TÉCNICO LISBOA

Storekeeper: A Security-Enhanced Cloud Storage Aggregation Service

Sancha Cipriano Pereira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Nuno Miguel Carvalho dos Santos
Prof. Ricardo Jorge Fernandes Chaves

Examination Committee

Chairperson:	Prof. José Manuel da Costa Alves Marques
Supervisor:	Prof. Nuno Miguel Carvalho dos Santos
Member of the Committee:	Prof. Miguel Filipe Leitão Pardal

May 2016

Acknowledgements

I would like to express my special thanks to my supervisors Nuno Santos and Ricardo Chaves for all the patience, guidance and motivation throughout this thesis. I would also like to thank my colleague André Pimenta for helping during the system's evaluation.

I extend my sincere gratitude to my brother and parents for their support during this thesis which would not be possible without them.

Lisboa, May 2016

Sancha Cipriano Pereira

For my parents,

Resumo

Hoje em dia os serviços de armazenamento na nuvem são uma mais-valia que permitem aos utilizadores guardar dados persistentemente, aceder a esses dados em qualquer lado e partilhá-los com os seu amigos ou colegas. No entanto, devido ao acumular de contas em diversos serviços e à inexistência de ligação entre as contas dos diferentes serviços, a gestão e a partilha destes dados na nuvem são um pesadelo para muitos utilizadores. Para minimizar este problema, surgiram sistemas agregadores de diferentes serviços de armazenamento na nuvem de forma a fornecer aos utilizadores uma visão global dos seus ficheiros espalhados pelos diferentes serviços. No entanto, estes sistemas têm limitações de segurança: não só não fornecem privacidade ponto-a-ponto a partir dos serviços na nuvem como obrigam os utilizadores a dar privilégios totais de gestão sobre cada conta de armazenamento na nuvem. Esta tese apresenta o Storekeeper, um serviço agregador de contas de armazenamento na nuvem que preserva a privacidade e permite partilha de ficheiros entre utilizadores com diversas contas de diferentes serviços de armazenamento na nuvem, mantendo a confidencialidade dos dados entre os serviços de armazenamento na nuvem e o serviço agregador. Para fornecer esta propriedade, a lógica da maioria dos serviços agregadores é descentralizada pelo Storekeeper para o lado cliente permitindo que as funções sensíveis de segurança sejam realizadas apenas nos pontos confiáveis do cliente. Esta descentralização acrescenta novos desafios como a propagação de atualizações dos ficheiros, controlo de acessos, autenticação de utilizadores, e gestão de chaves que são tratados pelo Storekeeper.

Abstract

Cloud storage services are currently a commodity that allows users to store data persistently, access the data from everywhere, and share it with friends or co-workers. However, due to the proliferation of cloud storage accounts and lack of interoperability between cloud services, managing and sharing cloud-hosted files is a nightmare for many users. To address this problem, specialized cloud aggregator systems emerged that provide users a global view of all files in their accounts and enable file sharing between users from different clouds. Such systems, however, have limited security: not only they fail to provide end-to-end privacy from cloud providers, but they require users to grant full access privileges to individual cloud storage accounts. This thesis presents Storekeeper, a privacy-preserving cloud aggregation service that enables file sharing on multi-user multi-cloud storage platforms while preserving data confidentiality from cloud providers and cloud aggregator service. To provide this property, Storekeeper decentralizes most of the cloud aggregation logic to the client side enabling security sensitive functions to be performed only on the trusted client endpoints. This decentralization brings new challenges related with file update propagation, access control, user authentication, and key management that are addressed by Storekeeper.

Palavras Chave
Keywords

Palavras Chave

Serviços de Armazenamento na Núvem

Partilha de Ficheiros

Segurança e Privacidade

Agregação de Serviços Núvem

Sistemas de Ficheiros Distribuídos

Keywords

Cloud Storage Services

File Sharing

Security and Privacy

Cloud Aggregation

Distributed File Systems

Contents

List of Figures

List of Tables

Acronyms

ACL Access Control List

API Application Programming Interface

AT Access Token

CSR Cloud Store Record

DNS Domain Name System

FR File Record

FRT File Record Table

JSON JavaScript Object Notation

KF File Encryption Key

KL Login Key

KR Read Key

KU User Key

PKI Public Key Infrastructure

RAID Redundant Array of Independent Disks

REST Representational State Transfer

SD Storekeeper Daemon

SDS Storekeeper Directory Server

SSL Secure Sockets Layer

UD User Descriptor

UDT User Descriptor Table

URL Uniform Resource Locator

WFR Workspace File Record

XML Extensible Markup Language

1 Introduction

Over the past years, cloud storage has become an invaluable resource for consumers. Cloud storage services such as Dropbox[1], Google Drive[2], or Microsoft OneDrive[3] have been widely adopted by many users for storing personal documents, photos, videos, etc., and sharing files with their friends. Oftentimes, given that many companies have limited storage resources, personal cloud storage accounts end up being used also for professional purposes, for example, to store work-related files, collaborate with co-workers, or back up corporate data.

Several factors concur to make cloud storage services so popular among consumers. Firstly, cloud providers put a lot of effort in making such services highly available, allowing users to access their files anytime everywhere (as long as there is network connectivity). Multiple file replicas are stored in the cloud, assuring higher fault tolerance than storing files locally on users' desktops or servers. Furthermore, users are provided with friendly interfaces that allow them to access their files and share them with friends. Through a typical Web interface, users can access all their files from a simple browser. All users have to do is to log into their accounts by providing username and password before they can navigate through their files and open them. To improve user experience, many services allow files to be directly opened within the browser through built-in applications that can interpret rich document formats such as Word, Excel, PDF, and others. Cloud storage services provide remote APIs that allow a client application installed on the users' devices to mount cloud accounts as remote drives accessible through the local file system. Client applications maintain a local cache of the user's files and provide for adequate file synchronization between the local cache and the cloud replicas to resolve possible inconsistencies in the event of write operations. And last but not the least, cloud storage services offer free storage space, which constitutes a great incentive for many users to adopt such services.

[1] https://www.dropbox.com/
[2] https://drive.google.com/
[3] https://onedrive.live.com/

1.1 Motivation

As users' dependency on cloud storage services increases, managing and sharing cloud-hosted files becomes more cumbersome. One reason for this to happen is the *relatively small storage capacity per user account*. In fact, to save costs, many users use only the default amount of free space offered by the cloud providers. Such space is normally in the order of tens or less of gigabytes (e.g., 2GB in Dropbox[4] and 15GB in Google Drive[5]), which is relatively small for the amount of data produced by a single user[6]. To accommodate growing storage demands, users end up signing up for multiple accounts, sometimes with different cloud providers (e.g., Dropbox and Google Drive), others within the same service provide under a different username and/or email. This entails a multiplication of accounts that need to be maintained. To worsen things up, *cloud storage services have incompatible interfaces*. For example, a Dropbox user is not allowed to share a file and edit it jointly with a Google Drive user. Because of this lack of interoperability, sharing files across different services becomes quite involved. To overcome this limitation, users tend to create accounts on a common cloud provider so that they can work on the same set of files. Thus, even the users that can afford to pay for larger storage space are normally forced to maintain multiple accounts in order to collaborate with other users. As a result, user files tend to become scattered across several accounts, making file maintenance and navigation a nightmare.

To overcome these file management issues, a new kind of services have been placed in the market implementing *cloud storage aggregation*. Services such as Cloudfogger [1], Odrive [5], and others [2,3,9,10] run on dedicated servers and implement an intermediate *multi-user multi-cloud* layer between the users and cloud storage providers. Cloud storage aggregators expose to the users a unified view of all files located in their individual accounts and some enable seamless file sharing across cloud accounts, for example between a Google Drive user and a Dropbox user. Users simply endorse such services so that they can access the files located in the users' cloud accounts, and the cloud aggregator provides mediated access between files across cloud provides so as to overcome existing incompatibilities. However, in spite of a considerable usability improvement, with existing cloud aggregator services, users incur additional security risks. In particular, in all these systems users have to grant to the cloud aggregator full access

[4] https://www.dropbox.com/plans
[5] https://support.google.com/drive/answer/6558?hl=en
[6] http://cloudtweaks.com/2015/03/surprising-facts-and-stats-about-the-big-data-industry/

permissions to users' cloud storage accounts so that the cloud aggregator can automatically exchange file updates between different cloud storage backends. Furthermore, with the exception of Cloudfogger, users have no native protection of data confidentiality from the cloud providers, since the files are not encrypted at the client endpoint. As a result, users must give away their privacy to a cloud aggregator in order to enjoy a better integration between cloud storage services.

In spite of all the research that has been done over the past years on cloud security, the problem of providing secure cloud storage aggregation has been overlooked. Systems like BlueSky [28] provide a security layer for file systems, but focus on single-user single-cloud platforms, i.e., the system simply encrypts file system data before storing it back on a single cloud service. SPORC [17] also interposes an encryption layer between the user's client and the cloud provider, and additionally allow for secure file sharing between users. However, since both BlueSky and SPORC focus on single clouds, they lack the mechanisms to enable secure exchange of files between multi-clouds. Multiple clouds have been handled by systems such as DepSky [12] and SCFS [13], but these systems are conceptually different from cloud storage aggregators. DepSky and SCFS combine multiple clouds into a conceptually unique cloud called cloud-of-clouds (CoC). A CoC can then be used by a single user (or by a set of users) to automatically maintain multiple file replicas in different cloud backends in order to increase fault tolerance or file availability. Users are oblivious to the actual location of each file. For example, in CoC if Alice shares a file with Bob, what both Alice and Bob see is a common administrative domain where files are located, without being aware of where files are located. In contrast, in cloud storage aggregators, users do not share a common set of cloud accounts. Instead, each user manages an individual pool of cloud accounts where her files are located, and freely he can exchange such files with other users. Thus, cloud storage aggregation requires the maintenance of independent administration domains that are individually controlled by their respective users, who demand to be aware of where their files are located and retain the access control to their files.

1.2 Contributions

This dissertation presents the design, implementation, and evaluation of Storekeeper, a security-enhanced cloud storage aggregation service. Storekeeper comprises a client application (akin to a Dropbox client) to be installed on the users' computers and a centralized cloud

aggregator server. Users sign up to the server and register their individual cloud storage accounts from Dropbox or Google Drive. Storekeeper provides a unified file namespace, where users can seamlessly browse files located in different accounts and share files with other users.

A key technical contribution of Storekeeper it its novel design, which provides privacy-preserving cloud aggregation by decentralizing the security-sensitive logic of the service. That logic is pushed from the cloud aggregator to trusted clients' endpoints, reducing the cloud aggregator to only a limited set of functions for storing meta-data that is shared between the users. For security reasons, Storekeeper preserves files in their respective owners' accounts and protects the confidentiality of files from both the cloud providers and cloud aggregator server. In particular, the server has no privileges on any of users' accounts.

When compared with a typical centralized cloud aggregator, Storekeeper's decentralized architecture introduces new challenges. First, there is the problem of user authentication and key management. Normally, different clouds manage different user identities. Storekeeper includes mechanisms to handle user identities from different cloud providers, bind encryption keys to their respective users, and provide key management and storage solution that are easy to use by the users. Secondly, it is necessary to devise an adequate security model that masks the heterogeneity of file permission models across cloud providers. Storekeeper defines a simple overarching permission model and implements it using encrypted ACLs. Thirdly, there is the problem of how to enable secure read and write operations to shared files located in different cloud accounts. Storekeeper needs to incorporate specific mechanisms in order to securely propagate file updates without introducing privilege escalation vulnerabilities against potentially malicious users.

We envision Storekeeper's cloud aggregator to be deployed internally by companies in order to enable file sharing between their employees without considerable investment. A company needs only to maintain a Storekeeper server and manage the user base of the system. Storage space will be fully contributed by the users as they sign up and register their cloud accounts.

1.3 Results

In summary, the results of this work are enumerated, as follows:

1. Design of a cloud aggregator platform that enables secure file sharing between users of

heterogeneous storage cloud services;

2. Implementation of a system prototype that demonstrates the effectiveness and viability of our design for supporting file sharing between two popular cloud services: Dropbox and Google Drive;

3. Experimental evaluation of this system using micro-benchmarks.

1.4 Structure of the Document

The rest of this document is organized as follows. Chapter 2 provides an overview of the related work. Chapter 3 describes the architecture of our system and the algorithms implemented by Storekeeper, and Chapter 4 presents the implementation details of our software prototype. Chapter 5 presents the results of the experimental evaluation. Finally, Chapter 6 concludes this document by summarizing its main achievements and future work.

2 Related Work

In this chapter, we present the related work. We start by providing a general historical overview of the security mechanisms that have been progressively introduced into traditional file systems in order to provide data confidentiality (Section 2.1). Then, we focus specifically on the cloud: firstly, we present secure cloud storage systems targeting a single-cloud setting (Section 2.2); and, secondly, we cover the systems that are mostly concerned with the multi-cloud setting (Section 2.3), and thus more consonant with the main goal of this thesis. Lastly, we make a comparative analysis of the state of the art (Section 2.4).

2.1 Securing Traditional File Systems

In traditional computer systems, user data is persistently stored on hard disks and maintained by local file systems such as NTFS, FAT, Ext, etc. Networked file systems such as NFS and AFS make files accessible to remote computers, which is crucial within the context of organizations, for example. Although both kinds of file systems were build with security in mind (e.g., implement file access control mechanisms), the underlying assumption in their design is that the storage medium is trustworthy, and thus the raw file system data (and meta-data) can be stored on disk unencrypted. However, in certain cases, this assumption does not hold any more: an attacker with physical access to the storage medium can extract users' data and compromise its confidentiality. To mitigate such attacks, systems emerged that use encryption to prevent unauthorized interpretation of raw file system data. Since some of such techniques can be used also in the cloud, we make an overview of some representative systems.

2.1.1 Cryptographic File System

One of the first file systems to guarantee file privacy is the *Cryptographic File System* (CFS) [14]. CFS pushes file encryption services into the file system: performs the file encryption

and key management functions and leaves the rest to the underlying file system. CFS runs at user level on the client machine. CFS aims to present the user with a secure file service that works in a seamless manner like a "normal" file system. This file service, where the encrypted files reside, can be any file system (local or remote). With CFS, users can associate a key with a directory (the key is created using a pass-phrase entered by the user), all files and metadata in there are encrypted with that key. The granularity of encryption is at the directory level, making CFS not completely transparent to the user: the user has to remember a pass-phrase for all encrypted directories. CFS uses a combination of ECB (electronic codebook) and OFB (output feedback) cipher modes. ECB is suitable for random data access however, with this mode, a plaintext block is always encrypted into the same ciphertext block (thus it does not hide data patterns). Therefore, ECB is combined with OFB. OFB generates keystream blocks that are XORed with the plaintext blocks to get the ciphertext blocks, each keystream block depends on all previous generated ones. Access control to the encrypted directories is done using the UNIX file protection mechanisms. CFS is not proper to file sharing since it does not assure integrity and does not include any key distribution mechanism. Also, since keys are created from pass-phrases, in case of password compromise, changing a pass phrase for a directory results in re-encrypting all files located in that directory. However, CFS does not support such mechanism.

2.1.2 Transparent Cryptographic File System

The *Transparent Cryptographic File System* (TCFS) [16] is similar to CFS, but, unlike CFS, also makes file encryption transparent to users and allows file sharing between a group (UNIX group) of users. File encryption is transparent to users in a way that the file system knows if a file is ciphered or not, by maintaining one bit information with each file indicating whether the file is ciphered or not. Users only have to maintain one password, a *master-key*, that encrypts all file-keys, rather than a password per directory with CFS. The master-key is encrypted with the login password of the user. Users also have the ability to select which encryption algorithm will be used by TCFS (that later ciphers file blocks in CBC mode - cipher block chaining mode). The granularity of encryption is at the file level, each random file-key is encrypted with the *master-key* and stored in the *file-key* field of the file header stored along with each file. A *block-key* is created per block of a file by hashing the result of the concatenation of the *file-key* and the block number, then the block is encrypted using the *block-key* in the CBC mode. With each

encrypted block is stored an authentication tag, which is the hash of the concatenation of block data and *block-key*, this assures the integrity of blocks. As stated, TCFS allows file sharing within a UNIX group by a threshold. To achieve this a *group-key* is created to encrypt all *file-keys* of the files belonging to that group. To acquire this *group-key*, a threshold number of share holders must be available on the same machine. To be able to decrypt the group-key, at least the threshold number of users must be online at the same time. This threshold information and the group members have to be defined by the file system administrator. The group creation utility generates the random *group-key* and gives a share of the key to each group user, each user's share is encrypted with the user's password. This means that the group creation utility should have the users' password in clear-text, which is a big vulnerability. Also, data is not protected from system administrators. In case of a password change, TCFS decrypts the *master-key* with the old password and re-encrypts it with the new password. However, in case of password compromise it is not enough to change the password and the master-key, the attacker could have collected all file-keys. The solution is to decrypt all files using their old-key and re-encrypt them with fresh keys. In this case, in a group sharing the system administrator has to create a new group-key and redistribute the shares, but it is not clear how TCFS handles this issue. TCFS security is guaranteed by means of the DES algorithm which is easily breakable.

2.1.3 Encrypting File System

The *Encrypting File System* (EFS) [25] has the same functional properties than TCFS, but as opposed to TCFS that works on top of UNIX, EFS works on top of Windows. For each user, EFS creates a public/private key pair and obtains a certificate on the public key from the CA (Certification Authority), configured by the system administrator, to sign the public key (if a CA is not present, EFS self signs it). In EFS, encryption can be at directory or file level using symmetric keys. On file (or directory) encryption, EFS generates a file encryption key (FEK) and encrypts the file (or directory). That FEK is then encrypted with the public key of the user and stored along with the encrypted file on a special EFS attribute called Data Decryption Field (DDF). The operations of decrypt/encrypt of files are transparent to the user, EFS automatically gets the FEK (by decrypting it using the private key of the user), and uses it to decrypt/encrypt the file. When a file (or directory) is shared among multiple users, the FEK is encrypted with the public key of each user (present on the ACL) and the list of encrypted

FEKs is stored in the DDF (encrypting the FEK for each user incurs a lot of overhead on the client side). To access an encrypted file, the EFS client acquires the file's FEK (by decrypting it with the private key of the user) and then decrypts/encrypts the file. Revoking the access of some user to a file is done using a CRL (Certificate Revocation List) and removing the DDF entry for that user. EFS does not re-encrypt the file with a different FEK, meaning that on physical access the revoked user can still decrypt the file. EFS supports file recovery, that is, if a user loses his private key, he can send the encrypted file to a recovery agent. Recovery agents are configured by the system administrator, are assigned a public/private key-pair and the public key is used to cipher the FEK and added to encrypted files in the special attribute called Data Recovery Field (DRF), like the DDF attribute . The recovery agent can then decrypt the file and send it back to the user. Any attacker that gains access to a local administrator account, for example by hacking the Administrator account using third-party tools, can see all protected files transparently decrypted. Also the keys for EFS are protected on disk by the user account password so are therefore susceptible to most password attacks.

2.1.4 Cepheus

Since TCFS and EFS have serious security flaws, *Cepheus* [18] appeared as one of the first secure file sharing schemes. It uses a group server that stores (and delivers) public keys of users and the encrypted *group-key* associated with each UNIX group. The *group-key* is created by the group owner and encrypted for each member of the group using the public key of each member. On file creation, or while changing the ACL of the file, the owner creates the *file-key*, a unique symmetric key using RC5 in CBC mode. This *file-key* is encrypted for the group using the *group-key* and is also encrypted for the owner using the his public key. These two encrypted file keys are placed in the file inode. On file writing, a keyed-hash (HMAC) [24] is generated by the writer on the root of the hash tree of file blocks and metadata to ensure file integrity (using the *file-key*). To reduce the amount of encryption for file writes, Cepheus uses a delayed encryption policy for newly-written blocks. Whenever a user accesses a file, the user first downloads the encrypted *group-key* from the group server, decrypts it using his private key, fetches the encrypted *file-key* from the inode and decrypts it using the *group-key*, verifies the HMAC and if the HMAC can be verified correctly, decrypts the file using the *file-key*. The group server authenticates the client by verifying his signature. On revocation, all files associated with

the group require re-encryption. To reduce the significant delay caused by re-encrypting possibly thousands of files, Cepheus performs delayed re-encryption. To evoke delayed re-encryption, the group owner first marks the cryptographic dirty bit of the file and sets up a new *file-key* in a lock box, then any group member can later re-encrypt the file and clear the dirty bit (the file does not need re-encryption until someone makes a change to the file). After Cepheus appeared two systems (almost at the same time), SiRiUS and Plutus. This systems are very similar to Cepheus but provide a distinction of members with read and write access.

2.1.5 SiRiUS

SiRiUS [21] is a user-level file system. SiRiUS is designed to operate over other file system, in this case insecure network file systems like NFS, or P2P file systems. SiRiUS aims to improve the security of the underlying file system without making any changes to it. SiRiUS ensures confidentiality, integrity of data and some integrity of metadata. All files encrypted by SiRiUS are stored in two parts: a data file, *d-file*, with the encrypted data and a metadata file, *md-file*, with the access control information. Also, in each directory there is a metadata freshness file, *mdf-file* which is the root of the hash tree of all *mdf-files* associated with sub-directories. The root *mdf-file* is signed from time to time by the owner (using his private key) to ensure freshness of the metadata. Both types of metadata files are hidden to the user. Each user has a public/private key-pair and with each file are associated two keys: a symmetric AES File Encryption Key (FEK) and an asymmetric DSA File Signing Key (FSK). A *d-file* is encrypted using the FEK and is signed by the writer using the FSK. The possession of FEK gives read only access to the file, the possession of both FSK and FEK gives read and write privileges to whomever has the keys. The root of hash-tree that is stored in the root *mdf-file* is built by hashing each *mdf-file* and concatenating with the *md-files* of each subdirectory (with SHA-1). The owner can verify it by regenerating the *mdf-file* in some directory and comparing with the current *mdf-file*. All this ensures integrity of metadata, however, SiRiUS does not consider privacy of metadata. To share some file, the owner adds an entry for every user in the *md-file*, each entry contains the FEK (for read access) and the FSK (if the user also has write access) encrypted with the public key of the user. It is assumed the existence of a key distribution mechanism to obtain the authenticated public keys of users. The *md-file* also contains the public key of FSK (to allow readers to verify the signature on the *d-file*), the relative filename (to prevent file-swapping attacks), hash of the

metadata signed by the owner (to ensure integrity of metadata) and the time-stamp of the last modification. On file read and/or write operations, a user gets both *d-file* and *md-file*, verifies the signature on the *d-file* (using the owner's public key of FSK), decrypts FEK and/or FSK and then performs the desired operation. File revocation is similar to file creation: the owner creates new FEK and FSK keys, encrypts the new keys with the public key of the remaining users and signs the *md-file* and *mdf-files*. The re-encryption of the FEK and FSK for all remaining users adds some performance overhead, therefore SiRiUS is good only for small user groups. The authors of SiRiUS have suggested the use of the NNL broadcast encryption algorithm [26] to encrypt the FEK of each file instead of encrypting it with each individual user's public key, thus reducing the performance overhead.

2.1.6 Plutus

Plutus [23] is, similarly to SiRiUS, an abstraction over some file system. A prototype of Plutus was built over OpenAFS, an implementation of AFS. Plutus has minimal trust on the storage server and enables end-to-end confidentiality and integrity of data and metadata (as stated, SiRiUS does not provide confidentiality of metadata). This system also provides secure file sharing. Key management and distribution is handled by the client. In Plutus, files are grouped with other files with identical sharing attributes, forming a file-group (containing permissions for the file-group and the list of members). This is because different classes of files (with different sharing attributes) is small. For each file-group, Plutus associates a unique symmetric key, the *lockbox-key*, and each block of a file is encrypted with a unique 3DES symmetric key, the *file-block key* (automatically created during block creation). The *lockbox-key* is used to encrypt all *file-block keys* belonging to one file-group. To share files with other user, a user creates a file-group (which contains permissions and a list of members) and the corresponding *lockbox-key*, which has to be distributed by the creator of the file-group to the members of the file-group. Associated with each file-group there is an RSA key pair: the private part is the *file-sign key* and the public part is the *file-verify key*. Readers receive only the *lockbox-key* whereas writers receive also the *file-sign keys*. Like in SiRiUS, the possession of signing key distinguishes writers from readers. The *file-sign key* is verified by writers using the corresponding *file-verify key*, proving to the readers that the file is being modified by authorized writers and also ensuring integrity of data and meta-data. Since the *file-sign key* is shared with all writers, it is not possible to verify

the last modifier of a file. Revocation is expensive because it requires re-distribution of keys, re-encryption of the data accessible to the revoked user and re-signing of the revoked data. To overcome these issues, Plutus exploits the concept of lazy revocation (proposed in Cepheus [18], discussed below) and key rotation. Briefly, lazy revocation delays re-encryption until a file is updated. When a user is revoked, the file-group keys are changed by creating a new file-group. So, the non-revoked users should have access to keys of both the revoked file-group and the new file-group. This is done with key rotation: when the new file-group is created, the creator creates a new *lockbox-key* by encrypting the current *lockbox-key* using his private key. Any member of the group can get older versions of the *lockbox-key* by recursively decrypting it with the creator's public key. Both *file-sign keys* and *file-verify keys* are rotated in the same fashion.

2.1.7 Discussion

All previous secure file sharing approaches require the owner of the files/group to securely share the key with other users (done either by encrypting the key with each user's password or each user's public key). While using passwords is insecure, using public-keys has a big computational overhead for large groups (because of the need to encrypt the key with each user's public key). To reduce this overhead there is another approach using proxy-based re-encryption techniques or the NNL broadcast encryption algorithm. Proxy-based re-encryption techniques started with *atomic proxy re-encryption* [15], in which a semi-trusted proxy converts a cipher-text computed under Alice's public key into a cipher-text that can be opened by Bob's secret key, without seeing the underlying plain-text. *Atomic re-encryption* has been prevented by considerable security risks. To offer security improvements over earlier approaches, came *Improved Proxy Re-encryption Schemes with Applications to Secure Distributed Storage* [11], that makes proxy re-encryption a reliable method of adding access control to a file system. The NNL [26], or Naor-Naor-Lotspiech, introduces the subset-sum framework of schemes for broadcast encryption, which is concerned with efficient transmission of a message to a group of receivers whose membership is not fixed.

The discussed traditional file systems are not a solution for the problem related to this thesis. For example, assuming that SiRiUS or Plutus are running on a computer with access to some public cloud storage provider such as Dropbox (via the file system plugin of Dropbox), a user U could not share his files with other users. This is because other users do not have access

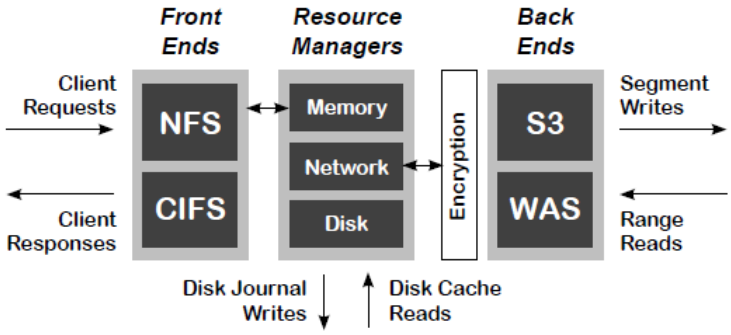

Figure 2.1: BlueSky proxy (by Vrable et al. [28]).

to the cloud provider where U has his files located. Moreover, SiRiUS and Plutus need to be always online, which is not the case of users accessing free public clouds.

2.2 Securing Single-Cloud Storage

Moving now from how files are secured in traditional file systems, in this section we focus on the state-of-the-art of security systems for the single-cloud setting. A common use for single-cloud storage clouds is to replace dedicated in-house storage servers with a cloud storage backend.

2.2.1 BlueSky

BlueSky [28] focuses on replacing existing services, like the traditional client-server application, to the services offered by cloud providers. In particular, how a traditional file service can be replaced with commodity cloud services. BlueSky acts like a local storage server that backs up data to the cloud (Figure 2.1). It is assumed that the end-system software will be unchanged, so BlueSky focuses on a proxy-based solution, where a dedicated proxy server provides the illusion of a single traditional file server (translating requests into appropriate cloud storage API calls).

This system is very focused on the enterprise setting. Therefore, outsourcing of data storage makes security a primary consideration. All client data is individually encrypted (with AES) and protected with a keyed message authentication code (HMAC-SHA-256) by the proxy before sending it over the network, so the cloud provider cannot read private data. Data stored at the provider also includes integrity checks to detect any tampering by the storage provider.

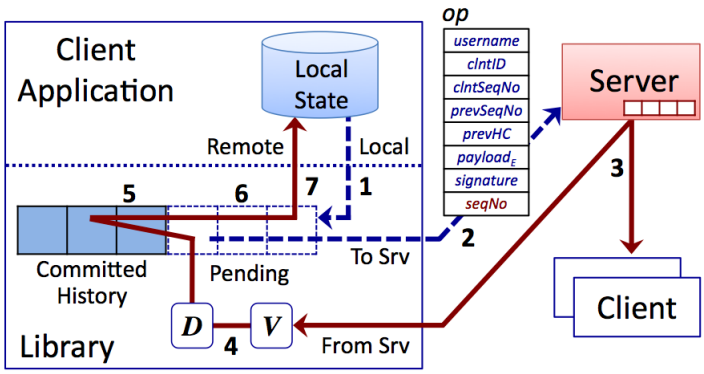

Figure 2.2: SPORC architecture (by Feldman et al. [17]).

BlueSky relies in the cloud provider for data availability. BlueSky prevents a provider from selectively rolling back file data by authenticating pointers between objects starting at the root. The provider can only roll back the entire file system to an earlier state, which users can detect.

The BlueSky file system is composed of 4 types of objects: data blocks, inodes, inode maps and checkpoints. The first two represent data and the latter represent metadata. Data blocks store file data, files are broken apart into 32 KB blocks. Inodes include basic metadata: ownership, access control, timestamps and a list of pointers to data blocks with the file contents. Inode maps list the locations in the log of the most recent version of each inode. Checkpoints determine the root of a file system snapshot: contains pointers to the locations of the current inode map objects. Data objects are encrypted while metadata objects are not. The keys for encryption and authentication are not shared with the cloud and the authors assume that users keep a safe backup of these keys for disaster recovery. The proxy uses its local disk storage to implement a write-back cache. It ensures that data is safely on disk before telling clients that data is committed and then writes are sent to the cloud provider asynchronously. The proxy also keeps a cache on disk to satisfy many read requests without going to the cloud. The proxy is fully trusted. File sharing is not discussed in the paper, but since the proxy operates over a distributed file system it surely uses the sharing scheme of the file system.

2.2.2 SPORC

SPORC [17] approaches the problem of fully trusting cloud providers, with potentially sensitive and important data, when cloud services are used to collaboratively edit some shared document by multiple users. SPORC is a flexible framework that allows a group of users who

trust each other to collaboratively edit some document with the help of a central untrusted server whose sole purpose is to order and store client-generated operations.

SPORC has the following goals: propagate modifications of the shared state quickly, keep data confidential from the central server and unauthorized users, tolerate slow or disconnected networks, detect a malicious server and recover from it. Figure 2.2 shows the architecture of this system, that is composed of a central untrusted server and clients. Clients exchange two types of operations: *document operations*, which represent changes to the content of the document, and *metadata operations*, which represent changes to document metadata such as the document's ACL. Every operation is labeled with the name of the user that created it and is digitally signed by that user's private key (every user has a public/private key pair and it is assumed that clients have a secure way to verify the public keys of other users). The central untrusted server sole propose is to order and store client-generated operations, maintaining all documents as a set of ordered operations. The server stores the operations in its encrypted history so that new clients joining a document or existing clients that have been disconnected can request the operations they are missing (allowing quickly propagation of modifications, only for existing clients).

To ensure that the server is well behaved, clients use sequence numbers and a hash chain to ensure that operations are properly serialized. Every operation has two sequence numbers: a client sequence number *clntSeqNo* (assigned by the client that submits the operation), a global sequence number *seqNo* (assigned by the server) and a global sequence number of the last committed operation *prevSeqNo* along with the corresponding hash chain value *prevHC*. On receiving an operation, a client verifies that the operation's *clntSeqNo* is one greater than the last *clntSeqNo* seen by the submitting client and that the operation's *seqNo* is one greater than the last *seqNo* seen by the client. On uploading a new operation to the server, the client sets the *prevSeqNo* and the *prevHC* fields. A client who receives a new operation compares its *prevHC* with his own hash chain over the committed history of the same document. If they match, fork* consistency is guaranteed. A misbehaving server cannot modify the *prevSeqNo* and*prevHC* fields because they are covered by the signature of the client that submits the operation. If they do not match, the server is a malicious/misbehaving server. SPORC assumes there exists some alternative server to switch to. Once a client validates a received operation from the server, there may exist conflicts between the new operation and other operations in his committed history and pending queue. This conflicts happen for two reasons: the server

may have committed additional operations since the new operation was generated or the client's local state might reflect uncommitted operations that reside on the client's pending queue that other client do not yet know about. To resolve this conflicts, a client must use OT. Operational Transformation (OT) provides a general model for synchronizing shared state while allowing each client to apply local updates. When clients generate new operations, they apply them locally before sending to others. To deal with conflicts that inevitably incur, each client first transforms the operations he receives from others before applying his new operations to his local state. If all clients transform incoming operations appropriately, OT guarantees that they will eventually converge to a consistent state.

Data confidentiality is achieved by encrypting all operations under a symmetric key, unknown to the central server. When a document is created, only the creator has access to it. The creator can then change the document's ACL by submitting meta-operations. The ACL of documents is sent to the server in clear text (every client maintains its own copy of the ACL and does not apply meta-operations that come from unauthorized users). When creating a new document, the creator generates a random AES key, encrypts it under his own public key and then writes the encrypted key to the document's initial create meta-operation. To add users to a document's ACL, the creator of the document submits a meta-operation that includes the document's AES symmetric key encrypted under the public key of each new user. To remove a user from a document's ACL, the creator of the document generates a new random AES key, encrypts it under the public keys of the remaining participants and then submits those encrypted keys as a new meta-operation. This meta-operation also includes an encryption of the old AES key under the new AES key to enable later users to decrypt old operations.

As stated above, for a new client to a reach the latest state of some document, he must download and apply the entire history of committed operations. This causes a lot of overhead. The authors suggest the use of *checkpoints*, a new client instead of downloading the entire history, downloads a checkpoint of operations and then only apply individual committed operations since the last checkpoint. To support checkpoints, each client from time to time uploads a compacted version of a document to the server, encrypting it under the current key of the document, as if it was a new meta-operation on the document. SPORC considers only a single cloud, the central untrusted server, and requires the cloud to have the ability to run code. Our intended system does not contemplate such kind of clouds.

2.2.3 Scheme by Fu and Sun

Fu and Sun [19] propose a *Scheme of Data Confidentiality and Fault-tolerance in Cloud Storage* that aims to ensure confidentiality of data, recovery of loss data and repair of error data. Data confidentiality is achieved by encrypting the data before being stored, unsing a boot password. This boot password is generated by a password (input by the user) and the file name. It avoids the difficulty of managing encrypted keys. By requiring the user to input the password several times, it makes the system less ubiquitous. Recovery of the lost data is done by dividing the original data into m blocks and encoding it into n blocks (where $n > m$) by an erasure code: tornado code. Recovery of the error data has two steps: detect whether the data has been tampered and recover it identically to the recovering of loss data. To detect whether the data has been tampered, each data block is stored with an hash of itself (hashed with a keyed-hash function using the data and the boot password).

Keeping user data confidential against untrusted servers in cloud computing is done by encrypting data through certain cryptographic primitive(s) and disclosing decryption keys only to authorized users. This general method has been widely adopted by existing works. These existing works resolve this issue by introducing a per file access control list (ACL) or by categorizing files into several *filegroups*. However, as the systems scales, the complexity of the ACL-based scheme would be proportional to the number of users in the system and the *filegroup*-based scheme is only able to provide coarse-grained data access control.

2.2.4 Scheme by Yu et al.

Yu et al. [31] address this open issue and propose a secure and scalable fine-grained data access control scheme. The authors have observed that in practical application scenarios each data file can be associated with a set of attributes which are meaningful in the context of interest: the access structure of each user thus can be defined as a unique logical expression over these attributes to reflect the scope of data files that the user is allowed to access (see an example scenario in Figure 2.3). Each attribute has a public key component, data files are encrypted using the public keys corresponding to their attributes and user secret keys are defined to reflect their access structures so that a user is able to decrypt a cipher text if and only if the data file attributes satisfy his access structure.

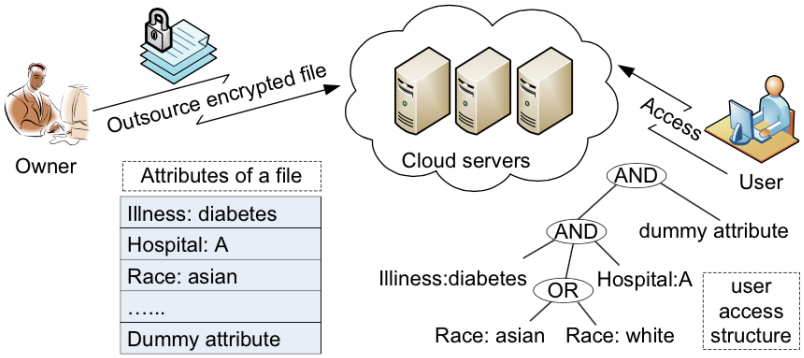

Figure 2.3: Exemplary case of attributes access control (by Yu et al. [17]).

This new design brings about the efficiency benefit that was lacking in other works: the complexity of encryption is just related to the number of attributes associated with a data file (thus is independent to the number of users) and operations of data file creation/deletion and new user grant do not involve system-wide data file update or re-keying. However on user revocation, re-encryption of data-files accessible to the leaving user is inevitable and also may need update of secret keys for all the remaining users. This introduces heavy computation to the data owner, so the authors propose to delegate this task of data file re-encryption and user secret key update to cloud servers.

The authors uniquely combine Key Policy Attribute-based Encryption (KP-ABE) [22, 30], Proxy Re-encryption (PRE) [15] and Lazy Re-encryption. As stated, each data file is associated with a set of attributes and each user is associated with an expressive access structure defined over the attributes. This kind of access control is done with KP-ABE to escort data encryption keys of files, such construction enables fine-grainedness of access control. However, this would introduce heavy computation overhead on the operation of user revocation (requires the data owner to re-encrypt all files accessible to the leaving user and to update secret keys for the remaining users). To overcome this issue, the authors combine PRE with KP-ABE thus enabling the data owner to delegate the computation intensive operations to cloud servers without disclosing the underlying file contents. To further reduce overhead on cloud servers, authors adopt lazy re-encryption allowing cloud servers to "aggregate" computation tasks.

- On system setup, the data owner creates the system public key PK and the system master key MK, then the data owner signs PK and sends it to cloud servers. On file creation the data owner assigns a unique ID to the file, randomly creates a symmetric encryption

key *DEK* and encrypts the data file using *DEK*. Then the data owner defines a set of attributes *I* for that file and encrypts *DEK* with *I* using KP-ABE. Finally, uploads the file to cloud servers.

- On adding a user to the system, the data owner assigns an access structure *P* and the corresponding secret key *SK* to this user. Then signs and encrypts the tuple (*P*, *SK*, *PK*) with the public key of the user. Next, the after owner signs and sends this encrypted tuple with the ID of the user and the secret key component of each attribute to cloud servers. On receiving, cloud servers verify the signature of the owner, if correct store the ID of the user and the secret key component of each attribute in the system user list *UL* and forwards the encrypted tuple (*P*, *SK*, *PK*) to the user. On receiving, the user first decrypts it with his private key, then verifies the signature and if correct accepts the tuple as his access structure, secret key and system public key.

- On user revocation, the data owner first determines a minimal set of attributes without which the leaving user's access structure P will never be satisfied. Next, the data owner updates these attributes by redefining their corresponding *PK* and *MK*. Then, the data owner updates user secret keys accordingly for all the remaining users. Finally, *DEKs* of affected data files are re-encrypted with the latest version of *PK*. As stated before, this introduces a heavy computation overhead so the tasks of data file re-encryption and user secret key update is delegated to cloud servers.

- On file access, cloud servers first verify if the requesting user is valid in *UL*, if true cloud servers send updated secret key components as well as the encrypted data file to the user. On receiving, the user first verifies if the received version of each attribute is newer than the current version he knows and, if so, decrypts the *DEK* and then decrypts the data file using *DEK*'s.

- File deletion can only be performed upon request by the data owner; the owner sends the file's ID along with his signature on this ID to cloud servers.

This work has similarities with systems discussed previously in this thesis. Plutus, a cryptographic file system, groups files with similar sharing attributes as a file-group and associates each file-group with a symmetric lockbox-key. Each file is encrypted using a unique file-block key which is further encrypted with the lockbox-key of the file-group to which the file belongs.

When the owner wants to share a file-group, he just delivers the corresponding lockbox-key to users. The complexity of key management is proportional to the number of file-groups, since the number of file-groups can be huge, Plutus is not suitable for fine-grained access control. SiRiUS, a layer over an existing file system such as NFS that provides end-to-end security, attaches each file with a metadata file that contains the file's access control list (ACL), each entry of which is the encryption of the FEK using the public key of an authorized user. SiRiUS has the same complexity in terms of each metadata file's size and encryption overhead, thus is not scalable.

2.2.5 Scheme by Zhao et al.

Zhao et al. [33] propose a system for trusted data sharing over untrusted cloud providers. First by encrypting the data before storing on the cloud. Second, on sharing the data, by re-encrypting it without being decrypted first (the re-encrypted data will then be cryptographically accessible to the authorized user only). This process does not reveal the clear data to the cloud provider at any stage. The cloud storage provider helps to enforce the authorization policy for data access, this enforcement should not reveal any information to the cloud storage provider. In order to allow a piece of data to be encrypted multiple times and be decrypted in a single operation, the authors propose a progressive encryption scheme based on Elliptic Curve Cryptography: the Progressive Elliptic Curve Encryption (PECE).

The sharing scenario is as follows. Alice has a piece of data that is kept on the cloud and wants to share it only with Bob. First, Alice encrypts her data with her private secret key and keeps the data on a cloud storage provider. Second, Bob sends a request to Alice asking for access permission to the data (sends his public key). Third, Alice sends a credential to the cloud storage provider for the re-encryption of the data and sends a credential for Bob to decrypt the re-encrypted data with his private key. Fourth, Bob acquires the re-encrypted data from the cloud storage provider and decrypts it.

The algorithm and protocol proposed by the authors requires a large amount of computation. This work assumes an active cloud storage provider able to re-encrypt data and send it to the authorized users. It is not clear how public keys are distributed nor how revocation is performed.

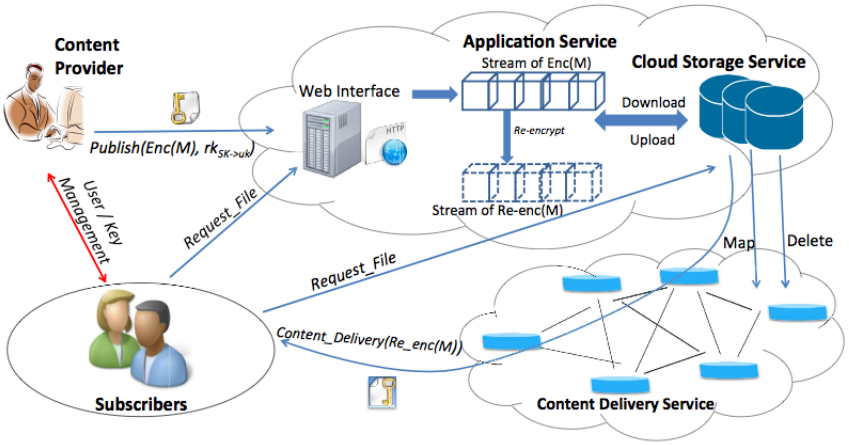

Figure 2.4: CloudSeal overview (by Xiong et al. [29]).

2.2.6 CloudSeal

CloudSeal [29] is an end-to-end solution for secure content storage and delivery via the public cloud, it ensures content confidentiality and content forward and backward security. CloudSeal is composed of a centralised storage service, a content delivery network (both provided by a cloud provider), a content provider and end users as subscribers. The content provider provides content to groups of subscribers using cloud based service from the cloud provider to store and distribute content. A group of subscribers are able to access a set of encrypted content stored in the cloud if the user successfully subscribes to the content provider (and can then decrypt delivered content).

The content provider encrypts content locally and then publishes the encrypted content to the public cloud-based storage. The content encryption performed is dual encryption with symmetric encryption at the first level and proxy-based encryption at the second level. The dual encryption enables CloudSeal to protect content confidentiality and uniquely bridge the proxy-based encryption scheme [11] and the secret sharing scheme [27]. The content provider has a public/private key pair (PK and SK). For each group of users the content provider randomly chooses the initial uk, the shared secret key for a group.

When the content provider wants to publish some content to the public cloud, it performs the dual encryption scheme as follows. First it encrypts the content M with a symmetric data encryption key DEK and then it further encrypts the content with the secret key SK. The

resulting encrypted content is stored in the cloud-based storage service by the application service. After publishing, to allow content to be retrieved by subscribers, the content provider generates a content re-encryption key rk with her secret key SK and the current decryption key uk. Then the application service obtains the newest rk from the content provider and re-encrypts the target encrypted content. After this the application server stores encrypted content in the cloud storage service and allows downloads from subscribers. Once a user obtains the encrypted content, the users decrypts the content with the current key uk. The key uk is obtained from the content provider when the user joins a group or is computed by the user.

When a subscriber is revoked, it requires key revocation operations in the group based on the k-out-of-n threshold secret sharing scheme. The content provider broadcasts this user's share of the secret key uk to the entire group so that the remaining users are able to generate the new secret key uk' autonomously. When a user joins a group, he has access to protected content. To prevent new users from accessing content published before they join (for forward secrecy), the key revocation process is required to be executed and the content provider issues the shares of future secret keys as well as the current secret key so the new user can compute the new secret key uk.

Similar security concerns in outsourcing data to untrusted cloud service have been discussed by Yu et al. in [31] and discussed above. In their approach data is encrypted by a symmetric key while the access to this symmetric key is controlled by KP-ABE algorithm. To manage dynamic user groups, they delegate *rekey* operations to the cloud and let the cloud server update secret keys of users and re-encrypt data without revealing the underlying plaintext. CloudSeal is different, CloudSeal only allows a content provider to perform *rekey* operation and the proxy re-encryption is performed directly on part of the encrypted content.

2.2.7 K2C

K2C, proposed by Zarandioon et al. [32], provides a secure and scalable access control protocol that supports easy sharing and revocation on hierarchically organized resources. Existing solutions reduce re-encryptions required as part of access revocation by using the lazy revocation technique. To support lazy revocation, cryptographic access control protocols need to use a key-updating scheme which provides key regression. Key regression enables a user holding a new key to derive an older key. However, these key-updating schemes are inefficient and not scalable as

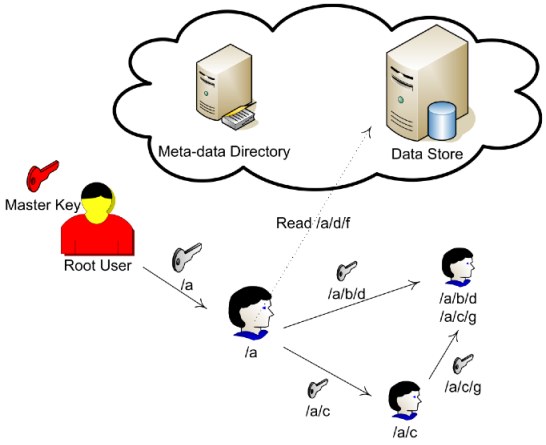

Figure 2.5: K2C Protocol (by Zarandioon et al. [29]).

they require complex data structures that need to be updated after each revocation. The authors introduce a new key-updating scheme called *AB-HKU* which is scalable, supports access hierarchies without requiring complex data structures and also enables support of lazy revocation without requiring any code to be executed on the cloud. *AB-HKU*, attribute-based hierarchical key-updating scheme, is a concrete construction for hierarchical key-updating (*HKU*) scheme. AB-HKU is realized on top of the Key-Policy Attribute-Based Encryption scheme (*KP-ABE*), used by Yu et al. in [31] (already discussed). K2C also introduces an attribute-based signature scheme called *AB-SIGN* which enables the verifier to ensure that a signature is produced by a user whose access policy is satisfiable by a set of attributes without learning the signer identity.

K2C runs between the root user, en-users and the cloud providers. The root user may be a system administrator who can specify access privileges of end-users. End-users may further delegate their access privileges to other users for easy sharing. Cloud providers in *K2C* are composed of two repositories: Metadata Directory and Data Store. The *Metadata Directory* is a cloud-based database such as Amazon SimpleDB[1] that provides all metadata associated with hierarchies and data objects, each object in the Metadata Directory has two properties: read access revision (RAR) and write access revision (WAR). The *Data Store* is a cloud key-value based storage system such as Amazon S3 and contains the actual content of each data object (the *key* is hierarchical path name of the data object and *value* is the actual content of the corresponding data object). Users keep a *Key-store* with all read/write access keys in their secure local machine. As illustrated in Figure 2.5.

[1] https://aws.amazon.com/simpledb/

The root user (system administrator) setups the K2C system. First signs up for the could services required to host the Metadata Directory and Data Store. Then uses *AB-HKU* scheme to generate public parameters and the master key and saves them in his Key-store. Next the root user shares the public parameters with the cloud providers that support K2C request authorization. Finally the root user defines the root directory by creating an entry in Metadata Directory with RAR and WAR initialized to zero. K2C request authorization enables cloud providers to block unauthorized requests based on the fact that each request is signed by user's access key for the target object using the *AB-SIGN* operation.

To write into a specific data object, a user needs to have the required write access key in his local *Key-store*. Then the user queries Metadata Directory to get RAR of the target object. The retrieved RAR and its path are encrypted using *AB-HKU* scheme and then signed with his write access key using *AB-SIGN* scheme. Finally the user constructs a key-value pair, formed by the path of the data object and the encrypted data (with the corresponding signature), and send the pair to Data Store. Data Store validates the request signature by querying Metadata Directory to get the WAR of the data object, if the signature is valid stores the data object.

To read a specific data object, a user needs to have the required read access key in his local Key-store. Then the user requests the data object from Data Store. Data Store validates the signature of the request and if valid sends the encrypted data to the user. Then the user decrypts the data, using *AB-HKU* scheme and the read access key, and validates the signature, using *AB-SIGN* scheme, to ensure that the data was produced by a user with proper write access. K2C treats sharing as a delegation operation where a user can authorize another user to a subset of his own access privileges. First the user needs defines the resource and access type (read/write) he wants to give to another user and gets the matching access key from his local Key-store. Then the user queries Metadata Directory to get the RAR/WAR for the target resource. Finally the user uses *AB-HKU* scheme to generate the required access key and sends it to the other user. To revoke a user's access on a specific data object, a user needs to make a request to Metadata Directory to increase the corresponding access revision number.

2.2.8 Discussion

The discussed single-cloud systems are not a solution for the problem related to this dissertation. They all rely on cloud service providers with the ability to run code. Other systems

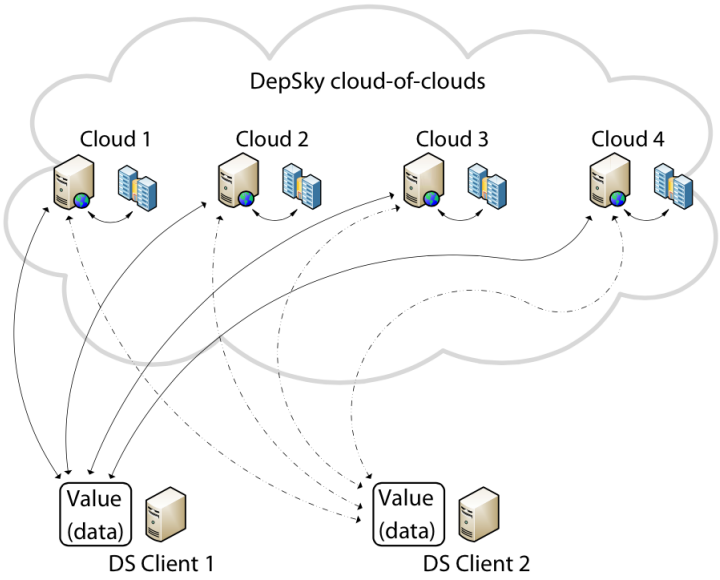

Figure 2.6: DepSky architecture (by Bessani et al. [12]).

that focus on access control techniques, such as CloudSeal and K2C, seem to provide a system that handles revocation more efficiently but the results prove otherwise.

2.3 Securing Multi-Cloud Storage Services

There are several works that provide a solution to better evaluate and manage free available store provided by different public clouds: aggregating different clouds in the same system/application. This systems are mainly commercial and do not provide confidentiality or when they do, they do not provide a way to transparently share files. In this section, existing systems are evaluated in terms of functionality and security guarantees.

2.3.1 Cloud of Clouds (CoC)

DepSky [12], illustrated in Figure 2.6, is a virtual storage cloud system that addresses the availability and the confidentiality of data stored using multiple cloud providers. Data is stored by combining a Byzantine quorum system [20] protocols and information-efficient secret sharing scheme (that combines symmetric encryption with a classical secret sharing scheme and an optimal erasure code). DepSky is located in the clients' machines as a software library to

communicate with each storage cloud, so there are no code being executed in the clouds. The DepSky library allows reading and writing operations with the storage clouds. Consequently, the DepSky system model contains: readers, writers and cloud storage providers (where readers and writers are the client's tasks). To control which readers are able to access the data stored, DepSky reuses the access control of the cloud provider itself.

The DepSky protocol guarantees availability because the data is stored by replicating it on several providers. Data is partitioned in a set of blocks using an optimal erasure code, in such a way that clients that have authorization to access the data will be granted access to the shares of (at least) $f + 1$ different clouds and will be able to rebuild the original data. To ensure confidentiality, the data is encrypted before being stored on the clouds. To do so without requiring a key distribution mechanism, DepSky employs a secret sharing scheme: a special party called dealer distributes a secret to n players, where each player gets only a share of this secret; at leat $f + 1 \leq n$ different shares are needed to recover the secret. This secret sharing scheme is integrated on the replication protocol: each cloud provider receives just a share of the secret, ensuring that no individual cloud will have access to the secret being stored. DepSky prevents individual clouds from disclosing the data: encrypts the data, stores the encryption key on the clouds without f faulty clouds being able to reconstruct it (using the secret sharing scheme) and reduces the size of the data stored in each cloud (using erasure codes). DepSky requires each client to have access to the same cloud providers.

2.3.2 Cloud Storage Aggregators

The systems discussed below aggregate all popular public clouds like GoogleDrive, Dropbox and OneDrive. They all require read and write permissions over user's files. Some provide a sharing mechanism.

Odrive [5] provides a file system plugin, like any other directory. When linking a cloud storage account to Odrive, the user gives read and write permissions on all user's files stored in the cloud, however Odrive does not store or see any user credentials (authentication is done against the cloud's API), the cloud authorization token is stored locally. Files of each linked account are placed in a directory named after the corresponding cloud provider inside Odrive's directory. This system uses progressive sync to avoid the overhead of syncing and downloading tons of files

the user does not need right away. Files and folders can be synced and unsynced as needed. There is no sharing mechanism available.

CloudHQ [3] is similar to Odrive but is instead a web application that whenever a user stores a new file in any of the linked clouds, replicates it to all other linked clouds. Also, no sharing mechanism is provided.

ZeroPC [9] is a web/mobile application that requires users to register in the system and provides a read only sharing mechanism between users. Similarly, CloudFuze [2] is an application that also requires users to register in the system. CloudFuze provides three sharing mechanisms: by creating a workspace with other users of CloudFuze, by sharing a file with another CloudFuze's user or by giving a link to a file to someone. The workspace serves to share files (stored in any linked cloud) with a group of users only with read permissions. Sharing a file directly with someone gives only read permissions to whom has the link, this link can be password protected, have an expiry data and maximum download count. Sharing a file with another user can be with read or write permissions, with write permissions the file is uploaded to CloudFuze servers and then it is updated by CloudFuze directly in the owner's cloud.

Podio, Otixo, and CloudKaf share the notion of a workspace to share files (with read only permissions). Podio [7] is a collaborative web application that allows users to link their cloud accounts so they can easily share file with other users. Otixo [6] is a web/mobile application that also allows users to move files between different clouds without downloading them: the file is instead cached to Otixo servers and deleted once the operation is completed. CloudKafé [4] is a web application only with workspace type of sharing which they call baskets of files.

SME [8] is a virtual cloud file system that can encrypt files before uploading to the cloud storage. Sharing of files can be done by workspace collaboration or via URL. Both cases are read access only. When accessing a shared encrypted file, the user needs to enter a password. No details are provided.

CloudFogger [1] is a desktop/mobile application that encrypts files with AES 256-bit where each file has its own unique key. Unlike other systems discussed in this section, CloudFogger does not aggregate clouds in itself, instead, uses local cloud applications of Dropbox, GoogleDrive and OneDrive (when installed by the user). Each user is assigned with an asymmetric RSA key

pair. To share a file with another user, the application encrypts the file's AES key with the other user's public key and appends it as an header to the file. Access and privileges to the file is done by the owner within the cloud. This kind of sharing mechanism is not convenient to the user.

The systems discussed in this section only provide an aggregator of clouds and sometimes a mechanism to share files with other users, however none provides a transparent file sharing with write permissions.

2.4 Summary and Comparative Analysis

All the systems discussed along the present section are summarized in Table 2.1 to easily correlate the properties of each work. The systems are compared in terms of confidentiality and integrity of data and metadata, existence of a data sharing mechanism and key distribution.

When data to be shared among hosts with some access permissions defined by the system administrator. This systems trust on the storage server(s) and the administrator(s), all players are in a trusted domain. Once data is stored in a not so trusted domain, concerns arise regarding data confidentiality. First appears CFS that encrypts directories but since it does not have a key distribution mechanism, it does not allow sharing of data (unless the owner manually gives the key to another player whom has access to the corresponding host). TCFS, EFS, SiRiUS, Plutus and Cepheus emerged to guarantee both sharing and confidentiality of data.. TCFS allows sharing of data within groups by creating a group-key that encrypts all file-keys of the files belonging to that group, the group-key is distributed by a threshold. EFS allows sharing of data by encrypting the file encryption key (FEK) with each public key of users that will have access to the file, stored along with the encrypted file. SiRiUS is much like EFS but does not rely on the underlying file system to access control permissions to files, uses read-write keys. On revocation, TCFS, EFS and SiRiUS immediately re-encrypt all affected keys and files. Plutus and Cepheus use lazy re-encryption on revocation to avoid some performance overhead.

Storage on distributed file systems started to be replaced with services offered by cloud providers where data is stored on untrusted domains. BlueSky appeared as a solution to substitute local storage servers with a proxy that provides the illusion of a single traditional server backing up data to the cloud. The authors do not provide any details about data sharing.

SPORC achieves all security properties desirable for Storekeeper however relies on the cloud to run code and trusts it to operate correctly (even though clients can detect a misbehaving server). Teapot is an implementation of Depot that provides guarantees of consistency and availability using multiple servers, it does not provide data confidentiality. This multiple servers are not the same as multi cloud support, it simply uses multiple Amazon S3 servers. SPORC, like EFS and SiRiUS, on revocation needs to perform re-encryptions as much times as the number of users remaining with access to some file. CloudSeal and K2C, both based on previous works by Yu et al. and Zhao et al, try to provide a fine-grained access control scheme that will not increase in complexity as the system scales. The experimental results reveal that their solutions are not as good as they seem. Also like SPORC they rely on clouds with the ability to run code.

With the existence of many public cloud's storage, systems emerged to provide a centralized access to all different accounts. Odrive and CloudHQ only provide this centralized access (CloudHQ also replicates files across the different accounts). ZeroPC, Podio, Otixo, CloudKafé and SME also provide a way to share files in a read-only access. CloudFuze provides a share with write access but the file is uploaded to CloudFuze's server to allow someone else to modify the file. No confidentiality of data is assured. SME provides confidentiality of data against the cloud providers but not against SME servers. CloudFogger provides the same confidentiality as SME but does not have innate support of data sharing and cloud storage.

DepSky is the most complete system: provides data confidentiality, keys are distributed securely through the clouds (no need of other servers) and ensures integrity and availability of data, however is not appropriate for transparent data sharing. Data access is done by the clouds and all users in the system need to have accounts in all pre-defined cloud providers.

As comparison reference with the systems in Table 2.1, Storekeeper falls in the multi-cloud category fully satisfying the properties of data sharing, data confidentiality, key distribution and data integrity, and some metadata confidentiality. The next chapter describes how such properties are achieved.

Category	System	Data Sharing	Data Confidentiality	Metadata Confidentiality	Key Distribution	Data Integrity	Metadata Integrity
File Systems	CFS [14]		✓	✓			
	TCFS [16]	✓	✓		✓	✓	
	EFS [25]	✓	✓				
	Cepheus [18]	✓	✓	✓	✓	✓	✓
	SiRiUS [21]	✓	✓	✓	✓	✓	✓
	Plutus [23]	✓	✓		✓	✓	✓
Single-Cloud	BlueSky [28]		✓			✓	
	SPORC [17]	✓	✓	✓	✓	✓	✓
	DepSky [12]		✓	✓	✓	✓	✓
	Fu and Sun [19]		✓	✓		✓	
	Yu et al. [31]	✓	✓		✓		
	Zhao et al. [33]	✓	✓		✓		
	CloudSeal [29]	✓	✓		✓	✓	
	K2C [32]	✓	✓	✓	✓	✓	✓
Multi-Cloud	Odrive [5]						
	CloudHQ [3]						
	ZeroPC [9]	✓					
	CloudFuze [2]	✓					
	Podio [7]	✓					
	Otixo [6]	✓					
	CloudKafé [4]	✓					
	SME [8]	✓	✓				
	CloudFogger [1]		✓	✓	✓		

Table 2.1: Systems comparison.

3 Storekeeper

In this chapter, we present the design of Storekeeper, a security-enhanced cloud storage aggregator. Firstly, we outline the goals and principles that guided the design of our system (Section 3.1). Next, we make an overview of the system by presenting its high-level architecture, describing its functionality, and clarifying our assumptions and threat model (Section 3.2). Then, we focus on the specific design details of our system: file namespace (Section 3.3), management of user identities and authentication (Section 3.4), consistency semantics of file operations (Section 3.5), access permission model (Section 3.6), file operations (Section 3.7), and lastly scalability and fault tolerance techniques (Section 3.8).

3.1 Design Goals and Design Principles

Our central goal is to design a system that can provide a secure cloud storage aggregation service. A user should be able to register her cloud accounts in the service and have a unified view of all her files irrespective of the specific account where files are actually located. A user should also be allowed to share her files with other users of the service, and possibly revoke access to previously shared files. The aggregation service should not demand for specific access to the user's cloud accounts and data confidentiality must be preserved. Note that, in this work, we are not focusing on preserving the integrity or availability of files.

In designing such a service, we follow three main principles:

P1: Users are allowed to write only on their accounts. Normally, individual users of cloud storage services tend to be frugal in terms of how they utilize the storage capacity available in their accounts. This is because most users are restricted to a few gigabytes of storage per account before they need to pay for extra storage space. To reflect this concern when designing a multi-user cloud storage aggregator, we restrict users from writing data on cloud-backed accounts that they do not own. By ensuring that each user can only store

content in their own accounts, we prevent abuses from users willing to "highjack" the cloud storage space owned by other users. This principle differs from Cloud of Clouds (CoC) in which there is a common pool of shared cloud-backed accounts. Users freely contribute to that pool knowing that the resulting combined storage space is shared and can be used by any other user to store content. In contrast, in a cloud storage aggregator (CSA), the cloud accounts of each user must be managed independently and isolated from each other.

P2: Users must have their files physically located on their accounts. Since in a CSA system, cloud accounts must be managed independently per user, it is important to ensure that users keep control of their files, even if they share it with others. Thus, by constraining the location of files to reside in cloud accounts contributed by the owners of such files, users retain the control of their files. In other words, their files will always be located in their own accounts, in spite of sharing a file with a user and possibly revoking access to that file in the future. In contrast, CoC aims to offer to their users complete file location independence and let it be controlled. Location independence provides a good abstraction for CoC because they were primarily designed to provide fault tolerance and offer the illusion of a single cloud repository, whereas in CSA, it is necessary to preserve independent storage domains for individual users.

P3: Users must not need to maintain persistent state at the client side. One of the factors that make cloud storage services so convenient for users is that all persistent state is stored online. All users need to remember to access their files is their username and password. This feature makes these services very convenient for users and also more robust to data loss or to security breaches than if the user was required, for example, to manage private keys or other sensitive cryptographic material. Similarly, to preserve these properties, a CSA system must take care not to require the user to maintain critical data on the client side. CoC systems, on the other hand, were not designed with usability in mind. For that reason, users are required to maintain cryptographic keys and provide for their adequate distribution in order to allow for file sharing.

Together, these principles constitute guidelines to the design of Storekeeper: P1 and P2 ensure that aggregated cloud space is isolated between users, and P3 preserves usability.

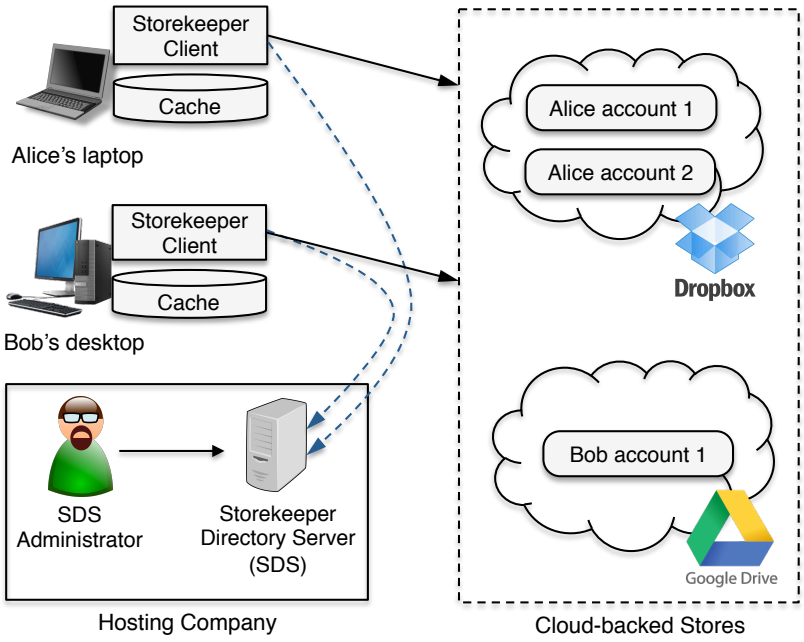

Figure 3.1: System overview.

3.2 System Overview

This section presents an overview of Storekeeper, a security-enhanced cloud storage aggregator that follows the design principles stated above. Storekeeper consists of two main components: a client application and the Storekeeper Directory Server (SDS). The client is an application that runs on the users' devices and serves as an interface to the system. Similarly to the Dropbox client application, the Storekeeper client maintains a local cache of the user files persistently stored on cloud-backed *stores*. Stores represent cloud accounts hosted by cloud services such as Dropbox or Google Drive and are contributed by the users. The SDS is the heart of Storekeeper. This component runs on a dedicated server and manages the meta-data associated with users, files, and stores. Files themselves are not stored in the SDS, but on stores provided by users.

Figure 3.1 illustrates the architecture of the system using a simple deployment scenario. Consider that Alice and Bob are faculty that work in the same university. Their university decided to deploy Storepeeker in order to allow its faculty to collaborate using their personal cloud accounts. To enable this, the IT designated an administrator who is responsible for installing the SDS on an internal server. Since Alice and Bob are two faculty members, the administrator creates user accounts in the local Storekeeper deployment and sends them access credentials (e.g., to their professional email accounts). Using these credentials—a username and

password—Alice and Bob can log into the system and register any personal cloud accounts they may have. In this case, Alice has two personal accounts in Dropbox and Bob has one account in Google Drive. By adding these accounts to the system, Storekeeper will interpret them as stores where files can be located and shared between users, Alice and Bob can have a unified view of all their files. Alice will see an aggregation of all her files from accounts 1 and 2, and Bob all files from account 1. This unified view that each user sees is named *workspace*.

Each user can then share files with each other, independently of whether or not they have accounts on the same cloud provider. For example, Alice can share a file which is located in Dropbox with Bob, who does not have any cloud in Dropbox. Storekeeper will provide that the file will be mounted on Bob's workspace. Depending on the permissions that Alice grants to Bob, he will be able to read or write the file's contents. At any point in the future, Alice can revoke access permissions that were previously granted to Bob, point at which he will not be allowed to access the file any longer. In all this process, Storekeeper ensures that Alice and Bob can write only on their accounts (P1). If Bob edits a shared file owned by Alice, the file updates performed by Bob are staged in one of Bob's accounts and then transparently fetched by Alice's client and incorporated in Alice's primary file version. We call this technique *file homing* (see Section 3.7.4). Storekeeper also provides that each users files reside in their owner's accounts (P2) and that all critical data and meta-data is stored both in stores and in the SDS so as to avoid burdening the user with state maintenance (P3). Files are encrypted at the client endpoint. The SDS is not entrusted with credentials that allow for direct access to users' cloud accounts. Instead, access to cloud stores is performed at the client side only, ensuring that users retain exclusive control of their accounts.

Storekeeper is designed to provide end-to-end data confidentiality. We consider cloud providers and SDS administrators to be *honest but curious*. This means that both these parties can passively listen to all exchanged messages and may try to learn information about users' files, but follow the protocols and do not launch active attacks. We assume that cloud providers and SDS administrators are not malicious, case in which they could actively attempt to extract information, e.g., bengaging with the user using social engineering. Note that it is not the focus of this work to preserve the integrity and availability of data. In particular, we do not prevent intentional modification of data or meta-data by SDS or cloud providers. Our focus is on confidentiality protection only. Nevertheless, we assume that the communication channels

are insecure. They can be actively eavesdropped or manipulated by external malicious agents. We do not mitigate side-channel attacks and assume that cryptographic algorithms are sound.

3.3 File Namespace

We now describe the design of Storekeeper in more detail starting with its file namespace. In a cloud aggregator service like Storekeeper, it is necessary to define how the files physically located on heterogeneous cloud-backed storage are exposed to the user under a common file naming scheme. In addition, it is necessary to map such names to the identifiers that cloud services use to uniquely identify user files. In doing so, we need to overcome low-level differences in cloud service APIs regarding the way how files are identified. For instance, in Google Drive REST API, files have names, but are uniquely identified by a long string ID internally generated by the service. As a result, there can be two files in the same directory with the same name. In Dropbox, however, files are uniquely identified by its absolute path which means that multiple files reside in the same directory if they have identical names. Storekeeper must handle this heterogeneity and provide users a consistent file naming semantics.

Another aspect that needs to be addressed when aggregating cloud accounts is how to handle filename collisions. If files with the same name exist in two different stores, when such stores are joined, it is necessary to avoid collisions. Similarly, if a user shares a file with another user, care must be taken in order to avoid collision between the name of the newly shared file and the name of files that previously exist in a user's store. Lastly, it is possible for single users to join multiple deployments of the Storekeeper system, managed by different organizations or departments. Therefore, it is necessary to prevent name collisions between files with similar names that are located in different Storekeeper accounts.

To illustrate how Storekeeper addresses these issues, we leverage the usage scenario introduced in Section 3.2. Figure 3.2 represents (1) the aggregated files (workspace) that Alice and Bob see mounted in their local devices, on the left, (2) the way these files are physically maintained on the cloud accounts (stored), on the right, and (3) how the file names in the workspace are mapped to the file names in the users' stores, represented by the arrows.

The first important aspect is the notion of *domain name*. Every deployment of Storekeeper is given a unique name. This name is associated with the SDS server where the system is deployed

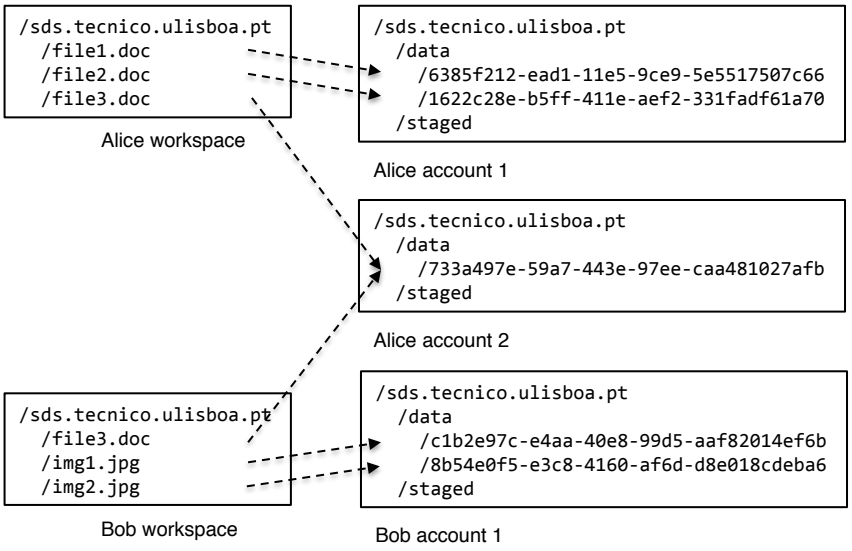

Figure 3.2: File name mapping in Storekeeper.

and is determined by the SDS administrator. To make this name unique, the SDS administrator may leverage the DNS name of the hosting organization. The domain name of this example is `sds.tecnico.ulisboa.pt`. Its purpose is to avoid name collision between different Storekeeper deployments while providing the user a complete view of all her files managed under Storekeeper. Collusion avoidance is achieved by using the domain name to mount / create unique folders on the workspace / stores of a given user. Aggregated files for each Storekeeper domain are then placed in such folders according to their respective domain name (see Figure 3.2). Storekeeper creates domain folders for each store. Each folder contains two directories: `data` and `staging`, which contain all user's data files and pending file updates, respectively (see Section 3.7.4).

To uniquely manage files in cloud-backed stores and overcome cloud API differences, filenames are assigned a unique identifier which we call *File ID* (FID). This identifier is not exposed to the user and is used internally by the Storekeeper client to fetch files from their respective cloud stores. To access a file, the Storekeeper client uses a URL that is provided by each specific cloud hosting service. In addition to FID, each file has a human-readable filename, which is the name that will be shown to the user. This name will then be mapped to the corresponding FID. If someone shares a file whose name matches the name of a file already present in the user's workspace, Storekeeper resolves this naming conflict by simply adding the prefix "shared-" to the shared file. Since FIDs are globally unique, conflicts are not expected to occur.

3.4 User Credentials

Storekeeper defines specific user credentials as a basis to overcome two challenges. The first challenge involves *user authentication* when accessing cloud stores. In order to access a cloud store s through the respective API, a typical third-party application (including the Storekeeper client) needs to authenticate itself towards the cloud service by providing a specific credential named *access token* (AT). Access tokens are provided by the cloud service to allow for secure authentication without the need for the user to interactively input username and password. Since the leakage of such tokens would allow for unrestricted access to users' stores, Storekeeper needs to maintain them securely. In particular, to prevent users from accessing accounts from each other, access tokens must be strictly associated with their respective owner and read-protected from the remaining users. Read protection must also cover the SDS server, otherwise a curious SDS administrator can gain unrestricted access to users' cloud accounts.

A second challenge is related with *confidentiality protection* of user files. In order to provide end-to-end confidentiality covering both cloud providers and SDS, file must be encrypted at the client side with a symmetric key – a *file encryption key* (KF) – before sending the file to a cloud store. The natural approach to protect this key while assuring that the file owner alone can access it is to use PKI: the user generates a public-key pair that represents a *user key* (KU) and encrypts KF with the public key (KU^+). By ensuring that the user's private key (KU^-) remains secret, access is restricted since only the user can decrypt the file encryption key using KU^-. The problem with this approach, however, is that the user is burdened with the responsibility to maintain the private key securely, which contradicts principle P3 (see Section 3.1).

Access tokens and user keys constitute Storekeeper's user credentials. To provide secure storage support for user credentials by encrypting access tokens and user keys with a symmetric key – *login key* (KL) – which is a function of a username u and a password p: $KL = h(u||p)$, where "||" represents concatenation. The username corresponds to a domain-specific user identity (e.g., `alice@sds.tecnico.ulisboa.pt`) which is created at the Storekeeper server by the SDS administrator. To use the system, the user must first sign up to the Storekeeper service with the username assigned by the SDS administrator and provide a password. When reading username and password from the user, the Storekeeper client calculates the login key KL, generates a new user key KU, and encrypts the username u and private key KU^- with KL. The resulting ciphertext $\{u, KU^-\}_{KL}$ is sent to the SDS along with the user's public key KU^+.

Note that, to prevent disclosure of the user key, neither the login key nor the password are sent to the SDS. Every time the user needs to recover the user key, it can download the ciphertext from the SDS, generate the login key from his login credentials (username and password), and decrypt the ciphertext, yielding the private part of KU.

Similarly, access tokens required to access each of the user accounts can be stored encrypted in the SDS based on the login key. Whenever the user adds a new cloud store s to the service, the Storekeeper client obtains the respective access token (AT_s) directly from the cloud provider and encrypts the token and username with the password-dependent symmetric key KL. The resulting ciphertext $\{u, AT_s\}_{KL}$ can be safely provisioned to the SDS and recovered to the client whenever the access token is required by the Storekeeper client to access each account. Note that the access token is never sent in clear text to the SDS. Thus, the SDS can avoid burdening the user with key management responsibilities by keeping user credentials encrypted:

$$u,\ \{u, KU^{-}\}_{KL},\ [\{u, AT_{s0}\}_{KL}, ..., \{u, AT_{sn}\}_{KL}]$$

This tuple corresponds to: username u, encrypted user key, and list of encrypted access tokens AT_s, one for each store s (from 0 to n) that the user has registered in the server. Later in this chapter, we explain how user keys and access tokens are used in Storekeeper.

3.5 Consistency Semantics

In systems such as Storekeeper, there can be multiple replicas of the same file located both in cloud stores and in client-side caches. As opposed to systems such as Depsky [12] and SCFS [13], which maintain multiple file replicas on the cloud for fault tolerance reasons, Storekeeper is primarily focused on cloud storage aggregation and therefore maintains a single file replica on the cloud. To enforce compliance with principle P2 (see Section 3.1), the primary file replica is physically preserved in one of the cloud stores registered by the owner of the file. We name *home store* the cloud store where a given file is located. As Storekeeper users deploy local clients on their devices, the client application downloads copies of user files from the files' respective home stores into a local cache. As writes are performed to the local file copies, potential inconsistencies may arise between local copies and the primary file version maintained in the cloud, inconsistencies that Storekeeper must resolve.

Many consistency schemes have been proposed in the literature targeting specific use cases. In the context of multi-cloud services, systems such as SCFS [13] were designed to provide strong-consistency guarantees. Strong consistency prevents conflicting modifications to file replicas and it is suitable for usage scenarios where network connectivity is high, for instance within a same company. However, cloud services such as Dropbox and GoogleDrive were designed to operate under different assumptions. Although they benefit from a highly connected environment, typical cloud storage services manage to operate well when devices are frequently disconnected from the network. To support such scenarios, a local client application operates in disconnected mode in which file copies are maintained in a local cache, local write operations are tracked, and when devices reconnect, a reconciliation mechanism takes place in order to resolve any conflicting writes to common files. This consistency model is broadly called *eventual consistency* in which users are allowed to read or write local copies which are eventually delivered and reconciled in the cloud. Given that eventual consistency favours file availability and is already familiar to users, we opted to adopt eventual consistency in the design of Storekeeper.

To provide eventual consistency semantics in Storekeeper, each file has a *version number* which is centrally managed by the SDS. The version number of a file starts in 1 and is incremented whenever the file is updated to its home store, hence we refer to this version number as v_h. When a file copy is created into the user's local cache, the Storekeeper client keeps a record containing (1) the file's *local version number* (v_l), which corresponds to v_h at the time the file was downloaded from the home store, and (2) a *dirty flag*, which tells whether or not the local file copy has been modified by the user. If the user modifies the local file replica, the client sets the dirty flag and places the file in a queue to be propagated to the SDS. This means that file updates are not immediately forwarded to home store and external user caches; local copies maintained by each client can be stale and may be need to be updated later. From a user's perspective, Storekeeper implements a *read-your-own-writes* semantics with respect to the local replica. In other words, local writes performed by the user are visible by local reads.

Periodically, provided that there is connectivity to files' home stores, the client contacts the SDS in order to synchronize the local cache with home stores. Synchronization involves checking both (1) if the local files are outdated (i.e., someone has updated the file) and (2) if the home files are outdated (i.e., the local user has written the file). Checking for file outdates is achieved by comparing the version numbers of local (v_l) and remote copies (v_h). If the dirty bit is clear,

no changes were performed to the local replica. Thus it is only necessary to check if there is a most recent version of the file in the cloud ($v_l < v_h$) and refresh the local cache. However, if the dirty bit is set two situations can happen. If $v_l = v_h$, then nobody else has submitted a new file version to the home store. Therefore, the client can simply send the new file version and tell the SDS to increment v_h. However, if $v_l < v_h$ a conflict has occurred since a concurrent write was performed to the file. To resolve this conflict, the local copy is put in quarantine in a special directory so that the user can perform manual reconciliation. This behavior is similar to what users are accustomed to today in popular cloud storage services, namely Dropbox.

3.6 Access Permissions Model

Existing cloud storage services allow files to be shared between internal users: users that have local accounts in the cloud service. The semantics of how file accesses can be performed is normally defined by an *access permission model* which is specific to each service. For example, Dropbox users can share files with other users and set access permissions. Read permissions are permitted to individual files, but write permissions are not. Write permissions can be granted only to the directories. Thus, to share a file with write-permission with another user, the file owner needs to share the directory where the file is located. This entails granting to the beneficiary user the privilege to create new files or read / write / delete any other files located in that directory. Google Drive, on the other hand, provides a finer permission setting granularity. It is possible to share individual files or directories in read-only or read-write modes. Read-write permission includes the privilege to delete a file / directory.

However, cloud storage services implement different access permission restrictions for internal and for external users: external users do not have local accounts in the cloud service. External users have access to files via URL and tend to have less privileges than internal users. For example, in Dropbox and Google Drive, external users can access individual files or directories with read permissions only; writes are not allowed. Given that different services offer different access permission semantics, Storekeeper needs to define an access permission model that can accommodate this heterogeneity. Furthermore, to provide fine-grained sharing capability, we require that users can set access permissions at the file level (similarly to Google Drive). Since directories can be seen as a special files, we focus primarily on file access permissions.

We design Storekeeper's access permission model to satisfy these requirements. In Storekeeper, every file has a file *owner*. The file owner has full access privileges over a file, which include: reading the file, writing the file (i.e., modifying or deleting it), and setting / clearing access permissions of the file. Modifying access permissions of a given file entail granting or revoking access privileges to a given user – the *grantee* – depending on the specific *access permission* that has been given to the user. A grantee can be given one out of three possible access permissions: *read* (R), *write* (W), and *share* (S). Read permission allows the grantee to read the content of a file. Write permission allows users to read the file, or modify / delete it. Share permission accumulates the privileges of write permission with the ability to share the file with other users. This means that a share permission grantee can set access permissions of other users to that file. Note, however, that such a grantee can never restrict the privileges of the file owner. The file owner retains full control of his files; at any time, a file owner is allowed to revoke any privileges that have previously been given to some grantee.

$$ACL_f : u_o, [(u_1, R), (u_2, W), (u_3, S), ...)]$$

Above, we see the basic data structure responsible to support Storekeeper's access permission model: an Access Control List (ACL). For each file f, the SDS contains an ACL that identifies the file owner (u_o) and a list of descriptors containing the file permission (R, W, or S) granted to each grantee (u_i). An empty list means that the file is not shared; it is private to the file owner. In the example above, users u_1, u_2, and u_3 have read (R), write (W), and share (S) permission, respectively. File permissions are enforced by a combination of access control mechanisms at the SDS and cryptographic protocols. Table 3.1 shows, on each row, the file operations supported by Storekeeper and the respective authorization result yielded to a grantee based on his access permissions (columns 2-4). File `read` operations are allowed under R, W, or S permissions. Creating, updating, or deleting a file are permitted with W or S permissions. Changing file permissions (`chperm`) is allowed with S permission only. Whenever file operation is performed to Storekeeper, the SDS must always authorize the operation by checking the ACL against the identity of the user who is performing the operation. Next sections explain how ACLs are managed as we present the details or Storekeeper's file access operations.

Operation	R	W	S
`read`	yes	no	no
`create`	yes	yes	no
`update`	yes	yes	no
`delete`	yes	yes	no
`chperm`	yes	yes	yes

Table 3.1: Storekeeper's access control matrix.

3.7 File Operations

This section presents the algorithms responsible for the implementation of Storekeeper's file operations: `create`, `read`, `update`, `delete`, and `chperm` (see Section 3.6). We start our description by presenting a basic algorithm and then revisit it to address emerging challenges.

3.7.1 A Basic Algorithm

To present the basic file operations algorithm, ignore for now file operations involved in granting or revoking access permissions (`chperm`). Consider for now that we have two types of operations: read and write. Write operations comprise: `create`, `update`, and `delete`. Consider also the use case depicted in Figure 3.2 in which Alice and Bob are Storekeeper users in domain `sds.tecnico.ulisboa.pt` and share file `file3.doc`. Assume that Alice is the owner of this file and that Bob has W privileges (i.e., he can read and write the file).

We start by presenting a basic approach to allow Alice and Bob to read / write the file while providing end-to-end data confidentiality. When Alice creates the file, the client application running on her device takes care to generate a file encryption key (KF), then (1) encrypt the file with this key and upload it to the file's home store, and (2) encrypt KF with the public part of Alice's user key (KU_A^+) and send it to the SDS (see Section 3.4). In order to read the file in the future, all that Alice's client needs to do is to download the encrypted file from the home store, fetch the encrypted KF from the SDS, and then use the private part of Alice's user key (KU_A^-) to decrypt the file key KF and then decrypt the file with KF. File updates can be performed by re-encrypting the new file and uploading it onto the home store. File delete can be implemented deleting the file from the home store and updating the SDS's internal data structures. Since both the file and the respective encryption file are encrypted and can be read by Alice alone, neither the SDS nor the cloud provider can read the file contents.

To allow Bob to read or write the file, all that needs to be done is to securely share the file key KF with Bob, which can be done by encrypting it with the public part of Bob's user key (KU_B^+). When Alice, the owner of the file, grants W permissions to Bob, in addition to updating the file's ACL with W on the SDS, the SDS receives from Alice's client the file access credential $\{KF\}_{KU_B^+}$. When Bob requests access to this file, the SDS forwards this credential to Bob's client, allowing the file key KF to be recovered and the file operation to carry on.

3.7.2 Permission Enforcement and Revocation

In Storekeeper, permission enforcement is based on a combination of access control checks performed at the SDS and cryptographic protocols implemented both by the client. The relevant data structures required in this process are maintained in the SDS. Considering the basic scenario described in the section above, the relevant data structures that control access to file f (`file3.doc`) can be represented by tuple P_f:

$$P_{f_0} : (\texttt{alice}, \{KF\}_{KU_A^+}), [(\texttt{bob}, W, \{KF\}_{KU_B^+})]$$

Essentially, P_{f_0} implements an ACL (Section 3.6) in which Alice is the owner of the file, and Bob has writing privileges to the file. In order that Alice and Bob can decrypt the key for reading and re-encrypt it for writing, the file key KF is encrypted with the public part of their respective user keys (see Section 3.4). P_{f_0} is verified every time a user performs a file operation to f; if the user does not have adequate permissions, the operation is refused by the SDS.

However, while this simple approach works properly for write and change permission operations, it does not properly handle reading privileges. In particular, this naïf approach is insecure when read permissions are revoked. When Bob reads the file for the first time (which he can because he has W permission), in addition to obtaining the file's key KF, Bob's client retrieves also the file URL, which points to the persistent location of the file in Alice's cloud store. If Alice decides to unshare the file with Bob and revoke his permissions, tuple P_{f_0} is updated so as to exclude Bob from the ACL and the SDS will no longer authorize file operations on f performed by Bob. However, if Bob retrieves the key KF and file URL from the local client, it is still possible to continue accessing the file, circumventing the mechanisms implemented by the SDS.

A commonly used approach to handle revocation is to re-encrypt the file whenever the revocation operation takes place. Triggered by the `chperms` operation, the idea is to generate

a new file key KF' and re-encrypt the file with KF'. To allow users with reading privileges to continue reading the file, the ACL needs to be updated to include the encrypted KF' key and exclude Bob from the ACL, as represented in the revised tuple:

$$P_{f_1} : (\texttt{alice}, \{KF'\}_{KU_A^+}), []$$

According to this tuple, which represents the ACL *after* revocation takes place, only Alice – the file owner – can read the file. Although Bob can still retrieve the encrypted file from the home store (using the file URL), he cannot retrieve the new key KF' with which the file has been encrypted. Therefore, Bob will not be able to read future versions of the file. The main downside of this approach, however, is the performance overhead of re-encrypting the file and encrypting new file key with public keys of authorized users as defined in the ACL.

To overcome this problem and implement efficient revocation without the need to reencrypt the file, we employ three techniques, enumerated below. To illustrate each step, we show the tuple state when both Alice and Bob belong to the readset (P'_{f_2}), Bob's permissions were revoked (P''_{f_2}), and Alice submits new update (P'''_{f_2}). Their differences are underlined.

1. **Readers have access to a read key only:** Instead of granting readers direct access to the file key KF, they are given access to an intermediate symmetric key that is shared between all users that can read the file. This key – named *read key* (KR) – is then encrypted with readers' public key and added into the ACL. The read key is then responsible for encrypting the file key KF, which effectively encrypts the contents of the file.

$$P'_{f_2} : \underline{\{KF\}_{KR}}, (\texttt{alice}, \{\underline{KR}\}_{KU_A^+}), [(\texttt{bob}, W, \{\underline{KR}\}_{KU_B^+})]$$

2. **Revocation generates a new read key:** Every time revocation occurs, a new read key KR' is generated and the ACL updated such that the new read set has access to the new read key. This means that KR' must be encrypted with readers' public keys and ACL updated with this information.

$$P''_{f_2} : \{KF\}_{\underline{KR'}}, (\texttt{alice}, \{\underline{KR'}\}_{KU_A^+}), []$$

3. **Writes generate new file key per write:** Every time a writer submits a file update, instead of encrypting the file with the file key KF of the previous file version, the writer

generates a new file key KF', encrypts the file with the new key, and replaces the encrypted file key KF with the new encrypted file key KF', which must be encrypted with the read key in order to allow readers of the ACL to continue reading the file.

$$P'''_{f_2} : \{\underline{KF'}\}_{KR'}, (\texttt{alice}, \{KR'\}_{KU_A^+}), []$$

Intuitively, with this method, revocation introduces a disruption such that readers in the old read set will not be allowed to see future writes performed by the new read set. Therefore, next time a write operation is performed the file key of a new file version will be encrypted with a new readers key which is inaccessible to the revoked user. In summary, this is achieved by adding an indirection key (the read key), re-encrypting this key upon revocation, and refreshing the file key every time a file is updated. As a result we manage to revoke access to the file without re-encrypting it.

3.7.3 Staging Space

The basic algorithm described in Section 3.7.1 ignores several issues. One first aspect that must be covered is how a grantee can access a file from the file's home store. To access `file3.doc`, Bob (the grantee) requires not only the file key KF in order to decrypt or re-encrypt the file content, but also some mechanism provided by the cloud provider where the file is stored that allows Bob's client to retrieve the encrypted file from Alice's account. Handing over to Bob the access credentials (AT) of the cloud store is out of the question since it would give Bob full control of Alice's account. An alternative is to provide an *external URL*. Cloud providers allow a file to be accessed externally by non-users for read-only operations. The external party only needs to obtain an URL to the file generated by the cloud provider. With such an URL Bob's client can fetch the encrypted file from Alice's account. However, this mechanism does not allow for Bob to upload a new re-encrypted file to Alice's account. For example, if a file is homed in a Dropbox store, it is possible to obtain an URL to externally fetch the file, which would serve the needs of read operations, but that URL cannot be used to submit file updates.

As discussed in Section 3.6, Dropbox does not allow files to be externally shared with write permissions (i.e., with a non-Dropbox user). Consequently, in order to allow Bob's client to send file updates to a Dropbox account owned by Alice, the access token of Alice's Dropbox account (Section 3.4) would have to be shared with Bob so Bob's client could impersonate Alice and

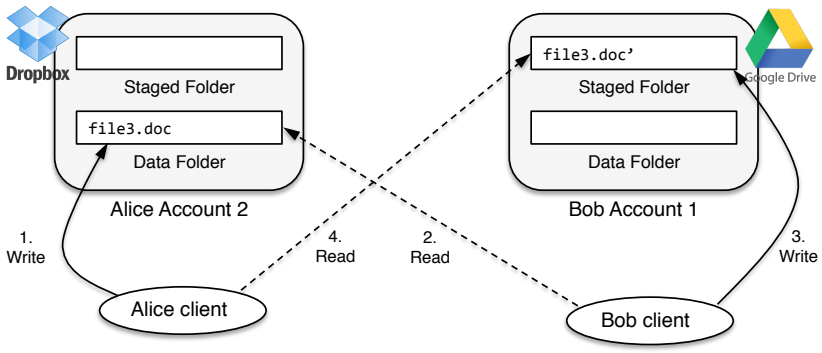

Figure 3.3: Example scenario to illustrate the use of the staging space. Read / write operations are represented in dashed / solid lines. First, Alice client writes the file in her account (1). Then, Bob fetches the file from the file's home account using a read-only URL (2). Bob updates the file and places the file temporarily in the staging space (3). Finally, Alice retrieves the updated file version from Bob's staging space using a read-only URL (4).

then send file updates.This opens up a new avenue for security vulnerabilities. For example, Bob could simply view, fetch, delete or modify any files located on that account or write arbitrarily large amounts of data to Alice's account.

Our goal is to support interoperability between cloud stores of different cloud providers such that files can be shared at a file-level granularity while following the principles stated in Section 3.1, in particular P1 and P2 which preserve isolation between user accounts. To implement this behavior, the key insight is to entirely prevent direct writes on other users' stores. In other words, Bob's client will be able to read Alice's file directly based on a URL to the file's home store, but will be restricted to writing on Bob's own cloud stores. Whenever Bob has pending writes, rather than submitting the new file version to Alice's account via impersonating Alice, the new file version is temporarily placed in a reserved area which we call *staging space*. Staging space is nothing but a dedicated folder in Bob's cloud stores where pending updates to foreign files can be staged (see Figure 3.2). To let Alice access the new file version, Bob's staged file is set to be readable and a corresponding URL given to Alice allows for the file to be pulled from Bob's account. Figure 3.3 illustrates these concepts for a simple scenario. By implementing a *remote-read local-write* policy, staging ensures interoperability and security.

3.7.4 File Homing

With the introduction of file staging, a few negative side effects emerge. In this section, we list these side effects and then describe a technique to counter them: *file homing*.

- **Dangling pointers:** By following principle P2 (see Section 3.1), a file is always expected to be found on its respective home store. The SDS keeps track of a public read-only URL that points to the (encrypted) file and makes this URL available to users with valid access permissions to the file. However, with the introduction of staging the location of the latest version of the file may change. For example, in the scenario depicted in Figure 3.3, after Bob finished updating the file (step 3) the latest file version is found not in the file's home store, i.e., Alice account 2, but in Bob's staging space. As a result, the URL stored in the SDS becomes stale. If a user reads the file from the URL stored in the SDS an old file version is returned. To prevent this behavior, when writing the file in the staging space, Bob's client also updates the SDS with the new URL location of the file. This URL is read-only and points to the (encrypted) file copy located in Bob's staging space, which becomes the file's *staged store*. However, if Bob removes his account from Storekeeper or deletes the staged file directly from the cloud store, the current URL becomes invalid and thus the file inaccessible. This is called a *dangling pointer* situation.

- **Lost updates:** There is a second consequence if Bob removes the cloud store where the pending file is staged. In such an event, not only a dangling pointer situation occurs, but also the updates that were performed by Bob will be lost because the file version that contains such updates was stored in Bob's staging space which is no longer available. We say that file updates were lost. The amount of lost updates can be even more serious. Suppose that a file is shared with more users, for example Claire and Dan. As these users collaborate on the file and submit updates, the file's latest version will be relocated to their respective staging spaces. If one of Bob, Claire, or Dan leaves Storekeeper all the cumulative updates will be lost. Only when Alice performs a write, updates will be reconciled into the file's primary version located in Alice's cloud store.

- **Free riding:** File staging can also lead to situations where a user is unconsciously contributing to increasing the capacity of someone else's aggregate cloud space. For example, suppose that the latest file version is hosted in Bob's staged space. If Alice revokes Bob's

permissions to access the file, Bob could no longer access the file but the file would still be provisioned from Bob's store space. As a result, part of Bob's storage space would be reserved for hosting someone else's files (in this case Alice's) without bringing benefits for Bob since he no longer is co-editing the file and the file is taking up some of his precious storage space in the cloud. Alice would then "free riding" on Bob's storage space.

To overcome these negative side effects, Storekeeper uses a technique which essentially consists of relocating the staged file copy back to the file's home copy; we call this technique *file homing*. In the example depicted in Figure 3.3 this is translated into (a) copying the most recent file version (`file3.doc'`) stored in Bob's staged space to the file's home store, i.e., Alice account 2, and (b) updating the current file URL to point to the primary version stored in the home store. Thus, if Bob's cloud store becomes inaccessible, dangling pointers and lost updates will be avoided. This is because the file has been relocated back to Alice's store: the file's primary replica has been updated to the new version and the file URL updated to point to the file's primary replica. By the same reasoning, if Bob's permissions are revoked, free riding is also prevented because the file is now served from Alice's storage space.

The challenge of implementing file homing is how to push the update file from the staged store back to the home store. File homing should ideally be performed by the time Bob submits his file update. The longer it takes to relocate the staged replica, the higher is the likelihood that the staged store is subtracted from the system, resulting in dangling pointer and lost update. However, keep in mind that the user responsible for producing the staged file version (Bob) has no privileges to write on the file's home store (Alice's account): only Alice (i.e., the file owner) is authorized to perform that operation.

File homing must then be carried out by the file owner (or by some party entrusted by the file owner). To avoid increasing the complexity of the system with distributed notification mechanisms or require the SDS to be entrusted with users' credentials in order to access their accounts, we opted to trigger file homing events at the client endpoint. In other words, Alice's client is responsible for checking whether or not the file is staged and relocate it if necessary. To detect if file homing is required, the SDS maintains an additional flag – *staging flag*. If Bob submits an update, in addition to updating the file version and the file URL in the SDS, the staging flag is also set to 1. Whenever Alice's client needs to check if the file is staged all it has to do is check this flag. It then clears the flag to 0 after finishing the file homing operation.

Given that file homing is not synchronous with Bob's file update operation, chances are that Alice's client triggers file homing *after* Bob's store has been removed from the system. As a result, the file URL in SDS would be invalid (dangling pointer) and the submitted update lost. To address this problem we take three complementary techniques:

- **Increase frequency of file homing events** at the client side so as to reduce the likelihood of such problems. To avoid consuming too many resources at the client, staging checks are piggybacked in periodic interactions with the SDS that the client already performs in two occasions: when refreshing the local cache, and when submitting writes.

- **Have a fallback mechanism** so that if the staged store becomes available the (stale) home version is returned instead. Rather than maintaining a single file URL and version number, the SDS will now maintain two pairs: *home URL - home version number*, and *staged URL - staged version number*. The home pair refers to the primary copy saved in the home store. The staged pair refers to the staged replica maintained in the stage space of another user. When such user submits a write, the SDS updates the staged URL and staged version number. To perform file homing, client of the file owner must copy the file from the staged URL into the home store and set the home version number equal to the stage version number. If the staged store is removed before file homing takes place and a client tries to access the file on the staged URL, an error will be generated. The client then triggers an healing process to fallback to the home version by simply clearing the staging flag. Although this process does not entirely prevent lost updates, it limits the amount of lost information to the updates that were performed since the last file homing event.

- **Perform garbage collection** to claim stale file replicas from users' stores. As a result of staging and file homing, such replicas will likely be left on staging directories, taking up precious storage space. To claim such space, Storekeeper includes a simple garbage collection mechanism which deletes stale file replicas from staging directories. Since only the owner of a given cloud store has the permission to delete such files, garbage collection must be triggered by a Storekeeper client acting on behalf of the cloud store's owner. Periodically, each client reads the list of files from the staging space and queries the SDS to determine whether such files are pending to be homed or not. Files that are not in such condition (i.e., staging bit is 0) are deleted from the staging directory.

3.8 Scalability and Fault Tolerance

In this section, we make some brief considerations about the scalability and fault tolerance of Storekeeper. Regarding the first of these two properties, a potential bottleneck to the scalability of the system can be caused by Storekeeper of a centralized architecture that depends on the SDS server. As a result, the throughput of the system in terms of number of requests served per unit of time will be limited to what a single SDS server can deliver. Similarly, the maximum of users, cloud stores, and files will be bound by the memory and disk available on the server to store the meta-data associated with that information.

As for fault tolerance, the SDS constitutes the most critical component in the system. Since the client endpoints store ephemeral data only and cloud providers make sure that the content of cloud stores is durable, failures that affect the correct behavior of the SDS can affect the availability of the service and durability of the data. To improve availability, it is possible to deploy multiple SDS servers operating in master-slave configuration so that failure of one of the servers can gracefully fallback to another server. To assure the durability of the meta-data maintained persistently by the SDS (e.g., in the case of hardware failures), replication can be implemented using RAID, backups, or both.

3.9 Summary

In this chapter we presented the design of Storekeeper. Storekeeper is a cloud storage aggregated system that provides end-to-end confidentiality of files. It enables seamless integration of multiple cloud storage accounts and sharing files between users. Among the several unique aspects involved in the design of this system, we highlight three. First, as opposed to many existing storage systems, Storekeeper avoids the need to fully re-encrypt user files whenever revocation of read permissions occurs. This is achieved using standard cryptographic techniques. Second, to ensure proper isolation between cloud user accounts, Storekeeper ensures that users can only submit writes to their own cloud accounts. This is achieved by incorporating a staging technique into the design of the system. Third, as a result of staging, it is necessary to provide secure exchange of updates between different cloud accounts. Storekeeper enables such exchange through file homing. The next chapter describes our implementation of this system.

4 Implementation

This chapter presents the implementation of Storekeeper. An introductory section provides an overview of the components of our system and a few basic implementation details (Section 4.1). Then, in Section 4.2, we present more in-depth explanations about the main data structures that constitute the core of the system state, and in Section 4.3 a description of the protocols that implement the most relevant operations supported by Storekeeper, such as adding cloud stores, reading files, writing files, setting up file permissions, etc.

4.1 Implementation Overview

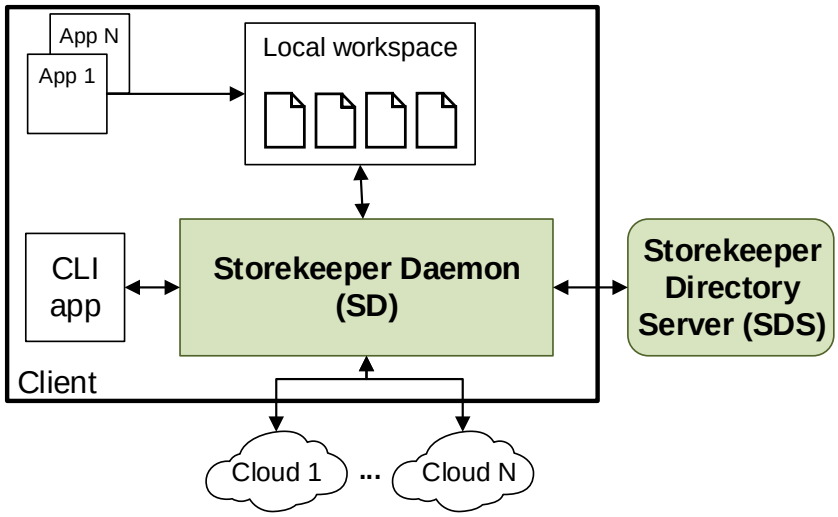

Figure 4.1: Storekeeper components.

Figure 4.1 represents the components of our Storekeeper implementation. The colored boxes highlight the parts that are specific to the system: the *Storekeeper Daemon* (SD) and the *Storekeeper Directory Server* (SDS). The SD is a standalone process running on the user's device and is responsible for the implementation of the client-side logic. Written in Java, the SD implements, on the one hand, an API to a client tool that allows the user to issue management

commands (e.g., add a new account, login to the service, etc.), and, on the other hand, a periodic monitoring service that synchronizes the local workspace with the cloud-backed stores. Although the current client tool provides a command line interface, the API provided by the SD allows for future satellite applications that can provide more friendly user interfaces.

The SDS is also implemented in Java and receives commands from the SD. The communication between SD and SDS takes place over SSL. By using SSL, we ensure that the SDS is properly authenticated, which is important for users to be certain that they connect to an authentic Storekeeper domain provider. SSL also provides integrity- and confidentiality-protected message exchage capability between SD and SDS. Once an SSL channel is established, SD and SDS exchange protocol-specific commands encoded in JSON. At the SDS side, persistence of meta-data is achieved by serializing it into local XML files. The SDS also provides an command line interface that allows the local administrator to manage the users of the system.

As also shown in Figure 4.1, the SD interacts with the cloud stores where user files are located. For this interaction, the SD uses specific libraries provided by cloud services in order to facilitate access to their respective REST APIs. Currently, the SD supports integration with two cloud services: Dropbox (API version 1) and Google Drive (API version 2). To support additional cloud services, the SD is internally designed in a modular fashion, such that adding support for a new cloud service can be done by developing a new module which in turn uses a service-specific library that acts like a proxy to the service.

To perform cryptographic operations we used the Java library provided by the JCA framework (Java Cryptography Architecture). Regarding symmetric encryption, we use AES cipher and generate 256-bit symmetric encryption keys randomly. For asymmetric encryption, we use 1024-bit RSA asymmetric keys randomly generated. SHA1 is used for hashing. Next, we describe the internal data structures of our system.

4.2 Data Structures

In order to implement the design described in the previous chapter, it is necessary to maintain important internal state. Since the state on the client side is ephemeral (i.e., it can be fully reconstructed from the SDS and user-provided authentication information) and the state maintained on the cloud stores is relatively straightforward (i.e., consists of encrypted files placed

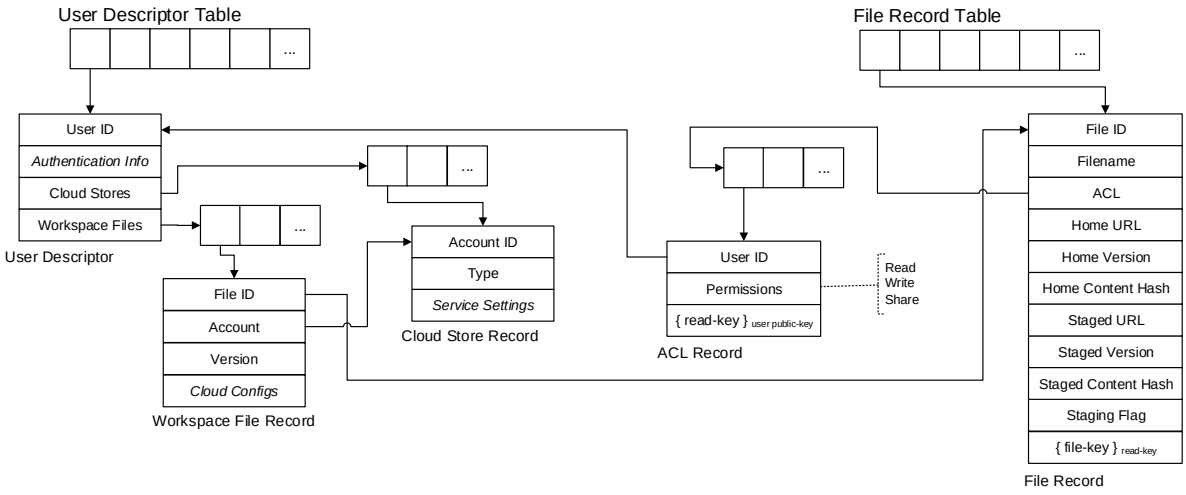

Figure 4.2: Data structures implemented by the SDS for a given Storekeeper domain.

in data and staged folders), we focus our attention primarily on the state maintained by the SDS. To manage this state, the SDS must not only represent this meta-data information, but represent in a way that facilitates efficient processing of user requests. Figure 4.2 represents the data structures of SDS, designed to serve both these purposes. We briefly describe them:

User Descriptor Table (UDT): Contains information about the users registered in the system. Individual user information is stored in the User Descriptor data structure. Implementation-wise, the UDT is implemented as a hash-map keyed by the user ID to speedup user information lookup, which is required in all methods of the SDS service.

User Descriptor (UD): Keeps track of relevant information about a given user, namely: the user ID, authentication info, list of user's cloud stores, and list of workspace files. The user ID (u) consists of a fully qualified name that includes the username appended with the Storekeeper domain name (e.g., `alice@sds.tecnico.ulisboa.pt`). The authentication info field ($auth$) corresponds to the private user key (KU^-) encrypted with the login key (KL), i.e., $\{u, KU^-\}_{KL}$ (see Section 3.4). The list of cloud stores points to data structures that contain information about the cloud stores to be aggregated by the system for that given user. Lastly, the list of workspace files contain information about the files mounted on the user's workspace for that particular Storekeeper domain (e.g., `sds.tecnico.ulisboa.pt`).

Cloud Store Record (CSR): This data structure contains information about a given cloud store. It includes an account ID (which is used for internal reference), the type, and the service

settings. The type field indicates the identity of the cloud service where this store is hosted, e.g., Dropbox, Google Drive, etc. The service settings fields is interpreted according to the type of service. It contains the pieces of information that are required to access the cloud store, in particular its access tokens. Note that, as explained in Section 3.4, access tokens (AT) are encrypted with the login key (KL), e.g., $\{\texttt{alice@sds.tecnico.ulisboa.pt}, AT_{\text{alice@dropbox}}\}_{KL}$.

Workspace File Record (WFR): Contains information about a file that is mounted on the workspace. It includes the file ID (FID), a pointer to the cloud account where the file is hosted (i.e., the home store), and service-specific meta-data about the file. Essentially, the WFR is just a wrapper to the File Record data structure that contains the details of the file. The goal of this data structure is only to increase the efficiency of returning the list of files in the workspace for a particular user.

File Record Table (FRT): Contains detailed information about all the files that have been aggregated by the SDS for that particular Storekeeper domain. It is optimized to be searched based on the file ID. Information about each individual file is contained in a File Record.

File Record (FR): This data structure contains relevant meta-data about a given file, namely: the file ID (FID), the file name (as shown to the user in the workspace), an ACL, the home URL, the home version, the staged URL, the staged version, staging bit, a hash of the file, and the encrypted read key. The semantics of these fields has been covered previously: Section 3.3 covers the FID and file name, Section 3.7.4 discusses fields home URL, home version, staged URL, staged version, and staging bit, and Section 3.7.2 focuses on ACL and encrypted read key.

4.3 Protocols

This section presents the protocols responsible for the implementation of Storekeeper operations and the role that these data structures play in such protocols. First, we describe the procedure involved in the initialization of the system. Then, we focus on the protocols triggered by the most important file operations issued by clients: read, write, delete, share, and revoke.

4.3.1 System Initialization

The initialization of the system involves the execution of a set of procedures by both the SDS administrator and clients. From the perspective of the SDS administrator, to bootstrap the system it is necessary to perform a sequence of steps: (1) install the SDS software on a server, (2) define a Storekeeper domain name for the service (e.g., `sds.tecnico.ulisboa.pt`), (3) generate an SSL certificate to enable server authentication in the communication between the SD and the SDS, and (4) register new usernames in the service (e.g., `alice` and `bob`), which will update the UDT with new UD data structures, one per added user. At this point the service becomes available for use. Clients can then start using the system. Their typical operations are listed next:

1. **Client software installation:** Install the SD daemon on the device on which the Storekeeper workspace will be mounted.

2. **User sign-up:** Using the username defined by the SDS administrator, the user signs up for the first time and defines his password. Based on the username and password, the client generates a 256-bit AES key using Hmac-SHA1. This key constitutes the login key KL. The client also generates the user key, an RSA 1024 bit keypair. Based on both the login key and the user key, the client generates the encrypted user key blob which will be assigned to the *auth* field of the user's UD data structure. In the blob generation, the password is hashed with Blowfish block cipher.

3. **User login:** The user can now log into the system by providing username and password. Based on these credentials the client re-generates the login key. Then retrieves the *auth* field from the SDS and decrypts it with the login key, yielding the private part of the user key, which will be maintained in memory by the client for future use.

4. **Add a cloud store:** The user can now add a new cloud store to his Storekeeper account. The user selects the type of cloud (e.g., Dropbox or Google Drive), is redirected to a browser to insert her user credentials on the cloud service's web site, so that Storekeeper client can access the account. The SD then obtains the access token (AT) for that account and additional information that identifies that account (e.g., the associated email address), and submits this information to the SDS, which creates a new CSR data structure for this account. Note that AT is encrypted with the login key before being sent to the SDS.

Once the user has added a cloud store to the system, it is possible to perform file management operations. Next, we describe the protocols that implement the most relevant operations: read, write, delete, share, and revoke. (Note that, share and revoke are in practice implemented by the same user command `chperms`, and create is also implemented by the write operation.)

4.3.2 Read Protocol

The read protocol is triggered periodically by the Storekeeper Daemon. Its function is to update a local file copy with the most recent version available on the cloud. The major steps that are performed by this protocol are:

1. **Verify permissions:** Check if the user has read permissions by consulting the ACL field of the file's FR data structure. Abort if does not have read permission.

2. **Test if staged:** Check check if the file is staged. If not, before proceed, activate the file homing event, then proceed.

3. **Check existence:** See if a file copy already exists in the local cache. If not, download the file, decrypt it with the respective readers key, and finish, otherwise proceed.

4. **Test freshness:** Check if the local file version is lower than the home version number. If not, then the local file is recent, otherwise, download and decrypt the new file version.

4.3.3 Write Protocol

Just like the read protocol, the write protocol is also triggered periodically by the SD in order to send local updates to cloud-backed stored. The write operation performs the steps:

1. **Encrypt file:** Generate a new file key and encrypt the file using that file key.

2. **Create file if needed:** If the file does not exist yet, the user is the file owner. Thus, create a FR data structure for it on the file record table, assign the file to a cloud store, upload the file to the cloud store, and finish. (This is possible because the file owner has the access token to the cloud store.) If the file exists, proceed.

3. **Verify and resolve conflicts:** Check for writing conflicts in the file. If conflicts exist, them the file must be put into quarantine and the operation aborted. Otherwise, proceed.

4. **Write as grantee:** Check if the user is owner or a grantee. If grantee, it is necessary to check the write permissions. If the user does not have write permissions, the operation is aborted, otherwise proceed by writing the file in a staging directory, update the SDS data structures, and conclude. If the user is owner, proceed.

5. **Write as owner:** If owner, upload the file to the file's home store, and update the SDS data structures accordingly. Done.

4.3.4 Delete Protocol

The delete protocol is triggered by the SD whenever it detects that a file was deleted from the local cache. The steps executed are:

1. **Check permissions:** The owner can proceed, otherwise it is necessary to guarantee that the user has writing privileges. If not, the operation is aborted.

2. **File is staged:** If the user is not the owner and the file is locally staged, the staged version is set to -1 so as to tell that the file must be deleted. When file homing occurs, -1 tells the owner's client to delete the file from the home store and update the SDS data structures accordingly.

3. **User is owner:** If the user is the owner, the client simply deletes the file from the home store and updates the SDS data structures accordingly. This is performed no matter whether the file is staged or not.

4.3.5 Share Protocol

The share protocol is very simple. Omitting some steps that are common to the previous protocols, namely triggering the file homing procedure, the steps executed by this protocol are:

1. **Obtain grantee key:** Download the public part of the user key owned by the grantee.

2. **Encrypt reader key:** Encrypt the reader key with the public part of the user key.

3. **Update the ACL:** Send the resulting key and the permission to be assigned (R, W, or S) to the respective ACL in the SDS.

4.3.6 Revoke Protocol

Just like the share protocol, after triggering the file homing procedure, the revoke protocol performs a simple sequence of steps:

1. **Generate reader key:** Generate a new reader key that will replace the current one.
2. **Reencrypt the reader key:** For each user in the new reader set (which excludes the revoked user), re-encrypt the new reader key with the public part of the reader's user key.
3. **Update the ACL:** Remove the entry of the ACL corresponding to the revoked user and send the new encrypted reader keys to the SDS.

4.4 Summary

In this chapter, we have been enlightened about the implementation details of Storekeeper. The current implementation was built in Java and supports the aggregation of popular cloud services, namely Dropbox and Google Drive. The internal architecture of our system is modular, enabling future extensions to provide support for additional cloud services. As far as the internals of the system is concerned, we have covered both its data structures and protocols. Given that the SDS performs a central role in maintaining the meta-data of the systems, we focused primarily in the SDS data structures, which were conceived not only to maintain all necessary information for SDS's proper function but also to ensure efficient operation of the SDS. These data structures are accessed and manipulated in the course of the execution of a set of protocols implemented between clients and SDS as triggered by user operations, such as read, write, share, etc. In the next chapter we present an experimental evaluation of our system.

5 Evaluation

This chapter introduces the experimental evaluation made to Storekeeper which tries to answer one main question: *What is the additional performance cost in using our system over storage clouds operations.* The answer to such question is obtained by quantifying the impact over the single use of storage clouds. Section 5.2 presents an overview on Storekeeper's overall performance regarding each supported operation, the performance of each operation is then detailed in the following sections. In the end of this chapter there is a summary of the evaluation, but first the experimental platform and used methodology are described.

5.1 Methodology

The evaluation of Storekeeper focuses on latency measurements that were obtained running several micro-benchmarks. Micro-benchmarks differ from benchmarks: this evaluation was not done by running a number of standard tests and trial against it, it was done by comparing this system's performance against the performance of cloud providers' API. Each micro-benchmark measures an operation provided by the Storekeeper API, such operations are sharing, access revocation, write, read and delete. The operations of Storekeeper API have different parameters of which their performance depend, as file size, cloud storage type and number of users. The file size parameter was used with different data unit sizes of 1KB, 10KB, 100KB, 1MB, 10MB and 100MB to be similar to the common file sizes found in cloud storages. The cloud storage parameter was filled with two different types, Google Drive and Dropbox, and the number of users parameter varied between 1 and 100 users. The time to take measurements was verified and is negligible.

The micro-benchmarks were executed individually for 100 times and the obtained results were analyzed with its mean time and correspondent standard deviation. All experiments have been performed using a 1-core Intel Xeon Family 2.5 GHz machine with 1GiB memory and 8

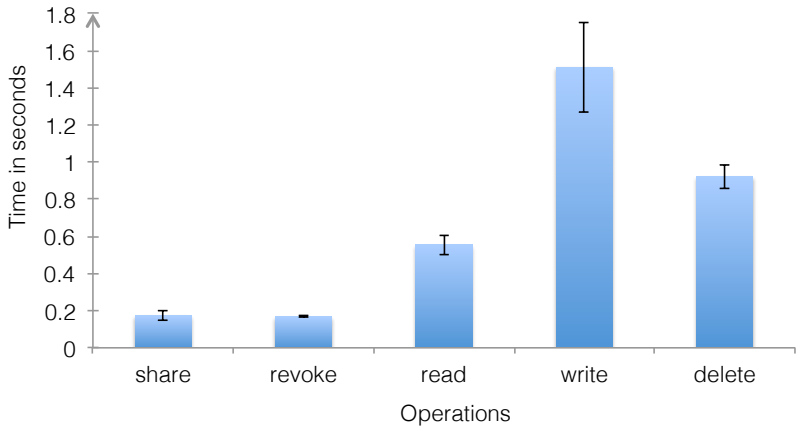

Figure 5.1: Lowest latency per operation and corresponding standard deviation.

GiB SSD, running Ubuntu Server 14.04 LTS. For all experiments the SDS was deployed in the same machine as the micro-benchmark. These experiments took place during 1 month but the values reported correspond to measurements done between September 15 and 20, 2015.

5.2 Overall Performance

In order to understand the performance of Storekeeper, the minimum possible latency for each operation is shown in Figure 5.1, varying between 0,17 and 1,51 seconds. The minimum possible latency corresponds to the value where the operation performs better regarding defined parameters. As stated in the previous section, the operations of Storekeeper API have different parameters and, according to that, their latency varies. Regarding the parameters impact on a operation's latency, an operation can be constant or variable. The operations with the lowest minimum latencies are the share and revoke operations, both with 0,17 seconds of latency, followed by the read operation with a latency of 0,55 seconds and the delete operation with 0,93 seconds. Lastly is the write operation with a latency of 1,51 seconds, which is the one with the highest minimum latency. The share operation is the only with constant latency because its parameters do not effect the operation's latency as detailed in Section 5.3, however the latency of the revoke operation is variable according to the number of users that remain in the file's ACL when revoking access to one user. In a situation where a file is shared with 100 users, the revoke latency increases only 83% in comparison to the represented latency that corresponds to a file that is available only to its owner (meaning a ACL of 1 user), this evolution is detailed

in Section 5.4. The latency of the remaining operations of read, write and delete depend on two parameters, file size and cloud storage type. For the three, the minimum latency values are achieved with a file of 1KB. The read and delete operations achieve the minimum latency along with GoogleDrive cloud storage, discussed in detail in Sections 5.5 and 5.7 respectively. And finally, the operation with the highest minimum latency achieves its minimum latency while performing a write-create with Dropbox cloud storage. The write specifics are evaluated in detail at Section 5.6.

5.3 Performance of Share Operation

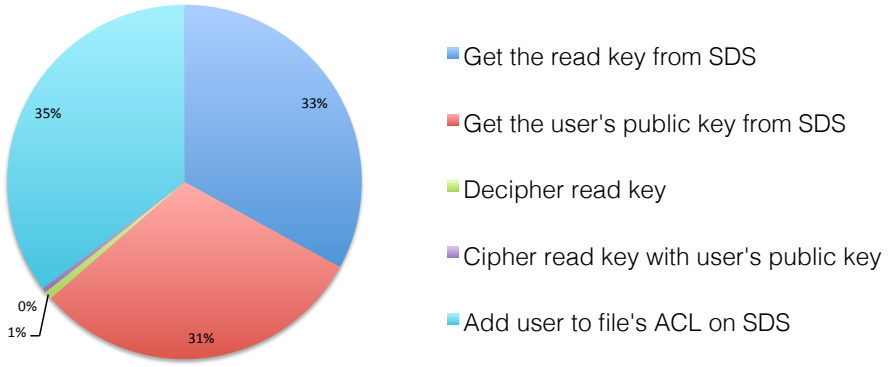

Figure 5.2: Percentage of time spent in each step to execute a share operation.

In order to fulfill a share operation, there are five steps to be performed. In a situation where Bob wants to share a file with Alice: (1) get the file's read-key from SDS, (2) get Alice's public key from the SDS, (3) decipher that read-key with Bob's private key, (4) cipher that read-key with Alice's public key, and (5) add Alice to the file's ACL on SDS. Such steps are represented in Figure 5.2 with the relative time spent in each step. This operation shares a file with one user at a time and it involves only metadata, so the latency depends solely on network throughput to the SDS and SDS processing times. Based on the results presented in the figure, several points can be highlighted. First, only 1,2% of the whole execution is spent on cryptographic operations. This is explained by the fact that the symmetric decipher deciphers only 256 bits of content and the asymmetric cipher ciphers the same content (content is a 256-bit AES key), coupled with the fact that the crypto library is cached on system initialization. Second, the remaining 98,8% of execution time are spent on lookups to the SDS. Such lookups are to obtain

the file's read key, then to get the public key of the user who will get the file and finally to add that user to the file's ACL.

Steps	Values
Get the file's read-key from SDS	56,9 ± 7,4
Get the user's public key from SDS	52,8 ± 7,2
Decipher file's read-key	1,3 ± 0,6
Cipher file's read-key with user's public key	0,8 ± 1,1
Add user to file's ACL on SDS	60,8 ± 17,1
Total	172,8 ± 27,1

Table 5.1: Average time (in milliseconds) spent in each step to perform a share operation, by order of execution.

Complementing the discussed figure, in Table 5.1 is presented the absolute time values by which were inferred the percentages for the figure. Based on the values in the table, it is confirmed that the share operation is a constant operation, as stated in Section 5.2, that does not depend on parameters such as file size or cloud storage type, and costs less than 0,2 seconds.

5.4 Performance of Revoke Operation

Steps	Values	%
Get file-key and file's read-key from SDS	41,8 ± 0,6	24,9
Remove user from file's ACL on SDS	42,1 ± 0,7	25,1
Get users' public keys on file's ACL from SDS	41,6 ± 0,5	24,8
Decipher file's read-key	1,2 ± 0,4	0,7
Decipher file-key	0,1 ± 0,3	0,1
Create new read-key	0,1 ± 0,3	0,1
Cipher file-key with new read-key	0,3 ± 0,5	0,2
Cipher new read-key with each public key	0,2 ± 0,4	0,1
Update file's ACL on SDS	40,5 ± 0,8	24,1
Total	167,9 ± 2,2	100,0

Table 5.2: Execution times in milliseconds of a revoke operation that results in a ACL with 1 user.

As stated before in Section 5.2, the minimum latency of the revoke operation happens when the file's ACL stays with only one user upon revoking one other user. To fulfill this operation eight steps have to be performed: (1) get file's file-key and read-key from SDS, (2) remove user from file's ACL on SDS, (3) get the public keys of all users that remain in the ACL from SDS, (4) decipher the read-key, (5) then decipher the file-key with the read-key, (6) create a new read-key, (7) cipher the file-key with the new read-key, (8) cipher the new read-key for with the previously retrieved public keys, (9) update the file's read-key field and ACL on SDS. Such steps

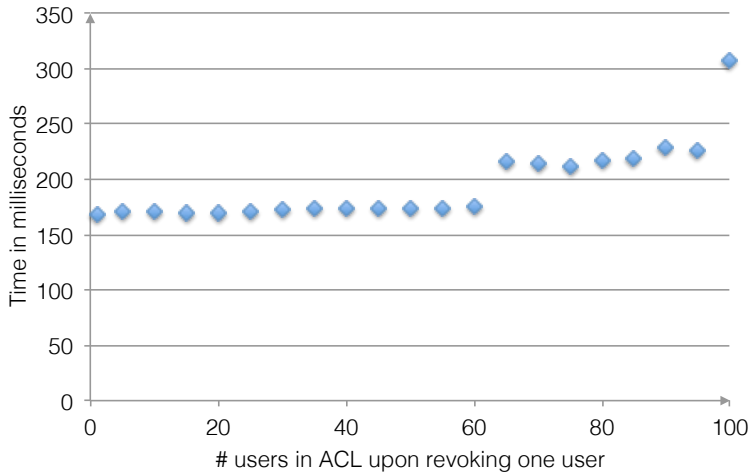

Figure 5.3: Evolution of revoke operation's latency with increasing ACL size.

are detailed in Table 5.2 along with the absolute and relative times per step when the file's ACL has one user. Based on the results present in the table, the main bulk of time is spent with SDS lookups, as it was discussed previously for the share operation. The three SDS lookups take 98,9% of the total time. In this execution there are four cryptographic operations, twice as much as on the share operation, and still such operations are close to negligible.

Upon revoking one user, the remaining number of users in a file's ACL can have any value, since the file can be shared with a number of users. Different micro-benchmarks were executed to study the performance's evolution on the revoke operation with an ACL up to 100 users. The results of this experiment are present in Figure 5.3 with the latency's evolution for different ACL sizes, from 1 to 100 remaining users. In this figure one can spot two table lands where the latency is more or less constant that result in two discontinuities, one around 60 and 65 users and the other around 95 and 100 users.

Looking in detail to the experiments, the discussed discontinuities occur precisely between 63 and 64 users and then between 99 and 100 users. In Table 5.3 is detailed the execution times per step on those discontinuities to understand what caused them. Starting with discontinuity 1, based on the values there is a difference of 40 milliseconds between ACL's size, that corresponds to an increase of 23%. This increase is caused by the third step of obtaining the public keys of all users that remain in the ACL from SDS, such step increases 93% varying from 41,9 to 81 milliseconds. The increase in the third step appears to be due the add of one more public key in the SDS's response that forces the SSL socket to perform another round-trip to the SDS.

Together with the results from Table 5.2, the first table land represents a very small spread of 4%. Looking at discontinuity 2, based on the values there is a difference of 64 milliseconds between the ACLs of 99 and 100 users, which corresponds to an increase of 26%. This increase results of two points, first by an increase of 119% while ciphering the new read-key with each retrieved public key, and second by an increase of 37% in sending the new read-key and updated ACL to the SDS. The second table land represents a spread of 13%, bigger than the first table land.

	Discontinuity 1		**Discontinuity 2**	
Steps	ACL = 63	ACL = 64	ACL = 99	ACL = 100
Get file-key and file's read-key from SDS	41,6	41,4	43,3	49,9
Remove user from file's ACL on SDS	42,7	42,7	48,0	53,5
Get users' public keys on file's ACL from SDS	41,9	81,0	78,9	82,4
Decipher file's read-key	1,1	1,0	1,2	2,2
Decipher file-key	0,0	0,0	0,1	1,0
Create new read-key	0,1	0,0	0,0	0,0
Cipher file-key with new read-key	0,0	0,1	0,0	0,3
Cipher new read-key with each public key	6,1	6,0	23,9	52,3
Update file's ACL on SDS	41,9	42,8	47,8	65,6
Total	175,4	215,0	243,2	307,2

Table 5.3: Performance discontinuities of the revoke operation. Detailed execution times per step, in milliseconds. Discontinuity 1 occurs between ACLs with 63 and 64 users and discontinuity 2 occurs between ACLs with 99 and 100 users.

According to the discussed results, the main bulk of time of the operation is on SDS accesses, similarly to the share operation. However it varies with the number of users remaining in the file's ACL, which results in increased latency when getting the public keys of all remaining users for more than 63 and when "re-ciphering" the read-key for more than 99 users. The "re-ciphering" step singularly increases from 0,2 milliseconds with 1 user to 52,3 milliseconds with 100 users, although this is an abysmal rise by percentage, it is not a bottleneck over the whole operation, unlike similar systems of the related work.

5.5 Performance of Read

The latency of the read operation has two variables, the size of the file and the cloud storage type. As stated before in Section 5.2 the minimum latency for this operation happens with a file of 1KB stored in Google Drive. To fulfill the operation five steps are performed: (1) get file's metadata from SDS, (2) decipher file's read-key, (3) decipher file-key with the deciphered

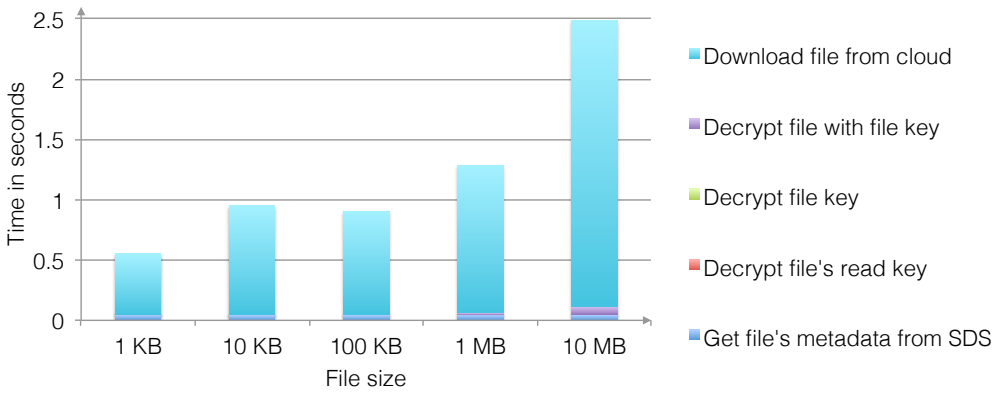

Figure 5.4: Time spent per step while reading different file sizes stored in GoogleDrive.

read-key, (4) download the file, and (5) decipher the file. These steps are represented in Figure 5.4, following the minimum latency case for increasing file sizes. Note that files greater than 25MB cannot be directly read from Google Drive using the interface herein considered. As such, these are herein not depicted. Solely based in the figure one can infer three points. First, that there are three steps that are constant independently of file size: the SDS lookup and the key decipherings (one asymmetric and other symmetric). For all file sizes, these three steps take 50 milliseconds but its weight decreases with increasing file sizes, for a 1KB file it represents 9% and for a 10MB file it represents 2% of the whole operation. Second, the latency of the file deciphering step increases with increasing file sizes and does not depend on cloud storage type. Deciphering a 1KB file takes 0,3 milliseconds, while deciphering a 10MB file takes 67,8 milliseconds and on the whole operation this has a slowly increased weight, for a 1KB file it represents 0,1% and for a 10MB file it represents 2,7% of the whole operation. And third, the main bulk of time is spent downloading the file which increases with increasing file sizes and its latency depends on the cloud storage type. For the illustrated case that corresponds to a Google Drive cloud storage type, the latency to download a file starts at 0,50 seconds for a 1KB file, then stabilizes for 10KB and 100KB files very closely between 0,85 and 0,90 seconds, afterwards increases for 1,2 seconds on 1MB file and at last takes 2,4 seconds to download a 10MB file. With such latency values it is sure that the cost per MB becomes more efficient as file's size increase.

The discussed case represents reads when a file is stored on a Google Drive cloud storage. When switching the cloud storage to Dropbox, 4 out of 5 steps keep the same latency values exactly since those do not depend on the cloud storage, only the download step might change

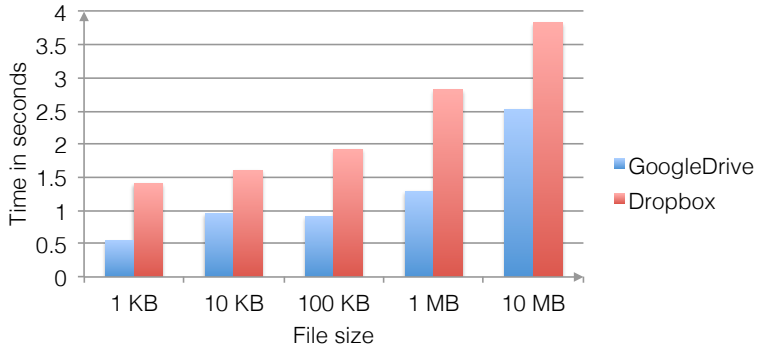

Figure 5.5: Reading comparison between cloud storage types.

its latency. The same experiments were performed for Dropbox cloud storage type and are compared with Google Drive in Figure 5.5. From the smallest data unit size, with Dropbox the latency of the operation is bigger than with Google Drive, as seen in the figure where the difference between each pair of columns represent how much more costly it is to use Dropbox over GoogleDrive. The whole operation takes 170% more time when reading a 1KB file from Dropbox than GoogleDrive, then 68% more time for a 10KB file and 112% for a 100KB file, this difference between reading 10KB and 100KB files is explained by the stabilization on Google Drive that is not accompanied by Dropbox which keeps increasing its latency, this is justified by the fact that for a 1MB file it takes 119% more over GoogleDrive. And finally Dropbox starts to get closer with GoogleDrive when reading a 10MB file that takes 52% more time for Dropbox. Regardless of the storage cloud type, in comparison to the previously discussed operations, accesses to the SDS is no longer the main bulk of the latency of the operation and represents a very small percentage almost negligable.

5.6 Performance of Write

Such as the read operation, the latency of the write operation also has the same two variables, the size of the file and the cloud storage type. Moreover, the write operation is divided into create file and update file. As stated before in Section 5.2 the minimum possible latency for the write operation corresponds to the creation of a 1KB file stored in GoogleDrive. To fulfill the operation, seven steps are performed: (1) get cloud credentials from SDS, (2) create file-key and read-key, cipher file-key with read-key, (3) cipher read-key with user's public-key, (4) cipher file with file-key, (5) connect to cloud, (6) write file to cloud, and (7) add new file entry on SDS.

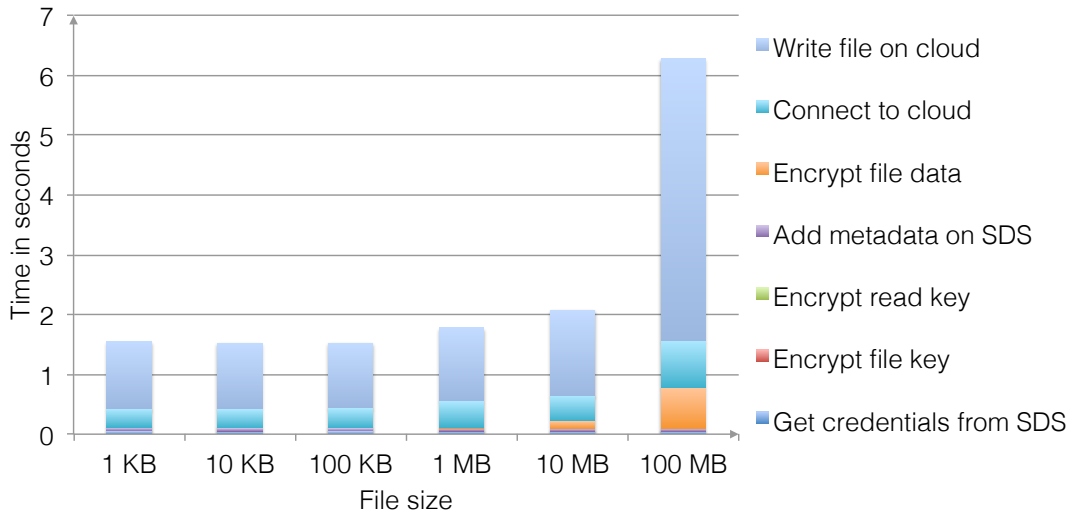

Figure 5.6: Performance of file creation. Evolution of time spent in each step for different file sizes using GoogleDrive.

These steps are represented in Figure 5.6, following the minimum latency case for increasing file sizes. Based in the figure there are three highlights. First, such as the read operation, that there are steps which are constant independently of file size: two SDS accesses, two key ciphers and connecting to the cloud. Such steps take on average 0,47 seconds for all file sizes, where 26,8% corresponds to SDS accesses, 0,2% for the key cipherings and 73% to the cloud connection. Since it is constant, the weight of such steps decreases with increasing file sizes. Second, the latency of file ciphering increases with increasing file size and does not depend on cloud storage type. Ciphering a 1KB file takes 1,3 milliseconds, while ciphering a 100MB file takes 680 milliseconds and on the whole operation this has an increased weight, for a 1KB file it represents 0,1% and for a 100MB file it represents 11% of the whole operation. Comparing with file deciphering latencies presented on the read operation, deciphering takes twice the time of ciphering. And third, the main bulk of time is spent on file writing to the cloud which increases with increasing file size and its latency depends on the cloud storage type. For the illustrated case that corresponds to a Google Drive cloud storage type, the latency to upload a file is stable for files until 100KB for around 1,1 seconds, then increases for 1,2 seconds on a 1MB file and 1,4 seconds for a 10MB file, at last it takes about four times more for a 100MB file with 4,8 seconds. With such latency values, the cost per MB becomes more efficient as file's size increase.

The discussed case represents a file creation, in addition to creating files, there is updating files which is pretty much the same as creating files, except that there is one more step in the

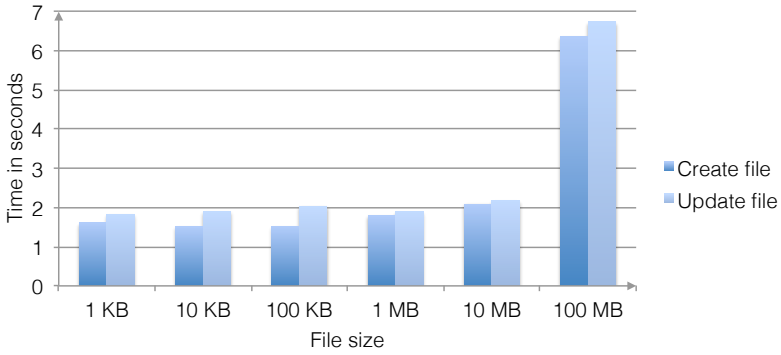

Figure 5.7: Performance comparison between file creation and file update for different file sizes, using GoogleDrive.

protocol which predates all steps of create. To accomplish an update operation there is one more access to the SDS for lookup the file's metadata to obtain the file's read-key and the most recent version number. Figure 5.7 presents a side-by-side comparison between the operations of create and update with increasing file sizes, writing in GoogleDrive. For example, a file of 1MB takes 1,8 seconds to create while it takes 1,9 seconds to update, the difference between them correspond to the additional step required for the update operation.

Either for create or update operations, the latency of their steps do not vary in regards of which cloud storage type the file is being written, except for the steps of cloud connection and cloud writing. When switching cloud storage type, only such points might change its latency. The last two figures discussed represent experiments writing on Google Drive cloud storage, the same experiments were performed for Dropbox. Since file updates are more frequent that file creations, Figure 5.8 presents the differences between cloud storage types for file updates. Based on the figure, both cloud types have similar performance until 1MB files, however Dropbox is on average 13% better than Google Drive until 100KB files and then Google Drive starts getting better from 1MB, in this case 18% better. But, for larger files, it is clear that GoogleDrive is more efficient than Dropbox with an abysmal difference, for example a 100MB file it takes 10 times more with Dropbox. It is noteworthy to take into consideration that Dropbox takes less time to connect than GoogleDrive, which explains why Dropbox has better performance for files until 100KB. This connection time is constant regardless of operation, having a mean time of 223 milliseconds for Dropbox against 313 milliseconds for Google Drive. Such connection times do not matter for the read operation discussed in Section 5.5 since files are downloaded by URL and not using its cloud API.

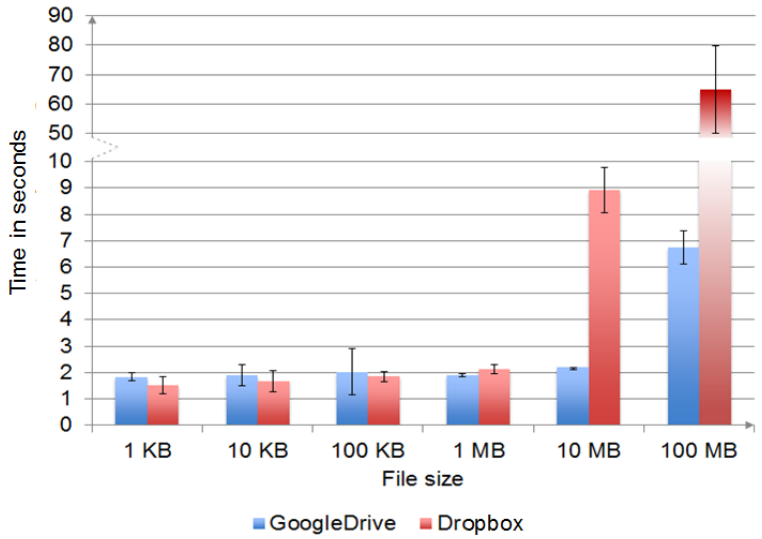

Figure 5.8: File update comparison between different cloud storage types.

Impact of encryption to write efficiency. Similarly to Storekeeper, the different cloud storage types also separate file writes between create or update. If there is such division, probably clouds implement mechanisms to optimize updates, in other words, when updating a file the cloud SDK would send only the binary-diff parts of a file's content. Storekeeper cannot enjoy such optimizations because in each write it performs a full file encryption with a different file-key, making the ciphered content completely different between versions. Even though Storekeeper cannot enjoy this possible optimization, it is relevant to analyze the impact of Storekeeper over cloud's update operation to better determine the Storekeeper's overhead. To test this possible optimization, the following experiments were performed using simply the cloud's API: (1) create a file and send to cloud, (2) change locally the first byte of that file's content and

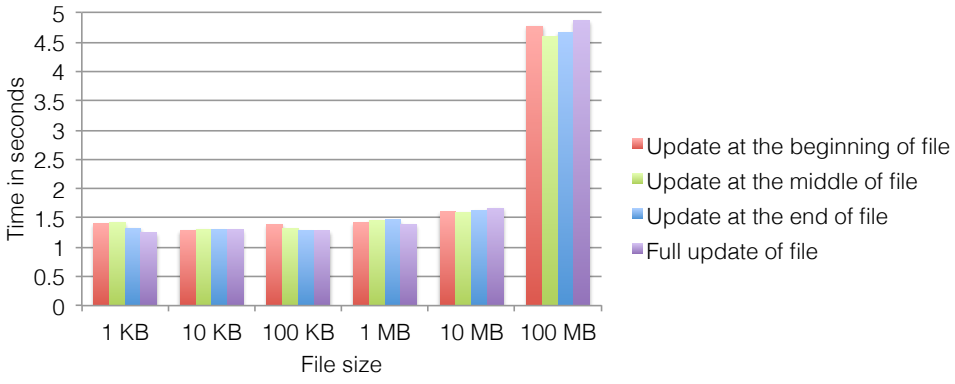

Figure 5.9: Google Drive performance for different content updates for increasing file sizes.

send to cloud, performing an update, (3) same as before but changing the byte on the middle of the file's content, (4) same as before but changing the last byte of the file's content, and (5) same as before but changing all bytes of the file's content. These experiments were performed for both cloud storage types and it was concluded that their behaviour is very similar though with different values. In Figure 5.9 is a comparison between those different experiments on GoogleDrive, with increasing file size, only for updates. Based on the results, two points are highlighted. First, there is no pattern proving that a full update takes much more time than the other updates, there are even cases where a full update is better, for example with a 1MB file. Second, GoogleDrive does not implement the discussed write optimizations, thus using encryption does not have a big impact on write efficiency. The same is valid for Dropbox.

5.7 Performance of Delete

Steps	**Values**		**%**
Get cloud credentials from SDS	52,7	± 3,9	5,7
Remove file's metadata from SDS	61,2	± 8,0	6,6
Delete file locally	2,0	± 1,0	0,2
Connect to cloud	311,8	± 20,2	33,8
Delete file from cloud	494,9	± 61,2	53,6
Total	922,6	± 64,3	100

Table 5.4: Execution times in milliseconds of a delete operation for a 1KB file stored on Google Drive.

Such as the read and write operations, the latency of the delete operation also has the same two variables, the size of the file and the cloud storage type. As stated before in Section 5.2 the minimum possible latency for the delete operation corresponds to the deletion of a 1KB file stored in GoogleDrive. To fulfill the operation five steps are performed: (1) get cloud credentials from SDS of the cloud where the file is stored, (2) remove file entry from SDS, (3) delete file locally from the user's workspace, (4) connect to the cloud, and (5) delete file on cloud. These steps are detailed in Table 5.4 for the minimum latency case with the values for each step along with its weight on the whole operation. Based on the results, the main bulk of time is spent on deleting the file on cloud. In accordance to previous discussions, the accesses to the SDS are constant regardless of file size and cloud type, and in this case even for the smallest data unit it only represents 12,3% of the whole execution.

The discussed case represents a delete when a 1KB file is stored on a Google Drive cloud

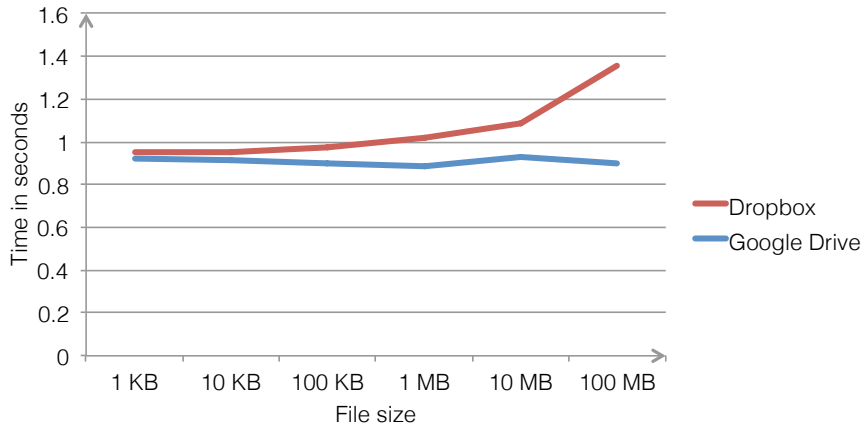

Figure 5.10: Delete comparison between cloud storage types with increasing file sizes.

storage. When performing the experiment for different file sizes and different cloud storage type, the latency of the operation has same changes. This experiments are represented in Figure 5.10 that presents a comparison between cloud storage types for increasing file sizes. Based on the results, there are two different patterns according to the cloud type. First, for GoogleDrive, the latency is constant regardless of file size, on average it takes 0,91 seconds. Second, for Dropbox, latency increases with increasing file sizes, initially is constant until 10KB, then increases 13% until 10MB and then has a steeper increase of 20% forward. In short, the difference of performance between both cloud storages is only considerable for files over 10MB.

5.8 Summary

In this chapter was introduced the experimental evaluation made to Storekeeper and its results. First was presented the overall performance of the system regarding the five operations offered by Storekeeper. Next, in each section, each operation was evaluated in more detail with variable parameters of file size, cloud storage type and number of users sharing a file. The chapter was organized like that for the purpose of answering the question introduced at the beginning: *What is the additional performance cost in using our system over storage clouds operations?* Storage clouds are not involved in operations of share and revoke, their cost correspond solely to Storekeeper performance. Sharing a file is a constant operation that takes 0,17 seconds to complete, such operation does not take into account the time it takes to read the file by the user receiving the file. Revoking one user from accessing a given file takes 0,16 to 0,31

seconds, from the case where the file's owner shares a file with Alice and revokes access to Alice, to the case where a file is shared between 101 users and one of them is revoked. Such performance is better than the existent in related work. The answer to the question is obtained over the operations of read, write and delete considering a range of file sizes between 1KB and 100MB and also considering both cloud storages Google Drive and Dropbox. Starting with the read operation, it implies an overhead between 3% and 10%, the lowest overheads occur when reading from Dropbox since it takes longer to download data thus decreasing the weight of Storekeeper's system. The overhead when reading from Dropbox is constant between 3% and 4%, when reading from Google Drive for the smallest data unit has 10% of overhead and then for increasing file sizes keeps constant between 5% and 6%. Such constant overheads are caused by the increasing times of file deciphering. The write operation implies an overhead between 1% and 13%, the lowest overheads occur when writing to Dropbox since it takes much longer to upload data thus decreasing the weight of Storekeeper's systems. When writing to Google Drive the overhead is more or less constant (varying between 7% and 12%) regardless the file size, because the increasing upload time is accompanied by the increasing file ciphering time. The delete operation implies an overhead between 8% and 13%, such operation is more or less constant among the storage cloud, in other words, when deleting from Google Drive the overhead varies between 11% and 13% and when deleting from Dropbox it varies between 8% and 10%. Overall, Storekeeper has an overhead of 1% to 13%, lower with bigger files and higher with smaller files.

The next chapter finishes this thesis by presenting the conclusions regarding the work developed and also introduces some directions in terms of future work.

6 Conclusions and Future Work

6.1 Conclusions

This thesis presented the design, implementation, and evaluation of Storekeeper, a privacy-preserving cloud aggregation service that enables file sharing on multi-user multi-cloud storage platforms while preserving data confidentiality from cloud providers and cloud aggregator services. Storekeeper is motivated by the fact that cloud storage services are currently a commodity that allows users to store data persistently, access the data from everywhere, and share it with friends or co-workers. However, due to the proliferation of cloud storage accounts and lack of interoperability between cloud services, managing and sharing cloud-hosted files is a nightmare for many users. To address this problem, specialized cloud aggregator systems emerged that provide users a global view of all files in their accounts and enable file sharing between users from different clouds. Such systems, however, have limited security: not only they fail to provide end-to-end privacy from cloud providers, but they require users to grant full access privileges to individual cloud storage accounts. Storekeeper aims to fill this gap.

In developing this work, a major finding was that building a privacy-preserving cloud aggregation service entails addressing many technical challenges that arise mostly from the heterogeneity and lack of interoperability between cloud services. In fact, to build Storekeeper it was necessary to devise adequate (1) file naming scheme, (2) user credentials, (3) consistency semantics, (4) access permissions model and (5) file operations protocols. Storekeeper contributes to the cloud storage landscape with an original design that provides solution to all these issues.

6.2 Future Work

For future work, there are several avenues that can be pursued in order to improve Storekeeper. We briefly discuss some possible directions:

1. Scale the centralized architecture: As mentioned in Section 3.8, the scalability of the current architecture is limited by relying on a single server. The throughput and the storage capacity of the service is limited by the amount of hardware resources available on that server, namely network, processing power, memory, and disk. While for smaller organizations a single server may suffice, for larger organizations the scalability properties of the current architecture may be clearly limited, not only because of the higher amount of user requests and meta-data, but also because larger organizations are typically located in multiple sites and run under different administration domains. To accommodate to the reality of larger organizations, the current centralized architecture of the SDS service may need to be re-thought by supporting a federated network of SDS services possibly deployed in different locations and managed by independent administrators. Such design introduces new technical challenges that need to be addressed, such as: how to split file meta-data across such a distributed system of SDS servers, how to handle migration of users across sites, how to handle network partitions, etc.

2. Devise a fully decentralized architecture: In addition to explore a system design where the SDS server is decentralized, decentralization can be taken to another level by designing a fully decentralized cloud aggregator service. In such an architecture no SDS server would be required at all. Instead, its logic would need to be implemented in a distributed fashion entirely by the Storekeeper clients in a peer-to-peer fashion. A peer-to-peer architecture allows for Storekeeper to be used without requiring a dedicated infrastructure to work, which is convenient for usability purposes. However, without having a semi-trusted component responsible for securely preserving file meta-data, clients themselves must cooperate among themselves in order to provide that service. To make it work, it will be necessary to study existing peer-to-peer systems and adapt some of their techniques to make Storekeeper work under such environment. For example, permission enforcement and revocation will be particularly challenging in this setting.

3. Provide stronger security properties: Lastly, orthogonally to the proposals for system decentralization both in terms of scaling the SDS service or suppressing the SDS service, one may also follow a path in terms of providing stronger security properties. The current Storekeeper design is focused only on providing end-to-end data confidentiality, but other important properties have been left out from the scope of this work, namely data integrity and service availability. Enhancing Storekeeper to provide such properties constitutes a promising area for future research.

References

[1] Cloudfogger. http://www.cloudfogger.com/, November 2014.

[2] Cloudfuze. https://www.cloudfuze.com/, November 2014.

[3] Cloudhq. https://www.cloudhq.net/, November 2014.

[4] Cloudkafé. https://www.cloudkafe.com/, November 2014.

[5] Odrive. http://www.odrive.com/, November 2014.

[6] Otixo. http://otixo.com/, November 2014.

[7] Podio. https://podio.com/, November 2014.

[8] Sme. http://storagemadeeasy.com/, November 2014.

[9] Zeropc. http://www.zeropc.com/, November 2014.

[10] MultiCloud. https://www.multcloud.com/, April 2016.

[11] Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. Improved Proxy Re-Encryption Schemes with Applications to Secure Distributed Storage. *ACM Transactions on Information and System Security (TISSEC)*, 9(1):1–30, 2006.

[12] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. *ACM Transactions on Storage (TOS)*, 9(4):12, 2013.

[13] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. SCFS: a shared cloud-backed file system. In *Proceedings of USENIX ATC*, 2014.

[14] Matt Blaze. A Cryptographic File System for UNIX. In *Proceedings of Computer and Communications Security (CCS)*, pages 9–16. ACM, 1993.

[15] Matt Blaze, Gerrit Bleumer, and Martin Strauss. Divertible Protocols and Atomic Proxy Cryptography. In *Proceedings of EUROCRYPT*, pages 127–144. Springer, 1998.

[16] Giuseppe Cattaneo, Luigi Catuogno, Aniello Del Sorbo, and Pino Persiano. The Design and Implementation of a Transparent Cryptographic Filesystem for UNIX. In *Proceedings of USENIX Annual Technical Conference*, number 1-880446, pages 10–3, 2001.

[17] Ariel J Feldman, William P Zeller, Michael J Freedman, and Edward W Felten. SPORC: Group Collaboration using Untrusted Cloud Resources. In *Proceedings of OSDI*, volume 10, pages 337–350, 2010.

[18] Kevin E Fu. *Group Sharing and Random Access in Cryptographic Storage File Systems*. PhD thesis, Massachusetts Institute of Technology, 1999.

[19] Yongkang Fu and Bin Sun. A scheme of data confidentiality and fault-tolerance in cloud storage. In *Proceedings of Cloud Computing and Intelligent Systems (CCIS)*, volume 1, pages 228–233. IEEE, 2012.

[20] Tim Kindberg Gordon Blair George Coulouris, Jean Dollimore. *Distributed Systems: Concepts and Design*. Addison-Wesley, 5th edition, May 2011.

[21] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. SiRiUS: Securing Remote Untrusted Storage. In *Proceedings of NDSS*, volume 3, pages 131–145, 2003.

[22] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 89–98. ACM, 2006.

[23] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proceedings of FAST*, volume 3, pages 29–42, 2003.

[24] Hugo Krawczyk, Ran Canetti, and Mihir Bellare. HMAC: Keyed-Hashing for Message Authentication. *RFC 2104*, 1997.

[25] Microsoft. The Encrypting File System. *http://technet.microsoft.com/en-us/library/cc700811.aspx*.

[26] Dalit Naor, Moni Naor, and Jeff Lotspiech. Revocation and Tracing Schemes for Stateless Receivers. In *Proceedings of CRYPTO*, pages 41–62. Springer, 2001.

[27] Moni Naor and Benny Pinkas. Efficient Trace and Revoke Schemes. In *Financial Cryptography*, pages 1–20. Springer, 2000.

[28] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. BlueSky: A Cloud-Backed File System for the Enterprise. In *Proceedings of File and Storage Technologies (FAST)*, pages 19–19, 2012.

[29] Huijun Xiong, Xinwen Zhang, Danfeng Yao, Xiaoxin Wu, and Yonggang Wen. Towards End-to-End Secure Content Storage and Delivery with Public Cloud. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 257–266. ACM, 2012.

[30] Shucheng Yu, Kui Ren, Wenjing Lou, and Jin Li. Defending against Key Abuse Attacks in KP-ABE Enabled Broadcast Systems. In *Security and Privacy in Communication Networks*, pages 311–329. Springer, 2009.

[31] Shucheng Yu, Cong Wang, Kui Ren, and Wenjing Lou. Achieving Secure, Scalable, and Fine-grained Data Access Control in Cloud Computing. In *Proceedings of INFOCOM*, pages 1–9. IEEE, 2010.

[32] Saman Zarandioon, Danfeng Daphne Yao, and Vinod Ganapathy. K2C: Cryptographic Cloud Storage with Lazy Revocation and Anonymous Access. In *Security and Privacy in Communication Networks*, pages 59–76. Springer, 2012.

[33] Gansen Zhao, Chunming Rong, Jin Li, Feng Zhang, and Yong Tang. Trusted Data Sharing over Untrusted Cloud Storage Providers. In *Proceedings of Cloud Computing Technology and Science (CloudCom)*, pages 97–103. IEEE, 2010.