University of Passau
Faculty of Computer Science and Mathematics

Security in Information Systems
Prof. Dr. Hans Peter Reiser

Master's Thesis

Client-Side Encryption and Dynamic Group Management for a Secure Network Storage Service

Fabian Förg

Date:	July 2012
Supervisor:	Prof. Dr. Hans Peter Reiser
Co-Supervisor:	Prof. Dr. Joachim Posegga

Declaration of Authorship

I declare that I have developed and written the enclosed thesis entirely by myself and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from these sources are clearly marked as such. This thesis was not submitted in the same or in a substantially similar version, not even partially, to any other authority to achieve an academic grading and was not published elsewhere.

Passau, 4th July 2012

. . . . . .
(Fabian Förg)

Supervisor Contacts

Prof. Dr. Hans Peter Reiser
Chair of Security in Information Systems
Universität Passau
EMail: hans.reiser@uni-passau.de
Web: http://www.fim.uni-passau.de/sis.html

Prof. Dr. Joachim Posegga
Chair of IT Security
Universität Passau
EMail: posegga@uni-passau.de
Web: http://web.sec.uni-passau.de/

Abstract

Network storage providers are usually untrusted. Even if a provider encrypts the files on behalf of its users, data confidentiality is at stake, as the secrets which were used to encrypt the files are in the provider's hands. To overcome this issue, we devise a network storage system which encrypts files on the client-side. Moreover, our system offers the ability to share files within groups. Since group membership can be dynamic, we propose a key management scheme for confidential file sharing in dynamic groups. Furthermore, this thesis presents a protocol and an algorithm which enable file versioning as well as synchronization. Finally, we provide a fully-functional prototype implementation of our secure network storage service. The implementation has proved to be performant in our test environment.

Contents

Chapter 1

Introduction

Most network storage services provide online storage of client data and file synchronization. Moreover, these services typically support encrypted data transmission as well as server-side encryption, but lack client-side encryption. With server-side encryption, the keys are handled by the service provider on behalf of its users. The problem with server-side encryption is that the server provider has direct access to the plaintext, as it is able to access the keys which were used to encrypt the files on the server-side. Even if the provider was absolutely trustworthy, an attack on its infrastructure could result in a data security breach, as the keys for the encrypted data are located at the provider. Furthermore, it is possible that the provider cooperates with private or federal organizations which could gain access to confidential client data through collusion.

Storage providers may keep the data of their users in a cloud. Despite the advantages of cloud storage, such as high availability and low costs, transferring data to the cloud raises concerns. In 2008, Richard Stallman talked about privacy concerns in clouds: "computer users should be keen to keep their information in their own hands, rather than hand it over to a third party" [Bob08]. Note that means other than hacker interventions may result in the disclosure of private data. For example, a software malfunction in Google Docs in 2009 was enough to lead to unauthorized access to user data [CKS09]. Moreover, cloud services might not ensure data integrity. For instance, a malfunction at Amazon caused silent data corruption of user data stored on Amazon's S3 (Simple Storage System) storage cloud in 2008 [CKS09]. Therefore measures which keep the trust put into the cloud provider at a minimum level are necessary.

Although modern data-hosting services provide security features, the confidentiality of private data might still be at stake. For example, the popular file-hosting service Dropbox [Dro12a] offers over-the-wire and server-side data encryption, but not client-side encryption. Therefore attackers which compromise Dropbox servers, might gain access to confidential user data. In addition, when the U.S. govern-

ment demands data of Dropbox users, Dropbox Inc. must hand over the data (cf. Chapter 2).

When services such as Dropbox which only deploy secure data transmission and server-side encryption become victims of attacks, the confidentiality of all client data is threatened. Client-side encryption is an effective measure to prevent persons and parties who have or can request access to the server data from reading client files. With client-side encryption, files are encrypted before they are uploaded to the server. An intruder in the server's network would need to obtain the secret keys of the client in order to be able to decrypt the files and read the plaintext.

Users might encrypt their data with tools such as TrueCrypt [Tru12], BoxCryptor [Sec12a], or SecretSync [Com12a] before they submit them to their Dropbox. However, group sharing becomes cumbersome with such tools, as they lack group key management capabilities. Therefore users must reveal their own passwords to group members. Moreover, using such tools along with Dropbox, might render file synchronization inefficient (cf. Chapter 2).

The service Wuala [LaC12a] provides client-side encryption, but the source code of the client and server are not published. Therefore it is not known whether Wuala contains backdoors (cf. Chapter 2).

There are many other synchronization services besides Dropbox and Wuala. However, their security features are often opaque to the user (cf. Chapter 2).

This work presents the design of a secure network storage service and discusses design alternatives. Moreover, this thesis explains the workings of the service in detail and provides a proof of concept in form of a multi-platform prototype.

Our solution relies on a client-server architecture. The service makes use of client-side encryption to provide file content confidentiality. It also protects the integrity of file content. Confidential group sharing is another distinguishing feature of our designed solution. Additionally, file synchronization and file versioning constitute an integral part of our service. Further design goals include, but are not limited to, simplicity, usability, and performance.

1.1 Challenges

Users may have multiple devices containing files which they want to synchronize with a server owing to availability. Additionally, users want to be sure that the files which they get from the server were not tampered with, i.e. clients must check whether the received file content was generated by an authorized client. This requires an appropriate integrity protection mechanism. Moreover, users might desire to share their files with other users in a confidential manner, meaning that unauthorized parties including adversaries on the server must be unable to read the contents of

shared files. In order to achieve this goal, we rely on client-side encryption, thereby ensuring file confidentiality, even when the storage provider is compromised. In turn, client-side encryption leads to the following challenges which motivated the design of our secure network storage service:

Encrypted file synchronization. When a client synchronizes its files with the server, changes from the client replica must be propagated to the server replica and vice versa. In order to prevent adversaries who are on the server from reading stored files, the server must not get access to the plaintext of client-supplied files. Therefore clients submit file ciphertexts rather than plaintexts to the server. Consequently, the server replica consists of ciphertexts only. However, on the client-side, the user works with the plaintext of files. Storing the ciphertext on the client-side as well, causes overhead, but allows to directly compare the client's ciphertexts to the server's ciphertexts. The challenge here is to find a computationally efficient solution that allows to propagate changes from clients to the server and vice versa. The storage space consumption of the solution should be low. Moreover, it is desirable to only exchange file changes rather than entire files between the client and server in order to keep network traffic low. Owing to client-side encryption, file changes can be computed based on ciphertexts or plaintexts. Both possibilities have their advantages and drawbacks which a suitable synchronization algorithm must take into account.

Dynamic group sharing. Sharing files in dynamic groups requires appropriate access control and key management schemes to prevent past group members from accessing the group's data. Thus, when a member leaves a group, the server should revoke access rights of the leaving member and the group key needs to be renewed. Joining group members must obtain the group key. Therefore another challenge associated with dynamic group management is to find a suitable key distribution mechanism.

1.2 Contributions

Our secure network storage service makes the following key contributions:

Encrypted file synchronization algorithm. We devise an algorithm that allows to efficiently synchronize encrypted client files with a server. Our file synchronization algorithm supports file versioning and file differences. Owing to file versioning, the server saves and keeps every file version that clients upload. The server does not remove files. Consequently, clients are able to restore locally deleted files, if the file to restore had been synchronized with the server. As our file synchronization algorithm takes file differences into account, we synchronize changes between file versions rather than entire files. Section 3.8

presents the details our algorithm, while Section 2.4 discusses related work.

Group key management scheme. We designed a group key management scheme that allows to share files in dynamic groups. File content confidentiality is provided for shared files through client-side encryption using the group key. A strength of our key management approach is its simplicity. Section 3.7 presents our key management scheme, while Section 2.2 presents other schemes.

Fully-functional prototype implementation. We realized a fully-functional prototype implementation of our service. It allows developers to write their own cryptographic and file difference plug-ins. Our service is compatible with various platforms, as it is written in Java, and showed better performance than Dropbox in our test environment. Chapter 4 describes the implementation in detail.

1.3 Roadmap

The rest of this thesis is organized into five chapters. Chapter 2 presents related work and draws comparisons to our solution. Chapter 3 presents the design of our secure network storage service. Moreover, it discusses alternative designs as well and defends the chosen approach. An overview of the implementation is provided in Chapter 4. Chapter 5 evaluates the implementation and design. Finally, Chapter 6 recapitulates on the key contributions of this thesis and suggests directions for future research.

Chapter 2

Related Work

This chapter provides an overview of related work. Section 2.1 discusses generic file synchronization services. Section 2.2 presents selected services from research which allow to share files in groups. Ideas from literature on how to manage access hierarchies are the subject of Section 2.3. In Section 2.4, we discuss related file synchronization algorithms.

2.1 Generic File Synchronization Services

Dropbox [Dro12a] is a popular file synchronization service that offers server-side encryption, but lacks client-side encryption. Files are sent from the Dropbox desktop client to the server over a 256-bit SSL (Secure Sockets Layer) connection where supported. Files are encrypted using AES-256 (Advanced Encryption Standard with 256-bit keys), after they had been uploaded, i.e. on the server-side [Dro12b]. Thereby Dropbox manages the keys on their user's behalf in order to be able to provide the most popular Dropbox features "like accessing ... files from the website, creating file previews, and sharing files with other people" [Dro11] without hassles. Although keys and data are stored on separate hosts in the Amazon S3 data center [Jef11], a remote entity manages the keys. Moreover, as Dropbox Inc. is located in San Francisco, U.S. law obligates the company to hand over user data to the government when the government demands it [Dro11]. In contrast, our service employs client-side encryption, while still offering group sharing (cf. Section 3.3).

Dropbox Inc. recommends users who want to manage keys on their own "products like TrueCrypt [Tru12] to store encrypted volumes within their Dropboxes" [Dro11]. There are also other products such as SecretSync [Com12a] which provide client-side encryption by encrypting the client's files before they are put into the Dropbox folder. The key is either derived from a user's passphrase or is stored in its entirety in the SecretSync data center [Com12b]. However, since TrueCrypt and SecretSync

use only a single key per volume or file tree, respectively, group sharing becomes cumbersome. BoxCryptor [Sec12a] is a similar tool which allows to encrypt and decrypt files from a certain folder and synchronize the ciphertexts using Dropbox or other services. It uses a single master key and dedicated keys for each folder which are encrypted with the master key [Sec12b]. Moreover, changing the key would result in a lot of overhead, as all files would need to be encrypted with the new key and transferred to Dropbox. BoxCryptor is only able to share subfolders with a workaround [Sec11]. The sharer has to reveal her BoxCryptor password from which the master key is derived. Moreover, using these tools in conjunction with a synchronization service, separates file encryption from file synchronization, since the file synchronization service is merely able to synchronize the ciphertexts as provided by the tools. When a minor plaintext change results in a significantly larger ciphertext difference owing to the employed cipher mode, the synchronization service will synchronize the large ciphertext change rather than the small plaintext change. Consequently, file synchronization is not as efficient, as it could be. On the contrary, our service synchronizes encrypted files efficiently (cf. Section 3.8). In addition, we properly support group sharing (cf. Section 3.7).

The online storage system Wuala [LaC12a] provides client-side file encryption and sharing through a cryptographic file system named Cryptree [GMSW06]. How this cryptographic file system works is explained in Section 2.3. Wuala stores the client's data redundantly across several data centers in Europe. Although the source code of Wuala's Webstart [Gee12b] is available, the desktop clients and the server software are closed source. However, Webstart is just a Java applet that loads the actual application code from a server. Therefore Wuala might contain a backdoor, as the core of the code is unpublished. LaCie AG develops and runs Wuala in Zürich [LaC12b]. Our network storage service, however, works on top of existing file systems and is consequently not integrated into a file system (cf. Section 3.4.1). Thus, we gain cross-platform compatibility and can take advantage of the features which the underlying file system offers. Additionally, our key management scheme is simple and therefore transparent (cf. Section 3.7). Moreover, since our protocol (cf. Section 3.11) is public, developers have the possibility to write their own backdoor-free client and server. Developers may also extend our client software with cryptography and diff plug-ins (cf. Section 5.4).

Apart from Dropbox and Wuala, many other data synchronization services exist. Web page [Raj10] lists selected data synchronization services and their features. As there are many online storage systems with different features whose workings are often unmentioned or cannot be checked, it is difficult to pick a secure, reliable, and functional service. In addition, article [G.F11] points out that synchronization service vendors sometimes mislead their users with their product descriptions.

2.2 Group Key Management

In order to synchronize and share confidential data within dynamic groups, we need a scheme which enables group key management. A group key management takes into account that keys must be renewed when a member leaves. We use the term *rekeying* for key renewals. When a member joins a group, the new member needs to get the current group key. Therefore an appropriate group key distribution mechanism is necessary.

Work in group key management can be divided into basic, hierarchical, batching, and trade-off schemes [JA03]. While the basic schemes do not offer efficient rekeying measures, hierarchical management schemes try to reduce rekeying overhead. Batching schemes reduce rekeying overhead by batching a number of joins or leaves before rekeying rather than updating the key on every membership change. It was shown that the number of rekeying messages for a group of n members is bounded by $\Omega(\log(n))$, if strict non-member confidentiality and non-collusion were required [FJA02]. Strict non-member confidentiality means that all entities which do not belong to the group are unable to derive the group key. Consequently, non-members cannot decrypt data that was encrypted with the group key. Collusion means that former group members co-operate in order to gain further information about the current group key. Trade-off schemes decrease the overhead beyond the $\Omega(\log(n))$ bound by trading off some collusion resistance [JA03]. Group key management is a broad field which we will not discuss here in-depth. Instead, we refer to the survey of group key management schemes [RH03] for an overview of solutions. In the following, we present some network storage services from literature and outline the role of group keys in these solutions.

The read-only, cryptographic file system Chefs [Fu05] employs group keys and content keys. Group keys are distributed over a secure, out-of-bound channel. Content keys are used to encrypt content and are stored in a lockbox. A lockbox is a "key encrypted with another key" [Fu99]. The lockbox metaphor refers to boxes used by real estate agents: Realtors put so called lockboxes on the door of houses for sale. These lockboxes contain the key to the house. Persons who know the combination to open the lockbox can enter it in order to get the key to the house [Fu99]. In Chefs, the content key of a lockbox is obtained by decrypting the lockbox with the group key. Our service relies on a secure, out-of-bound group key distribution as well (cf. Section 3.7). Although our design allows to integrate lockboxes, we do not adopt that concept (cf. Section 3.7). We distinguish between content and integrity keys (cf. Section 3.6.1). Moreover, our network storage service is layered above existing file systems, thereby gaining some positive properties (cf. Section 3.4.1). Furthermore, we provide efficient file synchronization and versioning mechanisms (cf. Section 3.8). File versioning enables users to restore old file versions.

Within the cryptographic storage file system Cepheus [Fu99], a lockbox contains a list consisting of multiple copies of a single file key. Each entry of the list, i.e. each

file key copy, is encrypted with the public key of a certain group member. Therefore the size of the list grows linearly with the number of group members. A group database server maintains lockboxes, user public keys as well as group membership lists, while a file server maintains group access rights [Fu99]. A user agent stores the private key of the corresponding users. When the user agent needs a group or file key, it requests the respective lockbox from the group database server and opens the lockbox with the user's private key. Our server controls access to resources as well (cf. Section 3.7). However, we do not provide a group database server, as we let resource owners distribute group keys out-of-bound (cf. Section 3.7), thereby avoiding a single point of failure which we would need to trust for key distribution. As our service acts on top of the file system layer, it is compatible with many operating systems. If our service were implemented as a file system, we would need to port the file system to the operating systems which we wanted to support.

SiRiUS [jGSMB03] is a secure storage file system which is layered on top of untrusted network and storage file systems. For each file, there is a metadata file named *md-file*. Keys are stored in metadata files using the concept of lockboxes. For each user who is allowed to access a certain file, the corresponding md-file contains a lockbox which is encrypted with the public key of the user. Depending on the access rights of the user, her lockbox contains at least a symmetric file encryption key for read access and a file signature key for write access. As for key distribution, [jGSMB03] recommends to use PKI, Identity-Based Encryption (IBE), or signature schemes, depending on the requirements of the user. SiRiUS supports symbolic links, file name encryption, and random data block access. Paper [jGSMB03] mentions an extension called SiRiUS-NNL which scales better with the number of users. The scheme on which the extension is based, NNL (Naor-Naor-Lotspiech), enables efficient key revocation. In contrast, our service employs only symmetric cryptography and does not use lockboxes (cf. Section 3.7). Moreover, our service does not support symbolic links, file name encryption, and random data access (cf. Section 3.3). However, we provide efficient file synchronization and versioning (cf. Section 3.8).

FARSITE (Federated, Available and Reliable Storage, for an Incompletely Trusted Environment) [ABC+02] is a secure and scalable file system which behaves like a central file server, though it is distributed over several untrusted machines. It guarantees file and directory integrity through a Byzantine-fault-tolerant protocol. Availability and reliability are provided by replication. When a user creates a file, she generates a random symmetric key. The key is encrypted with the public keys of all authorized readers. These lockboxes are stored along with the file. Note that files are encrypted block-wise with a one-way hash of the block used as the key. The hash of the block is encrypted with the key inside the lockbox and the resulting information is stored along with the block. File names are also encrypted with symmetric keys and the keys are kept in lockboxes which belong to directory metadata. File integrity is provided by building a Merkle hash tree [Mer80] over the file's data blocks. Although our service does not rely on lockboxes, it supports confidential group sharing of files as well (cf. Section 3.7). On the contrary, we do

not provide file integrity on the block-level. Instead, we protect the integrity of certain metadata which in turn includes a cryptographic hash of the file's content (cf. Section 3.2). We support file versioning and synchronize file changes, whereas FARSITE provides only block-level file access.

Plutus [KRS+03] is another cryptographic storage file system. It groups files with the same access rights into *filegroups*. Files of a filegroup share the same key without sacrificing security. One advantage of filegroups is that they reduce the number of keys. Therefore filegroups facilitate key distribution and key management. Plutus makes use of lockboxes to protect file content keys: File content keys reside inside a lockbox which is encrypted with a symmetric filegroup key. Thus, file content can be protected with different keys. This measure potentially decreases the vulnerability to known-plaintext and known-ciphertext attacks. As lockboxes are encrypted with a symmetric key, the size of a lockbox is constant. The symmetric filegroup keys are distributed by the owner over a secure-channel. Apart from symmetric filegroup keys, there also exists a public/private key pair for integrity protection. The key pair is used to sign and verify the cryptographic hash of the file content. Plutus encrypts names with *file-name keys* and *filegroup-name keys* which are distributed by the owner of the resource. We adopt the filegroup concept, as we use the same key for multiple files in the folder tree (cf. Section 3.7). Moreover, we let keys distribute out-of-band as well. However, we use a keyed MAC rather than a public/private key pair for integrity protection (cf. Section 3.7). Furthermore, we provide file versioning and synchronization.

PrPl [SSN+10] is a decentralized social networking infrastructure providing privacy-preserving information exchange. It uses so called Butlers which manage data on behalf of the users. Each Butler is associated with a public/private key pair and is able to grant tickets for data access. A Butler employs the decentralized OpenID [Ope12] system for identity management. PrPl application developers can utilize the SociaLite query language which allows to request information from a network of Butler services. On the contrary, our service relies on a client-server architecture for file distribution (cf. Section 3.4.1). Since the server is always online, we ensure data availability. Moreover, we take measures which enable file synchronization and versioning.

RFC 4046 presents the Multicast Security (MSEC) Group Key Management Architecture [Bau05]. Part of the architecture is the Group Controller and Key Server (GCKS) which takes care of group member authentication, authorization, and may provide rekey information distribution. RFC 4046 [Bau05] describes the corresponding protocols in detail. We distribute keys over a secure out-of-bound channel and thus do not depend on an infrastructure for group key management (cf. Section 3.7).

2.3 Key Management for Access Hierarchies

When users decide to share different folders with different groups, each shared folder must be equipped with a dedicated set of keys. In order to bequeath the access rights of a folder to another folder, a key management scheme is necessary. Paper [ABFF09] provides an elegant and efficient approach to handle key management in an access hierarchy. The access hierarchy is modeled as a directed graph where a key is assigned to each vertex and users who possess the key of a vertex can also access descending vertices through key derivation. As the access hierarchy model is a directed graph which may contain cycles, the key management scheme is applicable to many access hierarchy structures. For example, the vertices could be interpreted as files or folders of a file system tree hierarchy. The approach is suitable for role-based access control (RBAC) in which case the vertices represent the nodes of the role-based access hierarchy. The scheme relies on a trusted central authority which generates and distributes the keys.

Solution [ABFF09] has the following six properties:

1. Only hash functions are used to derive keys for descendant vertices of a vertex.
2. The space complexity for storing the public information of the hierarchy is equal to the space complexity of the hierarchy itself.
3. The private information of a vertex consists of a single key.
4. Updates of access rights can be handled locally in the hierarchy.
5. The scheme is provably secure against collusion.
6. The complexity of key derivation for a vertex is linear in the length of the path between an ancestor for which the key is known and the vertex.

In the base scheme [ABFF09], one key is assigned to each node. The keys which are to be distributed by a trusted entity, constitute the private information. The public information consists of a unique label which is assigned to each node. Additionally, edges belong to the public information. An edge (v_i, v_j) from vertex v_i to vertex v_j is associated with the public value $y_{i,j} = k_j - H(k_i, l_j)$ where k_j is the key of node v_j, k_i is the key of node v_i, H is a cryptographic hash function, and l_j denotes the label of node v_j. The operator $-$ denotes subtraction in a finite field F. All keys and all images of H must be in the corresponding set of F. For example, in the Galois field $GF(2^q)$ where q is a positive integer, an element of $GF(2^q)$ can be represented as a bit array of size q. Then the bit-wise exclusive or (XOR) operation serves as both addition and subtraction. With the knowledge of $y_{i,j}$, l_j, H (all public), and k_i (private), we can compute the key k_j of the descendant node v_j of node v_i with the formula $k_j = y_{i,j} + H(k_i, l_j)$.

Paper [ABFF09] describes an adaption of the base scheme called the dynamic version which allows to perform all changes of the hierarchy locally. The modification to

the base scheme is to use the key $k_i = H(\hat{k}_i, l_i)$ for the vertex v_i. The random key $\hat{k}_i$ must be distributed to all entities who are assigned access level v_i. This small modification makes it possible to change an access key of a node merely by changing the label of the node. Preventing ex-member access to a node v is done by changing its label and updating the edge values of all edges pointing to v apart from the edge which originates from the ex-member. The edge from the ex-member node to v has to be deleted. Paper [ABFF09] explains how the actions edge insertion, edge deletion, node insertion, node deletion, key replacement, and user revocation are performed.

An advantage of the scheme outlined in this section is that it is very flexible, as the hierarchy is modeled as a directed graph. Therefore the scheme supports complex access hierarchies. File or folder links can be modeled by representing files and folders as nodes. A link is then an edge from a node to a file/folder node. Users can be modeled as nodes and their access rights to resources, such as file and folders, can be represented as edges to resource nodes in the graph. Shortcuts, i.e. edges from a user node to a resource node, could be added in order to get constant time access to a resource. A drawback is that the server needs to handle and save public information (node labels and edges of the graph) in order to provide availability. This forces clients to trust the server with respect to managing the public information. Furthermore, user revocation might require plenty of graph updates [ABFF09]. As the graph can become arbitrarily complex with time, especially when there are many users and large groups, users and group members might lose track of access rights.

Another structure for key management in access hierarchies is Cryptree [GMSW06]. Cryptree is a cryptographic tree structure for access control on untrusted storage. It fulfills the three criteria semantics, efficiency, and simplicity. The term semantics refers to having intuitive and useful access control semantics, as well as to supporting confidentiality and dynamic inheritance of access rights. Efficiency means that the number of keys managed should not grow proportionally to both the number of involved users and the number of files. Simplicity demands that the access scheme is easy to understand and implement. Cryptree uses some of the techniques described in Section 3.7.4 to manage the keys. It keeps file and folder names confidential. Moreover, Cryptree supports downward as well as upward inheritance of access rights, and provides access rights confidentiality.

The main data structures of Cryptree are based on *cryptographic links*. There are two kinds of cryptographic links: symmetric and asymmetric links. Let k_1 and k_2 be two keys. The result of encrypting k_2 with k_1 using a symmetric cipher is called a symmetric cryptographic link. Thus, k_2 can be derived from k_1 and the link by decrypting the link with k_1. An asymmetric link is the result of encrypting a key k_2 with a public key. The public key can be stored with the resource, whereas the private key k_1 needs to be kept secret. Hence, k_2 can be derived from k_1 and the link by encrypting the link with k_1. The Cryptree data structures consist of keys and cryptographic links.

One advantage of Cryptree is its ability to grant access rights to a folder including all its subfolders in constant time. Furthermore, Cryptree allows to dynamically inherit access rights, which prevents scattering of access rights. Moreover, Cryptree enables its users to grant access to a file or folder without revealing other resources. Symmetric cryptography is sufficient for most features, i.e. Cryptree does not necessarily depend on costly asymmetric cryptography. However, paper [ABFF09] presents a more flexible approach than Cryptree, as the underlying access control model is based on graphs rather than folder trees. In addition, key derivation scheme [ABFF09] is based on hash functions and therefore more efficient by a constant factor. As Cryptree supports granting fine-granular write access, users may lose track of access rights. Although scheme [ABFF09] requires full access to the graph for granting write access to a user, it supports user hierarchies. On the contrary, Cryptree cannot manage user hierarchies.

In contrast to the two aforementioned schemes, our key management scheme focuses on transparency. Section 3.7 discusses our scheme in detail.

2.4 File Synchronization

The rsync protocol [Tri99, Tri12] allows to efficiently synchronize files and folders from a source folder tree to a destination folder tree. In order to synchronize changes, the source sends a file list containing pathnames and metadata information to the receiver. The receiving side then compares the file list to its local tree, skips unchanged files, and synchronizes changes from the sender. To synchronize changes of a file, the receiver generates checksums of the file blocks and transfers them to the sender. The sender figures out the differences between its and the receiver's file (cf. Section 3.8.2) and sends the changes to the receiver which updates the file accordingly. The file synchronization tool Unison [Pie12b] uses the rsync algorithm for performing updates [Pie12a].

Subversion [Fou12] is a version control system which records changes made to a folder tree, including file content changes. Versioning on the server is done using the bubble-up method [Col00]: in order to reflect changes, a new folder tree is built bottom-up from the change location. The root of the newly built tree is linked with the repository's revision array. Only the latest revision of a resource (file or folder node) is stored completely. Previous revisions of a resource are stored as diffs (difference file) to subsequent resources. Subversion eliminates some problems [Col00] of the Concurrent Versions System (CVS) [Inc12] version control system.

We do not provide all features which version control systems such as Subversion offer. For instance, we do not maintain multiple branches of a directory. However, our own file synchronization approach is efficient, since we transmit file changes rather than whole files. Another strength of our approach is that the server file hierarchy (cf.

Section 3.5) and the client file hierarchy (cf. Section 3.8.1) are simple. Furthermore, we support file versioning. A significant feature of our file synchronization algorithm is that it is able to synchronize unencrypted client files and encrypted server files. Section 3.8 presents our algorithm in detail.

Chapter 3

Design

This chapter outlines the design of the secure network storage service. It discusses design alternatives and justifies design decisions.

Section 3.1 presents the threat model on which our design is based. A classification from a security perspective of our network storage service is provided in Section 3.2. Section 3.3 discusses features of network storage systems and points out which features were considered in the design phase and implementation. Section 3.4 deals with client-server communication. It also discusses how the client authenticates itself towards the server. Section 3.5 deals with the design of the server's file hierarchy. The type of keys our service manages, key generation, and the integration of keys are outlined in Section 3.6. Section 3.7 presents possible ways to manage keys. Section 3.8 explains how clients synchronize their files with the server. Section 3.9 deals with how the client and server handle concurrent access. Section 3.10 explains how the clients and server cope with message omissions as well as system crashes. The communication protocol is defined in Section 3.11.

3.1 Threat Model

In this section, we discuss the threats which influenced the design of our storage security service. Section 5.3 deals with concrete attacks, their implications, and to which extent we counter them.

Paper [HMLY05] links the four CIAA (confidentiality, integrity, availability, authentication) aspects to storage systems. In the following, we discuss each aspect considering the security of our storage system service:

Confidentiality. Adversaries may attempt to read information that is present on the server without proper authorization. Server administrators, hackers which gained access to the server, and parties that collude with the server are poten-

tial adversaries. In addition, former group members are potential adversaries. The information may include client-supplied file contents and file metadata. On the server-side, data may be present on disk or in memory.

Integrity. Adversaries may attempt to modify information on the server without proper authorization. An attack may involve changing, adding, and destroying file content or metadata. Potential adversaries include server administrators, hackers that compromised the server, parties which collude with the server, and former as well as malicious group members.

Availability. The goal of an availability attack is to render the server unavailable. Any entity might attempt to carry out an availability attack. Denial-of-service (DoS) attacks aim to make services unavailable. A DoS attack may involve exhausting storage, network, memory, and computing resources through legitimate usage. Therefore DoS attacks are difficult to prevent.

Authentication. Adversaries may attempt to masquerade as legitimate users in order to read (confidentiality) or modify (integrity) information that is stored on the server. Attackers that masquerade as a legitimate user might also prevent others from accessing the server (availability). Moreover, adversaries may try to masquerade as the server in order to access (confidentiality) or modify (integrity) client information. In addition, an attacker which masquerades as the server might deny others access to the actual server information (availability).

3.2 Security Classification

This section classifies our network storage service from a security perspective. We define the involved players in Section 3.2.1. Additionally, we consider potential attacks in Section 3.2.2. Moreover, we summarize how our service realizes security primitives (Section 3.2.3) and discuss the granularity of protection that our service provides (Section 3.2.4). Our classification is based on the criteria from the framework for evaluating storage system security [RKS02].

3.2.1 Players

The following players interact with our network storage service (cf. Section "Players" of paper [RKS02]).

- We distinguish between owners, readers, and writers. An *owner* is the unique entity that is allowed to enforce and change access rights of a shared folder. Owners may always read and write their data. A *reader* is an entity that is allowed to read data. A *writer* is an entity that is allowed to both read and

write data. Therefore a writer is also a reader by our definition. Owners are writers of their data.

- A *group* is a set of readers, writers, and owners. A group is associated with exactly one shared folder and hence with exactly one owner.
- The *storage server* is the machine that stores client-supplied data. The *group server* manages group access rights. Since the storage server and group server services run on the same machine, we use the general term *server* to refer to that central machine.
- A *client* is the software that communicates with the server. We use the terms client and *client daemon* interchangeably.
- *Users* are persons who run clients.

3.2.2 Attacks

The following list presents potential attackers. Section "Attacks" of paper [RKS02] served as a basis for the construction of the list.

- Registered users who do not belong to a group and unregistered adversaries are potential attackers.
- Evicted group members may be adversarial.
- Malicious group members are potential attackers.

Attackers may either conduct their attacks alone or collude with the server. Each kind of attacker may attempt to carry out the following three attack types (cf. paper [RKS02]):

Leak. Adversaries try to gain access to unencrypted file content when they carry out a "leak" attack.

Change. A "change" attack involves carrying out valid modifications, i.e. modifications which readers cannot detect as incorrect.

Destroy. During a "destroy" attack, adversaries conduct invalid modifications, i.e. modifications which readers are able to identify as incorrect.

In addition, we consider the following three attack types:

- Attackers may be present on the wire and execute "message attacks":
  - Eavesdroppers may attempt to read data that is exchanged between the client and the server over the network.

  - Man-in-the-middle attackers may try to masquerade as the client or server. Moreover, such adversaries may attempt to modify transmitted data.

- Adversaries may attempt to launch denial-of-service attacks.
- Adversaries that have gained access to the group server may start attacks.

We discuss the resistance of our service against attacks which these adversaries may attempt to carry out in Section 5.3.

3.2.3 Security Primitives

The following list discusses to which extent we realize the six security primitives from Section "Core security primitives" of paper [RKS02].

Authentication. Players are authenticated in order to determine their identity and to authorize their actions. Our authentication service is provided by the group server and is therefore centralized. The server authenticates clients and authenticates itself towards clients using the TLS protocol.

Authorization. Authorization enables owners to delegate access to other players, thereby building a group. The owner sets access rights of other players on the group server that authorizes clients. Additionally, the owner distributes keys to the other group members over a secure, out-of-bound channel.

Securing data on the wire. We secure all data on the wire using TLS which provides both data integrity and data confidentiality for transmitted data that is exchanged between a client and a server.

Securing data on disk. As the server might be compromised, clients do not provide the server with unencrypted file contents. Clients encrypt file contents prior to sending them to the server with a symmetric cipher. However, clients do not encrypt metadata.

Key distribution. Owners distribute keys to readers and writers. The group server is not involved in key distribution. Writers and readers use keys to encrypt and decrypt data on the client-side. Apart from the keys that are used to provide confidentiality, our service relies on further, independent keys to offer protected metadata integrity. For each confidentiality key, an integrity key with the same version number exists, while the inverse is also true (cf. Section 3.6.1).

Section 4.4.3 lists the metadata which the server stores. Clients supply the server with metadata `PUT file` requests (cf. Section 3.11). The client-supplied metadata is a subset of the metadata which the server saves. A *metadata* field is either *protected* or *unprotected*. The following metadata fields are protected:

- `is_diff` value
- file size
- cryptographic hash of file content
- key version number
- `extra` value

The following metadata fields are unprotected:

- `id` value
- owner name
- folder name
- file name
- server-generated `modified` time stamp
- history version number
- MAC value

Writers can prove that they have the required keys by providing a keyed MAC based on the concatenation of protected metadata fields as the input message (cf. Section 3.11). Readers can use the keyed MAC in order to check whether the writer was allowed to carry out the change.

As the protected metadata includes a cryptographic hash of the file content, we provide file content integrity. However, clients cannot tell whether files which they receive from the server are *fresh*, i.e. the server might supply clients with an old file rather than with the actually requested, newer file.

Revocation. Revocation involves withdrawing access rights from past group members by changing the group key. We provide member eviction through lazy revocation, as we defer file re-encryption to the next time the file is updated. Clients never re-encrypt files which are already present on the server. Owners revoke access rights from past group members by updating the access rights on the group server as well.

3.2.4 Granularity of Protection

The following list outlines the granularity of protection that our service offers. The criteria are based on Section "Granularity of protection" of paper [RKS02].

- Owners distribute keys to group members who are allowed to access group data. Therefore group membership is distributed. Moreover, the owner sets

access rights on the group server. Hence group membership depends on a centralized entity as well.

- The keys used to encrypt data on the wire are short-lived, as TLS uses session keys.
- The keys for data confidentiality and integrity on the clients are long-lived, as they are only renewed after they had been leaked.

3.3 Features

This section discusses potential features of network storage services. The following features were considered in the design phase and addressed in the implementation (cf. Chapter 4).

Client-side encryption. Client-side encryption is one of the most significant and distinguishing features of our service. While some network storage services such as Wuala offer client-side encryption, many services merely provide server-side or over-the-wire encryption (cf. Chapter 2). The user interactions required to realize client-side encryption are to be kept at a minimum in order to make the service easy to use. This can be achieved by never showing the ciphertexts to the user and implementing a user-friendly key management scheme.

Data integrity. The integrity of file content should be protected. Clients should be able to verify that an authorized client submitted the file content which they receive from the server and that the content was not tampered with. The server should have the possibility to check the integrity of client-supplied file contents.

Group sharing. It should be possible to share confidential data within groups. Groups may be dynamic, i.e. members may leave or join a group during its lifetime. Efficient key management schemes are necessary to provide that feature.

File synchronization. It should be possible to synchronize files with different devices and users. Synchronization should be as efficient as possible, i.e. require minimal user interaction and keep the amount of necessary storage space, computation time, and network traffic as low as possible. The synchronization algorithm design needs to take into account that files on the clients are unencrypted, whereas the files on the server are encrypted.

Live file synchronization. Changes of the client's file tree should be detected and synchronized with the server when they occur. This makes data available to other clients, directly after the data was created.

File versioning. It is desirable to have access to all versions of files in order to recover deleted files, see the development of files over time, etc. File versioning might also contribute to efficient synchronization as changes rather than whole files can be distributed to clients.

Access control. Access to resources should be controllable. The access control policy implementation determines how fine-grained access rights are. Access control is particularly important with respect to group sharing, as groups might be dynamic. Centralized access control can be provided by the server. Owing to client-side encryption, access control becomes decentralized, as clients need to obtain keys in order to decrypt data.

Open protocols. The communication protocols should be open in order to enable developers to program their own client or server. While a closed source client and server from a third party might contain a backdoor, home-brew software can be built backdoor-free. As, among others, Ken Thompson [Tho84] and the case of the potential backdoor in OpenBSD [Tho10] showed, even open source software is prone to backdoors. Additionally, open protocols enable developers to extend the protocol and the functionality.

Changeable algorithms. Users should have the possibility to change the encryption and integrity protection algorithms. Thus, users are able to decide on the strength and other cryptographic parameters on their own.

Usability. The service should be easy to use. Therefore synchronization should be done in the background, i.e. without user intervention. It should be simple to provide keys and to share files. An intuitive Graphical User Interface (GUI) and console interface should be provided.

Portability. All parts of the software should be portable among different operating systems. This can be achieved by using a cross-platform compatible programming language such as Java and making the software compatible with standard file systems.

In the following, we outline features which a network storage system might provide, but which were neither integrated into the design nor into the implementation. However, Section 5.4 suggests extensions for our service.

File name confidentiality. Although our service provides file content confidentiality, we do not encrypt file names. The reasons for this decision are given in Section 3.6.1.

Random file access. Our service provides random access to server files. However, since files on the server might be diffs or encrypted with arbitrary cipher modes, we cannot always randomly access plaintext bytes of entire logical files (cf. Section 3.8.2).

Intrusion protection. The associated risks of a client or server attack should be

reduced. Trusted Platform Modules (TPMs) could be used to protect the key material on the client.

Searching. Functionality which allows clients to search for file names in the server's history could be implemented, assuming that file names are unencrypted. File content search on the server-side can only be provided for unencrypted files.

Quotas. A desirable feature is to limit the storage space, network traffic, and connection time on a per-user basis. Moreover, quota could be allocated per group.

Data compression. File content compression could be provided by the software (cf. Section 5.4). However, ciphertext compression is unlikely to reduce the file size significantly, as ciphertexts typically possess a high information entropy. Therefore it is recommendable to compress files prior to encryption. Some file systems provide compression.

Data redundancy. Data redundancy increases data availability. The client or server could make use of error-correcting codes for this purpose (cf. Section 5.4). Moreover, data could be stored redundantly with a RAID.

Deduplication. De-duplication on the server-side allows to save storage space. However, file deduplication becomes difficult when users employ cipher modes of operations which rely on a random initialization vector (IV). For example, CBC (Cipher Block Chaining) is such a mode. Encrypting identical plaintexts using different IVs, results in different ciphertexts. The software or the file system could take care of deduplication.

File links. A file link is a reference to an actual file. File links could be supported by the file system. Moreover, it is possible to implement logical file links into the software (cf. Section 5.4).

3.4 Client and Server Interaction

This section deals with interactions between clients and the server. Section 3.4.1 provides a general overview of the interactions. Section 3.4.2 explains how clients authenticates themselves towards the server.

3.4.1 General Overview

Our service has a single, central server that stores data. Multiple clients connect to the server and synchronize files which users save in a directory that we call the working directory (cf. Section 3.7). Users may synchronize their files with multiple devices and share folders in dynamic groups. In order to synchronize files, clients

connect to the server, authenticate themselves, and retrieve possible file changes from the server. Afterwards, clients submit their own working directory changes to the server (cf. Section 3.8.1). Clients typically connect to the server over the Internet, although the clients and server might be in the same LAN (Local Area Network) as well. Figure 3.1 illustrates the architecture of our service. Note that it only visualizes the most fundamental components.

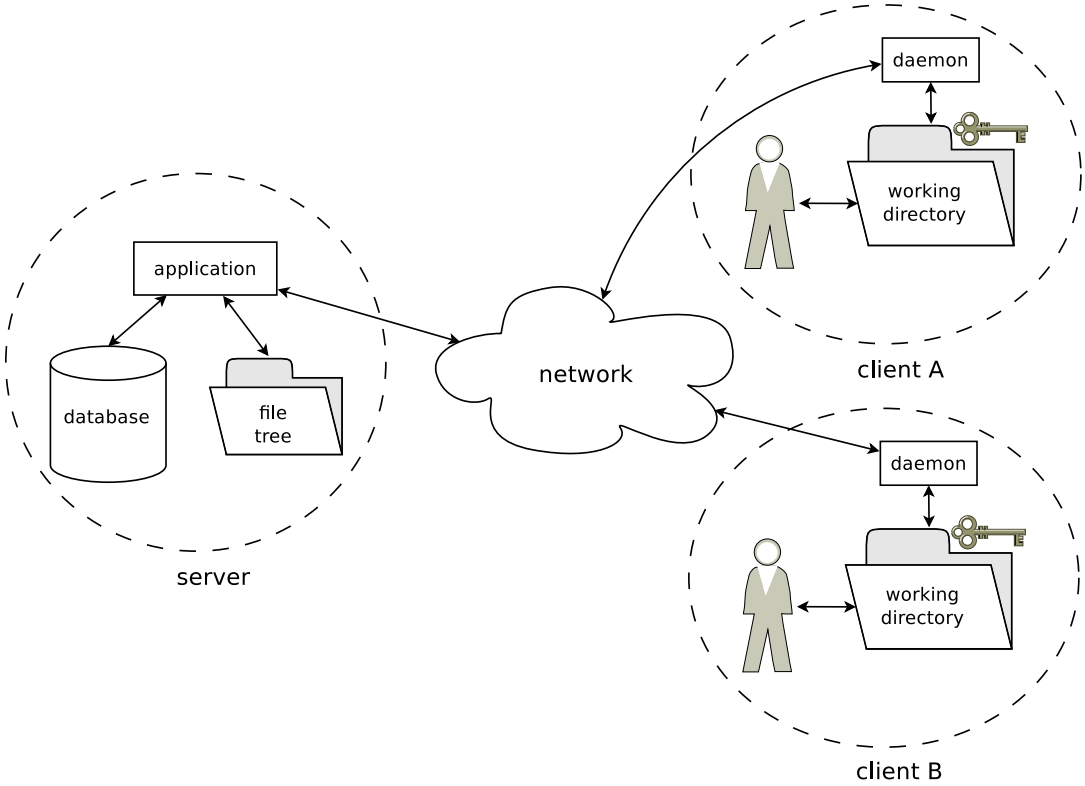

Figure 3.1: Architecture of our secure network storage service.

A daemon on the client computer takes care of synchronizing the user's data with the server. W. Richard Stevens defines the term *daemon* as "a process that executes 'in the background' (i.e., without an associated terminal or login shell) either waiting for some event to occur, or waiting to perform some specified task on a periodic basis" [Ste90]. This definition matches our notion of a daemon: The daemon runs in the background without user intervention and synchronizes the user's data periodically. The daemon also waits for changes of working directory files and synchronizes detected changes to the server immediately. The working directory contains the file keys as well (cf. Section 3.6.3).

Since the client daemon adheres to existing client file systems, users may access their files directly. However, if our service was integrated into a file system, users would have to attach the file system to their operating system prior to accessing any files. Furthermore, we would need to port the file system to the user's operating system. An advantage of integrating a storage service into a file system is that the

corresponding software sees file changes on a low level when they are made. This facilitates file synchronization. In contrast, we gain cross-platform compatibility by layering our service on top of existing file systems. In addition, users have the possibility to employ their preferred client file system which offers desired features. For example, the file system could implement redundancy measures.

Interactions obey to the standard three-tier client and server model [Ree00]. In a three-tier architecture multiple clients request data from and submit data to the server. Our server application processes client data on the server machine and saves files in a folder on the server's file system. We call the folder on the server which contains the client-supplied files the server file tree (cf. Section 3.5). The server keeps metadata and further higher-level data in a database (cf. Section 4.4.3). The clients are unable to determine how data is stored on the server, as they use a higher-level protocol (cf. Section 3.11) to communicate with the server. The server is able to change its business logic, while still providing its services to clients, if the fixed presentation tier (the protocol) remained unchanged. Moreover, the data tier on the server may be changed without affecting the business logic tier of the server (cf. Section 4.4.3). Clients do not need to adapt to the server's internal changes.

All connections between the client and server are secured with the Transport Layer Security (TLS) protocol [T. 08]. As TLS is built on top of the Transmission Control Protocol (TCP) [Pos81], TLS shares TCP's properties such as reliable data transmission, error detection, flow control, and congestion control. Additionally, TLS provides connection security with the properties privacy and reliability [T. 08]. Privacy is provided by data encryption using symmetric cryptography. Checking the integrity of messages using a keyed MAC makes the connection reliable. The TLS Handshake Protocol allows the server to authenticate itself towards the clients as well as to securely and reliably negotiate a shared secret [T. 08]. Although the TLS Handshake Protocol also enables clients to authenticate themselves towards the server, we take the approach described in Section 3.4.2 for the sake of user-friendliness.

Rather than relying on a Public Key Infrastructure (PKI) [NIS09], we embed the server's public key into the client program. Web browser suppliers also employ this both effective and simple approach, by integrating trust lists (lists of trust anchors which in turn combine a public key and the name of the entity possessing the corresponding private key) into browsers [Coo05]. However, when the public key of the server changes, the new public key has to be distributed over a secure channel. For example, the public key could be released on a web page which the client trusts, or exchanged over a secure out-of-bound channel.

3.4.2 Authentication

Secure authentication can be realized using symmetric or asymmetric cryptography. In both cases, a challenge-response protocol may handle the authentication process. With symmetric cryptography, the client and the server share the same key, whereas with asymmetric cryptography, the client stores a public and private key, while the server only possesses the client's public key. An advantage of challenge-response authentication using symmetric cryptography over asymmetric cryptography is that users may select passwords which they can memorize. Moreover, symmetric-key encryption algorithms are typically significantly faster than public-key (asymmetric) encryption schemes [MVO96]. However, if the user picks a weak symmetric key, the security of the service is at stake. On the contrary, asymmetric key pair generators are supposed to produce strong keys in any instance. Furthermore, if an attacker obtained access to the server and asymmetric cryptography is deployed, she could only steal the public key, but not the private key which is crucial for authentication. We gain the same benefit using symmetric cryptography, if the server stored a cryptographic hash rather than the cleartext of the symmetric key. An example for an authentication protocol which is based on symmetric cryptography is the Kerberos protocol [Neu05]. The PPP EAP DSS Public Key Authentication Protocol [Wil97] is an authentication protocol using asymmetric cryptography.

The PPP CHAP (Challenge Handshake Authentication Protocol) [W. 96] is a flexible protocol for two-party authentication. A drawback of CHAP is that it requires the server to store the secret in cleartext. This should be avoided, since we do not ultimately trust the server. Merely reading the secret from the server storage is enough to break the security.

With TLS, the simplest form of password authentication could be used instead, while still foiling eavesdropping attempts: Clients send their passwords in cleartext to the server over the secure TLS connection. Instead of saving the password in cleartext on the server, the server can store a derived key. On the client-side, the password is saved in cleartext which is conform to our security classification (cf. Section 3.2). RFC 2898 [B. 00] deals with password-based cryptography and presents a method to derive pseudo-random keys from a password. The presented function PBKDF2 takes a password, a salt (random byte string), an iteration count, as well as the intended output length in bytes and outputs a derived key after a number of steps determined by the iteration count. During each iteration a pseudorandom function (PRF) is applied to certain input combinations and intermediate results. The pseudorandom function takes two arguments and can be a keyed HMAC (Hash-based Message Authentication Code). The default PRF with respect to document [B. 00] is HMAC-SHA-1 [Kra97]. An increased iteration count increases the cost of producing the derived key, while also increasing the difficulty of attacks [B. 00]. In our authentication scheme, the server stores a random salt and the derived key using PBKDF2 from the user's cleartext password in its database (cf. Section 4.4.3). The

server software defines a constant iteration count and therefore the iteration count is not stored in the database as well. An opponent who steals the server's database will face difficulties to derive the cleartext password from the derived key, the salt, and the iteration count. Each time a client sends its cleartext password to the server in order to authenticate itself towards the server, the server applies PBKDF2 to derive a key and compares the result to the respective key from the database. If the result and the stored key matched, the server accepts the authentication request from the client. Otherwise, the server rejects the client's authentication request. A user name is also sent by the client in order to allow the server to identify the client.

Note that there are various other authentication schemes such as Lamport's password scheme [Lam81] on which S/KEY [Hal95] is based. Another option is to use the authentication functionality from TLS in order to authenticate clients towards the server. However, as it is easier for the average human to memorize password strings rather than strong private keys, we did not employ TLS client authentication. It goes without saying that the private key could be directly integrated into the client, but if users had multiple devices, they would need to securely transfer their private key to all devices. In contrast, a password can be entered on all devices manually without much effort. Overall, our scheme is both simple and secure.

The server authenticates itself towards the clients using TLS. Our client embeds the server's public key in order to be able to check the authenticity of the server. Moreover, our client aborts server connections when the server does not authenticate itself.

3.5 Server File Hierarchy

This section discusses how the file hierarchy on the server can be managed. Note that it is not necessary to manage the file hierarchy on the client. The client daemon gets a pointer to the folder containing the files to mange from the user. It takes the structure as is, and does not change it.

On the server-side, files are stored on a persistent storage device sporting a local or network file system. One of our goals is to make file storage on the server as flexible as possible. Therefore files may be stored at arbitrary locations on any file system supported by the operating system. Furthermore, this property facilitates backing up data, as standard backup tools can be used. The server administrator determines where the files are supposed to be stored. We could enrich our service with additional features, by deploying an appropriate file system. For example, we could choose a file system that automatically mirrors its content into the cloud.

Metadata is stored in a database for the sake of fast-lookup, concurrency, automatic constraint checking, as well as other features a database provides. In addition, the business logic of the server checks and processes client input, before any client-

supplied data is pushed into the database. Files are stored on the file system rather than in a database owing to the aforementioned advantages.

The server neither deletes nor modifies valid files that had been received from a client in order to maintain the features data integrity and file versioning (cf. Section 3.3). A file is valid, if the client-provided checksum matched the checksum of the received file, and if a corresponding metadata entry existed in the database. The server neither removes nor changes written metadata entries.

Our server file hierarchy design must support multiple users. Each user has a single private folder and may manage multiple shared folders. Managing means that the user takes care of group key management (cf. Section 3.7) and administrates access permissions of shared folders. We call the user who manages a shared folder the owner of the folder. The owner of a folder is the only entity with the permission to create and change the folder's access rights (cf. Section 3.2.1). Therefore each folder is associated with a unique owner. The server database stores the access rights of shared folders (cf. Section 4.4.3).

As the file name provided by the client may originate from different file systems implementing different naming conventions, names should be converted to a common representation. Note that if clients encrypted file names, the server would not be able to transform them. Therefore we convert file names on the client-side only. The server stores and retrieves names as they were originally supplied by the client.

On the server, files are identified by an owner name, a folder name, and a file name. Representing file names with the generic URI (Uniform Resource Identifier) syntax from RFC 3986 [Ber05] resolves the name representation issues outlined above. A folder name can refer to the private folder of the user or to a shared folder. It only consists of up to ten lower-case alphanumeric characters for file system compatibility purposes. In addition, it begins with a letter. The file names clients provide are relative to the storage folder path on the server. The server maintains a history (cf. Section 3.8.1) of the client's changes in its database (cf. Section 4.4.3). The combination of the user name, the folder name, and the file version serves as a candidate key for the history table (cf. Section 4.4.3). The version number is a positive integer that the server manages independently for each user and folder name combination (cf. Section 3.8.1).

Let us illustrate our server file hierarchy concept with an example. Let `f` be the path to the storage folder on the server's file system where all client files are supposed to be stored. A server administrator sets the file storage folder in the server's configuration file (cf. Section 4.2). Let `/` be the file path name component separator. Then the server saves the private file with version number `v` of a user `u` under `f/u/private/v`. For a shared folder named `s`, the location of a corresponding file from user `u` with version number `v` on the server's file system would be `f/u/s/v`. Note that the location of a client-supplied file on the server file system merely depends on the storage folder, the user name, the folder name, and the version number.

Consequently, the server does not use the file's name to determine a file's location. The server stores file names as metadata in the server database (cf. Section 4.4.3). A file name may refer to files which are on any level in the client's file hierarchy. The only requirement is that file names are relative to the client's working directory (cf. Section 3.7).

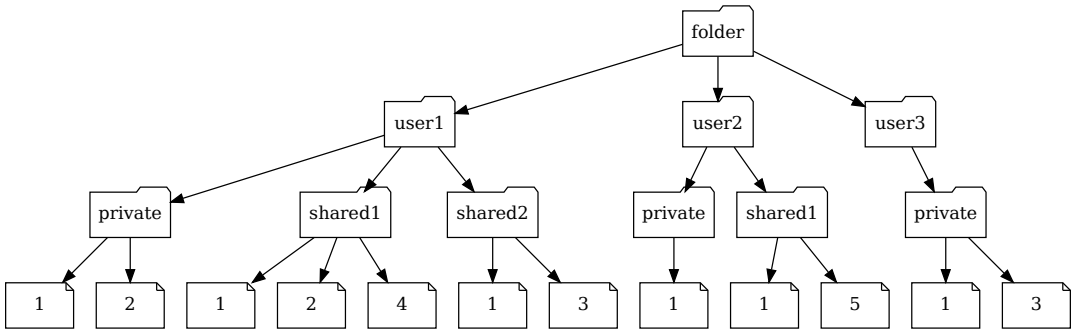

Figure 3.2: Server file tree example.

Figure 3.2 shows an instance of a server file tree. In this example, the storage folder is named `folder` and there are three users called `user1`, `user2`, and `user3`. Each user has at least one private file. User `user1` owns two shared folders with the names `shared1` and `shared2`. User `user2` has a shared folder that is also named `shared1`.

An advantage of our server file hierarchy scheme is that the server's file system only needs to support lower-case alphanumeric file names. Another advantage is that the server file names are short, since the file's version number is used rather than the user-supplied file name of almost arbitrary length (the user-supplied file name length is limited by the message length; cf. Section 4.6). Thus, we support file systems with high file name length restrictions.

3.6 Keys

This section describes which types of cryptographic keys are used in the service (Section 3.6.1). Furthermore, it is explained how keys are generated (Section 3.6.2) and integrated (Section 3.6.3) into the system. Keys are crucial for the security features client-side encryption and data integrity (cf. Section 3.3) which the system provides.

3.6.1 Key Types

The server stores a public/private key pair that is used for server-authentication (cf. Section 3.4.2) and TLS connections (cf. Section 3.4.1). Additionally, the server

stores derived keys and salts of client passphrases which are needed for client-authentication (cf. Section 3.4.2). Further keys are not present on the server-side.

On the client-side, there are two types of keys:

Access keys. A client stores symmetric keys for client-side encryption, client-side decryption, and data integrity (cf. Section 3.3). We call keys that are used for client-side encryption and decryption *content keys*. Keys which are associated with data integrity are named *integrity keys*. Content and integrity keys are independent of each other, i.e. content keys are not derived from integrity keys or vice versa. Moreover, each key is chosen randomly (cf. Section 3.6.2). Access keys are stored unencrypted in a file called *access bundle* (cf. Sections 3.7, 4.3). An access bundle contains a key version number and a cryptography algorithm name along with each key. For each confidentiality key, an integrity key with the same version number exists. The inverse is also true, i.e. for each integrity key, a confidentiality key with the same version number exists. As for content keys, the algorithm name refers to the name of a symmetric cipher. The name may contain further details such as the mode of operation or the name of the padding algorithm to use. As for integrity keys, the algorithm name refers to the name of a MAC implementation. Access keys must not be handed over to the server or an untrusted party. However, when cipher modes of operation which depend on more inputs than a key and plaintext, such as CBC, are used, the extra inputs may be stored as metadata on the server (cf. Sections 3.11, 4.1.3, 4.4.3).

Authentication credentials. Authentication credentials consist of a username and password string. The client sends the username and password combination to the server during client-authentication (cf. Section 3.4.2). Users store credentials in plaintext in a configuration file (cf. Section 4.2). In order to register a user on the server, the client needs to submit a `PUT auth` message (cf. Section 3.11) to the server. We provide a tool that allows to register and update user credentials on the server (cf. Section 4.8.1).

As our service does not provide file name confidentiality, there are no keys for this purpose. A problem is that file name confidentiality prevents the server from checking the validity of client-supplied requests, if the name key changed or if a file name was encrypted using a cipher mode which relies on random inputs such as an IV. A client request is valid, if the request is syntactically correct and consistent with the server history. For example, a syntactically correct file deletion request is valid, if the corresponding file existed according to the server history. If either a constant key and no random input or no key were used, the server is able to distinguish between the operations *add* and *modify* on its own, when the client puts a file (`PUT file`; cf. Section 3.11) on the server. Moreover, the server can always verify the parameters of a *rename* (`POST move`; cf. Section 3.11), or *delete* (`DELETE file`; cf. Section 3.11) request. It is important that the server checks the validity of requests, as otherwise a client could render the server history inconsistent (cf. Section 3.10.1).

However, having only a single key in order to let the server check the validity of requests, interferes with the concept of lazy revocation (cf. Section 3.7.4). To provide both file name confidentiality and server checks, all file names in the history (cf. Sections 3.8.1, 4.4.3) could re-encrypted with an undisclosed key by a client that possesses all file name keys. Re-encrypting involves decrypting file names which are encrypted with an old key and encrypting them with the new key. However, this approach is inefficient and does not honor lazy revocation. Section 5.4 outlines another way to integrate file name confidentiality into our design.

3.6.2 Key Generation

The tool from Section 4.8.2 is able to produce random keys. Users must provide the tool with their desired key length in bits. The tool lets the most preferred, cryptographically strong random number generator (RNG) of the system generate the random bits of the key.

An advantage of automatic, cryptographically strong, random key generation is that the resulting keys are unguessable and truly random [Eas05]. Furthermore, automatic generation is convenient and prevents users from picking weak keys and providing keys in the wrong format (encoding, length, etc.). Additionally, users usually do not know their keys by heart, as they possibly have not even seen them. This circumstance could foil social engineering attempts.

The key length is a significant parameter, as the security of the cryptographic system depends on it. There are various public and private organizations which recommend different minimum key lengths, even for the same cryptographic system. Page [Dam12] summarizes the reports from several organizations and implements mathematical formulas which allow visitors to quickly evaluate the minimum recommended key length for their systems. The access bundle tool from Section 4.8.2 provides recommendations for cipher and MAC algorithms whose key lengths fulfill the proposals from page [Dam12]. However, users may choose arbitrary algorithms.

Note that another idea which was considered but discarded during the design phase, is to have master keys, i.e. keys from which other keys are derived. Deriving a key from the master key could be achieved by padding the master key with constant strings and hashing the result with a cryptographic hash function. The advantage is that less keys need to be stored and distributed. Moreover, since the output of the cryptographic hash function is pseudo-random, derived keys are also pseudo-random. The master key cannot be deduced from the derived key, since a cryptographic hash-function is one-way. However, this approach has several disadvantages: In order to obtain a pseudo-random key this way, an appropriate cryptographic hash function has to be chosen. If attacks on the hash function emerge, the security of the system will be at stake. For example, if preimage resistance is broken, an attacker who knows a derived key can produce the master key and consequently any other derived

key. Moreover, the output length of the hash function must be greater than or equal to the desired key length. The input length of the hash function, i.e. the length of the master key, must be sufficiently long as well in order to generate strong keys [Kra97]. As users may specify the length of derived keys, a hash function cannot be chosen and integrated in advance. Moreover, users may demand different key lengths, as they may specify arbitrary cipher and MAC algorithms (cf. Section 4.1.3). Since security is a top priority, users may and should generate each key independently from other keys.

In contrast to access keys, we do not automatically produce authentication credentials (cf. Section 3.4.2). User pick credentials themselves and set them in a configuration file (cf. Section 4.2). Therefore, users are able to choose a password that they can remember, although their selected password might be guessable.

3.6.3 Key Integration

This section explains how keys are integrated in the system. All access keys are stored only on the client-side in files named access bundles (cf. Sections 3.7, 4.3). There may be a single private access bundle containing the keys for the private files of a user. The private access bundle is only present on the clients belonging to the corresponding user. Additionally, a group access bundle per shared folder and per client must exist. The keys in access bundles serve to provide client-side encryption as well as decryption and data integrity. The members of a group have identical access bundles which they use to access their commonly shared folder. If each key was generated independently, attackers are unable to deduce unknown keys from leaked keys.

Authentication credentials are stored in the user's configuration file (cf. Section 4.2), while the server stores a username, a derived key of the password, and a salt per user (cf. Section 3.4.2). The server's public key is embedded into the clients in order to be able to check the server certificate (cf. Section 3.4.2). The server's private key is only present at the server.

3.7 Key Management

Group keys are a set of content and integrity keys (cf. Section 3.6.1) that group members possess. To provide file sharing among group members, group keys need to be managed. We consider dynamic groups, i.e. members can join and leave the group during its lifetime. As members of a group may still possess the keys of their group when they leave, the group needs to create a new key and distribute it among the current members. When a member joins a group, it needs to get keys from the group. Group key management research addresses these problems by

proposing rekeying schemes for dynamic group memberships [JA03]. Moreover, as there may be several shared folders in the user's file hierarchy, a scheme to manage the keys for the hierarchy itself is required. Section 2.2 presents related group key management work, while Section 2.3 deals with related work concerning key management for access hierarchies. Although the related work influenced the design of our key management scheme, we pursuit a novel approach which we present and discuss in Section 3.7.1. Sections 3.7.2 and 3.7.3 describe how we manage content and integrity keys, respectively. Key update algorithms are discussed in Section 3.7.4.

3.7.1 General Overview

One of the design goals of our service is simplicity which makes the workings of our service transparent to the user (cf. Chapter 1). We aim to reach this goal through a straightforward key management scheme.

As for group key management, our system stores group keys in access bundles on the client-side (cf. Section 4.3). In order to provide key distribution, the system does neither rely on an architecture such as the Multicast Security (MSEC) Group Key Management Architecture (cf. Section 2.2) nor on a central, group database server as Cepheus (cf. Section 2.2) does. Instead, we adopt the approach from Chefs and Plutus (cf. Section 2.2), i.e. we let the resource owner distribute group keys over a secure, out-of-bound channel. The owner could, for instance, send the keys to the group members wrapped in encrypted e-mail messages or hand them over in person on a digital medium. However, users may transfer their private access bundle to other devices they possess, but must not pass it to other persons, since private access bundles contain keys for unshared files. As a consequence, we keep group key management transparent and avoid a single point of failure.

We built a tool (cf. Section 4.8.3) which allows to create shared folders on the server-side. Furthermore, our tool may be used to change access rights of a shared folder any time. Access rights are the combination of user names and permissions for each user. The server stores and enforces access rights. The supported permissions are read-only, read/history and read/write/history. History refers to read-only access to the server's synchronization history (cf. Section 3.8). Read-only access enables users holding that permission to download files from the server, but not to view the history. Therefore a user with read-only permissions needs to get pointers to files from a person who is eligible to view the history, in order to download the files. The owner of the shared folder is the only person which has the permission to modify access rights. Thus, other members of a group cannot interfere and change access rights without the permission of the owner. Consequently, group members are not able to revoke the owner. Furthermore, the owner of a folder possesses all access rights for the folder at all times. Only authorized clients are allowed to set access rights for folders they own. A dummy access right named public allows public file sharing. Any user may read the files and view the history of a folder that possesses

the public access right. Having only a single group member managing access rights prevents coordination problems, as only the owner produces keys, distributes them, grants and revokes member access.

Thesis [Fu05] distinguishes between the terms group member *eviction* and group member *revocation*. While an eviction prevents former members from accessing future content but not past content, a revocation prevents former members from accessing both past and future content. Our system supports member evictions, as owners of shared folders create and distribute a new key to the current group members, after a group member had left. Owners also command the server to deprive evicted members from access rights. Consequently, former members are unable to access content after their eviction. Strict revocation requires re-encrypting past content with a new key. However, as this causes a lot of overhead and former members possibly have a copy of the plaintext, past content stays on the server as is. Owners are able to prevent former members from viewing past content on the server by depriving them from their access rights on the server. However, this measure cannot stop former members from colluding with the server and thereby gaining access to past content despite missing access rights (cf. Section 5.3). When a member joins a group, the owner sends her the access bundle over a secure, out-of-bound channel. The sent access bundle must include the current key and may include former keys, depending on whether the new member is allowed to access past content.

When a user authenticates itself towards the server or creates an account on the server, the client sends the user-specified cleartext password to the server over a secure channel (cf. Section 3.4.2). The server stores only the derived key of the cleartext password in its database. Neither the server nor the user are permitted to distribute the password. Otherwise, offenders might spoof the identity of the user, thereby gaining all privileges of the user.

Key management schemes for access hierarchies such as the ones discussed in Section 2.3 are flexible and support complex hierarchies which may become opaque to the managing user. In contrast, we propose a simple folder hierarchy structure. Let `w` be the folder on the client machine that contains the files to synchronize with the server. We call `w` the *working directory*. All files which are directly located under `w` are private. Moreover, only direct sub-folders, i.e. folders `f` with the position `w/f` on the file system, can be shared. The folder hierarchy may be arbitrarily deep. If the *private access bundle*, i.e. the access bundle for private files which is located directly under `w`, does not exist, private files are not synchronized. This behavior prevents users from accidentally uploading files in plaintext over the secure connection. However, shared folders which must also contain an access bundle are synchronized regardless of the existence of the private access bundle. Users must put a *group access bundle* inside shared folders to share the content. Access bundle declare keys as well as the confidentiality and integrity algorithms to use along with them (cf. Section 4.3). If a folder does not contain an access bundle, the client

daemon will treat it like the private folder. Requiring the presence of a group access bundle inside shared folders, prevents users from accidentally sharing private folders. The client daemon does not transmit any content of access bundles to the server.

3.7.2 Content Key Management

Our system supports the concept of lockboxes (cf. Section 2.2), as the cipher algorithm implementation (cf. Section 4.1.3) is exchangeable. The server offers clients the possibility to store and later retrieve arbitrary information about a file through a metadata field named *extra* (cf. Sections 3.11, 4.4.3). The cipher algorithm implementation could use the extra field for the content of the lockbox. The lockbox may be encrypted with the content key and contain a random key for the corresponding file. As the content key is symmetric, only one lockbox per file is necessary and therefore schemes such as NNL (cf. Section 2.2) do not apply to our situation. Furthermore, the system supports cipher modes which depend on random input such as CBC through the extra field, since cipher algorithm implementations can store the random input under extra. The system supports filegroups (cf. Section 2.2), as an access bundle belongs to a folder and henceforth the keys inside an access bundle may be used for multiple files. We discuss the default cipher algorithm implementation in Section 4.1.3.

3.7.3 Integrity Key Management

Data integrity is supported by the system with the aid of an exchangeable integrity algorithm. For example, the default integrity algorithm implementation (cf. Section 4.1.3) protects certain metadata fields of files by computing a MAC using the corresponding integrity key of the client's access bundle. The integrity protected metadata of a file includes a cryptographic hash of the possibly encrypted file and further metadata fields (cf. Section 3.2). The client-supplied MAC values become part of the file's metadata on the server in a column called *mac* (cf. Sections 3.11, 4.4.3). The server compares the hash of received data to the hash value of the metadata, and accepts the file only if the hashes matched. As the server would potentially need to be adapted in order to implement a custom, user-supplied hash algorithm, the hash algorithm is fixed. Clients check the integrity of metadata before they request the actual data from the server. Malicious group members who have the latest integrity key, are able to generate valid metadata MACs for arbitrary file content that they submit. However, other group members are unable to identify the malicious group member, as every group member possesses the same integrity key that was used to generate the MAC (cf. Section 3.2).

The system basically supports the integration of signature algorithms. When a signature scheme is employed, each user would create an asymmetric key pair. While

the private key must be kept secret, the public key could be stored on the server and linked with the user name. Clients would create a signature of the protected metadata using their private key and submit the result to the server. The server is then able to verify the changes of a client based on the client-supplied signature and the public key.

An advantage of our MAC-based approach is that it is performant, as symmetric cryptography is used. Additionally, the approach is simple, since the server does not need to manage public keys. A disadvantage is that the server cannot verify the integrity of protected metadata itself. However, the server does not accept data from unauthorized clients. Therefore clients need to submit their authentication credentials over the TLS connection to the server, prior to sending data (cf. Section 3.4.2).

3.7.4 Key Updates

Key updates are necessary to prevent unauthorized parties from accessing confidential data after a key leak or a group member eviction. *Lazy revocation* [KRS+03] is a concept tied to key updating which is in particular useful when key updates occur on a regular basis. Lazy revocation delays re-encryption until it cannot be avoided anymore, i.e. when a file is to be updated. The underlying idea is that revoked readers may still have a copy of old content and that therefore lazy revocation does not severely impair security. However, revoked writers should be deprived from their access rights on the server immediately. Revoked readers may still have access to the content they had access to before their revocation, but not to new content, since new content is encrypted with a new key. A disadvantage of lazy revocation is that it may result in multiple fragments of files encrypted with different keys. This complicates access management, as possibly multiple keys are needed per file.

Our system uses the concept of lazy revocation: Whenever a member joins a group, the owner of the corresponding group folder generates a new key and distributes it to all group members. Moreover, after a group membership change, existing content is not re-encrypted, as former members might still possess a copy of the content, which they had been allowed to access prior to their eviction (cf. Section 3.7.1).

To overcome the issues of having multiple keys per file, paper [KRS+03] introduces the concept of *key rotation*. With key rotation, it becomes possible to have a single key from which every previous key can be derived. Future keys must not be derivable from the current key. However, thesis [Fu05] points out that key rotation suffers from a design flaw: It ensures unpredictability for future keys, but does not guarantee that future keys look pseudo-random to evicted members. Therefore thesis [Fu05] presents a scheme named *key regression* which serves the same purpose as key rotation, but provides pseudo-randomness. Thesis [Fu05] introduces the three key regression schemes `KR-SHA1`, `KR-AES`, `KR-RSA`, and proves their security. Each

key regression scheme consists of four algorithms: setup, wind, unwind, and key derivation.

`KR-SHA1` is similar to Lamport's password scheme [Lam81] and uses two distinct cryptographic hash functions h_1 and h_2. In the setup phase, the publisher (or the group manager) hashes a random number x with h_1 at first. Then the hash output is hashed with h_1 and the resulting hash output is hashed using h_1 again. In order to get maxwind keys, h_1 needs to be applied maxwind $- 1$ times. The random number x, the intermediate results, and the last result are indexed in descending order such that x gets label $\text{stm}_{\text{maxwind}}$, while the last result gets label stm_1. At first, the publisher distributes the member state stm_1 to the group members. After a group member revocation, the publisher releases the next member state stm_2 to the members. In the wind phase, the publisher increments the counter for the number of released states. The actual key used is the result of applying h_2 to the current member state stm_i where $i \in \{1, 2, \ldots, \text{maxwind}\}$ (key derivation). A member who possesses stm_i can compute any previous state stm_j where $j < i$ by applying $i - j$ times h_1 (unwind operation). Note that computing stm_k from stm_i where $k > i$ is infeasible, as h_1 is one-way.

The key regression scheme `KR-AES` works basically as `KR-SHA1`, but uses $h_1(x) := \text{AES}_x(0^n)$ and $h_2(x) := \text{AES}_x(1^n)$ where n is a supported block length in bits and x is the key input for the AES block cipher [NIS01]. Note that the plaintext is fixed and that h_1 as well as h_2 are one-way [Fu05].

`KR-RSA` uses the RSA cryptosystem [RSA78] to provide key regression. In the setup phase, the publisher creates an RSA key pair, distributes the public key, and keeps the private key secret. In the wind phase, the publisher encrypts the current member state with the private key. Initially, a random member state is created by the publisher and distributed to the group members. In order to unwind a member state, i.e. derive the previous member state, a group member encrypts the member state with the public key. By applying a cryptographic hash function to a member state, the actual key is derived. In contrast to `KR-SHA1` and `KR-AES`, the number of times the wind procedure of `KR-RSA` can be invoked, is practically infinite. However, as the underlying group of the RSA algorithm is finite, the member states cycle.

Our system does not adopt key regression, although `KR-RSA` requires only constant key storage space. Instead, we store every single version of a key in an access bundle that the owner distributes over a secure, out-of-band channel. Therefore the key storage space that our service requires grows linearly with the number of keys. Although our approach demands more transmission and storage overhead than key regression, the computation overhead is lower, as all keys can be directly retrieved from the access bundle and do not need to be derived. Moreover, the key distributor (the owner of a shared folder) can select past keys which she wants to distribute to a new member. The distributor is therefore able to restrict access to past content. This is impossible using key regression, since then all past keys can be derived from the current key.

3.8 File Synchronization

This section deals with file synchronization, i.e. it explains how the server handles updates of files from the working directory of a client and vice versa. Section 3.8.1 outlines possible file synchronization methods and explains our concept. Methods for computing file differences are discussed Section 3.8.2. Section 3.8.3 explains how the system handles synchronization for a general scenario, thereby discussing how the system deals with conflicts.

3.8.1 General Concept

As our system should support file versioning, we record changes unlike rsync (cf. Section 2.4) in order to be able to revert to previous file versions. Furthermore, this enables us to incrementally update clients. In addition, our approach is simpler than Subversion's bubble-up method (cf. Section 2.4): Instead of storing trees reflecting the state of a folder hierarchy at a revision, we use a log of changes. A *change* can be an add, delete, modify, or rename action. We call the log of changes on the server a *history*. Each log entry contains a version number, i.e. a positive integer. The first change gets version number 1, while the version number is incremented by 1 with each subsequent change. The server takes care of numbering and since a folder name belongs to a unique owner (cf. Section 3.5), the server manages history version numbers independently for each owner and folder name combination.

The file version number is identical to the version number in the history. In the version control system Concurrent Versions System (CVS) [Inc12], however, each file has its own version number. By using the same version number for files and the corresponding entry in the history, a happened-before relation for all file events is established. Moreover, this approach facilitates finding the corresponding history change for a file and vice versa. Another advantage over dedicated version numbers per file is that resource renames are easy to handle. For example, let us use a dedicated version number for each file. Let `A`, `B`, and `C` be files names. Assume that file `A` is at revision 1, file `B` is at revision 100, and file `C` does not exist. Now rename file `B` to `C` which results in the creation of `C` with revision 1. Renaming file `A` to `B`, would create revision 2 of file `B`. However, a revision 2 of file `B` already exists and therefore revision 2 of file `B` became ambiguous, as it refers to two different physical revisions of a file. If file revision numbers match the corresponding history version number, a rename will not lead to ambiguous file revisions, as the history version number is unique for each owner and folder name combination. Consequently, each file revision number is unique in the folder tree to which the file belongs.

```
1 void synchronize(String owner, Path folder, Change[] proposedChanges) {
2   int version = readVersionFile(owner, folder);
3   Change[] serverChanges = sendGetSync(owner, folder, version);
4
```

```
5   for (change : serverChanges) {
6     boolean success = applyLocally(change);
7
8     if (!success) {
9       error("Cannot apply server change " + change.toString());
10      return;
11    }
12  }
13
14  commit(proposedChanges, owner, folder);
15  commitLocalChanges(owner, folder);
16  sendPostSyncDone();
17  writeVersionFile(owner, folder);
18 }
```

Listing 3.1: Client synchronization algorithm in pseudocode.

Listing 3.1 shows our client synchronization algorithm in pseudocode. Each folder of a client with a different access bundles is synchronized independently, as each access bundle refers to a different owner and folder name combination (cf. Section 4.3). The synchronization algorithm updates the files in the working directory and another directory named the *synchronization directory* accordingly. The synchronization directory (cf. Section 4.4.1) contains files that are already synchronized with the server. Only unencrypted files are saved on the client in order to be able to efficiently compare the files in the synchronization directory to the files in the working directory. Clients need to remember their *synchronization status*, i.e. clients must store which changes from the server they have already committed locally and which changes they have submitted themselves. The synchronization status is saved by keeping the version number of the latest change in a so called *version file* (cf. Section 4.4.1) under the synchronization directory. After a successful synchronization, the content of the synchronization directory matches the content of the working directory apart from management files such as access bundles and version number files.

Clients synchronize with the server by first reading the current version number from the version file (cf. Listing 3.1, l. 2). Then clients request (cf. Listing 3.1, l. 3) and apply the changes from the server's history which they have not synchronized yet (cf. Listing 3.1, l. 6), i.e. changes whose version number is larger than the read version number. If and only if a client has successfully applied all changes from the server, it checks the validity of the proposed changes and commits the valid ones (cf. Listing 3.1, l. 14). A proposed change is a change which has already been carried out in the working directory, but is not synchronized with the server yet. Proposed changes might, for instance, originate from file system watcher notifications (cf. Section 4.5.2). They are valid, if they had actually been executed in the working directory. For instance, a rename change is valid, if the corresponding file was actually renamed. Regardless of whether the valid proposed changes were successfully committed, the client detects and commits its working directory changes to the server's history (cf. Listing 3.1, l. 15). Finally, the client notifies the server

that it finished the synchronization process (cf. Listing 3.1, l. 16), and writes the current version number into the version file (cf. Listing 3.1, l. 17).

```
1  void commitLocalChanges(String owner, Path folder) {
2    SyncCollection syncFiles = collectSyncFiles(owner, folder);
3    Set workFiles = collectWorkFiles(owner, folder);
4
5    for (workFile : workFiles) {
6      syncFile = syncFiles.get(workFile.name);
7
8      if ((syncFile != null) && (workFile.size == syncFile.size)
9            && (workFile.checksum == syncFile.checksum)) {
10       // workFile was not changed
11       syncFiles.remove(syncFile);
12     } else {
13       // was workFile renamed?
14       boolean renamed = false;
15       Set renameCandidates = syncFiles.get(workFile.size);
16
17       for (renameCandidate : renameCandidates) {
18         if (!renameCandidate.equals(syncFile)
19               && (workFile.checksum == renameCandidate.checksum)) {
20           commitRename(renameCandidate.name, workFile.name);
21           syncFiles.remove(renameCandidate);
22           renamed = true;
23           break;
24         }
25       }
26
27       if (!renamed) {
28         // was workFile modified or added?
29         if (syncFile != null) {
30           commitModify(workFile);
31           syncFiles.remove(syncFile);
32         } else {
33           commitAdd(workFile);
34         }
35       }
36     }
37   }
38
39   // remaining files in syncFiles were deleted
40   for (syncFile : syncFiles) {
41     commitDelete(syncFile);
42   }
43 }
```

Listing 3.2: Client change detection algorithm in pseudocode.

Listing 3.2 shows how clients detect and commit its working directory changes in pseudocode. Line 15 of Listing 3.1 calls the corresponding procedure. In order to compare the synchronization directory to the working directory, the client col-

lects information about the files inside the synchronization directory. The information can be represented by a data structure that maintains three hash tables (cf. `SyncCollection` in Listing 3.2, l. 2). One hash table could map file paths to file sizes, while another hash table manages the inverse mapping, i.e. it maps from file sizes to the corresponding set of file paths. A third hash table could map file paths to the corresponding file content checksums. As computing file content checksums is costly, it is advisable to only compute checksums when it is unavoidable. Known checksums should be put into the third hash table for potential, later retrieval. Alternatively, synchronization file information could be kept in a database. In addition to the aforementioned information, the client collects a set of path names of all files inside the working directory.

Changes are detected by comparing each file from the working directory to the files inside the synchronization directory using our synchronization file information. We maintain a set of synchronization file paths in a so called *synchronization collection* (named `syncFiles` in Listing 3.2) that we iteratively update. The synchronization collection may include synchronization file information. For each file in the working directory, we first check whether it was not changed since the last synchronization (cf. Listing 3.2, l. 9). This is done by comparing the working file to the synchronization file with the same name, given that the synchronization file exists. If it is unchanged, i.e. the sizes and checksums of the working and synchronization file match, we will remove the synchronization file from the synchronization collection. In this case, we continue with the next iteration, i.e. we consider the next working file. Otherwise, we check whether the working file was renamed by comparing its checksum to the checksums of synchronization files whose size equals the size of the working file (cf. Listing 3.2, ll. 14-25). If a matching synchronization file is found, we remove it from the synchronization collection and continue with the next iteration. If the working file had not been renamed, the working file was either added or modified. The working file was modified, if a synchronization file with the same name exists (cf. Listing 3.2, l. 30). In this case, we remove it from the synchronization collection and iterate. Otherwise, the working file was added since the last synchronization (cf. Listing 3.2, l. 33). Files which remain in the synchronization collection, after the change status of every working file had been identified, were removed from the working directory (cf. Listing 3.2, l. 41). The client commits each detected change to the server, when it becomes aware of it. The server verifies each commit by checking whether the change is consistent with the existing server history. The client updates the synchronization directory accordingly, if the server accepted the commit.

3.8.2 File Differences

When a file is modified at the client, the changes are transmitted to the server during the next synchronization phase and appended to the history. If only a fraction of the file is modified, transferring just the changes from the client to the server can

decrease the network and server storage usage. We call a data structure which represents the differences between two files a *diff* or *delta*. Applying a diff between file `A` and file `B` to file `A` outputs file `B`.

One approach to integrate diffs in our system is to compute diffs between ciphertexts. However, the user only works with plaintexts and therefore ciphertexts would either need to be stored or computed on demand. By *plaintext* we mean unencrypted data of any kind. Moreover, if the user chose a cipher mode with the property that a plaintext change possibly leads to a larger ciphertext change, we might get a large ciphertext diff despite a small plaintext modification. Computing diffs between two plaintext file versions does not come along with these overheads and therefore our system diffs plaintext files of any kind. As diffs are computed and applied on the client-side only, the server does not bear any computational costs due to diffs. A disadvantage is that random access is only supported for files which are available in its entirety on the server, since the server is unable to apply diffs. In addition, the files must be present in plaintext or encrypted with cipher modes that allow to directly decrypt random blocks in order to provide random data access. Moreover, the server cannot check the validity of the diffs, i.e. it does not know whether client-supplied diffs can be applied to the corresponding source file.

Diffs are in particular useful when large files change slightly. A suitable diff algorithm must be able to produce diffs of relatively small size. Additionally, an appropriate diff algorithm needs to have a low time and space complexity, since the processing power and memory of a client may be limited. As the users may synchronize arbitrary text files, binary files, and other file types, we look for a diff algorithm that is flexible with respect to input types. Since the user does not view the diffs, the diff does not need to be human-readable.

Text diff algorithms such as UNIX diffs are appropriate appropriate for text files. UNIX diff [HM76] is line-based, i.e. it compares lines rather than arbitrary character sequences. It solves "the longest common subsequence" problem in order to determine unchanged lines. The generated diff is printable in a human-readable format. Patches, i.e. files which define how to transform the source file into the target file, can be applied to files which are similar and not necessarily equal to the source file.

Subversion uses an algorithm named *vdelta* [Col00, HpVT96] to compute differences. The vdelta algorithm offers string matching technique which "runs efficiently and requires minimal main memory" [HpVT96]. It runs over the source as well as target data, and produces output for the target data only. During a vdelta run, each sequence is processed from the beginning to the end, thereby building a hash table. The hash table is keyed with four consecutive bytes of the sequence and maps to the starting position of the four bytes. The run time is indirectly proportional to the compressibility of the target data, while the space requirement is proportional to the size of the diff [HpVT96].

The algorithm rsync [TM96, Tri99] updates a file on one machine in such a way that

it becomes identical to the file on another machine. The machines are connected over a low-bandwidth, high-latency, bi-directional communication link. Suppose one machine has file `A` and another machine has file `B`. The algorithm turns file `B` into file `A`. First, `B` is split into non-overlapping fixed-sized blocks. For each block both a weak, rolling checksum and a strong checksum are computed. A rolling checksum maps a byte buffer with n bytes $x_1, x_2, \ldots, x_n$ to a value from the checksum space. It has the property that the checksum for the buffer $x_2, x_3, \ldots, x_{n+1}$ can be computed efficiently using x_1, x_{n+1}, and the checksum of $x_1, x_2, \ldots, x_n$ [TM96]. The checksums from file `B` are transmitted to the machine possessing `A`. File `A` is sought for blocks beginning at any offset whose checksums match checksums from file `B`. The machine which has file `A` sends instructions to the other machine, allowing to construct file `A` from file `B`.

The Xdelta algorithm [Mac00, Tri99] is based on rsync, but requires and takes advantage of the local presence of the files `A` and `B`. Xdelta uses just a rolling checksum and compares the files directly in order to identify match lengths. Its run time as well as its space complexity is linear with the size of the input files.

The tool BSDiff [Per03] produces very small diffs for executable files. It takes advantage of the fact that regions in two versions of binary files which correspond to the same source code region differ at most slightly. Although BSDiff creates significantly smaller diffs than Xdelta for some inputs, it is very memory-hungry [Per06a]. Another algorithm by the same author which is able to produce patches of less size is described in thesis [Per06b]. Google's Courgette [Ada09] also typically generates smaller patches than BSDiff. It disassembles the binary source and target files, adjusts the assembly instructions, and runs BSDiff on the result.

As UNIX diff is not very efficient and is only appropriate for text files, it does not fit into our system. Although BSDiff and Courgette are very efficient, they are intended for executables. Since Xdelta outperforms the already efficient rsync when files are locally available and works for any files, Xdelta is a suitable algorithm for our system. Vdelta is another appropriate diff algorithm candidate. We chose Xdelta as the default diff algorithm in our implementation (cf. Section 4.5.3).

3.8.3 Synchronization Process

Assume that user `U` has a file `F` in its working directory. When the working directory is synchronized with the server, the following events might occur with respect to `F`:

- File `F` is not present on the server.
  - `U` or another group member has removed an identical `F` from the server. Consequently, `U`'s client daemon removes the file on her machine as well.
  - `U` has not yet uploaded `F` or `F` had been moved. `U`'s client daemon uploads or moves the file, respectively.

- File `F` is present on the server.
  - `U`'s version is newer than the server's version. `U` has a plaintext copy of the file in the server's version inside the synchronization directory. `U`'s client daemon computes a diff between the server's version file and `F`, if desired. Furthermore, the daemon encrypts the diff, if desired, and sends it to the server.
  - `U`'s version is older than the server's version. If `U`'s version was not edited, `U` retrieves the diff to the newest version from the server and applies it. If the server's version is a complete file rather than a diff, the client daemon just downloads and saves the server file.

    If `F` is modified, a conflict will occur. Another user or `U` (on a different machine) have edited and uploaded the file. The version of `F` that `U` has in its working directory and the latest server version of `F` are *branched*. In order to resolve the conflict, `U`'s client daemon renames file `F`. The new name of `F` is the old file name plus a suffix containing the current timestamp and the hint that the file is conflicted. The new name also includes a number, if the original new name already existed locally. Then the daemon uploads the renamed file to the server and downloads the server's version. As a result, both `U`'s working directory version of `F` and the server version become available on the server. Consequently, all users that have access to `F` get the two versions of `F` after the next synchronization. Therefore our conflict resolution prevents data loss.

3.9 Concurrency

This section deals with measures to allow concurrent access on the server-side as well as on the client-side. The measures ensure that the client and server states remain consistent. Section 3.9.1 explains how concurrent access is handled for client-server interactions through a network. Section 3.9.2 deals with file locking.

3.9.1 Client-Server Interaction

In order to prevent clients from rendering a server's synchronization history inconsistent, write access to the server's history of a specific folder is provided exclusively for a single client at a time. An inconsistent synchronization history prevents clients from properly synchronizing changes from the server.

Both shared and private folder might be accessed concurrently, since multiple clients from possibly multiple users might attempt to access a shared folder at the same time. As a user may have multiple machines running clients which periodically

synchronize all folders with the server, private folders might be accessed concurrently as well.

Locking on Server-Side

In order to keep the server-history consistent, the server allows only one client to access a specific folder at a time. The server maintains a set of locks to keep track of the currently accessed folders. As folders are identified by their name and owner's name (cf. Section 3.5), a lock is associated with an owner and folder name pair.

Clients can lock a folder only by sending a `GET sync` request. This prevents clients from successfully submitting history modifications without a preceding synchronization request. However, the server cannot prevent the client from ignoring the server's synchronization reply that possibly includes changes. Clients which do not apply server changes locally, might detect and try to commit changes that are not consistent with the server history. However, the server rejects client commits which would render the server history inconsistent (cf. Section 3.10.1).

Each client-server connection is associated with at most one lock on the server-side in order to be able to check whether a lock is held when the server receives a client message over the connection. The servers releases the possibly held lock when the connection is closed or when the server receives the message `POST sync DONE` from the client.

Only one lock per client-server connection can be set at a time. Multiple concurrent locks per connection are not supported, as messages cannot be sent concurrently over the same connection. Therefore clients may synchronize one directory after the other over the same connection, or establish multiple connections to the server. In the latter case, the client is able to synchronize different directories concurrently.

When the server receives a modifying request message such as `PUT file`, `POST move`, or `DELETE file`, the server checks whether the connected client holds the corresponding lock. For the client messages `POST auth`, `PUT auth`, and `PUT folder`, folder locking is not necessary. `POST auth` does not modify any resources on the server. `PUT auth` inserts or updates a user account in the server database. We synchronize `PUT auth` requests with a dedicated lock rather than a folder lock. `PUT folder` creates or updates an entry in the server database. We synchronize `PUT folder` requests with another dedicated lock. The server answers read-only requests such as `GET metadata` or `GET file` without checking whether the client has locked the respective resource.

Locking on Client-Side

On the client-side, the client daemon implementation must not synchronize the same folder concurrently. If a client implementation decided to do so regardless, server locking would prevent concurrent access.

3.9.2 File Access

Concurrent access on a file could result in file data corruption when at least two processes write the file concurrently. If a process reads from a file, while another process writes to it, the read data might be corrupted.

Locking on Server-Side

Files are not locked on the server-side, as the same file cannot be written concurrently by multiple clients, since only one client at a time is allowed to put a file on the server in the same folder (cf. Section 3.9.1). Moreover, the server does not make files available, before they were written to the file system and associated metadata was added to the database. The server neither modifies nor deletes files whose content and metadata it has successfully written (cf. Section 3.5). In addition, the server never changes or removes existing history entries. File locking is not required for concurrent read-only access on the same file from multiple requesters.

Locking on Client-Side

The client daemon locks a working directory file whenever the daemon needs to access it. For example, in order to detect synchronization changes of a file in the working directory, the client daemon locks the working directory file with a shared lock and compares it to files in the synchronization directory. A shared lock prevents other processes from writing, but not reading the file. When a file in the working directory needs to be written, the client daemon locks the file with an exclusive lock in order to prevent other programs from accessing the file. An exclusive lock prevents other processes from both reading and writing the file.

If a lock cannot be acquired, the synchronization process will abort. The client daemon does not lock files in the synchronization directory, since only the single client daemon instance itself may access these files. The user must not run other programs which tamper with synchronization directory files.

A problem can occur when a write lock is required. In order to obtain a write lock, one must have a writable channel of the file. In turn, a writable channel requires that the file exists. Thus, if a file had been renamed or removed, before the write

lock using the original file name was acquired, a new file with the original file name is automatically created. The client daemon synchronizes the new, empty file with the server during the next synchronization, although a user had not created it.

Another problem is that files which are locked, cannot be renamed or deleted, while the lock is held. Thus, if a process `P` holds a file lock, it needs to release the lock, before it is able to delete or rename the file. As `P` releases the lock, before it deletes or renames the file, other processes are able to modify the file in between. `P` does not know that the file was modified, and therefore just deletes or renames the file, without synchronizing the change. Consequently, file changes might be lost.

3.10 Robustness

This section explains the measures taken to ensure robustness against message omissions and system crashes. Section 3.10.1 deals with server robustness, whereas Section 3.10.2 discusses robustness on the client-side.

3.10.1 Server-Side Robustness

Measures

The server commits only changes to the database, after it had successfully written all associated files and verified their integrity. When a change cannot be committed to the database, the server reverts any associated file system changes. These measures ensure that the database state is consistent with the file system state.

The server checks whether client requests are consistent with the present history, before the server commits the corresponding entries to the database. If a client request requires multiple database tables to change, the server will bundle and commit the changes through a database transaction. The server rolls back the transaction, if a statement of the transaction had failed (cf. Section 4.4.3).

The server responds to a client request with a `SUCCESS` message, if and only if the changes had successfully been stored on the file system and in the database beforehand. Consequently, the client can tell whether the server had accepted and successfully executed the change.

Clients and the server exchange messages over a reliable communication channel (cf. Section 3.4.1). Additionally, the client sends content checksums for every file it uploads, while the server compares the sent checksum to the checksum of the received file (cf. Section 3.5). If there is a checksum mismatch, the server rejects the file from the client.

The server stores each file which a client had uploaded as a unique physical file. Therefore the server never overwrites files (cf. Section 3.5).

Crash Handling

A crash on the server-side might lead to an inconsistency between the database and the file system. As changes to the database are only committed after a file had successfully been received and integrity checked, files on the file system without a reference in the database might be present. A recovery tool (cf. Section 4.8.5) detects these orphaned files and deletes them in order to save storage space.

3.10.2 Client-Side Robustness

Measures

The client checks whether requests were carried out successfully on the server. If the server had successfully carried out a client request, the server sends a `SUCCESS` message. The client logs failed requests, i.e. requests which do not result in a `SUCCESS` response. Moreover, the client does not commit changes which are associated with a failed request to the synchronization directory. When an error occurs during synchronization, the client daemon aborts synchronization and sets the version number to the version number of the last successfully executed change. Thus, failed actions can be repeated during the next synchronization.

The client computes a checksum of the possibly encrypted file and transmits the checksum to the server in a `PUT file` request (cf. Section 3.11). This allows the server to check the integrity of the received file. Moreover, every client that receives the file from the server can check data integrity as well. By default, the checksum and other metadata is protected by a MAC (cf. Sections 3.2, 3.11, 4.1.3). If the data the server sends does not match its checksum, synchronizing changes beyond this point will not work without user intervention. A user could manually edit the synchronization version file (cf. Section 3.8.1), in order to skip the synchronization of the corrupted file and continue the synchronization process.

When the client daemon receives a file from the server, the daemon writes it to the working directory at first and copies it to the synchronization directory afterwards. The client daemon copies files that are to be copied to a temporary directory and renames the temporary file to target file afterwards. If the file stores of the temporary and the target file differ, the client daemon will copy the source file to a temporary file in the target directory and then rename the temporary file to the target file. When the client daemon notices that it had unsuccessfully written the temporary file, the daemon removes it. The name of the temporary file allows tools to detect possibly partially written files (cf. Section 4.8).

The client deletes files which had only been partially written to the synchronization directory. It deletes files after all associated files had been written in order to prevent file loss.

It is possible that when server changes are to be applied to the working directory during synchronization, that there is a conflict between a local file and the server change (cf. Section 3.8.3). There are four types of conflicts, depending on the server action to be carried out:

1. The client needs to carry out an add action from the server. If a local file with the same name already exists, a conflict will occur.
2. The client needs to carry out a modify action from the server. If the local file was modified as well, that is the local file differs from the respective synchronization file, a conflict will occur.
3. The client needs to carry out a rename change from the server. If the target file already exists, a conflict will occur.
4. The client needs to carry out a delete change from the server. If the local file differs from the respective synchronization file, it was modified, and a conflict will occur.

The client daemon handles conflicts by renaming the conflicted file locally and uploading it to the server (cf. Section 3.8.3). The name of the renamed file consists of the old file name plus an appended string containing a timestamp. The string also indicates that the file is conflicted and a unique number is added, if multiple conflicts occurred at the same time. Finally, the client executes the conflicted action from the server.

Note that after a client had resolved a conflict, the server history might contain further actions apart from the conflicted file add action which the client has not synchronized yet. As the server appended the conflicted file add change to its history, the conflicted change has the highest version number. The client daemon remembers that it has already executed the conflicted action and does not carry it out again. If an error occurs during synchronization at a version number that is lower than the version number of the conflicted file add change, the client will not request the corresponding conflicted file during the next synchronization, as it is already present in the synchronization directory. However, the client is unable to check whether the synchronization and the server file match, as the server does not have a checksum of the unencrypted file (`PUT file` does not include such a parameter; cf. Section 3.11), unless the file is available in plaintext on the server-side. Consequently, the server file's metadata and the synchronization file's metadata alone are not enough to detect a mismatch. Encrypting the synchronization file and computing the checksum would enable the client to check whether the server and synchronization files match, but causes overhead. Moreover, the latter procedure works only for complete files. Therefore the client daemon assumes that the synchronization and

server files match which is indeed the case, provided that the synchronization file had not been tampered with. Consequently, the client daemon does not download a conflicted file that it had already synchronized. Note that it does not matter what the user has done with the local copy when the client daemon synchronizes changes from the server, since the daemon does not synchronize local changes before all server actions have been executed locally (cf. Section 3.8.1).

Crash Handling

Breaking down the synchronization process, helps to understand the effects of a crash during synchronization. On the client-side, the synchronization process essentially consists of two steps (cf. Section 3.8.1):

1. Synchronization and application of changes from the server on the client.
2. Commitment of local changes from the client to the server.

The client continues with the second step, if and only if the first step had successfully completed. Actions from step one which were not carried out completely, are repeated during the next synchronization. Due to a crash, the client might not have written all changes to the synchronization directory. As the client daemon writes the version file only after all changes had successfully been written to the synchronization directory (cf. Section 3.8.1), the client will repeat incomplete actions during the next synchronization.

Partially written regular files cannot be present on the client under normal circumstances, as the client daemon writes the file content into a temporary file that it renames to the target file, if the temporary file had been written successfully. However, partially written temporary files might exist in the synchronization or working directory after a crash. This could happen, for example, when the client applies a diff from the server, writes the result to a temporary file, and crashes, before the temporary file was completely written. As a result a temporary file is present which is not supposed to exist. Moreover, when the file system does not support atomic renames, both the temporary and the target file might be lost. The tool from Section 4.8.4 detects partially written files according to their temporary file name and deletes them.

Suppose that the file system's move operation is not atomic and that we move a temporary file to a target file. Then the temporary file as well as the target file could both be lost after a crash. Only the latter case is problematic. This might result in a lost synchronization file which is supposed to exist. In turn, this case might lead to the following problems (note that a consistent server history cannot contain a subsequent add event, as otherwise the synchronization file would not have existed in the first place):

- If the client needs to synchronize a modify change from the server and the

corresponding server file is a diff, the client will neither be able to apply the diff nor to continue the synchronization process. If the server file was complete, it is synchronized as usual.

- If the client needs to synchronize a delete change from the server, the daemon will log this event and abort the synchronization process.
- If the client needs to synchronize a rename change from the server, the daemon will log this event and abort the synchronization process.
- If no related action is present in the server history, but a working file with the same name as the missing synchronization file exists, the client will upload the complete working file in step two. The server considers it a modification, as the file exists on the server-side. Then possibly storage space on the server is wasted, as the server will store the whole file rather than a diff.

If a file is present in the synchronization directory, although it is not supposed to exist, the synchronization daemon submits a `DELETE file` request, if a file with the same name is not present in the working directory (cf. Section 3.8.1). However, if a corresponding working file exists as well and differs from the synchronization file, the client will commit a modification to the server. If the client submits the modification as a diff, the server history becomes inconsistent, since no file on which the diff is based is present on the server.

If a temporary file is present in the working directory, the client will consider the file to be new. Therefore the client will upload the temporary file in step two.

If both a temporary working directory file and corresponding working file target get lost when a file is copied to the working directory during synchronization step one, no data loss will occur, as the step can be repeated. Since the client daemon changes working directory files, before it updates the synchronization directory, this scenario has no problematic implications for the synchronization directory.

During synchronization step two, the daemon does not modify files in the working directory. The client commits only detected working directory changes to the server and if the server accepts the request, the daemon will update the synchronization directory accordingly. In this case, the server might carry out a change from the client, while the server message response is lost. If the client does not receive a response message, it will not update its synchronization directory. However, when the client receives a positive response message from the server, it will update its synchronization directory. When the client crashes during the update process, the synchronization directory might be in an inconsistent state afterwards. The recovery script is able to remove any temporary files, but the synchronization directory might need to be turned into a consistent state manually, when synchronization files are missing. The list above outlines the implications of missing synchronization files.

3.11 Protocol

The protocol described in this section allows clients to communicate with the server. As the protocol is fixed, the business logic of the server can be changed, while still supporting existing client software (cf. Section 3.4.1). A goal of our communication protocol is to be open in order to enable developers to build their own client and server applications, sporting possibly new features. Moreover, an open protocol allows to implement a trusted, backdoor-free client and server.

The syntax of the protocol messages is similar to the HTTP/1.1 [Fie99] message format. Therefore implementations can make use of existing HTTP libraries. Furthermore, developers may adapt other HTTP-compatible software such as web browsers, web servers, and proxies to support our protocol.

The server maintains an authentication state per connection. The authentication state indicates whether a connected client is authenticated and if it was, as which user the client was authenticated. Clients use a `POST auth` message to authenticate themselves towards the server. When the server receives any message type other than `POST auth`, it checks whether the requesting client is authenticated and possesses the required access rights, before it processes the message. If the client was not authorized, the server replies with the respective `FAIL` message. Subsequently, the server may decide to abort the connection and ban the client.

`POST` messages are associated with modification and update requests. `PUT` messages create or overwrite a resource on the server. Clients and the server delimit messages with a null byte. The server processes the client request `POST auth` only, if there was not any preceding, successful authentication request. The string `.` denotes the private folder path. For a more precise description of the syntax and encoding, refer to Section 4.6, or the implementation.

In the following, we define the syntax and semantics of the protocol. The client requests are to the left of `|`, whereas the server response is on the right-hand side. We mark parameters with `?`. Parameter values follow the parameter name and the symbol `=`. We mark optional parameters with `[]`. We enclose actions and remarks within curly braces: { }. For example, the client request `GET sync?folder=c2hhcmVk?version=13` is syntactically correct.

The system supports the following protocol messages:

- `GET sync[?owner][?folder][?version] |`
  - `SUCCESS GET sync` { followed by synchronization information, i.e. history tuples `?version?action?object` where the version numbers are greater than the given version. The tuples are sorted in ascending order according to the version. The newline byte as defined in the ASCII table [Vin69] delimits history tuples. The parameter value of `action` is one of `A` (add), `M` (modify), `R` (rename), `D` (delete). If `action` is `A`,

M, or D, the parameter value of object will be the client-supplied file name. If action is R, the parameter value of object will consist of the client-supplied source name followed by an ASCII tab character and the client-supplied target name. If owner is not specified, the authenticated user who sent the request will be taken as the owner. The private folder will be taken as the folder, if folder is unspecified. If version is unspecified, 0 will be used as the parameter value. Note that the combination of an owner and folder name identifies a folder in the server's file hierarchy (cf. Section 3.5). }

  - FAIL GET sync { followed by an ASCII-encoded error description. }

- GET metadata[?owner]?folder?file[?version] |
  - SUCCESS GET metadata { followed by the metadata of the requested file in the form ?version?is_diff?size?hash?key_version[?extra][?mac]. The parameter values are sent as provided in the corresponding PUT file request. If owner is not specified, the authenticated user who sent the request will be taken as the owner. If version is unspecified, the latest version will be used as the parameter value. }
  - FAIL GET metadata { followed by an ASCII-encoded error description. }
- GET file[?owner]?folder?file[?version][?byte_first][?byte_last] |
  - SUCCESS GET file { followed by a null byte and the requested bytes of the file. The file may either be a diff or a complete file. If owner is not specified, the authenticated user who sent the request will be taken as the owner. The newest file will be sent, if version is not specified. If byte_first is unspecified, 1 will be taken as the value. If byte_last is unspecified, the file size in bytes will be taken as the value. Note that the client needs to obtain the size of the file beforehand with GET metadata in order to know when the requested file was completely received. A client should check the MAC that is contained in the file's metadata before it requests the file's data. }
  - FAIL GET file { followed by an ASCII-encoded error description. }
- POST auth?user?password
  { user is the user name and password is the cleartext password } |
  - SUCCESS POST auth {, if the given credentials are valid. }
  - FAIL POST auth { followed by an ASCII-encoded error description. }
- PUT auth?user?password[?current_password]
  { user is the user name and password is the cleartext password to set. current_password is the current password, if a password already existed. } |

  - `SUCCESS PUT auth` {, if the request was valid. The server creates or updates the authentication credentials according to the given parameter values. If authentication credentials exist for the user, the server will have to verify that the provided current password is correct, before it updates the authentication credentials. }
  - `FAIL PUT auth` { followed by an ASCII-encoded error description. }

- `PUT folder?path[?permissions]?key_version`
  { The value of `path` must only consist of at most ten, ASCII-encoded, lower-case alphanumeric characters. Additionally, the `path` value must start with a letter. Permissions have the form `member1:permission/member2:permission` where `member1` as well as `member2` are user names and `permission` is one of `r` (read-only), `rh` (read data and history), `rwh` (read and write data as well as the history). There is the pre-defined member `public` which represents any user. If the request is called on an already existing folder, the access rights are changed according to given parameters. } |
  - `SUCCESS PUT folder` { The server creates or updates the folder according to the given parameter values. The authenticated user is taken as the owner of the folder. }
  - `FAIL PUT folder` { followed by an ASCII-encoded error description. }

- `PUT file[?owner]?folder?file?is_diff?size?hash?key_version [?extra][?mac]`
  { The value of `is_diff` denotes whether the file is a diff or complete and must be either `true` or `false`. The parameter `size` reflects the file size in bytes. The parameter `hash` represents the cryptographic hash of the file as sent. The cryptographic hash algorithm must be pre-defined and supported by the server, in order to enable the server to check the hash of the received file. `mac` is the MAC value which can be computed using the integrity key in version `key_version` as the key, and the concatenation of the parameter values `is_diff, size, hash, key_version, extra` as the input message. However, a developer may choose a different implementation (cf. Section 4.1.3). } |
  - `SUCCESS REQUEST PUT file` {, if the request was valid. When the client receives this message, it directly transmits the file's bytes to the server. The server checks the hash of the received byte array and if it matched the given `hash` value, the server replies with the message `SUCCESS PUT file`. Otherwise, the server sends `FAIL PUT file`. If `owner` is not specified, the authenticated user who sent the request will be taken as the owner. }
  - `FAIL REQUEST PUT file` { followed by an ASCII-encoded error description. }

- `DELETE file[?owner]?folder?file |`
  - `SUCCESS DELETE file` {, if the file existed according to the server history. If `owner` is not specified, the authenticated user who sent the request will be taken as the owner. }
  - `FAIL DELETE file` { followed by an ASCII-encoded error description, if the file to delete does not exist according to the server history. }
- `POST move[?owner]?folder?from?to |`
  - `SUCCESS POST move` {, if file `from` exists and file `to` does not exist according to the server history. If `owner` is not specified, the authenticated user who sent the request will be taken as the owner. }
  - `FAIL POST move` { followed by an ASCII-encoded error description, if the request is not in accordance with the server history. }
- `POST sync DONE |`
  { The server releases any locks held by the client, but does not send a response message. }

Chapter 4

Implementation

This chapter outlines the most important parts of the prototype implementation. The implementation is based on the design from Chapter 3.

Section 4.1 provides an overview of the architecture and outlines how the most important components interact. Section 4.2 describes how the most significant client and server configuration options, thereby pointing out some features of the implementation. Section 4.3 deals with access bundles. An overview of client and server data storage is provided in Section 4.4. Section 4.5 describes how the implementation handles files. The topic of Section 4.6 is the network communication between clients and the server. Finally, this chapter describes the GUI implementation (Section 4.7) and presents our tools (Section 4.8).

4.1 Architecture

The implementation was realized with Java Standard Edition (SE) 7 [Ora12b]. Figure 4.1 illustrates how the most significant parts of the implementation interact. On the server-side, the server application runs in a JVM (Java Virtual Machine). A JDBC driver connects the server application and the database (cf. Section 4.4.3). The server stores files directly on the file system in the server file tree (cf. Section 4.4.2). The server configuration (cf. Section 4.2) file is fed into the server application when the server is started. The server tools (cf. Section 4.8) also read the configuration and interact with the server file tree as well as with the server database.

The server and clients communicate over a network. Each client has a JVM which executes the client daemon and possibly tools (cf. Section 4.8). The client daemon and the tools get a configuration file (cf. Section 4.2) when the user starts them. They read and write files from the working as well as the synchronization directory (cf. Section 4.4.1). The working directory contains keys in form of access bundles

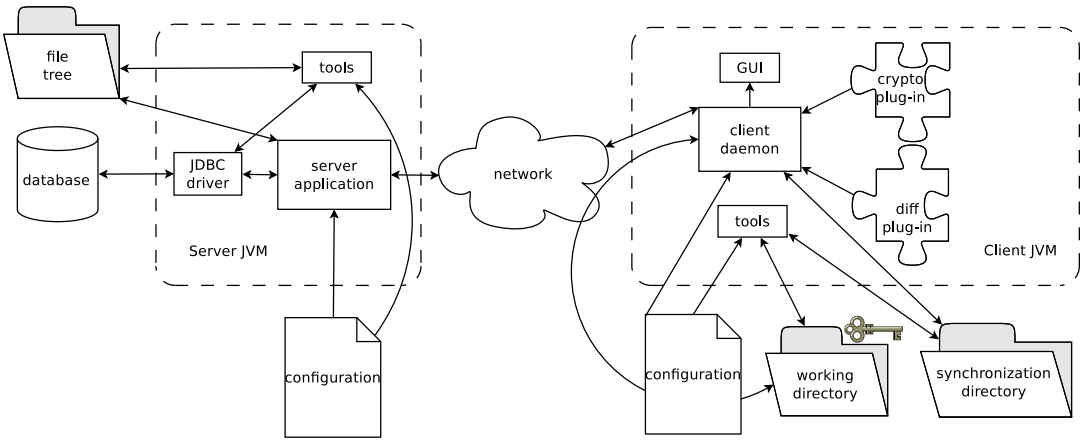

Figure 4.1: Architecture of the implementation.

(cf. Section 4.3). The client daemon notifies a GUI (cf. Section 4.7) when it changes its status. The functionality of the client daemon is extensible with cryptography (cf. Section 4.1.3) and diff (cf. Section 4.5.3) algorithm plug-ins. Our client daemon supports security providers [Ora12a] that are linked to the JVM as well.

A description of all implemented packages is provided in Section 4.1.1. Section 4.1.2 presents an overview of the client-server interaction implementation and explains significant class interactions on an abstract level with the aid of typical client-server interaction scenarios. Section 4.1.3 outlines how the client software supplies the features client-side encryption and data integrity (cf. Section 3.3).

4.1.1 Packages

The program consists of the following packages:

client This package contains the class `Client` which represents the client daemon. The class `ClientMain` starts a client. `ClientConnectionHandler` takes care of the network connection and message sending as well as receiving.

client.executors This package contains various client implementations. Their interfaces adhere to the `ClientExecutor` interface. `ClientExecutorFactory` is a factory for client implementation instances. `PutAuthExecutor` is a tool implementation that enables users to register and update server accounts (cf. Section 4.8). The tool implementation `PutFolderExecutor` allows to create and update folders on the server (cf. Section 4.8). `SynchronizationExecutor` synchronizes the files of a user on a periodic basis and, if desired, commits changes when they are detected (cf. Section 4.5.2). The class `TestExecutor` runs the test cases from Section 5.1.1. The client executors `TestPerformanceCommitter` and `TestPerformanceSyncer` carry out the performance tests from Section 5.2.1.

client.prepare This package reflects the file transmission and reception preparation implementations on the client-side. `PreparationProvider` defines the interface which preparation implementations need to provide. An implementation defines a conversion routine for files that are to be transmitted to the server and also declares how the client transforms files that were received from the server. Therefore an implementation may declare its own client-side encryption routine. `PreparationProviderFactory` allows to create preparation implementation instances. `PreparationProviderDefault` is the default preparation implementation whose functionality is discussed in Chapter 3.

client.tools This package contains tools which aid users of our service. `AccessBundleShell` allows to create and update access bundles. `ClientRecovery` recovers the client after a crash. `PutAuthShell` enables users to register and update user accounts on the server. Users may create and update folders with the tool `PutFolderShell`. Section 4.8 gives a more detailed description of the tools.

configuration The package includes the client configuration `ClientConfiguration` as well as the server configuration `ServerConfiguration` class. Additionally, it contains the abstract access bundle description `AccessBundle` and the concrete owner access bundle `OwnerAccessBundle` and group access bundle `GroupAccessBundle` implementations. Furthermore, the package provides key (class `Key`) and server permission (class `Permission`) representations.

misc This package contains general, re-usable classes for miscellaneous purposes. The class `Coder` provides conversion routines for strings and byte arrays. `FileHandler` implements methods that deal with files. `JSONPrettyPrintWriter` transforms JSON [D. 06] strings to a well-readable format. `Logger` allows to convert log messages and to write them to the desired destinations.

misc.diff This package contains the interface `Differ` which specifies methods to compute and apply diffs. `DifferXdelta` implements these methods using the Xdelta algorithm. `DifferFactory` is a factory class which generates `Differ` implementation instances.

misc.network This package provides classes which manage network connections. The class `ConnectionHandler` offers basic network functionality which both the client and server need. `SecureSocket`, `SecureFinalSocket`, and `SecureSelfHealSocket` provide a secure socket implementation (cf. Section 4.6).

misc.protocol This package contains classes for protocol message generation and parsing. `ClientProtocol` allows to generate client messages from parameter values and to parse server messages. The class `ServerProtocol` is used to

parse client messages and to produce server messages. Data structures representing message values are defined in the class `DataContainers`.

server This package package includes the server daemon class `Server`. `ServerMain` allows to create and launch a server instance. `ServerConnectionHandler` is used to handle a single client-server connection.

server.crypto This package includes only the class `Authentication`. `Authentication` implements password derivation and credential validation (cf. Section 3.4.2).

server.database This package contains database-related classes. `DatabaseCreation` creates the database tables (cf. Section 4.4.3). The class `DatabaseConnection` allows to get a shared as well as dedicated database connections. All necessary database queries and transactions are specified in `DatabaseQueries`.

server.tools This package includes the tool `ServerRecovery` that is able to recover the server after a crash. Section 4.8.5 gives a more detailed description of the tool.

test This package contains test classes. `ClientConfigurationTest`, `CoderTest`, `ConfigAndBundleTest`, `DiffTest`, `FileHandlerTest`, `LockTest`, and `MiscTest` provide unit tests for specific classes as well as functionalities of the program. The class `DatabaseInit` creates the database tables and adds test entries to the tables. `InitClientAndServer` initializes the database and the file hierarchy of the client as well as the server for testing purposes. `RandomFileGenerator` generates files with random content and is used for the test from Section 5.2.2. `TestCommit`, `TestConflict`, and `TestLiveWatcher` are test classes which test synchronization, i.e. the core of the program. An accurate description of these three test classes can be found in Section 5.1. `TestPerformance` runs the performance tests from Section 5.2.1.

view This package provides a GUI implementation through the class `SystemTrayDisplay`. Refer to Section 4.7 for a more precise description.

4.1.2 Client-Server Interaction

This section provides a general overview of client-server interactions from a developer perspective. Moreover, we explain significant class interactions on an abstract level using two typical scenarios. We also visualize important interactions using Unified Modeling Language (UML) sequence diagrams.

General Overview

In the following, we explain how clients connect to the server and authenticate themselves towards the server from a technical perspective.

First, we start the server by invoking the method `start()` on a `Server` instance. Subsequently, we create a `Client` instance and invoke `start()`. This call establishes a TLS connection between the client and the server. An instance of the class `ClientConnectionHandler` represents the connection on the client-side, while the class `ServerConnectionHandler` manages the connection on the server-side. The server spawns a worker thread for each client connection. It maintains a fixed-size thread pool that handles the connections. The size of the pool, i.e. the maximum number of threads which may be active at a time, can be set in the server configuration (cf. Section 4.2). Our server application degrades gracefully owing to the fixed-size thread pool [Ora12f].

The server invokes `next()` on its `ServerConnectionHandler` instance which blocks until a complete message from the client is received (`next()` waits for a message delimiter). The `Client` instance calls `authenticate(String, String)` on the `ClientConnectionHandler` instance in order to authenticate the client towards the server. When the server receives the authentication request from the client, it calls `handleAuthentication(int)` (the parameter denotes the determined message length). The aforementioned method checks whether the credentials the client provided in its message are valid and authorizes the connection, if they were. Subsequently, the server waits for the next client request by calling `next()`. After the client is authenticated, it calls `execute` on an instance of a class which implements the interface `ClientExecutor`.

The `ClientExecutor` instance signals the `Client` instance that it is done, by yielding `false` as the return value of `execute`. When `execute` returns `false`, the client closes the network connection to the server and invokes `join()` which waits for the client thread to finish. When the connection is closed on the server-side, `next()` aborts, since the `ServerConnectionHandler` will not receive messages from the client anymore. The corresponding worker thread finishes and the server releases any folder locks which the thread held.

Metadata and File Download

Figure 4.2 visualizes a scenario in which a client gets file metadata and file content from the server. Refer to the previous section, to get an explanation of the first and last interactions that Figure 4.2 depicts. In this scenario, the `ClientExecutor` invokes `getFile(GetFileData, Path, Path, Path)` on the `ClientConnectionHandler` instance in order to get a file from the server. In turn, the `getFile` method calls `getMetadata(GetMetadataData, Path, Path)` that re-

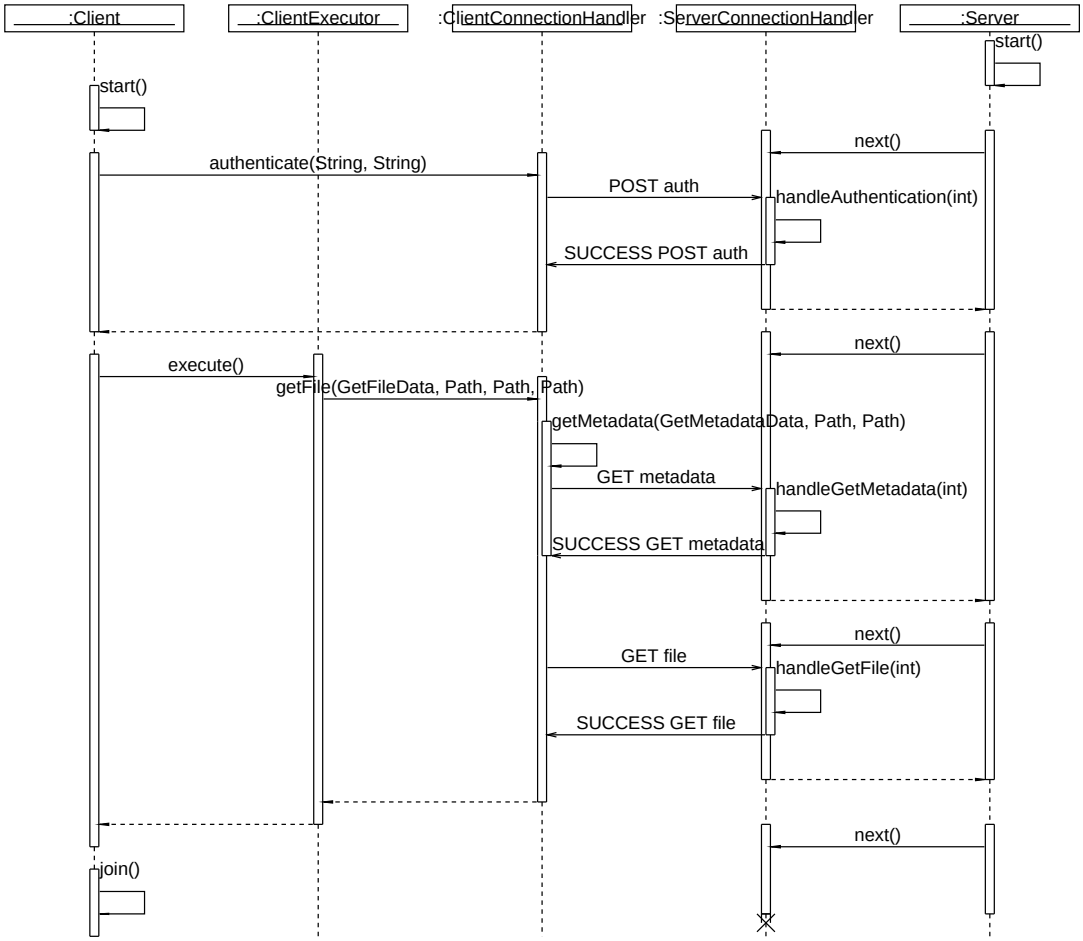

Figure 4.2: Sequence diagram of a file download.

quests metadata from the server. On the server-side, the `next()` method detects the request and calls `handleGetMetadata(int)` on the `ServerConnectionHandler` instance in order to look up and send the corresponding metadata. After the client had received and verified the MAC of the protected metadata (cf. Sections 3.2, 3.11, 4.1.3), the `ClientExecutor` implementation instance sends a `GET file` message to download the actual file content from the server. The server's `next()` method detects the request and calls `handleGetFile(int)` which transmits the requested file to the client. Afterwards the server calls `next()` again, listening for the next client request.

File Synchronization

Figure 4.3 visualizes a synchronization scenario. Refer to the two previous sections, to get an explanation of the first and last interactions which Figure 4.3 illustrates. After the client is authenticated towards the server in Figure 4.3, it calls `execute` on a `ClientExecutor` instance. The `ClientExecutor` instance could be, for ex-

ample, an instance of the class `SynchronizationExecutor`. This instance calls `getSync(GetSyncData, ActionData, Path)` on the `ClientConnectionHandler` instance which requests history changes from the server (cf. Section 3.8.1). The server's `next()` method detects the request and invokes `handleGetSync(int)`. In turn, the method `handleGetSync(int)` searches for the requested history changes in the database and sends them to the client. The client parses the changes from the server's response message. The only change in the response in this example is an add change. The client calls `getSyncAdd(String, Path, Path, ActionData)` to handle the change. The change requires getting a file from the server and therefore the client invokes `getFile(GetFileData, Path)` to get the metadata and file content from the server. On the server-side, `next()` detects the requests and handles them with `handleGetMetadata(int)` as well as `handleGetFile(int)`, respectively. After the client has obtained the file from the server, it synchronizes further changes from the server. As there are not any unprocessed server changes, the client synchronizes its local changes to the server using `getSyncLocal(String, Path, Path)`. The synchronization daemon detects that the user had deleted a file in the working directory and calls `deleteFile(String, Path, Path)` to inform the server about the change. The server handles the change with `handleDeleteFile(int)`. This method appends the delete change to the server history and sends a response to the client. Since the `ClientExecutor` instance does not detect any further changes, it sends a `POST sync DONE` message to the server. The server handles the client message with `handlePostSyncDone()` that releases any locks which are associated with the client-server connection. Then the server calls `next()`, awaiting the next client request.

4.1.3 Client-Side Cryptography

This section describes how the client daemon provides data confidentiality and data integrity.

Figure 4.4 illustrates the cryptography implementation with a UML class diagram. The class `client.prepare.PreparationProviderDefault` reflects the default cryptography implementation. It implements the interface `client.prepare.PreparationProvider` which specifies the encryption method `Triple <Path, ProtectedData, byte[] >prepareSend(Path file, Key encryptionKey, Key integrityKey, boolean isDiff)` and the decryption method `boolean prepareReceive(InputStream in, ProtectedData data, Key decryptionKey, Key integrityKey, Path store)`. The `Key` data structures contain the name of the confidentiality or integrity algorithm to use. The default implementation supports cryptography algorithms for which the Java Virtual Machine offers a provider (cf. web page [Ora12a]). The user specifies the desired algorithm name in the access bundles (cf. Section 4.3).

The method `prepareSend` encrypts the given file using the algorithm and key given

by `encryptionKey`. It works with algorithms which make use of an initialization vector (IV). The method `prepareSend` generates an IV and uses it to encrypt the file. It also produces a MAC using the algorithm and key specified by `integrityKey`. The input message for the MAC algorithm consists of the concatenation of the metadata fields `is_diff`, the file size in bytes, the hash of the ciphertext, the key version, and the IV. The method `prepareSend` returns the path where the ciphertext is stored, a data structure containing the file's metadata, and the MAC.

The method `prepareReceive` reads data from the given input stream, decrypts the data, and stores the plaintext under the given name. Note that `prepareReceive` does not verify the MAC. Client implementations must request the MAC with a `GET metadata` message and verify it, before the implementation requests and decrypts the file content. However, `prepareReceive` checks the hash of the received data. In turn, the message that the MAC protects contains the data hash. The method `prepareReceive` returns whether it had successfully received and decrypted the data.

Developers can implement their own `prepareSend` and `prepareReceive` methods and instantiate their implementation with the factory class `client.prepare.PreparationProviderFactory`. Thus, arbitrary cryptographic algorithms can easily be integrated into the software (cf. Section 5.4).

4.2 Configuration

```
{
  "root_path": "files/clients/joe1",
  "user": "joe1",
  "password": "joe1-secret",
  "server_host": "127.0.0.1",
  "server_port": 54321,
  "server_cert": "files/clients/joe1/.servercert",
  "server_cert_password": "cert-secret",
  "sync_path": "files/clients/joe1/.sync",
  "diff": true,
  "diff_threshold": 0.9,
  "sync_interval": 180,
  "log_file": "files/clients/joe1/.log.txt",
  "log_error_file": "files/clients/joe1/.log.error.txt"
}
```

Listing 4.1: Client configuration example.

Listing 4.1 shows the content of a client configuration example. Client configuration are written in JavaScript Object Notation (JSON) [D. 06]. The configuration parameter `server_cert` reflects the path to the server's public key certificate. Thus, a client can be directly bundled with a server certificate and the user does not

need to rely on a PKI. The server certificate must be stored in a JKS (Java KeyStore) [Ora12c]. The JKS is protected with the password specified by the parameter `server_cert_password`.

The parameter `diff_threshold` determines when diffs rather than complete files are to be uploaded. The threshold size is the value of `diff_threshold` multiplied by the file size of the target. If the size of the diff does not exceed the threshold size, the client will upload the diff rather than the complete file.

The parameter `sync_interval` reflects the periodic synchronization interval in seconds. The remaining parameters of the client configuration should be self-explanatory.

```
{
  "host": "localhost",
  "port": 54321,
  "root_path": "files/server/",
  "database_path": "files/server/.server.db",
  "keystore_path": "files/server/SSLKeyStore",
  "keystore_password": "secret",
  "max_connections": 300,
  "max_connections_per_address": 3,
  "connection_timeout": 10123,
  "max_failed_requests": 3,
  "block_timeout": 30123,
  "log_file": "files/server/.log.txt",
  "log_error_file": "files/server/.log.error.txt"
}
```

Listing 4.2: Server configuration example.

An example for a server configuration is provided in Listing 4.2. Server configurations are, like client configurations, represented in JSON. Clients which send `max_failed_requests` invalid request messages cannot reconnect to the server for `block_timeout` milliseconds. Furthermore, the server closes the connection over which the invalid requests were sent. The other parameters of the server configuration should be self-explanatory.

4.3 Access Bundles

This section describes how access bundles are represented. Access bundles are stored in JSON files under the name *.access*.

```
{
  "owner":"joe1",
  "folder":"folder1",
  "content_keys":[
    {
```

```
6         "version":1,
7         "algorithm":"AES\/CBC\/PKCS5Padding",
8         "key":"LRzGvx0fvwAgZVVr5APz_zjZfIVx-ZP7RpHly1w-78E\r\n"
9       }
10    ],
11    "integrity_keys":[
12      {
13        "version":1,
14        "algorithm":"HmacSHA256",
15        "key":"fO2d6hRyHIzkwBzTss0lx-v1ke5hS7gqUcyVMdF0cUA\r\n"
16      }
17    ]
18  }
```

Listing 4.3: Group access bundle example.

Listing 4.3 shows a group access bundle. The parameter `owner` reflects the name of the owner of the shared folder. The name of the shared folder on the server is given by the value of the parameter `folder`. The parameter `content_keys` refers to an array of arbitrary many content keys, i.e. the symmetric keys which are used for file encryptions and decryptions. Analogously, the parameter `integrity_keys` refers to an array of integrity keys, i.e. the keys used for MAC computations. The parameter `algorithm` reflects the algorithm name. The Java virtual machine must have a security provider [Ora12a] for the algorithm name or `client.prepare.PreparationProviderFactory` must know a cryptography provider plug-in (cf. Section 4.1.3) which supports the algorithm. The `key` value is encoded in URL and filename safe Base64 [Jos03].

A private access bundle looks like a group access bundle, but does not contain the parameters `owner` and `folder`. The owner name of the private folder is determined by the parameter `user` in the client configuration (cf. Section 4.2). The folder name of a private folder is always ".".

Encryption can be turned off for shared folders, by leaving out both the content and integrity key arrays in the group access bundle. However, files in the private folder or in subfolders of the working directory that do not have an access bundle are always encrypted with the latest content key from the private access bundle. The client daemon uses the latest integrity key from the private access bundle to compute the MAC for these files. A private access bundle must be present in the working directory. Otherwise, the client daemon will refrain from synchronizing any private files. Users may only share direct subfolders of the working directory. In order to share a folder, the user must provide a group access bundle for the folder and specify access rights on the server-side (cf. Section 3.5) with a tool (cf. Section 4.8.3). These measures protect the user from uploading private files in plaintext as well as inadvertently sharing files, if the user had forgotten to generate an access bundle.

4.4 Data Storage

This section describes how files are laid out on the client (Section 4.4.1) and the server (Section 4.4.2). Section 4.4.3 describes how metadata and other information is saved in the database.

4.4.1 Client Tree

Suppose that `w` is the name of the working directory and that `s` is the name of the synchronization directory of a client. Section 3.7 outlines the layout of the working directory, whereas Section 3.8.1 describes the layout of the synchronization directory.

```
1 w/.access
2 w/file1.txt
3 w/file2.txt
4 w/subdir1/file1.txt
5 w/subdir1/file2.txt
6 w/shared1/.access
7 w/shared1/.lock
8 w/shared1/file1.txt
9 w/shared1/file2.txt
10 w/shared1/subdir1/file1.txt
```

Listing 4.4: Client working directory example.

A working directory `w` of a client might, for example, look as the one from Listing 4.4. The private access bundle must be located under `w/.access`. Folder `w/subdir1` is private, as it does not contain an access bundle. The folder `w/shared1` is shared, as there is a group access bundle `w/shared1/.access`. Consequently, the client daemon encrypts all files under `w/shared1` with the content key having the highest version number out of all content keys in the group access bundle. An integrity key with the same version number as the used content key must exist. The client daemon uses this integrity key to compute a MAC. An integrity key with a higher version number than any content key must not exist. Files which are not located under `w/shared1` are considered private, as no group access bundle exists for them. The client daemon uses the keys from the private access bundle to provide confidentiality and integrity for these files.

Note that a so called *lock file* is present under `w/shared1/.lock`. As this file exists (existence is enough; content does not matter), the client daemon does not synchronize the folder `w/shared1` with the server. When the user is done editing files in `w/shared1`, she may delete the lock file in order to enable the synchronization daemon to commit her changes to the server. A lock file `w/.lock` would prevent the client daemon from synchronizing any files.

```
1 s/.version
2 s/file1.txt
```

```
3 s/file2.txt
4 s/subdir1/file1.txt
5 s/shared1/.version
6 s/shared1/file1.txt
7 s/shared1/file2.txt
8 s/shared1/subdir1/file1.txt
9 s/shared1/subdir1/file2.txt
```

Listing 4.5: Client synchronization directory example.

Listing 4.5 shows an example of a synchronization directory tree which corresponds to the working directory tree from Listing 4.4. The version file `s/.version` (cf. Section 3.8.1) contains the history version number for the private folder. This version number is used to determine which changes from the server history are already synchronized and which have to be carried out on the client (cf. Section 3.8.1). Analogously, `s/shared1/.version` contains the version number for the shared folder `shared1`. The client daemon compares the files in the working directory to the files in the synchronization directory and commits changes using our algorithms from Section 3.8.1. This is separately done for the private folder and for each shared folder.

Note that all private files from the working directory, except file `subdir1/file2.txt`, also exist in the synchronization directory. Therefore the client daemon might consider `w/subdir1/file2.txt` to be new. In that case, the client daemon would upload the file to the server.

Now consider the shared folder `shared1`. In the synchronization directory, a file named `shared1/subdir1/file2.txt` is present. However, there is no file with the same name in the working directory. Thus, the client daemon might submit a delete change for `shared1/subdir1/file2.txt` to the server. Depending on the content of the files, there might be other changes which the client daemon detects and synchronizes. Moreover, the changes which the client daemon detects might differ from the changes described here. For example, it is possible that the user had actually deleted `shared1/subdir1/file1.txt` and renamed `shared1/subdir1/file2.txt` to `shared1/subdir1/file1.txt`. If that was the case, the client daemon would detect those changes using our synchronization algorithm (cf. Section 3.8.1).

4.4.2 Server Tree

Section 3.5 describes the design of the server's file hierarchy. This section outlines how the server lays out client-provided files with an example.

```
1 f/u1/private/1
2 f/u1/private/2
3 f/u1/private/4
4 f/u1/shared1/1
5 f/u1/shared1/3
```

```
6 f/u2/private/1
7 f/u2/shared1/1
```

Listing 4.6: Server file tree example.

Suppose that the server stores files from clients under the folder `f`. Listing 4.6 shows a possible server tree for this situation. The listing reveals that the server manages data for two users named `u1` and `u2`. Both users have a shared folder named `shared1`. These shared folders are independent of each other. The user `u1` made at least four history changes. The first, second, and fourth change are reflected by the three files `f/u1/private/1`, `f/u1/private/2`, and `f/u1/private/4`. Note that change number three was either a delete or rename change, as no corresponding file exists for the change. By using the history version number as the file name rather than the user-provided file name, it is avoided that the file name length limit of the server file system is exceeded. In addition, the user as well as folder names must consist of pre-defined characters and must not exceed a pre-defined length limit. The server software ensures that a client cannot access files outside the server tree using directory traversal attacks. Furthermore, the server makes sure that the client has permissions to access the requested files.

4.4.3 Server Database

The server keeps file metadata, the history, and further management information in a relational database. Our implementation stores the data in an SQLite [D. 12] database, i.e. in a file. The server software uses Java Database Connectivity (JDBC) technology to access the database in conjunction with an SQLite JDBC Driver [Xer12]. Therefore the server supports various other database management systems (DBMSs) for which JDBC drivers exist as well. Switching to another DBMS requires only minor server software modifications.

The class `server.database.DataBaseConnection` opens and returns database connections. The server requests a new connection per transaction. Thus, the server performance might be improved by using a connection pool.

The database stores data in the following tables. The column names should give an idea of which information the server saves.

users Columns: `id`, `user`, `salted_hash`, `salt`.

folder Columns: `id`, `owner`, `path`, `key_version`, `group_no`.

file Columns: `id`, `owner`, `folder`, `name`, `modified`, `key_version`, `is_diff`, `hash`, `version`, `size`, `extra`, `mac`.
The column `modified` is a server-generated time stamp, reflecting the file's creation time. Column `version` is the history version number that is produced

by the server. The server adopts the respective client-provided parameter values from `PUT file` (cf. Section 3.11).

groups Columns: `id`, `group_no`, `member`, `permission`.

history Columns: `id`, `owner`, `folder`, `version`, `time`, `action`, `object1`, `object2`. The column `time` is a timestamp based on the server's clock, reflecting the time when the corresponding change occurred.

4.5 File Handling

This section highlights some significant aspects about the implementation with respect to files. Section 4.5.1 outlines how the client daemon converts file names. Section 4.5.2 explains how the client daemon detects file changes in a timely manner and synchronizes them with the server. Finally, Section 4.5.3 describes file difference implementations.

4.5.1 File Names

As multiple clients, which probably run on various operating systems, need to support the file names that other clients provide, clients convert file names to a common format before they submit them to the server. Therefore the client daemon transforms file names with the URI syntax (cf. Section 3.5) and submits the resulting name to the server. When a client receives a file name from the server, it converts the URI back to the original representation. The server does not transform the client-provided names in any way.

A problem arises when a file has a name which the file system of another client does not support. For example, UNIX systems use the slash character `/` as the file path name component separator, while under Microsoft Windows systems the backslash character \ serves as the separator. Our client daemon therefore converts backslashes to slashes in every filename sent to the server. Furthermore, file names may be encoded using different character sets and may contain non-ASCII characters. Converting the file names according to the URI syntax solves that problem. Moreover, note that there are case-sensitive file systems such as ext4 [The11] and case-insensitive file systems such as FAT [Mic12]. A client relying on ext4 can store files with names `a` and `A` in the same folder, whereas a client that stores files on a FAT-formatted partition cannot store both files in the same directory. Our implementation does not support the latter case.

4.5.2 File Change Watching

The client daemon class `client.executors.SynchronizationExecutor` allows to synchronize changes in the client's working directory with the server, directly after they had occurred. In order to detect those changes, `client.executors.SynchronizationExecutor` uses Java's `WatchService` API [Ora12g]. While the `WatchService` implementation tries to take advantage of the file system's file change notification service, it polls the file system when a native file change notification service is not available [Ora12g].

The client daemon monitors the client's working directory by creating a "watcher" for it. When the watcher detects a change, the client daemon starts to synchronize the corresponding folder. Thus, the daemon applies any changes from the server at first (cf. Section 3.8.1). Then the client daemon checks whether the reported change had actually been carried out. The daemon commits the change, if the change was present. Afterwards, the client daemon looks for further changes between the synchronization and the working directory and submits these changes as well.

Additionally, the client daemon synchronizes files with the server on a periodic basis (cf. Section 4.2) in order to receive changes which other clients had submitted in a timely manner. File change watching can also be turned off. When this option is not used, the client daemon just synchronizes files on a periodic basis.

4.5.3 File Differences

File differences are computed on the client-side only (cf. Section 3.8.2). The client daemon computes diffs with the Xdelta implementation *javaxdelta* [gen12]. Xdelta stores diffs in the GDIFF (Generic Diff Format) file format [Art97]. VCDIFF [Kor02] is an efficient diff format which is newer than the simple GDIFF format. The differential patching tool *diffable* [Jos12] provides a VCDIFF Java implementation, but works on strings rather than byte arrays. The project *j-vcdiff* [Dav12] aims to implement VCDIFF in Java, but was not ready at the time of writing.

Figure 4.5 illustrates our diff implementation with a UML class diagram. Class `misc.diff.DifferXdelta` represents the default Xdelta diff implementation. It adheres to the interface `misc.diff.Differ` that offers the two methods `Path diff(Path source, Path target)` and `boolean patch(Path source, Path delta, Path output)`. The method `diff` computes the differences between a source and a target file and returns the path where it stores the diff. The method `patch` applies a diff to a source file and stores the result in a file. It returns whether the output file was successfully written.

The factory class `misc.diff.DifferFactory` instantiates `misc.diff.Differ` implementations. Therefore it is possible to add further diff algorithms to the service (cf. Section 5.4).

4.6 Network Communication

The protocol messages (cf. Section 3.11) which the client and server exchange are encoded in ASCII [Vin69]. Any parameter value strings that do not only consist of numbers are converted to the URL and filename safe Base64 alphabet [Jos03]. The Base64 alphabet consists of 64 ASCII characters (hence the name). The advantage of Base64 is that it allows to represent arbitrary bytes with printable characters from ASCII. Thus, the client and server may log messages to ASCII files.

Deserializing messages boils down to parsing an ASCII string representing a URL (cf. Section 3.11) which is easy to do. Although there are more efficient ways to serialize and deserialize data structures [Goo12, The12], we use the approach outlined here for the sake of compatibility and simplicity. Additionally, our protocol messages are printable owing to their ASCII representation which facilitates debugging.

Developers may only access the methods that are associated with the messages `POST auth`, `PUT auth`, `PUT folder`, `GET sync`, `GET metadata`, and `GET file` (cf. Section 3.11). Methods which might modify the synchronization history must only be used after a synchronization request, i.e. within the method that sends `GET sync`. Thus, we made those methods private. Class `client.ClientConnectionHandler` implements the aforementioned methods.

We synchronize user accessible protocol methods (cf. Section 3.11), in order to prevent that the executions of these methods overlap. However, a developer must not synchronize the same folder concurrently. If the developer decides to do so regardless, the server lock management will ensure that only one thread at a time synchronizes the folder (cf. Section 3.9.1).

The client daemon uses so called *self-heal network sockets*, i.e. network sockets which automatically re-connect to the previous destination when the socket connection aborts (cf. class `misc.network.SecureSelfHealSocket`). Thus, if a connection to the server was dropped, before or while the client sends a message, the client will automatically re-establish a connection. After the connection had been established, the client sends a `PUT auth` message (cf. Section 3.11) in order to re-authenticate itself. In order to re-acquire any previously held locks, the client synchronizes the respective directory, if applicable. Finally, the client transmits the message which it originally intended to send. The `misc.network.SecureSelfHealSocket` class executes all aforementioned steps. Note that the server uses ordinary rather than self-heal sockets, since the server must not initiate client-server connections.

The client as well as the server use a message buffer of size 1 MiB (cf. classes `client.ClientConnectionHandler`, `server.ServerConnectionHandler`). The null byte delimits messages and must fit into the message buffer as well. Therefore the maximum message length is $2^{20} - 1$ characters.

4.7 Graphical User Interface

The client software is a daemon (cf. Section 3.4.1), i.e. it runs in the background and does not prompt the user for input. Thus, a graphical user interface which lets the user control the daemon is not provided. However, the daemon notifies the user graphically with system tray icons, if they were supported by the operating system. The GUI is an observer of the client daemon in terms of the observer design pattern.

There are three system tray icons which denote a certain synchronization state. One of them is a green check symbol that the client daemon shows when the synchronization procedure finishes successfully. Another system tray icon showing a red cross system conveys that synchronization failed. An icon with two white-green arrows which point in opposite directions indicates that synchronization is in progress.

4.8 Tools

This section describes tools which help users and administrators to manage their network storage systems. The tool from Section 4.8.1 enables users to register accounts on the server and to update their accounts. Section 4.8.2 presents a tool which is able to create and update access bundles. The tool described in Section 4.8.3 allows to manage shared folders. The tools for client and server recovery are presented in Section 4.8.4 and Section 4.8.5, respectively.

4.8.1 User Registration

The tool `client.tools.PutAuthShell` enables users to register user accounts on the server and to update them through a command-line interface (CLI). The user must specify the desired authentication credentials (cf. Section 3.6.1). If an account is to be updated, the user has to specify the current password of the account as well. The tool creates the corresponding `PUT auth` message (cf. Section 3.11) and sends it to the server. Finally, the tool displays the server's response.

4.8.2 Access Bundle Generator

Class `client.tools.AccessBundleShell` allows the user to generate and update access bundles through a CLI. The tool is able to generate and update private access bundles as well as group access bundles. It produces random confidentiality and integrity keys with the most preferred cryptographically strong random number generator (RNG). Such a RNG has the properties described on web page [Ora12e]. The user specifies the desired key length in bits or accepts the default. Keys are

encoded in URL and filename safe Base64 (cf. Section 4.3). Therefore, the user is able to define keys manually, if she did not trust in the randomness of the generator. Section 3.6.2 discusses automatic key generation.

4.8.3 Shared Folder Generator

Users are able to only access shared folders which are registered on the server-side. The owner of a shared folder is the only group member which has the access rights to register and update shared folders (cf. Section 3.5). The `PUT folder` command (cf. Section 3.11) can create and update shared folders. The tool `client.tools.PutFolderShell` offers a CLI which guides the user through the process of creating and updating shared folders. The user enters the folder's name, the minimum allowed key version, and the access permissions for the folder. The tool creates the corresponding `PUT folder` message and sends it to the server. Finally, the tool shows the server's response. Note that private folders cannot and must not be registered.

4.8.4 Client Recovery

The tool `client.tools.ClientRecovery` removes possibly existing temporary files from the working and synchronization directory of the client. Although the client daemon deletes temporary files when they are not needed anymore, after a client crash temporary files might remain in the working or synchronization directory (cf. Section 3.10.2). As temporary file names have a pre-defined suffix, the tool is able to identify temporary files by their name. Note that a corrupted synchronization directory might require manual recovery after a crash (cf. Section 3.10.2). The tool must only be run when the client daemon is not running, since the client daemon might access the temporary files. It is recommended to backup the synchronization and working directory before running the client recovery tool, in order to be able to recover accidentally deleted files.

4.8.5 Server Recovery

The server references files in the database only after it had checked their integrity. Thus, files might become orphaned after a crash, i.e. the files exist in the server's file directory, but lack a reference in the server database (cf. Section 3.10.1). The tool `server.tools.ServerRecovery` deletes orphaned files found in the server's file directory. It is recommended to backup the server's file directory, before running the server recovery tool, in order to be able to recover accidentally removed files.

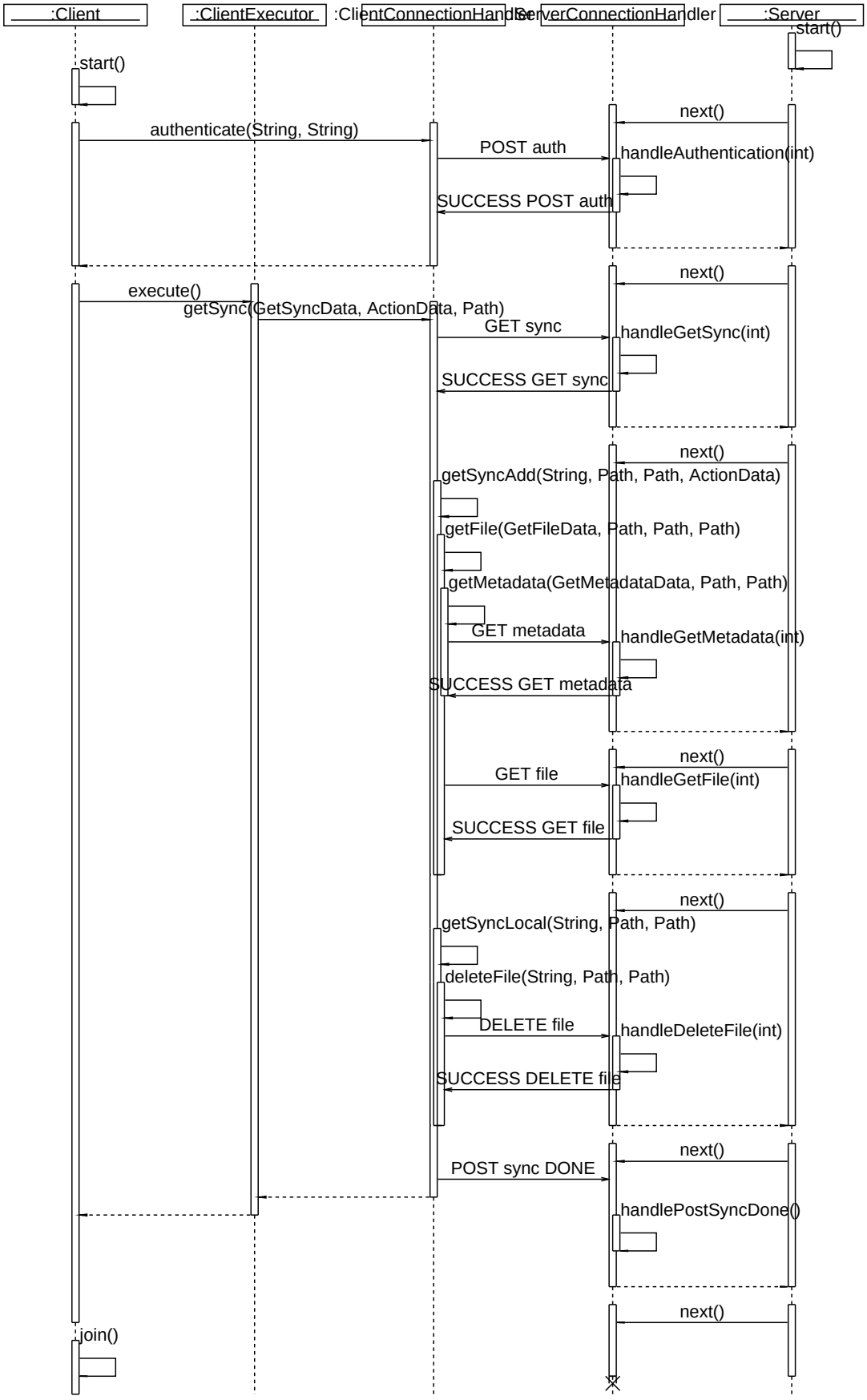

Figure 4.3: Sequence diagram of an example synchronization.

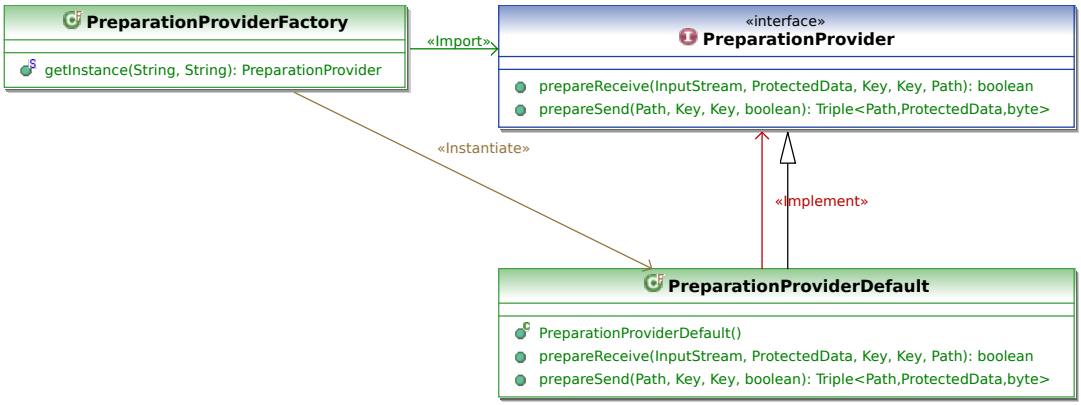

Figure 4.4: Class diagram of the cryptography implementation.

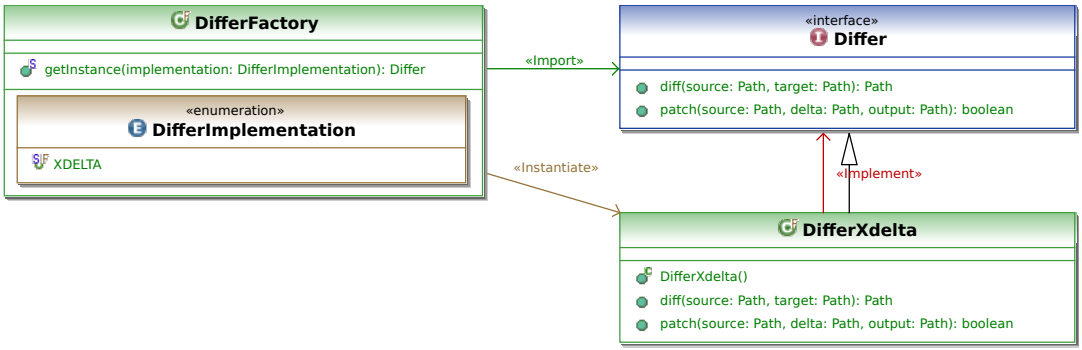

Figure 4.5: Class diagram of the diff implementation.

Chapter 5

Evaluation

In this Chapter we evaluate the implementation presented in Chapter 4. Section 5.1 presents tests which check whether the client and server software work properly. The performance of the network storage system is evaluated in Section 5.2. We analyze the attack resistance of our service in Section 5.3. Section 5.4 discusses potential extensions of the system.

5.1 Functional Tests

All tests in this section were successfully executed on Ubuntu Linux 11.10 as well as 12.04, Microsoft Windows XP, and Mac OS X Lion. We ran all applications in the OpenJDK 7u3 [Ora12d] Java runtime environment.

5.1.1 Synchronization Test

In order to verify that the client daemon software is able to properly synchronize file changes with the server, we designed an appropriate test. Class `test.TestCommit` launches our synchronization test. At first, `test.TestCommit` initializes the test environments of the clients and the server. The environment includes the folder tree and, in case of the server, the server database. Then `test.TestCommit` starts a server and a client which runs the test executor class `client.executors.TestExecutor`. The test executor executes the following test cases in a working directory:

1. Create the group-shared folder `folder1` on the server.
2. Create the publicly shared folder `folder2` on the server.
3. Add file `file1.txt`.

4. Add file `file2.txt`.
5. Modify `file1.txt` in such a way that the complete file rather than a diff to the previous version is synchronized. In order to ensure that the test executor synchronizes the complete file rather than a diff, the test executor modifies `file1.txt` so heavily that the diff threshold is exceeded (cf. Section 4.2).
6. Rename `file1.txt` to `file3.txt`.
7. Delete `file3.txt`.
8. Rename `file2.txt` to `file1.txt`.
9. Add `file2.txt`.
10. Modify `file2.txt` slightly, so that a diff to the previous version rather than the complete file is synchronized.
11. Move `file1.txt` into sub-folder `sub1`.
12. Move `file2.txt` into sub-folder `sub1`.
13. Rename `file2.txt` to `file3.txt`.
14. Modify `file3.txt`.
15. Rename folder `sub1` to `sub2`.
16. Delete `file1.txt`.
17. Delete `file3.txt`.
18. Delete `sub2`.
19. Add `shared1/file1.txt`.
20. Modify `shared1/file1.txt`.
21. Rename `shared1/file1.txt` to `shared1/file2.txt`.
22. Move `shared1/file2.txt` to private directory as `file1.txt`.
23. Add `shared1/file1.txt`.
24. Add `public1/file1.txt`.
25. Add `public1/fileÄ.txt`.
26. Move `public1/fileÄ.txt to fileÖ.txt`.
27. Modify `public1/fileÖ.txt`.
28. Delete `public1/fileÖ.txt`.

After the execution of each test case, the test executor thread synchronizes the changes with the server by calling the synchronization method which the client

daemon uses as well. As the same thread executes the test cases and synchronizes the changes, it is ensured that test case execution and synchronization are coordinated. Therefore every single test case result gets committed to the server. After each commit, the test executor checks whether the resulting version number is correct. After the test executor has committed all changes to the server, it examines whether the working directory files which must exist are actually present. In addition, the test executor verifies that the working directory files which must not exist after the complete test run, do not exist.

After the test executor had synchronized all changes to the server, a client daemon synchronizes all changes from the server to an empty client tree. The client daemon uses the same identity (user name) as the test executor. The purpose of that test is to check whether the client is able to apply the changes from the server and to reconstruct the file tree.

Finally, another client daemon synchronizes the contents of the group-shared and public folder to another, empty file tree. The client daemon authenticates itself as a group member of the `folder1` group. This tests whether group members can successfully synchronize a shared folder. Moreover, the client daemon synchronizes the public folder `folder2`.

The synchronization tests cover all server history change types (cf. Section 3.8) and all protocol messages from Section 3.11 (except `PUT auth` that we tested manually; cf. Section 5.1.4). Furthermore, the test cases involve the private folder, sub-folders, a group-shared, and a publicly shared folder. Diff creation and application (cf. Section 4.5.3) is tested as well as non-ASCII file name handling.

We manually verified that the contents of the synchronization and working directory files are correct. Moreover, we successfully checked the files in the server folder tree and the entries in the server database by hand.

5.1.2 Watcher Test

The test class named `test.TestLiveWatcher` tests whether the watcher functionality (cf. Section 4.5.2) of the client daemon works correctly. This test class launches a server and a client daemon at first. After that we start the test executor thread which runs the test cases from Section 5.1.1. The client daemon thread synchronizes detected changes to the server. When the test executor and watcher threads are done, two different client daemons synchronize the changes to their trees, one after the other, as in Section 5.1.1. In addition, we successfully tested the watcher functionality manually by carrying out changes in the working directory and observing the results on the client as well as the server.

When the test executor is faster than the client daemon, i.e. the client daemon detects a change, after the test executor had already caused further changes, the

client daemon might "summarize" test case results. For example, if the test executor deletes a file named `A` and afterwards creates a file named `A`, the client daemon might commit a modification change for file `A` rather than a delete and add change. Thus, if the test executor is too fast for the client daemon, the client daemon will not commit the result of every single test case. Consequently, we would not have tested the watcher functionality thoroughly.

In order to decrease the likeliness that the client daemon summarizes test cases, we let the test executor pause after each test case. If the sleep time is long enough, the client daemon will execute every single test case. By picking an appropriately low sleep time, the test executor and watcher thread of the client daemon might concurrently access the same file. This tests file locking in a non-deterministic manner. We successfully conducted our watcher functionality tests using diverse sleep times.

5.1.3 Conflict Test

The test cases from Section 5.1.1 do not take conflicts (cf. Section 3.8.3) into account. However, class `test.TestConflict` tests conflict handling (cf. Section 3.10.2). It removes the version file (cf. Section 4.4.1) of the private folder from the synchronization directory and creates files in the working directory which had existed in the server's history at some point. When the private folder is synchronized, the client daemon pulls the complete server history, since no version file is present. As files that are present in the working directory need to be pulled from the server, conflicts occur. If two different clients synchronized conflicted files, the same conflict handling routines that our approach triggers, would be called as well. We successfully checked the results of the test run manually by examining the server database entries as well as the folder trees of the client and the server. In order to build an appropriate server history for the test, we ran `test.TestCommit` prior to `test.TestConflict`.

5.1.4 User Registration Test

The test cases from Section 5.1.1 do not cover user registration tests, as `client.Client` instances rely on existing user accounts. Instead, we register test users with our database initialization script `test.DatabaseInit` prior to running a test. We successfully tested the user registration and update functionalities manually using our `client.tools.PutAuthShell` tool (cf. Section 4.8.1). The corresponding protocol command is `PUT auth` (cf. Section 3.11).

5.1.5 Folder Test

The first two test cases from Section 5.1.1 address the command `PUT folder` (cf. Section 3.11) which is able to create or modify shared folders on the server. However, the aforementioned test cases cover only the folder creation functionality, but not the folder update functionality. Therefore we used the tool `client.tools.PutFolderShell` (cf. Section 4.8.3) to extensively test `PUT folder` by hand.

5.2 Performance Tests

We evaluated the performance of our secure network storage service in a LAN (Section 5.2.1) and over the Internet (Section 5.2.2). We ran the Internet test for Dropbox as well.

5.2.1 Local Area Network Test

The aim of the LAN performance test is to expose the overhead which cryptographic and file difference computations cause. In addition, the test yields real world file synchronization times.

The class `test.TestPerformance` contains the main class that launches the LAN performance test on the client-side. We started the server with `server.ServerMain` and re-launched the server prior to each test run, in order to initialize the server database and file tree.

Class `test.TestPerformance` initializes the client tree and then starts a client that executes `client.executors.TestPerformanceCommitter`. Class `client.executors.TestPerformanceCommitter` can run in either create-only or in create-and-modify mode.

In create-only mode, it creates a file in the private folder of a test user. `client.executors.TestPerformanceCommitter` calls the synchronization method that the synchronization daemon uses. The synchronization method encrypts the private file prior to sending it to the server. The class also generates a file for a public folder and triggers the client daemon's plaintext synchronization procedure.

In create-and-modify mode, `client.executors.TestPerformanceCommitter` creates files as in create-only mode and synchronizes them using the client daemon's synchronization method. It also modifies the files by appending a byte and synchronizes the changes as well. The user may set the file size of the test files.

Class `client.executors.TestPerformanceCommitter` measures *synchronization times*. It uses the system clock's time with millisecond accuracy for the time mea-

surements. The class measures the synchronization times of the private and the public folder separately. It stores the measurement results in a CSV file for later analysis.

In create-only mode, the synchronization time is the execution time of the client daemon's synchronization method. Hence the synchronization time reflects the time it took to synchronize the changes to the server, i.e. the time to encrypt, integrity-protect, and send the changes, as well as to update the synchronization folder. Note that the public file is not encrypted before it is sent to the server. However, the synchronization time does not include the time to create the test files in the working directory.

In create-and-modify mode, the client daemon's synchronization method is called twice: after the file creation and after the file modification. In that case, the synchronization time is the sum of the two synchronization method execution times. The synchronization time does not include the times it took to produce and modify the test files in the working directory.

After `client.executors.TestPerformanceCommitter` is done, `test.TestPerformance` launches another client which synchronizes the private and public folder to a fresh client tree. The client calls the executor `client.executors.TestPerformanceSyncer` which uses the client daemon's synchronization method. The executor measures the synchronization time for the private as well as the public folder and writes those values into a CSV file. In that case, the synchronization time includes the time it took to receive the data, check the integrity of the data, possibly decrypt the data, possibly apply diffs, and to update the working as well as the synchronization directory.

Each created test file was 100 MB in size. The files were modified by appending a null byte. Private files were encrypted with AES-256 in CBC mode, whereas public files were not encrypted. Encrypted, unmodified files had a file size of 100 MB plus 16 B owing to padding. Plaintext diffs were 15 B large, whereas encrypted diffs had a size of 16 B. SHA-256 served as the hash function for data content and HMAC-SHA-256 was used for metadata protection. For public files, data content was hashed, but a HMAC was not computed. Private file content was hashed and private file metadata was protected with a HMAC.

Two Dell PowerEdge R415 machines executed the performance tests. One machine ran the server application, whereas the other one hosted the client. The machines were connected over a switched Gigabit Ethernet network. Each machine had 16 AMD Opteron 4200 series, model 4280 processors (2.8 GHz), 32 GB RAM, two Broadcom Corporation NetXtreme II BCM5716 Gigabit Ethernet NICs, and at least one Western Digital Corporation WD5003ABYX (3.5 in, SATA 3 Gb/s, 500 GB, 7200 RPM) HDD. A Dell PERC H700 controlled four HDDs, forming a RAID 5 set on the server machine, whereas the client machine had only one HDD. The client and the server applications ran inside dedicated virtual machines (VMs) on the different

physical machines. Each VM was equipped with four virtual cores, 4128 MB virtual RAM, and 16 GB virtual disk space. Debian GNU/Linux squeeze with Linux kernel version 2.6.32-5-xen-amd64 acted as both the host and guest operating system. Ext3 was used as the root file system and Xen 4.0.1 [Cit12] served as the hypervisor. The memory-based file system tmpfs [Sny90] managed temporary files. The client and server application were executed with OpenJDK 7u3 [Ora12d]. The load caused by other processes was low on both machines during the test runs.

We measured a round-trip time of 0.187 ms between the client and the server VM using ping. The tool iperf [Gee12a] measured a maximum TCP throughput between the client VM and the server VM of 943 Mb/s. Using hdparm and dd, we measured average disk read and write speeds of 407 MB/s and 379 MB/s, respectively, inside the VMs. Thus, the network connection is the bottleneck in our tests. We conducted the tests over the Gigabit Ethernet without throttling and also ran the test with a 100 Mb/s bandwidth. The tool trickle [Eri05] limited the bandwidth of the client and server applications to 100 Mb/s during the throttling experiment. When we throttled the bandwidth of iperf to 100 Mb/s with trickle, we observed a maximum TCP throughput between the client VM and the server VM of 96.7 Mb/s.

Test	Create-Only	Create-and-Modify	Time Difference
Private Files Commit Time	9.2 / 0.2	11.4 / 0.1	2.2
Public Files Commit Time	6.5 / 0.1	8.6 / 0.1	2.1
Commit Time Difference	2.7	2.8	0.1
Private Files Sync Time	7.2 / 0.1	13.4 / 0.1	6.2
Public Files Sync Time	5.2 / 0.1	11.3 / 0.1	6.1
Sync Time Difference	2.0	2.1	0.1

Table 5.1: Average, measured 1 Gb/s LAN performance test synchronization times / sample standard deviations in seconds. Computed synchronization time differences in seconds.

Test	Create-Only	Create-and-Modify	Time Difference
Private Files Commit Time	15.3 / 0.1	17.5 / 0.1	2.2
Public Files Commit Time	12.7 / 0.1	15.0 / 0.3	2.3
Commit Time Difference	2.6	2.5	0.1
Private Files Sync Time	8.8 / 0.0	15.0 / 0.1	6.2
Public Files Sync Time	8.6 / 0.1	14.8 / 0.2	6.2
Sync Time Difference	0.2	0.2	0.0

Table 5.2: Average, measured 100 Mb/s LAN performance test synchronization times / sample standard deviations in seconds. Computed synchronization time differences in seconds.

The create-only and the create-and-modify tests were executed ten times each. Table 5.1 shows the averaged, measured synchroniza-

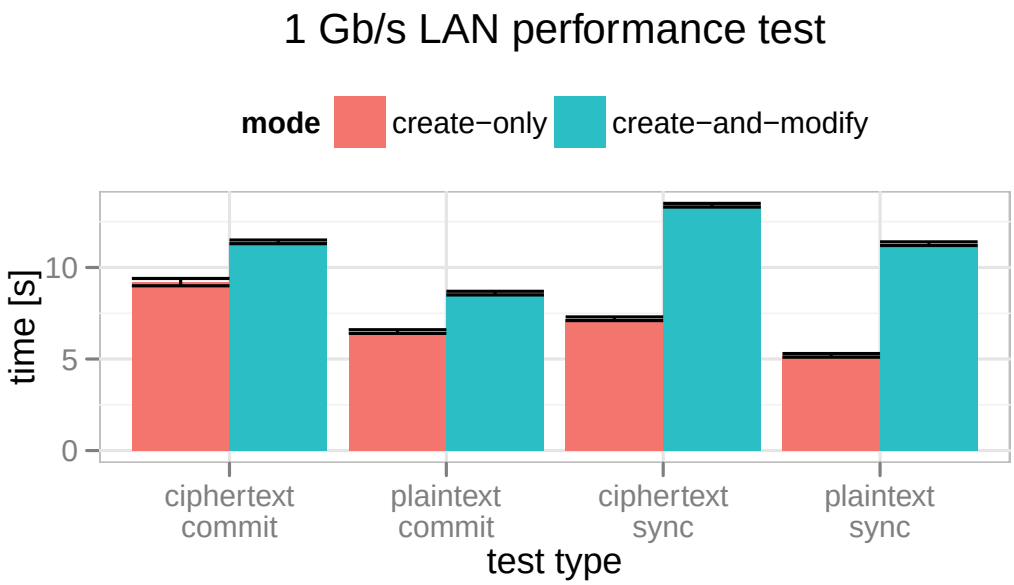

Figure 5.1: The long bars show the average, measured 1 Gb/s LAN performance test synchronization times. The short, black error bars have a length of two standard deviations.

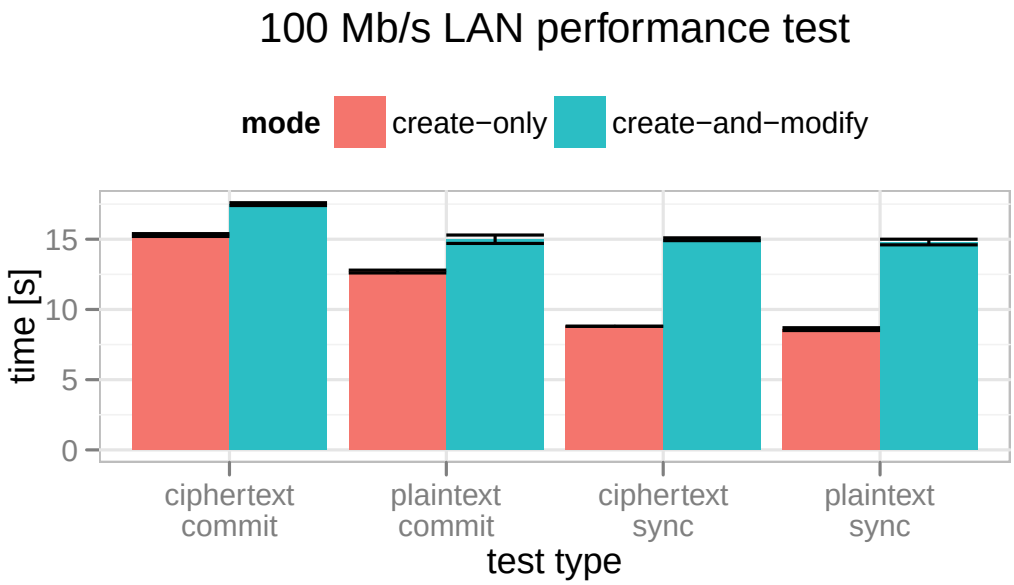

Figure 5.2: The long bars show the average, measured 100 Mb/s LAN performance test synchronization times. The short, black error bars have a length of two standard deviations.

tion times and the manually computed time differences for the 1 Gb/s test. "Commit Time" refers to the synchronization times measured by

`client.executors.TestPerformanceCommitter` and "Sync Time" represents the synchronization times from `client.executors.TestPerformanceSyncer`. Table 5.1 also shows the standard deviations of the measurements. Figure 5.1 visualizes the values from Table 5.1. The results of the 100 Mb/s experiment are shown in Table 5.2 and Figure 5.2.

In the following, we refer to the results of the 1 Gb/s experiment. Creating a diff consumes usually more time and space than applying a diff due to the complexity of the corresponding algorithms (cf. Section 3.8.2). Diff creation and the update of the synchronization directory are the main contributors of the public files commit time difference (2.1 s), as the diff to send is small. The public files sync time difference (6.1 s) mostly depends on the time it takes to apply the diff, copy the output into the working directory, and move the output into the synchronization directory. Since the patch output is saved as a temporary file in the synchronization directory, the patch output can be renamed to the synchronization file name without copying. The communication costs which are caused by requesting and receiving the small diff are negligible. The commit time differences (2.7 s and 2.8 s) mostly reflect the encryption time, while the sync time differences (2.0 s and 2.1 s) are mainly owing to decryption overhead. The public files commit time in create-only mode (6.5 s) is higher than the corresponding public files sync time (5.2 s), as `client.executors.TestPerformanceCommitter` computes and sends a file's hash prior to the file's transmission (the server checks the file's metadata before it accepts the file's content; cf. Section 3.11). Class `client.executors.TestPerformanceSyncer`, however, computes the hash of a file, while it is receiving the file from the server.

5.2.2 Internet Test

We also tested the performance of our network storage server over the Internet. We conducted a performance test for Dropbox as well in order to compare Dropbox's performance to ours.

In order to test the performance of our service, we launched a server inside a VM in Amazon's EC2 (Elastic Compute Cloud) [Ama12c]. The VM was geographically located in Northern California. The client ran on a computer in Passau, Germany. The client tree contained a 10 MB test file in its working directory under a user's private folder. The test file was encrypted with AES-256 CBC prior to its transmission to the server. Data content was hashed with SHA-256 and HMAC-SHA-256 protected metadata. For each test run the client's synchronization directory as well as the server's database and tree were initialized. We measured the time difference between the transmission of the first TCP SYN and the first FIN packet with tcpdump [Lui12] on the client computer. Here we call the absolute value of the time difference *synchronization time*. The client daemon sends the SYN and FIN packets, as it establishes and closes the single TLS connection to the server. When the

connection had been established, the client daemon authenticates itself towards the server and synchronizes detected changes. The detected changes only include the addition of the 10 MB test file. We did not measure the execution time of the client daemon's synchronization method as we did in Section 5.2.1, since we are unable to conduct analogous experiments with the closed source Dropbox daemon.

For the Dropbox performance tests, we created 10 MB files with random content using `test.RandomFileGenerator`. We generated different random files for each test run, since Dropbox only uploads the changes of a file, while we wanted to ensure that Dropbox needs to upload the entire file content instead. As we initialize our own client and server application prior to each test run, our own client daemon must upload a complete 10 MB test file as well. The Dropbox client also ran on our test machine in Passau, Germany. Dropbox keeps the client-supplied data on "Amazon's Simple Storage Service (S3) in multiple data centers located across the United States" [Dro12d]. All IP addresses of the Amazon EC2 servers which Dropbox established connections to during our tests belonged to the server farm in Northern Virginia (according to web page [Ama12a]). We used tcpdump to record TCP SYN as well as FIN packets and noticed that Dropbox opens a SSL connection to a server which is apparently located in San Francisco, when a file is dropped into the Dropbox folder. Subsequently, Dropbox opens a SSL connection to an Amazon EC2 server. When the synchronization finishes as indicated by Dropbox's system tray icon, Dropbox opens two connections to a server in San Francisco. A timeout at the Amazon EC2 and the San Francisco servers seems to trigger the connection termination, as only the server-side sends FIN packets. Moreover, the FIN packets arrive a couple of seconds after the system tray icon had indicated that Dropbox had successfully synchronized the files. During some test runs, multiple connections to different EC2 servers were established. Our measured Dropbox *synchronization time* is the absolute time difference between the first and the second SYN packet which the Dropbox client sends to a San Francisco server. The synchronization time coincides with the synchronization duration as indicated by Dropbox's system tray icon.

The client computer was a Fujitsu ESPRIMO P1500 running Ubuntu 12.04 (x86_64). The machine had the Intel Core2 Quad Processor Q8300 (2.5 GHz), 4 GB RAM, a NVIDIA Corporation MCP73 Gigabit Ethernet NIC, and one Hitachi HDT721010SLA360 (3.5 in, SATA 3 Gb/s, 1 TB, 7200 RPM) HDD. The Linux kernel version was 3.2.0-24-generic and the root as well as the temporary directory were formatted with the ext4 file system. OpenJDK 7u3 served as the Java runtime environment. We used version 1.4.0 of Dropbox. A Gigabit Ethernet switch connected our client machine to a DSL router. The Internet connection had a download bandwidth of 3456 kb/s and an upload bandwidth of 448 kb/s. The DSL connection is a bottleneck of the network path between the client and the server owing to its low bandwidth.

We ran the server VM as a Micro instance [Ama12b] in Amazon's EC2. A Micro

instance grants up to 2 EC2 Compute Units (for short periodic bursts), offers 613 MB RAM, and relies on EBS (Elastic Block Store) for data storage. The VM used one Intel Xeon E5430 (2.66 GHz) CPU. As for the operating system, we ran Ubuntu 12.04 Server (x86_64; AMI ID ami-87712ac2). The Linux kernel had version 3.2.0-24-virtual and Xen 3.4.3-kaos_t1micro acted as the hypervisor. The root and temporary directory were formatted with ext4. OpenJDK 7u3 executed the server application.

Service	Own	Dropbox (Auto Throttling)	Dropbox (No Throttling)
Synchronization Time	227.7 / 2.2	364.7 / 3.0	246.2 / 2.6

Services	Time Difference
Own vs. Dropbox (Auto Throttling)	137
Own vs. Dropbox (No Throttling)	18.5
Dropbox (Auto vs. No Throttling)	118.5

Table 5.3: Average, measured Internet performance test synchronization times / sample standard deviations in seconds. Computed synchronization time differences in seconds.

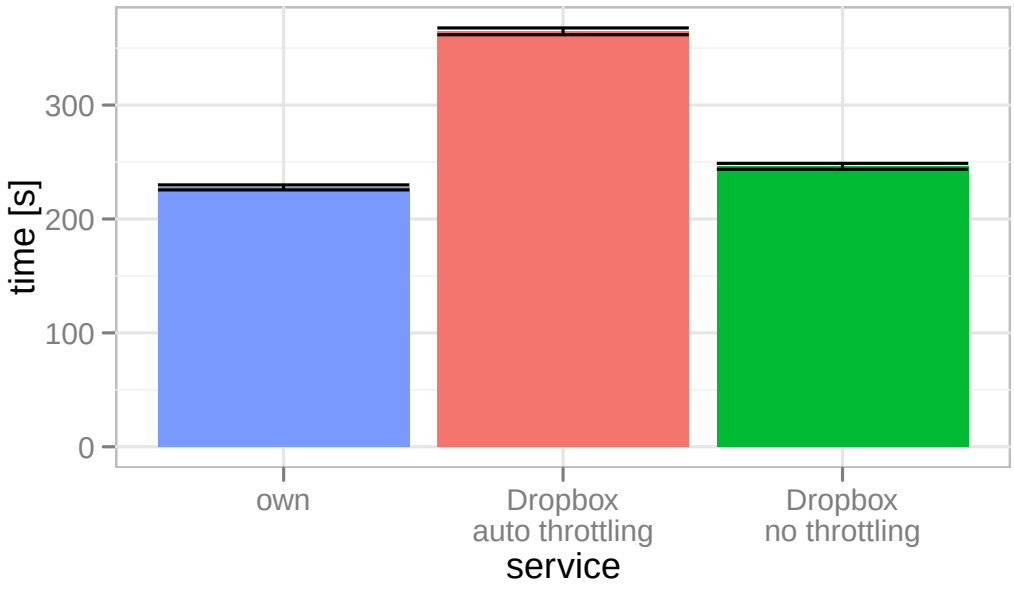

Figure 5.3: The long bars show the average, measured Internet performance test synchronization times. The short, black error bars have a length of two standard deviations.

Table 5.3 shows the average, measured synchronization times of our service and Dropbox. It summarizes the computed synchronization time differences between the different services as well. We ran each test ten times per service and averaged

the lowest five synchronization times, thereby ignoring outliers. Table 5.3 also shows the standard deviations of the lowest five synchronization times. The values from Table 5.3 are visualized in Figure 5.3.

Our service finished a test run in 227.7 s on average. As for Dropbox, we ran the tests with the default settings, i.e. we did not throttle the download speed, but chose to automatically throttle the upload speed. Dropbox then throttles the upload speed to 75 % of the maximum upload speed [Dro12c]. With this setting, it took Dropbox, on average, 364.7 s to synchronize the test file. In addition, we executed the Dropbox tests without throttling, which resulted in an average synchronization time of 246.2 s.

On average, our service finished the tests earlier than Dropbox, although Dropbox's Amazon servers were geographically closer to the client computer than the server our server application ran on. The geographical distance differences are reflected by the round-trip times between the hosts that we measured using ping. The round-trip time between our client computer and our Amazon server in Northern California was 215 ms, while the round-trip time between our client computer and a Dropbox Amazon server in Northern Virginia was 146 ms. Moreover, Dropbox does not have computational overhead owing to client-side encryption prior to transmission, whereas our service does. Dropbox's diff algorithm allegedly identifies file blocks which are already in the cloud and does not upload these blocks anymore [And12]. Determining the changed blocks contributes to the measured synchronization time, if the Dropbox client consulted the server for block information. Furthermore, it is possible that the synchronization time of Dropbox is higher than the synchronization of our service owing to the network route between the client and the server machine.

5.3 Attack Resistance

In this section, we analyze the implications of the attacks from Section 3.2.2. We explain to which extent we are able to foil those attacks. If not specified otherwise, data refers to file content.

Table 5.4 summarizes the security guarantees which our service provides, considering different attacks. Paper [RKS02] contains a similar table that compares the security guarantees of diverse storage systems.

A "change" attack involves carrying out valid modifications, whereas adversaries conduct invalid modifications during a "destroy" attack (cf. Section 3.2.2). Readers are able to detect destroy but not change attacks. In our case, file content, protected metadata, and MAC value changes are invalid modifications, if the adversary did not re-generate and update the keyed MAC value accordingly using the proper key. All other modifications are considered valid, since readers cannot identify them as incorrect.

Attack	leak	change	destroy
Adversary alone	yes	yes	yes
Adversary on server	yes	no	yes
Evicted group member	yes	yes	yes
Evicted group member on server	yes (future data)	no	yes (future data)
Malicious group member	no	yes	yes
Malicious group member on server	no	no	no
Message attacks	yes		
Denial-of-service	no		
Group server subversion	no		

Table 5.4: Overview of security guarantees provided by our service. A "yes" indicates that our service is able to prevent the attack. A "no" means that our service does not prevent the attack.

The following list analyzes our attack resistance in more detail.

- Registered users who do not belong to a group are unable to read and modify data as well as metadata of other users, provided that they do not collude with the server. Unregistered adversaries who are not on the server can neither read nor modify data and metadata of any user.
- An adversary who has access to the server is able to delete data and metadata. Adversaries on the server can also re-write the server's history in a valid way, without readers becoming aware of it. For example, an adversary on the server could change history version numbers and delete history entries. However, readers are able to detect invalid modifications which were carried out by an adversary who colludes with the server, owing to MAC verification (cf. Section 3.2.3). Moreover, such adversaries cannot decrypt data.
- Evicted group members that do not collude with the server can neither access the group's past nor future data as well as metadata, since the owner withdraws access rights from the member on the server during an eviction. However, evicted group members may possess copies of past group data which they had obtained prior to their eviction.
- An evicted group member on the server is able to read past, but not future data. Such attackers are unable to write future data without others becoming aware of the change. However, evicted group members on the server can modify past data and metadata in a valid way, if they colluded with the server. Other group members would be unable to detect the modification. We cannot identify adversarial group members, since they are able to generate valid MACs for their supplied data using the group's key.
- Malicious group members who are writers, are able to submit arbitrary group data to the server. Moreover, such group members are able to generate valid

MACs. Other group members are unable to identify the member who submitted the group data. However, malicious group members can neither modify nor destroy any data and metadata that is already present on the server. Malicious group members can decrypt the group's data.

- Malicious group members who have gained server access, are able to arbitrarily re-write the server history of the group, while the other group members are unable to determine which group member has carried out the changes.
- Eavesdroppers are unable to view transmitted plaintext data including plaintext authentication credentials, as we use over-the-wire encryption that is provided by TLS. Since we employ client as well as server authentication and protect the integrity of over-the-wire data using TLS (cf. Section 3.2.3), our service withstands man-in-the-middle attacks.
- Attackers may attempt to launch DoS attacks in order to render the server unavailable. We do no provide protection against DoS attacks.
- An adversary colluding with the group server can gain access rights to data, since they are able to add, change, and destroy authentication credentials as well as access right lists. Such adversaries are therefore able to withdraw access rights from users. However, adversaries that have access to the group server cannot decrypt data to which they have gained access. Moreover, such attackers are unable to commit history changes without victims becoming aware of the attack.

5.4 Extensions

Section 3.3 explains why our system does not support file name confidentiality and random file access. As for file name confidentiality, the client could randomize rather than encrypt file names. This could be achieved by providing the server with a cryptographic hash of the file name instead of the plaintext file name itself. The drawback is that other clients cannot reverse the hash, i.e. they cannot obtain the original file name, due to the one-way property of the hash function. However, the original file name could reside in a lockbox (cf. Section 3.7) that is encrypted with the content key. This allows all clients which possess the content key to derive the original file name.

The missing features intrusion protection, file name as well as file content searching, and users quotas, which Section 3.3 mentions, could be added to our implementation. The same holds for the features file compression, data redundancy, deduplication, and file links that are also outlined in Section 3.3, since the server or client file system could provide them. In addition, file compression and data redundancy could be implemented on the client-side by a `client.prepare.PreparationProvider` (cf.

Section 4.1.3). For example, the `client.prepare.PreparationProvider` implementation could compress the file, encrypt the plaintext, and encode the ciphertext with an error-correcting code. When the client receives a file from a server, the `client.prepare.PreparationProvider` implementation would reverse the steps, i.e. it would decode, decrypt, and uncompress the file.

A file link points to an actual file and users may access the link as a regular file. Therefore, a file link appears as a file copy to the user, although the file system does not necessarily store the same file content twice. However, our file synchronization algorithm (cf. Section 3.8) does not consider *copy changes*. Supporting copy changes would allow to save storage space and network traffic. Moreover, copy changes could facilitate group sharing, if each file was encrypted with a different key. When a user copies a file from the private or a group folder into another folder, the client would then submit copy change information and a lockbox (cf. Section 3.7) to the server. The lockbox would contain the file key and be encrypted with the latest content key from the access bundle (cf. Section 4.3) of the target folder.

Developers may integrate new cryptographic algorithms into the system owing to the preparation interface (cf. Section 4.1.3). For example, a developer could implement a `client.prepare.PreparationProvider` that encrypts each file with an individual key. The implementation could make use of the lockbox concept (cf. Sections 2.2, 3.7). Section 3.7 outlines how signatures could be used instead of MACs for integrity protection. A developer could realize that idea with a cryptography implementation.

As we provide a diff algorithm interface and a factory (cf. Section 4.5.3), it is easy to integrate further diff implementations. The user could specify the desired diff algorithm per folder by referencing the corresponding implementation in each access bundle (cf. Section 4.3), or set the diff algorithm globally in the configuration (cf. Section 4.2).

Chapter 6

Conclusions

This thesis presented the design of a simple, yet effective, and efficient secure network storage service. Our service enables to store data at untrusted providers without sacrificing confidentiality. Furthermore, the service makes it possible to share data in dynamic groups, while preserving data confidentiality. Our service offers file versioning and incremental data synchronization. Since our service is layered on top of the file system layer, we gain cross-platform compatibility. We compared our design to alternatives, pointing out its assets and drawbacks.

Moreover, it demonstrated that the design is practicable, by presenting a fully-functional implementation. The implementation is convenient to use owing to its autonomous nature. As the user may supply own cryptographic and file difference algorithms, it is extensible as well. It runs on all platforms which Java supports. Tests showed that our software is performant in a LAN as well as over the Internet. Our service outperformed the synchronization service Dropbox in our test scenario.

Future research could devise improvements and extensions for our service. Section 5.4 outlines ideas for possible extensions. Furthermore, future work could conduct tests on a large scale and combine our service with existing cloud technologies.

List of Figures

List of Tables

List of Listings

List of Abbreviations

AES	Advanced Encryption Standard
AMI	Amazon Machine Image
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CBC	Cipher Block Chaining
CHAP	Challenge Handshake Authentication Protocol
CIAA	Confidentiality, Integrity, Availability, Authentication
CLI	Command-Line Interface
CPU	Central Processing Unit
CSV	Comma-Separated Values
CVS	Concurrent Versions System
DBMS	Database Management System
DoS	Denial-of-Service
DSL	Digital Subscriber Line
DSS	Digital Signature Standard
EAP	Extensible Authentication Protocol
EBS	Elastic Block Store
EC2	Elastic Compute Cloud
ext4	fourth extended filesystem
FARSITE	Federated, Available and Reliable Storage, for an Incompletely Trusted Environment
FAT	File Allocation Table
GCKS	Group Controller and Key Server

GDIFF	Generic Diff Format
GNU	GNU's Not Unix
GUI	Graphical User Interface
HDD	Hard Disk Drive
HMAC	Hash-based Message Authentication Code
HTTP	Hypertext Transfer Protocol
IBE	Identity-Based Encryption
IP	Internet Protocol
IV	Initialization Vector
JDBC	Java Database Connectivity
JDK	Java Development Kit
JKS	Java KeyStore
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LAN	Local Area Network
MAC	Message Authentication Code
MSEC	Multicast Security
NIC	Network Interface Controller
NNL	Naor-Naor-Lotspiech
PBKDF2	Password-Based Key Derivation Function 2
PKI	Public Key Infrastructure
PPP	Point-to-Point Protocol
PRF	Pseudorandom Function
RAID	Redundant Array of Independent Disks
RAM	Random Access Memory
RBAC	Role-Based Access Control
RFC	Request For Comments
RNG	Random Number Generator
RPM	Revolutions Per Minute

RSA	Rivest Shamir Adleman
S3	Simple Storage Service
SATA	Serial Advanced Technology Attachment
SE	Standard Edition
SHA	Secure Hash Algorithm
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TPM	Trusted Platform Module
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine
XOR	exclusive or

References

[ABC+02] 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 Oper. Syst. Rev.*, 36:1–14, December 2002.

[ABFF09] Mikhail J. Atallah, Marina Blanton, Nelly Fazio, and Keith B. Frikken. Dynamic and efficient key management for access hierarchies. *ACM Trans. Inf. Syst. Secur.*, 12:18:1–18:43, January 2009.

[Ada09] Stephen Adams. Software updates: Courgette. `http://dev.chromium.org/developers/design-documents/software-updates-courgette`, 2009.

[Ama12a] Amazon EC2 Team. Announcement: Amazon EC2 Public IP Ranges. `https://forums.aws.amazon.com/ann.jspa?annID=1453`, 2012.

[Ama12b] Amazon Web Services LLC. Amazon EC2 Instance Types. `http://aws.amazon.com/ec2/instance-types/`, 2012.

[Ama12c] Amazon Web Services LLC. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`, 2012.

[And12] Andrew Maiman and Bibhas. When does Dropbox delta? `http://webapps.stackexchange.com/questions/26692/when-does-dropbox-delta/26700#26700`, 2012.

[Art97] Arthur van Hoff and Jonathan Payne. Generic Diff Format Specification. `http://www.w3.org/TR/NOTE-gdiff-19970901.html`, 1997.

[B. 00] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. `http://tools.ietf.org/html/rfc2898`, 2000.

[Bau05] Baugher, et al. Multicast Security (MSEC) Group Key Management Architecture. `http://tools.ietf.org/html/rfc4046`, 2005.

[Ber05] Berners-Lee, et al. Uniform Resource Identifier (URI): Generic Syntax. `http://tools.ietf.org/html/rfc3986`, 2005.

[Bob08] Bobbie Johnson. Cloud computing is a trap, warns GNU founder Richard Stallman. http://www.guardian.co.uk/technology/2008/sep/29/cloud.computing.richard.stallman, 2008.

[Cit12] Citrix Systems Inc. What is Xen? http://xen.org/, 2012.

[CKS09] Christian Cachin, Idit Keidar, and Alexander Shraer. Trusting the cloud. *SIGACT News*, 40:81–86, June 2009.

[Col00] CollabNet. Subversion design. http://svn.apache.org/repos/asf/subversion/trunk/notes/subversion-design.html, 2000.

[Com12a] CompletelyPrivateFiles. SecretSync - Client-side encryption for Dropbox. http://getsecretsync.com/ss/, 2012.

[Com12b] CompletelyPrivateFiles. SecretSync - Encrypted synchronization. http://getsecretsync.com/ss/faq/, 2012.

[Coo05] Cooper, et al. Internet X.509 Public Key Infrastructure: Certification Path Building. http://tools.ietf.org/html/rfc4158, 2005.

[D. 06] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). http://tools.ietf.org/html/rfc4627, 2006.

[D. 12] D. Richard Hipp. SQLite Home Page. http://www.sqlite.org, 2012.

[Dam12] Damien Giry. Keylength - Cryptographic Key Length Recommendation. http://www.keylength.com/, 2012.

[Dav12] David Ehrmann. j-vcdiff - Java port of Google's Open VCDIFF (RFC 3284). http://code.google.com/p/j-vcdiff/, 2012.

[Dro11] Dropbox Inc. The Dropbox Blog »Blog Archive »Privacy, Security & Your Dropbox (Updated). http://blog.dropbox.com/?p=735, 2011.

[Dro12a] Dropbox Inc. Dropbox - Simplify your life. https://www.dropbox.com/, 2012.

[Dro12b] Dropbox Inc. Dropbox - Terms - Simplify your life. https://www.dropbox.com/dmca#security, 2012.

[Dro12c] Dropbox Inc. How do I make Dropbox sync faster or control the bandwidth used? https://www.dropbox.com/help/26, 2012.

[Dro12d] Dropbox Inc. Where does Dropbox store everyone's data? https://www.dropbox.com/help/7, 2012.

[Eas05] Eastlake, et. al. Randomness Requirements for Security. http://tools.ietf.org/html/rfc4086, 2005.

[Eri05] Marius A. Eriksen. Trickle: a userland bandwidth shaper for unix-like systems. In *Proceedings of the annual conference on USENIX Annual*

Technical Conference, ATEC '05, pages 43–43, Berkeley, CA, USA, 2005. USENIX Association.

[Fie99] Fielding, et. al. Hypertext Transfer Protocol – HTTP/1.1. `http://tools.ietf.org/html/rfc2616`, 1999.

[FJA02] Jinliang Fan, P. Judge, and M.H. Ammar. Hysor: group key management with collusion-scalability tradeoffs using a hybrid structuring of receivers. In *Computer Communications and Networks, 2002. Proceedings. Eleventh International Conference on*, pages 196 – 201, oct. 2002.

[Fou12] The Apache Software Foundation. Apache subversion. `http://subversion.apache.org/`, 2012.

[Fu99] Kevin Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology, May 1999.

[Fu05] Kevin Fu. *Integrity and access control in untrusted content distribution networks*. PhD thesis, MIT, September 2005.

[Gee12a] Geeknet Inc. Iperf. `http://sourceforge.net/projects/iperf/`, 2012.

[Gee12b] Geeknet Inc. Wuala Webstart. `http://sourceforge.net/projects/wualawebstart/`, 2012.

[gen12] genman, heikok and pivot. javaxdelta. `http://sourceforge.net/projects/javaxdelta/`, 2012.

[G.F11] G.F. Keys to the cloud castle. `http://www.economist.com/blogs/babbage/2011/05/internet_security`, 2011.

[GMSW06] Dominik Grolimund, Luzius Meisser, Stefan Schmid, and Roger Wattenhofer. Cryptree: A folder tree structure for cryptographic file systems. Technical report, Department of Computer Science Purdue University, West Lafayette, IN, 2006.

[Goo12] Google Inc. Protocol Buffers. `https://developers.google.com/protocol-buffers/`, 2012.

[Hal95] Haller, N. The S/KEY One-Time Password System. `http://tools.ietf.org/html/rfc1760`, 1995.

[HM76] J. W. Hunt and M. D. McIlroy. An Algorithm for Differential File Comparison. Technical Report 41, Bell Telephone Laboratories, 1976.

[HMLY05] Ragib Hasan, Suvda Myagmar, Adam J. Lee, and William Yurcik. Toward a threat model for storage systems, 2005.

[HpVT96] James Hunt, Kiem phong Vo, and Walter F. Tichy. An empirical study of delta algorithms, 1996.

[Inc12] Free Software Foundation Inc. Concurrent versions system - summary. http://savannah.nongnu.org/projects/cvs, 2012.

[JA03] Paul Judge and Mostafa Ammar. Security issues and solutions in multicast content distribution: A survey. *IEEE Network*, 17:30–36, 2003.

[Jef11] Jeff Bar. Amazon S3 Server Side Encryption for Data at Rest. http://aws.typepad.com/aws/2011/10/new-amazon-s3-server-side-encryption.html, 2011.

[jGSMB03] Eu jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. Sirius: Securing remote untrusted storage. In *in Proc. Network and Distributed Systems Security (NDSS) Symposium 2003*, pages 131–145, 2003.

[Jos03] Josefsson, Simon. The Base16, Base32, and Base64 Data Encodings. http://tools.ietf.org/html/rfc3548, 2003.

[Jos12] Josh Harrison and James deBoer. diffable - Client-side differential patching of static resources. http://code.google.com/p/diffable/, 2012.

[Kor02] Korn, et. al. The VCDIFF Generic Differencing and Compression Data Format. http://tools.ietf.org/html/rfc3284, 2002.

[Kra97] Krawczyk, et. al. HMAC: Keyed-Hashing for Message Authentication. http://tools.ietf.org/html/rfc2104, 1997.

[KRS+03] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *FAST*. USENIX, 2003.

[LaC12a] LaCie AG. Wuala - Secure Cloud Storage - Backup. Sync. Share. Access Everywhere. http://www.wuala.com/, 2012.

[LaC12b] LaCie AG. Wuala - Secure Cloud Storage - Backup. Sync. Share. Access Everywhere. http://www.wuala.com/en/support/faq/c/1#id000114, 2012.

[Lam81] Leslie Lamport. Password authentication with insecure communication. *Commun. ACM*, 24(11):770–772, November 1981.

[Lui12] Luis MartinGarcia. TCPDUMP/LIBPCAP public repository. http://www.tcpdump.org/, 2012.

[Mac00] Joshua P. MacDonald. File system support for delta compression. Technical report, University of California, Berkeley, 2000.

[Mer80] Ralph C. Merkle. Protocols for public key cryptosystems. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[Mic12] Microsoft. File Systems. http://technet.microsoft.com/en-us/library/cc938915.aspx, 2012.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[Neu05] Neuman, et al. The Kerberos Network Authentication Service (V5). `http://tools.ietf.org/html/rfc4120`, 2005.

[NIS01] NIST. *Advanced Encryption Standard (AES) (FIPS PUB 197)*. National Institute of Standards and Technology, November 2001.

[NIS09] NIST. PUBLIC KEY INFRASTRUCTURES (PKI). `http://csrc.nist.gov/groups/ST/crypto_apps_infra/pki/pkiresearch.html`, 2009.

[Ope12] OpenID Foundation. OpenID Foundation website. `http://openid.net/`, 2012.

[Ora12a] Oracle. Java SE 7 Security Documentation. `http://docs.oracle.com/javase/7/docs/technotes/guides/security/`, 2012.

[Ora12b] Oracle. Java SE Documentation at a Glance. `http://www.oracle.com/technetwork/java/javase/documentation/index.html`, 2012.

[Ora12c] Oracle. keytool - Key and Certificate Management Tool. `http://docs.oracle.com/javase/7/docs/technotes/tools/solaris/keytool.html`, 2012.

[Ora12d] Oracle. OpenJDK. `http://openjdk.java.net/`, 2012.

[Ora12e] Oracle. SecureRandom (Java Platform SE 7). `http://docs.oracle.com/javase/7/docs/api/java/security/SecureRandom.html`, 2012.

[Ora12f] Oracle. Thread Pools. `http://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html`, 2012.

[Ora12g] Oracle. Watching a Directory for Changes. `http://docs.oracle.com/javase/tutorial/essential/io/notification.html`, 2012.

[Per03] Colin Percival. Naïve differences of executable code. `www.daemonology.net/papers/bsdiff.pdf`, 2003.

[Per06a] Colin Percival. Binary diff. `www.daemonology.net/bsdiff/`, 2006.

[Per06b] Colin Percival. *Matching with Mismatches and Assorted Applications*. PhD thesis, Oxford University, 2006.

[Pie12a] Benjamin C. Pierce. Unison faq - general questions. `https://alliance.seas.upenn.edu/~bcpierce/wiki/index.php?n=Main.UnisonFAQGeneral`, 2012.

[Pie12b] Benjamin C. Pierce. Unison file synchronizer. `http://www.cis.upenn.edu/~bcpierce/unison/`, 2012.

[Pos81] Postel, J. (ed.). TRANSMISSION CONTROL PROTOCOL. http://tools.ietf.org/html/rfc793, 1981.

[Raj10] Raju PP. Top 10 Dropbox Alternatives to Securely Sync and Share Files Online. http://techpp.com/2010/07/05/dropbox-alternatives-sync-files-online/, 2010.

[Ree00] George Reese. *Database Programming with JDBC and Java, Second Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2nd edition, 2000.

[RH03] Sandro Rafaeli and David Hutchison. A survey of key management for secure group communication. *ACM Comput. Surv.*, 35(3):309–329, September 2003.

[RKS02] Erik Riedel, Mahesh Kallahalla, and Ram Swaminathan. A framework for evaluating storage system security. In *In FAST '02*, pages 15–30, 2002.

[RSA78] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[Sec11] Secomba GmbH. Advice for sharing BC folder with Dropbox or SugarSync. http://forums.boxcryptor.com/topic/advice-for-sharing-bc-folder-with-dropbox-or-sugarsync#post-19, 2011.

[Sec12a] Secomba GmbH. BoxCryptor — On-the-fly Encryption for cloud storage. http://www.boxcryptor.com/, 2012.

[Sec12b] Secomba GmbH. How does BoxCryptor encrypt files? https://boxcryptorsupport.uservoice.com/knowledgebase/articles/35101-how-does-boxcryptor-encrypt-files-, 2012.

[Sny90] Peter Snyder. tmpfs: A virtual memory file system. In *In Proceedings of the Autumn 1990 European UNIX Users' Group Conference*, pages 241–248, 1990.

[SSN+10] Seok-Won Seong, Jiwon Seo, Matthew Nasielski, Debangsu Sengupta, Sudheendra Hangal, Seng Keat Teh, Ruven Chu, Ben Dodson, and Monica S. Lam. Prpl: a decentralized social networking infrastructure. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, MCS '10, pages 8:1–8:8, New York, NY, USA, 2010. ACM.

[Ste90] W. Richard Stevens. *Unix network programming*. Prentice Hall, 1990.

[T. 08] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. http://tools.ietf.org/html/rfc5246, 2008.

[The11] Theodore Ts'o, et. al. Ext4 Design. https://ext4.wiki.kernel.org/articles/e/x/t/Ext4_Design_87f6.html, 2011.

[The12] The Apache Software Foundation. Apache Thrift. http://thrift.apache.org/, 2012.

[Tho84] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27:761–763, 1984.

[Tho10] Thom Holwerda. More Details Emerge Regarding OpenBSD FBI Backdoors. http://www.osnews.com/story/24142/More_Details_Emerge_Regarding_OpenBSD_FBI_Backdoors/, 2010.

[TM96] Andrew Tridgell and Paul Mackerras. The rsync algorithm. cs.anu.edu.au/techreports/1996/TR-CS-96-05.pdf, 1996.

[Tri99] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University, February 1999.

[Tri12] Andrew Tridgell. How rsync works. http://rsync.samba.org/how-rsync-works.html, 2012.

[Tru12] TrueCrypt Developers Association. Free open-source disk encryption software for Windows 7/Vista/XP, Mac OS X, and Linux. http://www.truecrypt.org/, 2012.

[Vin69] Vint Cerf. ASCII format for Network Interchange. http://tools.ietf.org/html/rfc20, 1969.

[W. 96] W. Simpson. PPP Challenge Handshake Authentication Protocol (CHAP). http://tools.ietf.org/html/rfc1994, 1996.

[Wil97] William A. Nace and James E. Zmuda. PPP EAP DSS Public Key Authentication Protocol. http://tools.ietf.org/id/draft-ietf-pppext-eapdss-01.txt, 1997.

[Xer12] Xerial. sqlite-jdbc - SQLite JDBC Driver. http://code.google.com/p/sqlite-jdbc/, 2012.