.

This thesis proposes a new way to ensure data availability and durability in fully decentralized distributed storage systems. It argues that an algorithm that uses Random Linear Coding (RLC) can more efficiently generate redundancy blocks than the state-of-the-art redundancy algorithms used in peer-to-peer storage systems. To support this claim, the algorithms are evaluated with different parameter choices using both synthetic and real-world data in static and dynamic settings. Thereby, it is shown that the RLC approach can outperform all the other coding schemes that are evaluated in the thesis.

Most current decentralized file systems ([Ghemawat et al., 2003; Borthakur, 2008; Braam et al., 2004; Chen et al., 2002a,b; Juve et al., 2010; Kubiatowicz et al., 2000; Dabek et al., 2001b; Muthitacharoen et al., 2002; Bhagwan et al., 2004; Badishi et al., 2009; Peric, 2008; Peric et al., 2009; Richard et al., 2003], see also Chapter 4 on page 51) are lacking any kind of enforced permission system, where the local client cannot simply ignore the flags. The available permission systems are severely restricted and incompatible with the semantic a Unix-user expects. A distributed file system should be as transparent for its users as possible. Thus, the access permissions used in this thesis are modeled closely on the [POSIX.1] permissions which are used in Linux as well as many other systems.

1.2. Organization

The outline of the remainder of this thesis is shown in Figure 1.1 on page 21.
The thesis is structured in three parts:

Part I contains background information about the areas handled in this thesis.

Chapter 2 introduces the cryptographic algorithms used in this work including basics about symmetric and asymmetric cryptography and more advanced topics such as elliptic curve cryptography and group keying operations.

Chapter 3 gives an overview of peer-to-peer networking. It introduces the basic concepts and the different kinds of peer-to-peer networks. Several widely known networks are examined in detail.

Chapter 4 details the inner working of local as well as network file systems. It shows several file systems that have similarities to the approach outlined in this thesis.

Chapter 5 presents the Igor File System (IgorFs), which was the basis for the work done in this thesis. The differences of IgorFs and the file systems presented in Chapter 4 are shown.

Part II handles the topic of redundancy maintenance in distributed file systems

Chapter 6 introduces the different coding schemes usually used in peer-to-peer systems. It introduces random linear coding and gives a first comparison of the different coding schemes.

Chapter 7 compares the schemes presented in Chapter 6 using real-world network traces.

Part III of this thesis introduces a new cryptographically enforced permission system for decentralized file systems.

Chapter 8 gives an overview of the design of the cryptographic permission system developed as part of this thesis. An overview of related schemes in the published literature is given.

Chapter 9 presents the cryptographic permission system in detail.

Chapter 10 estimates the overhead introduced by the cryptographic permission system. It also discusses the limitations of the presented approach and shows ways to remedy them. Furthermore, possible extensions for the permission system are discussed.

Part IV concludes this thesis with **Chapter 11**, which summarizes the result of this thesis.

The **Appendix** shows how to use the programs developed over the course of this thesis.

Appendix A presents the simulation environment used to generate the results presented in Part II of this thesis.

Appendix B presents a guide on how to implement the cryptographic permission system presented in Part III of the thesis.

Appendix C introduces CryptFs, the prototype implementation of the cryptographic permission and integrity protection scheme devised in this thesis.

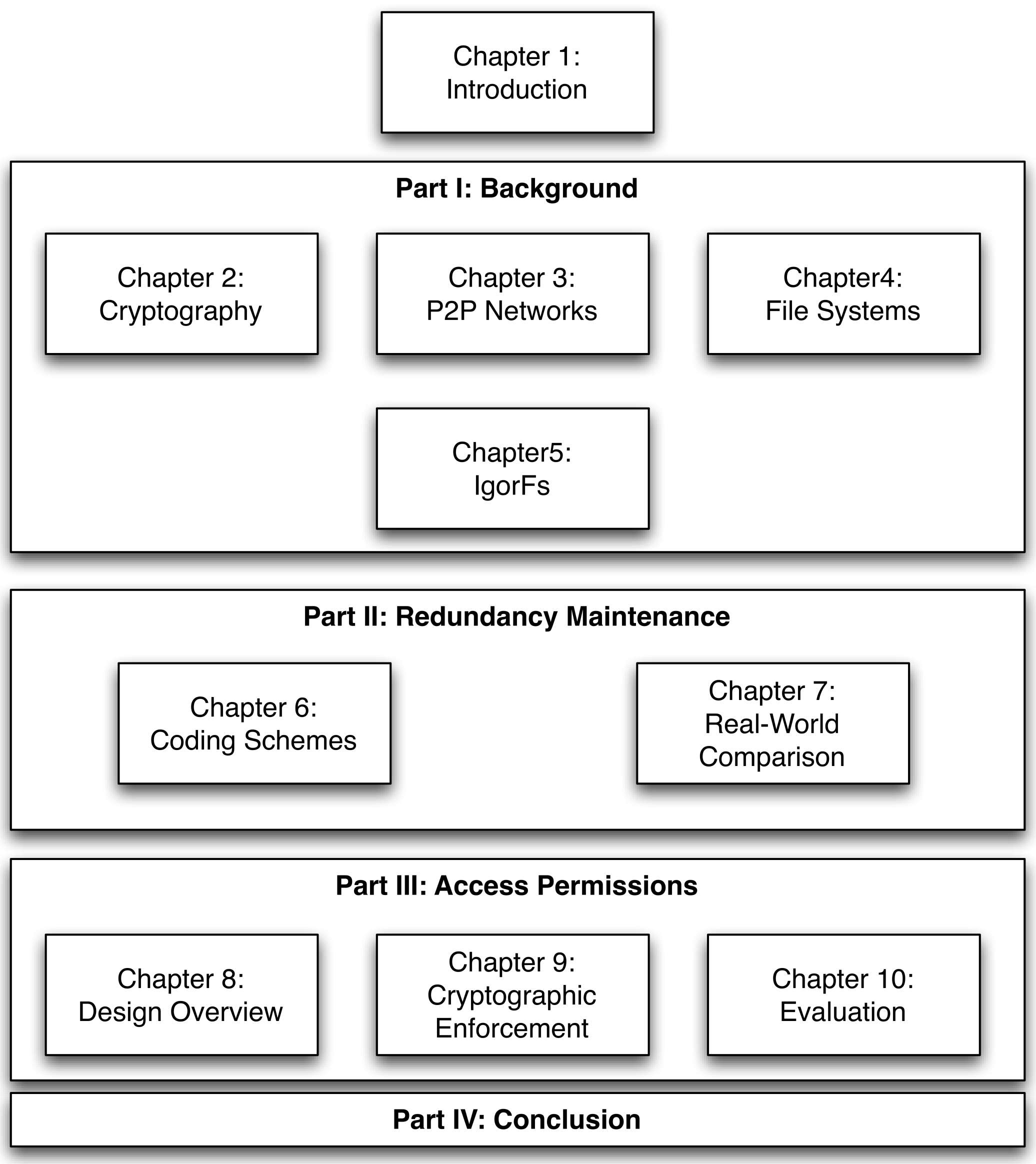


Figure 1.1.: Thesis outline

Part I

Background

2. Cryptographic algorithms

Keeping information from falling into the wrong hands is a topic that has concerned mankind for a long time. This topic is particularly important in distributed systems where messages are often routed over systems that the receiver does not know or control and which thus could be malicious.

This chapter gives an overview of the cryptographic algorithms that are used in the remainder of this dissertation. It will first discuss hashing algorithms. After that symmetric and asymmetric cryptographic algorithms will be presented. Finally some newer algorithms which mix several of the discussed cryptographic primitives will be discussed.

A comprehensive history of the development of cryptography is given by Kahn [1996] in his book *The Codebreakers* which was first published in 1967. The National Security Agency (NSA) tried to stop the publication; in the end some material concerning the NSA and the British GCHQ was removed [Bamford, 1983, pp. 168–173].

The following sections explain some of the current algorithms with examples. These examples use the person names that have been established in the cryptographic literature. *Alice* and *Bob* are two persons who want to exchange some information. *Eve* is a malicious person who can listen to all communication between Alice and Bob and tries to decrypt the traffic.

2.1. Hashing Algorithms

A one-way hash function is a mathematical deterministic function which is easy to compute in one direction but very hard to compute in the other direction. It takes an input of variable length and outputs a fixed length number. Figure 2.1 on the following page illustrates this. Hash functions are used to check the integrity of messages.

Hash functions were first developed at IBM. They appear to have been invented by H. P. Luhn in 1953, but the work was not published [Knuth, 1978, p. 541]. Hashing functions were independently published by Dumey [1956] and Ershov [1958]. The first paper dealing with the problem of searching in large files was published by Peterson [1957].

A first informal definition of such one-way hash functions was given by Merkle [1979, 1989] and Rabin [1978]:

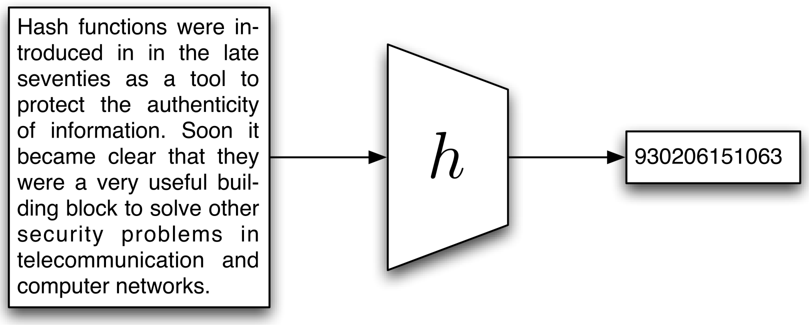

Hash functions were introduced in in the late seventies as a tool to protect the authenticity of information. Soon it became clear that they were a very useful building block to solve other security problems in telecommunication and computer networks.
h
930206151063

Figure 2.1.: A hash function [adapted from Preneel, 1994]

Definition 2.1 (from [Preneel, 1994])**.** *A* ***one-way hash function*** *is a function h satisfying the following conditions:*

1. *The argument X can be of arbitrary length and the result $h(X)$ has a fixed length of n bits (with $n \geq 64$).*
2. *The hash function must be one-way in the sense that given a Y in the image of h, it is "hard" to find a message X such that $h(X) = Y$, and given X and $h(X)$ it is "hard" to find a message $X' \neq X$ such that $h(X') = h(X)$.*

For a formal definition of one-way hash functions see [Preneel, 1999, pp. 2–4].

In the implementations done over the course of this work, the hash algorithms SHA-1 [Eastlake and Jones, 2001] and SHA-256 [FIPS 180–2] are used to generate content hashes. At the time of writing, SHA-1 is the de-facto standard hash algorithm, despite the fact that Wang et al. [2005] have found weaknesses in the algorithm allowing a faster search for collisions. Generally, the community suggests replacing SHA-1 with more secure algorithms such as SHA-256 [see e.g. Satoh, 2005].

SHA-1 is used in the implementation of this dissertation despite this fact because of the ready availability in cryptographic libraries. The work done in this thesis does not rely on a specific hash algorithm. Replacing SHA-1 in the code only would have increased the complexity without offering any new insights.

2.2. Symmetric Encryption Algorithms

Symmetric encryption algorithms are encryption algorithms that use the same key for message encryption and decryption.

Figure 2.2 on the next page illustrates this with an example. Alice wants to send a text file to Bob. For this purpose, Alice and Bob first have to share a key. This key has to be distributed using out-of-band communication. They could, for example, exchange it the next time they meet in person.

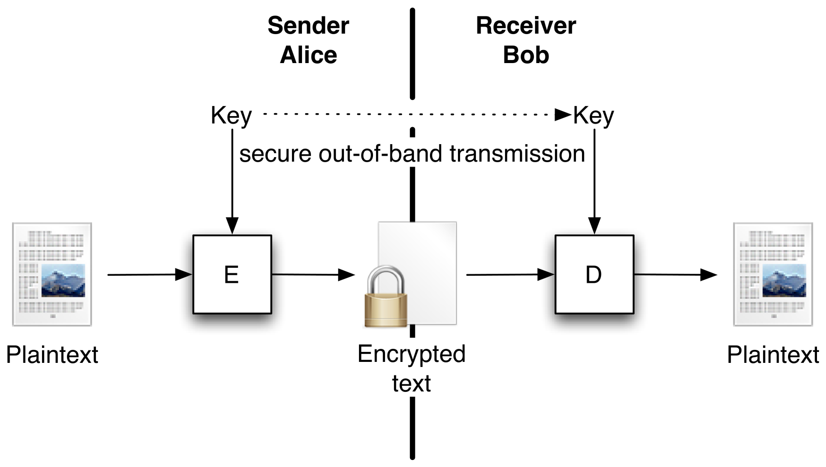


Figure 2.2.: A symmetric cryptosystem

Alice uses the key to encrypt the message with a symmetric encryption algorithm. Bob can then use the same key to decrypt the message.

Eve will only have knowledge of the encrypted message. The only way for Eve to decrypt the message, barring a weakness in the encryption algorithm, is to get knowledge of the secret encryption key.

In the implementations done over the course of this work, the Advanced Encryption Standard (AES) algorithm [Housley, 2004; FIPS 197] with 128 bit keys is used to perform all symmetric encryptions. At the time of writing, AES is the standard symmetric cipher used in all kinds of applications.

2.3. Public Key Algorithms

Public-key cryptographic algorithms, also called asymmetric cryptographic algorithms are algorithms, where different keys are used to encrypt and to decrypt the message. The encryption key, also known as the public key, does not have to be kept secret. Knowledge of the public key cannot be used to decrypt encrypted messages.

It is usually said that public-key cryptography was invented by Diffie and Hellman [1976] and Merkle [1978] independently. There are also some voices that attribute the invention to Ellis [1999] who worked at the Government Communications Headquarters (GCHQ) in Cheltenham for the British Government. A detailed report about this is given by Singh [1999, pp. 279–292]

Figure 2.3 on the following page illustrates public-key cryptography with an example. As a first step, Bob generates a public/private key pair. He sends the public part of the key to a directory service and keeps the private part of the key to himself.

Alice wants to send a text-file to Bob. To do this, she encrypts the file using Bob's public key, which she got for example from a public directory service. In high security situations, Alice would verify that she got the correct key from the directory. Usually this

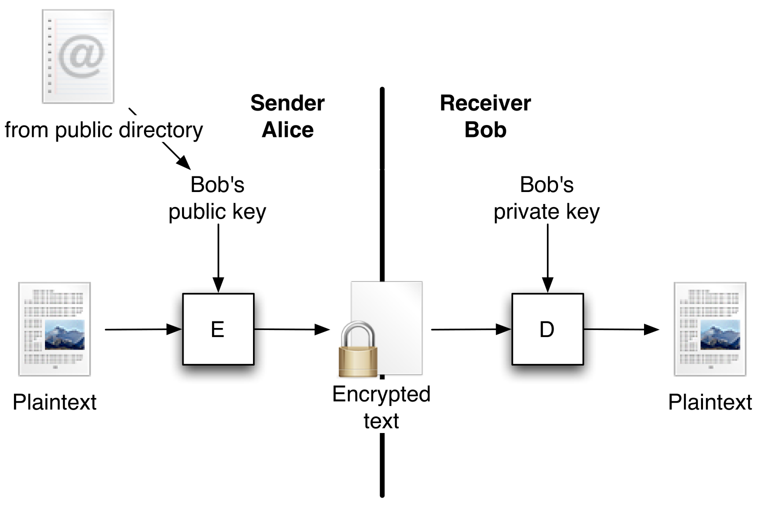


Figure 2.3.: An asymmetric cryptosystem

is done by communicating the hash of the public key via a secure out-of-band medium. In this context the hash of the public key is usually called the *fingerprint*.

Eve can only decrypt the transferred messages if she gets knowledge of Bob's private key. This is much more difficult than in the symmetric example because Bob never tells the key to anyone. A second method to read the transferred messages would be to replace the key in the public directory. However, this only works if the fingerprint is not verified. Bob would also notice this attack because he can no longer decrypt the message himself.

This flexibility comes at a cost: public-key cryptography is several orders of magnitude slower then symmetric algorithms with a comparable security. In practice, public-key cryptography is usually used to encrypt a symmetric cryptographic key, which is then used to encrypt the message. Using this method, only a small message has to be encrypted with the expensive public-key operations.

The most well known and widely used public key algorithm today is the Rivest-Shamir-Adleman (RSA) public-key cryptosystem [Rivest et al., 1977, 1978].

2.4. Signature Algorithms

Public-key cryptosystems can also be used to sign messages. A cryptographic signature proves that a message has not been altered since it has been sent.

Signatures work by using the keys of the public-key cryptosystem in reverse order. When Alice wants to send a signed message to Bob, she first encrypts it with her private key. Bob then receives the message and can decrypt it using the public key of Alice. The decryption only works if the message has not been changed.

When Eve tampers with the message while it is being transferred from Alice to Bob, Bob will no longer be able to decrypt it.

Once again, in practice, it is too computationally expensive to encrypt the whole message. So, in practice, a secure hashing algorithm is used to compute the digest of the message. The public-key algorithm is then used to sign the hash value of the message. The receiver can verify the message by first recomputing the hash and then checking the hash signature.

RSA, DSA [NIST-DSS] and ElGamal [El Gamal, 1985] are examples of public-key cryptosystems.

This work does not use these asymmetric algorithms. They are neither used for encryption nor for signature generation. Instead, the faster elliptic curve alternatives presented in the next section are employed.

2.5. Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) was independently proposed in the mid-eighties by Miller [1986] and Koblitz [1987]. ECC is based on the fact that a group operation can be embedded into the solution set of an elliptic curve over a field. This fact can be exploited to do cryptography, see [Win and Preneel, 1998] for a detailed introduction. Most interestingly, all Discrete Logarithm Problems (DLP)-based cryptosystems can be converted to use elliptic curves [Win and Preneel, 1998, p. 136].

Many public-key cryptosystems, e.g. the aforementioned ElGamal and DSA are DLP based. The definition of a DLP cryptosystem is:

Definition 2.2 (adapted from [Win and Preneel, 1998; Amann, 2007])**.** *Let G be a group and $\circ$ its group operation. Let x and y be elements of G and $n \in \mathbb{N}$. Define $y = x \circ x \circ \cdots \circ x$ where the number n of terms in the right-hand side of the equation is unknown. Alternatively, note $y = x^n$. The DLP consists of determining the value of n only knowing x and y and the group properties of G.*

Elliptic Curve Cryptography has several advantages in comparison to the aforementioned asymmetric cryptosystems. It provides approximately the same security as traditional signature algorithms such as RSA or ElGamal while requiring significantly shorter keys.

E.g. to achieve the same level of security as 1024-bit RSA, ECC requires a key size of only about 171-189 bits [Wiener, 1998, p. 4]. Furthermore, the complexity of the operations is much lower than for traditional algorithms [Eckert, 2006, p. 340]. Gupta et al. [2002] analyzed the performance of RSA primitives versus the ECC signing algorithm ECDSA and found that for larger key sizes (starting with 2048 bits) ECDSA is faster in all operations[1].

Hence, ECC signatures only have advantages – they need lower storage overhead and are computationally faster.

In this thesis, ECC is used in several places. In all these places, the Curve P-384 [FIPS 186–2, p. 32], with a key length of 348 bits is used. It was specified by the NIST for use by US government entities. According to Lenstra and Verheul [2001] an ECC signature with

[1]For a comparison of many different available ECC implementations see [López and Dahab, 2000].

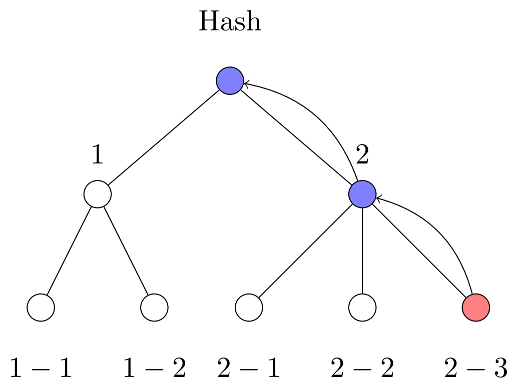


Figure 2.4.: Merkle hash tree

this key length should be secure until well after the year 2050. Signatures are generated with the Elliptic Curve Digital Signature Algorithm (ECDSA) [X9.62]. Note that the implementation in this paper is not susceptible to the ECDSA timing attack recently discovered by Brumley and Tuveri [2011]. This attack requires a weakness in a specific library (namely OpenSSL). While the implementation done over the course of this work uses OpenSSL, there is no live key exchange and, thus, the attack cannot be performed.

2.6. Merkle Hash Trees

Hash trees were first proposed by Merkle [1979, 1988]. For this reason they are also called Merkle trees or Merkle hash trees.

A hash tree is formed by hashing structured data in rounds. An example for such a hash tree is shown in Figure 2.4. The lowest level of the tree is formed by hashing the data blocks. In Figure 2.4 the resulting hashes are called 1-1, 1-2, etc. These hashes are then again hashed to form the hashes 1 and 2. They are again hashed and form the top-level hash.

One advantage of hash trees is that they can be rapidly recomputed when parts of the tree change. For example, in Figure 2.4 the rightmost data block changes. Hence, the hash 2-3 also changes. To recompute the top level hash, only the hash of 2 and the top-level hash have to be recomputed. R Recompution of the other block hashes, is not necessary if the hash values for all nodes in the tree have been saved.

Merkle trees are frequently used in modern file system designs, for both local and network file systems. An example for a file system using hash trees for integrity protection is the Zettabyte File System [ZFS].

2.7. Symmetric Signatures

Another approach to speed up signing operations is the use of *one-time signatures* which are also called *symmetric signatures*. Such signatures were first proposed by Lamport [1979] and Rabin [1978]. Merkle [1990] later presented a well-known one-time signature scheme.

Symmetric signatures use one-way hash functions instead of the asymmetric cryptographic primitives of public key signature algorithms. While one-way hash functions are significantly faster, their use in signature schemes has several disadvantages, which makes the schemes unattractive for most applications. As the name one-time signature already suggests, basic symmetric signatures can only be used to sign one message. Furthermore, the signature has a longer size than the asymmetric equivalent.

To allow multiple signatures, symmetric signatures require sophisticated key management schemes. Merkle [1990] uses the aforementioned Merkle hash tree to this end. More recent approaches like [Berman et al., 2007] improve this scheme and are able to handle big trees more efficiently.

Naor et al. [2006] present an even more advanced symmetric signature scheme called Fractal Merkle Tree Sequential Signatures. They analyzed the speedup that can be achieved by using one-time signatures instead of asymmetric algorithms. The practical speedups ranged from 10 to 15 times the speed of RSA in signing time and from 3 to 5 times the speed of RSA in verification time.

It is possible to use symmetric signatures in the permission system of this dissertation instead of the public key signature algorithms. This is discussed in Section 10.3.4 on page 180.

2.8. X.509 Certificates

A common problem in the Internet is the secure identity verification of persons and organizations. X.509 certificates are widely used today as a solution for this kind of problem; they can securely identify individual persons or services in the Internet. They are, for example, used to identify the owners of web sites when using HTTPS – HTTP over a SSL/TLS connection. They are also frequently used to identify the users of web servers or mail servers – a server can require a user to present a valid certificate before granting her access. Certificates are also used to encrypt or sign mail traffic between individual users and to sign software and device drivers[2].

A certificate contains, inter alia, the public key of the certified entity, the name of the entity, and the expiration date [Eckert, 2006, p. 390]. A certificate is signed by a Certificate Authority (CA).

To verify the identity of an entity, the system checks if the certificate is signed by a trusted CA. To this end, each system has a list of the certificates of all trusted CAs. This list is usually bundled with the operating system or the web browser. Big enterprises

[2]For example current versions of Microsoft Windows require all device drivers to be signed.

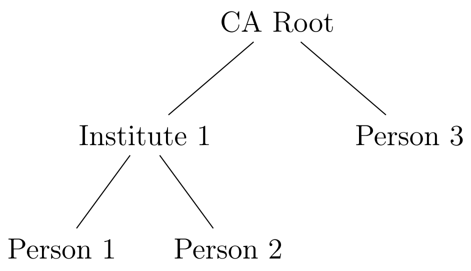


Figure 2.5.: Certificate hierarchy [from Amann, 2007, p. 21]

or universities often also have their own CA, which have to be added manually to the system by the user or the system administrator.

Certificates do not have to be signed directly by the Certificate Authority. Instead, they can be signed by an Intermediate Authority, which in turn is signed by the CA. This allows the delegation of the certification process. This is illustrated in Figure 2.5.

Companies can, for example, use certificates signed by their internal CA to authenticate users and allow them to access internal services. Each user is given a unique certificate, which is used to gain access. In this setting, an Intermediate Authority could, for example, be used for a joint project with a university department. Instead of creating a certificate for each user, a certificate which has the capability to sign other certificates (the Intermediate Authority certificate) is given to the university department. The university department can then generate certificates for all their users, which will be accepted by the company. After termination of the project only the Intermediate Authority certificate has to be revoked to revoke the access permissions of all users.

X.509 certificates are used in different distributed file systems discussed in Chapter 4.

2.9. Group Key Exchange

A problem often encountered in distributed environments is the encryption of data for a whole group of users, for example, via an encrypted broadcast or multicast. In a file system, group encryption can, for example, be used when granting file access to a user group. Group key management is a complex topic. The keys have to be changed each time a user joins or leaves the group.

The naïve implementation of a group key scheme assigns one key to each user. A central authority generates a group-key. This key is encrypted with the user-key of each user and sent to each user individually. Every time a user joins or leaves the system the group key is changed using the same mechanism. The complexity of this approach is clearly unacceptable.

Tree-based key management schemes to solve this problem have been proposed by Wong et al. [1998] and Wallner et al. [1999] almost simultaneously. Using their hierarchical approach, the key management overhead is relatively small while full forward and backward

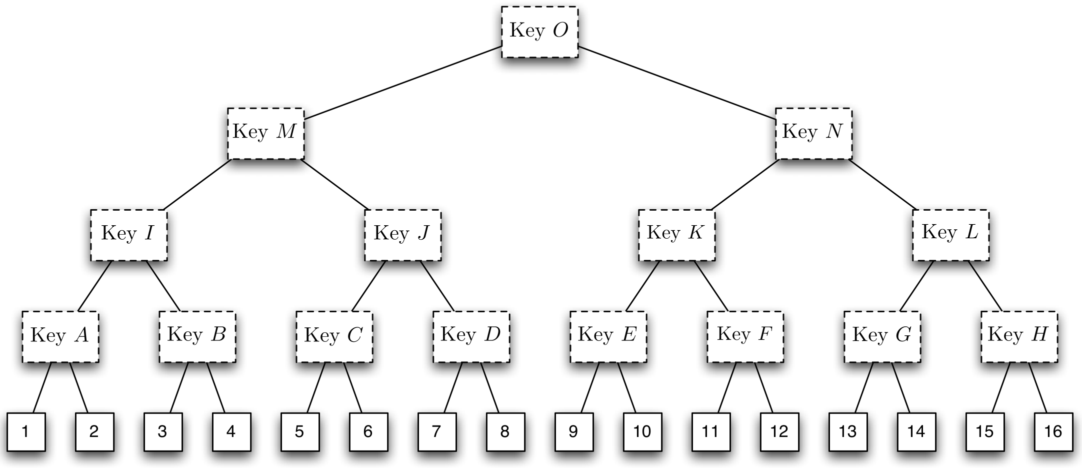


Figure 2.6.: Group key tree [adapted from Wallner et al., 1999]

confidentiality can be guaranteed. In the schemes, several keys are assigned to each participating user. Some of the keys are distributed to sets of nodes.

Figure 2.6 depicts an example key tree used for this kind of key management scheme. The inner nodes of the tree represent keys, the leaves represent users. The key O is used as the group key for the system. When a user joins or leaves the system, this key has to be replaced. For example, when user 9 leaves the group, the system generates a new version of O. This key is encrypted with the keys M, F, L and the individual key of user 10. The key is then sent to the group. After that all current members of the group have knowledge of the new group key. Using this scheme, the number of encryption operations is much lower. In the given example, the keys N, K, and E also had to be exchanged after the user left the system because the user had knowledge of these keys.

Later work by Mihaljević [2003] and Tseng [2003] significantly reduced the storage and processing overhead of such schemes.

Schemes like this already have a much lower overhead than the naïve approach; however, other recent developments like, for example, the schemes presented in the next section are even more suited to the task of group-management in distributed systems.

2.10. Subset Difference Revocation

Naor et al. [2001] proposed a new revocation scheme called the *Subset-Difference* (SD) revocation scheme. This scheme allows a publisher to send an encrypted message that every authorized receiver but no revoked receivers can decrypt. To be more exact, if $\mathcal{N}$ with $|\mathcal{N}| = N$ is the set of all users and $\mathcal{R} \subset \mathcal{N}$ with $|\mathcal{R}| = r$ is the set of revoked users, the publisher is able to transmit a message M to all users, so that any user $u \in \mathcal{N} \backslash \mathcal{R}$ can decrypt the message, but no user of $\mathcal{R}$ and no coalition of users from $\mathcal{R}$ can decrypt

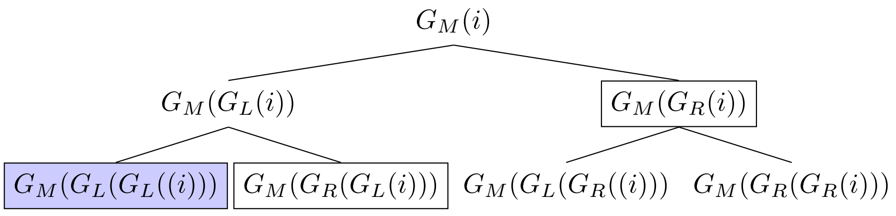


Figure 2.7.: Example tree built from i [adapted from Amann, 2007, p. 26]

it [Naor et al., 2001, p. 6]. The algorithm guarantees complete security even if all revoked users collude their keys [Lotspiech et al., 2001, p. 1].

Receivers are stateless, they do not have to store any information apart from their key material. The scheme requires the receivers to store $\frac{1}{2}\log^2 N$ keys, the message length is at most $2r$ keys and should be much shorter in most cases. The computing requirements are $O(1)$ for the receivers and $O(r)$ for the sender.

The algorithm is widely used – for example, the Advanced Access Content System [AACS] is based on it. This is the copy protection system for the BluRay and the now abandoned HD DVD format. It has been reported as broken by mainstream media several times [see e.g. Tu., 2007]. However, these reports are misleading. There is no known weakness in the algorithm itself. At the moment most of the available BluRays can be decrypted because users were able to retrieve cryptographic keys from the memory of their software or hardware DVD players. These keys are then used to decrypt the BluRay contents. This is no weakness of the revocation scheme, but only a weak point of this specific usage of the system.

The subset difference revocation scheme relies on a pseudorandom number generator G that takes an arbitrary number as input and calculates a number that is exactly three times the input's length. Assume i is the input for the generator G. The output is split into three parts; the left part $G_L(i)$, the middle part $G_M(i)$ and the right part $G_R(i)$. G has to be an one-way-function, it must not be possible to calculate i from any combination of $G_L(i)$, $G_M(i)$, and $G_R(i)$.

By using G starting with an initialization label i, a tree with an arbitrary height can be built. Each node in the tree is assigned a label. Starting with a node with label l, the label of the left child is $G_L(l)$ and the label of the right child is $G_R(l)$. The key of every node in the tree is set to $G_M(l)$. An example for a tree with height 3 starting with the label i is shown in Figure 2.7.

This tree is used as the basis for the cryptographic scheme. Every leaf of the tree represents a participant. Each node is made aware of all the keys that are not on a direct path from the root to itself.

For example, the shaded node at the lower left leaf of the tree in Figure 2.7 is made aware of $G_R(i)$ and $G_R(G_L(i))$. With these two labels the node can calculate the keys of all other nodes – except the nodes that are on a direct path from the root to it. The labels are distributed to all other participants in the same way. Hence, all participants can calculate all keys in the tree except the keys of their own node and the parent nodes.

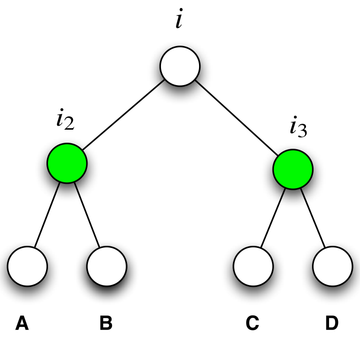


Figure 2.8.: Two roots

To encrypt a message that can be read by everyone but a specified participant, it can be encrypted with the key of the participant. For example, to encrypt a message that the shaded user cannot decrypt, $G_M(G_L(G_L(i)))$ is used as the message key. To send a message that only members of the right subtree of i can decrypt it has to be encrypted with the key $G_M(G_L(i))$.

However, this scheme fails if several users are revoked at the same time. To support this operation, the encryption scheme is expanded to use subtrees. Each subtree is initialized with a unique secret initialization number. For an example see Figure 2.8. In addition to the tree rooted at i, there are now two subtrees rooted at i_2 and i_3 which have two separate initialization numbers. The keys are distributed to the nodes in the same way for each subtree. Hence, in the tree spanned by i, A has the keys for the right part of the tree and for B. For the tree spanned by i_2, A has the keys for B. And for the tree spanned by i_3, A has no keys because A is not part of that tree.

To encrypt a message that cannot be decrypted by A and D, the message is encrypted twice – once with the key of A in i_2 and once with the key of D in i_3. Now B and C can decrypt the message, A and D cannot.

The number of keys every participant has to store is $\frac{h(h+1)}{2} + 1 = \frac{1}{2}\log^2 N + \frac{1}{2}\log N + 1$ (h being the tree height and N being the number of nodes in the tree). For a tree of height $h = 32$ the scheme can support up to $2^{32} - 1 = 4294967295$ users[3]. Each user in this tree has to save 529 keys. With a key length of 128 bits, that amounts to 8464 bytes of data.

The presented scheme is efficient for encrypted broadcasts where only a small number of the total number of users is revoked. For example, in [Amann, 2007] the scheme was used to encrypt the filesystem-wide notifications of new revisions which may be read by every current user.

The scheme, as presented here, is not especially suited for group encryption operations where the users that may access a resource are spread across the whole subset difference tree. To rectify this, several extensions to the scheme have been proposed. For example,

[3]One user always has to be revoked for the system to work – at least one key from the tree has to be chosen to be able to encrypt a message. The node with this key will not be able to decrypt the message.

Halevy and Shamir [2002] describe the Layered Subset Difference (LSD) Broadcast Encryption Scheme that allows the definition of a subset of users by a nested combination of inclusion and exclusion conditions. The number of messages (or the message length) is only proportional to the complexity of the description rather than the size of the subset.

The scheme only changes the algorithm used to select the keys that are chosen to encrypt the message; the exact details are not important for this thesis. The decryption algorithm for the LSD encryption scheme is exactly the same as for the subset difference scheme. Because it does not depend on the number of revoked users, this scheme is much more suitable for group operations. It is used for all group encryption operations in this thesis.

The paper of Mihaljević [2003] includes a comparison of the subset difference and the previously described group key management schemes. While the schemes of Section 2.9 require less key storage, they have a greater communication overhead which is – in my opinion – much more important for the described setting. The other algorithms could, for example, be preferred when the key material has to be stored on smart cards or some other device with a very limited amount of storage.

To the best of my knowledge, at the moment, no system design is known which offers a significant advantage over the SD and LSD encryption schemes. Hwang and Lee [2006] have presented another encryption scheme with similar characteristics but a lower number of keys at the client site, which is also based on a subset cover framework. However, storage at the client is of no concern for us and the system has a higher computational cost.

Boneh et al. [2005] have presented a broadcast encryption system using bilinear maps. The system also has similar characteristics to the presented encryption scheme with – depending on the exact setting – slightly lower storage overhead. The authors propose that their scheme could be combined with the presented scheme [Boneh et al., 2005, p. 271]; the more efficient one can then be used to encrypt the data on a case-by-case basis.

Other than that no significant changes seem to have occurred in the last years. Current work seems to focus on special cases such as, for example, the optimization of the schemes when the probability of revocation is not uniformly distributed over the participants [see D'Arco and De Santis, 2009].

2.11. Identity and Attribute Based Encryption

Identity Based Encryption (IBE) is a public key encryption system in which the public key can be an arbitrary identifier. The idea for IBE was first proposed by Shamir [1985] in the 80's, however, the first fully functional implementation was not available until 2001 [Boneh and Franklin, 2001].

The original motivation of Shamir was to devise a cryptosystem where email addresses can be used as public key.

In an IBE system there is a central instance in possession of a master-key. This master-key is used to derive the publicly available system parameters. The master-key is also used to generate the private keys for all users of the system.

To encrypt a message, a sender must know the public key of the receiver (i.e. his or her email address) and the publicly available system parameters. The receiver can then use the private key to decrypt the message.

Attribute Based Encryption (ABE) extends IBE to work with attributes. Each participant has a set of attributes like e.g. “student”, “math”, and “computer sciences”. Rules are given to an object, e.g. ((“student” and “math”) or (“professor”)). Only participants whose attributes satisfy the object’s rules will be able to decrypt it.

ABE was introduced first by Sahai and Waters [2004]. Goyal et al. [2006] proposed the first key-policy ABE that allows any monotone access structure. Later, ABE schemes were extended to work with non-monotone access structures [Ostrovsky et al., 2007].

Definition 2.3 (from [Beimel, 1996; Goyal et al., 2006])**.** *Let the following be a set of parties: $\{P_1, \ldots, P_n\}$. A collection $\mathbb{A} \subseteq 2^{\{P_1,\ldots,P_n\}}$ is monotone iff $\forall B, C$: if $B \in \mathbb{A}$ and $B \subseteq C$ then $C \in \mathbb{A}$. An access structure (resp., monotone access structure) is a collection (resp., monotone collection) $\mathbb{A}$ of non-empty sub-sets of $\{P_1, \ldots, P_n\}$, i.e., $A \subseteq 2^{\{P_1,\ldots,P_n\}} \backslash \{\emptyset\}$. The sets in $\mathbb{A}$ are called the authorized sets, and the sets not in $\mathbb{A}$ are called the unauthorized sets.*

Integrity and attribute based encryption could be used to implement ACLs in distributed file system. This is discussed in Section 10.3.3 on page 179.

Identity based encryption could, potentially, also be used as a replacement of the SD and LSD encryption schemes discussed in the previous section. However, the use of IBE to this end is a very recent idea. Schemes to adapt IBE for such usage scenarios have been proposed by Delerablée [2007] and Ren and Gu [2009], but they are not ready for practical deployment.

3. Peer-to-Peer Networks

The Internet is a network of millions of computers that are able to exchange messages. The systems exchanging information can usually be classified in two categories; either they use a client-server paradigm to exchange messages or they are based on peer-to-peer techniques.

Client-server based services use server systems – like HTTP servers – that offer a service that the clients can access. Many of the services used in the Internet use this paradigm; examples are the aforementioned HTTP, SMTP for mail transfer, NNTP for news, DNS for the domain resolution, etc. The problems of this approach are obvious: the server is a single-point of failure and can easily be overwhelmed with requests, for example, if there is a spike of popularity for a hosted site[1].

The second kind of systems which are handled in this thesis are Peer-to-Peer (P2P) networks. P2P networks do not differentiate between clients and servers. Information is exchanged between the different nodes and the modern P2P networks work completely without central entities. Thus, there is no central point of failure. P2P networks are inherently scalable. The capacity of the network grows with the number of nodes in the network. Thus, in the case of a popularity spike, the capacity of the network also spikes and as a consequence the load is evenly spread instead of saturating a single system. Thus, an ideal designed P2P system should never have a bottleneck.

Oram [2001] gives a definition of P2P networks, which was further detailed by Steinmetz and Wehrle [2004, 2005]:

Definition 3.1. *[a peer-to-peer system is] a self-organizing system of equal, autonomous entities (peers) [which] aims for the shared usage of distributed resources in a networked environment avoiding central services.*

Steinmetz and Wehrle [2005] further write that a P2P system is "a system with completely decentralized self-organization and resource usage".

The remainder of this chapter first gives a short history of peer-to-peer networking in Section 3.1. Section 3.2 discusses how P2P networks can be classified in different categories. Section 3.3 presents centralized P2P networks followed by pure P2P networks in Section 3.4, hybrid P2P networks in Section 3.5, and structured P2P networks in Section 3.6.

[1]Such popularity spikes are called a "flash crowd" or the "slashdot effect" [Clayton, 2006, p. 1]. For a number of other reasons why usage spikes can occur see [Clayton, 2006]

3.1. History

Oram [2001] gives a history of the development of peer-to-peer networking.

The author states that the Arpanet and the Internet until the mid-90s was mostly based on peer-to-peer paradigms. The protocols used in the network, like Telnet [Davidson et al., 1977] or FTP [Postel and Reynolds, 1985] are client/server protocols – but till the mid-90s most machines were running a client and a server at the same time. The Domain Name System (DNS) [Mockapetris and Dunlap, 1995] is another example for a system using the basic peer-to-peer paradigms. Since its inception it was scaled from a handful of machines to thousands of servers. A DNS server can act as both a server giving an authoritative reply and a client who forwards requests to other servers. As the answer propagates its way back along the request path, it is cached by the servers on the way.

Starting in the mid-nineties to 1999 the Internet radically changed. There was a great influx of people that were not scientists and just used the Internet for information retrieval. An example for the effect of this is the "eternal September" or "September that never ended" [Andrews, 2010]: Each year in September, a new wave of students was introduced to the Usenet. In the first few days, they usually were not aware of the unwritten rules of the medium, but after some time and introduction they acclimatized themselves and were no longer disruptive for the community. In 1993, America Online (AOL) began to offer Usenet access to its subscribers with a never ending amount of new users that were partially unwilling to adhere to the rules.

The Internet was also used in a much different manner by these people – they were just accessing information. Their machines usually used some form of dial-up to connect to the Internet. Hence, unlike for example, university machines, they did not have static IP addresses and they were not available at all times. Another reason for the abandonment of the peer-to-peer principle was the asymmetric nature of the connection speeds of home users. In the 90s they typically had very slow upload speeds.

The modern peer-to-peer networks appeared around the year 2000 when broadband Internet became more widely available [Eberspächer and Schollmeier, 2005, p. 35].

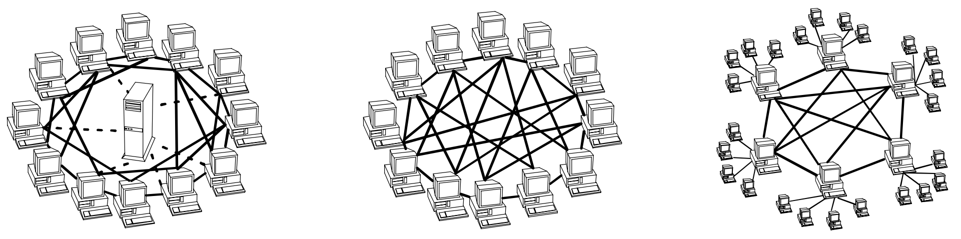
3.2. Classification of Peer-to-Peer Systems

The relevant literature uses two differing classification methods for peer-to-peer systems. In one method peer-to-peer systems are classified by the degree of decentralization. There are centralized, pure, and hybrid peer-to-peer networks (see Figure 3.1 on the next page).

In the second method peer-to-peer networks are classified by the structure of their internal network design. There are unstructured and structured peer-to-peer networks [Lua et al., 2005].

However, all common structured peer-to-peer networks are also pure P2P networks. Centralized and hybrid structures are usually only used in unstructured networks.

Because of this, this thesis mostly follows the classification of Lv et al. [2002]: The next few sections will first discuss structured networks, starting with centralized systems



(a) Centralized **(b)** Pure **(c)** Hybrid

Figure 3.1.: Different types of P2P networks [from Eberspächer and Schollmeier, 2005, p. 36]

and followed by pure and hybrid systems. After that, structured P2P networks will be introduced.

For a more detailed comparison of different peer-to-peer networks and their technologies, please see the survey paper of Androutsellis-Theotokis and Spinellis [2004].

3.3. Centralized Peer-to-Peer Networks

Centralized peer-to-peer networks use servers for some functionality combined with direct message exchange between the nodes for other functionality. A prominent example for an unstructured P2P network is Napster. Napster was one of the first P2P networks in widespread use [Kover, 2000]. It was mostly used for music sharing.

Napster uses a central directory server structure (see Figure 3.1a). Each user connects to a directory server and sends a list of files that are to be shared. Users can send search requests to this central server structure. Only the actual file-transfer is initiated between two peers.

A second, modern P2P protocol widely used today and employing a central server is BitTorrent [Cohen, 2003][2]. BitTorrent is a protocol especially suited to efficiently distribute large files or collections of files. Nodes that want to download content via BitTorrent connect to the responsible tracker. The tracker knows the IP-addresses of all other nodes that are currently downloading or seeding the content and acts as a rendezvous point [Izal et al., 2004, p. 2].

P2P systems with a central server are already much more scalable than their traditional client-server counterparts. This is, for example, evident by the rapid growth of Napster at the time – all bandwidth intensive file transmissions were carried out by the clients.

However, such systems still have a central point of failure – they do not work without their respective central servers. Thus, when these servers become overloaded, are successfully attacked or fail for any other reason, the whole network is no longer available. For

[2]Note that current versions of BitTorrent also support operation without a central server. However, this trackerless mode is mostly only used as a fallback solution.

example, Napster was shut down due to legal issues [Schmidt, 2002]. Without the server infrastructure the network was no longer operational. Networks without central servers do not have these issues – however, many operations are far easier to implement on a centralized infrastructure. For example, searching on a central server is an easy operation – when a search query arrived at a Napster directory server the server could answer it directly. Searching in networks without central servers is much more complicated – as are many other operations.

3.4. Pure Peer-to-Peer Networks

Pure P2P networks without a central server surfaced a short time after Napster. The typical hierarchy of such networks is shown in Figure 3.1b on page 41. These systems work without a supplementary central server hierarchy – all functions are carried out by the nodes in the network.

Because these systems work without any central servers there are certain complications in comparison to server-driven P2P networks.

The process of joining a peer-to-peer network is called *bootstrapping*.

To join the network, a new node has to contact another node that is already part of the network. But the number and addresses of the systems that are part of the network are in constant flux. There are several solutions for this problem, like e.g. periodically releasing lists of stable nodes on a website, random address probing [see Cramer et al., 2004] or even methods like bootstrapping over IRC [Knoll et al., 2009].

Searching for data without a central server also poses a problem. Because there is no central entity knowing which data is stored on which node, searching in unstructured networks is typically realized with a flooding mechanism. The search query is simply forwarded to all or some neighbors which in turn forward the query until the specified piece of data is found or the query packet's time-to-live expires.

This is not a reliable way to search for information. Same search queries may (and often will) give different results if they are run multiple times because they take a different way through the network each time.

3.4.1. Gnutella

[Gnutella] was one of the first pure peer-to-peer networks[3] [Lu and Callan, 2003, p. 200]. Gnutella uses a robust, albeit inefficient, flooding technique for its search queries, discussed above. This results in a constant background traffic; an exact analysis of this traffic is given by Schollmeier [2004, p. 9]. Due to its routing algorithm, Gnutella also leads to a high load on transatlantic connections. In 2003, about one third of the connections in the network were from the USA to Europe and vice versa [Schollmeier and Kunzmann, 2003]. Hence, queries often crossed the Atlantic multiple times while being routed from their source to their destination.

[3]This section discusses Gnutella v0.4. A later version of Gnutella (v0.6) uses a hybrid network approach.

Peers join the Gnutella network by randomly connecting to an average of three other hosts [Eberspächer and Schollmeier, 2005, p. 43] offering a decentralized search service. This can lead to separated subnetworks with no links between each other [Eberspächer and Schollmeier, 2005, p. 43].

To reduce the high message load of Gnutella, later versions of the protocol (more specifically Gnutella v0.6) use a hybrid routing approach with a hub based network [Eberspächer and Schollmeier, 2005, p. 49].

3.4.2. Freenet

Freenet [Clarke et al., 2001, 2002] is a very well-known P2P network that permits the anonymous publication, replication and retrieval of data.

This section explains Freenet up to version 0.6. Freenet 0.7 is a complete rewrite and works quite differently. A darknet [Sandberg, 2006] approach was taken there.

In Freenet, files are identified by a 160 bit key. This key is derived by SHA-1 hashing a file or key. There are several different types of keys to access the stored data. The two most important ones are CHKs (Content Hash Keys) and SSKs (Signed Subspace Keys).

A CHK is a key that identifies static content in Freenet. A typical CHK key looks like this [Freenet]:

```
CHK@file hash, decryption key, crypto settings
```

Signed Subspace Keys are used for data that is going to change over time. SSKs use public-key cryptography. Only nodes knowing the private key can insert new file versions.

A typical SSK key looks like this [Freenet]:

```
SSK@public key hash, decryption key, crypto settings/name-version
```

The file hash key is computed by taking the public key hash, XOR'ing it with the descriptive string (everything following the slash in the URI) and hashing the result once again [Clarke et al., 2001, p. 50].

This means that the key for the stored data is derived from only a version string and the public key of the publisher of the information.

Freenet uses a steepest-ascent hill-climbing search. An example query is depicted in Figure 3.2 on the following page. The destination of a request is always a 160-bit key. Each node remembers which keys it received from which nodes. When a node receives a query, it is routed to the node from which the most similar key has been previously received. If that request fails, the second most similar node is chosen and so on.

Data retrieval and data insertion requests are handled in the same way. Data is inserted several times so that it is stored on different nodes in the network.

The rationale behind the routing algorithm is that nodes should specialize on a part of the keyspace because they will receive insertion and retrieval requests for a small part of the ID space.

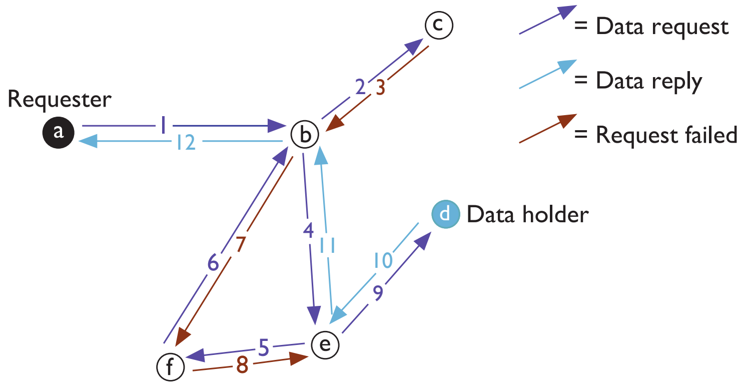


Figure 3.2.: Routing in Freenet [from Clarke et al., 2002]

3.5. Hybrid Peer-to-Peer Networks

Hybrid P2P networks were the next evolutionary step to compensate the mentioned problems of pure P2P networks. These network protocols try to combine the advantages of pure P2P systems with those of centralized P2P systems. In hybrid P2P networks, certain well-connected servers get a special status. These nodes are often called *supernodes* or *superpeers*. They are well interconnected and are responsible for a certain number of other normal nodes. A typical hierarchical P2P structure is shown in Figure 3.1c on page 41.

Supernodes handle certain tasks that are more suited to be handled centrally. For example, searches can be handled more efficiently when a supernode is present: each node sends its list of available files to its supernode. When a node searches for a file, it sends the search query to its supernode. The supernode forwards the query to the other supernodes so that the query can be answered without consulting any normal node. This is a much more efficient way than in pure P2P networks, but also opens up certain vulnerabilities not present before. One can, for example, try to block the access to the supernodes – which are relatively limited in number – and therefore diminish the speed or usability of the P2P network [see Lowth, 2003].

Examples for networks using hybrid P2P protocols include Gnutella 0.6 [Klingberg and Manfredi, 2002], eDonkey2000 [Kulbak and Bickson, 2005], and FastTrack (most prominently used by Kazaa) [Leibowitz et al., 2003]. Another very widely used hybrid network is Skype [Baset and Schulzrinne, 2006]. The basic structure of a Skype network is shown in Figure 3.3 on the facing page. Skype nodes first contact a centralized login server. This is the only centralized step; the rest of Skype is a pure peer-to-peer network with a supernode structure.

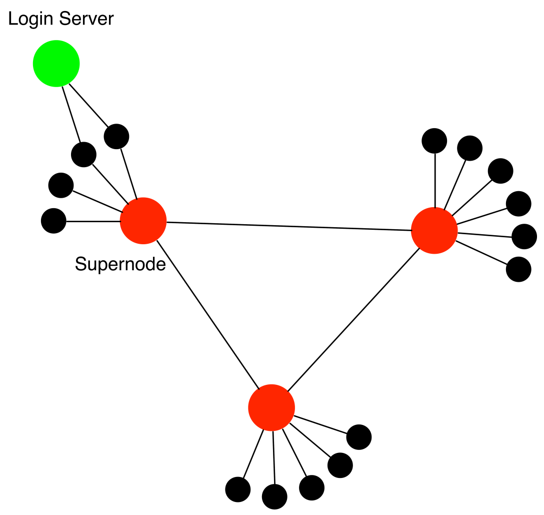


Figure 3.3.: Skype network structure [adapted from Baset and Schulzrinne, 2006]

3.6. Structured Peer-to-Peer Networks

In contrast to the unstructured P2P networks of the previous sections, structured overlay networks organize their nodes in a predetermined graph structure. This graph structure "allows them to locate objects by exchanging $O(\log N)$ messages where N is the number of nodes in the overlay" [Dabek et al., 2003].

The principal functionality offered by all structured overlay networks is a *Key Based Routing service* [Dabek et al., 2003]. A KBR service supports a single operation `route(key, value)` which routes the message with the content `value` to the node with the ID `key`. Such a KBR service can then be used to implement other services such as Distributed Hash Tables (DHTs), anycast or multicast applications. It is also possible to support several applications at once on top of one KBR service [Di et al., 2008].

3.6.1. Distributed Hash Tables

The most common application used in a structured peer-to-peer network is a Distributed Hash Table (DHT) [Wehrle et al., 2005, p. 84].

The basic idea of a distributed hash table is that a unique identifier is assigned to every piece of data. Often this identifier is derived by applying a collision-resistant hash function like SHA-1 to the data in question [Wehrle et al., 2005, p. 85]. Each node of the network node also gets a unique identifier in the same address space and is responsible for a certain interval of addresses near its node address. The exact interval a node is responsible for varies: a few P2P systems assign all identifiers that are larger than the node ID but smaller than the ID of the next node to the node in question. It is assumed that each node is responsible for all identifiers that are numerically closest to its ID. The

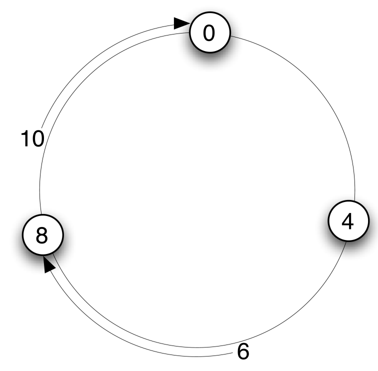


Figure 3.4.: Keyspace assignment in Chord

node-identifiers are usually computed by hashing certain pieces of information that are specific to the node.

3.6.2. Chord and Chord-Variants

Stoica et al. [2001] proposed a structured overlay network named Chord.

In Chord, each node is assigned an m-bit identifier which is computed by hashing the node's IP address. All nodes are then organized into a virtual ring structure. An example for this is depicted in Figure 3.4. Each node in the network is responsible for the part of the ID space following its predecessor, i.e. [nodeID; predecessor), with a wrap-around at 0. This is also depicted in the Figure 3.4; e.g. node 8 is responsible for IDs 5, 6, 7, and 8.

When a new node joins the network the responsibilities change. Nodes that previously were responsible for the identifier ranges affected send the stored data of the new node's ID space to it. When a node leaves the network, it sends the affected data to the nodes that assume responsibility for the leaving node's ID space. With this technique only very few nodes are affected when a new system joins or leaves the network. This approach is called "consistent hashing" and was presented in [Karger et al., 1997; Lewin, 1998].

Chord routes messages around the ring in one direction, from the lower to the higher node IDs. To minimize the amount of messages required to route a message to its destination, there are shortcuts – the message is not routed via all nodes between the sending and the receiving node. Instead, each node has a *finger table* containing the information of up to $\log N$ other nodes that can be directly contacted (with N being the number of nodes in the network). The farther away the node ID from the local node, the sparser the entries in the table.

Using this finger table, messages can usually be routed with a maximum of $\log N$ steps.

Some peer-to-peer networks with similar architectures to Chord are also quite well known and used by systems discussed later in this thesis:

Pastry by Rowstron and Druschel [2001a] uses the same underlying network structure as Chord. The Pastry ring has 2^{160} identifiers. One difference from Chord is the keyspace allocated to each node. Each node is responsible for all the keys nearest to the identifier – no matter if they are smaller or bigger than the node Id. Routing also works a little differently, but the basic design principle is the same. Pastry has a routing table which is coarser for more distant nodes. However, Pastry also uses network proximity for these distance calculations.

The Distributed K-ary System (DKS) [Ghodsi, 2006] is a structured peer-to-peer network using a ring structure; the routing algorithm is similar to that of Chord. In contrast to Chord, the message overhead for network stabilization is reduced. Furthermore, the join and leave operations of DKS are atomic; thus, unlike in Chord there is no time window where the routing is inconsistent. DKS also has got more built-in features like a DHT implementation and automatic replication.

3.6.3. Kademlia

Kademlia was proposed by Maymounkov and Mazières [2002] and is the only structured overlay network with a significant user base at the time this dissertation is written [Locher et al., 2010, p. 195].

Each node in Kademlia has a 160 bit node ID, which is constructed using the exact same method used in Chord. The distance between two peers in Kademlia is measured using a XOR metric; the distance between x and y is

$$d(x, y) = x \oplus y$$

The routing method used is illustrated in Figure 3.5 on the next page. It has great similarities to the subset difference encryption scheme already discussed in Section 2.10 on page 33. Each node keeps information about nodes in subtrees that do not match its path. Hence, the black node in Figure 3.5 knows at least one node in each of the dotted squares. Thus, each node which forwards a message can halve the remaining identifier space that remains to be searched.

Information about nodes in other parts of the trees is piggy-backed on messages that are sent via the network. Kademlia keeps the information about nodes in other parts of the tree in buckets for the respective tree-part. When a bucket is full, old nodes are deleted from it. If a message from a node that is already in a bucket is received, it is moved back to the list. The reasoning behind this is that nodes that have a long uptime tend to stay alive even longer.

3.6.4. CAN

Ratnasamy et al. [2001] proposed the Content-Addressable Network (CAN). CAN orders its nodes in a d-dimensional coordinate space which is split into zones. Each node is responsible for the coordinate space in its zone. Neighboring nodes have connections to each other. Thus, each node has connections to $2d$ neighboring nodes; the average path length is $\frac{d}{4}n^{\frac{1}{d}}$. When more nodes join the network, the node state stays constant, only

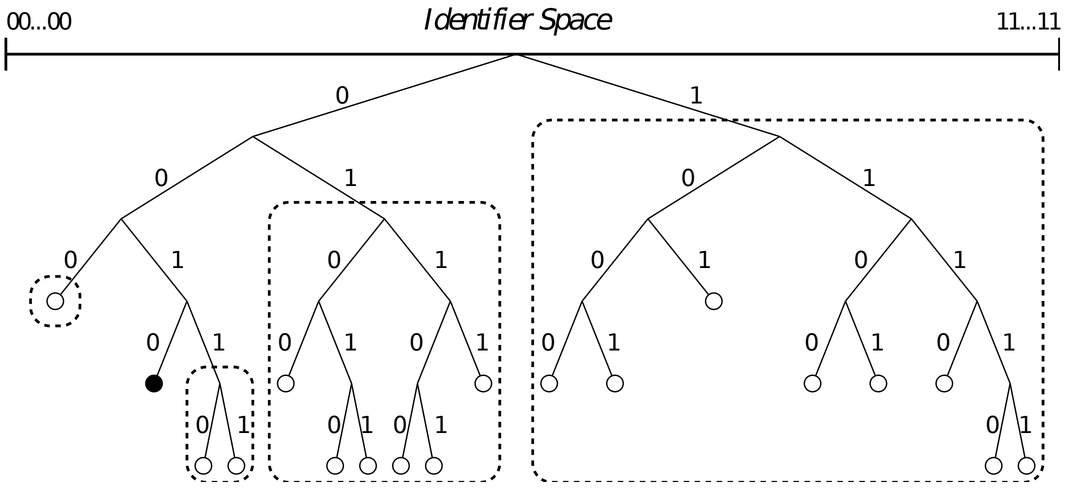


Figure 3.5.: Kademlia tree [from Götz et al., 2005, p. 114]

the path length increases. When new nodes join the network, the part of the coordinate space is split between the two nodes. Messages are routed to a d-dimensional coordinate using a simple greedy algorithm. A message has reached its destination when it arrives at the node responsible for the zone in which the coordinate lies.

Figure 3.6 on the next page shows a sample 2-dimensional CAN network with a message that is being routed between two nodes.

CAN has several advanced features that can be used to increase the network stability. It is, for example, possible for each node to be part of several different CAN networks called *realities*. The content of the network is replicated into each reality. This greatly increases the availability and decreases the average route length.

CAN also supports other features to increase network reliability such as zone overloading, where every zone is assigned to multiple nodes that keep each other in sync.

3.6.5. Disadvantages of Structured Peer-to-Peer Networks

Structured peer-to-peer networks have got certain inherent design disadvantages. They provide an efficient means of routing data across the network and lookups are guaranteed to yield an exact result, but this comes at a price: the most notable shortcoming of structured peer-to-peer networks is the absence of fuzzy searches. These are very easy to implement in unstructured P2P networks, but a big challenge in structured ones, which usually only allow the lookup of exactly identified pieces of information [see Harren et al., 2002].

Another disadvantage of structured peer-to-peer networks is that, in their basic form, they are very susceptible to different kinds of attacks. Locher et al. [2010] examine the resistance of the Kademlia network, which at the moment is the “most popular” and “only widely used peer-to-peer system based on a distributed hash table” [Locher et al., 2010, p. 195] against several types of attacks. Most of these attacks are special forms

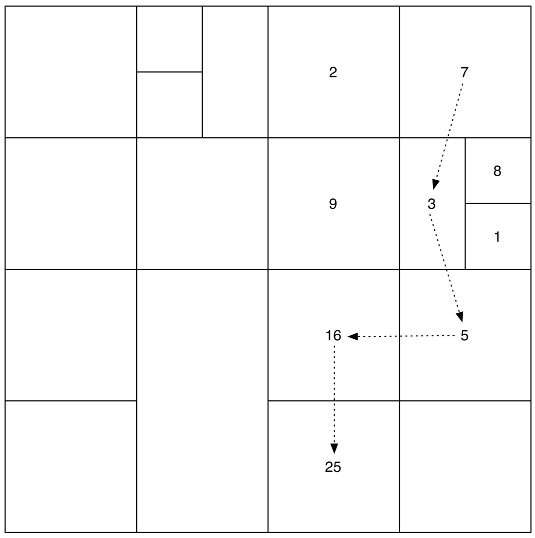


Figure 3.6.: Routing in CAN

of the *sybil attack* [Douceur, 2002], where one host connects multiple times to the same peer-to-peer network to be able to quickly insert bogus information or to take over big parts of the ID space. While the evaluation is specific to the Kademlia network, most of the attacks also should work on other structured peer-to-peer networks.

Kademlia is susceptible to *node insertion attacks*, where an attacker can freely choose the node ID and thus assume responsibility for a specific part of the Kademlia ID space. The attacker can then return false results for search queries that fall into its responsibilities. Note that e.g. Chord is not susceptible for this kind of attack: in Chord, the node ID depends on the current node IP [Dabek et al., 2001b, sec. 4.4].

Another attack presented is a *publish attack*, where the index tables of specific nodes are saturated so that they will only return bogus entries. Thus, the publication of information under specific search queries can be prevented.

A third kind of attack is an *eclipse attack*. Here a specific host is prevented from communicating with the Kademlia network altogether without the host noticing.

All attacks presented in [Locher et al., 2010] can be carried out with a very small resource usage.

Recently several Kademlia clients introduced protection mechanisms against certain kinds of sybil attacks. However, the described attacks do not depend on a massive amount of connected hosts and thus are not prevented by the mechanisms. For an in depth review of the Kademlia sybil attack protection, please see the work of Cholez et al. [2009].

4. File Systems

Computer systems are used for all kinds of data storage, processing, and retrieval. Some data in a computer system is only processed for a very short amount of time and only stored in volatile memory. The case handled in this chapter is data that is stored on a computer system for a long amount of time.

Tanenbaum [2001, p. 379] gives three *essential requirements* for long-term data storage:

1. It must be possible to store a very large amount of information.

2. The information must survive the termination of the process using it.

3. Multiple processes must be able to access the information concurrently.

In today's computing environments usually a file system is used to satisfy these requirements. File systems consist of *directories* which are sometimes also called *folders*. These directories can in turn contain other directories or *files*. A file is a structure that can contain any kind of binary data.

Figure 4.1 on the following page shows part of an exemplary directory structure as it can be found on a typical Unix system today. Files are shown with dotted borders. The structure contains several files and directories. Note that the structure is not a tree structure. One of the files named `project.tex` is present in two directories.

This is called a *hard link* because the actual data is referenced twice within the file system. Most modern file systems allow such structures. The directory structures of file systems that support hard links can be represented with a *directed acyclic graph* [Tanenbaum, 2001, p. 408]. The directory structure of file systems that do not support hard links can be represented with a tree structure.

A second type of link that is supported by most modern file systems is a *soft link* or *symbolic link*. Soft links are files that contain the name of the file or directory to which they are pointing. The redirection is resolved by the operating system on access. In contrast to hard links, soft links can be broken when the target object no longer exists. Broken links are still present in the file system but point to a non-existing location. Soft links can also point to directories whereas hard links can only point to files. Furthermore, soft links can span file systems and point to objects residing on a foreign file system.

Some file systems also support *variant* or *context dependent* symbolic links. Variant symbolic links, also known as *variable* symbolic links, may contain a variable in the name of the object to which they are pointing. When the link is accessed, the content of the variable is inserted into the path name [see DragonFly]. Context dependent symbolic links can embed context like the system hostname in the name of the object to which they are pointing [see e.g. Hancock, 2001, p. 13].

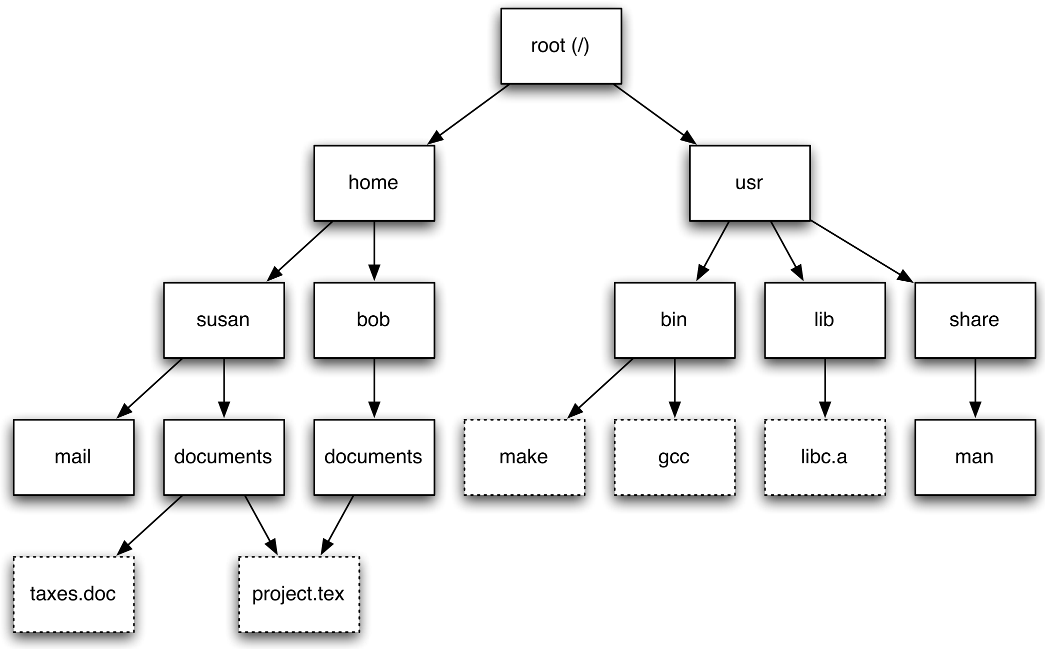


Figure 4.1.: Part of a Unix directory structure

This chapter presents the design of different local and networked file systems. A special emphasis is given to systems that handle cryptographic permissions, use encryption to secure their contents, or use special redundancy systems. Note that this chapter only handles systems which export a file-system like API to the user. Systems which offer a block-level device to the user, which is then formatted using a file system, are not handled. These systems are often used when encrypting local storage, examples are the CryptoGraphic Disk Driver [Dowdeswell and Ioannidis, 2003], FreeBSD's GEOM or Linux's loopback mounts. These systems are an alternative to systems like NCryptfs, which are presented in this paper – however, they operate on a completely different layer and do not offer features like individual user keys.

There are also systems which provide a block-level device in a networked environment; examples are iSCSI or Network-Attached Secure Disk (NASD) [Gibson et al., 1998; Gibson and Van Meter, 2000]. Those are also not handled in this thesis.

The remainder of this chapter is structured as follows: Section 4.1 gives an overview of local file systems like ext2. Section 4.2 presents server-based distributed file systems like NFS. Section 4.3 presents cluster file systems, like GFS, which do not have a strict server infrastructure like NFS, but still trust the other nodes in the network. Section 4.4 covers peer-to-peer or decentralized file systems like Plutus or Ivy, where the other file system nodes are untrusted. Finally, Section 4.5 presents a feature overview of the file systems presented in this chapter.

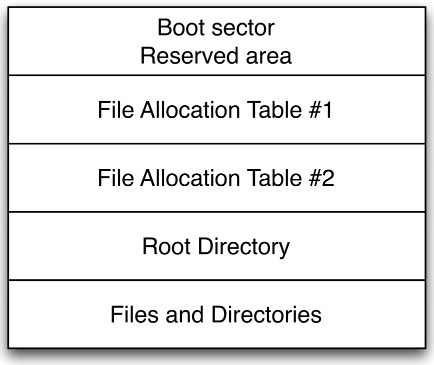


Figure 4.2.: Typical map of a MS-DOS file system. [adapted from Duncan, 1988, p. 179]

4.1. Local file systems

This section first presents two local file systems: the MS-DOS FAT File System, which is still very widely used today and the System V Unix File System, which is very similar in structure to current file systems for Unix-like system. The basic mechanisms and structures used in these file systems are also used in many of the later distributed and decentralized file systems.

4.1.1. MS-DOS File System

The MS-DOS FAT file system is named after the File Allocation Table – which is one of the most important concepts of the file system. Variants of the original FAT file systems are still in use today; they are, for example, often used on external media like USB-drives or the memory cards in digital cameras or MP3 players. Today, FAT32 is usually used.

The concept of a File Allocation Table was conceived by Bill Gates and Marc McDonald in 1977 "as a method of managing disk space in the NCR version of standalone Microsoft Disk BASIC" [Duncan, 1989].

Figure 4.2 shows the layout of a typical MS-DOS file system: The logical sector 0 of each FAT file system is the *boot sector*. If the hard drive is chosen as the boot-disk and the partition is marked as active, the boot-sector is automatically read to memory and executed. Hence, it contains the bootstrap routine, a short program to boot the operating system, which is stored on the file system. It also contains the important media characteristics such as the number of tracks and heads, the physical drive number, the number of bytes per sector, etc. Furthermore, it contains some other pieces of information like the number of FATs. For a full description please see Duncan [1988, p. 180].

Following the boot block is the first File Allocation Table. The FAT is a table with 12, 16 or 32 bit entries, depending on the medium size and the MS-DOS version number. The first two entries of the FAT contain the media descriptor byte, which can also be found in the boot-sector. The rest of the entries describe the data contained in the corresponding disk clusters. The possible values are described in Table 4.1 on the following page.

Entry	Description
Cluster available	Cluster is available and can be used to store data. When some new data is written to the file system, the operating system searches the FAT for this type of entry and stores the data in the corresponding disk cluster.
Reserved cluster	The cluster is reserved by the operating system.
Bad cluster	Corresponding on-disk cluster is damaged and should not be used.
Cluster-number	If an entry contains the number of another cluster that means that the corresponding cluster is used to store a file or directory. The number of the cluster given in the FAT is the number of the next cluster of the file or directory.
Last cluster	Cluster is in use for a file or directory and is the last cluster in the cluster-chain.

Table 4.1.: Possible FAT entries [see Duncan, 1988, p. 183]

Usually the FAT is followed by a second exact copy of the FAT which is used in case the first FAT is damaged. The number of FATs is set in the boot-sector; however, a number other than 2 is unusual.

The root directory of the file system is stored in the sectors adjacent to the FAT. Each directory is a list of entries. Each entry consists of a fixed-size file name, some file attributes and the starting cluster of the object. Later versions of the FAT file systems removed the fixed-size limitation of the file name.

4.1.2. The Unix File System

Many file systems used in Unix variants use the basic principles of the Unix File System of System V.

The design is very different from the FAT file system architecture. Like FAT, each Unix file system begins with a *boot block* which contains a basic *boot loader*. The boot loader enables the system to boot the operating system stored on the file system.

After the boot block, there is a *super block*, which describes the file system states such as the number of files that it can store, the size, etc.

The superblock is followed by the *inode list*. An *inode* describes an object stored in the file system and contains the information about the owner, group, permissions, size and the disc addresses at which the stored information can be found. The number of inodes is set when the file system is first created. When all inodes in the inode list are used, no new objects can be stored in the file system.

The inode list is followed by the data space where the actual file system blocks are stored.

The way disk blocks are referenced by an inode is shown in Figure 4.3 on page 56. Each inode can directly reference 10 data blocks on the disk. If the file is bigger, the inode points to another data block which contains the addresses of more data blocks. Usually one block can contain the addresses of 256 other blocks. If the space of this block is also exhausted, there are two more slots for double and triple indirect addressing – where the indirection scheme is layered over even more blocks (see Figure 4.3 on the following page).

To find free space in the file system, a linked list of free disk blocks is used. The first entries of this list are stored in the super block.

This basic design was enhanced many times e.g. for the BSD Fast File System (FFS) [McKusick et al., 1984]. However, the underlying design of all these systems is the same. The file system design is still used very often today e.g. in the Second Extended File System (ext2) [Card et al., 1986], which is the predecessor of the ext3 [Tweedie, 1998; Prabhakaran et al., 2005; Cao et al., 2005] and ext4[Mathur et al., 2007] file systems, which still are the standard file system of many Linux distributions[1]

The inode design of FFS, especially, is also used in many distributed file systems that will be discussed later in this chapter. For a thorough comparison of FFS and the MS-DOS file system see [Forin and Malan, 1994].

4.2. Single-Server based Distributed File Systems

"A *distributed file system (DFS)* is a distributed implementation of the classical time-sharing model of a file system, where multiple users share files and storage resources [...]. The purpose of a DFS is to support the same kind of sharing when the files are physically dispersed among the sites of a distributed system." [Silberschatz et al., 2005, p. 641]. Or to formulate it less formally: "A DFS is a file system whose clients, servers, and storage devices are dispersed among the machines of a distributed system" [Silberschatz et al., 2005, p. 642].

Distributed file systems have been extensively discussed by the research community. Thus, there are a great number of different designs, each fulfilling different specific requirements.

This section gives an overview of the server-based distributed file systems. The next sections will then introduce distributed file system designs that work without depending on a fixed server infrastructure. A special emphasis is put on the details of distributed systems that are relevant for the topics of this thesis.

Using computer networks for resource sharing of all kinds is an old idea [see for example Roberts and Wessler, 1970]. The first predecessors of today's network file systems were conceived in the 1970s.

An example of this is the Datacomputer [Marill and Stern, 1975]. It was used on the Arpanet [Heart et al., 1970]. The Datacomputer was not originally devised as a distributed file system; it was able to provide much higher level data sharing services. However, starting in 1973, it was used by a computer center on the Arpanet for remote

[1]There are a number of differences between the ext2 and the ext3 file system. One of the most known features is journaling; other features include support for extended attributes, etc. ext4 introduces new block mapping and delayed allocation algorithms and can use hash trees for directory indexing. Nevertheless the basic internal structure of the systems is very similar to ext2 and the presented System V Unix File System.

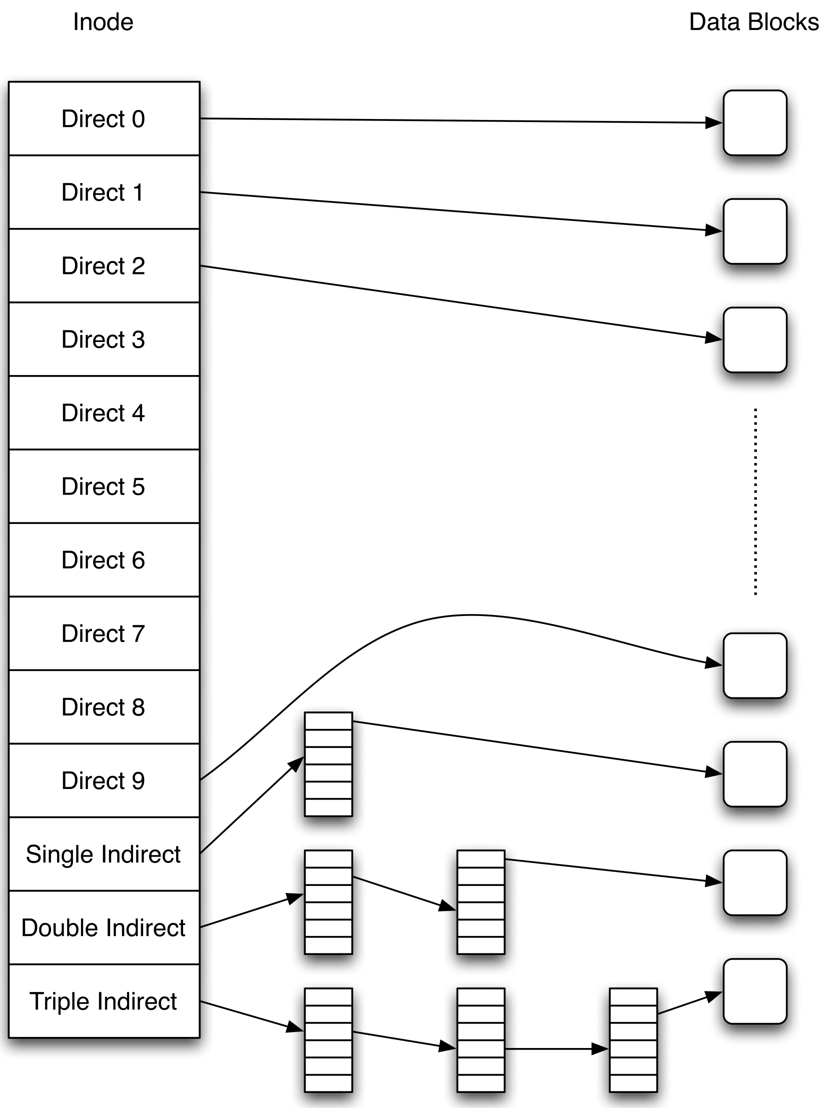


Figure 4.3.: Inode block references [adapted from Bach, 1986, p. 69]

file storage; the local users could transparently store files on the remote nodes [Marill and Stern, 1975, p. 394].

Xerox also built several of the first distributed file systems at their Palo Alto Research Center. For their Alto workstations they built the Juniper file server which was never used extensively because of poor performance [Thacker, 1986, p. 96]. The much simpler Interim File Server (IFS), which was written in 1976 for the same system, however, was so successful that it provided the majority of the file storage in the Xerox network well into the 80s [Thacker, 1986, p. 96]. The Woodstock File System [Swinehart et al., 1979], which was conceived in 1975, was another distributed file system by Xerox with a very basic protocol. The goal was to shift responsibilities and complexity from the server to the client to save computing power on the server side.

Other systems of the time were:

- The Xerox Distributed File System (XDFS) [Mitchell and Dion, 1982] already featured two-phase commit protocols, repeatable operations, etc.
- Alpine [Brown et al., 1985] was developed because XDFS tended to crash too often and used much memory.
- The Cambridge File Server [Dion, 1980] was very efficient and also featured file locking and a capability-like access control mechanism.
- Helix [Fridrich and Older, 1985] was a distributed file system that grouped foreign disks, floppies, etc. into objects.
- Swallow [Reed and Svobodova, 1981] was developed at MIT and handles both local and foreign stored objects. It put a special emphasis on atomic operations.

[Svobodova, 1984] gives a detailed comparison between many more distributed file systems of that time; [Levy and Silberschatz, 1990] gives a comparison of the more modern distributed server-based file systems.

The remainder of this chapter concentrates on the server-based distributed file systems which are still in use today.

4.2.1. NFS

The distributed file system most well known in the Unix world is undoubtedly the Network File System (NFS) [Callaghan et al., 1995]. It allows remote clients to access shared file systems on a server across network boundaries.

NFS was designed by Sun Microsystems (now Oracle) starting in March 1984 [Sandberg et al., 1988]. The design goals [Sandberg et al., 1988] were machine and operating system independence, crash recovery, transparent access, Unix semantics maintained on client and a reasonable performance. These goals are still valid for today's distributed file systems.

The protocol was designed for a stateless server. Each file system call contains all the information the server needs to fulfill the request. Hence, the server does not need to maintain any information about the clients that are currently accessing the file system. This makes crash recovery very easy. In the case of a crash, a client just needs to retry the request until the server becomes available again. It also simplifies the protocol – NFS is very easy to implement and is used as a front-end for some of the file-systems that are discussed in the remainder of this section.

NFS uses Remote Procedure Calls (RPC) [Srinivasan, 1995] to execute the requests on the file server. The RPC calls are all very simple and intuitive. For example, to read a portion of a file the `read(filehandle, offset, count)` RPC call is used which returns the requested data and the file attributes.

Because of the age of NFS, it lacks a few features that are desirable in today's environments. The default security system of NFS relies on permission checks on the client side – the server does no credential checking on its own. While there exist extensions to remedy this situation they are not mandatory and the implementations have interoperability problems. By default, NFS transfers its data using UDP; usually many of the packets get fragmented. This makes it difficult to use NFS when it has to cross firewalls – it is very easy to get the configuration wrong and quite difficult to debug such errors. For a more thorough analysis of the problematic points of the NFS design, see [Kirch, 2006].

4.2.2. NFSv4

NFSv4 [Shepler et al., 2000] is a major redesign of NFS. It no longer features a stateless server and therefore also has open and close operations. Permissions can now be verified on the server. The protocol also features file locking, which was not a part of the original NFS protocol. For a summary of the changes introduced by NFSv4 see for example [Harrington et al., 2006].

Due to these changes, the protocol is much more complex than the previous versions of NFS. It took a long time to be available on the different operating systems. Even today there are problems using some features that, according to the NFS specification, have to be provided. An example of this is Kerberos [Neuman and Ts'o, 1994] authentication over NFSv4. While in theory it is supported on most systems, in my experience there are problems when mounting Kerberos secured NFSv4 volumes with different Linux distributions.

4.2.3. Row-FS

The Redirect on Write File System (Row-FS) [Chadha and Figueiredo, 2007] is a modification of the NFS protocol to a more distributed file system. Row-FS uses a simple read-only NFS server. Writes are not committed to this NFS server – instead they are redirected to a local server and committed there. Subsequent reads to this modified data are also redirected to the local server. This allows the system to have several long running applications such as simulations that modify the data at the same time. After these applications terminate, the modified files can either be kept locally or committed back to the central storage. Row-FS also has several other modes of operation where not only the locally changed data is stored at the local server, but where, for example, all data that has ever been read is cached locally.

4.2.4. Cepheus

Cepheus [Fu, 1999] was one of the first systems where cryptographically enforced access permissions in a distributed file system were discussed. Cepheus puts a special emphasis on group operations. Cepheus depends on a central storage server infrastructure.

Cepheus enforces the traditional Unix-like permissions using cryptography. The group membership lists and group access rights are stored on a central group database server, which is also responsible for keeping and transferring the current group key to the users (using public key encryption). Cepheus introduced the notion of *lazy revocation* – though in the thesis it is called *delayed re-encryption*. In a system using lazy revocation, a revoked user may still read data which she was able to read at the time she was evicted from the system. The user may, however, not read newly written or changed data; this data is written using new cryptographic keys. The concept of lazy revocation was adopted by many storage systems like e.g. SiRiUS, Wuala and Plutus. Cepheus does not support ACLs.

Cepheus protects the integrity of the files stored in the system cryptographically – this includes the file contents and the file metadata. However, a group member with read access can collude with a storage server to circumvent this protection.

4.2.5. NCryptfs

In contrast to the previously discussed file systems, NCryptfs by Wright et al. [2003] is a cryptographic file system that is layered on top of an existing file system like, for example, NFS. All data that is stored into NCryptfs is first encrypted and then stored in a directory on the existing file system. NCryptfs can protect the data against network sniffing and server corruption when layered on top of a network file system. Because of this usage scenario, NCryptfs is presented in this section and not in the section about local file systems. In case of a local file system, the encryption protects the data when the computer is stolen.

NCryptfs is the successor of a simpler cryptographic file system called CryptFs [Zadok et al., 1998]. NCryptfs was developed with a stackable file system toolkit called FIST [Zadok and Nieh, 2000].

NCryptfs is first mounted by the superuser, usually to the directory `/mnt/ncryptfs`. Users can then attach their encrypted file systems as subdirectories of this directory using an operation which is similar to a user-mode mount in Unix. When attaching a file system, a user has to enter the file system password. The encryption key is derived from this password using a Password-Based Key Derivation Function; more specifically PKCS#5 PBKDF2 [Kaliski, 2000] is used.

NCryptfs supports group accesses to files. A file owner can give other users access to a file. To this end, the user who is given access first gives a hashed and salted representation of his personal passphrase to the file owner. The file owner then stores this hash in his configuration file. This salted hash is then used to authenticate the other user.

This scheme requires the file owner to be online when another user wants to access one of its files. It also does not work across machine boundaries because the owner's encryption key is only present on the machine on which she is currently active.

NCryptfs also supports the mapping of Unix groups to NCryptfs groups. NCryptfs does not feature any metadata or file integrity protection.

For a more in depth overview of NCryptfs and a comparison of NCryptfs with many other cryptographic file systems (some of which are also covered in this thesis) please see the survey paper of Kher and Kim [2005].

4.2.6. TCFS

The Transparent Cryptographic File System (TCFS) by Cattaneo et al. [2001] is a kernel-based cryptographic distributed file system sitting between the VFS layer and the storage file system. Like NCryptfs, TCFS only changes the application data and not the underlying structures; it can thus be layered on top of any kind of Unix file system. The available Linux implementation is an extended NFS client. TCFS can be used in a NFS environment without a need to change the server infrastructure. On 4.4BSD TCFS can be used as a file system layered and stacked on top of the local files system.

In TCFS the servers are untrusted. All encryption and decryption operations are made on the client side. Only encrypted data blocks are transferred over the network.

Each user has a *master key* which is saved in a system wide database and encrypted with the user password. For bigger systems the master key can also be stored using Kerberos. Each file is encrypted with a *file key*. The file key of each file is encrypted with the master key of the user and stored in the header of the file. Each block of an encrypted file is encrypted in CBC mode with a different *block key* which is derived by applying a hash function to the concatenation of the file key and the block number. For each file an authentication tag is computed, which is generated by hashing the concatenation of all block contents and block keys. The authentication tag is also stored in the file header.

Hence, each file can only be decrypted by its owner. The owner can verify the validity of the file contents by recomputing the authentication tag and comparing it to the tag stored in the file header. If the two values do not match, the file contents were altered. This does protect the file integrity against tampering, however, it does not protect against rollback attacks.

TCFS includes a group sharing variant called *threshold sharing*. Threshold sharing means that a minimum number of users need to be online in order that the files that belong to the group can be accessed. This is enforced by generating a group encryption key which is shared using a *threshold secret sharing scheme* proposed by Shamir [1979]. The group encryption key can only be reconstructed when enough group users that know parts of the key are online. When the number of online users falls under the threshold, the group encryption key is removed from the memory of the machine that did the reconstruction. This is an easy point of attack for a malicious user – thus, the authors recommended to replace this algorithm with a distributed alternative in the future. As far as I know, this feature was never added to TCFS.

4.2.7. SNAD

Secure Network Attached Disks (SNAD) [Miller et al., 2002] is not a file system per se, but a cryptographic permission system that can be layered on top of different network servers. However, SNAD depends on centralized trusted servers to perform operations correctly. Furthermore, it depends on timestamps to prevent replay attacks; thus the system is not viable in a peer-to-peer setting where clock synchronization is often impossible. SNAD supports group sharing; however, revoking users cannot be removed from a group once they have joined it.

4.2.8. SUNDR

The Secure Untrusted Data Repository (SUNDR) [Li et al., 2004] is a server based cryptographic integrity protection scheme. SUNDR only addresses data integrity but not data confidentiality. It supports Unix-like access permissions and supports user groups; however, these permissions are only enforced for writes. That means, SUNDR only protects the file system integrity. Anyone with access to the file system can read all data stored in the system. SUNDR supports *fork consistency*, the highest level of consistency possible in a system without trusted servers. Fork consistency is discussed in detail in Section 9.1.6 on page 152.

4.2.9. AFS

The Andrew File System (AFS) [Howard et al., 1988] is a server-based decentralized file system developed at the Carnegie-Mellon University (C-MU) and IBM.

In contrast to NFS, AFS offers a much more sophisticated caching mechanism. When a file is opened, the whole file is copied from the file servers to the local cache of the workstation [Howard et al., 1988, p. 190]. All subsequent read and write operations are executed on the local copy. The changes are propagated back to the file server after receiving the close system call. This architecture saves network load – server interaction is only needed when files are opened and closed.

The need for server interaction is further reduced by a callback mechanism [Silberschatz et al., 2005, p. 657]. If a cached copy of a file is already present at the local workstation upon receipt of an open call, all following operations are executed on the cached copy. To prevent a user from working with outdated data, the server keeps a list of all cached copies of a file and sends a callback to the respective AFS-client when a cached file is updated. The AFS-client then invalidates the cached copy.

AFS authenticates the individual users using Kerberos. AFS supports a fine granular ACL system which is much more powerful than the standard Unix ACL system and supports features like only allowing users to create files in folders or append-only files.

4.2.10. Coda

Coda by Satyanarayanan et al. [1990] "began as an epilogue to the Andrew File System" [Satyanarayanan, 2002, p. 86]. Coda is significantly more resistant to failure than AFS. Coda wanted to achieve "*constant data availability*, allowing a user to continue working regardless of failures elsewhere in the system" [Satyanarayanan et al., 1990, p. 447]. A second goal of Coda was to make it usable on portable computers which are often disconnected from the network. Newer versions of Coda can be used even when disconnected from the storage servers; changes are committed to the storage server when the client reconnects to the network.

However, this approach opens up a whole new field of problems. When many devices operate without being connected to the network, the probability of conflicting writes is much higher. Coda cannot resolve all conflicts – manual user interaction is required in these cases. Furthermore, the whole file caching is problematic when operating on large files. The downloading of these files can take considerable time and resources. This is also evidenced by the fact that Coda sometimes pops up dialog boxes, asking the user if a file shall really be downloaded, if it suspects the download could take a long time [Satyanarayanan, 2002, p. 106f].

4.2.11. xFS

xFS [Wang and Anderson, 1993] is a distributed file system that was the first to define many of the requirements of distributed wide area file systems; most of them also apply to the later peer-to-peer file systems.

xFS is server based – it distinguishes three kinds of servers: each file system has one home server, a number of consistency servers, and the client servers where the file system is accessible. The client server has to run on every machine that is part of the file system. All servers have to be trusted – that implies that the machines are centrally administrated and the users do not have full access to their machines.

Data is not replicated through the network or pushed back to the home servers; instead, if a user makes changes to a file, this change is only present on the user's client server. It is only transferred to another client when a user on that client tries to access the file.

xFS uses several mechanisms to reduce the load on the servers. For example, when reading or writing to files the clients have to gain read or write ownership. The servers have to keep a list of the clients that currently have read or write ownership of a file. They try to reduce the load of this listkeeping by aggregating the ownerships wherever possible; for example, they give the ownership of a whole subtree to a consistency server. When many clients have read ownership of a file, the servers discard their list and try to find out which clients currently have read-ownership by broadcasting through the network when a write operation occurs.

xFS does not support redundancy; the authors do not state if the system supports any kind of permission system.

4.3. Cluster File Systems

Cluster file systems are a special variant of decentralized file systems optimized for a local environment where many machines access the data at the same time.

The borders between cluster file systems and the previously mentioned server based systems are fuzzy; cluster systems typically still depend on certain server systems. However, the functionality of the servers is limited in comparison to the server based systems. Cluster based systems are typically very scalable.

This section describes systems that, in my opinion, have a less centralized structure than server based systems but not a real peer-to-peer design. They often use static methods such as hashing functions to determine which data ends up on which network node. The individual network nodes are trusted to return correct data. Most of these systems are only suitable for a limited network size and only work either in local or in wide area networks.

4.3.1. GFS

The Google File System (GFS) [Ghemawat et al., 2003] is Google's distributed file system. It is especially suited for the storage of large files with either large streaming reads or small random reads. It is assumed that files that have been written to the system once are seldom modified.

Despite being a distributed file system, the architecture of GFS is very centralized. Each GFS cluster has a single *master server* and multiple *chunkservers*. All files stored in GFS are divided in fixed-size chunks of typically 64 megabytes. The chunks are identified by a globally unique 64 bit ID. For reliability the chunks are stored on multiple chunkservers, usually on 3.

The master server contains all file system metadata, including the file-to-chunk mapping and the chunk-to-chunkserver mapping. The state of the master server is replicated onto multiple machines. In case the master fails, one of the replicas becomes active.

The file system does not provide a POSIX API, instead the GFS client library has to be used to access data stored in the file system. GFS does support a simple permission system, the exact nature of which is not disclosed.

The GFS file system design is simple and tailored to the exact needs of Google. It is not a general purpose file system and not suited for many applications. For example, the central nature of the metadata server can become an issue in systems when many different files have to be accessed. For reads, these problems can be mitigated by using shadow metadata servers which show a slightly out-of-date version of the metadata. These servers can answer requests for static, not recently created files, but all writes have to be handled by the master metaserver. GFS performance sometimes is an issue as evidenced by the special provisions of Bigtable for GFS latency spikes [Chang et al., 2008, p. 15]. Concurrent writes to the same area of a file can result in several differing replicas on the different chunkservers.

Apparently, Google also considered extending the design of GFS to offer peer-to-peer features [see Ghemawat et al., 2003, p. 31]. However, if such a system was ever implemented, it has not been published.

While GFS is only used internally by Google and not available for outside use, the Kosmos Distributed File System (KFS) [Schürmann, 2008] (also known as CloudStore) is available as open source software and uses the same basic architecture as GFS.

4.3.2. HDFS

The Hadoop Distributed File System (HDFS) [Borthakur, 2008] has similar goals to GFS. It is only suited for big files, the architecture documentation says: "A typical file in HDFS is gigabytes to terabytes in size" [Borthakur, 2008, p. 1].

The basic architecture is similar to GFS. A *NameNode* handles all metadata operations; the actual data is saved on a set of *DataNodes*. HDFS is designed to mirror all data on DataNodes. It continuously checks the availability of all nodes and regenerates lost replicas. However, the NameNode is a single point of failure. Recovery from the loss of a NameNode server requires manual intervention. Automatic restarts or failovers are not supported. The metadata on a NameNode has to be protected at all costs, losing the metadata can render the data in HDFS inaccessible. For this reason, NameNodes can keep several synchronously updated versions of the database on different discs.

Like GFS, HDFS does not offer a POSIX interface. HDFS does, however, offer a Unix-like permissions system [HDFS] with users and groups. Access permissions are enforced on the client side. HDFS does not support ACLs.

Unlike GFS, HDFS enforces exclusive write access. During the time one client is writing to a file, the NameNode will reject a second open request with write permissions.

4.3.3. Lustre

Lustre [Braam et al., 2004] is a high performance cluster file system which is used on a number of the world's fastest supercomputers. It is, for example, used by the TianHe-1A Supercomputer of the Chinese National University of Defense Technology [Yang et al., 2011, p. 345], the world's fastest supercomputer (as of November 2010) [TOP500].

The internal architecture is again similar to the architecture of HDFS and GFS. However, in recent versions of Lustre, the metadata is not stored on a single server, but on a small number of cooperating servers. The file data is stored on Lustre Object Targets, which can either be accessible via the networking layer or via a mix of the networking layer for control messages and SAN for actual data transfer (SANOSC or SANOST [Braam et al., 2004, sec. 2.6]).

Lustre is highly configurable and supports a large number of network and storage technologies. If configured well, Lustre is extremely scalable. However, the exact requirements of the storage system have to be known a long time before the network is actually established and the design has to be carefully planned.

4.3.4. CoStore

CoStore [Chen et al., 2002a,b] is a distributed file system developed by the Michigan State University and Sun Microsystems. CoStore is designed as a storage cluster. The file system responsibilities are evenly distributed across all members. There is no central managing node. CoStore was primarily designed to combine the unused disk space of user workstations into a single file system. The file system is not able to operate on the “normal” Internet, it only works in local area networks, e.g. in a company or university, because it depends heavily on multicast operations.

CoStore distinguishes between file system servers and clients. Each computer that offers file space is a server; computers which access files are clients.

In CoStore, each server has a unique identifier and IP address. Additionally, all servers listen to the same multicast IP-address. If a client does not know that a specific node is responsible for a specific piece of data, it sends the request to the multicast-IP-address and waits for an answer from one of the servers. All servers answer from their own IP-addresses not from the multicast IP-address.

The protocol that is used between the clients and the CoStore cluster is a modified version of NFS version 3. It was modified to use UDP instead of RPC calls to be able to support the multicast operation mode of the cluster.

Conflict resolution in CoStore is very simple: “the latest write prevails” [Chen et al., 2002b, sec. 2.2].

The internal architecture of CoStore is very similar to the Linux Second Extended File System (ext2) [Card et al., 1986] (see Section 4.1.2 on page 54). Each file in the file system is identified by an inode with a 32-bit address. The first 8 bit of each inode address is the unique server identifier, the rest identifies the file. The same holds true for data blocks; each data block has a 32-bit address with the first 8 bit being the unique server identifier[2].

CoStore trusts the user workstations. Permission bits are stored in the file’s inodes, like in traditional Unix file systems. Data reliability is ensured by using different servers in a RAID-configuration. For example, when using RAID 4 there is one master node and several slave nodes that exchange the blocks of the part of the file system for which the master node is responsible. All requests are usually answered by the master node but, when the master node fails, one of the slave nodes can upgrade itself and claim the responsibility of the master node. The papers do not cover the problem of guaranteeing consistency with multiple simultaneous writers.

4.3.5. GlusterFs

GlusterFs [Juve et al., 2010] is a cluster file system for use in trusted environments. The architecture of GlusterFs works best if it is used in a file system where all nodes can reach each other with low latency and a high bandwidth. Unfortunately, the exact architecture of GlusterFs is very poorly documented.

[2]This means that the maximum file system size in this design is 16 TB. However, the system can easily be extended to use 64-bit addresses.

GlusterFs is highly configurable and supports several modes of operation. Of special interest is the way how writes and reads are handled in the system.

GlusterFs supports an algorithm called *elastic hashing*, where the full pathname of an object is hashed. The resulting hash determines on which machines in the network the file is stored. Hence, when accessing a file, every node in the network can determine the storage locations just by inspecting the file name. Writes and reads should be equally distributed around the network. To prevent server-to-server moves of big files on renames, the system internally supports a redirection layer. GlusterFs also supports a second operation mode, where all data which is written is stored on the local mode – this mode is called Non-Uniform File Accesss (NUFA).

GlusterFs supports replication of stored data. Also, it supports the standard Unix permission system – however, it trusts the local nodes to enforce it; no server based checking is done[3].

4.3.6. BitVault

BitVault [Zhang et al., 2007] is a fully decentralized peer-to-peer storage system especially suited for the long term storage of seldom changing information. In contrast to most previously discussed systems, BitVault offers no file system access semantics - it only works as a object storage and retrieval layer and supports no notion of file system structures like directories, access permissions, etc. BitVault is mentioned here because of its massive parallel repair algorithm. BitVault is presented under the heading of cluster systems because, unlike the following decentralized file systems, it is primarily designed to be used in data centers. Its algorithms require a high-bandwidth, low-latency connection between the individual nodes of the system.

The basic architecture of BitVault is very similar to the architecture employed by other decentralized peer-to-peer storage systems like, for example, IgorFs. Objects are stored on random nodes and a DHT is used to store the current object locations.

In contrast to most other decentralized systems, each BitVault node is aware of all other BitVault nodes in the system. This knowledge is used to keep data alive in the system. When a node fails from the system (or when a new node joins the system), all currently active nodes are made aware of this change. The DHT index of a failed node and the data stored on a failed node is rapidly reconstructed by all nodes that are still alive in the system. Each system sends the list of concerned stored objects to the new system now responsible for the part of the DHT id space. Similarly, an index storing system requests objects to be replicated when single copies fail.

4.4. Decentralized Distributed File Systems

As already discussed in the previous section, the cluster file systems still place a big amount of trust into the other nodes which are active in the network.

[3]As noted, the available documentation is very brief; if GlusterFs should support a more secure scheme it was not detailed in any of the available papers.

In comparison, this section presents only decentralized or peer-to-peer file systems. Decentralized file systems are a big evolutionary step from the presented server based file systems. They are not limited to operate in a local or wide area network. They place no or a very limited amount of trust into the data which they read from other systems and have to deal with malicious nodes and users.

Designing such systems is a much more complex task, especially when considering problems such as user management or write synchronization. For this reason, many of the systems are not as feature complete as their counterparts discussed in the previous sections. This is especially true for many of the early systems.

This section first presents some of these earlier systems, which had a very limited functionality. After that more recent developments are shown.

4.4.1. OceanStore

OceanStore [Kubiatowicz et al., 2000; Bindel et al., 2002] is a global persistent data store. It has been designed to scale to billions of users. OceanStore assumes that any server in the infrastructure may either crash, leak information, or become compromised. The prototype implementation of OceanStore is called *Pond* [Rhea et al., 2003]. OceanStore uses the Tapestry DHT [Zhao et al., 2004] for object location.

OceanStore supports access control via file-based ACLs. Writes are only committed if a quorum of servers responsible for the respective file accepts the write operation. These servers are called the *inner ring*. Concurrent writes are serialized through the inner ring nodes. If enough of these servers become compromised, they can change any data under their control without anyone noticing. There are no further integrity or rollback protection schemes.

Unfortunately, the description of how access control lists are handled in OceanStore is very vague. Groups are apparently handled; however, there is no detailed description of how group cryptography works in OceanStore. It is also assumed that a certain set of nodes is mostly available and reliable.

4.4.2. CFS

The Cooperative File System (CFS) [Dabek et al., 2001b] is a peer-to-peer file system that stores all its data in the DHash [Dabek et al., 2001a] DHT. CFS uses a file system design that is inspired by the directory service proposed by Fu et al. [2000]. It is similar to the Unix file system design presented in Section 4.1.2.

Each CFS block is stored on the DHash node with the closest ID to the block's content hash. Thus, the data is equally spread across the network.

For the clients CFS is a read only file system; only a publisher may update the data stored in the system. To write a new file system version, the file system root block is exchanged. Root blocks have to be signed and a root block may only be exchanged if the new block is signed with the same key as the old one. Replays are prevented using a timestamp.

CFS uses an interesting garbage collection concept: data is only stored for a certain amount of time. If the data shall remain in the system, the publisher has to regularly inform the DHT nodes that they have to retain the data. If some data becomes outdated and is no longer required, it is deleted automatically after some time.

CFS features replication through DHash. It also offers quota support, which is very unusual for decentralized file systems; I am not aware of any other decentralized file system with quota support. However, the quota implementation is rather simple: the amount of storage for each other node is limited to a percentage of the local memory.

4.4.3. Ivy

The Ivy file system [Muthitacharoen et al., 2002] is a multi-user read/write peer-to-peer file system without centralized or dedicated components. Users do not need to fully trust each other or the underlying peer-to-peer storage system.

Ivy is a log-based system. Each participant has one log and modifies the file system by appending to its own log. Data is read by consulting the logs of all other participants of the file system. All logs are signed by their respective user. The logs are stored in the DHash [Dabek et al., 2001a] distributed hash table.

Reading data is an expensive operation in Ivy because one has to consult all the log files of all other participants. In order to avoid that, participants have to traverse the entire file system log when reading data. Each participant regularly constructs a private snapshot of the file system contents and stores it in DHash.

Access to the Ivy file system is given via a NFS loop-back server; the Ivy file system is mounted to the local file system via NFS.

In Ivy it is possible to exclude malicious user's changes from the file system by not accepting their log entries; however, this can lead to inconsistencies in the file system that have to be resolved manually.

File systems of multiple users are formed by creating an immutable block that points to the logs of all file system participants. This block is called a *view*. To create a view, key material has to be transferred between the participants using secure out of band communication. The participants that may access a file system are determined upon creation of the file system. It is not possible to add or remove users later; instead a new view has to be created.

It is possible that simultaneous updates from different users conflict. The file system is designed to handle such conflicts automatically; after the changes have been committed all Ivy instances show the same file system state and one of the conflicting updates is discarded. However, the system calls on the affected nodes will have returned success – the process whose write was discarded is not made aware of this fact. Ivy provides a tool to search such conflicting writes in the file system history. They can be restored manually.

Ivy does not feature user or group management. Each user may read or write every file in the file system.

4.4.4. Total Recall

The Total Recall File System (TRFS) by Bhagwan et al. [2004] is a system that aims to provide continuous data availability. To this end, the system uses *availability prediction*. The currently available nodes are continuously monitored. The measurements are used to predict future availability about single hosts and groups of hosts.

The system uses either erasure coding or replication, depending on the characteristics of the stored files.

Blocks that fail from the network are repaired automatically using either an *eager repair* or a *lazy repair* algorithm. Using eager repair, data is regenerated at once when a single block failure is detected. This scheme is used by most distributed systems. Using lazy repair, the missing blocks are not regenerated at once; instead, the repair is deferred to a later time. For this scheme to work, initially more redundancy is stored in the network as in the eager repair scheme. By delaying repairs as long as possible, unnecessary repairs of data on returning nodes are prevented.

A master node is responsible for committing newly written data to all storage hosts; hence, concurrent updates are no problem.

When a file is first created, the desired availability of the file has to be specified. The system chooses the redundancy algorithm as well as the repair method according to the specifications.

Neither user management nor data encryption are supported by the file system.

4.4.5. Keso

Keso [Amnefelt and Svenningsson, 2004] is a distributed peer-to-peer file system, developed at the Royal Institute of Technology in Stockholm, Sweden. Keso uses a separate underlying peer-to-peer network, the Distributed K-ary System (DKS), which was discussed in Section 3.6.2.

In Keso, data is split into blocks of equal size and referenced from a block list in the inode. Blocks are encrypted with convergent encryption (see Section 5.4 on page 83). Both blocks and inodes are stored in DKS at the address of their content hash. That means when a file changes, it gets both new block-identifiers and new inode-identifiers.

Directories act as name/inode lookup service and are always stored at the same DKS-identifier, even when the contents of a directory change.

The Keso File System uses public key cryptography to secure directories. It partially relies on other nodes to be online to verify the validity of newly inserted information. The node that is responsible for the ID at which the new version of a directory is saved has to verify that the changes to the directory are valid. If this node is malicious, parts of directories can be tampered with without anyone else noticing unless they retrieve the whole change history for a specific directory.

The inode-identifiers of old file-versions have to be kept in the directory entry to be able to access them.

For access control each user and node has a public/private key pair. Each directory has a symmetric key which is used to protect the data contents of this directory. This symmetric key is encrypted with the public key of all users that are able to access this directory.

Keso does not support file system wide groups. However, a user can share files with a set of users by generating a public/private key pair and giving the private key to all group members. Directories with group access are then encrypted using the public key of the group. Unfortunately, the revocation of users from a group is not easy. Keso depends on the individual network nodes to verify that a user is a current group member before returning a data block to that user. If, for example, a revoked user is responsible for the ID space of a directory, the user will still be able to read newly written data.

The symmetric key is used to encrypt the inodes – they are not encrypted using convergent encryption.

For tamper protection, changes that are committed to a directory entry are signed. However, Keso does not guarantee the freshness of directory entries that are read from a node. A node can simply return an outdated directory version which will be accepted from all clients. This is a problem because the node which is responsible for a specific directory does not change over time – if a malicious node owns the ID space of a directory that is interesting, the node could potentially show old versions to select clients and not be detected for a long time.

4.4.6. Pastis

Pastis [Busca et al., 2005] is a distributed read/write peer-to-peer file system developed at INRIA. It is especially designed to be highly scalable, locality aware and fault tolerant. It ensures consistency during concurrent write operations while maintaining a good performance.

Pastis uses the Pastry peer-to-peer network [Rowstron and Druschel, 2001a] to route its messages, which was discussed in Section 3.6.2 on page 46. Furthermore, it uses Past [Rowstron and Druschel, 2001b], a DHT-abstraction layer for Pastry which ensures data availability by using replication.

Pastis is completely decentralized. The structure of the file system has some similarities to Keso. It uses data blocks which are stored at the address of their hash.

The second type of blocks are inode blocks, which are called *User Certificate Blocks* (UCBs) in Pastry. Each inode block contains the meta information about a file or directory (its owner, etc.) and points to the data blocks where the actual directory or file contents are stored. For each UCB, there is a private/public key pair which is generated by the user creating the object.

Consistency during simultaneous writes is maintained with the help of version numbers. Each UCB has a version number attached to itself. A new version of a UCB is only accepted by a node when the version number is greater than the last known version. When two nodes try to insert a new version of a UCB simultaneously, one of the writes will be aborted by the node responsible for the UCB's ID space.

Pastis supports two different consistency models; the strictest model is the *close to open consistency* [Howard et al., 1988]. This is the same model that is used in many distributed file systems like NFS and AFS.

In the close to open consistency model, changes only have to be distributed to the other network nodes, when a file is *closed*. The changes between a file open and the file close can occur only on local storage and do not have to be committed to the network.

Security-wise, Pastis does not cryptographically enforce read access control. Write access control and data integrity are ensured via certificates. For each file, a write certificate is issued to each user who may write to the file. When a file is modified it has to be signed with one of these certificates. The DHT nodes check the signatures before inserting new revisions of an inode. The signature is also checked when reading the data. Pastis is resistant against rollback attacks, as long as at least one replica of the UCB has not been rolled back.

4.4.7. SiRiUS

SiRiUS [Goh et al., 2003] is a secure file system that has been designed to be layered over insecure network and peer-to-peer file systems. SiRiUS assumes untrusted storage and implements read write cryptographic access control for file level sharing. Access control information is stored together with the file data on the servers. The file is split into a data and a meta data part.

Each user maintains a key for asymmetric encryption and a signature key. Each file has a unique symmetric encryption key and a unique signature key. Possession of the encryption key gives read access to the file, possession of both keys gives read-write access.

Freshness of the file system is guaranteed using a hash-tree. To update the top-level hash of the hash-tree each change has to propagate to the root-directory.

SiRiUS does not provide support for file system wide groups of users. Instead, access to a specific file can be given to a set of users manually. Usually SiRiUS is aimed at environments where files are only shared in small user-groups. Certain operations like the revocation of a single user's access are expensive when a large group of users may access a file.

However, SiRiUS proposes a scheme to handle large numbers of users more efficiently using the subset difference encryption scheme introduced in Section 2.10.

SiRiUS does not try to provide a unified file system to a set of users; instead, each user owns a separate file system. These systems can be combined with union-mounts, but this approach requires searching the file system of every user that has access to a specific file for the most recent version.

SiRiUS is prone to certain kinds of rollback attacks. The freshness guarantees in SiRiUS only apply to the meta data. Hence, data files can be reverted to a previous revision because the previous revision also has a valid signature. It is, however, impossible to revert permission changes on files. Only the file content can be reverted to a previous revision. A method to prevent such rollback attacks is presented in the paper; however, it has scalability problems. The complexity is $O(n)$ with n being the number of users that can write to the file.

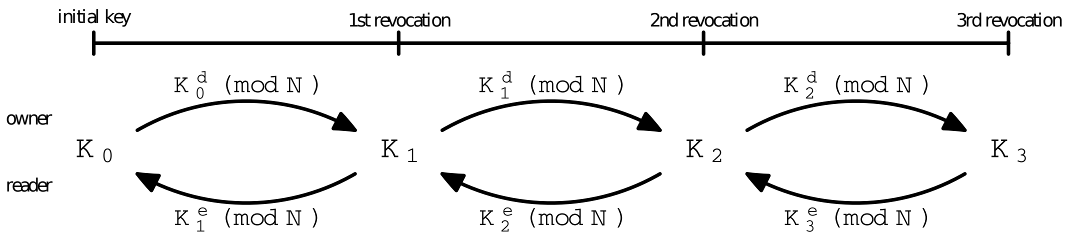


Figure 4.4.: Plutus key rotation [from Kallahalla et al., 2003, p. 34]

4.4.8. Plutus

Plutus [Kallahalla et al., 2003] is a cryptographic storage system that allows secure file sharing using untrusted file servers.

The prototype implementation of Plutus is based on OpenAFS. It was included in this section because the design of the underlying cryptographic scheme is geared towards a decentralized system and independent of the underlying storage architecture.

Plutus uses the lazy revocation scheme introduced by Cepheus. To this end, Plutus introduces a *key rotation* scheme which enables users to share files for reading or writing in a cryptographically secure way. Files with the same access rights are grouped in *file groups*.

Keys for a file group are rotated forward by the file group owner using the RSA decrypt operation with a private key. Figure 4.4 illustrates this scheme: Keys can be rotated backward using the RSA encrypt operation and the public key. Every user who possesses the current file group key can determine all the previous file group keys. Hence, the scheme does not require re-encryption of data upon user revocation.

Instead, the owner generates a new group-key with the forward rotation operation using his secret private key. All files can still be read by the current users by rotating the key back with the public key. When a file is changed the new encryption keys are used and users possessing the old keys will no longer be able to read the files.

Plutus does not support Unix-like user groups. Instead, users have to be added and removed individually from the file groups. Plutus also does not support versioning. Once a change is committed, the old version of the file can no longer be read. Hence, it is important to prevent writes from unauthorized nodes. To this end, Plutus uses server verified writes using hashed *write tokens*. Only users that are aware of the write tokens that are generated by the owner can commit changes to the storage. The untrusted servers have to perform this operation correctly.

After the publication of Plutus, cryptographic weaknesses in the approach where found. Hence, Fu et al. [2006] later changed the scheme and called it *key regression*. However, the general usage of the system is the same. Naor et al. [2005] also propose to replace the scheme with a hash-chain scheme made more efficient by using *fractal hash chain traversal* [Jakobsson, 2002].

Whenever this thesis refers to the security scheme of Plutus, it assumes that one of these secure variants is used.

4.4.9. Celeste and Pacisso

Celeste [Badishi et al., 2009; Caronni et al., 2005] is a highly-available, ad hoc, distributed, peer-to-peer data store. Celeste can use any standard DHT as backend as long as it supports replication. The system design of Celeste is very similar to the other presented file systems. Each object in Celeste is stored in the DHT. Celeste supports versioning; when a new version of an object is written, it contains pointers to the previous version of an object.

Unlike many of the previously discussed file systems, Celeste can securely delete the data of removed entries from the DHT. But Celeste cannot be accessed like a regular file system – like e.g. GFS it has to be accessed using a specialized API.

Celeste itself does not support file level security, but [Koç et al., 2007] [see also Koç, 2006] describes a security model named PACISSO which is layered on top of Celeste.

PACISSO needs active online nodes participating in its protocol. An object owner designates a number of gatekeepers, which are responsible for granting read or write access to the data. If enough of these gatekeepers are compromised, they can deny legitimate read access or allow unauthorized parties to modify the objects under their control. They can, however, not read the stored data.

4.4.10. Wuala

[Wuala] is a commercial, distributed storage system. Unfortunately, due to the commercial nature of Wuala, the available description of the internal architecture is superficial in nature and only covers very few aspects of the system in depth. Wuala is not fully decentralized, it depends on some central servers for authentication and partially for storage – the exact way these servers operate is unknown.

Wuala uses a peer-to-peer network similar to Chord [see Grolimund, 2007]. The whole data in the file system is encrypted with a cryptographic folder structure named *cryptree* [Grolimund et al., 2006a]. The cryptree folder structure offers a relatively fine-granular method of granting access rights. One can, for example, allow other persons to access single files or whole subtrees of a directory tree. However, if a person can read a directory she is able to read all subdirectories and files in this directory. The same problem also applies to write operations, if a person can write to a directory, the person can also write to all subdirectories.

While, in the beginning, the security of Wuala was disputed due to the chosen way the cryptographic operations were performed [Percival, 2007a,b], later changes to the encryption architecture solved most of these problems [Percival, 2008]. According to Percival [2008], a remaining problem is the integrity protection of the files stored in Wuala. Apparently, it is fully dependent on central servers that have to operate correctly.

Wuala is a *social online storage* system. Participants get a base storage of one gigabyte. If they need more storage space, they can either buy it – in this case the data is probably saved in the company server infrastructure – or trade their own hard drive storage space. The traded space is used by Wuala to save encrypted data of other users. The amount of remote storage gained per megabyte is the local traded space multiplied with the percental online time per day. For every megabyte of local storage, a user gets one megabyte of remote storage given the user is online 100% of the time and 500 kilobytes given the user is online 50% of the time. Wuala uses a reputation system called *Havelaar* [Grolimund et al., 2006b] to prioritize well-behaving clients.

Files in Wuala are coded with erasure coding for better availability. For small changes in big files this can be problematic because the whole file has to be re-uploaded. In a comment on [Percival, 2007a], the CTO of Wuala stated, that for changes up to four kilobytes in size the data is not updated, but a patchset is stored in the file metadata. However, this does not remedy this problem for many operations. Wuala can be integrated in the local file system.

For a more in depth review of Wuala see [Wohlfahrt, 2009].

4.4.11. Tahoe

The Tahoe Least-Authority Filesystem [Wilcox-O'Hearn and Warner, 2008] is a secure distributed storage system which was used in a commercial backup storage solution called *allmydata.com*. Unfortunately, the service seems to have ceased to operate. However, Tahoe is still being actively developed and can be freely downloaded [Tahoe].

Tahoe is not a file system in the sense that it can be mounted and accessed like usual file systems. It has non-standard access semantics – but files stored in Tahoe are grouped into a directory structure[4].

Tahoe puts a special focus on the security of the stored files and uses several different cryptographic algorithms to allow fine-granular access control. Tahoe offers two different types of files – mutable and immutable files. In contrast to mutable files, immutable files cannot be changed after they have been written to the system.

A user can have several per-file capabilities. The verify-cap allows only the verification of the stored data. The read-only-cap allows a user to read and verify a file. The read-write cap allows a user to write and read and verify a file. Capabilities can be downgraded. A user with a read-write-cap can generate a read-cap or a verify-cap and a user in possession of a read-cap can generate a verify-cap. The generated capability can be given to other users.

For immutable files a user can either have a read-only-cap or a verify-cap. Immutable files only use symmetric cryptography; the contents are encrypted using AES. The file integrity is protected using Merkle hash trees [see Wilcox-O'Hearn and Warner, 2008, p. 23].

For mutable files, a user can have a read-write-cap, a read-cap or a verify-cap. Mutable files are encrypted using RSA, the integrity protection scheme is the same as for immutable files.

Directories are lists of their child directories and files. The read-cap is stored together with each entry. Hence, every user who can read a directory tree can read everything stored in this directory tree. The read-write-cap of the individual files is also stored in the directory, but it is encrypted with the directory read-write-cap key. Hence, users having

[4]Tahoe does support Fuse, but the support is incomplete [Tahoe Summary] and Fuse is not a good match to the usual access semantics of the system [Warner et al.].

a read-cap on a directory can read all subdirectories and files; users with a read-write-cap can also write to all subdirectories and files.

Group-access, ACLs or other more fine-granular access rights, like preventing a user from reading a subdirectory, are not supported by Tahoe.

Tahoe guarantees the freshness of the read data by comparing the version numbers of the different erasure code fragments it reads from different servers. When the version numbers differ, only the fragments with the highest available version number are used.

Other than that, Tahoe has no integrity protection of individual files; an entity in control over a big enough number of servers can rollback parts of the file system without anyone noticing.

Tahoe ensures the availability of the data in the face of node failures by using Reed-Solomon erasure codes (see Section 6.3 on page 95).

4.4.12. DRFS

The Distributed Reliable File System (DRFS) [Peric, 2008; Peric et al., 2009] is a distributed, decentralized read-write P2P file system. It is based on the TomP2P DHT [TomP2P], which uses a Kademlia based routing protocol.

DRFS supports multiple concurrent writers. Each file is split into fixed-size fragments. Each of these fragments has a variable value v associated with it, which is changed on every write. The old value of v has to be present in the write request. Hence, if a file is written by multiple clients simultaneously, the first write to arrive at the data storing node is successful and all later writes fail due to the changed v.

DRFS supports the standard Unix permission bits, however, these are not enforced by cryptography – clients can simply choose to ignore it. The prototype implementation does not even include this feature.

DRFS uses replication to secure itself against node failures.

4.4.13. Farsite

Farsite [Adya et al., 2002; Douceur and Howell, 2006] is a distributed file system which provides file availability through randomized replication, secures the file contents with encryption and maintains file system integrity through the use of a byzantine fault tolerant protocol. The encryption scheme of Farsite is ACL based [Douceur et al., 2002a]. The owner of a file or directory can add or remove any other user of the file system to the list of readers or writers. The integrity of directories is provided through a byzantine fault tolerant server group. It works correctly with up to $T = \lfloor (S-1)/3 \rfloor$ misbehaving servers (with S being the size of the server group). Writes are only performed when the server group validated the permissions of the writer. Rollback attacks are impossible with less than $T+1$ misbehaving servers. Farsite does not support Unix-like group sharing.

4.4.14. Clique

Clique [Richard et al., 2003] is a P2P file system suited for data sharing in small user groups. It uses an epidemic replication algorithm where all files are stored on all nodes of the network. Clique takes special consideration on how to resolve write conflicts. It uses file signatures to determine differences between files stored on different nodes. Where automatic conflict resolution is not possible, both conflicting versions are kept (one of the versions is moved to a special directory). No permission system is supported.

4.5. Comparison

Table 4.2 shows a feature comparison of nearly all file systems presented in this section. The file systems that were not added to the table have feature sets that are very similar to the feature sets of other file systems like, for example, the Kosmos file system which is nearly equivalent to GFS.

The rows of the table show the following information:

Permissions: Does the system have any kind of permission system?

Unix Perms.: Does the permission system have similar functionality to a standard Unix permission system?

ACL: Does the system support ACLs?

Groups: Does the system support file system-wide user groups for its permission system?

Encryption: is the permission system enforced with cryptography?
If this feature is not available, clients may simply choose to disregard the set permission or ACL flags.

Concurrent Writes: Does the system support concurrent writes of several users?

Versioning: Does the file system support versioning?

Redundancy: Does the file system use some kind of redundancy scheme?

Redundancy Algorithm: Specifies the used redundancy algorithm. **R**eplication, **E**rasure coding, a combination of both, or RAI**D**.

Untrusted Servers: Is the information of the data storing nodes verified or are servers expected to return correct data?

Untrusted Clients: Are clients untrusted, and the access permissions enforced by the server, or may they simply read or write data at will?

Peer-to-peer: Does the system work without centralized servers?

Transparency: Does the system look like a local file system to the user?

Integrity protection: Is the information stored on the network protected from unauthorized modifications in some kind?
Note that this is not the same as untrusted servers because systems can have integrity protection that depends on the correct answers of a trusted server.

Rollback Resistant: Does the system offer some kind of rollback protection?

Fork Consistency: Does the system offer fork consistency?

Directories: Does the file system support directories?

Efficient random access: Is read and write random data access inexpensive, for both small and large writes? Please note that this is not meant to be an exact metric of the efficiency of the system – this is out of scope for this work. It just identifies systems where, due to the way of data storage, it is expensive to perform some of these operations. There are good and valid reasons for these choices in those systems; however, it makes them less suitable to be used as a all-purpose file system.

The possible values for each entry are ✔ if the feature is supported and ✘ if it is unsupported. ○ shows that a feature is partially supported or, for layered systems, if the support depends on the underlying storage system. ? is used when it is unknown if the feature is supported.

For IgorFs, which is described in the next chapter, the table shows ✎ for features that IgorFs is missing and whose design is described in the remainder of this thesis.

Name	Type	Permissions	Unix Perms.	ACL	Groups	Encryption	Concurrent Writes	Versioning	Redundancy	Redundancy Algorithm	Transparency	Untrusted Servers	Untrusted Clients	P2P	Integrity Protections	Rollback Resistant	Fork Consistency	Directories	Efficient Random Access
AFS	Server Based	✓	✓	✓	✓	✗	✓	✗	✗		✓	✗	✓	✗	✗	✗	✗	✓	✓
CFS	Peer-to-Peer	?	?	✗	✗	✗	✗	✗	✓	R	✓	✓	✓	✓	✓	✓	✗	✓	✓
Cepheus	Server Based	✓	✓	✗	✓	✓	✓	✗	✗		✓	✗	✓	✗	○	✗	✗	✓	✓
CoStore	Cluster	✓	✓	✗	✓	✗	✓	✗	✓	D	✓	✗	✗	✗	✗	✗	✗	✓	✓
Coda	Server Based	✓	✓	✓	✓	✗	✓	✗	✗		✓	✗	✓	✗	✗	✗	✗	✓	✓
DRFS	Peer-to-Peer	○	○	✗	✗	✗	✓	✗	✓	R	✓	✗	✗	✓	✗	✗	✗	✓	✓
FAT	Local	✗	✗	✗	✗	✗	✓	✗	✗		✓	✗	✗	✗	✗	✗	✗	✓	✓
FFS	Local	✓	✓	○	✓	✗	✓	✗	✗		✓	✗	✗	✗	✗	✗	✗	✓	✓
Farsite	Peer-to-Peer	✓	✗	✓	✗	✓	✓	✗	✓	R	✓	○	○	✗	✓	○	✗	✓	✓
GFS	Cluster	✓	?	?	?	✗	○	✗	✓	R	✗	✗	✗	✗	✗	✗	✗	✓	✗
GlusterFS	Cluster	✓	✓	✗	✓	✗	✓	✗	✓	R	✓	✗	✗	○	✗	✗	✗	✓	✓
HDFS	Cluster	✓	✓	✗	✓	✗	○	✗	✓	R	✗	✗	✗	✗	✗	✗	✗	✓	✗
IgorFs	Peer-to-Peer	✎	✎	✎	✎	✎	○	✓	✎	✎	✓	✓	✓	✓	✎	✎	✎	✓	✓
Ivy	Peer-to-Peer	✗	✗	✗	✗	✓	✓	✓	○		✓	✓	✓	✓	✓	✓	?	✓	✗
Keso	Peer-to-Peer	✓	✗	✓	✗	✓	✓	✓	✓	R	✓	○	✓	✓	✓	○	✗	✓	✓
NCryptFS	Local	✗	✗	○	○	✓	✓	✗	✗		✓	✗	✗	✗	✗	✗	✗	✓	✓
NFS	Server Based	✓	✓	✗	✓	✗	✓	✗	✗		✓	✗	○	✗	✗	✗	✗	✓	✓
NFSv4	Server Based	✓	✓	✓	✓	✗	✓	✗	✗		✓	✗	○	✗	✗	✗	✗	✓	✓
OceanStore	Peer-to-Peer	✓	✓	✓	?	✓	✓	✓	✓	R/E	✓	○	✓	○	✗	✗	✗	✓	✓
Pastis	Peer-to-Peer	○	✗	○	✗	✓	✓	✗	✓	R	✓	✓	✓	✓	✓	○	✗	✓	✓
Plutus	AFS Layer	✓	✗	✓	✗	✓	✓	✗	✗		✓	✗	✓	✗	✗	✗	✗	✓	✓
SNAD	Layered	✓	✗	✓	✓	✓	○	✗	○		✓	✗	✓	✗	✓	✗	✗	✓	○
SUNDR	Server Based	○	○	✗	✓	✓	✓	✓	✗		✓	✓	✓	✗	✓	✓	✓	✓	✓
SiRiUS	Layered	✓	✗	✓	○	✓	○	✗	○		○	○	○	○	✓	○	✗	✓	○
TCFS	NFS Layer	✓	✓	✗	○	✓	✓	✗	✗		✓	✓	○	✗	✓	✗	✗	✓	✓
Tahoe	Peer-to-Peer	✓	✗	✗	✗	✓	?	?	✓	E	○	✗	✓	✓	✓	○	✗	✓	✓
Total Recall	Peer-to-Peer	✗	✗	✗	✗	✗	✓	✗	✓	R/E	✗	✗	✗	✓	✗	✗	✗	✗	○
Wuala	Part. P2P	✓	✗	✓	✗	✓	?	✗	✓	E	✗	✗	✓	✗	?	?	✗	✓	✗
xFS	Peer-to-Peer	?	?	?	✓	✗	✓	✗	✗		✓	✗	✗	✗	✗	✗	✗	✓	✓

Table 4.2.: File system comparison

5. IgorFs

The work in this thesis is based on the experiences with the distributed file system *IgorFs*, which was developed in the working group for self organized systems at the chair of network architectures and services of the TUM. While the remainder of this work is separate from IgorFs and can be used with any kind of distributed storage systems, it is always assumed that a system similar to IgorFs is used. Hence, this chapter gives an introduction to the architecture of IgorFs.

The origin of IgorFs can be found in the area of bioinformatics. Meanwhile, it is also applied in high energy physics.

In bioinformatics, there are many large databases containing information about genes, proteins, molecules, etc. These databases can reach sizes of several terabytes with millions of files. The databases change daily - however, only a very small amount of data will change at once. One example for such a database is the worldwide protein data bank [WWPDB] which contains molecule structure information.

Research applications e.g. analysis software and simulations need access to the current version of the databases. However, they typically only access very small parts of the databases.

Usually, sites set up a local mirror to allow the software and researchers to access the current data. However, this is very wasteful because the mirror has to contain the whole huge research database and has to sync itself regularly to the master copy.

The best solution would be a file system that only contains the data needed for a specific analysis or simulation. To this end, it only needs to download the deltas for that specific data when a new version of the database is available.

The result of the research of the working group for self organized systems at the chair of network architectures and services of the TUM into these problems was IgorFs, a shared distributed file system with a single writer for each dedicated file system tree. The only information needed to access an IgorFs file system is the identifier and the cryptographic key of the file system root block. In IgorFs, a local node only downloads the information from the network which is requested by the applications accessing the file system. All downloaded information is automatically cached. When a new version of a file system tree is generated, only the changed parts of files that are accessed are downloaded from the network.

IgorFs also features an automatic notification system for new file system versions - clients become aware of newly inserted data very quickly after it has been made available to the network. The file system also encrypts all transmitted data. It offers an interface to an external user-management system e.g. to handle subscriptions. Hence, it is possible to give paid access to institutions and to revoke the access later.

The design of IgorFs makes it suitable for a large field of applications. For example, in the area of high-energy physics, the CERN is currently optimizing the distribution of their research software to the associated computing centers. Just like the biology databases, their software packages are huge and include many libraries, of which only a small number are used depending on the exact simulation parameters. The software repository changes up to a few times per week and it is important that new versions are available at the computing centers after a short amount of time.

At the moment, each computing center has its own software repository that is updated separately. Because only a single server per computing center is hosting all the software, there are performance bottlenecks when many simulations start at the same time. IgorFs also suits this scenario and could significantly decrease the administrative overhead and increase the performance at the same time.

A collaboration with the CERN is currently researching this field.

5.1. Igor

IgorFs is based on a structured P2P overlay network called Igor, which provides a *key based routing service* similar to Chord or Pastry (see Section 3.6.2).

Igor routes messages based on a destination key and a service identifier [Di et al., 2008]. Thus, several different services can co-exist in a single Igor network.

Applications that want to use the overlay network first connect to an Igor node. Usually the Igor node and the application reside on the same physical machine. The application submits its service identifier to the Igor node. After that, the Igor node will receive messages that are sent to this service identifier. Nodes that do not run a service will not receive messages for that specific service.

Igor exports a convenient API which is similar to the well-known Berkeley socket API [Tanenbaum, 2002, sec. 6.1.3]. Messages can be sent and received after an Igor socket has been bound. At the end of the session, the socket is unbound and closed.

Igor uses Proximity Route Selection (PRS) and Proximity Neighbor Selection (PNS) [Gummadi et al., 2003; Kising, 2007; Elser et al., 2010] to exploit the proximity of nodes in the underlying Internet topology. Thereby, services can easily benefit from local nodes running the same service.

Alongside IgorFs, there are a multitude of projects which use the Igor overlay network. These projects include a distributed video recorder [Kutzner et al., 2005], a distributed SIP-client [Bucher, 2009] and a serverless Jabber network [Hudelmaier, 2008].

5.2. Basic Design

IgorFs is a fully decentralized distributed file system. The different IgorFs nodes use the Igor overlay network to communicate with each other. All nodes are egalitarian: there are no nodes with special properties or special trust in each other. Most importantly, there are no centralized nodes because they would introduce a single point of failure and a performance bottleneck.

In IgorFs, a file system is identified by a unique file system root block. Every node can create a new file system by generating a new root block. To access an existing file system, a node needs to be able to access its root block, i.e. it needs the block identifier and cryptographic key. This will be explained in detail in Section 5.4 on page 83.

Every node can create its own IgorFs file system. It can also open other IgorFs file systems if it has the corresponding key material.

IgorFs is implemented as a process network [Kahn, 1974]. Hence, IgorFs is implemented as a number of independent modules that communicate with each other solely using messages sent via FIFO channels. Apart from that the individual modules are completely self-contained and not interdependent on each other. The individual modules are deterministic. Hence, it is possible to test the functionality of individual parts of IgorFs by passing in test messages and examining the output messages. It is also possible to swap out the individual modules of IgorFs without changing any other part of the system. An example for this is the Application Interface. Two different modules (FuseInterface or NfsInterface) can be used to enable the user to access the file system either via Fuse or NFS. This is explained in more detail in the next section. Furthermore, in the future this design could be used to leverage the advantages of current multi-core architectures – each IgorFs module could run as a separate operating system process or in a separate thread.

The most important modules of IgorFs are:

- The *Application Interface* module (i.e. the FuseInterface or NfsInterface) receives operating system calls, converts them into IgorFs calls and sends them to the File and Folder Module.
- The *File and Folder Module* handles all file and directory operations. All file system requests are handled here. The data needed to satisfy the requests is read or written with the BlockTransfer Module.
- The *BlockTransfer Module* sends and retrieves data chunks via the BlockCache.
- The *BlockCache Module* handles requests for data chunks. If a chunk is found in the cache it is returned, otherwise it is fetched via the BlockFetcher.
- The *BlockFetcher Module* transfers modules from remote nodes to the local node when they are requested.
- The *PointerCache Module* is a DHT-like system which stores the information about which blocks are stored on which nodes.
- The *IgorInterface Module* is responsible for the communication with the Igor daemons. All inbound and outbound messages pass through this module.

The functionality of these different modules will be described in more detail in the following sections.

Figure 5.1 on the following page shows the modules and which modules communicate with each other. In the figure, cornered blocks are IgorFs modules; round blocks are external components.

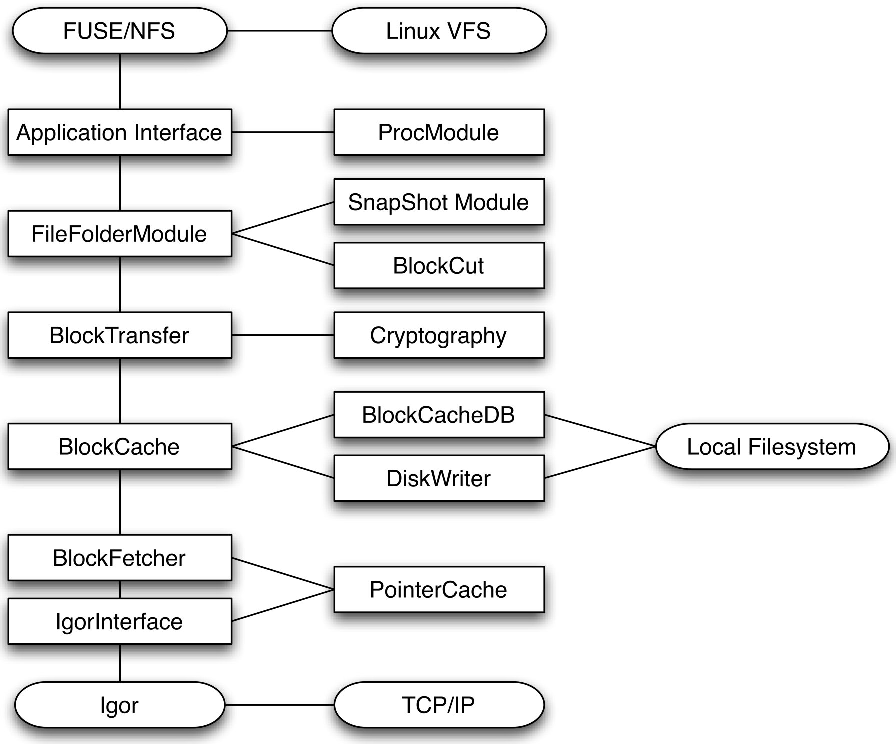


Figure 5.1.: Module communication in IgorFs [adapted from Kutzner, 2008]

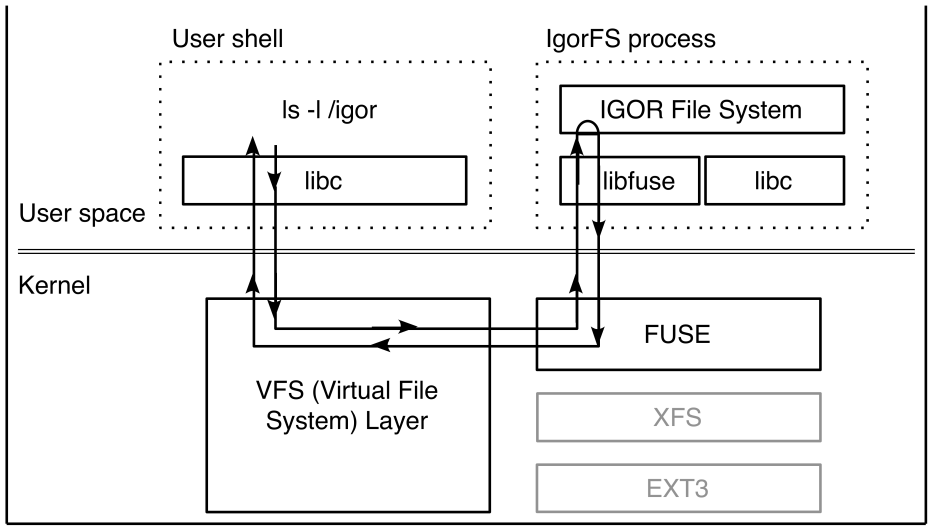


Figure 5.2.: FUSE sample request [adapted from Voras and Žagar, 2006]

5.3. Application Interface

There are two different ways to attach IgorFs to the operating system. The preferred option is to mount IgorFs into the Unix directory tree, just like any other kind of file system. In Unix, the kernel is responsible for handling the file system calls made by applications. Calls are handled by the part of the kernel responsible for the respective file system. However, implementing file systems in the operating system kernel is a complex task.

With [Fuse] (Filesystem in Userspace), it is possible to implement file systems completely in the userspace. Fuse works by using a kernel module that re-routes requests from the kernel's VFS layer to a userspace program. Figure 5.2 shows this approach in more detail. With this approach, IgorFs looks just like any other local file system to the user.

The second way to attach IgorFs to the operating system is via NFSv3. IgorFs can re-export the current IgorFs file system via NFS. A client can then mount the IgorFs file system via its local NFS client software. This approach is more cross platform compatible, but has a lower performance and misses some advanced features such as extended attributes.

5.4. File Storage

All files that are stored in IgorFs are first cut into variably sized chunks. The points at which a file is split into different chunks is determined using a rolling checksum algorithm [Kutzner and Fuhrmann, 2006, sec. 3.2] [Kutzner, 2008, chap. 11.4]. The rolling checksum is calculated “for a sliding window which is moved over the data” [Kutzner, 2008, p. 114]. The algorithm used for the calculations is XOR32 [see Kutzner, 2008]:

$$\mathrm{RC}_x = \bigoplus_{k=0}^{b-1} 2^k \cdot d_{x-k} \pmod{2^b}$$

With x being the checksum position to compute and b being the bit width. Given the checksum at position x, the checksum at $x+1$ can directly be computed with

$$\mathrm{RC}_{x+1} = (2 \cdot \mathrm{RC}) \oplus d_{x+1} \pmod{2^b}$$

Cutting marks are inserted into the data stream when the checksum reaches a certain threshold. The approximate size of the data chunks can be changed by adjusting the threshold parameter.

As a result of this algorithm, two input streams which have parts with the same data will generate the exactly same data chunks for these parts. In contrast to a fixed block size scheme, this also means that if part of a file is modified, only a few data chunks will change, most of the chunks will stay unchanged.

The block size has to be set to a suitable size, depending on the files stored in the network, the network speed, etc. It is good to have relatively big data blocks because this results in a lower number of chunks per file which have to be managed. Big data blocks are also bad for random access of small parts of information because the read latency is increased. The current default setting is a block size of several hundred kilobytes – for other applications sizes up to a few megabytes could be sensible.

Each of these data chunks B in IgorFs is identified by a 256 bit wide identifier I which is assumed to be globally unique. Each chunk is encrypted with its own 256 bit key k. Both the ID I and the key k are obtained by hashing the chunk with a cryptographic hash function H. $k = H(B)$ is obtained by hashing the original data. $I = H(E_{H(B)}(B))$ is obtained by hashing the encrypted data chunk. E denotes the encryption function. Thus, independent IgorFs instances map the same clear text chunk to the same cipher text chunk. Blocks with identical content have the same identifier I. Thus, if files with shared areas are copied to one node, these areas are deduplicated on the fly and only stored once. If two nodes insert two files that have shared areas into IgorFs, these shared areas will be mapped to the same encrypted data chunks. Nodes requesting any of the files can then download the data from the nearer node. This is a very important feature in practice – for example, Bolosky et al. [2000] found that in a sample of 550 file systems at Microsoft up to 47% of storage space could be saved by eliminating duplicate data [Bolosky et al., 2000, sec. 4.1.2].

This cryptographic scheme is also known as *convergent encryption* and was first proposed under this name by Douceur et al. [2002b]. However, there are earlier schemes which are very similar to this, for example, the content hash keys of Freenet (see Section 3.4.2 on page 43).

In order to be able to read a chunk, an IgorFs instance needs to have the chunk's (ID, Key)-tuple. All (ID, Key)-tuples of a object stored in the file system are called the *reference* to this object.

5.5. Directories

Directories are handled similarly to files. A directory consists of a list of file names, file properties and the (ID, Key)-tuples for the file chunks. This list is serialized into a data block which is stored in IgorFs just like an ordinary file.

Thus, when a node can read a directory it can recursively determine all the (ID, Key)-tuples for all files and subdirectories stored in the directory. When a node can read the root-directory, it can recursively read the contents of the whole file system.

Multiple, separated file systems can share the network and the attached storage capacity. They just use different root-blocks.

5.6. Block Transfer and Caching

In IgorFs, data chunks are not stored directly in the DHT. Instead, IgorFs uses an indirect approach. When a new chunk is inserted, it is stored on the local node only. The information from which node the block can be retrieved is then inserted into the PointerCache (see Section 5.2 on page 80).

When a node requests a chunk, it first asks its local pointer cache for the location of the data block. The local pointer cache either knows the location or sends a request to the remote pointer cache that is responsible for the ID area of the request.

After the pointer cache returns the data location, the block fetcher module retrieves the actual data and stores it into the local block cache.

As of September 2011, when this thesis was finished, IgorFs does not have a redundancy maintenance system. When a node inserts some new data, for example, by changing present files or writing new files to the system, the blocks only are present on the inserting node. If this node goes offline these blocks become unavailable. Data is only replicated through the network when it is read by other nodes. In this case the PointerCache points to multiple copies on different hosts from which the chunk can be read.

5.7. File system Updates

Whenever a file changes, all directories on the path from the file to the root also change. The reason for this is that whenever a file changes, it also gets a new content hash and hence a new (ID, Key)-tuple. This new tuple has to be stored in the directory, which in turn also gets a new (ID, Key)-tuple. The changes propagate up in the file system tree until they arrive at the root-directory.

Hence, when a node wants to see the current version of a specific file system, it only needs to know the current (ID, Key)-tuple of the root-directory. IgorFs uses a publish/subscribe system to share that key among the peers.

The current implementation of IgorFs does not support multiple concurrent writes from different nodes due to the way changes cascade up to the root.

If multiple writers are required, the system design of IgorFs could easily be changed to a design like used in Plutus, where the directory blocks in the system are modified on writes. Such a design does not have the problem of cascading writes; however, it is no longer possible to easily access all older file system revisions.

5.8. Comparison

This section briefly compares IgorFs and some of the file systems mentioned in Chapter 4.

IgorFs is not really comparable to distributed file systems like NFS, Coda, or xFS which depend on a centralized server infrastructure. As already detailed in Chapter 1, centralized systems with trusted servers have a much more straightforward architecture than peer-to-peer file systems. On the other hand, decentralized file systems are more scalable, self-organizing, and can be made very resistant against individual node failures.

Thus, this section will mostly compare IgorFs to the other decentralized file system designs, with the notable exception of Row-FS. While Row-FS is a server-based distributed file system, it has one interesting similarity to IgorFs: Row-FS and IgorFs both support a special operating mode, that file systems do not often support but that can be useful in certain circumstances: Row-FS supports a no write back operating mode, where changes on the local node are not committed back to central storage. This is e.g. desirable when "provisioning system images for diskless clients or virtual machines" [Chadha and Figueiredo, 2007, sec. 5]. IgorFs supports the exact same operating mode for the same reason.

The peer-to-peer file system most similar to IgorFs is Keso [Amnefelt and Svenningsson, 2004]. While the general architecture of Keso is similar to IgorFs, there are many small differences. Keso uses a fixed block size to store its data files, opposed to the rolling checksum algorithm employed by IgorFs. Thus, when only a small portion of a file changes, in Keso the whole file has to be cut into new blocks and distributed to the network again. In IgorFs, only a few blocks surrounding the changed data will change, the rest of the blocks will stay the same.

There are also some scalability issues in Keso's system design which have been avoided in IgorFs: all old file revisions have to be kept in the directory blocks in Keso; in IgorFs, each directory block is tied to a specific file system revision, only. Thus, in Keso, the size of the directory will grow linearly with the number of file revisions stored in it. The advantage of this design is that all old file revisions are directly accessible whereas in IgorFs a a binary search over the file system versions has to be performed.

The overlay network used by Keso (named DKS) has some high level functionality like a DHT system and a replication built in. In IgorFs both the DHT and the replication functionality are built into the file system layer and not into the networking layer. Other than that, the functionality of DKS is similar to Igor; both build a ring structure that is used for routing.

In conclusion, the basic system design of Keso and IgorFs are quite similar. Note that this section did not summarize the differences between the integrity guarantees and the permission system of Keso and the system presented in this thesis. These components are compared after the introduction of the new system in Section 8.7.

In contrast to IgorFs, Celeste [Badishi et al., 2009] does not offer a traditional file system interface; a specialized API has to be used to access the stored data. The file system design of Celeste is more fragile than the design of IgorFs: For example, it supports versioning by ordering all object versions into a single-linked list. If even one of the list elements fails from the network, all older file revisions will become unavailable. Due to this design, access to old file versions is slow and, in contrast to IgorFs, it is virtually impossible to view the file system state as it was at a certain point in time. Like Keso, Celeste does not feature dynamic block sizes.

SiRiUS [Goh et al., 2003] and Plutus [Kallahalla et al., 2003] are not comparable to plain IgorFs; they are layered file systems that add permissions to a underlying storage system like NFS; Section 8.7 compares their permission systems to the approach of this thesis.

Unlike IgorFs, Pastis [Busca et al., 2005] uses mutable data blocks and does not support file versions. Data stored in the system is overwritten when it is changed. Thus, as detailed in the previous section, Pastis can support multiple concurrent writers.

Ivy [Muthitacharoen et al., 2002] is completely log-based. In contrast to IgorFs, it cannot really be used as a general purpose file system because the design only allows a very small number of users.

Because [Wuala] is a commercial product, the architecture is mostly unknown. Thus, comparing it with IgorFs is difficult. Wuala uses a similar peer-to-peer network base and has more similarities with a file-sharing program than with a real file system. The use of erasure coding is already a strong hint that real file system properties are not really desired because even on relatively small changes in files, the whole file has to be re-uploaded.

Similar to Wuala, Tahoe [Wilcox-O'Hearn and Warner, 2008] is not really a file system and is also difficult to compare to IgorFs. Furthermore, does not use a peer-to-peer network to store the data. Instead, the system design is based on a client/server design [Tahoe Architecture] and depends on centralized instances for finding the storage servers. The servers itself are used as key/value stores. Tahoe usually uses a private API for file accesses; in principle it is also possible to access Tahoe via Fuse; however the Fuse interface is incomplete [Tahoe Summary]. Furthermore, when using Tahoe via Fuse it does behave different from other file systems, which can lead to problems when programs want to access or modify stored data [Warner et al.]. In contrast to this, IgorFs tries to keep the usual POSIX access semantics wherever possible.

BitVault [Zhang et al., 2007] is again not a fully fledged file system, but only an object storage layer. However, the basic architecture of BitVault is very similar to IgorFs. Like IgorFs, BitVault uses a DHT. Also like IgorFs, the DHT is not used to store the actual data contained in the system. Instead, it only is used to store the current location of a data object. However, the similarities between the systems end here. In contrast to IgorFs, BitVault is designed to be only used in high-bandwidth, low-latency environments. Each peer of the system has to know all other peers that are currently active in the system. Furthermore, BitVault offers no file system access semantics – it supports no access permissions of any kind, no updates of stored data and no directory structures.

Part II

Redundancy Maintenance

6. Redundancy Coding Schemes for Distributed Storage

A problem all peer-to-peer storage systems have to face, is keeping the data that has been stored in the network available. In peer-to-peer systems, nodes may leave the network at any time – either because they are shut down by their administrator or user – or due to other unforeseen influences such as software, hardware, network or power failures. This process is known as *churn* [Binzenhöfer and Leibnitz, 2007].

In case of a regular shutdown, also called a *friendly leave*, a node notifies its peers that it is leaving the network. It can also transfer a limited amount of data to other nodes. The transfer amount is limited by the time the shutdown may take; if it takes too long either the user or the operating system will intervene and kill the process.

In case of a forced shutdown, also called a *node failure*, the node is unable to notify its peers and is also unable to transfer any data to the rest of the network. Software using peer-to-peer techniques has to account for node failures. Node failures have to be tolerated up to a certain sensible threshold. It is unrealistic to assume that no data is lost, e.g. in a decentralized file system, if a huge number of peers fail simultaneously.

Data loss in the case of node failure is tolerated by generating *redundancy* in the network. The definition of redundancy in engineering is:

Definition 6.1 (from [McKean, 2005]). *Redundancy: the inclusion of extra components that are not strictly necessary to functioning, in case of failure of other components.*

In a peer-to-peer networking context, redundancy means that, in addition to the necessary data, redundant information – which is usually not required – is stored on the network nodes. If some nodes leave the network and thus the data on these nodes is no longer available, the missing data can be reconstructed using the redundant information.

Storing redundancy in the network is expensive. It costs storage space on the individual nodes and the redundant data has to be inserted into the network, resulting in more insertion traffic. Furthermore, especially in highly unstable networks, the redundancy in the network has to be *maintained* to prevent data loss due to cumulative node failures. The checks for the availability of the redundancy blocks become more expensive with a higher level of redundancy because a bigger amount of blocks on a larger number of nodes has to be checked. Thus, the amount of redundancy stored in the network should be limited to the *minimum amount necessary* to prevent data loss.

The necessary amount of redundancy depends on many factors such as, for example, the *churn rate* of the network and the desired level of reliability. In networks with low churn rates, a lower redundancy is sufficient than in a network with a high churn rate where nodes tend to fail very often.

Those network characteristics usually cannot be influenced by software design. However, the necessary amount of redundancy also depends on characteristics that can be influenced when designing a peer-to-peer system, like, for example, the employed *redundancy coding scheme*. Several of these schemes are discussed in this chapter.

As hinted above, peer-to-peer storage systems require some kind of *redundancy maintenance*. Without redundancy maintenance, more and more blocks will fail from the network over time as the node failures accumulate. Thus, data in the network will become unavailable after a long enough amount of time. A redundancy maintenance algorithm continually checks the consistency of the stored information and regenerates lost blocks. The overhead imposed by such an algorithm should also be as low as possible.

Despite its great importance, only few works investigate the practical details of suitable redundancy mechanisms in the context of peer-to-peer file systems. Many authors study the theoretic properties of various different coding schemes, but only few consider the availability and durability of data that is protected with these codes. The works which consider these topics choose metrics which are not fitting for general purpose file systems.

In Section 6.1 this chapter first presents the redundancy metrics that are important for a decentralized file system. After that, Sections 6.2 and 6.3 introduce and compare the two most widely known and used redundancy schemes, *replication* and *erasure coding*. Section 6.4 discusses a slightly more complex *hybrid coding* scheme which uses a combination of replication and erasure coding. Section 6.5 introduces the use of *Random Linear Coding* as a redundancy scheme. Finally Section 6.6, analyzes the related work in the area of redundancy algorithms for peer-to-peer storage systems and Section 6.7 lists the contribution of this work and details how this work differs from previous works

The effectiveness of the different algorithms in the remainder of this chapter is illustrated with a number of graphs. All graphs in this chapter assume a peer-to-peer network with 40 000 nodes and a *redundancy factor* of 3, i.e. for every data block there are three additional redundancy blocks. Note that nearly all graphs in this section show analytical results that are directly derived from the given formulæ. Graphs which were generated with simulations are marked and the reasons for this are given in the corresponding text.

All simulations in this thesis were generated using a custom-built simulation environment. Existing network simulation environments like [OMNeT++] or [SSFNet] are much more complex than the used one. Furthermore, they offer a feature set that does not fit the requirements of the work. Most of the features offered by these environments are not needed for the simulations of this thesis because the simulations in this work are network agnostic. For traffic calculations only the number and type of exchanged messages are counted, no specific network topology or protocol usage is assumed. Without the need of a network modeling layer, the complexity of these environments makes their usage very unappealing.

The PeerSim peer-to-peer simulator [Jelasity et al.] in cycle based mode is the closest match to the requirements of this work known to me. But it still supports many advanced features like the possibility to run multiple protocols at once that just introduce more complexity to the code without offering any advantages for the problem at hand. The developed simulation environment is described in Appendix A.

Chapter 7 evaluates the schemes presented in this chapter in several realistic scenarios using real-world network traces.

6.1. Evaluation Scenario

This thesis analyzes several classes of redundancy codes in terms of effectiveness and efficiency for peer-to-peer storage systems. In order to determine these figures, the different redundancy codes are used in a realistic simulation of a peer-to-peer system. The different of these simulations is then evaluated. The most important information is the data loss after a variable amount of time for the different schemes.

For the evaluation scenario, an underlying storage network that is based on peer-to-peer techniques is assumed. A predetermined amount of information is stored into the network by an outside peer using one of the redundancy coding schemes. To guarantee the fairness of the simulation, every coding scheme may initially store the same amount of data into the network – including the redundancy information. It is assumed that the transfer of all information into the network takes place before the start of the simulation. Hence, there are no node failures during the insertion of the original data into the storage network. In my opinion, such failures would only complicate the task without offering any new insights. The inserting node could, for example, always check the presence of the data just after it finished uploading all parts. Furthermore, in real-world scenarios the data is inserted gradually into the network. Thus, large scale failures during the writes are impossible.

In the remainder of this thesis, two different kinds of simulations are evaluated. In this chapter and the beginning of the next chapter, a purely static approach is taken. Data is inserted into the network without maintaining it afterwards. Because failed data blocks are not regenerated, more and more data gradually fails. The primary result of these simulations is the *block failure rate* – the number of blocks that cannot be reconstructed at a certain time. The failure rates of the different storage schemes are evaluated and compared against each other.

In Section 7.2, a redundancy maintenance algorithm is presented. The different redundancy coding schemes are examined again in a network using this algorithm. In these simulations, in addition to the failure rate, the traffic that is exchanged between the individual nodes is compared. More specifically the number of *data probes* and *fetches* that are exchanged is compared.

The basic units used during the simulations are:

- *Blocks*: It is assumed that all blocks that are stored into the system have the same fixed size. A block is the smallest unit of data present in the system.
- *Probes*: A probe is a request and reply message pair exchanged between two nodes to determine the presence of a specific block on the probed node.

- *Fetches*: A fetch is the transfer of a block from one node to another node including the request message.

Another interesting metric that is recorded in the simulations is the percentage of nodes that may fail until the *first* data block can no longer be read. In a distributed file system even the loss of a single data block may be catastrophic – if it is in an important file. Thus, it is important to know which redundancy algorithm manages to lose data after the longest amount of time.

This evaluation assumes a usage scenario where the data inserting node is not a part of the storage network itself. There are several reasons for this choice. Firstly, this is the exact usage semantic that can be found in the current file storage applications that are in use in the Internet, known as *cloud storage*. Examples for such applications are [Dropbox][1] or [SpiderOak]. These applications run on the local machine and store the data into "the cloud", hence into a set of machines from which the data can be retrieved at a later time. The exact architecture of the server software for these applications is secret – however, these services are providing a distributed storage system and thus have to be use some kind of redundancy coding algorithm, most likely erasure coding. The simulations present exactly the scenario of such services.

The second reason for this choice is that network simulations where the data inserting peer is part of the system are much more complex: no single peer may insert the whole data because, otherwise, the complete data of the network would always be present at this peer unless it has failed. Instead, a number of peers that depends on the network characteristics have to insert small parts of data in an optimal way in the network. Future works could handle these scenarios; this is discussed in Section 7.3 on page 131.

As shown in the previous chapters, peer-to-peer file systems use many different underlying network structures. This thesis makes no assumptions about the underlying network. The algorithms can be applied to any kind of peer-to-peer network. Therefore, special network characteristics are not exploited. In practice the algorithms can be adjusted to the specific characteristics of the used peer-to-peer network. For example, in Chord [Stoica et al., 2001], which was described in Section 3.6.2, hosts from the *neighbor set* could be used for data replication.

6.2. Replication

Replication is the most basic technique to ensure data availability and durability. Replication provides redundancy by storing exact copies of data blocks on different nodes in the network. Assuming a large number of nodes and independent failures of these nodes, the probability to lose all replicas of a block – and thus to lose the data – is

$$p = \left(\frac{M}{N}\right)^r$$

[1] For a brief glimpse into the architecture of Dropbox and a comparison of several more cloud storage applications, see [Hu et al., 2010]

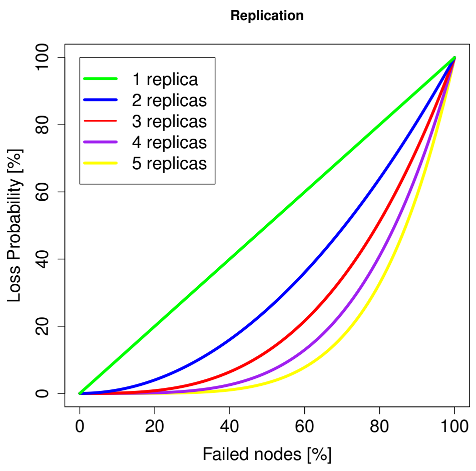


Figure 6.1.: Block losses for replication

where N is the total number of peers in the system, M is the number of unavailable peers, and r is the number of replicas.

Replication is easy to implement: When reading data, any of the available replicas can be read. When a replica gets lost, the required redundancy can be restored by creating another copy of any of the remaining replicas. To replace data in the file system or to add new data to the system, new blocks are written to r nodes.

Figure 6.1 shows the probability that a data block is lost when a given percentage of total peers of the system have failed. Note that a higher number of replicas – and thus a lower failure probability – also means that more data chunks have to be stored in the network and have to be kept alive.

Table 6.1 shows the percentage of nodes that have to fail until the first data block is lost. It is evident that replication does not perform well for this metric.

6.3. Erasure Coding

The second common redundancy scheme is *erasure coding*. Maximum Distance Separable (MDS) erasure codes [Singleton, 1964] are the optimal erasure codes in terms of redundancy vs. reliability. The best known family of MDS erasure codes are Reed-Solomon codes

Combined blocks	Random linear coding	Erasure coding	Hybrid coding	Replication	Theoretical optimum
2	24.9% ±4.1%	21.0% ±3.6%	14.9% ±3.0%		
3	37.6% ±4.4%	29.7% ±3.7%	19.4% ±2.9%	9.1% ±2.5%	75.0%
4	47.1% ±4.4%	35.5% ±3.7%	22.3% ±2.9%		

Table 6.1.: Average node failure probability until the first data block is lost.

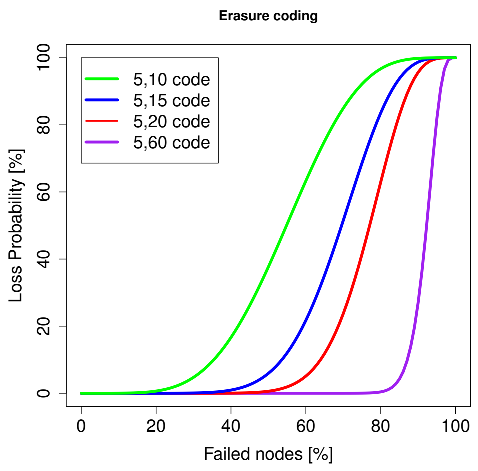


Figure 6.2.: Block losses for erasure coding

[Reed and Solomon, 1960]. Whenever this work refers to erasure codes, it refers to MDS erasure codes unless otherwise stated.

When using a $(m,\ n)$ MDS erasure code, a block b is expanded into n erasure code fragments f, with size$(b) < n\cdot$size(f). m of these fragments suffice to reconstruct the original block, with size$(b) = m\cdot$size(f). If less than m fragments are available, the original data cannot be recovered.

For this coding scheme, the probability of data loss can be determined by using the probability mass function for the binomial distribution. A data block can be reconstructed if at least m fragments can be read. This is equal to the probability that at least m Bernoulli experiments succeed. The probability for this event is $P(X \geq m)$. The probability that a data block cannot be reconstructed is the probability of the complementary event

$$p = 1 - P(X \geq m) = 1 - \sum_{i=m}^{n} \binom{n}{i} \left(1 - \frac{M}{N}\right)^{i} \left(\frac{M}{N}\right)^{n-i}$$

with M being the number of unavailable peers and N being the number of total peers.

Figure 6.2 shows the probability that a data block is lost for several different erasure code settings. It shows the percentage of available blocks when a specified percentage of nodes have failed. Note that like for replication in the previous section more storage is required for a lower probability of data losses.

6.3.1. Erasure Coding in Distributed File Systems

Erasure coding is very efficient. For example, consider a network where the goal is to have a loss probability of less than 5% when 40% of all nodes have failed. For replication, the redundancy factor is at least 4. Hence, for a 100 Megabyte file, 400 Megabytes of redundancy have to be stored in the network. For erasure coding, e.g. a $(52, 100)$ code has a loss probability of 4.2%; Hence, about 193 Megabytes of data have to be stored.

This example already shows that erasure coding is more efficient when used with relatively big values for n. In this case, the $n - m$ redundancy blocks constitute only a relatively small overhead in comparison to other coding schemes; for MDS codes the overhead can even be reduced to the theoretically possible minimum. When using smaller values for n, and thus larger fragment sizes, erasure coding becomes much less efficient; for example, with $n = 10$, a $(3, 10)$ coding has to be used for a loss probability $< 5\%$ – hence 334 Megabytes of data have to be stored in the network.

This is one of the reasons that make erasure coding a problematic choice for file systems: When using big values for n, this either means that the block sizes have to be very small or that the amount of data in the erasure groups is large. Published work typically assumes that a whole file is encoded using erasure codes. This is not a reasonable assumption for all distributed storage systems. There are lower bounds to block sizes in distributed storage systems since these systems quickly become inefficient with very small blocks. Thus, the block size cannot be decreased. This means that for large values of n, a big amount of data is combined into one erasure group.

But in erasure coding, typically, a modification of one block requires all redundancy blocks to be re-encoded and re-inserted into the distributed file system[2]. Thus, the amount of data that is encoded has to be kept small; otherwise the overhead for file modifications is unacceptable. Moreover, when one block has been lost, m blocks need to be retrieved from the file system to recover the lost block. Hence, the value for n has to be kept small. A value of about $n = 10$ seems to be realistic – to change one data block, 9 other data blocks have to be updated.

Tahoe is a distributed storage system under active development that operated commercially for some time and uses erasure coding (see Section 4.4.11 on page 74). Tahoe uses Reed-Solomon codes, a MDS erasure coding scheme just as it was assumed in this thesis. And also like assumed in the thesis, they chose a very small number of blocks to combine – the default installation uses a (3,10) erasure coding scheme, which was also used in the commercial service [Wilcox-O'Hearn and Warner, 2008, p. 21]. Another reason for choosing these small values mentioned by the authors was the computational overhead of erasure coding. The overhead grows with larger values of n. However, for small values of n the overhead is negligible. Wilcox-O'Hearn and Warner [2008] state that for $n = 10$ the computational cost of erasure coding was "lower than the cost of the encryption and secure hash algorithms" used by Tahoe. For an evaluation and comparison of the speed of several different erasure coding algorithms and libraries see [Plank et al., 2009].

Erasure coding also introduces a higher complexity into the system. Even when used on a per-file basis, the community is divided if the added complexity is worth the encoding advantages [see e.g. Rodrigues and Liskov, 2005]).

Note that the restriction that at least m blocks have to be retrieved also holds for the more recent rateless erasure codes like fountain codes [Byers et al., 1998; Shokrollahi, 2006]. Even though it is possible to generate arbitrarily many new redundancy symbols using these codes, a node needs to have knowledge of the entire original message to

[2]Alternatively, deltas to the stored data could be saved. However, using deltas is problematic both from a performance and a security point of view.

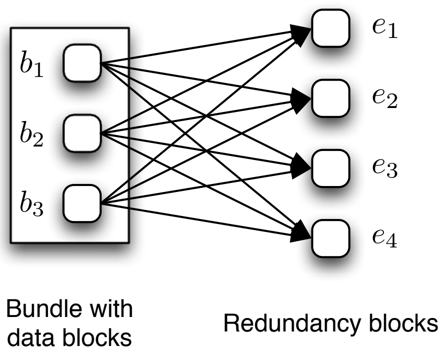


Figure 6.3.: Erasure block combination

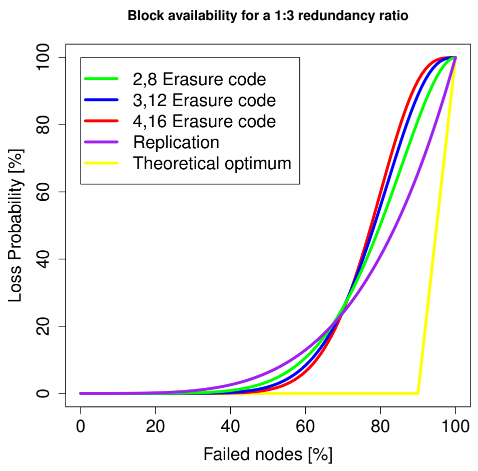


Figure 6.4.: Erasure coding

generate these symbols. Alternatively, it needs enough symbols to reconstruct the original message. These coding schemes also have a higher storage overhead than MDS codes and many of the codes are patented. Furthermore, there is no consensus on how such rateless coding schemes can be used in a distributed storage environment [Rodrigues and Liskov, 2005, p. 228]. For these reasons they were not examined in this thesis – other authors did not consider them for exactly the same reasons [see e.g. Rodrigues and Liskov, 2005; Wilcox-O'Hearn and Warner, 2008].

6.3.2. Comparing Erasure Coding

Section 6.1 mentioned that uniform block sizes are used in the simulations. Single blocks cannot be split into smaller sub-blocks. Hence, erasure coding cannot be applied in the traditional sense.

Instead, when using a (m, n) erasure code, m original data blocks are chosen and expanded into n erasure-coded blocks. The m different file system blocks share the same fate. If one of those m blocks can no longer be reconstructed, all of the blocks can no longer be reconstructed. Figure 6.3 illustrates this behavior. It shows $m = 3$ data blocks which are encoded into $n = 4$ erasure blocks.

Figure 6.4 compares erasure coding and replication in a scenario where both coding schemes may store the same amount of data in the network. It shows that erasure coding performs better than replication, especially for larger m. (Replication could be viewed as (1,4) erasure code in this setting.) Still, the overall behavior of the schemes is similar.

Note, when combining more blocks arbitrarily, good results can be archived with erasure coding; but due to the aforementioned constraints of distributed storage systems, the fate-sharing has to be kept at a minimum. For these reasons, I chose the (2,8), (3,12), and (4,16) coding in the comparison. Note that (3,12) coding already offers a higher reliability than the coding scheme used by Tahoe. Additionally, the simulations in Section 7.2 show that bigger block sizes also cause more maintenance traffic.

Table 6.1 on page 95 gives the number of nodes that may fail until the first data block is lost. The numbers have been generated in a simulation assuming 10,000 data blocks and 30,000 redundancy blocks. The simulation was repeated 500 times for each case.

6.4. Hybrid Replication–Erasure Coding Scheme

Often, hybrid schemes between erasure coding and replication are proposed [see e.g. Rodrigues and Liskov, 2005; Williams et al., 2007; Chen et al., 2008a]. In such schemes, one copy of the original data always remains in the system in addition to the erasure coded data. The reason for this is that the regeneration of erasure code fragments is very costly. To regenerate a fragment of a (m, n) MDS erasure code, a node needs to download $m-1$ fragments (assuming that the node already stores 1 fragment). However, with hybrid coding, if one of the erasure code fragments is lost, the node with the original data can directly regenerate the lost fragment. $m-1$ fragments only need to be downloaded if the original data is lost.

When considering Figure 6.3 on page 98 this means that in contrast to normal erasure coding the bundle containing b_1, b_2, and b_3 is also stored in the network alongside e_1, e_2, e_3, and e_4.

In this case, the probability of failure is the probability of erasure coding multiplied by the probability that the original data block is lost:

$$p = \left(\frac{M}{N}\right)\left(1 - \sum_{i=m}^{n}\binom{n}{i}\left(1-\frac{M}{N}\right)^{i}\left(\frac{M}{N}\right)^{n-i}\right)$$

In the formula, M is the number of unavailable nodes, N is the number of total nodes, n is the total number of erasure code fragments, and m is the number of erasure code fragments needed to be able to restore the data.

Figure 6.5 on the next page shows the probability that a data block is lost for several different hybrid coding settings; note again that a lower failure probability comes with a higher storage overhead.

Figure 6.6 compares three different hybrid coding schemes, where all schemes may store the same amount of data in the network. Like erasure coding, hybrid coding performs better than replication and also gets better for larger m.

Figure 6.7 on the following page shows a comparison between erasure coding and hybrid coding where both schemes may store the same amount of data in the system. The same value of $n = 4$ is chosen for both schemes to allow a fair comparison. Note that, using this setting, erasure coding uses $m = 16$ and hybrid coding $m = 12$ because hybrid coding also stores the 4 original blocks in the network. The picture looks similar for different choices of n. Erasure coding is always a bit more efficient than the hybrid scheme.

Table 6.1 on page 95 again shows the number of blocks that may fail until the first data block is lost.

In this completely static setting, hybrid coding is at a distinct disadvantage in comparison to erasure coding. This was to be expected because the cited advantage of the hybrid coding scheme, the reduced repair traffic in case an erasure block is lost, does not

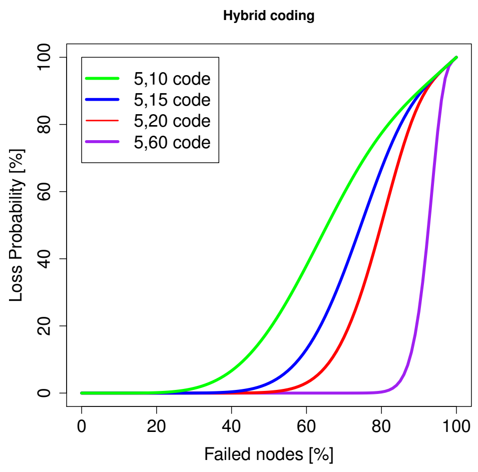


Figure 6.5.: Block losses for hybrid coding

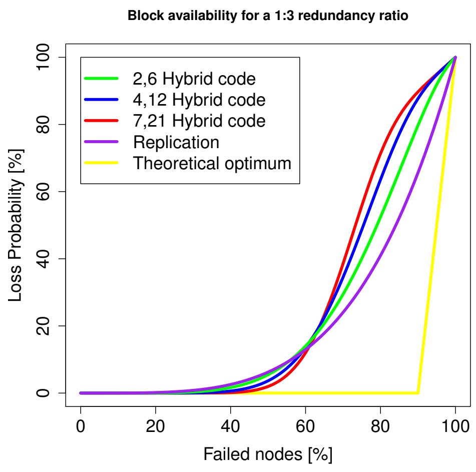


Figure 6.6.: Hybrid codes with same storage requirements

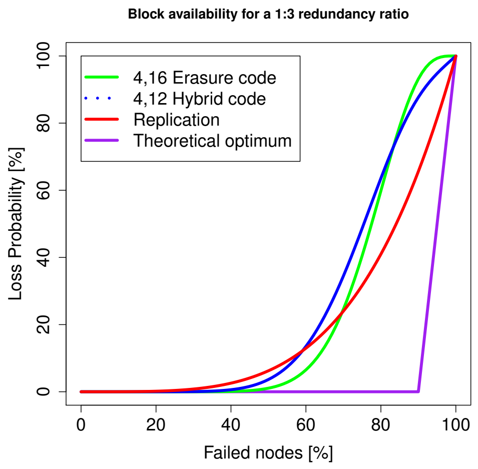


Figure 6.7.: Hybrid coding versus erasure coding

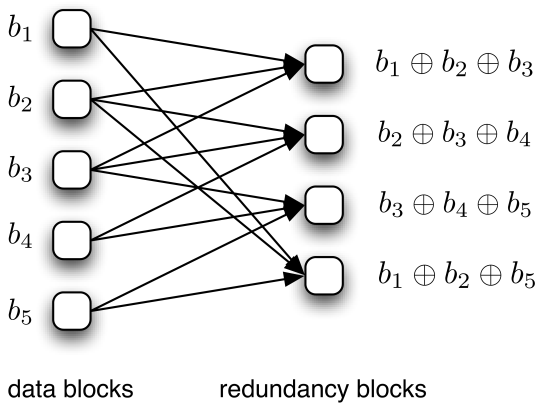

b_1
b_2
b_3
b_4
b_5
$b_1 \oplus b_2 \oplus b_3$
$b_2 \oplus b_3 \oplus b_4$
$b_3 \oplus b_4 \oplus b_5$
$b_1 \oplus b_2 \oplus b_5$
data blocks
redundancy blocks

Figure 6.8.: RLC block combination

apply in a purely static evaluation. In Chapter 7, when evaluating a setting where data in the system is continually refreshed, a fairer comparison will be possible.

6.5. Random Linear Codes

The fourth algorithm analyzed is a simple linear block code, which chooses the encoded blocks randomly and is thus called Random Linear Coding (RLC) in this thesis.

Linear block codes expand a message with m information bits into a message with $n > m$ information bits, where the additional bits are used for error correction [see e.g. Ryan and Lin, 2009, p. 95ff]. The aforementioned Reed-Solomon-Codes are an example of a linear block code.

The basic idea for the RLC code presented in this work is a slight variation of this basic idea (compare Figure 6.8):

1. Combine m file system blocks via XOR to create one redundancy block:

$$r = b_1 \oplus b_2 \oplus \ldots \oplus b_m$$

2. Repeat this step with different combinations of input blocks until a sufficient amount of redundancy has been achieved.

A resulting redundancy block r can be used to reconstruct any of its original blocks b_i, provided that all b_j with $i \neq j$ are still available. This is very similar to the way RAID-5 hard disk arrays operate.

The presented code is a linear code, where m input blocks are transformed into $m+1$ output blocks using the XOR operation. Formally, if each of the m blocks has k bits and the ith bit of the mth block is known as $b_m(i)$, the input message u for the code is

$$u = (b_1(1), b_1(2), \ldots, b_1(k), b_2(1), \ldots, b_m(k))$$

The generator matrix G for the code is:

$$G = \begin{array}{c} \begin{array}{cc} \overbrace{\hphantom{1\;0\;0\;\cdots\cdots\cdots\cdots\cdots\;0}}^{n\cdot k \text{ message bits}} & \overbrace{\hphantom{1\;0\;0\;\cdots\;0}}^{k \text{ red. bits}} \end{array} \\ \left(\begin{array}{cccccccccc|ccccc}
1 & 0 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 1 & 0 & 0 & \cdots & 0 \\
0 & 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 0 & 1 & 0 & \cdots & 0 \\
0 & 0 & 1 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 0 & 0 & 1 & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & & & & & & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 1 \\
0 & 0 & 0 & \cdots & 0 & 1 & 0 & 0 & \cdots & 0 & 1 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & \cdots & 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 1 & 0 & \cdots & 0 \\
0 & 0 & 0 & \cdots & 0 & 0 & 0 & 1 & \cdots & 0 & 0 & 0 & 1 & \cdots & 0 \\
\vdots & \vdots & \vdots & & & & & & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 1 & 0 & 0 & 0 & \cdots & 1
\end{array}\right) \end{array}
\begin{array}{l} \\ \left.\vphantom{\begin{array}{c}1\\1\\1\\1\\1\end{array}}\right\} k \text{ lines} \\ \left.\vphantom{\begin{array}{c}1\\1\\1\\1\\1\end{array}}\right\} (m-1)\cdot k \text{ lines} \end{array}$$

Multiplication of u with G gives the codeword v:

$$v = u \cdot G$$

$$v = (b_1(1), \ldots, b_1(k), b_2(1), \ldots, b_m(k), (b_1(1) \oplus \ldots \oplus b_1(k)), \ldots, (b_m(1) \oplus \ldots \oplus b_m(k)))$$

v is then split into $m + 1$ k-bit blocks.

The operations required for random linear coding are very simple and computationally inexpensive in comparison to the presented erasure coding scheme.

Unfortunately, giving an explicit formula to calculate the block loss probability is difficult for RLC. This is, in part, due the high number of algorithm parameters that can be varied. The first obvious parameter is m, the number of blocks that are intertwined. The second parameter is the number of redundancy blocks. The algorithm naturally performs better with a higher number of redundancy blocks. The third not so obvious parameter is the way the blocks to be combined with each other are chosen. This third parameter is very important and can significantly alter loss probability. Representing it in a formula is particularly difficult – probably one formula is needed for each different block distribution method. Even when having a fixed block generation scenario, the calculation of the probability is difficult because of the multitude of possibilities in which blocks can be combined.

A reason for this is shown in Figure 6.9: In this example, the block b_1 can be reconstructed by first reconstructing b_3 by calculating $r_3 \oplus d_4$. After that b_2 and finally b_1 can be reconstructed. When dealing with scenarios where more blocks are intertwined and where, for example, the redundancy blocks may also be intertwined with other redundancy or data blocks, things get even more complex.

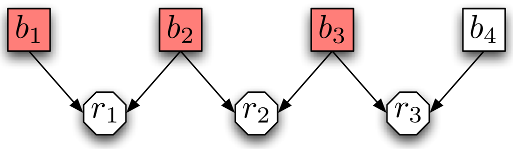


Figure 6.9.: RLC restoration chain

To the best of my knowledge, at the time of writing, there is no known formula to calculate RLC loss probabilities for non-trivial scenarios. For this reason all graphs in this section are derived from simulations.

A upper bound for the probability that a data block can not be reconstructed can be given by calculating the probability that either a data block is available or one of the redundancy blocks and all the directly intertwined data blocks are available. Thus, the upper bound for the probability is

$$p = 1 - \frac{M}{N} \cdot \sum_{i=1}^{r} \binom{r}{i} \left(1 - \frac{M}{N}\right)^{mi} \left(\frac{M}{N}\right)^{m(r-i)}$$

with M being the number of unavailable peers, N being the number of total peers, m being the number of blocks being intertwined and r being the number of redundancy blocks for this data blocks. This assumes that neither the redundancy blocks nor the intertwined data blocks can be reconstructed using other redundancy or intertwined data blocks – which is a very pessimistic assumption. In practice the loss probability will be significantly lower.

6.5.1. RLC Block Generation

As mentioned, the performance of RLC depends on the number of redundancy blocks, on the number of blocks to be combined with each other to form a redundancy block, and on the way blocks which are combined are chosen.

The simulations done over the course of this work show that all these parameters have to be chosen carefully to guarantee the best results.

In the first preliminary RLC simulations done by me, the blocks that were to be combined were chosen completely randomly from all blocks present in the storage networks. New redundancy blocks were generated by combining data blocks with data blocks, redundancy blocks with data blocks, or redundancy blocks with redundancy blocks; some blocks were protected multiple times, some blocks were not protected at all. Clearly, this is not a good system design – yet even this very simple trial yields surprisingly good results.

Figure 6.10 on the following page shows this. While random linear coding loses more blocks than replication when more than approximately 50% of the nodes have failed, it loses significantly fewer blocks for node losses less than 50%. This is a very encouraging result because even a single block failure can be catastrophic for a file system. Thus, it is important that the block losses are kept as low as possible for as long as possible. When this is no longer possible it will probably not matter if 40% or 60% of the blocks in the file system are no longer readable – the file system is lost anyway.

Detailed examinations of the simulation results for this first very simple setting show that – unsurprisingly – the number one reason for the loss of a data block is that it has no redundancy blocks at all because it was not chosen by the random algorithm.

The best method found to generate redundancy blocks for decentralized storage systems over the course of this work is to combine the blocks according to an equal distribution. Each block gets the same amount of redundancy blocks. Somewhat surprisingly, the

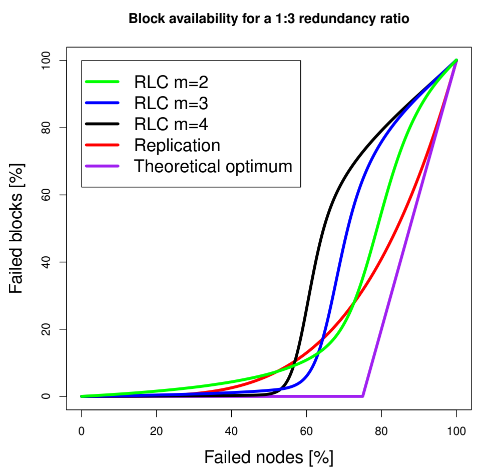


Figure 6.10.: RLC random block comparison (simulated)

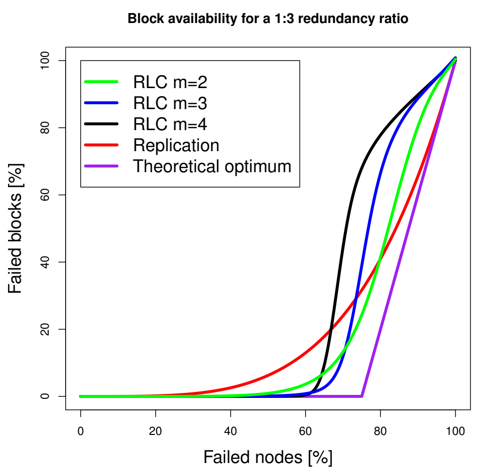


Figure 6.11.: Random linear coding (simulated)

scheme works best if only data blocks are combined into redundancy blocks and redundancy blocks themselves are not protected by a second order of redundancy blocks. This is due to the fact that each scheme may initially commit a uniform number of blocks to storage. Protecting a redundancy block from failure also means that a data block cannot be protected with an additional redundancy block.

Figure 6.11 shows the evaluation for this scenario. The result is an obvious advancement in comparison to Figure 6.10.

The algorithm which is used to generate the redundancy blocks in the simulations is shown in Algorithm 1 on the facing page. The algorithm assumes global knowledge. In a real world system the redundancy algorithm would generate the redundancy blocks by choosing blocks that have less than a given number of available redundancy blocks until the desired redundancy level has been reached.

6.5.2. Comparison

As evident in Figure 6.11, RLC offers a significant improvement over replication.

For $m = 2$, random linear coding outperforms replication until about 75% failed nodes. For larger values of m, these figures are slightly worse. However, the curves for larger values of m are steeper and the first block loss occurs at a later point in time. RLC only fails more quickly after the first data blocks are lost. This is shown in Table 6.1 on page 95. At the time RLC loses its first data block, replication has already lost about 5% of its data. RLC also significantly outperforms erasure coding when considering this metric.

Figure 6.12 on page 106 shows a direct comparison of the (4,16) erasure code, replication, and RLC. As expected, erasure coding clearly outperforms replication. However, all RLC variants outperform the erasure (4,16) code. Up to about 85% node failure probability, RLC with $m = 3$ loses less blocks than erasure coding. (For $m = 2$ RLC outperforms a

Algorithm 1: RLC block distribution

```
Input:
  $B$: set of all data blocks
  $m$: number of blocks combined into a RLC block
  $r$: total number of redundancy blocks
Output:
  Set of redundancy blocks
1  function blockDistribution($B, m, r$)
2  begin
3      Randomized $\leftarrow \emptyset$
4      RedundancyBlocks $\leftarrow \emptyset$
5      for $i \leftarrow 1$ to $\frac{r}{m}$ do
           // Randomized is emptied several times during block distribution
6          if Randomized $= \emptyset$ then
7              Randomized $\leftarrow$ randomPermutation($B$)
8          CombineBlocks $\leftarrow$ first $m$ blocks in Randomized
9          Randomized $\leftarrow$ Randomized $\setminus$CombineBlocks
10         ResultBlock $\leftarrow 0$
11         for $b \in$ CombineBlocks do
12             ResultBlock $\leftarrow$ ResultBlock $\oplus b$
13         RedundancyBlocks $\leftarrow$ RedundancyBlocks $\cup$ { ResultBlock }
14     return RedundancyBlocks
```

(4,16) erasure coding even up to about 95% node failures.) Erasure coding only performs better than RLC when there is a very high percentage of failed nodes. But in these cases the amount of lost data probably already makes the whole system unusable.

The sharp rise of the RLC block failure probability means that, just before that rise, RLC can recover about 10 percentage points more data than erasure coding. I consider this the most important advantage of RLC.

Figure 6.13 examines the reasons for RLC data loss (for $m = 3$). It shows the probability that

- a data block is lost,
- a data block is lost because there was no redundancy block left, and
- a data block is lost because at least one of the data blocks belonging to the same redundancy group was lost.

The plot demonstrates that, in almost all cases, the data block could not be reconstructed because there were no redundancy blocks left for that block. Only at very high node failure probabilities, where there is already a significant number of failed blocks, the graph shows block failures that are not caused by the complete absence of redundancy blocks.

As already mentioned in the previous section, experiments were also done with block distribution schemes where redundancy blocks are protected by a second layer of redundancy blocks. The results of these experiments were not encouraging. Using this scheme,

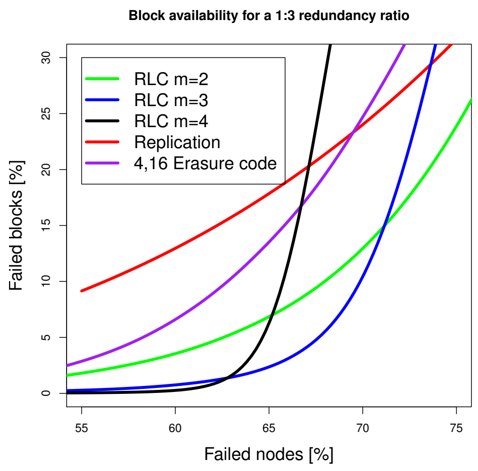


Figure 6.12.: Erasure code combination in detail (simulated)

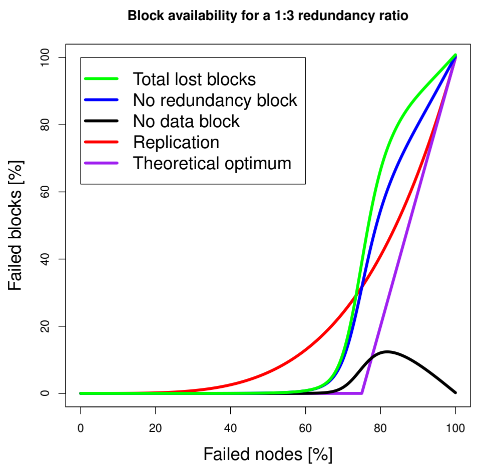


Figure 6.13.: Block loss reasons (simulated)

it might be possible to reconstruct a lost redundancy block and use this redundancy block to reconstruct a data block. But, if the total numbers of blocks that may be created at the start of the simulation is kept constant, each redundancy block protecting a redundancy block means that there is one less redundancy block for a data block. In practice, the scheme is also more complex because the nodes would have to resolve longer, more complex chains of redundancy blocks to reconstruct their data blocks.

6.6. Related Work

All existing distributed file systems that the author is aware of either use erasure coding, replication or a combination of these schemes. Systems using replication include Farsite [Adya et al., 2002], Ivy [Muthitacharoen et al., 2002], Keso [Amnefelt and Svenningsson, 2004], Pastis [Busca et al., 2005], BitVault [Zhang et al., 2007], and DRFS [Peric, 2008; Peric et al., 2009]. Erasure coding is, for example, used by Wuala and Tahoe [Wilcox-O'Hearn and Warner, 2008]. All these systems have been discussed in Chapter 4.

Bhagwan et al. [2004] decided to use both replication and erasure coding in their Total Recall File System (TRFS), which was introduced in Section 4.4.4. Replication is used for files with a size of less than 32 kilobytes. For files with a greater size, on-line codes [Maymounkov and Mazières, 2003] are used. On-line codes are a non-optimal variant of erasure codes They are computational efficient and have a low handshaking overhead between peers. Total Recall also features a data distribution algorithm, which aims to send data that is not yet present in the network to downloading nodes. Thus, when two nodes download from a single source and this source leaves the network before the download is finished, the downloading nodes can try to reconstruct the missing data by combining the blocks they received. This is similar to the super-seed mode of many Bittorrent clients [Chen et al., 2008b].

OceanStore [Kubiatowicz et al., 2000] also uses both replication and erasure coding. Replication is used for normal files and erasure coding for archive storage.

There has been a great deal of discussion in the community as to whether replication or other coding schemes are better suited for peer-to-peer storage systems. During their design of a Chord based DHT, Dabek et al. [2004] considered whether to use replication or IDA coding [Rabin, 1989], a coding scheme with very similar properties to erasure coding. They concluded that the choice depends on the usage scenario. Read-intensive workloads have lower latencies using replication, whereas write-intensive workloads consume less bandwidth with coding. However, updating partial files was not considered in their work; entries in a DHT are always replaced as a whole.

Zhang and Lian [2002] proposed a scheme that makes it feasible to replace replication with erasure coding in DHTs. The authors cut every file into a number of units, which are then divided into erasure code fragments. The individual fragments are stored in the DHT. The authors compare the efficiency of their erasure coding scheme with replication using a purely static approach similar to the evaluation of this chapter. They do not compare their scheme with other redundancy schemes and they do not consider the subject of data maintenance and the resulting repair traffic for the different schemes.

Utard and Vernois [2004] quantitatively studied data durability in peer-to-peer systems. Unlike the work in this thesis, their study only considered replication and erasure coding. The authors also did not use simulations based on real-world data, but based their evaluation on stochastic methods. The results of their work are consistent with the results of this thesis: they concluded replication performs better than erasure coding in high-loss environments and that erasure coding has a higher communication overhead.

Weatherspoon and Kubiatowicz [2002] compare erasure coding to replication. They identify the higher number of nodes that have to be contacted for erasure coding and the smaller size of the individual fragments as the most important disadvantages for erasure coding. The authors concede that “both of these issues could be considered enough of a liability to outweigh [the advantages of erasure coding]”. They also concede that the read overhead is increased for erasure coding. Nevertheless, they conclude that erasure coding is to be preferred over replication. To mitigate the problems of erasure coding, they propose a soft-state replication algorithm which is used in addition to the erasure coded durability scheme. This is similar to the hybrid coding scheme analyzed in this work. Furthermore, they propose combining several erasure code fragments into larger messages. However, later works like [Rodrigues and Liskov, 2005] presented in the next section show, that hybrid coding schemes are not always superior to replication.

Rodrigues and Liskov [2005] suggested the hybrid scheme presented in Section 6.4. The scheme uses erasure coding, but always keeps a full copy of the data block available in the system. This is somewhat similar to the soft-state algorithm proposed by Weatherspoon and Kubiatowicz [2002]. However, the reasons for storing the data are different. In a distributed system where peers often leave the network, erasure coding generates a greater bandwidth overhead when repairing the lost redundancy than replication because a potentially big number of fragments have to be retrieved from the other nodes. This requires a significant amount of bandwidth, which is a limiting factor for the scalability of

P2P storage systems [Blake and Rodrigues, 2003]. Rodrigues and Liskov [2005] propose that using the hybrid scheme, missing erasure blocks can be reconstructed using the original data copy. The authors compare this hybrid scheme with replication. They conclude that erasure coding can result in a better availability in scenarios with low node availability. However, they advise that such coding schemes should only be used when necessary because of the complexity they add to the system.

Dimakis et al. [2007, 2010] introduce a new coding scheme which they call *regenerating codes*, which are a special case of erasure codes. The coding scheme has similarities to schemes already discussed in the context of content distribution [Gkantsidis and Rodriguez, 2005]. Regenerating codes aim at minimizing the repair bandwidth by reducing the bandwidth overhead of erasure coding schemes. The authors present a coding scheme that manages to restore lost blocks with a traffic overhead that is lower than for traditional erasure codes. They construct Minimum-Storage Regenerating (MSR) and Minimum-Bandwidth Regenerating (MBR) codes. As the names already suggest, MSR codes store the minimal necessary amount of information at each node and MBR codes have the minimal bandwidth requirement for reconstruction. The authors compare their scheme to the hybrid scheme of Rodrigues and Liskov [2005]. In comparison to the hybrid scheme, MBR codes consume slightly less bandwidth. However, they need communication among many more nodes and the scheme incurs an overhead on the network bandwidth that is required to read a file. For the examples given in their publication, the read-overhead is between 13 and 30%. The authors name the simplicity of their approach as an advantage: their system does not have to support two different redundancy schemes as with hybrid coding.

Because Dimakis et al. [2007, 2010] did not discuss implementation issues or the computational properties of regenerating codes, Duminuco and Biersack [2009] analyzed them in this respect. They concluded that these coding schemes come at a significant computational cost – their implementation could at most read or write about 1 gigabyte per hour from the network. The coding scheme is thus only suitable for scenarios, where no big amounts of data are read or written from the system, but where block repairs are fairly common. In my opinion, this severely limits the usability of this coding scheme for a peer-to-peer file system.

When comparing MSR codes with hybrid codes, MSR codes require slightly more bandwidth but less storage at the individual nodes. The other advantages and disadvantages remain the same. Because the hybrid coding scheme of Rodrigues and Liskov [2005] has very similar characteristics, regenerating codes are not discussed further in this thesis. When using a system design where hybrid codes are a good fit, one should consider replacing them either with MSR or MBR codes, depending on the requirements. To the best of my knowledge, these coding schemes have not been implemented yet. While they may be simpler in theory, traditional erasure coding is widely known in the community and a myriad of implementations are available. In my opinion, the scheme is also more complex than the presented RLC scheme.

Acedański et al. [2005] analyzed an approach that is more similar to the one in this thesis. The authors showed analytically and through simulation that random linear coding performs as well as erasure coding without requiring a central server to encode the data.

Unlike the results in this thesis, their analysis applies to the cooperative downloading of files in BitTorrent-like networks. Hence, they do not study data loss probabilities and probing traffic. They focus on files rather than individual blocks, which allows large encoding groups which are not suitable in the scenario of this thesis.

During the course of this work, Duminuco and Biersack [2010] presented a coding scheme that they call *hierarchical coding*, which is a new class of erasure coding. The first proposal of hierarchical codes was presented in [Duminuco and Biersack, 2008], a more detailed view was given in [Duminuco, 2009] and the latest description can be found in [Duminuco and Biersack, 2010] [3]. Hierarchical codes reduce the network traffic required for maintenance at the cost of a lower failure resilience. Briefly put, in contrast to traditional erasure codes, the distance of a fragment to another fragment is of importance. The nearer the fragments are to each other, the lower the number of fragments required to restore a missing fragment. Duminuco et al. used simulations with real-world network traces to compare their coding scheme to classical erasure codes. They state that hierarchical codes require a higher number of repairs, but a lower amount of data that has to be transferred over the network. However, in contrast to my work, their work does not include the probing traffic or any kind of decentralized mechanism to trigger repairs in their simulations. Instead, they assume the presence of a central entity, that is aware of all nodes in the network and is aware of all the pieces of information stored on all nodes. This entity is immediately aware when nodes join or leave the network and will trigger repairs when needed.

This has direct consequences for the choices of code parameters in their paper. They decided to combine a much larger number of blocks with each other, going up to 64 blocks that are intermingled. This choice is rather random, the authors state themselves that they did "not explore the tuning of the parameters", but leave that as future work [Duminuco and Biersack, 2010, p. 59]. This kind of coding is a poor fit for the decentralized probing algorithm presented in this work – and would lead to a very high number of exchanged probing messages.

Even without considering the probing traffic, their results are not easily comparable, due to the different ways the codes are built. Hierarchical codes are applied just as traditional erasure codes: given a fixed number of input blocks, a fixed number of output blocks is generated by a fixed scheme. This is explained in great detail in [Duminuco, 2009, p. 82f]. In contrast, the approach presented in this work assumes that the data in the system will be changed at later times and is in a state of flux. Because of this, the system presented in this work avoids any kind of scheme where the initial data blocks are fixed. Due to this difference in the preferred setting, any comparisons between the two schemes are biased, depending on the selected usage scenario.

The rest of the comparison will assume that the scheme presented in this work is used in the same way hierarchical codes are usually used - to encode a fixed, unchanging set of data. Please note that this is a restriction of our original goal - in such a scenario one would choose another coding scheme. When considering the primary goal of Duminuco and

[3] Please note that several different versions of this journal article exists. The cited version is the newest one which should be cited, according to the authors.

Biersack [2010] – which was to reduce repair traffic in their set scenario – the comparison for this metric is quite easy: The RLC algorithm presented in my thesis would certainly perform worse than their presented algorithm. This is due to the way their algorithm was optimized for this metric - they use the hierarchies in the code to be able to use a lower number of data transfer operations. The coding scheme presented in this work does not have a comparable feature, because it would not be applicable with the parameters chosen in this work.

Comparing the probability of data loss is more difficult without measurements. I believe it would be possible to get to a similar, but probably slightly worse level of reliability using the RLC code presented in this work by setting m to a quite high number like 16 or 32, which are comparable to the numbers used in their work. However, like aforementioned, this would result in a much higher repair traffic, making RLC very unattractive in comparison. Hence, the algorithm presented in this thesis would not be a good fit for their chosen environment.

Due to the way their coding scheme uses hierarchies it is not possible invert this comparison in order to give a fair comparison of their scheme with the parameters chosen in this work. When using the parameters as chosen in this work, there would not be enough blocks to build a multi-level hierarchy. Thus, the advantages of using hierarchies in the algorithm presented by Duminuco et al. (especially the lower repair traffic) would not manifest. Over the course of my work, a few tests also used a scheme, where redundancy blocks were protected with other redundancy blocks (see Section 6.5.2 on page 104). When doing this for e.g. with $m = 2$ this is basically equivalent to the (4,3) hierarchical code introduces as an example by Duminuco et al. However, this approach was abandoned because of a bad performance in the setting chosen in this thesis (and that parameter choice is also just mentioned as an easy example by Duminuco et al. They use much larger parameters to get good results).

In conclusion, the two algorithms are not directly comparable. The choice depends on the setting in which the algorithms will be used. With static data in a network that does not need probing, the algorithm presented by Duminuco et al. will perform better. With dynamic changing data in a network that needs a decentralized probing algorithm the RLC schemes presented in this work will be more adequate.

In addition to the regenerating and hierarchical codes, Duminuco [2009] presents an *adaptive proactive repair policy* that aims to “maximize the smoothness of the repair bandwidth needed“ [Duminuco, 2009, p. 97]. This was not a requirement of our usage scenario and was thus not examined in any way in this thesis.

Fountain codes are rateless erasure codes that can generate a virtually limitless number of encoded data blocks from a number of source blocks. The general mechanisms that are used to construct fountain codes are similar to those used in the construction of the RLC coding scheme in this paper. Especially, many of the different fountain codes also heavily rely on XOR operations. In the next few paragraphs the most well known fountain codes as well as the differences to the approach presented in this thesis are shown. For a more in depth introduction and comparison which also gives a number of use cases, please refer to [Mitzenmacher, 2004].

The first practically viable class of fountain codes were presented by Luby et al. [1997]. These coding schemes are today known as *Luby Transform* or *LT Codes*. The basic idea of LT codes is, that messages are sent from a sender to one (or several) receivers. These messages are generated from the source data by first splitting the source data into blocks of a uniform length and then XORing n of these blocks which were randomly chosen. n can be different for each individual message. The messages also have to include n, as well as a way to identify the source blocks that they contain. When enough of these messages have been retrieved, the client can reconstruct the original source message by resolving the XOR operations.

For an efficient distribution of the degree n as well as the set of source block that was used, pseudorandom number generators that are seeded with the same values on the server and client sides can be used [Luby et al., 1997, sec. 1.1]. For LT codes to work efficiently, the degree distribution has to be chosen correctly (this is discussed in great detail by Luby et al. [1997]). When chosen correctly, the overhead is a very small fraction of the data size ($k + o(k)$ symbols for k message symbols [Mitzenmacher, 2004]).

Raptor codes [Shokrollahi, 2006] build onto the idea of LT codes. One of the problems of LT codes is the difficulty of choosing the correct degree distribution. To avoid this problem, Raptor codes introduce the idea of a precode, where the original message is first encoded with a fixed rate erasure code before using the LT coding scheme. This avoids the necessity of every source block having to be covered by the code, which is the reason for the problems of choosing the correct degree distribution in LT codes. Decoding needs only a few more encoded blocks than the number of source blocks. The degree n as well as the source data blocks that are contained in a message have to be known to the server and to the client for each message. Once again, pseudorandom number generators can be used for this.

Typical usage scenarios of these coding schemes are applications such as multicasting or one-to-many-tcp [see Mitzenmacher, 2004], where several clients want to receive the same source data. When using a fountain code, the sender can just continue sending until all clients have received enough symbols to decode the original message. It is not necessary to re-send the individual messages that a client missed.

These coding schemes have some similarities to the presented scheme - however, they are far more complex and cannot easily be used in our chosen scenario. All the schemes assume a fixed source message; when using such a system altering data that is stored in the system would be basically impossible. Furthermore, new data that is added to the system would have no relationship to the data that is already stored in the system. In the scheme presented in this work, new data is simply XOR-ed with old data – because there is no fixed scheme in which the coded blocks have to be generated.

6.7. Contribution and differences of the presented algorithm

Now that the RLC algorithm that will be analyzed in more detail in the next chapter has been presented, this section details the contributions to the current state of the art in the research community as it was presented in Section 6.6.

One of the main points that differentiate this work against previous works is the global view on the data that is stored in the system. The usual goal of data redundancy algorithms is to secure one specific (mostly large) file against data loss. This scheme is then applied independently to all files stored in the system. This work takes another approach and tries to secure the content of a whole file system against data loss, without caring about file borders. Instead it is assumed that all data is stored in data chunks of a common length. Redundancy data is generated without paying respects to file borders. Instead, in systems having a large, diverse amount of data stored in it, it would be common for a redundancy block to be generated by using data blocks of different files that have no relationship to each other.

This is one of the key differences of the approach presented in this work to the current systems which were presented in the related work section, and necessitates a different design. Due to this holistic view of the data in the system and fact that data in a file system will change over time, using other already established coding systems like fountain-codes or hierarchical codes is not directly possible in our case. The performance of these schemes depends on characteristics of the underlying systems, which do not hold true in our scenario. One of the most important characteristics of those encoding schemes is that they grow more efficient with a growing amount of data that is secured against failure. To accomplish their high efficiency the systems assume that the secured data is unchanging. Current file systems that provide data manipulation and use a coding scheme other than replication either just replace the entire file when it is changed (e.g. Tahoe, discussed in Section 4.4.11), or store "patches" to the data in the system if the changes so not cross a certain threshold (e.g. Wuala, for details of way small changes in files are handled see Section 4.4.10). However, if bigger portions of a file are changed, those systems also re-encode the complete file which can be a very length process depending on the file size.

For our system one of the key design goals was the ability to transparently handle fluctuating data. However, having fluctuating data without a huge performance impact on data changes necessitates having relatively small chunk sizes; this problem was discussed in Section 6.3.1.

Due to this, the presented RLC algorithm is relevant for the distributed file system community and presents a novel approach that was not previously considered. The key characteristics of the presented algorithm is its suitability for distributed file systems with changing data coupled with the simplicity of the approach and a quite high data security in comparison to replication and erasure coding, when applied as necessary in our scenario. Note that this work does *not* claim that the presented algorithm is generally superior to erasure coding and/or the other coding schemes presented in Section 6.6. However, in the scenario chosen in this work, the performance of the algorithm presented in this dissertation is very good as shown by the evaluations in this and in the next chapter.

The robustness of the presented approach is shown through the extensive simulations in a realistic scenario that will be presented in the next chapter; with the right parameter choice data loss can be prevented even in environments with a massive amount of churn (as evident in the simulations using the Kademlia trace). Due to the distributed nature the system is implicitly scalable. The general viability of the presented approach is verified with the simulations in the next chapter. When nodes of the system become overloaded, the load on individual network nodes could be reduced by adding more nodes to the network and thus lowering the average number of blocks per node.

7. Real-World Comparison

In this chapter, the presented redundancy schemes are compared using real-world network traces. In contrast to the purely synthetic evaluations of Chapter 6, these traces contain information about which nodes were online at a specific point of time. Thus, the simulator can determine which data blocks were unavailable at which time.

This chapter is subdivided into three parts. Section 7.1 evaluates the coding schemes using a static approach. All data is inserted into the network at the beginning of the simulation. After that, just like in Chapter 6, no maintenance is done; the data stored in the network is left to itself. During the simulation the data loss ratio for the different redundancy coding schemes is measured and compared. Section 7.2 assesses the coding schemes using a dynamic approach where the data stored in the network is continuously refreshed by means of a redundancy maintenance algorithm. The stored data is probed regularly and lost data blocks are restored. Section 7.2 also presents the scalable distributed redundancy maintenance algorithm used in the simulations. Section 7.3 concludes the evaluation of the redundancy schemes and identifies possible future work.

7.1. Trace Presentation and Static Comparison

This section presents the characteristics of the two set of traces used in the simulations done for this chapter. The static simulation results are presented in conjunction with the traces.

Section 7.1.1 presents a Kademlia network trace with a high level of churn. Section 7.1.2 presents a trace from computers at Microsoft, which have a low level of churn. Section 7.1.3 details why these two traces were chosen and presents other available network traces.

7.1.1. The Kademlia Trace

Steiner et al. [2007b] measured and evaluated the behavior of peers in the Kademlia network (see Section 3.6.3). They published the generated network traces together with their analysis [KAD Trace]. One of their traces, called the Zonecrawl, shows the individual availability of the network peers in a region of the network over a period of slightly less than half a year. The trace has a five minute granularity. Hence, the reachable and unreachable nodes can be determined for each five minute interval. Thus, this trace can be used to assess *how long* data would have survived in the network in practice when protected by the different redundancy schemes.

Steiner et al. also generated several other network traces; however, the other traces only have a granularity of one day, which makes them much less interesting for redundancy simulations.

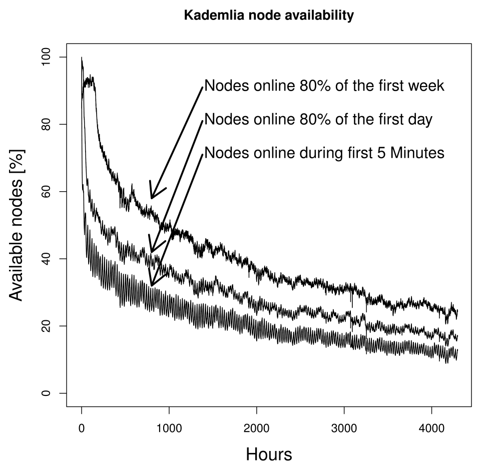


Figure 7.1.: Kademlia node availability of nodes online in the first period

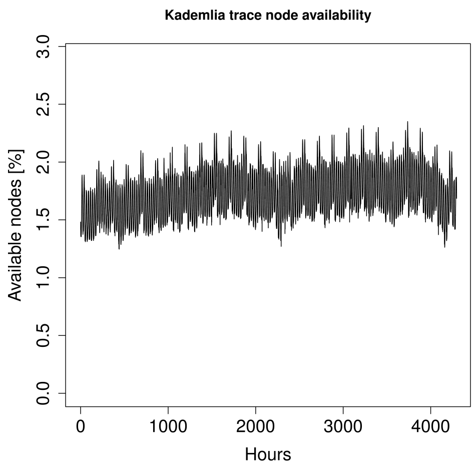


Figure 7.2.: Simultaneously online nodes in Kademlia

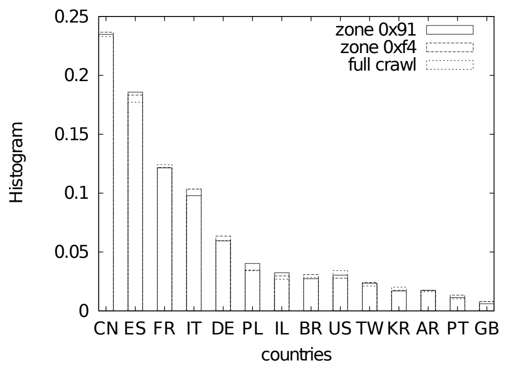


Figure 7.3.: Geographic distribution of peers [from Steiner et al., 2007a, p. 9]

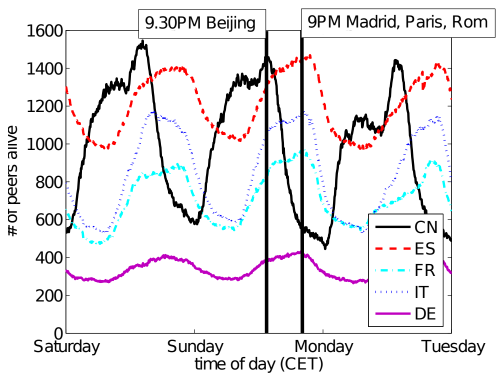


Figure 7.4.: Peers online according to country of origin [from Steiner et al., 2007a, p. 10]

The Kademlia Zonecrawl, henceforth known as *the trace*, consists of about 50 000 network snapshots. They were obtained by regularly probing all the peers of the Kademlia network whose high-order bits are $0x5b$. During the entire measurement, a total number of 400 375 different peers were encountered. 5 670 of these peers were found to be available at the time of the first probe. 2 880 nodes were available at 80% of the probes during the first day. 1 377 nodes were available at 80% of the probes during the first week. Figure 7.1 shows how many of these initially available peers were still available over the course of the measurement. Nodes that have a high availability at the beginning of the measurement continue to be more available later on. However, the node availability in the Kademlia network is generally quite low. Only approximately 2% of the total encountered nodes are online concurrently at any time (cf. Figure 7.2). The maximum number of simultaneous online nodes is 9 413 (2.4%), the minimal number is 4 991 (1.2%).

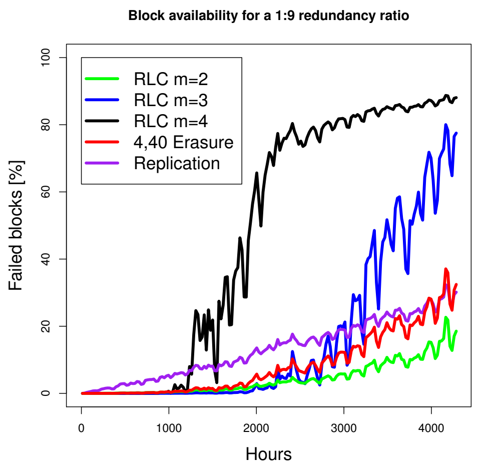


Figure 7.5.: Unavailable blocks (all nodes)

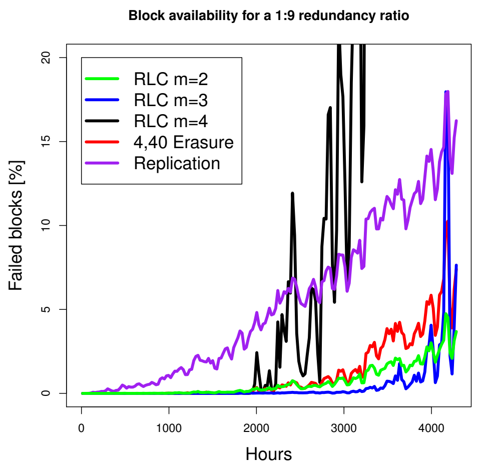


Figure 7.6.: Unavailable blocks (day nodes)

Figure 7.3 on page 116 shows the country distribution nodes in the network. Most of the nodes are from China followed by Spain and France. The node availability correlates with the time of day in their respective location. This is depicted in Figure 7.4.

The different redundancy algorithms are evaluated with the three sets of Kademlia network traces presented in Figure 7.1. The first set, henceforth called the *all nodes set*, consists of the nodes that are online at the time of the first probe, the second set, henceforth called the *day nodes set*, consists of the nodes with an uptime of at least 80% during the first day, and the third set, henceforth called *week nodes set*, consists of the nodes with an uptime of at least 80% during the first week.

These three sets of peers are used to assess the durability of data when it has been protected by the different redundancy schemes. The approach is as follows:

10 000 data blocks and 90 000 redundancy blocks are equally distributed among these nodes. This high level of redundancy is required because of the high churn in the network. If a lower level of redundancy is chosen, all the schemes fail after a short amount of time. The chosen level is just at the edge where data restoration is possible in most cases but fails sometimes. This is the most interesting scenario because it shows how the algorithms behave in extreme cases.

The simulation runs show how many of the data blocks cannot be recovered for each of the 5 minute probing intervals in the trace.

Figures 7.5 to 7.7 show the results for the simulations for replication, erasure coding, and RLC.

At the beginning – as expected from the synthetic data simulation – random linear coding outperforms both replication and erasure coding. When the number of unavailable peers exceeds a certain threshold, the availability of data that has been protected by RLC degrades very rapidly.

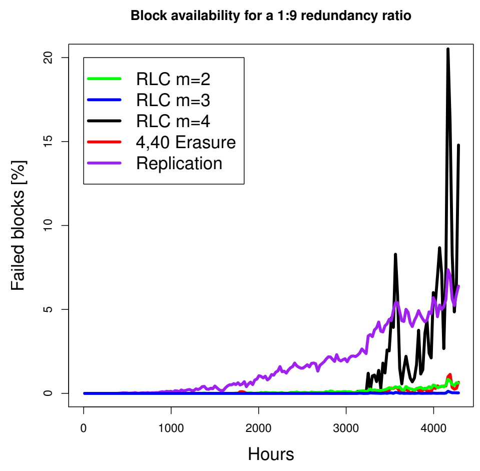


Figure 7.7.: Unavailable blocks (week nodes)

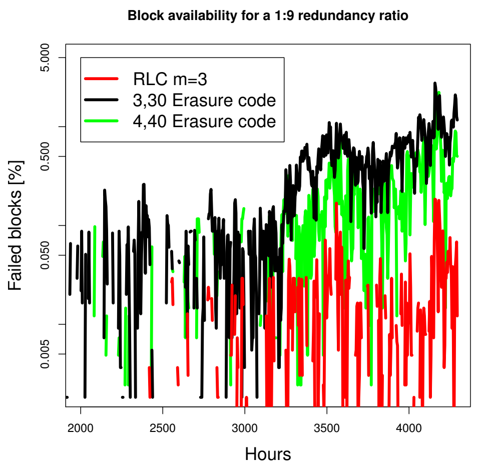


Figure 7.8.: Unavailable blocks (week nodes)

In the all nodes set, RLC with $m = 4$ loses the data after about 1 000 hours. With $m = 3$ the sudden degradation sets in after about 3 000 hours. RLC with $m = 2$, erasure coding and replication do not show such a sudden degradation, but continuously lose the data.

In the day nodes set, which contains the peers that were available for at least 80% of the first day, the sudden degradation of the block availability sets in later. RLC with $m = 4$ loses the data only after about 2 000 to 3 000 hours. At that time, replication has already lost about 5% of its data. RLC with $m = 3$ loses the data only after about 4 000 hours. At that time, replication has already lost about 15% of its data.

In the week nodes set with the peers that were available at least 80% of the first week, the difference to replication is even more pronounced as shown in Figure 7.7. Figure 7.8 shows part of Figure 7.7 in more detail. It compares random linear coding with two different erasure coding strategies. RLC provides better results than erasure coding. Note that the plot is in log-scale.

The hybrid coding scheme performed a little bit worse than the equivalent erasure coding scheme (cf. Figure 7.9 on the facing page). This was expected because the hybrid scheme has more original data blocks at the expense of erasure coding blocks which offer a better protection. The data transfer reduction anticipated by Rodrigues and Liskov [2005] (see Section 6.6) in a dynamic scenario is evaluated in Section 7.2.

The simulation results show that random linear coding can provide excellent availability and durability of data in peer-to-peer file systems. Unlike replication, it prevents data loss over a comparatively long time. When RLC can no longer protect the data it degrades quite rapidly. These characteristics fit the need for a distributed file system because file systems should not lose data at all. When data is lost it usually does not matter whether 20% or 50% are lost, the stored data will be unreadable anyway.

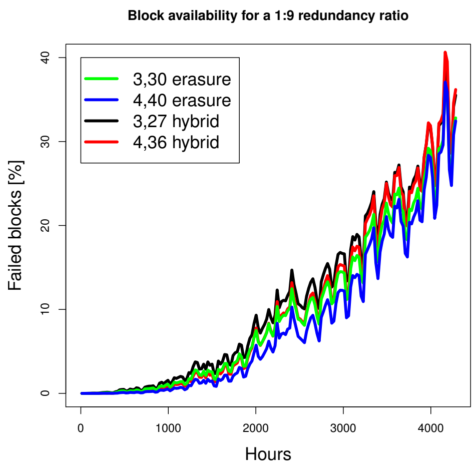


Figure 7.9.: Hybrid erasure comparison (all nodes)

At the time RLC begins to lose data blocks, both replication and erasure coding have already lost a significant number of data blocks. The exact behavior of RLC depends on the parameter m. $m = 3$ has been found to be a good choice.

7.1.2. The Microsoft Trace

The Kademlia trace evaluated in the previous section has a very high level of churn. However, it is also interesting to examine the performance of the schemes in a more stable environment.

For this reason, the simulations were repeated using a second network trace, which was generated in a cooperate environment. The trace was created by Bolosky et al. [2000] at Microsoft by pinging 64 619 PCs for about one month.

The general machine uptime is much higher for this trace as compared to the Kademlia trace. Figure 7.10 on the next page shows the percentage of available peers.

It is difficult to compare the redundancy schemes in high-uptime environments because even with a quite low redundancy level, all the schemes manage to retain the data.

The redundancy factor used for the simulations is 1, i.e. exactly the same amount of data and redundancy blocks are stored in the network. A lower redundancy factor is not sensible for replication; otherwise some of the blocks would not be replicated at all. For erasure coding and RLC redundancy, factors less than 1 would be possible. For example, for RLC with $m = 2$ and a redundancy factor of 0.5, 2 data blocks would be protected by one redundancy block. For hybrid coding, the redundancy factor has to be greater than 1. A redundancy factor of 1 would lead to worse results than replication – it would result in an original data block and m (m, m) erasure coded blocks ($m = n$ in this case). Hence, the hybrid scheme has not been considered in this setting.

Figure 7.11 shows the results of a simulation with 150 000 original data blocks and 150 000 redundancy blocks. Replication is worst, followed by erasure coding (with $n = 6$, $m = 3$) and RLC (with $m = 3$). Note that erasure coding can once again perform

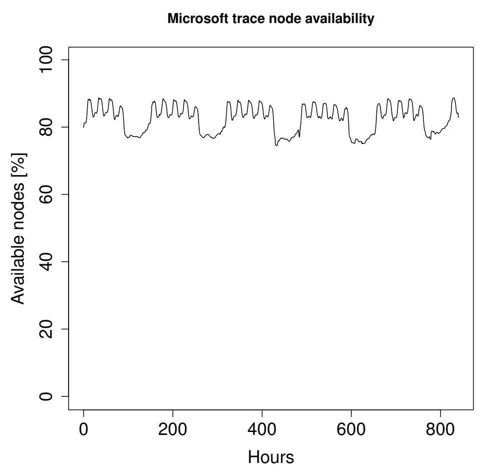


Figure 7.10.: Node availability (Microsoft trace)

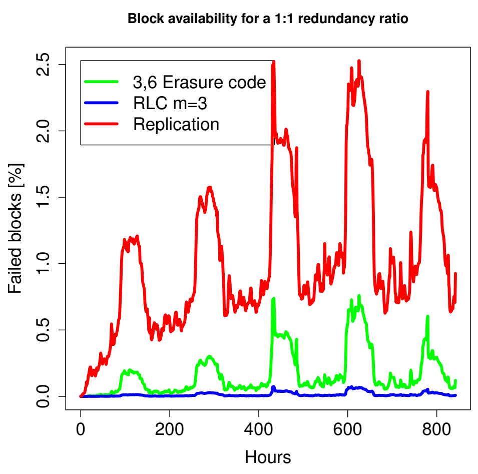


Figure 7.11.: Unavailable blocks (Microsoft trace)

arbitrarily good for large enough values of n; however, as already mentioned, a large value of n makes updating more difficult. Also note that the maximum data loss is only about 2.5%.

7.1.3. Other Availability Traces

The previous two sections covered a scenario with a very low general node availability and a scenario with a high general node availability. It would be interesting to confirm the results using traces from an even more diverse range of networks.

However, these traces are not easy to come by. To the best of my knowledge, the two presented traces are the most suitable traces which are available. A number of other network traces is available at [AVT Traces].

All of these traces have certain disadvantages in comparison to the chosen trace. For example, the PlanetLab All Pairs Ping[1] contained in the repository only has the availability information of 720 nodes. Tang et al. [2007] also state that the node availability in the trace is generally greater than 80% and thus similar to the Microsoft trace.

Bakkaloglu et al. [2002] performed a trace of the availability of 129 websites with a 10-minute granularity. The data of 99 of these nodes is usable; the other 30 were excluded due to protocol incompatibilities and other problems. 99 nodes is a very small number for the scenario presented in this paper. Additionally, the trace contains periods where the source network apparently had connectivity issues and thus nearly all nodes were unavailable. This trace is clearly unsuitable for this thesis.

Pang et al. [2004] performed a probe of 62 201 DNS servers. This trace is not suitable for this thesis because the general node availability is too high – the mean availability is 97.85% with a standard deviation of 9.39%. This level of availability is even better than

[1]The website containing the raw data of the PlanetLab All Pairs Ping is no longer available.

the Microsoft trace. Furthermore, the probing time is only three weeks with a relatively coarse grained resolution of about one hour.

Guha et al. present a trace of 4 000 Skype nodes with a granularity of 30 minutes. This node number is also low in comparison to the chosen traces; in addition, the trace contains several artifacts like a one-day outage of the measurement and several spikes due to network problems.

Zhang et al. [2010] present the peer-to-peer trace archive of the Delft University of Technology. The traces are available at [P2PTA]. Traces in this archive generally cover a very small amount of time (less than 1 month). Furthermore, the focus of the traces is not on node availability, but on other information such as current traffic, download progress, size of the shared files, etc.

[AVT Traces] contains a list of other available network traces; those traces were not chosen for similar reasons (either because they cover a very small amount of nodes, a very short amount of time or do not include fine granular node availability information).

7.2. Dynamic Real-world Comparison

In this section, the redundancy scheme analysis of Section 7.1 is extended to a dynamic setting.

So far, the simulations only determined what happens to the data when it is put it into the system once: *Can the data still be retrieved after a given number of peers have failed?* In practice, most systems will try to prevent data loss by continuously checking and refreshing the data stored within the system.

This second set of simulations evaluates the different redundancy schemes using a probing algorithm that ensures data availability continuously. While doing so, the associated traffic costs are logged. As already discussed in Section 6.1 on page 93, a distinction is made between the *probing traffic* and the *block transfer traffic*. Both parameters comprise all the required traffic, i.e. all the required request and response messages.

Section 7.2.1 presents the redundancy maintenance algorithm that was used in the simulations. Sections 7.2.2 to 7.2.5 present the exact way the maintenance scheme works for replication, erasure coding, the hybrid scheme, and RLC respectively. Section 7.2.6 concludes this part of the thesis with an evaluation of the different redundancy schemes using the maintenance algorithm. Both traces presented in Section 7.1 are used in the evaluation. Possible extensions to the work done in the thesis will be discussed in the conclusion in Section 7.3.

7.2.1. General Probing Process

It is assumed that all peers that store data actively participate in the proposed protocol, i.e. the peers can mutually probe each other for the availability of the data blocks. In contrast, if storage nodes that do not actively communicate among each other are to be used, the active peers would have to regularly probe the stored blocks. This option is briefly discussed in Section 7.3 on page 131.

The probing peers regularly check if the desired redundancy level is still maintained. I propose an algorithm that uses *exponential back-off probing*: Consider a data block b that has been stored in the system at time $t = 0$. The algorithm checks the data block after $t = 1$ for the first time. If the data block is available, the next check is scheduled for $t = 2$. If it is once again available, the next checks are scheduled for $t = 4$, $t = 8$, $t = 16$, and so on. If the data block is missing during one of these checks, e.g. at $t = 16$, the system tries to reconstruct it using the redundancy blocks. If it can do so, the next probe will be scheduled at $t = 17$, then at $t = 19$, $t = 23$, and so on.

The rationale for that design is the increasing trust that the system has in the nodes on which the data block is residing: The longer a peer has been available, the less likely it will fail in the next time period. In other words, peer availability is heavy-tailed (cf. Figure 7.1 on page 116): a large number of peers have rapidly fluctuating activity; a small number of peers are available for a very long time [see Maymounkov and Mazières, 2002; Godfrey et al., 2006; Tati and Voelker, 2006].

Since it is assumed that all the storing peers are active, the peers can mutually probe each other for the data blocks. To limit the number of probes to a minimum, the concept of *redundancy groups* is introduced.

Definition 7.1. *A redundancy group is a set of blocks that depend on each other (as within e.g. an erasure code group) or that are copies of each other (as with replication). All nodes (having a block) in a redundancy group have to probe each other.*

To minimize the traffic, it is assumed that only one node in a redundancy group probes for the presence of all other blocks. In a real network, this behavior would be established by spreading the probes out over a sufficiently large time interval so that – with high probability – only one node storing a block of a given redundancy group probes the other nodes storing blocks of that group. When a node notices that it has been probed for a block, it assumes that node to be responsible for the redundancy group during the current epoch, i.e. it has to probe all other nodes of the group and potentially restore the redundancy.

Restored blocks are sent uniform-random to nodes in the network. Note that this strategy performs very good, better than many node preselection strategies [Godfrey et al., 2006].

7.2.2. Probing Replication

In the chosen setting, when using a redundancy factor of r, $r + 1$ copies of each data block are stored on randomly chosen different nodes. As long as no more than r of them fail, the block remains available.

The probing algorithm tries to ensure that $r + 1$ replicas remain available. At the scheduled probing time, one of the nodes sends a probe to its peers. If one of the replicas is found unavailable, the probing node creates another replica and stores it on a random peer in the system.

Algorithm 2 on the next page shows the probing algorithm in more detail.

Algorithm 2: Replication probing

```
Input:
  $B$: set of alive data blocks
  $r$: number of replicas of each block
  $e$: current epoch
1  function replicationProbing($B, r, e$)
2  begin
3      for $b \in B$ do
           // check if $b$ has to be probed this epoch
4          if $b$ .nextProbingEpoch $> e$ then
5              next
6          probe($b$.replicaSet)
7          $A \leftarrow \{b' \in b$.replicaSet: $b'$ is alive$\}$
8          if $|A| < r$ then // a replica failed
9              for $|A|$ to $r$ do
10                 distributeOnRandomNode($b$)
               // a block failed; reset exp.  backoff probing
11             $\forall b' \in \{b\} \cup A$: resetExponentialBackoff($b', e$)
12         else
               // set next probing epochs with exponential backoff
13             $\forall b' \in \{b\} \cup A$: setEpochWithExponentialBackoff($b', e$)
14         $B \leftarrow B \backslash A$ // all blocks in $A$ have been probed
15 function setEpochWithExponentialBackoff($b, e$)
16 begin
17     $b$.nextProbingEpoch $\leftarrow e + b$.probingStepSize
18     $b$.probingStepSize $\leftarrow b$.probingStepSize $\cdot 2$
19 function resetExponentialBackoff($b, e$)
20 begin
21     $b$.nextProbingEpoch $\leftarrow e + 1$
22     $b$.probingStepSize $\leftarrow 1$
```

7.2.3. Probing Erasure Codes

In the case of erasure coding, each of the peers storing a fragment of a redundancy group probes the availability of all other fragments. Again, the first peer that happens to send the probes handles that epoch. Missing fragments are restored by the probing peer, which will fetch the necessary number of other fragments and push the missing fragment to another peer. Algorithm 3 on the following page shows this in more detail.

7.2.4. Probing Hybrid Codes

In the hybrid probing scheme, there are two different kinds of blocks; the original data blocks as well as the erasure coded hybrid blocks.

Algorithm 3: Erasure code probing

```
   Input:
     $B$: set of alive data blocks
     $n$: number of fragments required to be able to reconstruct data
     $m$: total number of fragments present
     $e$: current epoch
1  function erasureProbing($B, n, m, e$)
2  begin
3      for $b \in B$ do
           // check if $b$ has to be probed this epoch
4          if $b.nextProbingEpoch > e$ then
5              next
6          probe($b$.erasureSet)
7          $A \leftarrow \{b' \in b.\text{erasureGroup}: b' \text{ is alive}\}$
8          $B \leftarrow B \setminus A$ // do not handle blocks again
9          if $|A| < m$ then // a fragment failed
10             if $|A| < n$ then // cannot reconstruct
11                 $\forall b' \in \{b\} \cup A$: $b'$.nextProbingEpoch $\leftarrow \infty$ // disable probing
12             else
13                 fetch blocks from $A$ until $n$ blocks are on node
14                 reconstructed $\leftarrow$ reconstructed missing blocks
15                 $\forall b' \in$ reconstructed: distributeOnRandomNode($b'$)
16                 $\forall b' \in \{b\} \cup A$: resetExponentialBackoff($b', e$)
17         else
18             $\forall b' \in \{b\} \cup A$: setEpochWithExponentialBackoff($b', e$)
```

The probing scheme is detailed in Algorithm 4 on the next page. The peer with the original data blocks only probes the presence of the erasure coded blocks. If an erasure coded block is missing, it can be reconstructed on the fly using the original data blocks that are already present on the probing node.

The erasure coded blocks probe the peer with the data blocks. (In this work, it is assumed that the blocks do the probing where – of course – the respective peers have to execute the algorithm.) If the peer and hence the data blocks are lost, the entire bundle is reconstructed by the probing node. The node fetches as many erasure blocks as needed for the reconstruction, and then it reconstructs the data blocks. To minimize the traffic, it only sends the erasure block that it previously stored to another node after the reconstruction is complete. It is now the peer containing all the original data blocks of the group.

7.2.5. Probing Random Linear Codes

As in the hybrid scheme, there are two different types of blocks: redundancy blocks and data blocks. The probing scheme is also similar. Nodes with data blocks probe the availability of all their associated redundancy blocks. If a redundancy block is missing, the node fetches all other blocks that were used to generate the redundancy block. The redundancy block is regenerated and sent to a random node in the network.

Note that in random linear coding, the epoch in which the probing occurs is no longer synced in the whole redundancy group. That means that data blocks probe the redundancy blocks in other epochs than the redundancy blocks probe their data blocks. The reason for this is that redundancy blocks connect data blocks, which are connected to more redundancy blocks. The probing times cannot be synchronized because the connected component could span the entirety of all blocks in the network. When a data block fails, the probing interval for the redundancy blocks is reset – the node of the data block was unreliable, thus the data block has to be probed more often by the redundancy blocks

Nodes with redundancy blocks probe the availability of their associated data blocks. If a data block is missing, the node likewise fetches all other data blocks that were used to generate it, regenerates the lost block and sends it to a random node. Because a data block fails, it makes no sense to increase the checking frequency with which the redundancy blocks are checked. Hence, the probing interval of the new data block is set to the maximum probing interval of the data blocks that are intertwined with the redundancy blocks that reconstruct the data block. The probing interval of the redundancy blocks probing the new data block is reset.

The scheme is detailed in Algorithm 5 on page 127.

7.2.6. Evaluation

In this section, the simulation results of the dynamic evaluation of the schemes mentioned above are shown, starting with the simulations based on the Kademlia network trace. All Kademlia simulations were run using a redundancy factor of 3. In the first epoch, $t = 0$, 5 000 data blocks and 15 000 redundancy blocks are distributed randomly and equally

Algorithm 4: Hybrid code probing

```
Input:
  $B$: set of alive data blocks
  $E$: set of alive erasure blocks
  $n, m, e$: see Algorithm Algorithm 3 on page 124
function hybridProbing($B, E, n, m, e$)
begin
    for $b \in B$ do
        // check if $b$ has to be probed this epoch
        if $b.nextProbingEpoch > e$ then
            next
        reconstructed $\leftarrow \emptyset$
        // if $b$ is alive, all data blocks of the group are alive
        for $b_e \in b$.erasureSet do
            probe($b_e$)
            if $b_e$ is !alive then
                reconstruct $b_e$ using $b$.originalBlocksSet
                distributeOnRandomNode($b_e$)
                reconstructed $\leftarrow$ reconstructed $\cup \{b_e\}$
        $R \leftarrow$ ($b$.originalBlocksSet $\cup$ $b$.erasureSet)
        if reconstructed $= \emptyset$ then
            $\forall b' \in R$: setEpochWithExponentialBackoff($b', e$)
        else
            $\forall b' \in R$: resetExponentialBackoff($b', e$)
        $B \leftarrow B \backslash R$
        $E \leftarrow E \backslash R$
    for $b \in E$ do
        // check if $b$ has to be probed this epoch
        if $b.nextProbingEpoch > e$ then
            next
        // if $b$ is probed here, the data blocks of $b$ were lost
        probe($b$.erasureSet)
        $R \leftarrow \{b' \in b$.erasureSet: $b'$ is alive$\}$
        if $|R| \geq n$ then
            fetch required number of blocks from $R$
            // reconstruct original block set
            $\forall b' \in b$.originalBlocksSet: reconstruct($b'$)
            $\forall b' \in R \cup b$.originalBlocksSet: resetExponentialBackoff($b', e$)
            if $|R| < m$ then
                $R_r \leftarrow$ reconstruct missing erasure fragments
                $\forall b' \in R_r$: distributeOnRandomNode($b'$)
            // now this node has the data blocks - move erasure blocks away
            distributeOnRandomNode($b$) // copy erasure block to other node
            deleteFromLocalNode($R$)
        else
            // no reconstruction possible
            $\forall b' \in (R \cup b$.erasureSet): $b'$.nextProbingEpoch $\leftarrow \infty$
```

Algorithm 5: RLC probing

```
Input:
  $B$: set of alive data blocks
  $R$: set of alive redundancy
  $e$: current epoch
function RLCProbing($B, R, e$)
begin
    for $b \in B$ do
        // check if $b$ has to be probed this epoch
        if $b.nextProbingEpoch > e$ then
            next
        reconstructed $\leftarrow \emptyset$
        for $r \in b$.redundancyBlocksSet do
            probe($r$)
            if $r$ is alive then
                setEpochWithExponentialBackoff($b, e$)
            else
                $A \leftarrow \{b' \in b$.dataBlocksSet: $b'$ is alive$\}$
                if $|A| = |b$.dataBlocksSet$|$ then
                    fetch($A$)
                    $b_r \leftarrow$ reconstruct missing redundancy block
                    $\forall b' \in b$.dataBlocksSet: resetExponentialBackoff($b', e$)
                    // Defer execution of the following lines until all blocks
                    // in $r$.redundancyBlocksSet have been probed.
                    $R \leftarrow \{b' \in b$.redundancyBlocksSet: $b'$ is alive$\}$
                    $b_r$.nextProbingEpoch $\leftarrow$ max($R$.nextProbingEpoch)
                    $b_r$.probingStepSize $\leftarrow$ max($R$.probingStepSize)
                    distributeOnRandomNode($b_r$)
                else
                    // $r$ cannot be reconstructed
        if reconstructed $= \emptyset$ then
            setEpochWithExponentialBackoff($b$)
    for $r \in R$ do
        $A \leftarrow \{b' \in b$.dataBlocksSet: $b'$ is alive$\}$
        if $|A| = |b$.dataBlocksSet$|$ then
            setEpochWithExponentialBackoff($r, e$)
        else if $|A| = |b$.dataBlocksSet$| - 1$ then
            fetch($A$)
            $n \leftarrow$ reconstruct missing data block with $A$
            $N_r \leftarrow \{b' : (b'$ is in redundancy block of $n)\}$
            distributeOnRandomNode($n$)
            $\forall b' \in N_r$: resetExponentialBackoff($b', e$)
            $n$.nextProbingEpoch $\leftarrow$ max($A$.nextProbingEpoch)
            $n$.probingStepSize $\leftarrow$ max($A$.probingStepSize)
        else
            $r$.nextProbingEpoch $\leftarrow \infty$
```

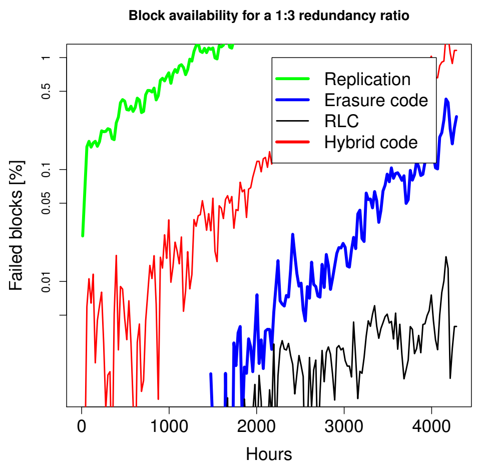


Figure 7.12.: Unavailable blocks (Kademlia trace)

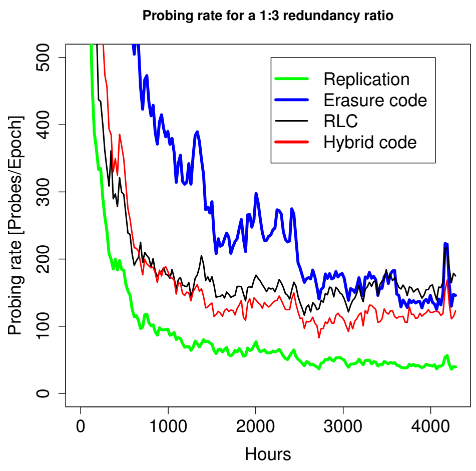


Figure 7.13.: Block probing rate (Kademlia trace)

onto the nodes that are online at $t = 0$. Note that due to the data maintenance algorithm, a much lower redundancy factor of 3 is used. In Section 7.1 the redundancy factor was 9.

For the erasure code algorithm and the hybrid algorithm, the coded redundancy group is generated from three data original data blocks. Likewise, for the RLC algorithm, each redundancy block is generated from three data blocks.

Figure 7.12 shows the block availability for the four schemes. Note that the y-axis is in logarithmic scale. With the chosen settings, replication has big problems to maintain the availability of the data. Hybrid coding is already about one order of magnitude better, but it still has an unacceptably high block loss ratio. Erasure coding performs one order of magnitude better than hybrid coding, but RLC is superior to all other schemes.

Figure 7.13 shows the amount of probing traffic for the four schemes. The probing rate for replication is the lowest, followed by RLC and the hybrid scheme. Initially, erasure coding has the highest rate, before it settles to the level of RLC and the hybrid scheme. All three schemes have a higher message overhead than replication because the blocks are much more intertwined. However, the message overhead for RLC is still significantly lower than for erasure coding for most of the time. Only in the last quarter of the simulation is the probing message exchange similar for both schemes. One of the reasons for the decreasing message overhead of erasure coding can be found in the number of failed blocks. When a redundancy group is lost, the algorithm stops all probing attempts for the redundancy group. Thus, when the number of lost blocks rises, the maintenance traffic decreases.

Figure 7.14 on the next page shows the number of block transmissions. The result is very similar to that in Figure 7.13. At the beginning, RLC has a slight advantage over the hybrid scheme.

Figure 7.15 shows the average redundancy level, i.e. the number of stored blocks per original data block. Blocks on dead nodes do not count towards the figure, but blocks

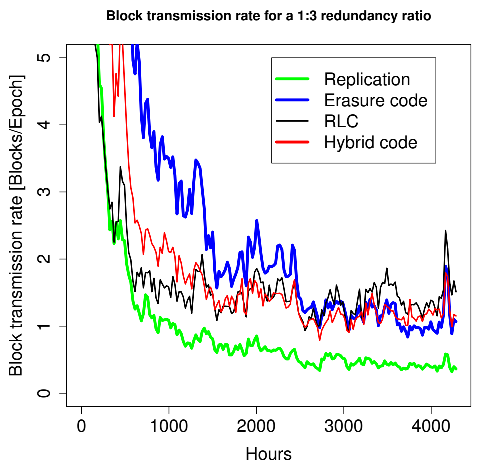


Figure 7.14.: Block transmission rate (Kademlia trace)

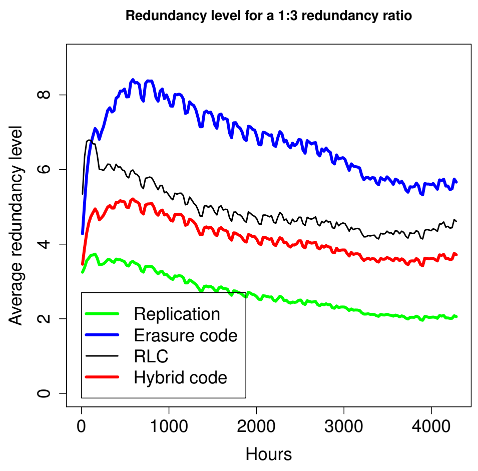


Figure 7.15.: Average redundancy level (Kademlia trace)

on returning nodes are reflected in the figure. All schemes start with 3.53 blocks stored per original data block (20 000 initial blocks on 5 670 peers available in the first epoch). The number of stored blocks rises quickly because blocks that had been temporarily unavailable, and thus have been replaced reappear. Replication incurs the lowest storage overhead, followed by the hybrid scheme and RLC. Erasure coding also needs the most storage. This is due to the way block caching is handled in the simulation. Erasure coding needs more block transfers than the other encodings. Nodes that retrieve a block from another node, for example, to reconstruct a failed block, will cache the retrieved block in case they need it again. If they discarded them after reconstruction, the storage overhead of erasure coding would be lower at the expense of higher transmission traffic and a slightly higher loss rate.

Hence, erasure coding, hybrid coding, and RLC are all viable alternatives in the setting of this thesis, with RLC having a slight advantage in all chosen performance metrics. The reliability of erasure coding could be increased by choosing a larger value for n; however, this would increase the message overhead even further.

The Microsoft trace data was also used for simulations with the maintenance algorithm. As mentioned in Section 7.1.2, the availability for the trace is very high. Even in a static setting, a redundancy factor of 1 is enough. Hence, in the following simulations a redundancy factor of 0.5 was chosen – with 150000 data blocks and 75000 redundancy blocks. The simulation was only performed for erasure coding and RLC. As discussed in Section 7.1.2, both replication and hybrid coding are unsuitable for such settings.

Figures 7.16 and 7.17 show the lost blocks and the required probing traffic for a (4,6) erasure code and RLC with $m = 3$ and $m = 4$. The probing traffic is nearly identical for all schemes. The same holds true for the block transfer traffic; the figure was omitted for this reason. RLC performance is a bit surprising in this case: with $m = 4$ RLC performs significantly better than the equivalent (4,6) erasure code; with $m = 3$, RLC performs a

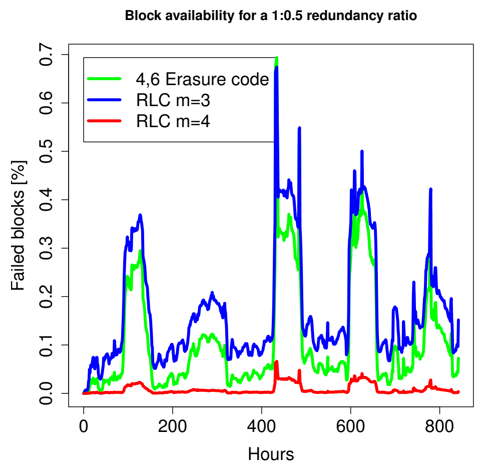


Figure 7.16.: Unavailable blocks (Microsoft trace)

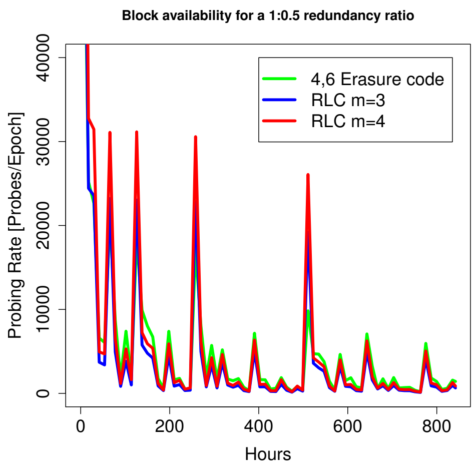


Figure 7.17.: Block probing rate (Microsoft trace)

little bit worse.

This is in contrast to the previous simulations, where $m = 3$ usually performed better – at least at the beginning of the simulation. The reason for this lies in the extremely low redundancy factor in this simulation. A very important cause for the very good performance of RLC lies in the fact that it generates chains of blocks that depend on each other. This was discussed in Section 6.5 and depicted in Figure 6.9 on page 102.

In this extremely low redundancy case, the chain building is severely limited. When using $m = 2$, each data block has *exactly* one redundancy block. There are *no chains at all.* This case is not depicted in Figures 7.16 and 7.17 but performs a little bit worse than $m = 3$ When using $m = 3$, one half of the blocks in the simulation have two redundancy blocks and the other half have one redundancy block. Hence, many blocks have again no chains at all. Blocks that are present in a chain have a *very short chain length.* When using $m = 4$, each data block has two redundancy blocks – and a *potentially long chains length.*

Chains are very important for the reconstruction of data using random linear coding. When a failed block is not present in a chain, it is lost when either the redundancy block or another data block in the redundancy group fails. But when each data block has two redundancy blocks, there is a chance that is can be recovered when another data block in the redundancy group has failed – if this data block is recoverable via its second redundancy block.

The importance of chaining is depicted in two more figures: Figure 7.18 on the next page shows the block loss reasons for $m = 3$ and Figure 7.19 for $m = 4$. With $m = 4$ the amount of blocks that is lost because a data block is missing is drastically lower than for $m = 3$. However, this special case only applies with a very low number of redundancy blocks.

In conclusion, random linear coding has proven to be superior to the other redundancy

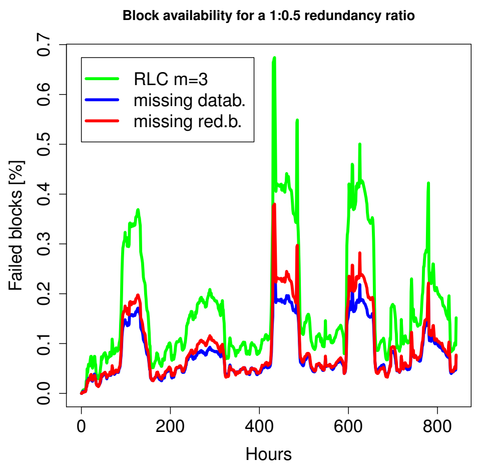


Figure 7.18.: Block loss reasons for $m = 3$ (Microsoft trace)

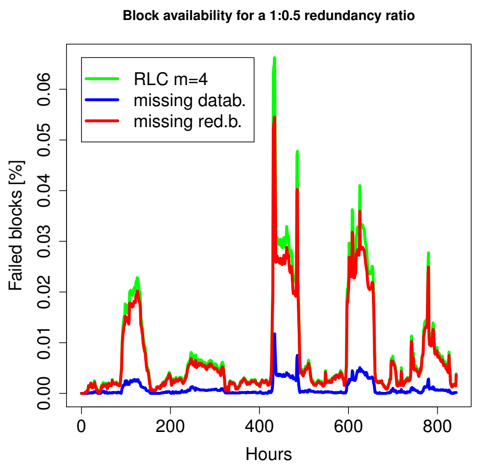


Figure 7.19.: Block loss reasons for $m = 4$ (Microsoft trace)

schemes in the chosen environment. The coding schemes are used in a way that only a small number of blocks have to be updated when the content stored in the file system changes. This puts erasure coding at a serious disadvantage. However, this choice is reasonable – for example, Tahoe used very similar parameters in their commercial service (see Section 4.4.11 on page 74). Rewriting big chunks of data for small changes is not sustainable for a random access file system.

Note that, in the scenarios as presented in the simulations, RLC usually needs to rewrite a much smaller amount of blocks when data in the network changes. For example, consider the last simulation scenario with a (4,6) erasure code and $m = 4$. When a single block changes, 6 data blocks have to be rewritten when using erasure coding. However, when using RLC only the data block and two redundancy blocks have to be rewritten. RLC should also perform better when data is read from the system because in contrast to erasure coding, the original data blocks do not have to be decoded after they have been retrieved from the system.

7.3. Future work

This thesis assumes that peers are active protocol participants. If the storage nodes do not actively participate in the protocol and only store the blocks, like, for example, in a DHT, other peers would have to probe the availability of the stored blocks. In this case, the peers inserting the data would have to check the availability of the data continuously. Another possibility would be that the availability is checked by a chosen subset of long-lived nodes in the system that can be trusted to be available with high probability. These variants could be interesting in special cases and could be evaluated in future work.

Another assumption of the current simulations is that all data is inserted by nodes that are not a part of the network. As mentioned, the simulated scenario is equivalent to the usage scenario of current cloud storage schemes. Future simulations could be modified in a way that the individual network nodes generate and insert the data into the network themselves. Unfortunately, running such simulations is quite complex. To simulate this scenario in a realistic way, it would be necessary to have file access traces of big network file systems. Because the actual file system usage is highly dependent on the environment, having access to a number of traces with different characteristics would be desirable. The characteristics of the network traces also have to fit the characteristics of the file access traces, for example, the number of nodes, and the duration of the traces have to be similar. Finding adequate network traces could be very difficult – probably they would have to be generated first. Thus this study is left to future work.

Another design alternative concerns the redundancy restoration method. This thesis assumes that the exact redundancy block that has been lost is reconstructed. Instead of doing so, for RLC, one could instead generate a new redundancy block combining the affected data blocks with other blocks. This would reduce the block fetching traffic because the second data block does not have to be transferred over the network. Moreover, it spreads the redundancy further because, if the lost block re-appears, the two redundancy blocks provide independent information. Evaluating this scenario requires that the individual nodes store foreign data. Thus, for this to work, a simulation including file access traces is required because only file accesses will distribute the necessary blocks throughout the network in a realistic way. The study of this variants is also left to future work.

Some authors propose using different redundancy algorithms and/or redundancy levels depending on the importance of specific files [see e.g. Abd-El-Malek et al., 2005; Bhagwan et al., 2004]. They conclude that the right choice of parameters for specific files can have a large impact on file system performance. While setting the importance of files usually requires manual user intervention, such a feature could be desirable in some cases. The impact of adding this feature could be explored in future works.

Cox and Noble [2003] discuss how to enforce fairness in peer-to-peer storage systems. This thesis assumes that the network is composed of well-behaving nodes. For future practical implementations, the behavior of the proposed algorithms in the face of malicious nodes should be analyzed.

Part III

Access Permissions

8. Access Permissions for Distributed File Systems

As discussed in Chapters 1 and 4, most of today's decentralized file systems lack a full-fledged permission system. Many file system designs do not support permissions at all. Those that do support them lack important features, such as the support of user groups, or have serious scalability issues.

This chapter introduces a Unix-like permission system for fully decentralized file systems. The permissions are enforced using cryptographic algorithms. Where possible, the system utilizes fast symmetric cryptographic primitives. It works on any kind of block oriented distributed storage system. Stored data is protected from rollback attacks and outside modifications.

The architecture of the permission system consists of several interdependent parts. The basic design is introduced briefly in this chapter before being examined in more detail in the following chapters.

The remainder of this chapter is structured as follows: Section 8.1 details the basic security assumptions. It describes which parts of the network are trusted and what kind of adversary the system can handle. It also lists the desired security properties. Section 8.2 presents the permission system features. Section 8.3 introduces the central components of the permission system design. Section 8.4 briefly summarizes the data integrity protection layer. Section 8.5 describes the architecture of the cryptographic permission system. Section 8.6 lists the cryptographic keys that are required for the permission system. Finally, Section 8.7 summarizes the related work and compares this approach to permission systems used in other file systems.

8.1. Security Assumptions and Goals

This section first defines the security assumptions of this thesis followed by a description of the desired security properties.

Security assumptions

Trust in own machine: It is assumed that a user can trust the own machine. This implies trust in the system administrator if the machine is not administered by the user. Scenarios where an attacker reads the cryptographic key material from volatile memory are not part of the threat scenario. In contrast, non-volatile local memory has not to be trusted; all file system data committed to disc storage is encrypted. Only the key material of the user has to be kept in a safe location, either by encrypting it using a secret pass phrase or by storing it on a trusted medium like a USB-stick or a smart card.

Untrusted nodes: It is assumed that all network nodes except the local machine are untrusted. All information returned by other network nodes has to be verified. There are no nodes that have a special amount of trust placed in them; all nodes in the network are equal. Thus, access rights enforcement must not depend on other nodes. All access rules have to be enforced by cryptography, not by node-side policies.

Trusted file system superuser: This thesis presents a closed file system design where, just like in traditional network file systems, only authorized users may access the file system. Hence, the system requires centralized user management. The entity that may add or remove users from the distributed file system is the file system superuser. The superuser generates the key material for the individual nodes. It is assumed that the superuser operates correctly and cannot be corrupted.

Required security features

Data integrity: The file system may only return valid data to the user. The file system must not return data inserted or modified by unauthorized third parties. Such modifications must be prevented using cryptographic means. Users with access to the system must only be able to modify files to which they have write permissions. Rollback attacks must be prevented to the extent possible in a distributed environment without online trusted servers.

Data confidentiality: Users must not be able to read data to which they do not have adequate permissions. Due to the untrusted nature of network nodes, file permissions must be enforced using cryptography. A client must not be able to decrypt any material for which the user has no read permissions.

No on-line third parties: The scheme shall work without requiring any third parties to be online and without requiring specific other nodes to be online. This includes the trusted superuser. Thus, while the system has a virtual centralized entity, this entity only has to be online when performing administrative tasks.

User management: Adding users to the system may require secure out-of-band communication between the added users and the trusted file system superuser. This exchange is necessary to establish trust between the entities and to exchange the necessary key material. Other than that the system must not require out-of-band communication. Adding a user must not require out-of-band-communication to the rest of the network. The file system superuser must be able to revoke users from the system without the need for out-of-band communication.

Tracing of misbehaving users: In a fully decentralized system without server-verified writes, it is not possible to prevent file system users from committing invalid data to the system. The data integrity protection detects invalid data when it is read from the system. When such failures in the file system are detected, it should be possible to find the user responsible for the alteration.

Speed: The overhead incurred by the access permission system should be as low as possible. Computationally expensive operations should only be used where strictly necessary. Thus, the use of expensive public key cryptography should be limited to an absolute minimum.

8.2. Supported Permission System

As already stated, the goal of our research group is the creation of a general purpose fully decentralized file system for Unix-like systems, which is invisible to the user. Thus, the access permissions supported in this work are modeled closely on the [POSIX.1] permission system, which is used in Unix-like systems such as Linux, FreeBSD, or Mac OS X. The differences between the permission system presented in this thesis and the POSIX.1 permission system are small. The differences and the reasons for these differences are discussed in Section 10.2.

In Unix, each object that is stored in the file system is owned by a user and a group. The user who owns the object is called the *owner*. A group consists of an arbitrary number of users. Each user can be a member of an arbitrary number of groups.

Permissions to access an object can be set separately for the owner, the group and for all other users (the *world*). The possible permissions for an object are *read* (`r`) and *write* (`w`). A user with the read permission of an object may read the object in question. For a file this means that the file contents can be read, for a directory this means that the directory entries can be listed. Note that, read permissions for a directory do not imply that all objects in the directories can be read. Users with the write permission of an object may write to the object. For a file, this means that the file contents can be modified. For a directory, this means that a new object can be created in the directory and objects currently residing in the directory can be moved to any other location in the file system to which the user has write permissions. Non-directory objects in a directory may be deleted by a user with write permissions. Subdirectories of a directory may only be deleted if the user can delete all objects contained in the subdirectory.

The object owner may change the access permissions of an object. The owner may also change the group of an object to any other group of which she is a member.

The users and groups of the decentralized file system do not need to match the users and groups of the underlying Unix system. Thus, there may be groups in the decentralized file system that are not present on the local system and vice versa. It is, however, desirable to have a mapping from the local groups to the groups in the file system. The exact mechanism in which such a mapping could be implemented depends on the specific usage scenario and operating system. An example for a mapping scheme which is used in practice is the NFSv4 ID to name mapper daemon (idmapd) [Kofler, 2011, p. 934].

For the remainder of the description, it is assumed that a user that may write to an object also has the permission to read the object. This is no big restriction, write only files are very unusual. Section 10.2 will discuss this special case.

POSIX.1 also supports several other permission flags such as execute (`x`), the set user ID, and set group ID flags. These flags are also covered in Section 10.2.

In the remainder of this thesis, the access permissions are given with the standard syntax found on Unix systems. The read and write flags for the owner, the group and other persons are concatenated. For example, `rw----` is an object to which the owner has read and write permissions. Group members and other users may not read from or write to the object. `rwr---` means that the owner has full access, group members may read and other users have no access. `rwrwr-` means that the owner and the group have full access and every other user of the file system has read access. Henceforth, objects that can be read by every user with access to the file system are called *world-readable*. Objects that can be written by every user with access to the file system are called *world-writable*.

Many systems support additional access rights on top of this simple system. A way to implement ACLs on top of the proposed access permission system is presented in Section 10.3.1.

8.3. Design Overview

Each file system is controlled by a so-called *file system superuser* which was already briefly introduced in the previous section. The superuser creates the file system, generates the keys for new users and groups, adds users to groups, removes users from groups, and removes users from the file system. The superuser can also modify the access permissions of arbitrary files stored in the system. The superuser is typically the system administrator of an institution. The role of the file system superuser in the design is similar to the *certificate authority* (CA) in public key infrastructure schemes (see Section 2.8).

The superuser is not a centralized component in the file system because it does not have to be online and it is not bound to a specific physical machine. The superuser is simply the person who has the keys that are needed to perform privileged operations, which are restricted to the root user in traditional Unix systems.

Anyone can start his or her own file system by generating a new set of superuser keys.

When using this access permission system on top of IgorFs, multiple, cryptographically separated file systems can share the network and the attached storage capacity. All such created file systems are separated beyond the scope of the access rights described in this thesis: neither the superuser nor any user can access the content of another file system. Nevertheless, the encrypted data shares the same underlying storage, so that (partly) identical files can be stored efficiently.

According to the design goals of IgorFs, this is not a security problem. It is rather a consequence of the use of convergent encryption in IgorFs – the same original data will always yield the same encrypted data block. Knowledge of an unencrypted data block can be used to prove that a specific node is saving an (encrypted) copy of the data. If this is of concern, a different underlying block encryption algorithm must be used.

Throughout the remainder of this thesis, it is assumed that IgorFs is used as the underlying storage network. Thus, every data chunk is identified by a (ID, Key)-tuple. A user in possession of these two pieces of information can retrieve and decrypt the respective data chunk. An object stored in IgorFs is stored in one or several data chunks. The (ID, Key)-tuples of all data chunks of an object are called the *reference* to this object.

When another storage system is used, the (ID, Key)-tuple has to be replaced with the data addressing scheme of the specific system.

The architecture consists of two tightly connected parts: the data integrity protection and the data confidentiality protection. Because of their complex interactions, they are first briefly described in the two following sections of this chapter and then again in more detail in Chapter 9.

8.4. Data Integrity

The first goal of the permission architecture presented in this thesis is providing *data integrity*: Illegitimate modifications of data have to be detected when accessing the file system.

Unauthorized manipulation of the files and directories stored in a fully decentralized system cannot be entirely prevented [see e.g. Mazières and Shasha, 2002]. For example, attacks where current data in the system is overwritten with old data are very hard to prevent. These kinds of attacks are known as *rollback attacks*.

To completely prevent all attacks, it would be necessary to check each change of the file system for validity at the moment the change happens. This is impossible without a trusted central authority that can return the latest state of the file system. Instead, each client has to check the files for inconsistencies on its own. Hence, a rollback can only be noticed when a client that has already seen more recent changes to the file system, notices that they have been reverted. In file systems with a large number of concurrent users, it is a sound assumption that such manipulations will be rapidly detected; in file systems that are only accessed very rarely, such manipulations could go unnoticed for a long time.

The presented system prevents rollback attacks in the sense that it is able to guarantee that a client that has the knowledge of a current file will never accept an older revision of the file and show it to the user. It also prevents malicious nodes from only showing selected new files to a client. If a node sees a current file change of another user, it can also verify the current state of the other files owned by this user.

In order to establish this system, the directory structure is changed in a way that is transparent to the user. For the user, everything looks like a traditional Unix directory structure (cf. Figure 9.1 on page 146). This structure that is exposed to the user is called the *external directory structure*.

Internally, the file system uses a completely different directory structure. This so-called *internal directory structure* is mapped to the external directory structure on the fly, i.e. when a user accesses the file system. This is completely transparent to the user. The internal directory structure contains more information than the external directory structure. This information is needed for the security features.

The internal structure is enhanced as follows (cf. Figure 9.2 on page 148): each user is assigned a *user-root* directory. The user-root of a user contains all the files belonging to that specific user. It does not contain any files of other users. The user-root directory of each user contains a version number, a hash and a signature, which can be used to check the freshness and integrity of each file in the user-root.

The different user-roots are glued together with redirects. That means that, if a user directory seems to contain a file belonging to another user, it contains in fact only an invisible link pointing to the real file location within the other user's user-root.

The architecture of this integrity protection scheme is presented in more detail in Section 9.1 on page 145.

8.5. Data Confidentiality

The second goal of the permission architecture presented in this thesis is providing *data confidentiality*: Users must only able to access data to which they have the appropriate permissions. Hence, as a second step, the data integrity architecture introduced in the previous section is extended with a new method for keeping the data in the file system confidential.

To this end, a dedicated group-root directory is added for each group in the system. Each group-root directory is split into a private and a public section (cf. Figure 9.2). The public section can be read by anyone; the private section is encrypted and can only be read by group members.

The reference to each file in the file system is encrypted, unless the file is world-readable. As mentioned in Section 8.3 the reference to a file consists of the list of all (ID, Key)-tuples for the file.

Files stored in the file systems are referenced from multiple locations in the directory tree. The first reference to the file resides in the home directory of the user (see previous section). If the file is world-readable, this reference is not encrypted; otherwise the reference is encrypted with the user's private key, so that only the user in question can decrypt the file.

Depending on the file permissions, a second copy of the reference may be present in the group-root; in the public section if it is world-readable or in the private section if it is only group-readable.

If a file is modified by the file owner, the references in the home directory as, well as in the group directory, are updated. If another group member updates the file, only the entry in the group directory is updated. Thus, if a node wants to access a file, it has to check both entries and choose the one that has been updated most recently.

Using this structure, most of the Unix permissions can be mapped to the file system. A detailed architecture of this approach is described in Section 9.2 on page 154.

8.6. Client Keys

The presented file system design uses cryptography for many operations. This section introduces all cryptographic keys and gives a short overview of their usage. Later sections will refer to this information and explain the usage in more detail.

The cryptographic keys used in the file system are shown in Table 8.1 on the next page. The cryptographic algorithms used with these keys were introduced in Chapter 2.

Name	Type	Use	Generation
K_u	symmetric	user encryption key	by superuser on user generation
S_u	asymmetric	user signature key	by superuser on user generation
K_g	symmetric	group encryption key	by superuser on group membership change
S_g	asymmetric	group signature key	by superuser on group membership change
D_u	symmetric	user SD key	by superuser on user generation
K_d	symmetric	directory forward key	by user or group on directory write
ID, Key	symmetric	data chunk key	by client upon write

Table 8.1.: Keys used in the file system

Each user has a symmetric user encryption key K_u and an asymmetric user signature key S_u. Both keys are generated by the superuser when the user is first added to the file system. The keys are transferred to the user using secure out-of-band communication. K_u is used to encrypt the user's private data. This key may never be divulged to another node. S_u is used by the user to sign the user's changes to the directory structure. The private portion of this key may never be divulged to another node.

Each group in the file system has a symmetric group key K_g. This key is generated by the superuser when the group is first created. It is regenerated by the superuser on each group membership change. This is done to prevent new users from using the cryptographic material to read old data that has already been deleted from the file system (*backward confidentiality*). It also prevents old users that have been revoked from the system from reading new material written to the file system (*forward confidentiality*).

The group signature key S_g is handled accordingly.

In addition to K_u and S_u, each user has a set of subset difference keys D_u, which is generated by the superuser when the user is first added to the file system. Like K_u and S_u, the subset difference keys are transferred to the user using secure out-of-band communication. The Layered Subset Difference (LSD) encryption scheme is used to provide a method to make the group keys available to all current group members. It has been introduced in Section 2.10.

K_g and S_g are encrypted using the subset difference algorithm. The encrypted keys are stored in the file system. The exact storage locations are shown in Section 9.1.2 on page 147. Current members of the group g can use D_u to decrypt K_g and S_g. Thus, every group member can retrieve the current group key directly from the file system. Note that only symmetric cryptographic primitives are used for the decryption of K_g and S_g.

Each directory has a directory forward key K_d which is a simple symmetric key that is regenerated on each directory write operation. The exact function of this key will be discussed in Section 9.2.3 on page 159.

The IgorFs (ID, Key)-tuples (or the reference) for data chunks were introduced in Section 5.4 on page 84.

8.7. Related Work

Distributed file systems and their features were already covered in Chapter 4. This section compares existing permission systems to the presented approach the descriptions of all referred file systems can be found in Chapter 4.

The classic approach for permissions in cryptographic file systems is a *key list*, where each file has a list of keys, one key per user who may access the file. Several systems use this approach, for example Keso, SiRiUs, or Farsite. However, this approach does not scale because it requires $O(n)$ encryption operations (n being the numbers of users that may access a file). The systems using this scheme also do not usually support user groups.

Cepheus [Fu, 1999] introduced *lazy revocation.* It is also one of the first systems that enforces access permissions with cryptography. It supports group encryption. Contrary to the approach presented in this thesis, it uses a key server for group key distribution and is thus not fully decentralized. Furthermore, it relies heavily on asymmetric cryptography.

SiRiUS [Goh et al., 2003] implements read/write cryptographic access control for file level sharing. In contrast to my work, SiRiUS depends heavily on asymmetric cryptography. SiRiUS also does not provide support for groups of users (see Section 4.4.7).

Plutus [Kallahalla et al., 2003] features a permission approach using *file groups.* All files in a file group are encrypted with the same key. Permissions to all files in a file group can be granted by revealing a single key. For efficient revocation, Plutus introduced a key rotation scheme. Unlike my approach, Plutus uses asymmetric cryptographic primitives for many operations. It also does not offer full group support, copes very badly with rollback attacks and uses a scheme called server-verified writes which depends on the correct operation of other nodes in the network.

PACISSO [Koç et al., 2007] uses the same file groups permission approach as Plutus. In contrast to the work presented in this thesis, read and write access is not completely enforced using cryptography. Instead, a file owner designates a set of gatekeepers, which are responsible for granting read or write access to the data. When enough of these gatekeepers are compromised, unauthorized parties can change stored objects and legitimate file accesses can be denied. PACISSO also uses asymmetric cryptographic primitives in many operations. PACISSO uses a secure version identifier to determine the most current version of an object. This is similar to the way the most recent file system version is determined my proposal.

Like my proposed algorithm, Tahoe [Wilcox-O'Hearn and Warner, 2008] uses cryptography to enforce file permissions. However, the permission system is weaker than my design. Users which can read a directory can read all subdirectories and files stored within the directory. Users can only write to files for which they have the write key. The write keys are stored in the directory structure. Users able to write to a directory can also write to all subdirectories and files. Tahoe does not support user groups of any kind. It also does not support the revocation of access permissions. The integrity guarantees of Tahoe are very weak in comparison to the approach outlined in this thesis.

Wuala, also uses cryptography to enforce its file permissions [see Grolimund et al., 2006a]. The access semantics of Wuala are very similar to those of Tahoe. Once again, users that can read a directory can read the whole file system subtree below that directory – including all files. Users that can write to a directory can also write to all subdirectories and files. Unlike Tahoe, Wuala has methods to revoke users from the read or write access to a directory. Read access can be revoked lazily – the file is only rewritten on the next write operation. In contrast to my work, revoking write access to is very expensive in Wuala. It requires the generation of new symmetric write asymmetric write verification keys for all subdirectories of the revoked directory.

SUNDR [Li et al., 2004], introduced in is the only other system that, like the scheme proposed in this thesis, can provide fork consistency. However, SUNDR is based on centralized servers and does not feature a full-fledged permission system. Permissions are only enforced for writes, anyone with access to the file system can read all data stored in the system.

Naor et al. [2005] discuss methods to secure untrusted storage without public key operations. In their design, they concentrate on removing public key operations from other proposals such as SiRiUS. They do this by introducing a new key distribution protocol that works without asymmetric cryptographic primitives. Furthermore, they replace public-key signatures with symmetric signature schemes.

However, the security scheme considered in their work is different from the security scheme used in this thesis. They do not consider the problem of user groups. The key-distribution protocol discussed in their paper is not applicable to this work because my design does not require asymmetric cryptographic primitives for key exchanges. The protocol will, however, be discussed in the context of ACLs in Section 10.3.2 on page 179.

The authors present two schemes to replace public-key signatures with symmetric cryptographic primitives. The first scheme replaces public key signatures with MACs, the second replaces them with one-time signatures. The MAC scheme is only suitable for systems with a high number of writers that may write to all files in the system and a low number of readers. The second scheme is an one-time-signature scheme, which is only suitable for systems with a large number of read-only users and a low number of writers. Both restrictions are not suitable for the scenario assumed in this dissertation.

9. Enforcing File Permissions

This chapter presents the new permission system that was briefly introduced in Chapter 8. The chapter splits into two parts. Section 9.1 details the integrity protection scheme and Section 9.2 explains the file permission enforcement scheme.

9.1. Data Integrity

This section explains the cryptographic integrity protection scheme used to fortify the file system against several types of attacks.

The basic requirements for the integrity protection systems are as follows:

- The file system automatically validates all data while it is being read.
- The system must only return data which has been inserted by authorized parties.
- Unauthorized users must neither be able to insert new data in the system, nor to change data that is already present in the system, nor to remove data that is already present in the system.

If a manipulation is detected, the file system sends an input/output error to the process reading the data. It also sends a detailed error message with a description of the detected manipulation to the system log service.

Alternatively, the system could automatically revert to an earlier file system revision, where the manipulation is not present. This approach would be fully transparent to the user. However, this transparency would also mean that the user would not notice when working with an outdated file system revision. The user would also not notice that someone tried to attack the file system. Even worse, if the user makes changes to the file system, the other clients might not accept the new file system revisions due to the still present inconsistencies. Hence, in my opinion, this alternative should only be chosen as a last resort when requiring manual user interaction is absolutely impossible.

For file systems used in a cloud setting or in large institutions, a mixed approach could be used: When an inconsistency is detected, the file system automatically reverts to an earlier revision. Simultaneously, the file system switches to the read only mode in order to prevent the possibility of consistency problem. Furthermore, a message is dispatched to the system administration department, which can assess and remedy the situation.

The remainder of this section is structured as follows: Section 9.1.1 presents the architecture of the changed directory structure. Section 9.1.2 provides an exemplary directory structure. Section 9.1.3 details the way hashes are used to protect the directory

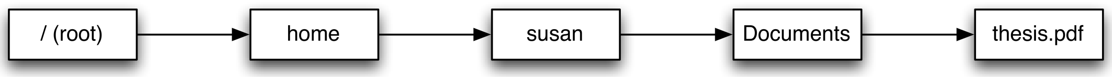


Figure 9.1.: Normal Unix directory structure

structure from tampering and Section 9.1.4 shows the way those hashes are calculated. Section 9.1.5 looks at the rollback protection architecture and Section 9.1.6 introduces a method to offer fork consistency, the strongest level of consistency possible in a fully decentralized setting. Section 9.1.7 shows how malicious users can be detected and removed from the file system.

9.1.1. Splitting the Directory-Tree

As explained in Chapter 8, the internal directory representation of the file system differs from the externally visible structure. To the outside, the directory structure looks like the normal Unix directory structure presented in Figure 4.1 on page 52.

For the remainder of this chapter, a much simpler example structure will be used. It is depicted in Figure 9.1. In the internal directory layout, the files are grouped by the users and by the groups to which they belong. Each user is assigned an individual user-root directory r_u, which only contains the user's own files and directories. Likewise, every group gets assigned an individual group-root directory r_g, which contains all files and directories belonging to the group. Depending on their exact access permissions, the directory entries of a file are either present in the user-root of the file owner or in the user-root of the owner and the group-root of the file group. Thus, the directory entries of one file can be present multiple times in the internal directory structure. This does not impose a significant storage overhead; the file data is still only stored once in the system, the file directory entries only contain the reference (see Section 8.3) to the actual file contents.

The internal directory structure has to be mapped to the traditional directory structure expected by a user accessing the file system. To enable the file system to transfer the internal representation to the visible external representation, an internal construct called a *redirect* is used. Redirects glue the different user-roots and group-roots together and enable the translation layer to deduce which files reside in which directory of the external directory structure.

Technically, a redirect R is similar in functionality to a symbolic link in Unix. Formally, R is a n-tuple pointing to one or more other objects in the file system like a regular file, directory, symbolic link, socket, pipe, block device file, or character device file[1] [see POSIX.1, p. 360f]. A redirect must not point to another redirect. When a redirect is encountered, it is automatically resolved by the file system and the referenced data is returned. Redirects are thus invisible to users of the file system. The way in which redirects are resolved depends on the type of the referenced object.

[1]On a Unix-like system all possible file types are defined in `/usr/include/sys/stat.h`.

In the remainder of this thesis, directories mostly have to be handled separately from all other object types like regular files, symbolic link, sockets, etc. From now on, whenever this thesis refers to a file, the same also applies to all other object types with the exception of directories.

Redirects are used on two occasions in the file system. The most common case is a redirect pointing to a directory. Directories in the external directory structure are typically represented by three directories in the internal directory structure. One of them resides in the user-root of the owner, the other two in the public and private parts of the group-root of the directory group. Thus, this kind of redirect is typically a 3-tuple (Section 9.2.7 on page 164 shows all possibilities.).

The same file can be present in different versions in multiple of these locations. The reason for this was briefly mentioned in the previous chapter: depending on the file permissions, a file has to reside in multiple locations because not all users may update files stored in all locations. This is explained in detail in Section 9.2. To differentiate between different versions of a file, each file has a version number v, which is incremented upon each change.

When encountering a redirect pointing to n directories, the file system reads the referenced directories, merges their entries, and eliminates duplicates by removing the items with the lower version numbers.

The second case where redirects are used is when referencing files (or symbolic links, pipes, etc.) belonging to another user. In this case, R is typically a 2-tuple, pointing to the object locations in the foreign user- and group-root. When encountering such a redirect, the system returns the item with the higher version number.

9.1.2. Example Directory Structure

Figure 9.2 on the next page shows the internal directory structure for the external directory layout of Figure 9.1. The internal root directory contains a `.users` directory, which in turn contains all the user-root directories. The root also contains a `.groups` directory, which contains all the group-root directories. Finally, it contains a `.keys` directory, which contains the encrypted access keys for group access, which were introduced in Section 8.6 on page 140.

When a node accesses the file system, it first accesses the visible external root-directory. It is contained within the user-root of the superuser and is aptly named "`/`". In the visible external directory structure, `/` has a subdirectory named `home`. In the internal directory structure, `/` contains a redirect named `home` which points to the locations of the `home` directory in the user- and group-root. (To not overly complicate the figure, the entries for `home` in the group-root have been omitted from Figure 9.1) The home-directory contains the entry `susan`, which, once again, is just a redirect to a directory inside the user-root of the user Susan. The directory `susan`, in turn, contains the `Documents` directory. For this directory, all three locations are shown. One directory location is in the user-root of Susan. The other two locations are in the public and private parts of the group-root. Depending on their permissions, files are placed in either of these directories in addition to the user-root.

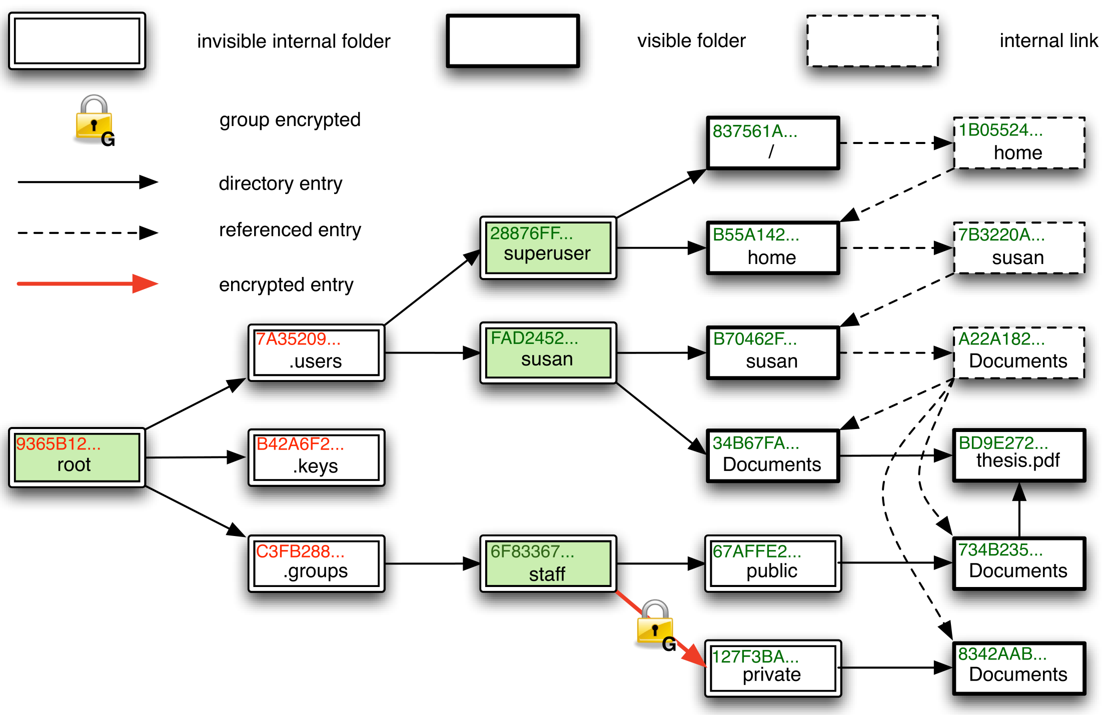


Figure 9.2.: Internal directory structure

When a user accesses the `Documents` directory within the home directory of the user Susan, she will follow the three parts of the redirect and retrieve the `Documents` directory in the user-root of Susan, in the public part of the group-root of the staff group and, if she is a member of the group, also the `Documents` directory in the private part of the group-root.

Due to its access permissions (`rwrwr-`: owner and group writable, world-readable), the file reference for the file `thesis.pdf` is present in the user-root and in the public part of the group-root of the `Documents` directory. When a user accesses the file, the file system has to check which version of the file is more recent. To this end, the version numbers of both files are compared and the more recent file entry is used. Note the use of version numbers instead of the more traditional choice of timestamps. Hence, no clock synchronization is required.

The directory structure in Figure 9.2 has a fixed height. All files in the user-roots reside four levels below the root of the internal directory tree, all files in the group-roots reside five levels below the root. It is much easier to allow certain kinds of move operations with this fixed-height directory structure; this is discussed in Section 9.2.5. The overhead imposed by this approach is very reasonable – this is discussed in Section 10.1.1.

9.1.3. Securing the User- and Group-Roots

It has already been established that all files and directories that are stored in the file system are present in a user-root and, depending on the access permissions, also in a group-root. Files and directories in a user-root may only be modified by the owner of the user-root – unless they are they are world-writable. The same holds true for group-roots: files and directories in group-roots may only be modified by members of the group – once again, unless they are world-writable. This behavior is enforced using hash trees which were introduced in Section 2.6 on page 30.

An object stored in the file system has different attributes like the file name, the file size, the permissions, etc. The attributes that may be modified by users other than the owner depend on the access permissions. For example, a user without write-access to a directory may not change the name of objects stored in a directory. Users without write-access to a file may not change the modification time or the content of the file. All object attributes that may not be changed are combined into a hash value. The attributes are listed in Table 9.1 on the following page.

The resulting hash value is stored in the parent directory of the objects. All hashes in the directory are then combined into the directory hash, which, in turn, is stored in the parent of the directory. It has already been established that each user u and each group g has an own root directory r_u or r_g with a version number v. The hashing process is repeated until the system arrives at the r_u or r_g directory. In this directory, the hash-value is also computed and then signed with the user's signature key S_u or the group's signature key S_g together with the directory version v.

When a directory entry changes, the hashes along the path from the directory to the root have to be recomputed. To put it simply, all the hashes in all parent directories of the changed directory have to be updated. Assume, for example, that Susan changes the file `thesis.pdf` shown in Figure 9.2 on page 148. Because of this change, the entry in the directory `Documents` changes and thus the hash-value of the `Documents` directory also changes. These changes propagate up the directory tree. Since the hash-value of the user-root of Susan also changes, she has to sign it again.

The signed hash-trees are used to verify the validity of all objects stored in user- or group-roots on access.

Figure 9.2 on page 148 shows the hashes above the directory names. The directories with signed hashes are drawn with a shaded background. All user-roots and group-roots contain signed hash values. Additionally, the root of the internal directory tree also contains a signed hash. This hash is signed by the file system superuser and protects the integrity of the file system up to the level of the user- and group-roots. Thus, no-one but the superuser can change the `.keys` directory or can generate or remove directories in `.users` or `.groups`.

9.1.4. Calculating the Hashes

As the previous section already mentioned, the object attributes that are hashed depend on the file permissions.

Attribute	Hashed	Precedence	Description
Name	Always	Version	The object name
Type	Always	Owner	The object type
Device	Always	Owner	The object device number (if device)
Mode	Always	Owner	The object mode
Owner	Always	N/A	The object owner
Group	Always	Owner	The object group
Size	Partially	Version	The current object size
Access time	No	N/A	The last object access time
Creation time	Always	Owner	The object creation time
Modification time	Partially	Version	Time of the last modification
Version	Partially	Version	Object version
Content reference	Partially	Version	Reference to the object contents
Reference Hash	Always	N/A	Hash of content reference (see Section 9.2.1)
Forward Key	No	N/A	Directory Forward key (see Section 9.2.3)

Table 9.1.: All file attributes and their protection

Note that these protections only apply to non-world-writable directories. The protection in world-writable directories is covered in Section 9.2.6.

Files and other objects are at least present once, namely in the user-root of the owner. Depending on their permissions, they might also be present in the group-root of the object group.

Table 9.1 shows the file system object attributes and their protection. The table shows the attributes together with a description and the information when these attributes are included in the hash. The precedence is required for the permission system and is explained in Section 9.2.1.

The modification time, version, and size are hashed if the object is not world-writable. The content reference is hashed if the object is not world-writable and if it is not a directory. For directories, the content reference is only hashed, if the directory is only readable by the owner. In this case, the directory contents can only be accessed by the owner and, hence, can also only be changed by the owner. If a directory is also readable by other users, the reference may not be hashed because the directory might contain an object that can be changed by other users. In IgorFs, each change of a file's contents causes a change of the file's reference. Hence, the file's directory entry has to be updated and the content of the directory block also changes. Thus, the directory reference has to be modifiable by other users. Note that the directory version is only incremented when a file in the directory is added, removed, or renamed or when the ownership or permission flags of a file changes. The version of the directory is not incremented when only the contents of one of its files change.

Table 9.2 on the facing page shows a few example directory entries of the home-directory of Susan. All values printed in bold are secured by the hash tree; all other values are not part of the hash tree.

For files that can only be written by Susan herself (`Secret` and `Calendar`) all information of the table is included in the directory hash. Any manipulations of the meta data of

Object Type	Name	Owner	Group	Permissions	Version	Object Location	Forward Key
Redirect	**Documents**	**susan**	**staff**	`rwr-r-`	**12**	**targets**	B2A3...
File	**Secret**	**susan**	**staff**	`rw----`	**5**	**E(ID, Key)**	
File	**Calendar**	**susan**	**staff**	`rwr-r-`	**6**	**(ID, Key)**	
File	**Share**	**susan**	**staff**	`rwrwrw`	95	(ID, Key)	
File	**Feedback**	**susan**	**staff**	`rwrw--`	**40**	**E(ID, Key)**	

Table 9.2.: Example directory entries for `/.users/susan/susan/`

these files will invalidate the hash.

For files that are writable by everyone (`Share`), the version and the location reference are not protected by the directory hash. Every user may change the file contents by adjusting the content reference and increment the file version number.

For files that are writable by the group (`Feedback`), all the information is protected. The group members can change the file contents in the group-root, but they cannot change the content in the user-root.

For redirects (`Documents`), all information about the redirect is protected with the exception of the version number and the forward key, which will be discussed in Section 9.2.3 on page 159.

The group-roots are secured in exactly the same way as the user-roots. They are signed with the group signature key S_g instead of the individual user signature key S_u.

The tables in Section 9.2.7 show a detailed list of all possible file access rights combinations and which protection is used in which case.

9.1.5. Rollback Attacks

In a rollback attack, current data in the file system is replaced with old data that has previously been written to the system. Many systems are susceptible to such attacks because the old data typically has valid, albeit outdated, signatures. Rollback attacks in distributed systems have been discussed at length in the literature [see e.g. Mazières and Shasha, 2002].

The presented architecture is very resistant against rollback attacks: Replacing a single object in a user- or group-root with an older version of the object is impossible[2]. It would invalidate the hash-tree protecting the user- or group-root. A client would notice the modification when accessing a directory containing a modified object.

However, it is theoretically possible to replace a complete user- or group-root with an earlier version. In this case, the signature matches the content of the directory. But replacing a complete user- or group-root with an earlier version is only possible in very few cases: As already discussed, every object in the file system, including the user- and group-roots has a version number v attached to it. All clients track the version numbers of all user- and group directories in the system. They read all versions every time they access

[2]With the notable exception of world-writable files. However, for these files everyone may change the contents anyhow.

the `.users` or `.groups` directory respectively. If a client notices that the version $v(r)$ of a directory has been decremented since the last access, the directory was illegitimately modified. The system reports an error in this case.

This approach limits rollback attacks to very narrow cases:

For content in a user's own user-root, a rollback attack is impossible. A user knows the current directory version of her user-root and will not accept any other version[3].

For user-roots of other users, new directory versions can be withheld. For example, when a user went offline when the user-root of another user had the version v_{old}, the user's file system instance will accept any file system revision with $v > v_{old}$ when coming back online, even if v is not the current version. It is also possible for an adversary to withhold specific changes from the system. If the adversary controls all peers to which a user is connected, the adversary could withhold the user from receiving current versions of selected user- and group-roots. These weaknesses cannot be fully prevented in a system without central authorities. Section 9.1.6 shows a way to minimize the impact to the theoretically possible minimum.

In group-root directories, another kind of rollback attack is possible by other group members. Assume the head version of the group directory is $v_{current}$. If a node with group access is presented an old version v_{old}, it does not notice the manipulation, unless that node has already read a version v' with $v' > v_{old}$. If an adversary takes an old version and modifies the group directory until it has produced a version v_{new} with $v_{new} > v_{current}$, all other nodes in the file system will accept that version. I.e. the adversary has successfully performed a rollback attack and removed a change of another node from the version history.

This kind of attack is, however, very complex and the only result is that a change is removed from the version history. Any user who is able to perform this kind of attack currently has access to the group-root and can already modify all files and directories contained within. Thus, the user could also perform the desired operation without resorting to this attack; the only difference is that the operation would remain in the version history.

In my opinion this level of robustness is satisfactory for most scenarios. It is, however, possible to guarantee an even higher level of consistency in the file system at the cost of a higher complexity and a higher storage overhead. This is detailed in the next section.

9.1.6. Fork Consistency

The maximum level of consistency that can be achieved in a fully decentralized system is called *Fork Consistency* or also *Fork Linearizability*: "Fork consistency is the strongest notion of integrity possible without on-line trusted parties." [Li et al., 2004, p. 124]. An exact definition of fork consistency together with a proof that this is the highest achievable level of consistency without a trusted server is given by [Mazières and Shasha, 2002].

[3] A rollback attack might be possible if the user changes the physical machines during the attack and the new machine is not aware of the version. However, this is a very narrow case which can, for example, be counteracted by including the current user-root version in the cryptographic material of the user.

A malicious server can fork a file system at a specific point of time and not show any modifications made by other nodes. This is called a *fork attack*. A node that only receives information from the malicious server cannot tell that it obtained an outdated instance of the file system. Fork consistence means that, if only a single modification by another node is revealed, all other modifications before this modification also have to be revealed.

It is impossible to prevent a fork attack without an online third party. With a third party, the node could consult this third party and ask for the most recent file system revision. However, mounting a forking attack is complex. The malicious user E has to control all nodes another user U is connected to. From the time of the fork, U must be completely separated from the rest of the network. A single change that propagates from U to the rest of the network or from the rest of the network to U will reveal the malicious activity.

Fork consistency can be added to the file system structure just as it was done by [Li et al., 2004] for SUNDR: All user-roots and all group-roots are extended with a *version vector* that contains the versions of all user- and group-roots together with the top-level hash of their respective hash-trees. Upon each write, a user increments the user-root version and updates the version vector. Upon writing to a group, the group-root version and version vector are also updated. Thereby, an attacker can only fork the file system. The attacker cannot create inconsistencies.

This approach also covers the attacks by group members mentioned in the previous section. When a group member tries to change the version history of a group, the hash tree of the group will change and be out of sync with the hash values saved in the user-root of the other users.

This approach does not, however, protect world-writable files. When these files are modified by a user who is not the file-owner and also not a member of the file-group, neither the user-root version nor the group-root version are incremented. Li et al. [2004] avoid this problem by not allowing world-writable files.

Hence, in the presented approach, world-writable files are not fully fork-consistent. In my opinion this is acceptable – world-writable files are not used on many occasions. When they are used, they cannot be used to store important data because everyone can change them.

To avoid the overhead of the version vectors, it is recommended to implement this extension only if it is actually needed.

9.1.7. Tracing Malicious Users

One of the desired security features mentioned in Section 8.1 is the ability to trace malicious users. In a system without central servers that check the validity of each change, it is not possible to prevent invalid changes from being committed to storage. However, when such a change is detected by the discussed integrity protection scheme, it should be possible to find the user responsible for the change.

In order to allow the tracing of malicious users, an additional signature is introduced: In IgorFs, each file system change that is committed to the network results in a new root block with a specific (ID, Key)-tuple. Each user has to sign this tuple with his private signature key S_u when sending the block to the network; unsigned blocks will not be accepted by other clients accessing the file system[4].

When an invalid entry in the directory structure is encountered, the responsible user can be found by searching for the revision in which the specific change was first created. The user who signed the root block of this file system revision is responsible for the invalid change. The file system superuser can then take appropriate actions, for example revoke all file system access permissions of the user.

Searching for the exact revision where an invalid change was committed is easy in IgorFs. A simple binary search over the file system revisions yields the revision very quickly.

This tracing scheme requires an underlying storage system with a complete version history. A possible alternative to this system is an unversioned storage system which verifies each write when it is committed. However, tracing malicious changes in a storage system, where untrusted users can manipulate arbitrary already written blocks is impossible.

The tracing scheme also does not work if the underlying system supports a garbage collection operation and deletes old unneeded revisions from the version history. However, at the moment neither IgorFs, nor any other distributed storage system known to me supports automatic garbage collection operations – the nearest equivalent is [Badishi et al., 2009], where garbage collection is initiated manually.

When such an automatic garbage collection becomes available, either the tracing or the garbage collection scheme has to be adjusted. An easy solution to the problem would be to only remove file blocks from the system and leave the directory blocks and thus the directory structure intact. This imposes only a minor storage overhead on the system. Invalid changes can still be traced to their originator using the directory structure.

9.2. Enforcing Permissions Cryptographically

This section describes the cryptographical file permission enforcement. It explains in detail how the integrity protected content references are encrypted, how the users can access it, and how it can be changed by users with sufficient rights.

Section 9.2.1 covers the general method used to protect file contents cryptographically. Section 9.2.2 presents the way in which directory permissions are enforced. Section 9.2.3 explains the need and use of forward keys in the file system. Section 9.2.4 handles the case of files stored in foreign user directories. Section 9.2.5 discusses how file name changes are handled. Section 9.2.6 shows how the security scheme works for world-writable directories. The encryption scheme is wrapped up in Section 9.2.7, which lists all possible combinations of file access permissions and the appropriate way the permissions are represented in the new file system structure.

[4]The validity of the signature can be checked by accessing the `.keys` directory in the file system. A client that already is connected to the file system can check the validity of the signature before accepting the new root block. Newly connecting clients have to check the signature after having accepted the root block. If it is found to be invalid, the user is notified.

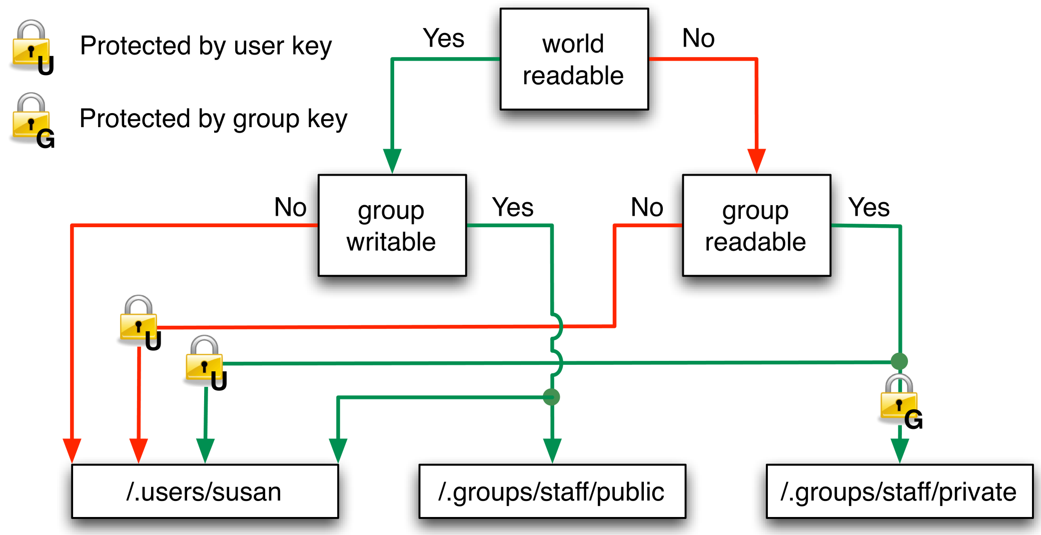


Figure 9.3.: Decision graph for file and symlink locations

9.2.1. Enforcing Permissions for Files

As discussed previously in Section 8.1, each file stored in the file system can be readable by the owner, by the owner and the group, or by everyone who has access to the file system (the world). The file location in the internal directory structure depends on its permission flags.

Section 8.5 already covered that all files that belong to a user u are located somewhere in her user-root-directory r_u. Each user has an encryption key K_u and a signature key S_u (see Table 8.1 on page 141). Likewise, each group g has a group-root r_g, just like the user-roots for the user directories. The `.groups` directory is split into two subdirectories, named `public` and `private`.

All entries in the `public` directory are stored unencrypted. The reference to the `private` directory is encrypted with the symmetric group key K_g (see Table 8.1). K_g is encrypted using the LSD scheme presented in Section 2.10. The encrypted K_g is stored in the `.keys` directory. All current group members can decrypt it using their subset difference keys D_u. Thus, all users that can access the file system can access all data stored in a group's `public` directory, but only current group members can access the `private` directory.

All access permission combinations can be enforced using the presented keys and directory structure. The decision graph in Figure 9.3 shows the locations of files belonging to the user Susan and the group Staff depending on their access permissions. It also shows if the references are stored encrypted or in plain text.

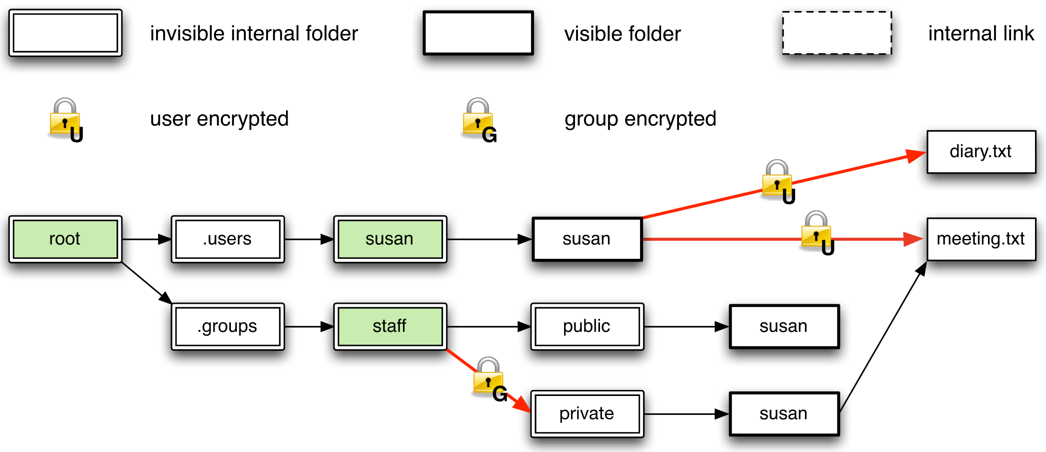


Figure 9.4.: Example directory structure with owner- and group-readable files

Files only readable by the owner

For files that are only readable by the owner, an entry is placed in the user-root. The content reference of this entry is encrypted with K_u. The owner can decrypt the reference using K_u and read the file contents. All other users of the file system can only read the unencrypted file attributes such as the size, the name, the owner, the permissions, etc. They can, however, not decrypt the reference and are thus unable to read the file contents.

An example for this is the file `diary.txt` in Figure 9.4. The entry in the user-root of the owner is protected by the hash tree – including the content reference. The owner can freely modify the file content, but all other users are unable to do so; any change would invalidate the hash tree.

Files readable by the owner and the group

For files that are readable by the owner and the group an additional second reference is placed in the private part of the group-root. The owner can still access the file reference using K_u via her user-root. Group members can access the private part of the group-root using K_g and thus also access the file reference. For users that are not group members, the situation is the same as in the previous case. They can only see the entry in the user-root of the owner and are unable to decrypt the content reference. An example for this is the file `meeting.txt` in Figure 9.4.

Files that are readable by the owner and the group can either be writable by only the owner or by the owner and the group.

First assume that the file is writable by the owner and the group. In this case, when Susan changes the file, she updates both the references in the user- as well in the group-root. When another group member changes the file, the member can only update the reference in the group-root. As covered in Section 9.1.1, the current version of the file is chosen by comparing the version numbers. Hence, if `meeting.txt` is group-writable and a group member modifies the file, only the reference in the group-root is updated. In

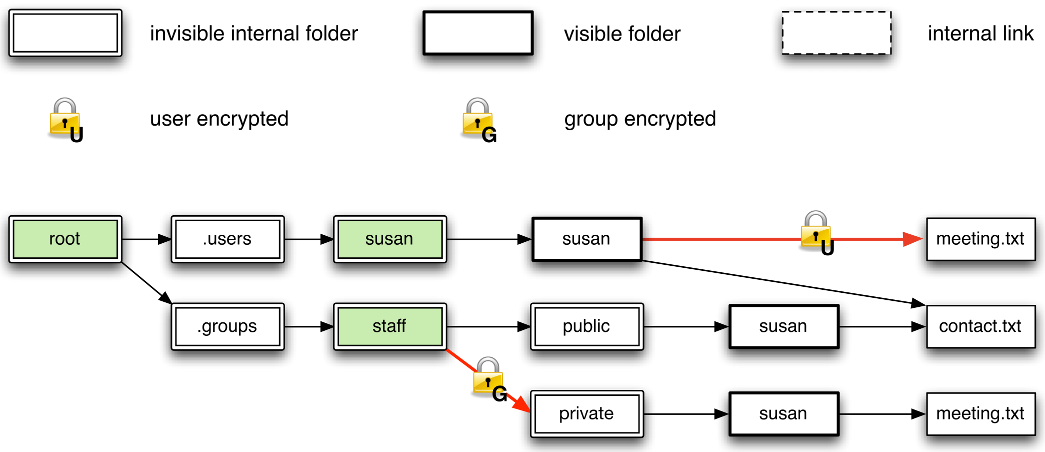


Figure 9.5.: Example directory structure with a world-readable file

this case, all users accessing the file have to search the user-root and the group-root for the most recent file version. An example for a file system after such a change is depicted in Figure 9.5 where `meeting.txt` has been modified by a group member and the second reference no longer points to the same data the reference in the user-root points to.

However, group members can also change all other file attributes in the group-root such as the creation date or even the file owner. Hence, the entry in the group-root has to be checked before being displayed.

For this reason, the concept of *attribute precedence* is introduced. Many of the attributes like, for example, the permissions, may only be modified by the owner. However, a file may be present in the user-root and the group-root of a group. In the user-root, only the owner can change the permissions of a file without invalidating the hash tree. In the group-root, each group member can change the permissions without invalidating the tree; however, group members may not change the permissions. The precedence specifies which attributes are chosen when the file versions are in conflict. Table 9.1 on page 150 which was presented in Section 9.1.4 shows the precedence of the different file attributes. For example, for the permissions, the owner's version from the user-root is always used. For attributes that may be changed like, for example, the content reference, the more recent version of the file is used.

If `meeting.txt` is only writable by the owner and not by the group, the entry in the group-root may not be changed by group members. To prevent such changes, the user-root of a user stores a hash of the unencrypted reference together with each encrypted entry (see Table 9.1). In this case, the version in the user-root takes precedence for all attributes. When a group member accesses the directory and decrypts the reference for the file `meeting.txt`, the decrypted reference is verified against the hash in Susan's user-root.

World-readable files

For files that are world-readable, the file-reference in the user-root is stored unencrypted. If the file is not group-writable, no second file-reference is necessary.

If the file is also group-writable, a second reference is stored in the public part of the group root. An example for this is `contact.txt` in Figure 9.5 on page 157. The precedence concept applies just as introduced above.

If the file is world-writable, this second reference is not necessary because everyone may change the contents directly in the user-root. In this case, the content reference in the owner's user-root is not protected by the hash-tree (see Section 9.1.4 on page 149).

As this example shows, enforcing permissions with this directory scheme is quite complex. This section only covered the case of a file, which is stored in a directory belonging to the file owner; there are many other cases which are covered in Section 9.2.7.

9.2.2. Enforcing Permissions for Directories

The previous section introduced the basic way to enforce file permissions: To protect a file, the pointers to the file are encrypted using different keys.

The protection scheme for directories has some important differences from the approach taken for files. Consider a directory that is only readable and writable by its owner. Other users, especially the group, have no access. In this case, just like in the previous section, the directory only exists in the user-root of the user. The directory is not created in the public and private part of the group-root. Without group-access these additional locations are not required.

The directory reference in the owner's group-root is encrypted with the owner's private key K_u. Hence, only the owner can read the directory. Figure 9.6 on the facing page depicts an example of this where only Susan can read her home directory.

For a directory that is readable by the owner and the group, the directory exists in the user-root of the user and in the private part of the group-root. The reference to the directory in the user-root is encrypted using K_u, the reference to the directory in the group-root is encrypted using K_g. However, for directories, group members have to be able to access both the directory in the user-root and the directory in the group-root; otherwise, they cannot access files that are only present in the user-root. To allow group members to access the directory in the user-root, an additional reference to the directory is created and placed in the private part of the group-root. The home-directory of Bob in Figure 9.6 is an example for this setting. The additional reference from the group-root to the user-root is shown in blue. This directory entry is not included in the hash tree of the group[5]. Now, group members can access both variants of Bob's home directory. Whenever Bob himself changes his home-directory, he has to update both references to the directory.

[5]Actually, the hash tree is not absolutely necessary in the private part of the group-root because everyone who can access it also knows the private group signature key S_g. Thus, the private group-root could be excluded from the hash tree for a small speed gain.

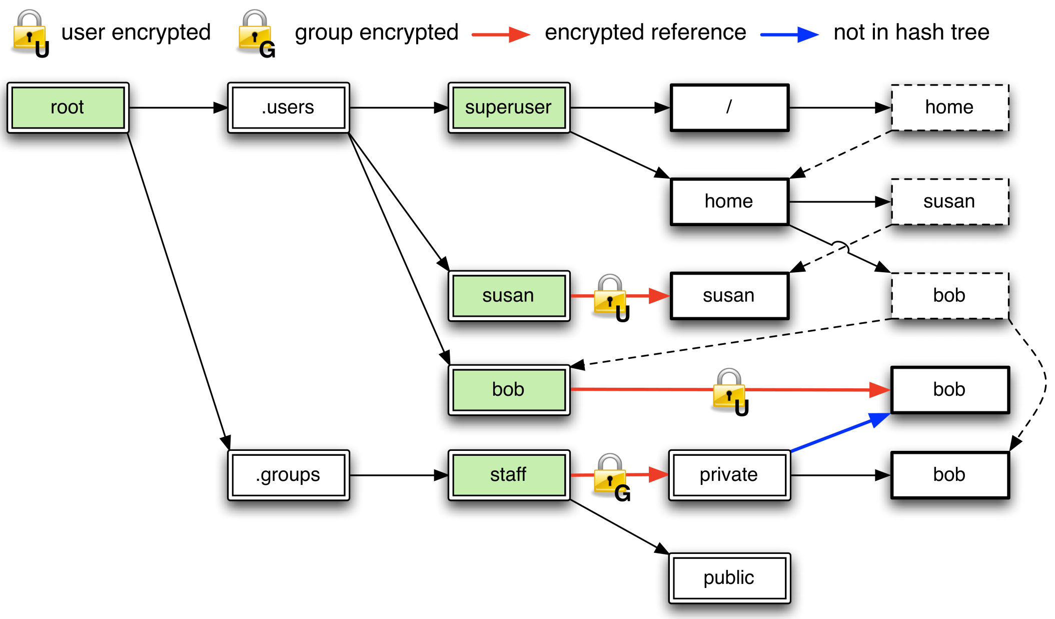


Figure 9.6.: Directory permissions

For world-readable directories, the directory forward-key remains unencrypted and the directories are created in the user-root and the public and private parts of the group-root.

9.2.3. Forward Keys

In Unix systems, a file or directory may only be read if the user has access to all directories along the path up to and including the object in question. With the presented design, this protection can be circumvented in certain settings as depicted by Figure 9.7 on the next page: The `home`-directory contains a directory named `susan`, which is private to Susan. `susan` in turn contains a directory named `Documents`, which happens to be public.

In Unix systems, accessing this directory is impossible because the parent directory cannot be accessed. In the system described so far, the encryption can be circumvented by accessing the `Documents` directly.

The presented cryptographic scheme uses *forward keys* to enforce the Unix behavior in the file system. A directory forward key K_d is added to each directory in the structure. This key is used to encrypt the directory contents. It is stored in the visible parent directory. Because in Unix each directory has got exactly one parent directory[6], one key for each directory[7] is needed.

[6]The root directory is an exception to this rule. It has no parent directory and the contents of the root-directory are not encrypted with a directory forward key.

[7]Note that the directory structure changes of the previous section are not in conflict to this statement. In the internal directory structure, there are folders with two parent directories, however, the keys are derived from the external directory structure.

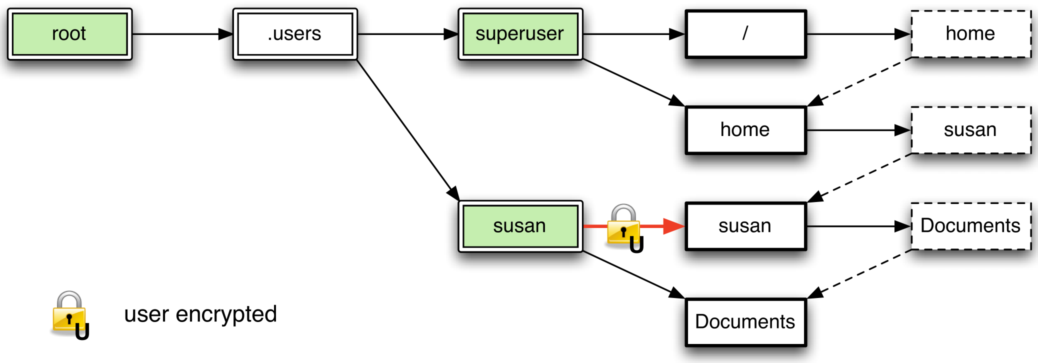


Figure 9.7.: Evading access restrictions without forward keys

In this example, the contents of the directory `susan` are encrypted with the directory key K_d(`susan`). This key is stored in the redirect in the `home` directory. The `Documents` directory is encrypted using the key K_d(`Documents`), which is stored in the reference to `Documents` in the directory `susan`. Note that unlike the user and group keys K_u and K_g, which were used to encrypt the reference to a directory, K_d is used to encrypt the actual directory block.

Thus, a user can now only access the `Documents` directory if she can access the `susan` directory that contains K_d(`Documents`). K_d is changed on each write operation to a directory to prevent users who may no longer access a directory from gaining unauthorized access.

Note that when a directory is present multiple times in the system (e.g. in the user-root and in the group-root), each of these directories has a different K_d. All values of K_d (the possible maximum is three) are saved in the reference.

9.2.4. Files Stored in Foreign Directories

The previous sections only covered scenarios where the objects that are stored in a directory belong to the directory owner. However, the files in a directory often belong to other users, e.g. when a directory is group-writable.

If a group member that is not the owner of the current directory creates a new file in a group-writable directory, the group member cannot add the file to the user-root of the directory owner. In this case, the file has to be stored in a different manner: If the file is only readable by the writer herself, she creates a redirect in the public part of the group-root pointing to the actual file location within her user-root. The file reference in her user-root is encrypted with her private key K_u to prevent other users from reading the file. Hence, anyone reading the directory can see that the file is present, but only the owner can read it. Figure 9.8 on the facing page depicts an example for this case.

If the file is readable by the writer and the group, the writer stores the file reference in the private part of the group-root in addition to the reference in her user-root and the entry in the public group-root. The entry in the public group-root is required to enable non-group members to see the file and its attributes. To prevent group members from

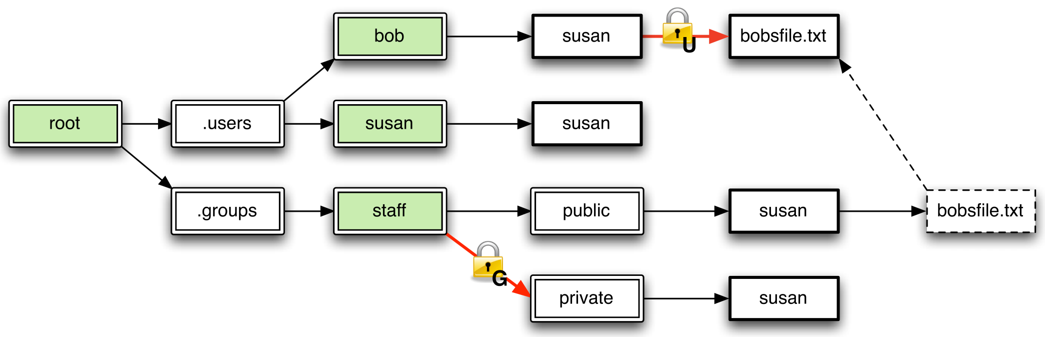


Figure 9.8.: Private file in a foreign directory

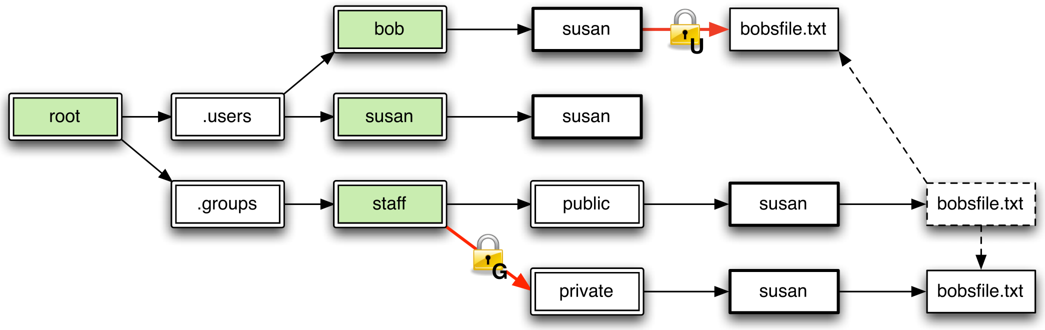


Figure 9.9.: Group-readable file in a foreign directory

changing the file or the file attributes, the precedence concept introduced in Section 9.2.1 is used – the entry in the owner's user-root is authoritative.

If the file is also group-writable, the pointers are stored in the same locations; the only change is that group members are allowed to change the content reference. Figure 9.9 depicts an example for this case.

If the file is readable by everyone and not writable by anyone but the writer, the reference is stored in the owner's user-root and the public group-root contains a redirect to the reference. If the file is readable by everyone and writable by the group members, the reference is stored in the public part of the group-root and the owner's user-root. All group members can change the content reference without invalidating the hash, the rest of the attributes are protected by the entries in the writer's root using the precedence concept. If the file is readable by everyone and writable by everyone, the user stores the file-reference in the public part of the group-root and in her user-root. The content reference is not included in the directory hash and everyone may change it.

In Unix, each group member with write permissions to a directory can remove any file stored in the directory – including files that belong to the directory owner. In the internal directory structure, all group members can delete any file of another group member from a group-writable directory, when it is saved with any method that was discussed in this section. However, files that belong to the directory owner cannot be deleted in this way because they are present in the owner's user-root from where they cannot be removed. In

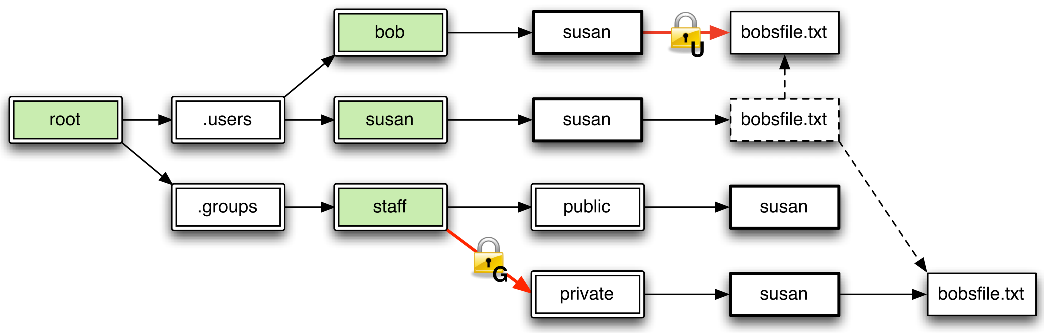


Figure 9.10.: Group-readable file in a foreign directory, which is no longer writable

this case, a *whiteout file* [see Wright and Zadok, 2004] is created by the user that tells other users reading the directory that the file in question is no longer available. This method is also employed by other Unix overlay file systems like, for example, the Linux Unionfs [Quigley et al., 2006].

For directories, the situation is different. In classical Unix systems, members with write permissions to a directory may remove empty subdirectories, even if the writer has no access to the directory contents. Non-empty directories may not be deleted. However, a user can simply move the directory to a location in the directory tree that other users cannot access. In practice, this is equivalent to the directory being deleted because the directory entry is gone and cannot be found by other users. Thus, the restriction that non-empty directories cannot be deleted is not enforced – redirects to directories are handled in the same way files are handled.

SUNDR (discussed in Section 4.2.8) also allows the deletion of foreign directories for exactly the same reasons [Li et al., 2004, p. 129].

When the group-writable flag is removed, the file system creates a redirect for every object that does not belong to the directory owner in the directory owner's user-root. This redirect ensures that these files can no longer be removed from the directory without invalidating the directory. An example of this is depicted in Figure 9.10: `bobsfile.txt` belongs to Bob, but is stored in a directory belonging to Susan that is no longer group-writable. If Bob removes `bobsfile.txt` from his user-root, the redirect points to a nonexistent location. Susan can tell that Bob tampered with the directory structure in a way that is not allowed and alert the system administration.

To prevent such broken links, Susan also could save an additional copy of the reference at the time of the directory permission change in the newly created redirect. This reference could be used as a backup; however, they would not represent new changes.

9.2.5. Renaming Files and Directories

As mentioned in Section 9.1.2, the internal directory structure has a fixed depth; in the examples, file entries are always four levels below the root if they are located in a user-root or five levels below the root if they are located in a group-root.

The reason for this is the way renaming a file or directory works in Unix: A user who has write permissions to a directory may remove all files and directories in that directory. The user also may rename all files and subdirectories within the directory. In Unix, the rename operation may even cross directory boundaries and move the object to a completely different directory in the directory structure – as long as the user has write permissions to that directory.

Due to the fixed height, the rename operation is very simple for directories. A visible directory in the external directory structure never contains real subdirectories – all subdirectory entries are always redirects. These redirects can be moved around the directory structure without having to recalculate the hash trees and without concern to the owner of the directory in which they are saved.

Moving a directory in the external directory structure is equivalent to the move of a redirect in the internal directory structure. Hence, to move a directory, the redirect in the original location is removed from the internal directory structure and the redirect in the new location is created. If the user cannot remove the redirect from the original location, for example, because it is protected by the directory owner's user-root and she is only a group member, a whiteout entry is created in the original location instead.

Files are renamed in exactly the same way.

The proposed fixed depth directory structure, where all of the user's directories are stored directly in the user-root, can become infeasible if a user has a huge number of directories. In this case, the directory block for the user-root could become too large to be manageable.

There are two common solutions to this problem; the first solution is to create a fixed depth folder structure in each user-root that is filled uniformly. This approach is used by many software projects which have to deal with huge numbers of objects like e.g. the squid web cache. An example for such a structure is shown in Figure 10.1 on page 168 with one additional directory layer. This structure is very suitable for usage scenarios where every user has a huge number of directories.

The second solution is a directory structure with a depth that grows with the number of objects that is stored. This design is, for example, also used for the inode references in a directory and was presented in Figure 4.3 on page 56. It should be preferred if users typically own a small number of directories.

9.2.6. World-Writable Directories

In world-writable directories, everyone may create objects. Existing objects may be deleted or moved to another directory even if the user does not own them and may not read their contents.

For directories, this is no problem. Only redirects are stored in a world-writable directory - everyone can move these freely to other locations.

For files this is a little bit more problematic – a world-writable directory can contain files of a user that are not world-writable. If these files would be stored directly in the world-writable directory, anyone could change the contents without invalidating a hash tree. Thus, files are also not stored directly in world-writable directories.

Instead, the same redirect scheme that is used for directories is applied – a user storing a file in a world-writable directory creates the file in her user-root and creates a redirect to the file in the world-writable directory.

9.2.7. Reference Locations

The preceding sections explained the way in which the scheme enforces file permissions. Table 9.3 on the next page and Table 9.4 on page 166 show all possible combinations of user access rights and where the references are stored. Table 9.3 on the next page shows where each type of reference is created, in all the different possible permission combinations, when a directory is not world-writable. Table 9.4 on page 166 shows the same for a world-writable directory.

As discussed in the previous sections, the locations where references are created depend on the following properties and conditions:

- the owner and the group of the object
- the object's permissions
- the object type (file or directory)
- the owner and the group of the parent directory

In the tables, the following abbreviations are used:

- PD is short for "parent directory"
- GR is short for "group-root"
- R means: file is readable
- W means: file is writable
- P means: file reference
- E(P) means: encrypted file reference
- +H means: the reference of the entry is protected by the respective hash-tree

Because of the complexity of the file permissions Table 9.3 needs a few notes:

1. If the directory containing the file/redirect has another group, these references are stored in the group-root of the file/redirect group.
2. The references are stored in the user-root of the file owner, not in the user-root of the directory owner. They are looked up via the redirect in the public part of the group-root, which is visible to everyone accessing the directory.
3. If the directory containing the file/redirect has another group, these references are stored in the group-root of the parent directory owner.

Type	PD has	PD	Access by		References in		
	same owner	writeable	Group	All	User-root	Public GR	Private GR
			File owned by parent directory owner				
File	Yes	Yes	-	-	E(P)+H	-	-
File	Yes	Yes	R	-	E(P)+H	-	P+H[1]
File	Yes	Yes	W	-	E(P)+H	-	P+H[1]
File	Yes	Yes	R	R	P+H	-	-
File	Yes	Yes	W	R	P+H	P+H[1]	-
File	Yes	Yes	W	W	P	-	-
			Directory owned by parent directory owner				
Dir	Yes	Yes	-	-	E(R)+H	-	-
Dir	Yes	Yes	R|W	-	E(R)+H	-	R+H[1]
Dir	Yes	Yes	R|W	R|W	R+H	-	-
			File not owned by parent directory owner, write via group				
File	No	Yes	-	-	E(P)+H[2]	R+H[3]	-
File	No	Yes	R	-	E(P)+H[2]	R+H[3]	P+H[1]
File	No	Yes	W	-	E(P)+H[2]	R+H[3]	P+H[4]
File	No	Yes	R	R	P+H[2]	P+H[4]	-
File	No	Yes	W	R	P+H[2]	P+H[4]	-
File	No	Yes	W	W	P+H[2]	P	-
			Directory not owned by parent directory owner, write via group				
Dir	No	Yes	-	-	E(R)+H[2]	R+H[3]	-
Dir	No	Yes	R|W	-	E(R)+H[2]	R+H[3]	R+H
Dir	No	Yes	R|W	R|W	R+H[2]	R+H[3]	-
			File not owned by parent directory owner. Parent-directory is not writable[5]				
File	No	No	-	-	E(P)+H[6]	-	-
File	No	No	R	-	E(P)+H[6]	-	P+H[1]
File	No	No	W	-	R[7]	-	P+H[1]
File	No	No	R	R	P+H[6]	-	-
File	No	No	W	R	R[7]	P+H[1]	-
File	No	No	W	W	P[6]	-	-
			Directory not owned by parent directory owner. Parent-directory is not writable[5]				
Dir	No	No	-	-	E(R)+H[6]	-	-
Dir	No	No	R|W	-	E(R)+H[6]	-	R+H[1]
Dir	No	No	R|W	R|W	R+H[6]	-	-

Table 9.3.: Reference locations for non world-writable directories. See Section 9.2.7 on page 164 for an explanation of the abbreviations

Type	Access by		References in			
	Group	All	Directory	User-root	Public group-root	Private group-root
File	-	-	R	E(P)+H	-	-
File	R|W	-	R	E(P)+H	-	E(P)+H
File	R	R	R	P+H	-	-
File	W	R	R	P+H	P+H	-
File	W	W	R	P	-	-
Dir	-	-	R	E(R)+H	-	-
Dir	R|W	-	R	E(R)+H	-	E(R)+H
Dir	R	R	R	-	-	-
Dir	W	R	R	-	-	-
Dir	W	W	R	-	-	-

Table 9.4.: Reference locations for world-writable directories. See Section 9.2.7 on page 164 for an explanation of the abbreviations

4. If the group of the file differs from the group of the parent directory, the group-root of the parent directory group still contains a redirect pointing to the entry in the group-root of the file group.

5. This situation can occur if a directory was group-writable or writable by everyone and later the permissions were changed by the directory owner or the file system superuser. When the mode is changed the file system automatically creates the redirects in the user-root of the directory owner.

6. The encrypted references are stored in the user-root of the file owner. In addition, the user-root of the directory owner contains a redirect to the user-root of the file-owner. This redirect is created by the directory owner when she changes the mode of the directory to no longer allow arbitrary file creation. The redirect can also be created by the superuser.

7. In this case, the redirect is only located in the user-root of the directory owner. The actual references are only stored in the private group-root because each group member may change the file as she wishes, anyway. File deletion is prevented by the redirect entry.

In Table 9.4, the directory only contains redirects. For files, this redirect points to either the user-root or the user- and group-roots where the – perhaps encrypted – copy of the file can be found. For directories, this redirect either points to the user-root or the user- and group-roots where the (encrypted) redirect to the actual directory can be found. If the access is not restricted, the redirect points to the destination directory.

10. Evaluation & Extensions

This chapter evaluates the proposed cryptographic file system design, details the restrictions of the chosen approach and shows several possible extensions to the design.

Section 10.1 estimates the overhead that the new directory structure and the cryptographic functions introduce. Section 10.2 shows the differences between POSIX access permissions and the presented system; it also details the reason for these differences and shows how the differences can be avoided. Section 10.3 shows how the permission system could be extended with ACLs or symmetric signatures.

10.1. Communication Overhead and Cryptographic Cost

This section estimates the overhead introduced by the system design, first theoretically and then through measurements of the prototype implementation.

10.1.1. Theoretical Analysis

As mentioned in Chapter 8, this thesis was inspired by the goal to equip an existing peer-to-peer file system with user and group access permissions. The resulting system implements access permissions by layering a new directory structure on top of the existing directory structure of a plain peer-to-peer file system. Each operation in the new directory structure causes one or more operations in the underlying hidden directory structure, which in turn causes one or more operations in the underlying peer-to-peer network.

In this section, the number of these operations as well as the associated cryptographic cost is evaluated. The cost calculation is based on the operations that occur in the underlying IgorFs peer-to-peer file system (see Chapter 5). When using other systems, especially systems with mutable blocks, the overhead can potentially be smaller.

IgorFs uses a local cache for data that has been downloaded from the network. Hence, data that has already been transferred once does not need to be transferred a second time. Data that has been decrypted is kept in the main memory for several minutes. If it is accessed regularly it is kept there indefinitely. When using such caching schemes, the proposed cryptographic permission system will have virtually no performance impact after the first read operations. Thus, the calculations in the remainder of this section assume that all local caches are empty or disabled.

Communication overhead

In IgorFs as presented in Chapter 5 (with empty caches), upon each file lookup, the file system has to fetch all the directories, starting from the root block down to the directory

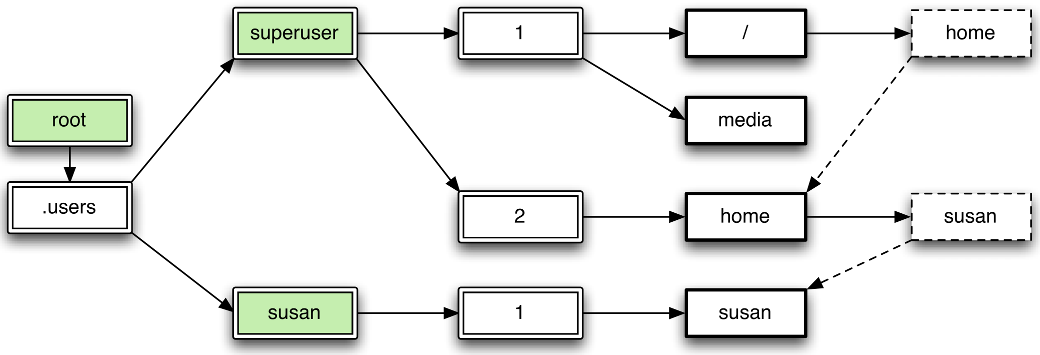


Figure 10.1.: Internal directory structure with $d = 2$

containing the file in question. Thus, the number of requests in the peer-to-peer network depends on the length of the path that is accessed. For a file in a subdirectory at depth n, exactly n directory lookups are needed. Each of the data blocks read is encrypted using a symmetric key, thus n symmetric decryption operations are needed to read the data.

In the enhanced file system, the communication cost depends on the ownership of the directories and files. Assume a directory structure where all files and directories belong to the superuser, without a support for user groups. In this case, $n + 3$ directory lookups are needed because there are three hidden directories (`root`, `.users`, and `superuser`) which must be read before the first visible directory can be read.

In the case of a single user file system, redirects do not add to the communication cost because all directories reside in the same user-root.

Section 9.2.5 on page 162 discussed that additional directory layers may be required to prevent the user-roots from becoming too large to handle efficiently. To simplify the calculations, this section assumes a fixed-depth directory structure in the user- and group-roots where the directories are rooted d layers below the user-root. The second design of user-roots and group-roots proposed in Section 9.2.5 where the depth grows with the number of stored directories are not considered. As introduced in Section 9.2.5, the case that has been used for the examples throughout this paper has the visible directories rooted a depth of $d = 1$ below the user- and group-roots. The $d = 2$ case is depicted in Figure 9.2 on page 148.

Taking this layer structure into account, $3 + n + d$ lookups are needed in the optimal case – when all objects that are looked up have the same parent directory in the hidden directory structure. The directories `/` and `media` in Figure 10.1 are an example for this. In the worst case, $3 + nd$ lookups are needed – like for the directories `/` and `home`.

In a file system with multiple users, the communication cost is larger because a new user-root has to be read when encountering a directory that belongs to another user. Let u be the number of users (including the superuser) that own directories in the path to the file. Then the number of directory lookups $2 + u + nd$ in the worst case. The *root* and *.users* directory have to be read in addition to the user-root of each affected user and the n directories which are traverse at depth d in the different user-roots.

When considering the presence of groups, the lookup costs are even larger. Depending on the exact file and directory permissions, the files and directories are present in the user- and in the group-roots and both places have to be checked.

That means, when encountering one group in the worst-case, $3 + nd$ additional lookups are needed (one lookup for *.groups*, one for the group, one for the *public* or *private* group directory and d for the specific subdirectory). When encountering g groups, $1 + 2g + nd$ group lookups are needed in the worst-case, incurring a total cost of $3 + 2g + u + 2nd$ lookups.

When dealing with world-writable directories, this figure can get even worse – in this case up to three directory lookups are necessary for one directory (see Table 9.4 on page 166). This brings the worst-case cost to $C = 3 + 2g + u + 3nd$ lookup operations.

However, these worst-case scenarios are highly unlikely. For example, when Susan accesses a file in her home-directory, the cost typically will be the minimum of 6 lookups (assuming $d = 1$) because no group-writable directories are encountered on the path from the visible root to her home.

Cryptographic overhead

In addition to the communication overhead, the proposed design incurs cryptographic cost. However, due to the near-complete abstention from the use of asymmetric cryptographic primitives in the file system design, the cost incurred by these operations is very reasonable. As already mentioned, IgorFs already encrypts each block stored in the file system once using AES. Many other distributed storage system do the same.

Thus, the cryptographic cost of the operation consists of

- Encryption and decryption of the individual file system blocks:
  The number of cryptographic operations is equal to the number of necessary directory operations calculated in the previous section plus the number of encryption/decryption operations necessary to read or write the file. Hence, the worst case number is once again $C = 3 + 2g + u + 3nd$ cryptographic operations plus the file encryption/decryption operations.

- Encryption and decryption of pointers to non world-readable files and directories:
  This number highly depends on the file system layout – typically it will be quite small. The worst case is once again C operations.

- Encryption and decryption of directories using forward keys:
  Once again, C operations is the worst case needed.

- Creation and validation of the hash trees:
  The worst-case number of hash calculations is one hashing operation for each entry in each traversed directory plus one hashing operation per traversed directory. However, hashing is a very cheap operation used very often in peer-to-peer file systems.

- Creation and validation of the hash tree signatures:
  The worst case number of hash tree signatures to be created or validated when writing or reading a file is $u + g + 1$ (one validation per user-root that needs to be accessed, one validation per group-root that needs to be accessed, and one validation for the top-level hash of the internal directory structure). The case number is $u + 1$ validations.

- Decryption of the current group-keys using the subset-difference algorithm:
  The worst-case number of symmetric cryptographic operations is $2g$: one symmetric cryptographic operation to decrypt the current group key and one operation to encrypt or decrypt the group-root. In addition, the SD-scheme requires $O(\log N)$ hashing operations [Naor et al., 2001, p. 15].

The required amount of cryptographic operations is very reasonable. Especially for hashing and symmetric cryptographic operations modern CPUs require only a few milliseconds for thousands of operations: Even a relatively slow computer[1] is able to calculate over 80 000 SHA-256 hashes of one kilobyte data blocks per second, more modern CPUs can do much better. The same holds true for the symmetric operations with over 10 000 AES-256 operations per second.

ECDSA is much more complex; using the curve P-384 [FIPS 186–2, p. 32], which was used in the prototype implementation, the computer can perform just over 1 000 sign or 200 verify operations per second. But even this is not problematic in practice because it is unlikely that a user has to verify the integrity of several hundred group-roots at one time.

All in all the overhead incurred by the cryptography is insignificant on modern machines.

10.1.2. Practical Measurement

In this section, a practical measurement of the cost using the proof-of-concept implementation is presented. Please note that the implementation was not optimized for speed in any way - the overhead for all operations can probably be significantly reduced.

For many operations, enabling the cryptographic access permissions has no effect on the speed of the operations. For example for write operations it is very difficult to establish the exact amount of overhead because the signature and encryption operations are not performed immediately for each file operation. Instead, they are deferred to a later time, i.e. until after the insertion of the data is complete.

Thus, for the benchmark, a scenario is chosen that is typical for real-world systems and has a noticeable speed difference. When opening a directory and accessing the files in the directory, the scheme has to verify the directory contents by hashing the entries, comparing the calculated hash with the stored hash signatures, decrypting the file references, etc. All these operations have to be executed immediately because otherwise the user cannot read his/her files.

To simulate this scenario, a typical Unix directory structure consisting mostly of small files has been copied into the file system. The structure consisted of 1 980 files in 290 directories with a total size of just over 25 megabytes.

[1]Using OpenSSL 1.0.0d on an Intel Core 2 Duo T7300 CPU running Mac OS X 10.6.7.

Encryption Layer	Redirection Layer	Run	Time [s]
Off	Off	1	1.557
Off	Off	2	1.379
On	Off	1	1.790
On	Off	2	1.431
On	On	1	2.303
On	On	2	1.854

Table 10.1.: Encryption layer speed measurements

The test of the file system speed consisted of the following steps:

- copy data into the file system
- restart file system to clear all caches
- read all data from the file system and measure time
- read all data from the file system a second time and measure time (measuring the cache speed)

Table 10.1 shows the execution times for both the first and the second, cached, read. The first measurement shows the time the file system takes in a *cold state*, where all decryption and integrity verification operations have to be performed. The second measurement shows the time the file system takes when the decrypted keys and the directory integrity have already been cached.

When only using encryption and no redirection layer, the numbers for this case should be equivalent to the case when no encryption at all is used. The overhead in the implementation is due to the higher complexity of the code that has to be executed. As already mentioned, this is an area where the prototype could be optimized.

When using encryption and the redirection layer, the overhead is also due to the higher number of lookups.

Table 10.1 shows the results for a measurement on a system with a 2.3 GHz CPU. The use of the new scheme incurs a worst-case overhead of about 50% when reading data for the first time. But with caching, this overhead is already much lower at less than 35 %. In my opinion this is not a bad result for a proof-of-concept system and shows the feasibility of this approach.

Unfortunately, these results are not easily comparable against the speed of other cryptographic file systems. As mentioned, the prototype implementation is in not speed optimized at all. This fact alone makes direct comparisons between different implementations impossible. But even if the prototype implementation of the file system was properly optimized it would be very hard to compare it against other file system security schemes. Most other implementations never left the prototype stage and are not freely available. The speeds given in the published papers are not easily comparable due to the differing system performances.

10.2. Differences to POSIX

The approach presented in this thesis implements most of the POSIX.1 mandated access control features. However, there are a few differences and restrictions, which are described in this section.

10.2.1. The Execute Flag

In addition to the read and write permissions for files and directories presented in this thesis, Unix also provides an execute permission flag (`x`). A person that only has the execute permission on a file may run the program but may not read the binary program data. A person that only has the execute permission on a directory may access the directory and any subdirectories and files in the directory she has access to. The person may, however, not list the directory contents. She has to know the exact name of the object she wants to access.

These features were not implemented. For files, this is due to the simple necessity that in a distributed setting a node must be able to read an executable file to run it. Thus, this access restriction would have to be enforced by the local client – which can be altered at will by its controlling users.

For directories, it is theoretically possible to implement the execute-flag. One possible solution is to encrypt each directory entries with a separate key, derived by its file name. Kaliski [2000] lists several methods for key generation from strings that could be employed to this end.

When trying to read a file with a specific name, a client first generates the key for the file name and then tries to decrypt each directory entry with this key. Due to the speed of symmetric cryptographic primitives this should lead to no noticeable performance issues.

However, when using this approach, the security of directory entries is dependent on the file name length and uniqueness. A user can simply try to brute force the names of the directory entries. However, this is also possible on a non-distributed Unix system where a client can simply try to guess the file names.

Another problem introduced by this approach is that a user can tell how many files are stored in the directory even if it cannot read them. If this is of concern, bogus entries could be introduced to the directories with this permission constellation. In an optimal case, the number of bogus entries is large enough that all directories with such permissions have the exact same numbers of entries.

10.2.2. Blind Writes to Files

This thesis assumes that a person with write permissions to a file also has read permissions to said file. This does not have to be true; POSIX allows users and groups to have write permissions to a file without having read permissions. In the literature, a write to a file without reading from it is known as *blind write*.

This access permission constellation is not supported by the permission system so far because it is very uncommon and introduces a number of complications to the system design. Note that, according to POSIX, a person without read permissions but with write permissions may overwrite the complete contents of a file, even if the user may not read it. Thus, having write permissions on a file without read permissions does not mean that the writer only can append new data to the file. I am not aware of any case where this permission combination is used in practice.

However, if a use case for this can be found, support for blind writes can be added to the file system. In addition to the symmetric keys, a public/private key pair is assigned to each user and group. The public keys are published in the file system. If a user wants to write to a file, she uses the public keys of the user and/or the group to encrypt the new file references. After that, she will not be able to read the file contents via the file system because she is not able to decrypt the references herself. The group or owner of the file, however, will be able to decrypt the references with their private keys.

The asymmetric cryptography that is needed for this feature introduces a high cost and complexity. It is thus advisable to only implement this feature if it is really needed. Moreover, the mechanism also introduces some backward confidentiality problems when it is used with groups: The symmetric key of the groups can be changed each time a user is added to or removed from the group because it encrypts only a single directory reference (the reference to the private group directory). However, files encrypted with the public key of the group could potentially be spread throughout the whole group directory tree. Changing the asymmetric key upon each group change could thus take very long because each file would have to be found and re-encrypted. Otherwise, if the key was not changed, each new member of the group could access every file that was encrypted with public key cryptography. This would include files that had been deleted before the user was added to the group.

10.2.3. Blind Writes to Directories

Blind writes to directories are also possible when a user has the execute and write permissions but no read permission to a directory.

It is not possible to implement blind writes on directories with the exact same features as in a local file system. As explained in Section 9.2.4, users with write permissions to a directory can delete or move all entries in the directory. Thus, in this case, a user could simply remove all encrypted pointers from the directory even when she cannot decrypt them. This is not possible on a traditional Unix system without brute forcing the file names.

There are once again methods that can be used to impede such attacks like, for example, the introduction of bogus entries that invalidate the directory tree when they are removed; however, this is not a satisfying solution.

Letting a user add files to such a directory also would be possible using public key cryptography. When a client creates an entry in such a directory, the entry is encrypted with its file name, just as explained in the previous section. The file name is also encrypted using the public key of the directory owner and, if group members may read the directory, using the public key of the directory group. Users with read permissions to the directory can resolve the file name by decrypting the directory entries.

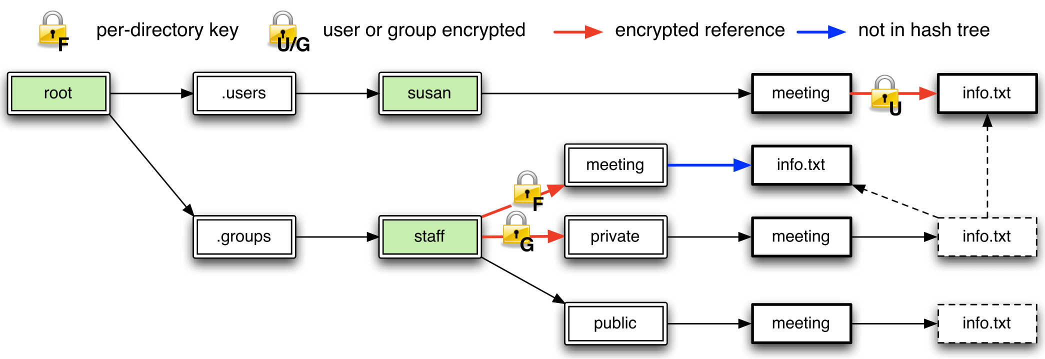


Figure 10.2.: File with foreign group

The only system supporting blind writes that I am aware of is Farsite (introduced in Section 4.4.13). Farsite enforces this behavior by relying on byzantine fault tolerant server groups that have to behave correctly [see Douceur et al., 2002a, sec. 2.2]. One of the design decisions of this thesis was that there are no nodes or groups of nodes with a special trust placed into them; thus, this solution is not suitable for this thesis.

10.2.4. Owner Group Membership

Another design assumption of the system was that the file owner is a member of the group to which the file belongs. This is a sound assumption because in Unix a user may only change the group of a file she owns to a group of which she is a member.

However, in Unix systems, the superuser can override this restriction and change the group of a file arbitrarily.

In the presented design, the file owner is unable to update the file entries in the group-root if she does not belong to the file group.

This restriction can be circumvented using several different methods; one method is using public key cryptography like for blind writes – however, that would once again introduce backward confidentiality problems.

Another method is the extension of the directory scheme combined with the introduction of new keys.

Figure 10.2 depicts an example. Assume that Susan is not a member of the group Staff. `info.txt` belongs to Susan and the access permissions are `rwrw--` (group members may read and change the file, other users have no permissions at all). The directory `meeting` does not belong to Susan but to an unspecified user.

According to Table 9.4, in this case, file entries have to be present in the user-root of the file owner, in the public group-root (to allow non group members to see the file presence), and in the private group-root (to allow group members to read the file). When the user is not a member of the group, she cannot update the references in the group-root upon file modifications.

To prevent this problem, the superuser generates a new, file-specific key in such constellations. This key is made available to members of the group and to the user. The copy in the group-root is not placed in the private part of the group-root; instead a dedicated directory is created that can only be accessed with knowledge of the key.

The entry in the group-root is not included in the hash tree; this is not necessary because only group members (which can already modify the hash-tree) and the user can access the directory. Thus, the file owner can update the entry in the group-root without invalidating the group-root hash tree.

The other permission combinations can be covered with similar constructs.

Note that to ensure forward and backward confidentiality, the superuser has to exchange the file-specific keys of all such files belonging to a group upon each group membership change. With many files this can quickly become infeasible.

Thus, this extension should only be implemented when absolutely necessary.

10.2.5. Setuid, Setgid and Sticky

POSIX defines some special file permissions: the set-user-ID-on-execution flag *setuid*, the set-group-ID-on-execution flag *setgid*, and the *sticky bit* [see POSIX.1, p. 361]. The setuid and setgid bits are easy to implement; they can be secured with the hash-trees just like the remainder of the data in the file system. Special care has to be taken to ensure that the user- and group-IDs are in sync across the network; alternatively a mapping system like the idmapd of NFSv4 has to be used (see Section 8.2).

The *sticky bit* is usually set on directories: when it is set, files can only be renamed or deleted by the directory owner, the file owner, or the superuser [POSIX.1, p. 97]. This is usually used in conjunction with group- or world-writable directory. The result is a directory where all users with write access to the directory users can create objects – but in contrast to usual group- or world-writable directories, the created objects cannot be removed or renamed by everyone. Thus, the flag is also known as the *restricted deletion flag*[2].

In today's Unix systems, the sticky bit is usually set on the `/temp` directory. It has also other uses; it can e.g. be used to allow students to deliver their homework directly to a course instructor's home directory.

Unfortunately, adding the sticky bit to the proposed file system design is not easy. It is easy to prevent the deletion of files and directories that belong to the directory owner – they simply are added to the hash-tree. However, files added by other users will not be included in the hash tree and thus can be deleted by anyone without invalidating the directory tree.

Adding support for the sticky bit to the system is left to future work.

[2]The name *sticky bit* is historical: on earlier Unix systems the sticky bit was set on often used executables. It instructed the operating system to keep the code of the program in (swap) memory after the program exit. Today, according to [POSIX.1], the behavior of the flag is unspecified when set on non-directory objects.

10.2.6. Deletion of Subdirectories

In a traditional POSIX file system, if a directory owned by A contains a directory owned by B and A has no write access to B, A may delete the subdirectory of B if it is empty. This even holds true if A has no read permissions to the directory owned by B. However, A may not delete B once B contains at least one object.

In the proposed design A may delete all objects stores in directories of which A is the owner. This includes empty and full subdirectories. The reason for this was presented in Section 9.2.4 on page 160 in detail. Briefly put, users can always move subdirectories to an inaccessible location – this is virtually equivalent to a delete operation. Thus, does not make sense to enforce this restriction.

10.2.7. Access Exclusion of Groups

A relatively unknown feature of POSIX access permissions is the possibility to exclude a certain group of users from accessing a file or directory.

When a file f belongs to a user u and a group g and the file access permissions are `rw--r-` (owner may access, group has no access, and file is world-readable), one might suspect that everyone may read the file because it is world-readable. Actually, every user of the file system except members of g may read the file in this case – with the exception of the owner who naturally may read the file nevertheless.

Support for this constellation can easily be added to the file system. There are two possible constellations: either the file is only writable by the owner, or the file is writable by the owner and the world.

In the first case, the file reference is included in the owner's hash tree. The object is encrypted with a per-file key k. The owner uses his subset difference key set D_u (see Section 8.6 on page 140) to encrypt the object reference using the subset difference algorithm introduced in Section 2.10. To be able to encrypt the file in a way that no group members can read it, the list of members of each group has to be published by the file system superuser. The list can e.g. be made available in the `.keys` directory.

If the object is also writable by the group, special care has to be taken that group members cannot write to the file. One way to do this is by introducing a special *secret token* is added to the file attribute, encrypted and added to the user's hash tree. This token is included in the user's hash tree. When the directory is read, the presence of the correct token in the object entry is verified. Users that are not a member of the group can read the token. When changing the object, they simply encrypt the token together with the new object reference.

As mentioned before, this POSIX feature is widely unknown. While it does not impose a significant overhead, it is makes the file system more complex. This feature should thus only be implemented in environments where such access permission combinations are necessary.

10.2.8. Process Group Memberships

In this thesis, it is assumed that permission changes like e.g. group evictions take place immediately. E.g. after Susan is evicted from the group staff she will no longer have access to any files private to this group.

This does not represent the semantics of Unix systems exactly. In Unix, the current group memberships are set per process. If a user is evicted from a group, all processes of that user that were started while the user was a group member retain the group membership. The processes can still access group owned files.

These semantics cannot be reproduced in a fully decentralized setting with no trust in the client machines because there is no way for other machines to know what requests originated from which user processes.

This problem is non-trivial (and to my knowledge unsolved) even in the currently available network file systems. For a more detailed discussion of this problem, see [Falkner and Week, 2006, sec. 6].

10.3. Extensions

This section presents several possible methods to implement ACLs on top of the file system design in Sections 10.3.1 to 10.3.3. Finally, the use of symmetric signatures is briefly discussed in Section 10.3.4.

10.3.1. ACL Support

The permission system presented in this thesis does not support Access Control Lists (ACLs). This section sketches a method to implement ACL support on top of the proposed design.

ACLs allow users to set more fine-grained access permissions on their files and directories. Note that on Unix, ACL schemes *extend* the traditional Unix permission system that was presented in this thesis, they do *not replace* it. Unix systems supporting ACLs still have to support all facets of the presented permission system. Thus, any Unix-like ACL system for decentralized file systems has to be layered on top of the presented system or on top of another system with a similar set of features.

As of September 2011, there is no ACL standard that is supported by all operating systems. Linux and many Unix systems use ACLs that are based on a POSIX draft [POSIX ACL]. The specific implementations can differ in detail [see Grünbacher, 2003]. There are also extensions for NFSv3 to support POSIX style ACLs. However, NFSv4 introduced a new, incompatible ACL standard [Shepler et al., 2000, sec. 5.9] with permissions that are similar to the permissions found on Windows systems[3]. Because the underlying Linux file systems at the moment do not support NFSv4 ACLs, the server has to map them to each others [Eriksen and Fields, 2005]. But NFSv4 ACLs are much finer grained and in many

[3]The incompatibility of the NFSv4 and the POSIX ACLs leads to several complications; for a discussion see e.g. [Kirch, 2006]

cases the mapping is impossible. In these cases, current NFS servers simply reject the operation. These problems make the NFSv4 ACL systems much less useful in practice.

Solaris uses either POSIX style ACLS or NFSv4 ACLs, depending on the underlying file system[4]. Mac OS X uses yet another incompatible set of ACLs in its HFS+ file system [Mac Security, p. 54ff].

All these ACL schemes are non-trivial, with features such as permission inheritance or append-only directories. Implementing such permission systems is difficult in centralized systems; in a fully decentralized environment these problems multiply.

Hence, this section cannot show a way how to implement a full-fledged ACL system on top of the proposed design. It only sketches a way to support rudimentary ACLs based on the POSIX draft, without support for advanced features such as ACL inheritance or user exclusions. A brief, understandable description of POSIX ACLs is given in [Solaris Security, p. 131ff].

This section shows how to implement a permission system that allows the file owner to give an arbitrary number of users and groups read or write permissions to a file or directory in addition to the permissions that are already given by the Unix flags.

To support ACLs with the system presented in this thesis, the encryption system has to be modified. Every user and every group needs to have a public/private key pair.

Each non world-readable file and directory has a unique symmetric key, which encrypts the object reference. This symmetric key f has to be made available to all users and groups with read-permissions by encrypting it with the user's or group's public key. Thus, the encryption time rises linearly with the number of ACL entries. Files and directories that use access control lists always have to be stored in the public part of the group-root to allow other users and groups to read them. If a file is present multiple times in the directory structure, f is different for each location.

A file or directory is present in the user-roots and group-roots of all users and groups that have write permissions to the object. When accessing such a file or directory, all user- and group-roots have to be checked for the latest revision. Thus, the lookup overhead increases linearly with the number of ACL entries.

One possibility to limit the overhead is to limit the number of ACL entries. This is not unusual; many file systems in Linux also have such restrictions. e.g. for XFS the maximum number of ACL entries is 25 [Grünbacher, 2003, p. 266]. Note that ACLs also introduce a considerable overhead to local Linux file systems. The first lookup operation for a file can take up to several hundred times as long when using a five-entry ACL [Grünbacher, 2003, p. 266].

Forward and backward confidentiality for group operations is difficult in this system because the asymmetric group key has to be changed on each change of group memberships while still allowing current group members to access files encrypted with the old keys. The key rotation scheme introduced in Plutus [Kallahalla et al., 2003] is a suitable solution for this purpose (see Section 4.4.8). Using this scheme, the file system superuser generates a new symmetric encryption key k_i for the group. For each group revision a new asymmetric key pair has to be created. The private key is encrypted with k_i and stored together

[4]POSIX ACLs are used for UFS, NFSv4 ACLs for [ZFS].

with the unencrypted public key in a readily accessible location on the file system. Thus, all group members with a valid current key can determine all previous symmetric and asymmetric keys. Current group members can derive all previous key values k_{i-j} from the new key; revoked members have no knowledge of the new key and cannot calculate the new key from the previous keys. The group key first encrypted using the LSD algorithm (see Sections 2.10 and 8.6) and then stored in the `.keys` directory of the file system.

This approach has the same backward confidentiality problems that have previously been discussed in Section 10.2 on page 172. For example, new group members will be able to read the content of files that were deleted before they joined the group.

Due to the complexity and the overhead of this approach it should only be implemented when absolutely necessary.

10.3.2. ACLs Without Public Key Encryption

The ACL scheme proposed uses public key cryptography to read protect the data. Naor et al. [2005] propose to use the secret key exchange scheme of Leighton and Micali [1994] to avoid public key cryptography in decentralized file systems. Leighton and Micali [1994] present a scheme for secret key exchange between two entities using a trusted agent. The trusted agent need not be online at all times, the agent only needs to generate some key material and make it available to all users. Thus, in the presented file system design, the file system superuser would be the trusted agent.

The scheme could be implemented to replace public key encryptions in the proposed design. However, the scheme does not scale. It requires the file system superuser to create two matrices of size n times n (with n being the number of users added to the number of groups in the system). This obviously does not scale for large numbers of users. Hence, in my opinion, this scheme should not be used in a distributed file system.

Naor et al. [2005] also sketch a way to use the scheme to replace public key signatures. This scheme requires m encryptions for each signature (m being the number of users that can verify the signature) – this is also clearly unacceptable.

10.3.3. ACLs Through Identity and Attribute Based Encryption

Identity and attribute based encryption are two new cryptographic schemes which were introduced in Section 2.11 on page 36.

In principle, ABE and IBE schemes could be used to implement ACLs in distributed systems. To allow some arbitrary user to access a file, the current file keys could simply be encrypted with the user-ID. Unix groups could be seen as attributes: each user has a set of group-attributes that they can read. The current file keys are then encrypted with the group-ID.

However, there are some serious problems when trying to use this approach with groups: key revocation in IBE and ABE schemes is a difficult subject; proposals for usable schemes have only been made recently, Boldyreva et al. [2008] proposed a revocation scheme with a non-prohibitive cost for IBE which can also be adapted to ABEs. This scheme was further extended for ABE usage by Attrapadung and Imai [2009].

These recent findings could make the use of IBE and ABE viable for distributed file systems. However, to the best of the my knowledge, there are no implementations available yet that support revocation schemes[5]. Revocation is absolutely essential for file systems because otherwise it would e.g. be impossible to revoke old group members from a group.

IBE and also ABE both rely on asymmetric cryptographic primitives and thus are slower than the respective symmetric operations used in the approach of this thesis. IBE and ABE also only solve the cryptographic problems associated with user- and group-management. This thesis also discusses and how to guarantee the validity of the data in the file system to protect the data e.g. against rollback attacks.

Future implementation should re-evaluate the suitability of IBE and ABE schemes for decentralized file systems.

10.3.4. Signature Speed Improvements

Symmetric signatures (also known as one time signatures) were introduced in Section 2.7 on page 31.

Usually the use of symmetric signatures in distributed file systems is difficult; e.g. Naor et al. [2005] argue that such signatures become difficult to implement when multiple writers are required because the internal state of the signature has to be synchronized for all of them. In the architecture of this thesis this problem does not occur. Each user has a dedicated signature, which is never shared. Only groups have to share the internal state of the signature scheme; they can simply use their symmetric cryptographic key to this end.

The file system superuser has to generate a new group-signature in the case of a user-revocation; however, this is also the case when using traditional signature algorithms like RSA or ECDSA.

I chose to not use symmetric signatures in this work because they have higher signature lengths and open source implementations are not as readily available. However, this choice should be reevaluated in the future.

[5]To be exact, to the best of my knowledge only the implementation by Bethencourt et al. [2007] is freely available.

Part IV

Conclusion

11. Conclusion

This thesis has covered the topics of redundancy schemes and permissions for fully decentralized file systems which – up to now – were not solved satisfactorily.

At the time of writing, most peer-to-peer file systems use replication as redundancy algorithm, albeit more efficient coding schemes have been known for a long time. Some systems use erasure codes – however, in practice, they are problematic for decentralized storage systems because of their higher complexity and bandwidth requirements for block repairs.

In this thesis, several redundancy algorithms were analyzed in terms of suitability for peer-to-peer storage systems. Besides replication, MDS erasure codes and a replication/erasure hybrid scheme are evaluated. Additionally, this work also considers a simple XOR-based random linear coding redundancy scheme which performed extremely well. The behavior of the different algorithms has been explored in simulations using synthetic data as well as real-world network traces.

Furthermore, this thesis has presented a distributed adaptive redundancy maintenance algorithm which ensures data availability in peer-to-peer storage networks. The presented algorithm does not depend on any special features of the underlying peer-to-peer storage network; it can be implemented on any kind of peer-to-peer environment. The algorithm was tested both in high and low churn environments, in conjunction with the aforementioned redundancy schemes.

All simulations performed over the course of this work, both with and without the redundancy maintenance algorithm, show that random linear coding is the best fit for decentralized storage systems.

In contrast to the other algorithms, RLC loses a very low amount of data even for a significant number of failed peers. Once data loss sets in, RLC shows a quite rapid degradation. This is not considered a drawback because P2P storage systems should avoid data loss altogether.

Simulations with a Kademlia peer-to-peer network trace show that, even without active regeneration of the stored data, data loss sets in only after a few thousand hours. At this time, replication – the most widely used redundancy mechanism in peer-to-peer systems today – has already lost about 5% of its data. Dynamic simulations show that RLC imposes a lower maintenance traffic than all other tested redundancy schemes.

Additionally, with random linear coding a far lower number of blocks have to be changed on small write operations than with erasure coding. For many peer-to-peer file systems, this problem of erasure coding is one of the most important reasons for preferring replication.

Random data access is also faster for RLC than for MDS erasure codes because the data can be read directly from the network. With MDS codes, a (configuration dependent) number of blocks has to be read and decoded first. Especially for small reads, this can impose a significant overhead.

Future work could try to measure the performance of RLC and the other presented schemes in a scenario where a decentralized file system is actively used. These measurements could either be derived from simulations using file system access traces or from implementations that are actually used in a real network.

Besides the work on redundancy schemes, this thesis has handled the topic of integrity protection and file access permissions in peer-to-peer file systems. Enforcing access permissions in decentralized systems is much harder than in server-based environments because the nodes of a peer-to-peer system cannot be trusted to enforce the permissions by themselves – malicious peers of the network could simply choose to disregard the permission information. Hence, enforcement of access permissions in peer-to-peer file system has to be based on cryptography.

The existing permission systems for decentralized file systems are all lacking in features in comparison to the POSIX permission system widely used in Unix-like systems. This is due to the complexity of mapping the POSIX permission system to cryptographic structures. Thus, many peer-to-peer file systems only offer very simple access semantics without support for user groups. Only a few, more sophisticated, systems offer group-access for files, which is widely used in Unix-style operating systems. However, even these systems are very limited in comparison to the POSIX permission system. Furthermore, at the time this thesis is written, all permission systems used in peer-to-peer file systems depend heavily on slow asymmetric cryptographic primitives.

This thesis has presented the first proposal that provides cryptographically enforced Unix-like access permissions in a *fully decentralized* manner. The design is based on a hidden internal file system structure, which adds the required meta-information to a plain peer-to-peer file system. This hidden internal file system structure is mapped on the fly to the usual Unix directory structure. This is transparent for the system's users. The actual file permissions are enforced using a completely cryptographic data protection scheme where users can only read and decrypt the files to which they have access.

In addition to the POSIX user- and group-permissions, the file system structure also protects the integrity of the information stored within the file system. An integrity verification algorithm checks the validity of the file system state upon access. The presented approach can even provide *fork consistency*, the highest level of consistency possible in a distributed environment without trusted servers.

The proposed design is very efficient. In contrast to other permission systems for peer-to-peer storage systems, the number of asymmetric cryptographic operations is kept very low. Only one asymmetric cryptographic operation is required per user and per group, from which data is read. Other systems typically require one operation per object that is accessed.

The proposed design matches the POSIX standard as closely as possible without seriously impacting the system's performance or complexity. All deviations from the standard have been discussed and ways have been shown to implement the specific features, if they are needed.

The open-source implementation of this proposal, which was used for the performance tests, is available for download at the web site of our research group Amann [2011b].

Future work could discuss the implementation of different ACL schemes. This thesis sketched a way to add a simple ACL layer to the presented permission system – however, some ACL systems offer advanced features like append-only directories that the presented approach cannot yet support.

Together, the presented redundancy maintenance algorithm and the cryptographic permission system solve two of the most important open design problems for decentralized file systems. Future decentralized file systems can use the results of this work to offer Unix-like permissions and efficiently protect the availability and durability of the shared data.

Appendix

A. The Simulation Environment

All redundancy simulations in this thesis were computed using a simulation environment custom built for this task. Unfortunately no existing simulation environment that fit the specific requirements was found. The simulation environment is freely available on the web site of our working group [Amann, 2011a].

The simulation environment was written in Perl. While Perl scripts are much slower than comparable programs in, for example, C, the wide array of available libraries allowed rapid development. The system is also very modular and easily extendable.

Each simulation using the Kademlia data takes about 3 days with an 8 core 3 GHz processor. Each simulation on the Microsoft trace takes about 9 hours; simulations on synthetic data take only two to three hours.

The parameters for each simulation were given at the start of the respective sections. There are several configuration files included in the `config` directory with preset parameter values for the simulations performed in this thesis. The results for each simulation were averaged over a minimum of 20 simulation runs.

A.1. System Requirements

The minimal version of Perl required to run the simulator is Perl 5.10.0. The following CPAN Perl modules are needed in addition to the Perl core modules:

- forks
- Moose
- MooseX::Runnable
- MooseX::Getopt
- MooseX::SimpleConfig
- MooseX::Log::Log4perl
- Statistics::Descriptive
- Carp::Assert
- Log::Log4perl
- Inline with support for the C language

A.2. Usage

The simulator is started by running the Perl script named `runner.pl`. When started without options, the script will show an overview over the possible command line options. The available options depend on the exact choice of modules – usually not all of the following options will be available.

```
usage: runner.pl [-?] [long options...]
   -? --usage --help            Prints this usage information.
   --configfile                 Configuration file to use
   --module                     Module to use (Replication,
                                Erasure, ...)
   --refresh                    Refresh module to use. Default:
                                NoRefresh
   --out                        output file
   --shooternum                 shooter (1=simple, 2=kad/64 bit
                                systems, 3=kad/32 bit systems)
   --shootercache               shooter cache file for
                                getOnlineNodes. Not required, but
                                simulations things faster
   --kadfile                    data file for kad shooter
   --erasurecount               erasure level
   --numblocks                  number of data blocks
   --numredblocks               number of redundancy blocks
   --numpeers                   number of peers (default: numblocks +
                                numredblocks)
   --startepoch                 start at epoch x (default: 0)
   --maxepoch                   maximal epoch. default for shooter=1
                                = numpeers, for shooter=2 51551
   --stepping                   epoch stepping to use; default 1 (do
                                simulation for every epoch)
   --repetitions                how often to perform each operation;
                                default: 10
   --threads                    how many simulations run simultaneously.
                                default: 3
   --max_peers_in_shooterfile   How many nodes do we have in our
                                inputfile at max? Default: 400375.
                                Usage saves memory.
                                information. default: /dev/null
   --lostout                    output file for lost block
                                information. default: /dev/null
   --first_stepsize             after how many epochs is a block is
                                checked for the first tim? (default: 1)
   --maxstepsize                Maximum step size for checking
```

```
                                   blocks. Default: 9999999999
   --speedhack                     Skip simulation of epochs where no
                                   message exchange occurs? Default: 0.
```

The `-?` and other help options are self-explanatory, the other options are explained in more detail below

- configfile
  The name of a configuration file containing command line options in yaml or XML format.

- module
  The name of the redundancy algorithm that is to be used in this simulation run. Possible values are `Replication`, `Erasure`, `ErasureHybrid`, and `SeveralBlocks`. There also is a module named `FindFirst`, which finds the epoch in which the first data block is lost. This module can be combined with any of these redundancy algorithms and overrides parts of their functionality. For example, to find the epoch in which the replication module loses the first block use `FindFirst,Replication`.

- refresh
  The refresh algorithm which is used in the system. Possible values are `NoRefresh`, which simply does not refresh data in the network or `ContinuousRefresh`, which uses the exponential backoff maintenance algorithm.

- out
  Name of the output files. Several output files will be generated which are prepended with the given filename. Typically `[name]0` is generated, which contains fill simulation state in every epoch. Additionally `[name]0.raw` is generated, which contains a raw dump of the Perl data structures of the simulator. The raw dump contains additional information that was not used for my thesis. When using the `ContinuousRefresh` module, additionally several files named `[name]lost[X]` are generated, with X being an incrementing number. One file is generated per simulation run and contains additional information from the refresh module about unusual situations that occurred during the simulation.

- shooternum
  Which shooter module is used. Possible values are 1, 2, and 3. 1 chooses a simple synthetic shooter, which kills one node per epoch. Hence, in epoch 0 all nodes are alive, in epoch 1 one node is dead, etc. Shooters 2 and 3 are the shooters using network traces for 32 and 64 bit systems respectively.

- shootercache
  Name of a cache file containing additional information for the shooter module when using a network trace. The `scripts` directory contains a Perl script to generate the cache. Using the option is only advised when running a massive amount of simulations because in the current versions the speed improvements are very small.

- kadfile
  Input file containing the network trace. The option name is historic, because for the first simulations only the Kademlia network trace was used.

- erasurecount
  Number of blocks which are combined. The option is called erasurecount for historical reasons; redundancycount would be a better name.

- numblocks
  Number of data blocks in the whole simulated network.

- numredblocks
  Number of redundancy blocks in the whole simulated network.

- numpeers
  Number of nodes in the simulated network.

- startepoch
  The epoch in which the simulation is started.

- maxepoch
  The epoch in which the simulation ends.

- stepping
  The epoch step size of the simulation. Typically this is 1, meaning that every epoch is simulated.

- repetitions
  How often the simulation is run with the specified settings.

- threads
  How many of the simulations are run simultaneously.

- max_peers_in_shooterfile
  The number of peers the network specified in the `kadfile` has. Default is 400 375 for the number of nodes in the Kademlia network trace. All other traces have a much lower number of nodes – setting this option for them saves several gigabytes of main memory per simulation.

- lostout
  When using the data maintenance algorithm presented in the paper, the simulator can output information about the blocks that were restored in each epoch. The information is output to the specified file. By default this points to /dev/null, hence the output is not saved. Note that, depending on the simulation scenario, the output file can reach a sizes well over ten gigabytes.

- first_stepsize
  This parameter sets how big the initial step size of the redundancy maintenance

algorithm is. Hence, it specifies after how many epochs a new block is checked for the first time. The default value is 1, it is checked in the next epoch.

- maxstepsize
  This parameter sets the maximal step size for the redundancy algorithm. After this limit is reached, the step size will not increase any more. The default value is 9999999999.
- speedhack
  When set to 1, the emulator skips periods where the data maintenance algorithm does not check for lost blocks. This speeds up the simulations; however, it also means that no information about the block loss is available for the skipped periods.

B. Implementation

Due to the complexity of implementing the cryptographic permission system presented in this thesis directly, the first proof of concept version of the file system has been implemented as a Perl application instead of being directly added to IgorFs. This has allowed for a much faster development cycle, because many issues complicating development could be ignored. For example, thread safety do not have to be considered which is of big concern in IgorFs. This approach also ignores concurrency issues like the concurrent access of different clients, etc. which do not have any association with the cryptographic problems but only make the implementation more difficult.

Figure B.1 on the next page shows how the proposed cryptographic extensions can be added to a system like IgorFs.

The different modules of IgorFs were already briefly discussed in Section 5.2 on page 80. In IgorFs, the file system calls which have been received from FUSE or NFS are sent to the internal messaging system. Without the new additions, these calls are directly forwarded to the File and Folder Module. The File and Folder Module either answers the requests directly or forwards them to the Directory Handler. The Directory Handler is a subsystem of the File and Folder Module that can return the block identifiers of a file system object when it is given a pathname.

When using the new cryptographic scheme, translations have to happen at several layers of the file system. File system calls are not sent directly to the File and Folder Module; instead they are passed through a translation layer that converts the requests in the external directory layout to requests in the internal directory layout. The directory handler is complemented with a cryptographic extension that can decrypt directory entries to which the node has access. It also verifies the validity of the directory structure. The required keys are stored in the key store.

B.1. The Key Store

The according data structure is called the key store. It is used by the directory handler for its operations. Section 8.6 on page 140 already mentioned that the group keys are directly stored in the file system. Actually, besides the group keys several more of the file system keys are also stored directly in the file system.

There are three different types of keys stored in the file system: root-keys, group-keys, and user-keys. Root-keys are only used by the file system superuser. Group-keys are used by group members and user-keys are used for user-operations. In practice, these keys are stored in three different locations in the file system (compare Figure 9.2 on page 148): In the internal directory layout root-keys are stored at `/.keys/rootkeys`, group-keys are

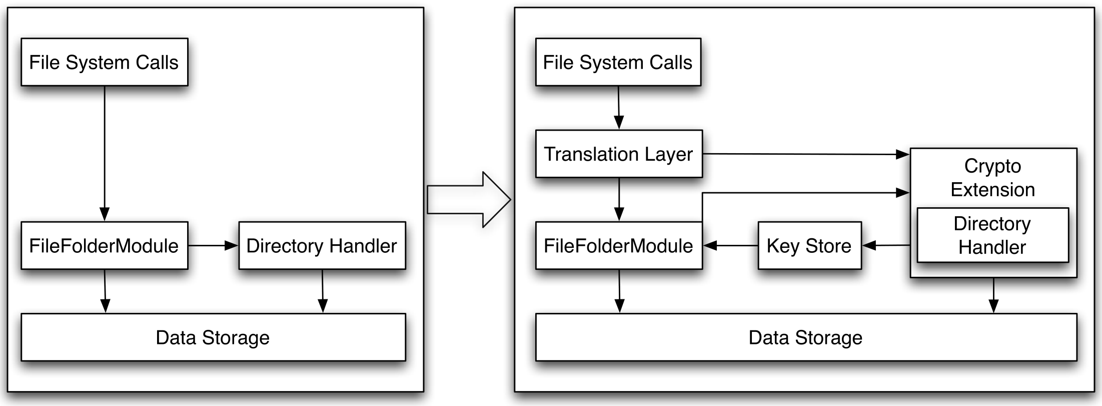


Figure B.1.: IgorFs with and without cryptographic extension

stored at `/.keys/groupkeys` and user-keys are stored at `/.keys/keys`. These directories are not present in the external directory structure and thus invisible to the user.

Section 8.6 on page 140 already established which keys are needed in the file system setup: Each user possesses three keys K_u, S_u and D_u. These keys need to be given to the user via a secure out-of-band communication method. Additionally, K_u and S_u are stored in the `/.keys/rootkeys` directory encrypted with the private key of the file system superuser. This allows the file system superuser to read, change, and sign all files of the user. It also enables the superuser to give additional copies of the keys to the user if they have been lost accidentally. Since the superuser can regenerate D_u without any additional information, D_u does not have to be stored in the file system [see Amann, 2007]. The public part of S_u is also stored unencrypted in the `/.keys/keys` folder to allow every file system user to verify user signatures.

Each group is associated with two keys: K_g and S_g. These keys are encrypted with the subset-difference encryption scheme and then stored in the `/.keys/groupkeys` folder. Additionally, the public part of S_g is stored unencrypted in the `/.keys/groupkeys` folder to allow every file system user to verify group signatures.

B.2. User Management Operations

This section describes how the user-management operations are executed with the added security structure.

There are six important operations that the file system superuser has to perform:

1. Add new users to the file system (and generate their keys).
2. Evict users from the file system (and revoke their keys).
3. Add new groups to the file system (and generate their keys).
4. Remove groups from the file system.

5. Add users to groups.
6. Remove users from groups.

B.2.1. Adding and Evicting Users

The superuser adds a new user to the file system by generating a new subset-difference key D_u, a private key K_u, and a signature key pair S_u. All these keys have to be given to the user in a secure manner using an out-of-band channel. The file system superuser also creates the directory `/.users/[uid]` and gives ownership of this folder to the new user.

The superuser evicts users from the file system by removing them from all groups of which they are a member. To do this, a new group-key is generated for each group. The key is encrypted in a way that no longer allows the evicted member to decrypt it using the subset difference algorithm. The new group-key is then used to encrypt the reference to the private part of the group-root. Furthermore, a new version of the root folder is created. The (ID, Key)-tuple for the root is also encrypted in a way that does no longer allow the evicted user u to read it with D_u.

The directory `/.users/[uid]` may, however, not be removed from the file system if it still contains files or directories because otherwise the file system could end up in an invalid state. An example for this is a directory containing directories of other users. Removing the user-root would render all subdirectories of all directories belonging to the user inaccessible. The superuser may, however, delete all files belonging to the user and all empty directories belonging to the user. The ownership of the remaining non-empty directories has to be changed – this moves them to another user-root. After that the user-root may be removed from the file system to free up space and unclutter the file system tree.

B.2.2. Adding and Removing Groups

The superuser adds a new group to the file system by generating K_g and S_g. These keys are then encrypted using the subset difference scheme and saved in the `/.keys` directory. The directories `/.groups/[gid]/private` and `/.groups/[gid]/public` are created and given the appropriate permissions.

Groups are removed by deleting the key material and removing the group-root. The group-root has to be emptied first, just as discussed in the case of the user-root.

B.2.3. Adding and Removing Group Members

When a user is added to a group, the file system superuser first regenerates K_g and S_g. The reference to `/.groups/[gid]/private` is then encrypted with the new K_g. K_g and S_g are encrypted with the subset difference scheme in a way that allows the new user to decrypt them. The new encrypted keys are then stored within the `/.keys` directory.

User removal works exactly the same way, but the new versions of K_g and S_g are encrypted in a way that no longer allows the removed user to decrypt them.

B.3. File System Operations

This section describes how the different file system operations are executed when using the translation layer in Figure B.1 to enable the cryptographic extensions on a file system. This section assumes that the file system has a working directory handler with cryptographic extensions. Hence, the client will be able to decrypt all directories to which the user has access, automatically verifies the integrity of accessed directory and automatically creates new hashes and signatures when necessary.

getattr

Arguments: *filename.*
The call returns the file attributes (size, owner, group, etc).

This call is passed through nearly unchanged by the translation layer. The translation layer only looks up the most recent version of *filename* and exchanges the location in the call.

readlink

Arguments: *filename*
The call returns the contents of a symbolic link.

This call works just like getattr. The translation layer adjusts the path of *filename* and passes the call through to the file and folder module.

getdir

Arguments: *directory name*
The call returns the contents of a directory.

In the new scheme, each directory is present up to three times in the directory structure. The translation layer executes a getdir for each of the present directories and merges the contents of all calls. The merged list is returned.

mknod

Arguments: *filename, numeric modes, numeric device*
The call returns an errno.

This call creates a new file in the file system. Files can be present at several locations in the directory structure, depending on the permission bits. The translation layer examines the numeric mode and determines the locations at which the file has to be created. The translation layer then forwards the mknod calls to the file and folder module.

mkdir

Arguments: *directory name, numeric modes*
The call returns an errno.

This call creates a new directory in the file system. Directories are created up to three times in the new directory structure. Hence, the translation layer sends up to three mkdir calls to the file and folder module, one for the user-root, one for the public part of the group-root and one for the private part of the group-root.

After that, the redirects to the new directories are created in the appropriate locations.

unlink

Arguments: *filename*
The call returns an errno.

This call removes *filename.* The translation layer translates the call to one or several unlink calls for the file and folder module.

If a directory is group-writable, and the file which shall be deleted belongs to the directory owner, it cannot be actually removed from the directory without invalidating the directory hash. In this case, the file is merely marked as deleted in the group-part of the directory structure by creating a whiteout entry. The next time the owner reads the group directory, the system removes the file from the directory hash.

rmdir

Arguments: *pathname*
The call returns an errno.

This call removes the directory at *pathname.* The translation layer translates this to the necessary number rmdir operations, one for the user-root, one for the public part of the group-root, and one for the private-part of the group-root.

Additionally, the associated redirects are also removed.

symlink

Arguments: *filename, symlink name.*
Returns an errno.

Call works exactly like mknod.

rename

Arguments: *old filename, new filename*
Returns an errno.

This changes the name of an object from *old filename* to *new filename.* Note that both the old and the new file names are full path names. The rename call cannot only change the name of the file, but also move it to an arbitrary new location within the file system.

The target of this call can either be a directory, or a file. For files, this call converts the rename operations to several rename operations depending on the number of redirects that are currently present in the file system.

For directories, this call converts the rename operations to one or several rename operations for the redirect that is pointing to the real directory locations. Note that only the redirect is moved to a new location, the actual hidden directory entries are not moved.

link

Arguments: *filename, hardlink target*
Returns an errno.

IgorFs does not support hard links because of its directory structure. As a result, hard links are also not supported in the cryptographic layer. On a system with hard link support, the implementation is the same as for the symlink operation.

chmod

Arguments: *pathname, numeric modes*
Returns an errno.

This call changes the permissions of a given object in the file system. It can only be called by the owner of a file or by the file system superuser.

When this call is executed, the locations of the references in the file system have to be changed to match the new permissions according to Tables 9.3 and 9.4. This is a very complex operation. Consider, for example, a directory that has not been world-writable and is now made world-writable. In this case, all files stored directly in the directory have to be moved to the respective owners' directories. After that, redirects to the new storage locations have to be created.

chown

Arguments: *pathname, user id, group id*
Returns an errno.

This call changes the user and/or the group of a given object. The owner of a file can call it to change the group of the file to another group to which she belongs. The file system superuser can call it to change the owner and the group of the file arbitrarily.

Along chmod, this is one of the most complex operations because, to change the ownership of a file or directory, the file or directory has to be moved to a different user- or group-root.

When chown is called, the file or directory is moved to its new location according to Tables 9.3 and 9.4.

Special care has to be taken for directories. The contents of the directory still belong to the original owners because a chown call only changes the ownership of the directory, not of the directory contents. Thus, the files and directories in the directory have to be moved back to the user-roots of their respective owners. Redirects have to be created in the directory whose ownership was changed to make them accessible – this is also done according to Tables 9.3 and 9.4.

truncate

Arguments: *filename, offset*
Returns an errno.

This call truncates the named file at the given *offset*.

The translation layer looks up all locations at which the file references are stored in the hidden directory structure. The truncate call is then executed for all the reference locations to which the current user has write permissions.

For example if a file is group-writable it is present one time in the user-root of the owner and one time in the group-root of the group (see Tables 9.3 and 9.4). If the file is truncated by the owner, the references at both locations will be updated. If the file is truncated by a group member other than the owner, only the references in the group-root will be updated. The most current version can be determined using the version number of the file.

read

Arguments: *filename, size, offset*
Returns data or an errno.

This call attempts to read *size* bytes from *filename* starting at *offset*. The translation layer looks up all locations at which the file references are stored in the hidden directory structure. The read call is then executed for the more recent references.

write

Arguments: *filename, buffer, offset*
Returns an errno.

This call tries to (over)write a portion of *filename*. The translation layer looks up all locations at which the file references are stored in the hidden directory structure and executes the call just like for truncate.

C. CryptFs

This appendix shows how to use the proof-of-concept file system implementation named CryptFs, which was developed for this thesis.

C.1. System Requirements

The minimal version of Perl required to run the file system is Perl 5.10.0. The file system also requires a correctly configured Fuse installation.

The following CPAN Perl modules are required in addition to the Perl core modules:

- Moose
- MooseX::Runnable
- MooseX::Singleton
- MooseX::Getopt
- MooseX::Log::Log4perl
- Digest::SHA1
- Perl6::Slurp
- Crypt::CBC
- Crypt::OpenSSL::DSA
- Data::UUId

The module Crypt::Subset::Subscribe also has to be installed; this module was also developed by for this thesis and is available together with the file system prototype.

C.2. Options

The file system is started by running the Perl script named runner.pl. When started without options, the script will show an overview over the possible command line options:

```
usage: runner.pl [-?] [long options...]
    -? --usage --help  Prints this usage information.
    --mountpoint       Mountpoint for the file system
```

```
  --basedir          Storage directory for file system data chunks
  --debug            Turn on fuse debugging. Default: 0
  --uid              User-id of this user inside of the file system.
                     Default: 10001
  --gids             Group-ids of this user inside of the file system.
                     Default: 10000 109 10000 10001 10004
  --encryption       Use encryption? Default: 0
  --redirect         Use redirect layer? Default: 0
  --userkey          Path to user key file
  --rootkey          Path to file system subset difference root key
```

The `-?` and other help options are self-explanatory, the other options are explained in more detail below.

- <u>mountpoint</u>
  This option sets the mountpoint, where the file system will be attached.
- <u>basedir</u>
  This option sets the storage directory for the file system data chunks. All data and meta-data of the file system will be stored in this directory. Note that this storage directory can also be placed into a distributed storage system, e.g. on an IgorFs volume to extend it with a permission system.
- <u>debug</u>
  This option enables fuse debugging. With fuse debugging enabled, each file system operation that is received from fuse will be output to the console.
- <u>uid</u>
  This option sets the user-ID of the current user inside of the file system. Please note that the user-ID in the file system does not have to be equal to the current effective user ID. Each user can create an own, dedicated file system where the user assumes the role of the file system superuser. In this case the uid option is used to set the user ID to 0.
- <u>gid</u>
  This option sets the group IDs of the current users. It works the same as the `uid` option.
- <u>encryption</u>
  When this option is set the file system encryption algorithm is activated. Without the option the file system will not use encryption or signatures or integrity hashing. This option can be used to measure the overhead introduced by the encryption layer.
- <u>redirect</u>
  When this option is set the file system redirect layer is activated. This option

requires the encryption option to be activated. When the encryption is turned on but the redirect layer is deactivated, the hidden underlying directory structure is exposed and can be analyzed. This option can also be used to measure the overhead introduced by the redirect layer.

- userkey
  The path to the current user key file. This key file is needed to access non-world-readable files in file systems with encryption enabled. When a new file system is created this option must not to be set.

- rootkey
  The path to the file system subset difference root key. This key file is required to enable the file system superuser to create new users and groups. When a new file system is created this option must not be set. When the current user ID is different from 0 i.e. for users that are not the superuser, this option must not be set.

C.3. Creating a New Encrypted File System

This section will show the necessary steps to set up a basic encrypted file system.

The first step is to start the file system with encryption enabled, but without activating the new directory scheme:

```
$ perl runner.pl --basedir /export/store --mountpoint /export/mnt \
--encryption 1 --redirect 0 --uid 0 --gids 0
```

After that the file system can be accessed:

```
$ cd /export/mnt/
$ ls -l
drwxr-xr-x 1 root root 0 Feb 11 11:16 groupkeys
drwxr-xr-x 1 root root 0 Feb 11 11:16 groups
drwxr-xr-x 1 root root 0 Feb 11 11:16 keys
dr-xr-xr-x 1 root root 0 Feb 11 11:16 proc
drwx------ 1 root root 0 Feb 11 11:16 rootkeys
drwxr-xr-x 1 root root 0 Feb 11 11:16 users
```

All the directories that were presented in this thesis (`groups, users, keys`) are presented. There are a few additional directories which are used for easier management of the file system, but that are not strictly necessary (`groupkeys, rootkeys, proc`). `groupkeys` contains the group cryptographic keys. They are stored separately from the user cryptographic keys in `keys` to make it easier to distinguish between them. `rootkeys` is a directory only accessible by the file system superuser, which contains all keys in an unencrypted form to allow the superuser to access all files and directories. The `proc` directory contains special files which allow us to manage the file system.

A new user and group will now be added to the file system using the `proc` files:

```
$ cd proc
$ ls -l
-rwxrwxrwx 1 root root 30 Feb 11 11:16 addUserToGroup
-rwxrwxrwx 1 root root 30 Feb 11 11:16 createGroupById
-rwxrwxrwx 1 root root 30 Feb 11 11:16 createUserById
```

The special file `addUserToGroup` can be used to add existing users to existing groups. `createGroupById` can be used to create new groups and `createUserById` can be used to create new users.

A new group with gid 1 and a new user with uid 1 can be created with the following commands:

```
$ echo 1 > createUserById
$ echo 1 > createGroupById
```

Finally the user can be added to the group by running:

```
$ echo "1 to 1" > addUserToGroup
```

Now the `keys` directory will contain all the respective keys:

```
$ cd ../keys
$ ls -l
-rw------- 1 root root 15985 Feb 11 11:16 0
-rw------- 1 root root 15999 Feb 11 11:17 1
-rw------- 1 root root   991 Feb 11 11:16 group-0
-rw------- 1 root root  1068 Feb 11 11:19 group-1
-rw------- 1 root root   694 Feb 11 11:16 sdtree
```

The key files have now to be copied to a location outside of the file system. Without them, the file system can not be mounted again. The file `0` contains the private file system key of the superuser. The file `1` contains the private file system key of the user that has just been generated. The file `sdtree` contains the secret subset difference main key of the file system superuser. After copying the files, the file system can be stopped.

```
$ cp 0 1 sdtree ../..
$ cd ../..
$ fusermount -u mnt
```

Now the file system can be started with the directory layer. To start it for the superuser the user- and group-ID 0 is used:

```
$ perl runner.pl --basedir /export/store --mountpoint /export/mnt \
--encryption 1 --redirect 0 --uid 0 --gids 0 --rootkey /export/sdtree \
--userkey /export/ba/0
```

To start it for the user with the user-ID 1 which was previously created the following command like can be used:

```
$ perl runner.pl --basedir /export/store --mountpoint /export/mnt \
--encryption 1 --redirect 0 --uid 1 --gids 1 ---userkey /export/1
```

List of Figures

List of Tables

Bibliography

–∼ **A** ∼–

AACS. *Advanded Access Content System (AACS): Introduction and Common Cryptographic Elements.* AACS LA, February 2006. Revision 0.91.

Michael **Abd-El-Malek**, William V. **Courtright**, II, Chuck **Cranor**, Gregory R. **Ganger**, James **Hendricks**, Andrew J. **Klosterman**, Michael **Mesnier**, Manish **Prasad**, Brandon **Salmon**, Raja R. **Sambasivan**, Shafeeq **Sinnamohideen**, John D. **Strunk**, Eno **Thereska**, Matthew **Wachs**, and Jay J. **Wylie**. "Ursa Minor: Versatile Cluster-Based Storage". In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies*, FAST '05, pp. 59–72. 2005.

Szymon **Acedański**, Supratim **Deb**, Muriel **Médard**, and Ralf **Koetter**. "How Good is Random Linear Coding Based Distributed Networked Storage". In *Proceedings of the First Workshop on Network Coding, Theory and Applications*, NetCod '05. 2005.

Atul **Adya**, William J. **Bolosky**, Miguel **Castro**, Gerald **Cermak**, Ronnie **Chaiken**, John R. **Douceur**, Jon **Howell**, Jacob R. **Lorch**, Marvin **Theimer**, and Roger P. **Wattenhofer**. "Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment". *SIGOPS Operating Systems Review*, Volume 36(SI):pp. 1–14, 2002.

Johanna **Amann**. "Secure Asynchronous Change Notifications for a Distributed File System". Diplomarbeit, Chair for Network Architectures, Technische Universität München, 2007.

Johanna **Amann**. "Perl Distributed Network Redundancy Simulator". 2011a. *http://www.so.in.tum.de/~ja/redsim*.

Johanna **Amann**. "Posix ACL File System Prototype". 2011b. *http://www.so.in.tum.de/~ja/soft*.

Mattias **Amnefelt** and Johanna **Svenningsson**. "Keso - A Scalable, Reliable and Secure Read/Write Peer-to-Peer File System". Master's thesis, KTH/Royal Institute of Technology, Stockholm, May 2004.

Gillian **Andrews**. "Bloggers Vs. "AOL Users": A Digital Divide of Use and Literacy". In *Proceedings of the Fourty-Third Hawaii International Conference on System Sciences*, HICSS '10, pp. 1–10. 2010.

Stephanos **Androutsellis-Theotokis** and Diomidis **Spinellis**. "A Survey of Peer-to-Peer Content Distribution Technologies". *ACM Computing Surveys*, Volume 36:pp. 335–371, December 2004.

Nuttapong **Attrapadung** and Hideki **Imai**. "Attribute-Based Encryption Supporting Direct/Indirect Revocation Modes". In *Proceedings of the Twelfth IMA International Conference on Cryptography and Coding*, Volume 5921 of *Lecture Notes in Computer Science*, pp. 278–300. 2009.

AVT Traces. "Repository of Availability Traces". *http://www.cs.uiuc.edu/homes/pbg/availability*.

–∼ B ∼–

Maurice J. **Bach**. *The Design of the UNIX Operating System.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986. ISBN 0-13-201799-7.

Gal **Badishi**, Germano **Caronni**, Idit **Keidar**, Raphael **Rom**, and Glenn **Scott**. "Deleting Files in the Celeste Peer-to-Peer Storage System". *Journal of Parallel and Distributed Computing*, Volume 69(7):pp. 613–622, 2009.

Mehmet **Bakkaloglu**, Jay J. **Wylie**, Chenxi **Wang**, and Gregory R. **Ganger**. "On Correlated Failures in Survivable Storage Systems". Technical Report CMU-CS-02-129, Carnegie Mellon University, May 2002.

James **Bamford**. *The Puzzle Palace: Inside the National Security Agency, America's Most Secret Intelligence Organization.* Penguin Books, 1983. ISBN 0-140-06748-4.

Salman A. **Baset** and Henning G. **Schulzrinne**. "An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol". In *Proceedings of the Twenty-Fifth IEEE International Conference on Computer Communications*, INFOCOM '06. 2006.

Amos **Beimel**. "Secure Schemes for Secret Sharing and Key Distribution". Ph.D. thesis, Israel Institute of Technology, Technion, Haifa, Israel, 1996.

Piotr **Berman**, Marek **Karpinski**, and Yakov **Nekrich**. "Optimal Trade-Off for Merkle Tree Traversal". In Joaquim **Filipe**, Helder **Coelhas**, and Monica **Saramago** (editors), *E-business and Telecommunication Networks*, Volume 3 of *Communications in Computer and Information Science*, pp. 150–162. 2007.

John **Bethencourt**, Amit **Sahai**, and Brent **Waters**. "Ciphertext-Policy Attribute-Based Encryption". In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pp. 321–334. 2007.

Stefan **Betschon**. "Schlacht der Cyber-Utopisten". *Neue Züricher Zeitung*, (28):p. 58, 3rd February 2011.

Ranjita **Bhagwan**, Kiran **Tati**, Yu-Chung **Cheng**, Stefan **Savage**, and Geoffrey M. **Voelker**. “Total Recall: System Support for Automated Availability Management”. In *Proceedings of the First Symposium on Networked Systems Design and Implementation*. 2004.

David **Bindel**, Yan **Chen**, Patrick **Eaton**, Dennis **Geels**, Ramakrishna **Gummadi**, Sean **Rhea**, Hakim **Weatherspoon**, Westly **Weimer**, Christopher **Wells**, Ben **Zhao**, and John **Kubiatowicz**. “OceanStore: An Extremely Wide-Area Storage System”. Technical Report CSD-00-1102, Berkeley, CA, USA, 2002.

Andreas **Binzenhöfer** and Kenji **Leibnitz**. “Estimating Churn in Structured P2P Networks”. In *Managing Traffic Performance in Converged Networks*, Volume 4516 of *Lecture Notes in Computer Science*, pp. 630–641. 2007.

Charles **Blake** and Rodrigo **Rodrigues**. “High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two”. In *Proceedings of the Ninth Conference on Hot Topics in Operating Systems*, HotOs IX. 2003.

Alexandra **Boldyreva**, Vipul **Goyal**, and Virendra **Kumar**. “Identity-based Encryption with Efficient Revocation”. In *Proceedings of the Fifteenth ACM Conference on Computer and Communications Security*, CCS ’08, pp. 417–426. 2008.

William J. **Bolosky**, John R. **Douceur**, David **Ely**, and Marvin **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*, SIGMETRICS ’00, pp. 34–43. 2000.

Dan **Boneh** and Matthew K. **Franklin**. “Identity-Based Encryption from the Weil Pairing”. In *Advances in Cryptology – CRYPTO 2001*, Volume 2139 of *Lecture Notes in Computer Science*, pp. 213–229. 2001.

Dan **Boneh**, Craig **Gentry**, and Brent **Waters**. “Collusion Resistant Broadcast Encryption with Short Ciphertexts and Private Keys”. In *Advances in Cryptology – CRYPTO 2005*, Volume 3621 of *Lecture Notes in Computer Science*, pp. 258–275. 2005.

Dhruba **Borthakur**. “HDFS Architecture Guide”. 2008.

Peter J. **Braam et al.** “The Lustre Storage Architecture”. August 2004. *ftp://ftp.uni-duisburg.de/Linux/filesys/Lustre/lustre.pdf*.

Mark R. **Brown**, Karen N. **Kolling**, and Edward A. **Taft**. “The Alpine File System”. *ACM Transactions on Computer Systems*, Volume 3:pp. 261–293, November 1985.

Billy Bob **Brumley** and Nicola **Tuveri**. “Remote Timing Attacks are Still Practical”. Cryptology ePrint Archive, Report 2011/232, May 2011. *http://eprint.iacr.org/2011/232*.

Markus **Bucher**. “Evaluation of Current P2P-SIP Proposals with Respect to the Igor/SSR API”. Diplomarbeit, Lehrstuhl für Netzarchitekturen und Netzdienste, Technische Universität München, 2009.

Jean-Michel **Busca**, Fabio **Picconi**, and Pierre **Sens**. “Pastis: A Highly-Scalable Multi-user Peer-to-Peer File System”. In *Euro-Par 2005 Parallel Processing*, Volume 3648 of *Lecture Notes in Computer Science*, pp. 1173–1182. 2005.

Randy **Bush**. “FidoNet: Technology, Tools, and History”. *Communications of the ACM*, Volume 36:pp. 31–35, August 1993.

John W. **Byers**, Michael **Luby**, Michael **Mitzenmacher**, and Ashutosh **Rege**. “A Digital Fountain Approach to Reliable Distribution of Bulk Data”. *SIGCOMM Computer Communication Review*, Volume 28(4):pp. 56–67, 1998.

–∼ C ∼–

Brent **Callaghan**, Brian **Pawlowski**, and Peter **Staubach**. “NFS Version 3 Protocol Specification”. RFC 1813, Internet Engineering Task Force, June 1995.

Mingming **Cao**, Theodore Y. **Ts’o**, Badari **Pulavarty**, and Suparna **Bhattacharya**. “State of the Art: Where we are with the Ext3 filesystem”. OLS ’05. 2005.

Rémy **Card**, Theodore **Ts’o**, and Stephen **Tweedie**. “Design and Implementation of the Second Extended Filesystem”. In *Proceedings of the First Durch International Symposium on Linux*. 1986.

Germano **Caronni**, Raphael J. **Rom**, and Glenn C. **Scott**. “Celeste: An Automatic Storage System”. 2005. White Paper.

Giuseppe **Cattaneo**, Luigi **Catuogno**, Aniello Del **Sorbo**, and Pino **Persiano**. “The Design and Implementation of a Transparent Cryptographic File System for UNIX”. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pp. 199–212. 2001.

Vineet **Chadha** and Renato J. **Figueiredo**. “ROW-FS: A User-Level Virtualized Redirect-on-Write Distributed File System for Wide Area Applications”. In *Proceedings of the Fourteenth International Conference on High Performance Computing*, HiPC ’07, pp. 21–34. 2007.

Fay **Chang**, Jeffrey **Dean**, Sanjay **Ghemawat**, Wilson C. **Hsieh**, Deborah A. **Wallach**, Mike **Burrows**, Tushar **Chandra**, Andrew **Fikes**, and Robert E. **Gruber**. “Bigtable: A Distributed Storage System for Structured Data”. *ACM Transactions on Computer Systems*, Volume 26:pp. 4:1–4:26, June 2008.

Guihai **Chen**, Tongqing **Qiu**, and Fan **Wu**. “Insight Into Redundancy Schemes in DHTs”. *The Journal of Supercomputing*, Volume 43:pp. 183–198, 2008a.

Yong **Chen**, L. M. **Ni**, Mingyao **Yang**, and P. **Mohapatra**. “CoStore: a Serverless Distributed File System Utilizing Disk Space on Workstation Clusters”. In *Proceedings of the Twenty-First IEEE International Performance, Computing, and Communications Conference*, PCC ’02, pp. 393–398. 2002a.

Yong **Chen**, Lionel M. **Ni**, and Mingyao **Yang**. “CoStore: A Storage Cluster Architecture Using Network Attached Storage Devices”. In *Proceedings of the Ninth International Conference on Parallel and Distributed Systems*, ICPADS ’02, pp. 301–306. 2002b.

Zhijia **Chen**, Yang **Chen**, Chuang **Lin**, V. **Nivargi**, and Pei **Cao**. “Experimental Analysis of Super-Seeding in BitTorrent”. In *Proceedings of the 2008 IEEE International Conference on Communications*, ICC ’08, pp. 65–69. 2008b.

Thibault **Cholez**, Isabelle **Chrisment**, and Olivier **Festor**. “Evaluation of Sybil Attacks Protection Schemes in KAD”. In Ramin **Sadre** and Aiko **Pras** (editors), *Scalability of Networks and Services*, Volume 5637 of *Lecture Notes in Computer Science*, pp. 70–82. 2009.

Ian **Clarke**, Scott G. **Miller**, Theodore W. **Hong**, Oskar **Sandberg**, and Brandon **Wiley**. “Protecting Free Expression Online with Freenet”. *IEEE Internet Computing*, Volume 6(1):pp. 40–49, 2002.

Ian **Clarke**, Oskar **Sandberg**, Brandon **Wiley**, and Theodore **w. Hong**. “Freenet: A Distributed Anonymous Information Storage and Retrieval System”. In *Proceedings of the International Workshop on Design Issues in Anonymity and Unobservability*, Volume 2009 of *Lecture Notes in Computer Science*, pp. 46–64. 2001.

Richard **Clayton**. “The Rising Tide: DDoS by Defective Designs and Defaults”. In *Proceedings of the Second Conference on Steps to Reducing Unwanted Traffic on the Internet*, SRUTI ’06, pp. 1–6. 2006.

Bram **Cohen**. “Incentives Build Robustness in BitTorrent”. In *Proceedings of the First Workshop of Economics of Peer-to-Peer Systems*. 2003.

Landon P. **Cox** and Brian D. **Noble**. “Samsara: Honor Among Thieves in Peer-to-Peer Storage”. In *Proceedings of the Nineteenth ACM Symposium on Operating systems Principles*, SOSP ’03, pp. 120–132. 2003.

Curt **Cramer**, Kendy **Kutzner**, and Thomas **Fuhrmann**. “Bootstrapping Locality-Aware P2P Networks”. In *Proceedings of the Twelfth IEEE International Conference on Networks*, ICON ’04, pp. 357–361. 2004.

–∼ **D** ∼–

Frank **Dabek**, Emma **Brunskill**, M. Frans **Kaashoek**, David **Karger**, Robert **Morris**, Ion **Stoica**, and Hari **Balakrishnan**. “Building Peer-to-Peer Systems with Chord, a Distributed Lookup Service”. In *Proceedings of the Eighth IEEE Workshop on Hot Topics in Operating Systems*, HotOS VIII, pp. 71–76. 2001a.

Frank **Dabek**, M. Frans **Kaashoek**, David **Karger**, Robert **Morris**, and Ion **Stoica**. "Wide-Area Cooperative Storage with CFS". In *Proceedings of the Eighteenth ACM Symposium on Operating System Principles*, SOSP '01. 2001b.

Frank **Dabek**, Jinyang **Li**, Emil **Sit**, James **Robertson**, M. Frans **Kaashoek**, and Robert **Morris**. "Designing a DHT for Low Latency and High Throughput". In *Proceedings of the First Symposium on Networked Systems Design and Implementation*, NSDI '04. 2004.

Frank **Dabek**, Ben **Zhao**, Peter **Druschel**, John **Kubiatowicz**, and Ion **Stoica**. "Towards a Common API for Structured Peer-to-Peer Overlays". In *Proceedings of the Second International Workshop on Peer-to-Peer Systems*, IPTPS'03. 2003.

Paolo **D'Arco** and Alfredo **De Santis**. "Optimising SD and LSD in Presence of Non-uniform Probabilities of Revocation". In *Information Theoretic Security*, Volume 4883 of *Lecture Notes in Computer Science*, pp. 46–64. Springer Berlin / Heidelberg, 2009.

J. **Davidson**, W. **Hathaway**, J. **Postel**, N. **Mimno**, R. **Thomas**, and D. **Walden**. "The Arpanet Telnet Protocol: Its Purpose, Principles, Implementation, and Impact on Host Operating System Design". In *Proceedings of the Fifth Symposium on Data Communications*, SIGCOMM '77, pp. 4.10–4.18. 1977.

Cécile **Delerablée**. "Identity-Based Broadcast Encryption with Constant Size Ciphertexts and Private Keys". In *Advances in Cryptology – ASIACRYPT 2007*, Volume 4833 of *Lecture Notes in Computer Science*, pp. 200–215. 2007.

Pengfei **Di**, Kendy **Kutzner**, and Thomas **Fuhrmann**. "Providing KBR Service for Multiple Applications". In *Proceedings of the Seventh International Workshop on Peer-to-Peer Systems*, IPTPS '08. 2008.

Whitfield **Diffie** and Martin E. **Hellman**. "New Directions in Cryptography". *IEEE Transactions on Information Theory*, Volume 22(6):pp. 644–654, November 1976.

Alexandros G. **Dimakis**, Brighten **Godfrey**, Martin J. **Wainwright**, and Kannan **Ramchandran**. "Network Coding for Distributed Storage Systems". In *Proceedings of the Twenty-Sixth IEEE International Conference on Computer Communications*, INFOCOM, pp. 2000–2008. 2007.

Alexandros G. **Dimakis**, P. Brighten **Godfrey**, Yunnan **Wu**, Martin J. **Wainwright**, and Kannan **Ramchandran**. "Network Coding for Distributed Storage Systems". *IEEE Transactions on Information Theory*, Volume 56:pp. 4539–4551, September 2010.

Jeremy **Dion**. "The Cambridge File Server". *SIGOPS Operating Systems Review*, Volume 14:pp. 26–35, October 1980.

John R. **Douceur**. "The Sybil Attack". In *Peer-to-Peer Systems*, Volume 2429 of *Lecture Notes in Computer Science*, pp. 251–260. 2002.

John R. **Douceur**, Atul **Adya**, Josh **Benaloh**, William J. **Bolosky**, and Gideon **Yuval**. “A Secure Directory Service based on Exclusive Encryption”. In *Proceedings of the Eighteenth Annual Computer Security Applications Conference*, ACSAC ’02. 2002a.

John R. **Douceur**, Atul **Adya**, William J. **Bolosky**, Dan **Simon**, and Marvin **Theimer**. “Reclaiming Space from Duplicate Files in a Serverless Distributed File System”. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, ICDCS ’02, pp. 617–624. 2002b.

John R. **Douceur** and Jon **Howell**. “Distributed Directory Service in the Farsite File System”. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation*, OSDI ’06, pp. 321–334. 2006.

Roland C. **Dowdeswell** and John **Ioannidis**. “The CryptoGraphic Disk Driver”. In *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference*, pp. 179–186. 2003.

DragonFly. “DragonFly 2.11 General Commands Manual – LN(1)”. 2009. *http://leaf.dragonflybsd.org/cgi/web-man?command=ln&section=ANY*.

Dropbox. *http://www.dropbox.com*.

Arnold I. **Dumey**. “Indexing for Rapid Access Memory Systems”. *Computers and Automation*, Volume 5:pp. 6–9, December 1956.

Alessandro **Duminuco**. “Data redundancy and maintenance for peer-to-peer file backup systems”. Ph.D. thesis, Télécom ParisTech, 10 2009.

Alessandro **Duminuco** and Ernst **Biersack**. “A Practical Study of Regenerating Codes for Peer-to-Peer Backup Systems”. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, ICDCS ’09, pp. 376–384. 2009.

Alessandro **Duminuco** and Ernst **Biersack**. “Hierarchical Codes: A Flexible Trade-off for Erasure Codes in Peer-to-Peer Storage Systems”. *Peer-to-Peer Networking and Applications*, Volume 3:pp. 52–66, 2010.

Alessandro **Duminuco** and Ernst W **Biersack**. “Hierarchical Codes: How to Make Erasure Codes Attractive for Peer-to-Peer Storage Systems”. In *Proceedings of the 8th IEEE International Conference on Peer-to-Peer Computing*, P2P’08. 2008.

Ray **Duncan**. *Advanced MS-DOS Programming: the Microsoft Guide for Assembly Language and C Programmers*. Microsoft Press, Redmond, Washington, USA, Second Edition, 1988. ISBN 1-55615-157-8.

Roy **Duncan**. “Design Goals and Implementation of the new High Performance File System”. *Microsoft Systems Journal*, Volume 4(5):pp. 1–13, September 1989.

–∼ E ∼–

Donald E. **Eastlake** and Paul E. **Jones**. “US Secure Hash Algorithm 1 (SHA1)”. RFC 3174, Internet Engineering Task Force, September 2001.

Jörg **Eberspächer** and Rüdiger **Schollmeier**. “First and Second Generation of Peer-to-Peer Systems”. In Ralf **Steinmetz** and Klaus **Wehrle** (editors), *Peer-to-Peer Systems and Applications*, Volume 3485 of *Lecture Notes in Computer Science*, pp. 35–56. Springer Berlin / Heidelberg, 2005.

Claudia **Eckert**. *IT-Sicherheit.* Oldenbourg, München, fourth Edition, 2006. ISBN 3-486-57851-0.

Dave **Eichmann** and Diana **Harris**. “EBB and Flow: An Electronic Bulletin Board”. In *Proceedings of the Ninth Annual ACM SIGUCCS Conference on User Services*, SIGUCCS ’81, pp. 203–207. 1981.

Taher **El Gamal**. “A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms”. In *Advances in Cryptology – CRYPTO ’84*, Volume 196 of *Lecture Notes in Computer Science*, pp. 10–18. 1985.

James H. **Ellis**. “The History of Non-Secret Encryption”. *Cryptologia*, Volume 23(3):pp. 267–273, 1999.

Benedikt **Elser**, Andreas **Förschler**, and Thomas **Fuhrmann**. “Spring for Vivaldi – Orchestrating Hierarchical Network Coordinates”. In *Proceedings of the Tenth IEEE International Conference on Peer-to-Peer Computing*, P2P’10. 2010.

Marius Aamodt **Eriksen** and J. Bruce **Fields**. “Mapping Between NFSv4 and Posix Draft ACLs”. Internet Draft, February 2005. *http://tools.ietf.org/id/draft-ietf-nfsv4-acl-mapping-03.txt*.

Andrey Petrovych **Ershov**. “On Programming of Arithmetic Operations”. *Communications of the ACM*, Volume 1:pp. 3–6, August 1958.

–∼ F ∼–

Sam **Falkner** and Lisa **Week**. “NFS Version 4 ACLs”. Internet Draft, February 2006. *http://tools.ietf.org/id/draft-ietf-nfsv4-acls-00.txt*.

FIPS 180–2. “Specifications for the Secure Hash Standard”. Federal Information Processing Standards Publication 180–2, U.S. Department of Commerce/National Institute of Standards and Technology (NIST), August 2002.

FIPS 186–2. “Digital Signature Standard (DSS)”. Federal Information Processing Standards Publication 186–2, U.S. Department of Commerce/National Institute of Standards and Technology (NIST), January 2000.

FIPS 197. "Specification for the Advanced Encryption Standard (AES)". Federal Information Processing Standards Publication 197, U.S. Department of Commerce/National Institute of Standards and Technology (NIST), November 2001.

Alessandro **Forin** and Gerald R. **Malan**. "An MS-DOS file system for UNIX". In *Proceedings of the USENIX Winter 1994 Technical Conference*, pp. 26–43. 1994.

Freenet. "The Freenet Project: Understand Freenet". accessed 6th May 2011 . *http://freenetproject.org/understand.html*.

Marek **Fridrich** and William **Older**. "Helix: The Architecture of the XMS Distributed File system". *IEEE Software*, Volume 2:pp. 21–29, May 1985.

Kevin **Fu**, M. Frans **Kaashoek**, and David **Mazières**. "Fast and Secure Distributed Read-Only File System". In *Proceedings of the Fourth Conference on Symposium on Operating System Design & Implementation*, OSDI '00. 2000.

Kevin **Fu**, Seny **Kamara**, and Tadayoshi **Kohno**. "Key Regression: Enabling Efficient Key Distribution for Secure Distributed Storage". In *Proceedings of the Network and Distributed Systems Security Symposium*, NDSS '06. 2006.

Kevin E. **Fu**. "Group Sharing and Random Access in Cryptographic Storage File Systems". Master's thesis, MIT, Cambridge, June 1999.

Fuse. "File System in Userspace". *http://fuse.sourceforge.net*.

–∼ G ∼–

Sanjay **Ghemawat**, Howard **Gobioff**, and Shun-Tak **Leung**. "The Google File System". *SIGOPS Operating Syststems Review*, Volume 37:pp. 29–43, October 2003.

Ali **Ghodsi**. "Distributed k-ary System: Algorithms for Distributed Hash Tables". PhD dissertation, KTH—Royal Institute of Technology, Stockholm, Sweden, October 2006.

Garth A. **Gibson**, David F. **Nagle**, Khalil **Amiri**, Jeff **Butler**, Fay W. **Chang**, Howard **Gobioff**, Charles **Hardin**, Erik **Riedel**, David **Rochberg**, and Jim **Zelenka**. "A Cost-Effective, High-Bandwidth Storage Architecture". *SIGOPS Operating Systems Review*, Volume 32(5):pp. 92–103, 1998.

Garth A. **Gibson** and Rodney **Van Meter**. "Network attached storage architecture". *Communications of the ACM*, Volume 43:pp. 37–45, November 2000.

Christos **Gkantsidis** and Pablo **Rodriguez**. "Network Coding for Large Scale Content Distribution". In *Proceedings of the Twenty-Fourth Annual Joint Conference of the IEEE Computer and Communications Societies*, Volume 4 of *INFOCOM '05*, pp. 2235–2245. 2005.

Gnutella. “The Annotated Gnutella Protocol Specification v0.4”. Document revision 1.6, The Gnutella Developer Forum (GDF). *http://rfc-gnutella.sourceforge.net/developer/stable/index.html*.

P. Brighten **Godfrey**, Scott **Shenker**, and Ion **Stoica**. “Minimizing Churn in Distributed Systems”. *SIGCOMM Computer Communication Review*, Volume 36:pp. 147–158, August 2006.

Eu-jin **Goh**, Hovav **Shacham**, Nagendra **Modadugu**, and Dan **Boneh**. “Sirius: Securing Remote Untrusted Storage”. In *Proceedings of the Network and Distributed Systems Security Symposium 2003*, NDSS ’03, pp. 131–145. 2003.

Vipul **Goyal**, Omkant **Pandey**, Amit **Sahai**, and Brent **Waters**. “Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data”. In *Proceedings of the Thirteenth ACM Conference on Computer and Communications Security*, CCS ’06, pp. 89–98. 2006.

Dominik **Grolimund**. “Wuala - A Distributed File System”. 2007. Google Tech Talk: *http://www.youtube.com/watch?v=3xKZ4KGkQY8*.

Dominik **Grolimund**, Luzius **Meisser**, Stefan **Schmid**, and Roger **Wattenhofer**. “Cryptree: A Folder Tree Structure for Cryptographic File Systems”. In *Proceedings of the Twenty-Fifth IEEE Symposium on Reliable Distributed Systems*, pp. 189–198. 2006a.

Dominik **Grolimund**, Luzius **Meisser**, Stefan **Schmid**, and Roger **Wattenhofer**. “Havelaar: A Robust and Efficient Reputation System for Active Peer-to-Peer Systems”. In *First Workshop on the Economics of Networked Systems*, NetEcon ’06. 2006b.

Andreas **Grünbacher**. “POSIX Access Control Lists on Linux”. In *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference*, pp. 259–272. 2003.

Saikat **Guha**, Neil **Daswani**, and Ravi **Jain**. “An Experimental Study of the Skype Peer-to-Peer VoIP System”. In *Proceedings of the Fifth International Workshop on Peer-to-Peer Systems*, IPTPS ’06.

P. Krishna **Gummadi**, Ramakrishna **Gummadi**, Steve **Gribble**, Sylvia **Ratnasamy**, Scott **Shenker**, and Ion **Stoica**. “The Impact of DHT Routing Geometry on Resilience and Proximity”. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGOMM’03, pp. 381–394. 2003.

Vipul **Gupta**, Sumit **Gupta**, Sheueling **Chang**, and Douglas **Stebila**. “Performance Analysis of Elliptic Curve Cryptography for SSL”. In *Proceedings of the First ACM Workshop on Wireless Security*, WiSE ’02, pp. 87–94. 2002.

Stefan **Götz**, Simon **Rieche**, and Klaus **Wehrle**. “Selected DHT Algorithms”. In Ralf **Steinmetz** and Klaus **Wehrle** (editors), *Peer-to-Peer Systems and Applications*, Volume 3485 of *LNCS*, pp. 95–117. 2005.

–∼ H ∼–

Dani **Halevy** and Adi **Shamir**. “The LSD Broadcast Encryption Scheme”. In *Advances in Cryptology – CRYPTO 2002*, Volume 2442 of *Lecture Notes in Computer Science*, pp. 47–60. 2002.

Steven M. **Hancock**. *Tru64 UNIX File System Administration Handbook*. Digital Press, Newton, MA, USA, 2001. ISBN 1-55558-227-3.

Matthew **Harren**, Joseph M. **Hellerstein**, Ryan **Huebsch**, Boon Thau **Loo**, Scott **Shenker**, and Ion **Stoica**. “Complex Queries in DHT-based Peer-to-Peer Networks”. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS ’01, pp. 242–259. 2002.

Bryce **Harrington**, Aurelien **Charbon**, Tony **Reix**, Vincent **Roqueta**, J. Bruce **Fields**, Trond **Myklebust**, Suresh **Jayaraman**, Jeff **Needle**, and Berry **Marson**. “NFSv4 Test Project”. In *Proceedings of the 2006 Linux Symposium*, pp. 268–285. 2006.

HDFS. “HDFS Permissions Guide”. February 2010. *http://hadoop.apache.org/common/docs/r0.20.2/hdfs_permissions_guide.html*.

Frank E. **Heart**, Robert E. **Kahn**, Severo M. **Ornstein**, Will R. **Crowther**, and Dave C. **Walden**. “The Interface Message Processor for the ARPA Computer Network”. In *Proceedings of the May 5-7, 1970, Spring Joint Computer Conference*, AFIPS ’70 (Spring), pp. 551–567. 1970.

Russell **Housley**. “Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP)”. RFC 3686, Internet Engineering Task Force, January 2004.

John H. **Howard**, Michael L. **Kazar**, Sherri G. **Menees**, David A. **Nichols**, M. **Satyanarayanan**, Robert N. **Sidebotham**, and Michael J. **West**. “Scale and Performance in a Distributed File System”. *ACM Transactions on Computer Systems*, Volume 6(1):pp. 51–81, 1988.

Wenjin **Hu**, Tao **Yang**, and Jeanna N. **Matthews**. “The Good, the Bad and the Ugly of Consumer Cloud Storage”. *SIGOPS Operating Systems Review*, Volume 44:pp. 110–115, August 2010.

Stefan **Hudelmaier**. “Realization of the "Extensible Messaging and Presence Protocol" Based on a Structured Overlay Network”. Diplomarbeit, Lehrstuhl für Netzarchitekturen und Netzdienste, Technische Universität München, 2008.

Yong **Hwang** and Pil **Lee**. “Efficient Broadcast Encryption Scheme with Log-Key Storage”. In Giovanni **Di Crescenzo** and Avi **Rubin** (editors), *Financial Cryptography and Data Security*, Volume 4107 of *Lecture Notes in Computer Science*, pp. 281–295. 2006.

–∼ I ∼–

Mikel **Izal**, Guillaume **Urvoy-Keller**, Ernst W. **Biersack**, Pascal A. **Felber**, Anwar Al **Hamra**, and Luis **Garcés-Erice**. “Dissecting BitTorrent: Five Months in a Torrent’s Lifetime”. In *Passive and Active Network Measurement*, Volume 3015 of *Lecture Notes in Computer Science*, pp. 1–11. 2004.

–∼ J ∼–

Markus **Jakobsson**. “Fractal Hash Sequence Representation and Traversal”. In *Proceedings of the 2002 IEEE International Symposium on Information Theory*, p. 437. 2002.

Márk **Jelasity**, Alberto **Montresor**, Gian Paolo **Jesi**, and Spyros **Voulgaris**. “The Peersim Simulator”. *http://peersim.sf.net*.

Gideon **Juve**, Ewa **Deelman**, Karan **Vahi**, Gaurang **Mehta**, Bruce **Berriman**, Benjamin P. **Berman**, and Phil **Maechling**. “Data Sharing Options for Scientific Workflows on Amazon EC2”. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’10. 2010.

–∼ K ∼–

KAD Trace. “Eurecom KAD Trace”. accessed 6. April 2011 . *http://www.eurecom.fr/~btroup/kadtraces/*.

David **Kahn**. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet.* Scribner, Rev Sub Edition, December 1996.

Gilles **Kahn**. “The Semantics of a Simple Language for Parallel Programming”. In J. L. **Rosenfeld** (editor), *Information Processing*, pp. 471–475. 1974.

Burton S. **Kaliski**. “PKCS #5: Password-Based Cryptography Specification Version 2.0”. RFC 2898 (Informational), September 2000.

Mahesh **Kallahalla**, Erik **Riedel**, Ram **Swaminathan**, Qian **Wang**, and Kevin **Fu**. “Plutus: Scalable Secure File Sharing on Untrusted Storage”. In *Proceedings of the Second USENIX Conference on File and Storage Technologies*, FAST ’03, pp. 29–42. 2003.

David **Karger**, Eric **Lehman**, Tom **Leighton**, Rina **Panigrahy**, Matthew **Levine**, and Daniel **Lewin**. “Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web”. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC 1997, pp. 654–663. 1997.

Vishal **Kher** and Yongdae **Kim**. “Securing Distributed Storage: Challenges, Techniques, and Systems”. In *Proceedings of the 2005 ACM Workshop on Storage Security and survivability*, StorageSS ’05, pp. 9–25. 2005.

Olaf **Kirch**. “Why NFS Sucks”. In *Proceedings of the 2006 Linux Symposium*, pp. 51–63. 2006.

Yves Philippe **Kising**. “Proximity Neighbor Selection and Proximity Route Selection for the Overlay-Network IGOR”. Diplomarbeit, Lehrstuhl für Netzwerkarchitekturen, Technische Universität München, 2007.

Tor **Klingberg** and Raphael **Manfredi**. “Gnutella 0.6”. RFC Draft, June 2002. *http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html*.

Mirko **Knoll**, Matthias **Helling**, Arno **Wacker**, Sebastian **Holzapfel**, and Torben **Weis**. “Bootstrapping Peer-to-Peer Systems Using IRC”. In *Proceedings of the Eighteenth IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises*, WETICE ’09, pp. 122–127. 2009.

Donald E. **Knuth**. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley Publishing Company, Inc., Reading, Massechusetts, USA, 1978. ISBN 0-201-83803-X.

Neal **Koblitz**. “Elliptic Curve Cryptosystems”. *Mathematics of Computation*, Volume 48(177):pp. 203–209, January 1987.

Michael **Kofler**. *Linux 2011.* Addison-Wesley, 10th Edition, 2011. ISBN 3-8273-3025-3.

Amy **Kover**. “Napster: The Hot Idea of the Year”. *FORTUNE*, Volume 142(1):pp. 128+, JUN 26 2000.

Erol **Koç**. “Access Control in Peer-to-Peer Storage Systems”. Master’s thesis, ETH Zurich, October 2006.

Erol **Koç**, Marcel **Baur**, and Germano **Caronni**. “PACISSO: P2P Access Control Incorporating Scalability and Self-Organization for Storage Systems”. Technical Report SMLI TR-2007-165, Mountain View, CA, USA, 2007.

John **Kubiatowicz**, David **Bindel**, Yan **Chen**, Steven **Czerwinski**, Patrick **Eaton**, Dennis **Geels**, Ramakrishan **Gummadi**, Sean **Rhea**, Hakim **Weatherspoon**, Westley **Weimer**, Chris **Wells**, and Ben **Zhao**. “OceanStore: an Architecture for Global-Scale Persistent Storage”. *ACM SIGPLAN Notices*, Volume 35(11):pp. 190–201, 2000.

Yoram **Kulbak** and Danny **Bickson**. “The eMule Protocol Speficiation”. Technical Report TR-2005-03, Leibniz Center, School of Computer Science and Engineering, The Hebrew University, Jerusalem, Israel, 2005.

Kendy **Kutzner**. “The Decentralized File System Igor-FS as an Application for Overlay Networks”. Ph.D. Thesis, Universität Karlsruhe (TH), Karlsruhe, Germany, 2008.

Kendy **Kutzner**, Curt **Cramer**, and Thomas **Fuhrmann**. “A Self-Organizing Job Scheduling Algorithm for a Distributed VDR”. In *Workshop "Peer-to-Peer-Systeme* *und -Anwendungen", 14. Fachtagung Kommunikation in Verteilten Systemen*, KiVS '05. 2005.

Kendy **Kutzner** and Thomas **Fuhrmann**. "The IGOR File System for Efficient Data Distribution in the GRID". In *Proceedings of the Cracow Grid Workshop*, CGW '06. 2006.

–∼ **L** ∼–

Leslie **Lamport**. "Constructing Digital Signatures from a One-Way Function". Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, October 1979.

Nathaniel **Leibowitz**, Matei **Ripeanu**, and Adam **Wierzbicki**. "Deconstructing the Kazaa Network". In *Proceedings of the Third IEEE Workshop on Internet Applications*, WIAPP '03, pp. 112–120. 2003.

Frank T. **Leighton** and Silvio **Micali**. "Secret-Key Agreement without Public-Key Cryptography". In *Proceedings of the Thirteenth Annual International Cryptology Conference on Advances in Cryptology*, Volume 773 of *Lecture Notes in Computer Science*, pp. 456–479. 1994.

Barry M. **Leiner**, Donald L. **Nielson**, and Fouad A. **Tobagi**. "Packet Radio in the Amateur Service". *Selected Areas in Communications, IEEE Journal on*, Volume 3(3):pp. 431 – 439, may 1985.

Arjen K. **Lenstra** and Eric R. **Verheul**. "Selecting Cryptographic Key Sizes". *Journal of Cryptology*, Volume 14:pp. 255–293, 2001.

Eliezer **Levy** and Abraham **Silberschatz**. "Distributed File Systems: Concepts and Examples". *ACM Computing Survey*, Volume 22:pp. 321–374, December 1990.

Daniel **Lewin**. "Consistent Hashing and Random Trees: Algorithms for Caching in Distributed Networks". Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1998.

Jinyuan **Li**, Maxwell **Krohn**, David **Mazières**, and Dennis **Shasha**. "Secure Untrusted Data Repository (SUNDR)". In *Proceedings of the Sixth Symposium on Opearting Systems Design & Implementation*, OSDI '04, pp. 121–136. 2004.

Thomas **Locher**, David **Mysicka**, Stefan **Schmid**, and Roger **Wattenhofer**. "Poisoning the Kad Network". In *Distributed Computing and Networking*, Volume 5935 of *Lecture Notes in Computer Science*, pp. 195–206. 2010.

Jeff **Lotspiech**, Moni **Naor**, and Dalit **Naor**. "Subset-Difference based Key Management for Secure Multicast". IRTF Draft, Internet Research Task Force, July 2001.

Chris **Lowth**. "Securing Your Network Against Kazaa". *Linux Journal*, Volume 2003(114):p. 3, 2003.

Jie **Lu** and Jamie **Callan**. "Content-Based Retrieval in Hybrid Peer-to-Peer Networks". In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, CIKM '03, pp. 199–206. 2003.

Eng Keong **Lua**, Jon **Crowcroft**, Marcelo **Pias**, Ravi **Sharma**, and Steven **Lim**. "A Survey and Comparison of Peer-to-Peer Overlay Network Schemes". *IEEE Communications Surveys and Tutorials*, Volume 7:pp. 72–93, 2005.

Michael G. **Luby**, Michael **Mitzenmacher**, M. Amin **Shokrollahi**, Daniel A. **Spielman**, and Volker **Stemann**. "Practical Loss-Resilient Codes". In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC 1997, pp. 150–159. 1997.

Qin **Lv**, Pei **Cao**, Edith **Cohen**, Kai **Li**, and Scott **Shenker**. "Search and Replication in Unstructured Peer-to-Peer Networks". In *Proceedings of the Sixteenth International Conference on Supercomputing*, ICS '02, pp. 84–95. 2002.

Julio **Lôpez** and Ricardo **Dahab**. "Performance of Elliptic Curve Cryptosystems". Technical Report IC-00-08, Institute of Computing, University of Campinas, São Paulo, Brasil, May 2000.

–∼ M ∼–

Mac Security. "Security Overview". In *Mac OS X Developer Library.* Apple Inc., July 2010. *http://developer.apple.com/library/mac/documentation/Security/Conceptual/Security_Overview/Security_Overview.pdf*.

Thomas **Marill** and Dale **Stern**. "The Datacomputer: A Network Data Utility". In *Proceedings of the May 19-22, 1975, Cational Computer Conference and Exposition*, AFIPS '75, pp. 389–395. 1975.

Avantika **Mathur**, Mingming **Cao**, Suparna **Bhattacharya**, Andreas **Dilger**, Alex **Tomas**, and Laurent **Vivier**. "The New ext4 Filesystem: Current Status and Future Plans". In *Proceedings of the 2007 Linux Symposium.* 2007.

Petar **Maymounkov** and David **Mazières**. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '02, pp. 53–65. 2002.

Petar **Maymounkov** and David **Mazières**. "Rateless Codes and Big Downloads". In *Proceedings of the Second International Workshop on Peer-to-Peer Systems*, IPTPS '03. 2003.

David **Mazières** and Dennis **Shasha**. "Building Secure File Systems out of Byzantine Storage". In *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing*, pp. 108–117. 2002.

Erin **McKean**. *The New Oxford American Dictionary.* Oxford University Press, USA, Second Edition, 2005. ISBN 0-1951-7077-1.

Marshall K. **McKusick**, William N. **Joy**, Samuel J. **Leffler**, and Robert S. **Fabry**. "A Fast File System For UNIX". *ACM Transactions on Computer Systems*, Volume 2:pp. 181–197, August 1984.

Ralph C. **Merkle**. "Secure Communications over Insecure Channels". *Communications of the ACM*, Volume 21(4):pp. 294–299, April 1978.

Ralph C. **Merkle**. "Secrecy, Authentication, and Public Key Systems". Ph.D. thesis, Stanford University, Stanford, CA, USA, 1979.

Ralph C. **Merkle**. "A Digital Signature Based on a Conventional Encryption Function". In *Advances in Cryptology – CRYPTO '87*, Volume 293 of *Lecture Notes in Computer Science*, pp. 369–378. 1988.

Ralph C. **Merkle**. "One Way Hash Functions and DES". In *Advances in Cryptology – Crypto '89*, Volume 435 of *Lecture Notes in Computer Science*, pp. 428–446. 1989.

Ralph C. **Merkle**. "A Certified Digital Signature". In *Advances in Cryptology – CRYPTO 89*, Volume 435 of *Lecture Notes in Computer Science*, pp. 218–238. Springer Berlin / Heidelberg, 1990.

Miodrag **Mihaljević**. "Key Management Schemes for Stateless Receivers Based on Time Varying Heterogeneous Logical Key Hierarchy". In *Advances in Cryptology - ASIACRYPT 2003*, Volume 2894 of *Lecture Notes in Computer Science*, pp. 137–154. 2003.

Ethan **Miller**, Darrell **Long**, William **Freeman**, and Benjamin **Reed**. "Strong Security for Distributed File Systems". In *Proceedings of the Twentieth IEEE International Performance, Computing, and Communications Conference*, IPCCC '02, pp. 34–40. 2002.

Victor S. **Miller**. "Use of Elliptic Curves in Cryptography". In *Advances in Cryptology - CRYPTO '85*, Volume 218 of *Lecture Notes in Computer Science*. 1986.

James G. **Mitchell** and Jeremy **Dion**. "A Comparison of Two Betwork-Based File Servers". *Communications of the ACM*, Volume 25:pp. 233–245, April 1982.

Michael **Mitzenmacher**. "Digital Fountains: A Survey and Look Forward". In *IEEE Information Theory Workshop*, pp. 271 – 276. 2004.

Paul V. **Mockapetris** and Kevin J. **Dunlap**. "Development of the Domain Name System". *SIGCOMM Computer Communication Review*, Volume 25:pp. 112–122, January 1995.

Athicha **Muthitacharoen**, Robert **Morris**, Thomer M. **Gil**, and Benjie **Chen**. "Ivy: A Read/Write Peer-to-Peer File System". In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation"*, OSDI '02, pp. 31–44. 2002.

—∼ N ∼—

Dalit **Naor**, Moni **Naor**, and Jeff **Lotspiech**. “Revocation and Tracing Schemes for Stateless Receivers”. In *Advances in Cryptology - CRYPTO 2001: 21st Annual International Cryptology Conference*, Volume 2139 of *Lecture Notes in Computer Science*, pp. 41–62. 2001.

Dalit **Naor**, Amir **Shenhav**, and Avishai **Wool**. “Toward Securing Untrusted Storage Without Public-Key Operations”. In *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability*, StorageSS ’05, pp. 51–56. 2005.

Dalit **Naor**, Amir **Shenhav**, and Avishai **Wool**. “One-Time Signatures Revisited: Practical Fast Signatures Using Fractal Merkle Tree Traversal”. In *Proceedings of the Twenty-Fourth IEEE Convention of Electrical and Electronics Engineers in Israel*, pp. 255–259. 2006.

B. Clifford **Neuman** and Theodore **Ts’o**. “Kerberos: An Authentication Service for Computer Networks”. *IEEE Commmunication Magazine*, Volume 32(9):pp. 33–38, September 1994.

NIST-DSS. “The Digital Signature Standard proposed by NIST”. *Communications of the ACM*, Volume 35(7):pp. 36–40, 1992.

—∼ O ∼—

OMNeT++. “OMNeT++ Network Simulation Framework”. *http://www.omnetpp.org*.

Andy **Oram**. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* O’Reilly & Associates, First Edition, 2001.

Rafail **Ostrovsky**, Amit **Sahai**, and Brent **Waters**. “Attribute-Based Encryption with Mon-Monotonic Access Structures”. In *Proceedings of the Fourteenth ACM Conference on Computer and Communications Security*, CCS ’07, pp. 195–203. 2007.

—∼ P ∼—

P2PTA. “The Peer-to-Peer Trace Archive”. *http://p2pta.ewi.tudelft.nl/pmwiki*.

Jeffrey **Pang**, James **Hendricks**, Aditya **Akella**, Roberto **De Prisco**, Bruce **Maggs**, and Srinivasan **Seshan**. “Availability, Usage, and Deployment Characteristics of the Domain Name System”. In *Proceedings of the Fourth ACM SIGCOMM Conference on Internet Measurement*, IMC ’04, pp. 1–14. 2004.

Colin **Percival**. “Wuala Update”. October 2007a. *http://www.daemonology.net/blog/2007-10-26-wuala-update.html*.

Colin **Percival**. “Wuala: Willful Ignorance, or Fraud?” October 2007b. *http://www.daemonology.net/blog/2007-10-21-wuala-willful-ignorance.html*.

Colin **Percival**. “Wuala’s Improved Security”. November 2008. *http://www.daemonology.net/blog/2008-11-07-wuala-security.html*.

Dalibor **Peric**. “DRFS: Distributed Reliable File System based on TomP2P”. Bachelor’s thesis, University of Zurich, Zurich, Switzerland, October 2008.

Dalibor **Peric**, Thomas **Bocek**, Fabio Victora **Hecht**, David **Hausheer**, and Burkhard **Stiller**. “The Design and Evaluation of a Distributed Reliable File System”. In *Proceedings of the 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT ’09, pp. 348–353. 2009.

W. Wesley **Peterson**. “Addressing for Random-Access Storage”. *IBM Journal of Research and Development*, Volume 1:pp. 130–146, April 1957.

James S. **Plank**, Jianqiang **Luo**, Catherine D. **Schuman**, Lihao **Xu**, and Zooko **Wilcox-O’Hearn**. “A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage”. In *Proccedings of the Seventh Conference on File and Storage Technologies*, FAST ’09, pp. 253–265. 2009.

POSIX ACL. “POSIX 1003.1e / 1003.2c Draft Standard 17 (withdrawn)”. IEEE, 1997.

POSIX.1. “The Open Group Technical Standard Base Specifications, Issue 6.” In *Standard for Information Technology - Portable Operating System Interface (POSIX). Shell and Utilities*, IEEE Std. 1003.1. Institute of Electrical and Electronics Engineers (IEEE), 2004.

Jon **Postel** and Joyce **Reynolds**. “File Transfer Protocol”. RFC 959, Internet Engineering Task Force, October 1985.

Vijayan **Prabhakaran**, Andrea C. **Arpaci-Dusseau**, and Remzi H. **Arpaci-Dusseau**. “Analysis and Evolution of Journaling File Systems”. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC ’05. 2005.

Bart **Preneel**. “Cryptographic Hash Functions”. *European Transactions on Telecommunications*, Volume 5(4):pp. 431–448, 1994.

Bart **Preneel**. “The State of Cryptographic Hash Functions”. In *Lectures on Data Security: Modern Cryptology in Theory and Practice*, LNCS Tutorial, pp. 158–182. 1999.

–∼ **Q** ∼–

David **Quigley**, Josef **Sipek**, Charles P. **Wright**, and Erez **Zadok**. “UnionFS: User- and Community-Oriented Development of a Unification File System”. In *Proceedings of the 2006 Linux Symposium*, pp. 349–362. 2006.

–∼ R ∼–

Michael O. **Rabin**. “Digitalized Signatures”. In Richard J. **Lipton**, David P. **Dobkin**, and Anita K. **Jones** (editors), *Foundations of Secure Computation*, pp. 155–166. Academic Press, Inc., Orlando, FL, USA, 1978. ISBN 0-1221-0350-5.

Michael O. **Rabin**. “Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance”. *Journal of the ACM*, Volume 36:pp. 335–348, April 1989.

Sylvia **Ratnasamy**, Paul **Francis**, Mark **Handley**, Richard **Karp**, and Scott **Shenker**. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM ’01, pp. 161–172. 2001.

David **Reed** and Liba **Svobodova**. “SWALLOW: A Distributed Data Storage System for a Local Network”. In A. **West** and P. **Janson** (editors), *Local Networks for Computer Communications*, pp. 355–373. 1981.

Irving S. **Reed** and Gustave **Solomon**. “Polynomial Codes Over Certain Finite Fields”. *Journal of the Society for Industrial and Applied Mathematics*, Volume 8(2):pp. 300–304, 1960.

Yanli **Ren** and Dawu **Gu**. “Fully CCA2 Secure Identity Based Broadcast Encryption Without Random Oracles”. *Information Processing Letters*, Volume 109(11):pp. 527–533, 2009.

Sean **Rhea**, Patrick **Eaton**, Dennis **Geels**, Hakim **Weatherspoon**, Ben **Zhao**, and John **Kubiatowicz**. “Pond: The OceanStore Prototype”. In *Proceedings of the Second USENIX Conference on File and Storage Technologies*, FAST ’03, pp. 1–14. 2003.

Bruno **Richard**, Donal Mac **Nioclais**, and Denis **Chalon**. “Clique: A Transparent, Peer-to-Peer Replicated File System”. In *Proceedings of the Fourth International Conference on Mobile Data Management*, MDM ’03, pp. 351–355. 2003.

Ronald L. **Rivest**, Adi **Shamir**, and Leonard M. **Adleman**. “Cryptographic Communications System and Method”. United States Patent 4,405,829, December 1977.

Ronald L. **Rivest**, Adi **Shamir**, and Leonard M. **Adleman**. “A Method for Obtaining Digital Signatures and Public-key Cryptosystems”. *Communications of the ACM*, Volume 21(2):pp. 120–126, 1978.

Lawrence G. **Roberts** and Barry D. **Wessler**. “Computer Network Development to Achieve Resource Sharing”. In *Proceedings of the May 5-7, 1970, Spring Joint Computer Conference*, AFIPS ’70 (Spring), pp. 543–549. 1970.

Rodrigo **Rodrigues** and Barbara **Liskov**. “High Availability in DHTs: Erasure Coding vs. Replication”. In *Peer-to-Peer Systems IV*, Volume 3640 of *Lecture Notes in Computer Science*, pp. 226–239. 2005.

Antony **Rowstron** and Peter **Druschel**. "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems". In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware, pp. 329–350. 2001a.

Antony **Rowstron** and Peter **Druschel**. "Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility". In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pp. 188–201. 2001b.

William E. **Ryan** and Shu **Lin**. *Channel Codes: Classical and Modern.* Cambridge University Press, 2009. ISBN 0-5218-4868-7.

–∼ S ∼–

Amit **Sahai** and Brent **Waters**. "Fuzzy Identity-Based Encryption". In *Advances in Cryptology – EUROCRYPT 2005*, Volume 3494 of *Lecture Notes in Computer Science*, pp. 457–473. 2004.

Oskar **Sandberg**. "Distributed Routing in Small-World Networks". In *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments*, ALENEX06. 2006.

Russel **Sandberg**, David **Goldberg**, Steve **Kleiman**, Dan **Walsh**, and Bob **Lyon**. "Design and Implementation of the Sun Network Filesystem". In *Innovations in Internetworking*, pp. 379–390. Artech House, Inc., Norwood, MA, USA, 1988. ISBN 0-89006-337-0.

Akashi **Satoh**. "Hardware Architecture and Cost Estimates for Breaking SHA-1". In *Information Security*, Volume 3650, pp. 259–273. Springer Berlin / Heidelberg, 2005.

Mahadev **Satyanarayanan**. "The Evolution of Coda". *ACM Transactions on Computer Systems*, Volume 20:pp. 85–124, May 2002.

Mahadev **Satyanarayanan**, James J. **Kistler**, Puneet **Kumar**, Maria E. **Okasaki**, Ellen H. **Siegel**, and David C. **Steere**. "Coda: A Highly Available File System for a Distributed Workstation Environment". *IEEE Transactions on Computers*, Volume 39(4):pp. 447–459, April 1990.

Holger **Schmidt**. "Napster war da". *Frankfurter Allgemeine Zeitung*, (206):p. 22, 5th September 2002.

Rüdiger **Schollmeier**. "Signaling and Networking in Unstructured Peer-To-Peer Networks". Dissertation, Technische Universität München, Munich, Germany, 2004.

Rüdiger **Schollmeier** and Gerald **Kunzmann**. "GnuViz – Mapping the Gnutella Network to its Geographical Locations". *PIK – Praxis der Informationsverarbeitung und Kommunikation*, Volume 26, 2003.

Tim **Schürmann**. "Das verteilte Dateisystem Kosmos-FS". *Linux-Magazin*, April 2008.

Kareem **Shaheen**. "Upgrades Vital to Avoid Internet Disruptions". *The National*, Volume 3(350):p. 4, 3rd April 2011.

Adi **Shamir**. "How to Share a Secret". *Communications of the ACM*, Volume 22:pp. 612–613, November 1979.

Adi **Shamir**. "Identity-Based Cryptosystems and Signature Schemes". In *Advances in Cryptology – CRYPTO '84*, Volume 196 of *Lecture Notes in Computer Science*, pp. 47–53. 1985.

Spencer **Shepler**, Brent **Callaghan**, David **Robinson**, Robert **Thurlow**, Carl **Beame**, Mike **Eisler**, and David **Noveck**. "NFS Version 4 Protocol". RFC 3010, Internet Engineering Task Force, December 2000.

Amin **Shokrollahi**. "Raptor Codes". *IEEE/ACM Transactions on Networking*, Volume 14(SI):pp. 2551–2567, 2006.

Abraham **Silberschatz**, Peter Baer **Galvin**, and Greg **Gagne**. *Operating System Concepts.* Wiley Publishing, Seventh Edition, 2005. ISBN 0-4716-9466-5.

Simon **Singh**. *The Code Book: The Evolution of Secrecy from Mary, Queen of Scots, to Quantum Cryptography.* Doubleday, New York, NY, USA, First Edition, 1999. ISBN 0-3854-9531-5.

Richard C. **Singleton**. "Maximum Distance Q-Nary Codes". *IEEE Transactions on Information Theory*, Volume 10(2):pp. 116–118, April 1964.

Solaris Security. "System Administration Guide: Security Services". In *Oracle Solaris 10 System Administrator Collection.* Part No: 816–4557–19, Oracle, September 2010.

SpiderOak. *http://spideroak.com.*

Raj **Srinivasan**. "RPC: Remote Procedure Call Protocol Specification Version 2". RFC 1831, August 1995.

SSFNet. "Scalable Simulation Framework (SSFNet)". *http://www.ssfnet.org.*

Moritz **Steiner**, Taoufik **En-Najjary**, and Ernst W **Biersack**. "Analyzing Peer Behavior in KAD". Technical Report EURECOM+2358, Institut Eurecom, France, October 2007a.

Moritz **Steiner**, Taoufik **En-Najjary**, and Ernst W. **Biersack**. "A Global View of KAD". In *Proceedings of the Seventh ACM SIGCOMM Conference on Internet Measurement*, IMC '07, pp. 117–122. 2007b.

Ralf **Steinmetz** and Klaus **Wehrle**. "Peer-to-Peer-Networking & -Computing". *Informatik Spektrum*, Volume 27(1):pp. 51–54, 2004.

Ralf **Steinmetz** and Klaus **Wehrle**. “What Is This “Peer-to-Peer” About?” In *Peer-to-Peer Systems and Applications*, Volume 3485 of *Lecture Notes in Computer Science*, pp. 79–93. 2005. ISBN 3-540-29192-3.

Ion **Stoica**, Robert **Morris**, David **Karger**, M. Frans **Kaashoek**, and Hari **Balakrishnan**. “Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications”. In *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 149–160. 2001.

Liba **Svobodova**. “File Servers for Network-Based Distributed Systems”. *ACM Computing Surveys*, Volume 16:pp. 353–398, December 1984.

Daniel **Swinehart**, Gene **McDaniel**, and David **Boggs**. “WFS: a Simple Shared File System for a Distributed Environment”. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, SOSP ’79, pp. 9–17. 1979.

–∼ **T** ∼–

Tahoe. *http://tahoe-lafs.org*.

Tahoe Architecture. “Tahoe-LAFS Architecture”. accessed 6. December 2011 . *https://tahoe-lafs.org/trac/tahoe-lafs/browser/trunk/docs/architecture.rst?rev=4887*.

Tahoe Summary. “Tahoe Summary”. accessed 6. December 2011 . *https://tahoe-lafs.org/trac/tahoe-lafs/browser/trunk/docs/about.rst?rev=5345*.

Andrew S. **Tanenbaum**. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, second Edition, 2001. ISBN 0-1303-1358-0.

Andrew S. **Tanenbaum**. *Computer Networks*. Prentice Hall PTR, Upper Saddle River, NJ, USA, Fourth Edition, 2002. ISBN 0-1306-6102-3.

Li **Tang**, Yin **Chen**, Fei **Li**, Hui **Zhang**, and Jun **Li**. “Empirical Study on the Evolution of PlanetLab”. In *Proceedings of the Sixth International Conference on Networking*, ICN ’07, pp. 64–70. 2007.

Kiran **Tati** and Geoffrey M. **Voelker**. “On Object Maintenance in Peer-to-Peer Systems”. In *Proceedings of the Fifth International Workshop on Peer-to-Peer Systems*, IPTPS ’06. 2006.

Chuck **Thacker**. “Personal Distributed Computing: the Alto and Ethernet Hardware”. In *Proceedings of the ACM Conference on the History of Personal Workstations*, HPW ’86, pp. 87–100. 1986.

TomP2P. *http://www.csg.uzh.ch/publications/software/TomP2P*.

TOP500. “TOP500 Supercomputing Sites November 2010”. *http://www.top500.org/lists/2010/11*.

Yuh-Min **Tseng**. “A Scalable Key-Management Scheme with Minimizing Key Storage for Secure Group Communications”. *International Journal of Network Management*, Volume 13:pp. 419–425, November 2003.

Tu. “Legitime Interessen”. *Frankfurter Allgemeine Zeitung*, (67):p. T 1, 3rd March 2007.

Stephen C. **Tweedie**. “Journaling the Linux ext2fs Filesystem”. In *Proceedings of the Fourth Annual Linux Expo*. 1998.

–∼ **U** ∼–

Gil **Utard** and Antoine **Vernois**. “Data Durability in Peer to Peer Storage Systems”. In *Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid*, CCGRID ’04, pp. 90–97. 2004.

–∼ **V** ∼–

Ivan **Voras** and Mario **Žagar**. “Network Distributed File System in User Space”. In *Proceedings of the Twenty-Eigth International Conference on Information Technology Interfaces*, pp. 669–674. 2006.

–∼ **W** ∼–

Debby M. **Wallner**, Eric J. **Harder**, and Ryan C. **Agee**. “Key Management for Multicast: Issues and Architectures”. RFC 2627, Internet Engineering Task Force, June 1999.

Randolph Y. **Wang** and Thomas E. **Anderson**. “xFS: A Wide Area Mass Storage File System”. In *Workshop on Workstation Operating Systems*, pp. 71–78. 1993.

Xiaoyun **Wang**, Yiqun **Yin**, and Hongbo **Yu**. “Finding Collisions in the Full SHA-1”. In *Advances in Cryptology – CRYPTO 2005*, Volume 3621 of *Lecture Notes in Computer Science*, pp. 17–36. 2005.

Brian **Warner**, Zooko **Wilcox-O’Hearn**, and Rob **Kinninmont**. “Tahoe: A Secure Distributed Filesystem”. accessed 6. December 2011 . *https://tahoe-lafs.org/~warner/pycon-tahoe.html*.

Hakim **Weatherspoon** and John **Kubiatowicz**. “Erasure Coding Vs. Replication: A Quantitative Comparison”. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS ’01, pp. 328–338. 2002.

Klaus **Wehrle**, Stefan **Götz**, and Simon **Rieche**. “Distributed Hash Tables”. In *Peer-to-Peer Systems and Applications*, Volume 3485 of *Lecture Notes in Computer Science*, pp. 79–93. 2005. ISBN 3-540-29192-3.

Michael J. **Wiener**. “Performance Comparison of Public-Key Cryptosystems”. *Crypto-Bytes*, Volume 4(1):pp. 1–5, 1998.

Zooko **Wilcox-O'Hearn** and Brian **Warner**. "Tahoe: The Least-Authority Filesystem". In *Proceedings of the Fourth ACM International Workshop on Storage Security and Survivability*, StorageSS '08, pp. 21–26. 2008.

Chris **Williams**, Philippe **Huibonhoa**, JoAnne **Holliday**, Andy **Hospodor**, and Thomas **Schwarz**. "Redundancy Management for P2P Storage". In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '07, pp. 15–22. 2007.

Erik De **Win** and Bart **Preneel**. "Elliptic Curve Public-Key Cryptosystems - an Introduction". In *State of the Art in Applied Cryptography: Course on Computer Security and Industrial Cryptography*, Volume 1528 of *Lecture Notes in Computer Science*. 1998.

Florian **Wohlfahrt**. "Wuala". In Georg **Carle** and Corinna **Schmidt** (editors), *Proceedings of the Seminar Future Internet*, FI SS2009. 2009.

Chung Kei **Wong**, Mohamed **Gouda**, and Simon S. **Lam**. "Secure Group Communications using Key Graphs". In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 68–79. 1998.

Charles P. **Wright**, Michael **Martino**, and Erez **Zadok**. "NCryptfs: A Secure and Convenient Cryptographic File System". In *Proceedings of the Annual USENIX Technical Conference*, pp. 197–210. 2003.

Charles P. **Wright** and Erez **Zadok**. "Kernel Korner: Unionfs: Bringing Filesystems Together". *Linux Journal*, Volume 2004, December 2004.

Wuala. *http://wua.la*.

Matthias **Wuttig** and Noboru **Yamada**. "Phase-Change Materials for Rewriteable Data Storage". *Nature Materials*, Volume 6:pp. 824 – 832, 2007.

WWPDB. "Worldwide Proteine Data Bank". *http://www.wwpdb.org*.

–∼ **X** ∼–

X9.62. "Public Key Cryptography for the Financial Services Industry, The Elliptic Curve Digital Signature Algorithm (ECDSA)". American National Standard X9.62:2005, Accredited Standards Commitee X9, November 2005.

–∼ **Y** ∼–

Xue-Jun **Yang**, Xiang-Ke **Liao**, Kai **Lu**, Qing-Feng **Hu**, Jun-Qiang **Song**, and Jin-Shu **Su**. "The TianHe-1A Supercomputer: Its Hardware and Software". *Journal of Computer Science and Technology*, Volume 26:pp. 344–351, 2011.

YouTube. *http://www.youtube.com*.

–∼ Z ∼–

Erez **Zadok**, Ion **Bădulescu**, and Alex **Shender**. "Cryptfs: A Stackable Vnode Level Encryption File System". Technical Report CUCS-021-98, Computer Science Department, Columbia University, June 1998.

Erez **Zadok** and Jason **Nieh**. "FiST: A Language for Stackable File Systems". In *Proceedings of the Annual USENIX Technical Conference*, pp. 55–70. 2000.

ZFS. "ZFS On-Disk Specification". Draft, Sun Microsystems, Inc., 2006. *http://hub.opensolaris.org/bin/download/Community+Group+zfs/docs/ondiskformat0822.pdf*.

Boxun **Zhang**, Alexandru **Iosup**, and Dick **Epema**. "The Peer-to-Peer Trace Archive: Design and Comparative Trace Analysis". Technical Report PDS-2010-003, Delft University of Technology, April 2010.

Zheng **Zhang** and Qiao **Lian**. "Reperasure: Replication Protocol Using Erasure-Code in Peer-to-Peer Storage Network". In *Proceedings of the Twenty-First IEEE Symposium on Reliable Distributed Systems*, SRDS '02, pp. 330–335. 2002.

Zheng **Zhang**, Qiao **Lian**, Shiding **Lin**, Wei **Chen**, Yu **Chen**, and Chao **Jin**. "BitVault: a highly reliable distributed data retention platform". *SIGOPS Operating Systems Review*, Volume 41:pp. 27–36, April 2007.

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

Dong **Zhen**. "6 Weeks for Sea Cable Repairs". *Shanghai Daily*, Volume 12(3564):p. A4, 25th March 2011.

Citation Index

Index