Design and implementation of a distributed back-up system

Thomas Mager

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2014-ENST-0036

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « réseaux et sécurité »

présentée et soutenue publiquement par

Thomas MAGER

le 30 juin 2014

Conception et implémentation d'un système de sauvegarde distribué

Directeur de thèse : **Prof. Ernst BIERSACK**

THÈSE

Jury

M. Pietro MICHIARDI, EURECOM, Sophia-Antipolis - France	Examinateur
M. Guillaume URVOY-KELLER, Université Nice Sophia Antipolis - France	Examinateur et Président
M. Georg CARLE, TU Munich, Munich - Allemagne	Rapporteur
M. Pascal FELBER, Université de Neuchâtel, Neuchâtel - Suisse	Rapporteur

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

Abstract

As computer users, we create increasing amounts of data, such as digital documents, pictures, and videos. Because these data have high value in our daily life the need for back-ups arises. The creation of local back-ups on external hard drives or optical disks is a common approach, but is insufficient in the event of natural disasters or theft.

In this thesis, we provide a proof of concept for a distributed back-up system that induces only low overhead, and respects user needs to easily recover a state of a file system in a snapshot-based manner. We store distributed back-ups on residential gateways and use a central tracker as coordinator. We introduce index files in order to map the full state of a file system to a single data structure. These index files allow to reference files that have been previously uploaded so that over time only modified files need to be transferred. We divide the system into distinct swarms of flexible size so that accessing data and monitoring is easy. To increase back-up performance in this setup, we accept a lower storage space efficiency for small files by storing them close to their metadata. We show that this is reasonable due to the low amount of storage space they typically account for. For big files we use an interleaving scheme that allows us to reduce the memory footprint and make use of partially transferred data. All files and their metadata are encrypted before beeing uploaded, so that the system ensures data confidentiality. We further use state-of-the-art technologies in order to design a tracker that is highly scalable, fault-tolerant, and is even replaceable in case it entirely leaves the system. In fact, the load of the tracker only depends on the number of participants, not on the amount of data stored in the network. The system allows to configure a time span within which a user needs to recover his data in case of local data loss. We analyze this approach by using real world connectivity traces of residential gateways and show that it results in low resource demands. Together with simulations on these traces, we underline the feasibility of our service.

Résumé

En tant qu'utilisateurs d'ordinateurs, nous générons des données en quantité de plus en plus abondante. Leur importance dans notre vie quotidienne est telle qu'une méthode de sauvegarde s'avère nécessaire. La création de sauvegardes locales est une approche commune, mais insuffisante en cas de vol ou de destruction.

Dans cette thèse, nous concevons un système de sauvegarde distribué qui permet de restaurer l'état d'un système de fichiers de manière simple, grâce à la constitution d'instantanés. Nous stockons ces instantanés sur les passerelles résidentielles et utilisons un tracker centralisé pour les coordonner. Nous présentons le concept de fichiers d'indexes qui permet la correspondance entre l'état complet d'un système de fichiers et une structure de donnée. Nous divisons notre système en swarms de tailles variables, rendant l'accès aux données, et son suivi, simple. Les fichiers à sauvegarder sont traités différemment suivant leurs tailles, afin d'améliorer la performance globale du système, et de réduire les ressources nécessaires. Tous les fichiers, ainsi que leurs méta-données, sont chiffrés avant d'être téléchargés afin de garantir leur confidentialité. De plus, nous utilisons les techniques de l'état de l'art afin de rendre le tracker résistant aux pannes, et de pouvoir le remplacer s'il quitte le système. La charge sur le tracker ne dépend que du nombre de participants, et donc pas de la quantité de données stockée. Nous évaluons le système sur base de traces provenant de passerelles résidentielles réelles, avec lesquelles nous démontrons un faible impact global sur les ressources. En leur adjoignant des simulations, nous prouvons la faisabilité de notre service.

Contents

Frequently Used Variables

N	total number of gateways in the federated network
k	Number of fragments into which a file is divided
h	Number of additional redundant fragments for a file
n	Number of fragments for a file on alive storage nodes $= k + h$
r	Redundancy factor $= \frac{k+h}{k}$
t_o	Timeout value used for permanent failure detection
t_r	Time span within which a user needs to replace its gateway
t_d	Time span reserved for downloading a back-up
t_{iso}	Maximum time the system guarantees a back-up to survive without further maintenance from the swarm leader
S_t	Total amount of data to be backed up by one swarm leader
S_f	Size of a file f
α	Availability of a gateway
τ	Average lifetime of a gateway
$T_{f,i}$	Transmission block for file f and erasure coding index i

Chapter 1

Introduction

In this chapter we give a general introduction into the topic of back-up creation and the motivation to store back-ups in a distributed way. We also outline the gateway-based scenario that is focused on by our architecture. We further provide an overview of the contributions of this work and describe its organization.

1.1 Motivation

In the following we explain our motivation for this work. In particular we name the benefits of a distributed back-up and show why our approach can be a worthwhile alternative to modern cloud storage solutions.

1.1.1 The Need for Back-up

Today, an increasing amount of digital data are stored on computers. In fact, a study [Int12] conducted in 2012 shows that the amount of digital data in our world is vastly growing. As shown in Figure 1.1, the study also predicts that this trend will continue in the future.

It is generally agreed that computer data play an ever growing part in our daily lives and thus have become increasingly valuable to us. Losing all family pictures and important documents is a scenario nobody wants to face.

Unfortunately, electronic devices such as hard disk drives do not have an infinite lifetime. Stored data can become partially or even completely inaccessible without any prior indication. In many cases, data of these devices can be recovered by specialized data rescue companies [The14a, Kro14]. This procedure is, however, associated with high monetary costs [The14b] and may take a long time.

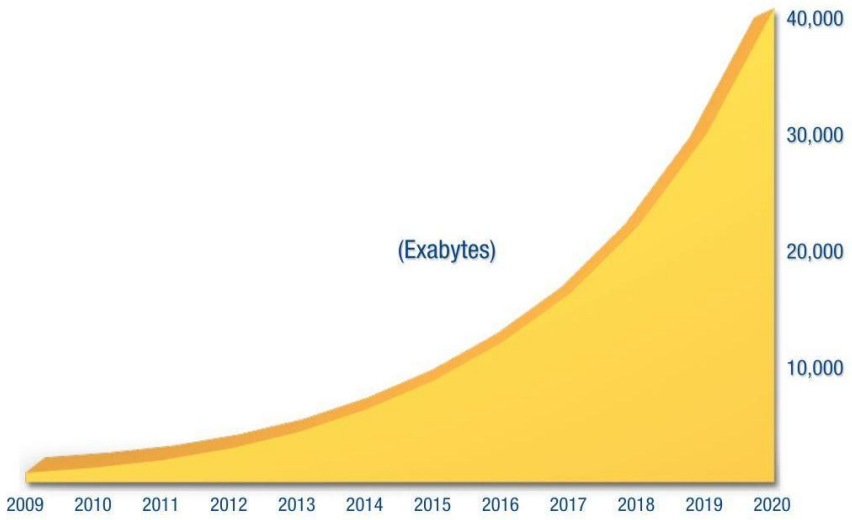


Figure 1.1: "Digital Universe", History and Projection; from [Int12]

This calls for solutions that allow people to recover from failure autonomously. As a manual approach, people often create back-up volumes by using, e.g., CD-ROM or external hard disk drives. Despite this approach there are two cases in which a user still faces data loss. First, the volumes need to be created on a regular basis, which a user may forget to do. In addition, natural disasters or theft may still lead to data loss, since the data is stored at one location only. To make matters worse, people often use many devices within a home network. This requires a more sophisticated approach since data of all devices needs to be backed up.

For these reasons, a solution that regularly gathers data and distributes them to different locations turns out to be useful. It goes without saying that this approach must take into account the confidentiality of data: placing a user's data at different locations must not involve that the original data are accessible to other parties.

1.1.2 Why Not Use the Cloud for Back-Up?

Nowadays, the cloud as a centralized service is more and more used to store data. Well-known cloud service providers are, among others, Dropbox [Dro14a] and Amazon S3 [Ama14a]. These services have in common that the original data is transferred into their systems and they internally ensure that data is protected against loss. As the increasing memory storage density [Wal05] leads to decreasing prices for storage (as illustrated in Figure 1.2), storing a back-up in the cloud appears to be an attractive solution.

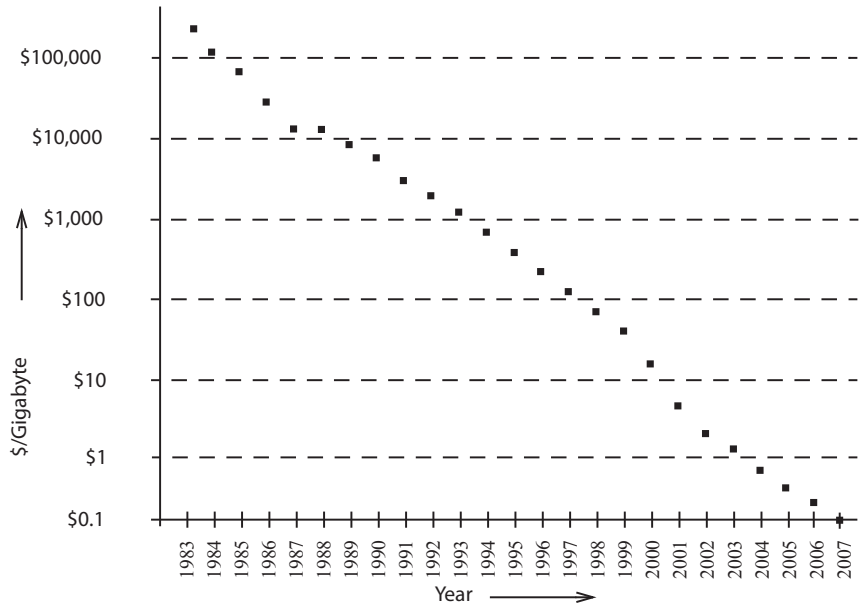


Figure 1.2: Magnetic Disk Price; from [SK09]

However, for example Amazon has adapted its price structure [Ama14b] to better represent the occurring costs, which also include the energy costs for accessing and transferring data. With energy costs generally rising over the past years [U.S14], this has become a factor of growing importance for providers of cloud based services. Further, storing the original data in the cloud makes it accessible to other parties such as employees [Dro14b] or even to the public in some cases [Dro11], which clearly is not in the users' interest.

Fortunately, we can profit from the fact that there is a mutual interest of users to store a back-up in a different location. To leverage the resources of other users to store a back-up does not result in costs if devices can be used that are already present in the users' home networks. As a result, every user can create a back-up for free and does not encounter the disadvantage of a vendor lock-in [RKYG13], which hinders a user from switching the back-up service provider, e.g., when the terms of service change. By performing continuous encryption of data before the transmission to other locations, we also achieve data confidentiality. Therefore, it is possible to keep the users' interest in the foreground and provide a service that goes beyond what a purely centralized cloud service offers.

1.2 Gateway-Based Federated Network

In this work we focus on an architecture in which residential gateways play a key role. We use them to interconnect different home networks over the Internet in order to provide our distributed service. In fact, this work was developed in connection with the European FP7 project FIGARO [FIG14], which also focuses on such an environment.

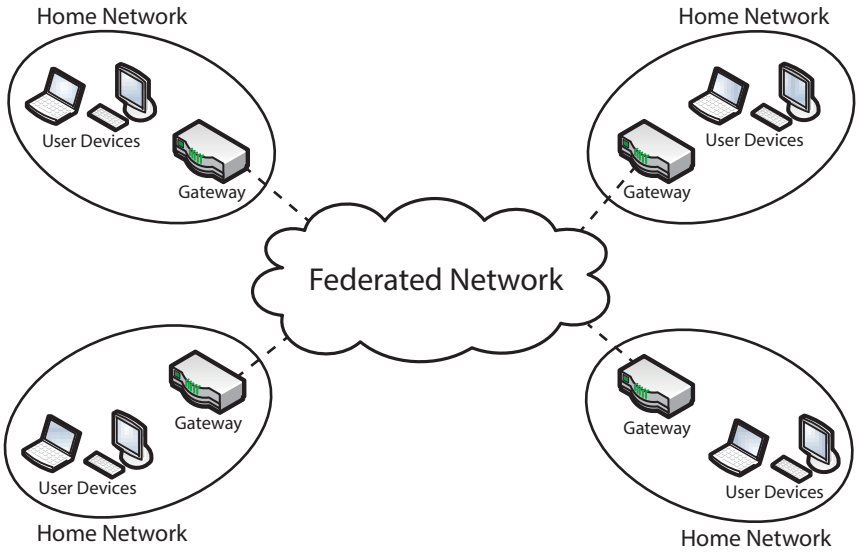


Figure 1.3: Federated Network

The global architecture consists of the following entities, as illustrated in Figure 1.3:

- **User Devices**
  User devices are heterogeneous devices such as desktop computers, laptops, mobile phones, or cameras. A user stores data on these devices and connects them to the home network.
- **Home Network**
  A home network is a local area network that is typically deployed in a single household. It interconnects all user devices that are used within this network, e.g., via wired (Ethernet) or wireless (Wi-Fi) connections.
- **Gateway**
  A gateway is a low-end computer that is generally turned on and connected to both, the home network and the Internet. A gateway has access on a local non-volatile storage system such as a hard disk drive or a Network-Attached Storage (NAS).

- **Federated Network**
  The federated network includes all gateways that join our distributed back-up service. Every gateway is reachable via the Internet for other gateways.

In this scenario we face two different network speeds, which are a determining factor for our architecture. Transfers within the home network are generally fast, while we expect lower transmission rates over the Internet. In the following, we discuss different ways in which back-ups are generally created and make a choice that takes into account the bandwidth restrictions resulting from this setup.

1.3 Back-up Plan

When creating back-ups, devising a reasonable and efficient back-up plan should always be the first step. This plan determines how often to perform a back-up and which content to transfer during the back-up process. Keeping too much data might lead to maintenance burdens, while keeping too little data might result in data loss. This section outlines possible approaches and concludes with a back-up plan for our scenario introduced in the previous section.

1.3.1 Back-up Strategies

There exist several strategies for performing a back-up [IBM06, Mic06], each of them meeting different requirements regarding storage space and transmission bandwidth, as well as as flexibility in accessing stored data. In the following we present some well known back-up strategies and outline their characteristics.

Back-up Everything

A complete copy of all data is placed on remote-site each time a back-up is performed. This back-up is easy to manage but consumes a lot of resources. For each back-up, the full amount of data is transferred and stored, no matter what is already stored on back-up-site. Due to this, each back-up can also be deleted without any restrictions.

Incremental Back-up

After storing an immutable full back-up, only differences to the previous back-ups are transferred. This leads to successive copies containing only data that changed from the preceding back-up. Although this leads to minimum storage and bandwidth requirements for back-up creation, this approach entails drawbacks. A full recovery requires the download of the last full back-up plus all

the incremental back-ups until the desired point in time. For all changed and deleted data, this leads to increased data transfer. Additionally, it is not possible to delete an old back-up without having stored a more recent full back-up before.

Differential Back-up

All differences to the last full back-up are included in each differential back-up. As a consequence, the recovery process only accesses a full back-up and the desired differential back-up. This reduces time to recover from failure. However, compared to incremental back-up, in average, differential back-up needs more time for creation. In order to delete an old back-up, again, a more recent full back-up is required.

Snapshot-based Back-up

In contrast to the previous strategies, snapshot-based back-up does not use an immutable full back-up. Instead, it uses pointers to blocks respectively files to define a consistent state of all data at a particular point in time. Similar to incremental back-up, only added or changed data since the last back-up needs to be transferred. For recovery, only relevant data needs to be accessed again, resulting in a fast time to recover. In order to free storage space, least recently used (LRU) or reference counting can be used in order to identify and delete data solely used by particular old snapshots.

1.3.2 Frequency of Back-up

How often a back-up needs to be performed may be different from one user to another. In the following we show major characteristics which lead to different requirements on back-up frequency:

Importance of Data

Some data may be very valuable to a user, others may not. To provide an example, a music file can usually be obtained from its source again, while a tax report involves a lot of individual work. Therefore, important data needs to be backed up more frequently, while less important data may be backed up less often or even be ignored. Unfortunately, an automated classification is prone to mis-classification. In our case of the music file, the back-up system could classify it to be less important. However, the user could have recorded this particular file himself or paid a fee to obtain it. In both cases he will be dissatisfied when the file is lost. To avoid the risk, we could rely on user input to classify the

importance of each file. However, this tends to be impractical considering the high number of files in nowadays' file systems and low costs for storage space.

Likelihood of Data to Change

If data frequently changes, more back-ups are necessary. The frequency of changes can be expected to vary a lot from user to user, and even over time for a single user. However, a study on file system workloads by Leung et al. [LPGM08] reveals that 90% of file re-opens concern files opened within the last 24 hours and, thus, seem to concern work in progress. A similar observation was made in [Vog99]. Therefore, performing back-ups more frequently helps to reduce the amount of work that is lost due to a failure. Conversely, ' a lower frequency for older back-ups can be acceptable since work in progress is no longer concerned.

1.3.3 Back-up Location

We distinguish two different locations where a back-up is placed. Both locations come with different characteristics, explained in the following:

On-site Back-up

This refers to a back-up stored at home. Data transfers to the gateway use the local area network and are very fast. An on-site back-up is useful whenever a single user device fails because it can be recovered very quickly. However, in the event of natural disasters or theft, we expect both entities, the user device and the gateway, to fail at the same time.

Off-site Back-up

For creating an off-site back-up, data is transferred to gateways at locations that are geographically different to the place of the gateway. Because data is transmitted over the Internet, transfers are slow and affect the bandwidth usable by the user. When data cannot be recovered on-site, a user is still able to recover data using an off-site back-up.

Back-up Retention

When a user loses data due to accidental deletion or some other reason (e.g., a computer virus), a long period may elapse until the problem is noticed. In such a case, it is possible that recent back-ups do not include the file anymore or the data is corrupted. Therefore, a back-up plan also needs to consider that a user

may need to recover back-ups from various times in the past. However, keeping more back-ups also requires more storage space on remote-site. So if storage space is limited, in consequence, the total number of back-ups is limited as well.

1.3.4 Back-up Plan for the Gateway Architecture

In our scenario, different users store data of varying importance on a single gateway. We cannot make assumptions on what will be stored, nor on how often it will be changed. Even user input about these characteristics might be imprecise since users may miss-categorize their data, resulting in possible data loss. In consequence, and also to provide a seamless service, this work considers the following case:

- all user data is valuable
- a high back-up frequency is required
- we keep old back-ups as long as possible

Since the storage space provided by other gateways is limited, we need to be able to free storage of old back-ups. This needs to be done whenever a more recent back-up cannot be stored anymore. Back-up everything and snapshot-based back-up both support cheap deletion of previous back-ups. Of these two, however, only snapshot-based back-up can take advantage of data already used by previous back-ups. In this case, a high back-up frequency only marginally increases the amount of required storage space. In consequence, we conclude with the following recommendation for a back-up plan in our scenario:

- We target a snapshot-based back-up solution
- We store on-site back-ups with high frequency (at least once a day)
- We store off-site back-ups less often (e.g., once a day)
- We decrease the frequency for older back-ups (e.g., after one week, we only keep weekly back-ups)
- Finally, we keep older back-ups as long as storage space is available

1.4 Focus and Contribution of this Thesis

In this work we focus on the feasibility of a distributed back-up system that supports snapshot-based back-ups. This entails several challenges we need to meet, such as the system's scalability, its resistence to failures and malicious behaviour, together with an adequate data placement strategy. Since we store data on participating gateways, we need to cope with storage that is unreliable in terms of availability and requires additional measures to achieve data confidentiality.

The contribution of this work is as follows:

1. We provide a proof of concept for a distributed storage system with support for snapshot-based back-ups. This also includes the management and enforcement of storage space quotas.
2. We introduce a swarm based architecture that uses file level access and reduces metadata to a very low level. Further, it is easy to monitor the data stored in a swarm.
3. We provide solutions for transferring and storing files of different sizes efficiently in such a distributed system.
4. We illustrate how to use state-of-the-art technologies in order to include a central instance that coordinates data placement in the network. This central instance is highly scalable, fault-tolerant, and replaceable in case it disappears. In addition, it is only exposed to a moderate operational load.
5. We show how to manage encryption keys so that user data can be stored confidentially and participants can be authenticated.
6. We analyze the applicability of our system by using real world availability traces. Further, we study the impact of system parameters and provide a comparison to the performance of cloud services.

1.5 Organization of this Thesis

This thesis is structured as follows:

Subsequent to this introduction, we present related work in Chapter 2, which includes established techniques used in order to achieve fault-tolerance in storage systems. Further we provide an overview of existing storage systems.

In Chapter 3 we introduce our swarm-based architecture, which includes considerations as to data placement, data encryption, and how we cope with data loss due to participants leaving the federated network.

Subsequently, in Chapter 4 we supply more details concerning our implementation. We explain how we communicate between participants and how we create back-ups. We further outline the functionality of the central instance.

We analyze the underlying failure model in Chapter 5 by using statistical tests and simulations based on a failure trace. This is accompanied by a general discussion about correlated failures.

We simulate the creation of back-ups in Chapter 6. In particular, this includes simulations concerning the time required to create a back-up and the influence of system parameters.

Finally, we conclude the thesis in Chapter 7 and provide an outlook for future work.

Chapter 2

Related Work

> "Another piece of folk wisdom is that the more elaborate the backup system, the less likely that it actually works."
>
> Citation from [SK09]

2.1 Introduction

Data storage in distributed systems is a topic that is discussed in literature since several years. In this chapter we outline some established methods used in such systems and cover the differences between storage and back-up applications. Finally, we illustrate the design and features of some existing systems.

2.2 Redundancy Strategies

Data redundancy, in general, is essential for systems in order to provide service in spite of data loss. Since in practice many systems face data loss, much research is done in the area of data redundancy. In fact, data redundancy finds its use in data transmission as well as in data storage.

There is research done on redundancy in data transmission, for example in radio broadcasting [LS01], fibre optic channels [Mas81], and satellite communication [LGZ+09, PPT09]. For data storage, there also exist plenty of application scenarios where redundancy is introduced. For barcodes, such as QR codes [TYP13], and optical disks, such as CD-ROM [KG94], redundant data is added so that the stored information is still accessible when some spots on

the media are unreadable. For storage directly connected to computers, such as ECC memory [YKMI88] and Redundant Array of Independent Disks (RAID) systems [CLG+94], redundancy is added in order to provide resilience to failures of single components. Similar to this, in distributed storage systems, data redundancy allows to tolerate failures of nodes in the network. In distributed systems, it is further possible to actively observe the current amount of redundancy in the system. Whenever the redundancy level drops below a threshold, this can be counteracted by placing additional redundancy in the system. By using such **repair** or **maintenance** process, it is possible to provide guarantees for the availability [BTC+04] or durability [LF04] of the stored data.

There are numerous methods to introduce data redundancy in a system. These methods can be subdivided into two categories: replication and erasure coding.

2.2.1 Replication

A straightforward approach to introduce redundancy into a system is the use of replication. For this, all data is simply replicated to n_{rep} different nodes in the network. In consequence, it is possible to retrieve the data even if up to $n_{rep} - 1$ replicas are lost. A lost replica can be repaired by a simple transfer of one of the remaining replicas. This redundancy scheme is very popular [DKK+01, DR01, ABC+02, BPS05, GGL03] although it entails high storage overhead [BTC+04]. To name but two examples, the Google File System (GFS) [GGL03] and Windows Azure Storage [CWO+11] use replication in order to enhance performance [HSX+12]. Especially in the area of distributed computing, replicas can be leveraged in order to increase parallelization, as seen in MapReduce [DG04].

2.2.2 Erasure Coding

Erasure coding is generally used to transform a message that consists of k blocks into n blocks such that the original data can be recovered by using a subset of the n blocks. There are two different types of erasure codes: *optimal erasure codes* and *near-optimal erasure codes*. Optimal erasure codes are also referred to as Maximum Distance Separable (MDS) codes since they only require exactly k out of the n blocks to recover the original data. XOR parities [CLG+94] and Reed-Solomon coding [WB99] fulfill this property. In contrast, near-optimal erasure codes require slightly more than k blocks in order to recover lost data. We can further differentiate between two different kinds of near-optimal erasure codes. There are *fountain codes* such as Online codes [May02], LT codes [Lub02], and Raptor codes [Sho06]. Fountain codes are also referred to as *rateless codes* because they allow to generate an unlimited number of encoded blocks for a given set of original blocks. Another kind

of of near-optimal codes are *regenerating codes* [DGW+07], which are designed to reduce the necessary bandwidth whenever redundancy in a system needs to be repaired after blocks are lost. Instead of downloading k blocks in order to reconstruct a single block, they allow to download functions of existing blocks so that it is possible to reach any point in the tradeoff between storage and bandwidth requirements, depending on the configuration used.

2.3 Repair Strategies

For on-line storage, in general, there are different policies for responding to failures. Repairs can be performed in a way that is called *reactive* (as e.g., in [DR01, CDH+06, DKK+01]), which means whenever the redundancy level is below a threshold, the system triggers a repair. Depending on the threshold used, reactive repairs are *eager* or *lazy*. In contrast to eager repair, lazy repair keeps more redundancy in the system so that a single failure does not necessarily require the upload of new redundancy. This is advantageous if the construction of new redundancy requires a prior download of data stored in the system, involving bandwidth costs. Work in [DBEN07, PJGL10] uses *proactive* repair, which injects redundancy at a constant rate with the aid of estimators. This approach is of particular interest if due to the repair process, spikes in bandwidth usage are observed and need to be smoothed.

2.4 Storage vs. Back-up

For peer-to-peer systems, lots of research has been done with focus on distributed *on-line storage* [HAY+05], targeting features that are usually provided by common file systems. These features include, for example, requirements for the latency or consistency of concurrent reads and writes. Since peers are prone to churn, these solutions need additional redundancy and maintenance mechanisms to ensure data availability over time. The redundancy is generated by using simple replication, or more sophisticated coding techniques such as Reed-Solomon coding [WB99], fountain codes [Lub02, Sho06], or regenerating codes [DGW+07]. Unfortunately, there is a trade-off [PJB11]: these codes either involve higher initial bandwidth requirements and increased storage costs, or a higher demand of repair bandwidth later on.

In contrast to on-line storage, for the scenario of *on-line back-up*, Toka et al. show that requirements on the system are relaxed [TCDM12]. Writes are solely performed by the data owner and latency is less crucial. Furthermore, a local replica can be used to inject redundancy at minimum bandwidth costs. According to the level of injected redundancy, a certain period without further maintenance can be bridged, so that no data loss is to be feared.

2.5 Existing Systems

There exist many systems that allow to store back-ups. In the following, we first describe how Apple Time Machine creates local back-ups. Subsequently, we depict different distributed storage services.

Apple Time Machine

With Mac OS X 10.5 (Leopard), Apple introduced Time Machine [App14], which is a software to create snapshot-based back-ups on a local hard drive or on a network attached storage system.

The functionality of Time Machine required Apple to change the underlying file system HFS+. The first time a back-up is created, a full copy is stored. From that point on, a journal provides information concerning FSEvents. These FSEvents allow to keep track of changes in the file system so that for the next back-up only changed files need to be inspected. This leads to very fast back-up creation, compared to the procedure of scanning the whole file system for changes, which is what most back-up tools need to do. For a new back-up, Time Machine creates a new folder on the back-up destination. Within this folder, the whole directory tree for the current state of the file system is represented. The result is a list of snapshots that is indexed by their date of creation, which we also focus in our work. As HFS+ supports references to directories, it is possible to include parts of an existing folder tree. In fact, the feature of Mac OS X to reference directories is hidden to users since these references potentially result in loops in the file system.

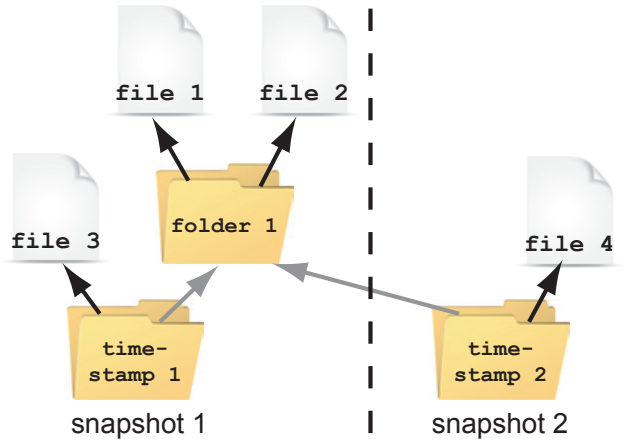


Figure 2.1: File System Tree for Two Different Snapshots in Time Machine; For Snapshot 2 we delete File 3 and add File 4.

We show an example for the creation of snapshots in Figure 2.1, where for the second snapshot file 3 is deleted and file 4 is added. The reference to folder 1 includes the folder and the two underlying files to both snapshots. Time Machine uses reference counting in order to determine whether a folder is still referenced by any snapshot, and deletes them only in case this counter equals zero. Therefore, folder 1 together with file 1 and file 2 will not be removed in case snapshot 1 is eventually deleted.

Tahoe Least-Authority File System

The decentralized Tahoe Least-Authority File System (LAFS) [WOW08] aims at providing a reliable file system and preserving data confidentiality of its users' data. With this open source project, anybody may create a storage grid out of different storage servers. This makes Tahoe-LAFS interesting for personal use, as it lends itself to setting up a storage grid with friends. Commercial use may be interesting, too, for instance by using servers to store back-up fragments. As all data are encrypted before their upload, nobody except the user himself has access to them.

To upload files, a Tahoe-LAFS gateway is required. A user can send files from other devices to this gateway by using HTTP(S) or (S)FTP. After that, the gateway encrypts the file and performs erasure coding. Subsequently, fragments are delivered by using an encrypted connection to storage servers, which, as a matter of fact, might also be Tahoe-LAFS gateways. From time to time, the gateway performs integrity tests in order to ensure that enough fragments are available in the storage grid.

The parameters for erasure coding can be individually adapted to the characteristics of a storage grid concerning, e.g., the availability of storage servers. This allows to save resources of participating storage servers while requirements on the storage grid can be met.

Files in Tahoe-LAFS are encrypted before they are broken into segments. These segments are encoded by using Reed-Solomon codes to generate blocks of which a subset is sufficient to reconstruct the segment later on. One block from each segment is sent to a storage server, which makes up one share in total. This procedure is referred to as *interleaving*. The destination of a share is chosen according to the hash of its content so that on average the shares are distributed evenly in the grid. However, each file is typically stored on a different set of storage servers, and storage servers may also receive multiple shares for the same file.

Tahoe-LAFS optionally supports automated deletion of unreferenced content. When a client has not refreshed the reference on a file by a pre-defined timeout value, the storage server frees the corresponding storage space.

Wuala

The Wuala back-up and file sharing system first was first launched in 2008 as a hybrid service, consisting of servers and peers. Wuala allowed users to trade local storage for increased storage within the distributed system.

Our measurement study [TMEBPM12] revealed that Wuala, like Tahoe-LAFS, used an interleaving scheme together with Reed-Solomon coding to generate redundant data. When a file was uploaded into the system, an encrypted copy was stored on the servers. For files bigger than 1 MiB, additional redundancy was uploaded to peers, which contributed as a cache by delivering frequently-accessed data. Where clients uploaded their data was controlled by a central coordination server. It was solely up to servers to ensure that data was accessible at any time and never got lost. Wuala implemented a sophisticated UDP-based transport protocol, which allowed to improve transfer performance when downloading from over 100 sources in parallel.

However, since the end of 2011, Wuala has not stored any data on peers anymore so that the architecture of Wuala now is a purely centralized one. Data is broken into Binary Large Objects (BLOBs) of 4 MiB that are transferred only to a single server via Hypertext Transfer Protocol (HTTP). Wuala still uses erasure codes in order to generate redundancy among servers. This procedure, however, is only internal to the data center and therefore releases clients. Especially mobile devices with restrictions on computational power and bandwidth benefit from this modification.

Glacier

The Glacier [HMD05] system is a peer-to-peer archival system that operates in an intranet environment within a company. It assumes the presence of a primary storage that uses replication to enhance access performance. The data in the primary storage is stored in the Glacier system in order to avoid data loss in case of a high number of correlated failures. To achieve this, Glacier uses erasure coding in combination with massive redundancy. The system is designed to run on desktop computers with high durability, fast network connectivity, and generally high availability.

All objects stored in Glacier are immutable and are repaired by using an eager repair strategy. Glacier uses aggregates to create collections of small files. Whenever a small file changes, the corresponding aggregate needs to be recreated and the old aggregate is discarded. Big files are stored directly in the system.

As in Tahoe-LAFS, Glacier uses so-called leases in order to keep stored objects alive. These leases need to be renewed periodically, otherwise the objects are deleted after a timeout period.

Total Recall

Total Recall [BTC+04] interconnects participating peers over the Internet to form a storage system. It uses a Distributed Hash Table (DHT) among all peers for data lookups. For each file, a unique identifier is generated, which determines the peer in the DHT that is in charge of keeping the file available in the system. The system assumes no partitioning to occur so that from different peers the DHT returns the same value for a given key at any point in time.

The system creates replicas for metadata and small files and uses an eager repair policy to respond to data loss. Big files, in contrast, are stored by using erasure coding and a lazy repair policy. This combination increases storage space efficiency in the system while keeping the required repair bandwidth low. Overall, Total Recall uses a redundancy factor of 4.

Total Recall uses a concept similar to inodes, known from Unix-like file systems: each file has a data structure in order to represent the location of corresponding immutable blocks. The consistency of updates on this data structure is ensured by the peer that is responsible for the particular file, the so-called *file master*.

The storage system provided by Total Recall supports an interface similar to Network File System (NFS) in order to access data in a convenient way.

pStore

pStore [BBST02] is a peer-to-peer back-up system based on the Chord [SMLN+03] DHT. The system allows to perform versioning on single files by using so-called file block lists. Such lists contain references to blocks that belong to a file at a certain time. In consequence, whenever the content of a file changes, the system only uploads changed blocks of this file. However, this approach increases metadata overhead and shows only limited use for most used file formats, as analyzed more closely in [MB09].

In contrast to pStore, in our work we focus on different snapshots of the whole file system on file-level, which allows us to reduce metadata and management overhead.

The pStore system uses replication in order to keep data available. Further, every peer encrypts all data before they are uploaded and performs integrity checks over time. Files can be shared; then they are stored in a namespace to which several people have access. The deletion of files is only allowed to the data owner by overwriting blocks with delete chunks.

OceanStore

OceanStore [Ke00] is an infrastructure designed in order to provide secure access to persistent objects on a global scale. It relies on untrusted servers so that all data is encrypted before it is stored in the system. It replicates frequently accessed objects on multiple servers to achieve availability, and addresses locality by caching objects close to the location at which they are needed. Further, up to a certain degree, it is designed to cope with denial of service attacks.

Every persistent object has a globally unique identifier (GUID) used for addressing. These keys are used for routing in the peer overlay network. Different versions of objects are kept instead of dealing with update-in-place consistency problems and in order to allow clean recovery in case of system failures. Barely accessed objects are moved into the *deep archival storage*, where they are stored by using erasure coding in order to increase storage space efficiency. The amount of required redundancy is determined by the system itself as it collects statistical properties of its participants.

Pastiche

The Peer-to-Peer (P2P) system Pastiche [CMN02] is primarily designed for back-up creation. It divides files into immutable chunks, which are replicated among participants to assure their availability. It uses convergent encryption [SGLM08] so that users having files with the same content end up with the same binary representation for the files' ciphertext and, thus, need to store the encrypted data in the system only once. Like pStore, Pastiche holds a list of chunks for each file and allows to reconstruct different versions of files.

A special feature of Pastiche is that it uses Pastry [RD01] to identify participants with an overlap in the data to be backed up. The overlap of the data is determined by Rabin fingerprints [Rab81], as known from LBFS [MCM01]. In this way, commonly occuring files such as installation files of operating systems can be skipped in order to increase speed and save storage space for back-up creation. Pastiche further considers the locality of peers by preferring peers that are close to each other.

Pastiche requires relationships between peers to be symmetric, so that equal amounts of data are exchanged between two peers. As this method does not lead to good results in practice, Samsara [CN03] provides an alternative storage layer below Pastiche. Samsara communicates directly over IP instead of Pastry. It uses *claims* that can be freely moved throughout the system in order to create dependency chains.

PeerStore

PeerStore [LZT04] provides another approach to create back-ups in a distributed system. It uses replication in order to create redundancy in the system. Its main contribution is to decouple metadata from actual back-up data storage in order to optimize access on metadata, while beeing flexible in data placement for the back-up storage. It further uses a symmetric trading scheme: a peer that wants to store data on another peer must accept data of this particular peer as well.

PeerStore also adapts the concept of a file block list, which allows to store different versions of files. Convergent encryption is used so that blocks need to be stored in the system only once. The metadata record of a block holds the information on which peers a replica was placed before. However, this leads to a lot of metadata, which also needs to be updated regularly.

iDIBS

Morcos et al. [MCL+06] published an improved version of the P2P-based Distributed Internet Backup System (DIBS) [Ope14]. The basic version of DIBS uses Reed-Solomon codes to create redundancy in the system. In contrast, iDIBS suggests to use LT codes in order to reduce encoding complexity and to profit from their property of beeing a rateless code.

Similar to our system, for each peer, DIBS retains a list of n peers where fragments of the back-up are stored. However, whenever a peer is suspected to have permanently left the system, all fragments previously stored on this peer are moved to a new peer. If, contrary to expectations, the suspected peer reappears in the system, all fragments stored on it will be deleted. This results in an unnecessary workload, a problem we solve in our work by reintegrating reappearing participants. DIBS uses GPG [The14c] for asymetric key encryption so that all data stored in the system remains confidential.

Chapter 3

Swarm Architecture

"Divide et impera."

Maxime attributed to
Julius Caesar

3.1 Introduction

Since our system leverages storage resources of participating gateways, we need to cope with unreliable storage that does not provide any guarantee for data remaining secure or durable. A gateway may permanently leave the system so that stored data of other participants can be inaccessible forever. For this reason, in addition to the data to be backed up, we need to store additional redundancy so that we can tolerate failures. Furthermore, we regularly need to check the presence of such redundancy in the system. If the redundancy drops below a critical threshold, a gateway needs to react by uploading more redundancy into the system again, which is referred to as *maintenance*. It is essential that the progress of maintenance is not blocked by the unavailability of particular resources. In general, however, when a distributed system grows, more participants may need to interact concurrently on shared resources [SK09]. This can lead to bottlenecks that may hinder progress in the system and, in the worst case, result in data loss.

Moreover, when a back-up is stored in a distributed way, it is important to keep track of all the locations where pieces of data are stored in the system. This information is necessary to recover from failure when we need to relocate data in order to download and reconstruct the back-up. If such information is lost or invalid, the back-up recovery will fail.

Since the aforementioned tasks are crucial for our system, we need an architecture to support the following properties in particular:

- **Scalability**
  The system is able to scale. It does not fail because of bottlenecks, when the number of users performing a back-up or the amount of data stored increases. This is important especially for systems designed to operate on a global scale.
- **Concurrency**
  Participants in the system can act simultaneously. No back-up process is blocked due to starvation [SK09] when shared resources needs to be accessed, nor does it have to wait for another process to finish first. This property ensures liveness of the system and allows progress in the upload of new data and maintenance.
- **Consistency**
  The system must not encounter states that violate consistency. In particular, this includes referential integrity, where references in the system are required to point to existing values only. For systems of bigger scale, this property is sometimes relaxed to *eventual consistency* [Vog09]. Eventual consistency does not require all data to be consistent at any point in time but guarantees it to be consistent eventually.

Throughout this chapter, we present an overview on our system architecture, that focuses on providing these properties in conformity with snapshot-based back-up. We also point at resulting characteristics which have led to the choice of the introduced architecture.

3.2 Swarm Architecture Overview

In this section, we provide an overview on our general system architecture. We explain how devices in our scenario of federated networks interact with each other and define their roles and duties. In addition to user devices and gateways (cf. Section 1.2), we introduce a centralized instance, referred to as *tracker*. We organize the federated network FN into *swarms* $SW \subset FN$, which are individual sets of storage nodes for each swarm leader in order to store its back-up. Figure 3.1 illustrates this architecture from the point of view of a single swarm leader. We subsequently discuss the shown entities in detail.

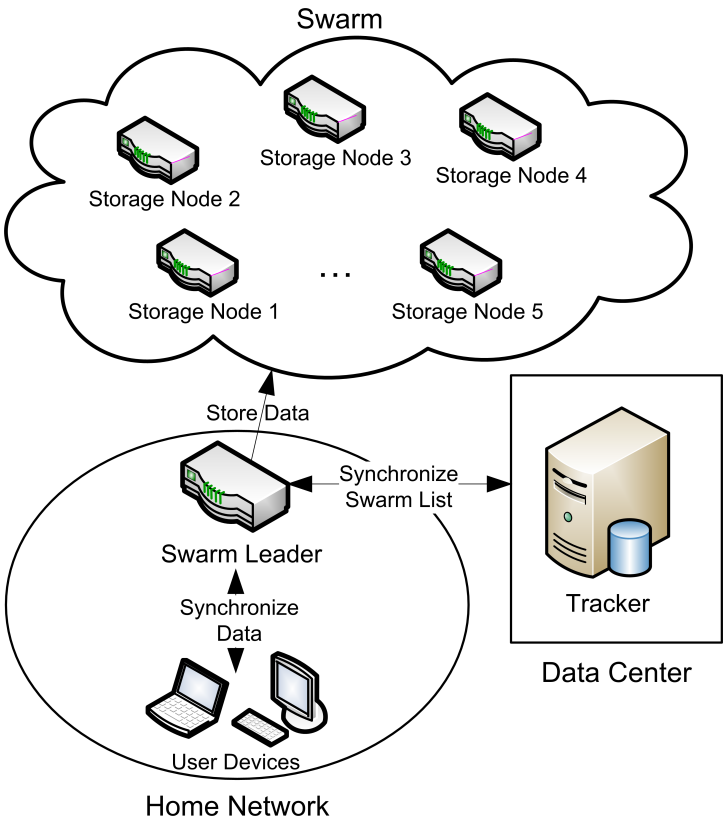


Figure 3.1: General Architecture

3.2.1 Gateway

A gateway is the intermediate for both networks, the home network and the federated network. It stores all data to be backed up within a home network and is in charge of uploading an external back-up to other gateways which are part of the federated network. On the other hand, it receives such data fragments from other gateways and is obliged to hold them. A gateways therefore plays the role of a swarm leader and the role of a storage node, as explained in the following.

Swarm Leader

A swarm leader keeps a copy of all the data of devices within a home network to be backed up. It does so by regularly synchronizing with the devices, so that an on-site back-up within the home network is available. Whenever a device fails, this **on-site copy** can be used to recover at the speed of the home network.

In addition, the swarm leader uses the local on-site copy to **create data fragments** to be uploaded to other gateways. This way, we relieve user devices from staying connected to the Internet in order to create an off-site back-up. This off-site back-up is managed by the swarm leader, which we expect to be on-line most of the time. Apart from the upload of the initial back-up, the swarm leader is also in charge of performing **maintenance** of the off-site back-up. Since storage on other gateways is unreliable, we need to consider malicious behaviour, which may corrupt our back-up. Therefore, a swarm leader regularly **checks the integrity** of the off-site back-up.

As a result, by synchronizing all valuable data from the user device to the swarm leader, we achieve a fast on-site back-up and outsource the more extensive task of creating an off-site back-up to the swarm leader. A gateway only uploads fragments for its own off-site back-up and therefore has the role of a swarm leader exactly once.

Storage Node

A storage node $sn \in SW$ **stores data fragments** related to a foreign swarm leader's back-up. For this, it offers a key-value store that accepts data until a given quota is reached. It further provides a method for storage reclamation, so that old back-ups can be substituted by new ones.

We consider the storage offered by a storage node to be **unreliable** in terms of confidentiality and integrity. Hence, before sending data fragments to a storage node, a swarm leader encrypts the data so that a storage node cannot see the original data of a back-up. For a swarm leader, in order to recover, several storage nodes must be contacted. In return, a gateway has the role of a storage node several times for different swarm leaders.

3.2.2 Swarm

A swarm is a set of randomly chosen storage nodes $SW = \{sn_1, sn_2, \ldots, sn_i\}$. Each swarm leader backs up all its data on such an individual swarm. The size of the swarm set is flexible over time, hence, storage nodes can be added and eventually removed by the swarm leader.

Since the swarm leader is the only source for data stored in a swarm, its presence has a direct impact on the amount of available redundancy. Therefore, a swarm can be in one of the following states:

- **Intact Swarm:**
  A swarm leader generally performs maintenance. As storage nodes in the swarm leave the system, the swarm leader adds new storage nodes and

uploads additional data fragments to them. We explain the maintenance process in detail in Section 3.5. A swarm is considered intact even though a swarm leader may be off-line for a short period (e.g., several days).

- **Isolated Swarm:**
  The swarm leader is off-line for a long period (e.g., several weeks). It does not perform maintenance, and thus, the redundancy present in the swarm decreases over time. We design the redundancy level in a swarm to be high enough so that a swarm can be in the isolated state for a predefined period of up to several years. Within this period, the back-up can be restored using the data available in the swarm.

After this period, the redundancy available in the swarm drops below a threshold so that data loss may occur. It is only in this case that the user can not download its back-up from the swarm anymore. Therefore, before we reach this point, a third party, e.g., a service within a data center, can download and reconstruct all encrypted files and hold them ready for download for the back-up owner.

However, access patterns in file systems [LPGM08, LPGM08] show that the probability that a user reopens files decreases over time. When data loss occurs, a back-up is the only source to reopen last accessed files again. In consequence, we also expect the probability that a user accesses a back-up after local data loss to decrease over time. In our system, the time a swarm can remain in the isolated state without data loss is a parameter; thus, we are able to provide a user sufficient time to notice the failure of its gateway, setup a new gateway, and download its back-up. Given the user is aware of this period and its duration is sufficiently long, we assume that in general no (costly) third party is required to recover a back-up. This is why, in this work, we do not consider such a third party to guarantee back-up durability forever.

3.2.3 Tracker

The tracker is a trusted intermediary, running as a central instance within a data center.

In order to join the federated network, a gateway first contacts the tracker. The tracker provides information about the states of other gateways. In particular, the tracker fulfills the following functionalities:

- **Back-Up of Swarm Set**
  When a gateway fails, it loses not only the on-site copy, but also the information about where previous snapshots are stored within the federated network. For this reason the tracker keeps a copy of each swarm set, which, in case of failure, must be requested to be able to start recovery.

- **Track Gateways**
  The tracker keeps track of the gateways in the federated network. Therefore, the gateways occasionally send heartbeat messages to the tracker. The tracker updates a timestamp t_l for the particular gateway to the current time whenever such a heartbeat is received. The tracking also includes the task to map a static gateway identifier Id_{gw} to the currently assigned IP address of a gateway. Other gateways need to query this information in order to open connections to particular gateways.

- **Observe Fairness in the System**
  The tracker receives reports from gateways about possible misbehaviour of other gateways. It regularly aggregates this information and eventually excludes misbehaving gateways from the federated network. Such misbehaviour, e.g., the illegitimate deletion of fragments as a storage node or the creation of too much redundancy as a swarm leader, can be detected by the tracker due to its global knowledge over the federated network.

- **Certificate Authority**
  For each gateway in the federated network the tracker keeps a certificate, binding a gateway's identity to a corresponding public key. This way, participants can validate each others identity and, in case of misbehaviour, we are able to revoke a gateways authorization to participate in the federated network.

Clearly, a central instance increases the risk to create a bottleneck, possibly preventing a system to scale [HAY+05]. In fact, the functionality offered by the tracker can also be realized by using a DHT among gateways. In pure P2P systems, however, it is difficult to create consensus on the state of the system. The Byzantine Generals' Problem by Lamport et al. [LSP82] is often used to illustrate this problem. It states that in order to decide about the current state in a system of untrusted parties, at least $3c+1$ total participants are required in order to tolerate c faulty or misbehaving participants. In consequence, to achieve consensus, a lot of messaging overhead is required, and yet, the system can be compromised, e.g., by creating a large number of pseudonymous identities, known as a Sybil attack [Dou02].

In contrast, by using the tracker as a single instance with global knowledge, such decisions are made by the tracker and participants simply adapt to it. This may lead to higher load on the tracker so that its scalability is very important and excessive usage should be avoided by design. In fact, the tracker may consist of several machines so that a single hardware failure does not lead to data loss or service interruption.

Our architecture requires a user to trust the central instance to provide support in creating, maintaining, and potentially recovering his back-ups. We explicitly

do not, however, rely on the tracker in terms of data confidentiality. Data fragments in our system exclusively contain encrypted data that only the data owner is able to decrypt (as we show in Section 3.7).

3.3 Snapshot Representation

The term **snapshot** v_t refers to a consistent state of a file system's folder structure with all its underlying files at a particular point in time t when a snapshot is created. In our system, therefore, each snapshot consists of the following:

- The **file set** FS_t, which is the set of all **file contents** $f_1, f_2, \ldots, f_q$ that are included in snapshot v_t. Here we distinguish file contents from files. In case two files have the same content, in most file systems, each of these files still occupies the storage space corresponding to its file size. In constrast, a file *content* in our system can be referenced multiple times, e.g., by different snapshots, so that the storage space needs to be allocated only once.
- All **metadata** M_t required to reconstruct the folder structure and properties of files for a snapshot v_t. This includes information about file names, used folders, and also file metadata such as file ownership information, access permissions, and file access times. Furthermore, this metadata involves references to the file contents mentioned above and, in case of an off-site back-up, keys to decrypt file contents.

As a consequence, we represent each snapshot as a tuple (M_t, FS_t) which contains all data required to entirely reconstruct a snapshot: $v_t \rightarrow (M_t, FS_t)$. In general, changes in file systems only concern a small portion of the total data stored [LPGM08]. As a consequence, we expect that most file contents from one snapshot to another remains unchanged. By referencing identical file contents we therefore avoid the necessity to store duplicate file contents that is already used by previous snapshots.

Below we give an outline of how we consistently retain file contents and metadata for the on-site copy as well as for our off-site back-up.

3.3.1 On-Site Snapshot Representation

In order to represent snapshots on the gateway, we leverage its file system. For each snapshot, we have a dedicated folder on the gateway. Within such a folder we replicate all corresponding metadata M_t. Instead of replicating the file

contents for each snapshot, however, we **deduplicate file contents** of the file set FS_t of the new snapshot and the file set FS_{t-1} of the preceding snapshot.

We therefore only require the file contents $FS_t \setminus FS_{t-1}$ to be transferred from a user device to the gateway. In conjunction with the typically high transmission rate within the home network, this results in fast on-site backup creation and low resource usage. Figure 3.2 shows an example for two snapshots. For each

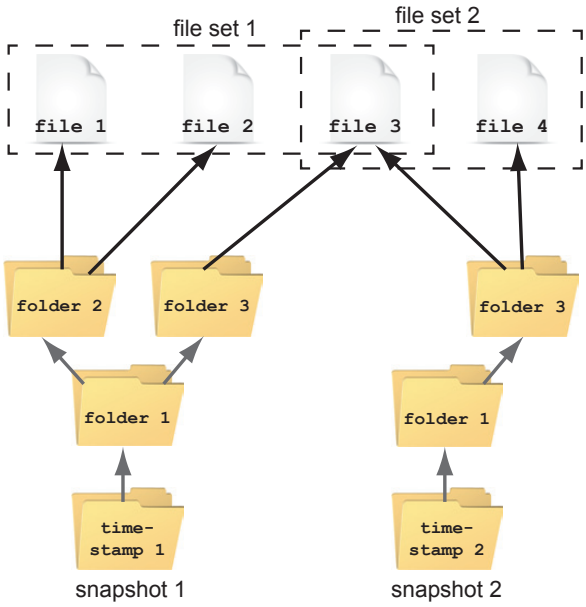


Figure 3.2: Snapshot Information in the File System for On-Site Back-up

snapshot there is a folder named by the current timestamp. From snapshot 1 to snapshot 2, a folder (folder 2) including two files is removed and a new file (file 4) is added. Both snapshots have file 3 in common so that we use a reference to the existing file content on the gateway.

From this representation follows that, after a snapshot is created on the gateway, all data relevant to a snapshot is **immutable**. Even though a snapshot may share file contents with future snapshots, all referenced file contents and metadata information for this snapshot will remain unchanged. This property is relevant for the maintenance of the off-site back-up in the federated network, where we rely on original file contents to be preserved over time.

3.3.2 Off-Site Snapshot Representation

To create off-site back-ups, we need only the data on the gateway and, hence, require no access to data stored on user devices anymore. The goal is to map

the snapshot representation of the on-site back-up onto the distributed off-site back-up.

In consequence, we need to find a way to deduplicate files stored within a swarm. We achieve this by adding a level of indirection [SK09], which allows us to reference files based on their content. For this we introduce a **file content identifier** Id_f, which is deterministically computed as $Id_f = H(H(f))$, which means by calculating the hash of a file content's hash[1].

This procedure ensures that the identifier for the same file content will remain unchanged for later snapshots, even in case of data loss on the gateway. Furthermore, if a file content identifier already exists in the swarm, the file content does also exist and its transfer to storage nodes can be skipped.

For each snapshot, we create an **index file** that holds all metadata M_t necessary to reconstruct the original state in the file system. As we will see in Section 4.3.3, we can also profit from embedding small files into the index file. After creation, we store the index file like an ordinary file within the swarm. Since the index file is the starting point when it comes to snapshot recovery, we need to be able to locate it after failure. Therefore, each storage node in a swarm keeps a list of file content identifiers of index files together with their corresponding creation time. Figure 3.3 illustrates the result using the same example as for the on-

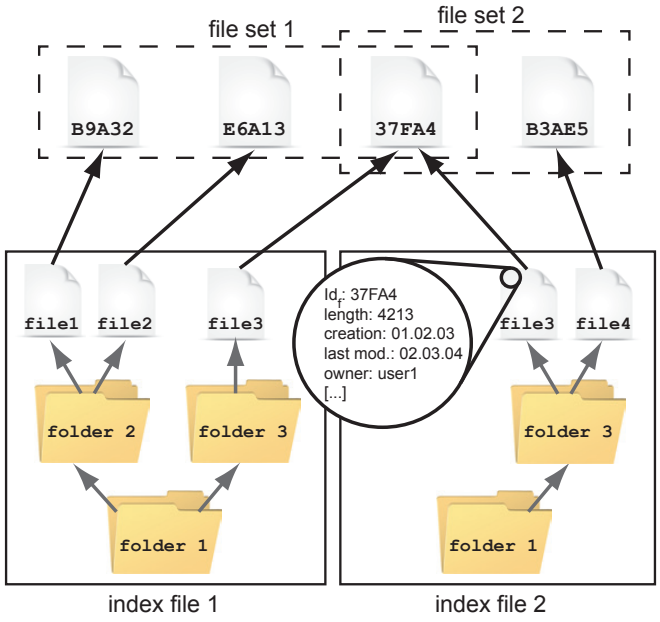


Figure 3.3: Snapshot Information Embedded into Index Files for Off-Site Back-up

[1]We use the result of a single hash operation to derive the encryption key, as we explain in Section 3.7.2.

site back-up in Figure 3.2. Since we address file contents by using identifiers, we keep these references embedded into the index file, which holds all other metadata concerning a snapshot as well.

3.4 Data Management Strategy

In this section, we explain how we distribute data among gateways of the federated network and the central tracker. This includes the policy for storage node selection as well as the way we address data in the network. We explain how we split files into fragments and where these fragments are placed subsequently. Further, we show how we take into account that there is only a limited amount of storage space on gateways in our system.

The way we manage data placement does not only have an impact on the amount of metadata necessary to relocate data, but also influences how maintenance in the system is performed (as we explain in Section 3.5).

3.4.1 Data Placement Policy

In our system, the tracker assigns new storage nodes to a swarm uniformly at **random**. This corresponds to a **global data placement policy**, in which potentially any of the N total storage nodes in the federated network can be selected to store data fragments of a swarm leader.

As analyzed in [GMP09] and [VI12], a global data placement policy leads to higher system reliability compared to local placement policies, which store data on neighboring nodes. This is reasoned by faster repairs of lost redundancy in an environment where bandwidth on storage nodes is limited. While the average bandwidth for all placement policies is equal, since the number of repairs remains equal as well, the distribution of the workload for repairs is different.

We show this by comparing the repair process for both, the buddy placement policy and our global placement policy. In the buddy placement policy, the network is organized into groups where gateways store data in a reciprocative way: when a gateway A stores some of its data on another gateway B, gateway B in turn also stores some of its data on gateway A. This results in *symmetric* relationship. In our global placement policy, on the other hand, we are not restricted to groups. The relation that gateway A stores its data on gateway B does not imply that B also stores its data on gateway A. The relationships in the global placement policy therefore can be *unidirectional*, which is why in our system we differentiate between the two roles of a gateway as swarm leader respectively storage node. We visualize the mentioned data placement policies in Figure 3.4.

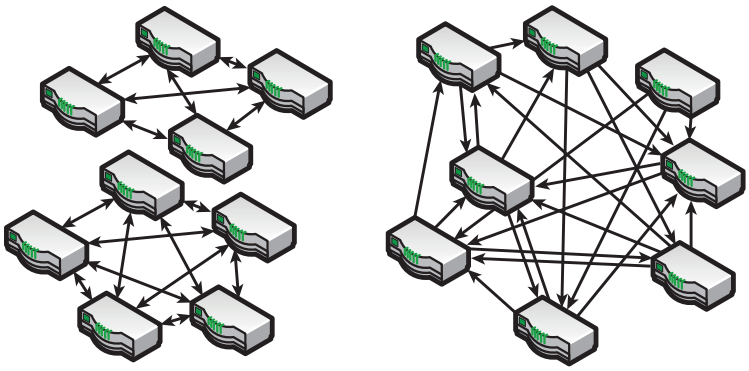


Figure 3.4: Data Placement Policies

In both scenarios, a gateway holds data of n other gateways. When the gateway leaves the system, these n gateways eventually select a new gateway to replace the previous member.

Here the differences between both placement policies become apparent. According to the buddy placement policy, the n gateways in the group need to choose the *same* newcomer for their group. As soon as the newcomer joins this group, it receives data from these n gateways at the same time. This high number of concurrent transfers to a single gateway can result in a bottleneck, extend the time required to finish the repair process, and, therefore, lead to a less reliable system.

In contrast, regarding the global placement policy, the n gateways that need to select a newcomer choose a gateway at random. This leads to the choice of n newcomers that are *distinct* with a probability of $\frac{\binom{N}{n} \cdot n!}{N^n}$, which is high in the case of $N \gg n$. In case the federated network is big, we therefore only expect a single transfer to different storage nodes. One gateway leaving the system in the global placement policy therefore leads to a load that is evenly distributed over n gateways in the federated network.

3.4.2 Swarms as Distributed Key-Value Stores

Within a swarm, a swarm leader addresses a file content using a file content identifier Id_f. Whenever a swarm leader delivers a fragment of a file content to a storage node, the fragment is accompanied by its corresponding file identifier. A storage node stores all incoming fragments indexed by their file content identifiers. As a result, fragments can be requested by using their corresponding file content identifier. The storage structure on storage nodes therefore corresponds

to the concept of **key-value stores**, where a key (in our case the file content identifier) is in relation to a value (a particular fragment of the file content).

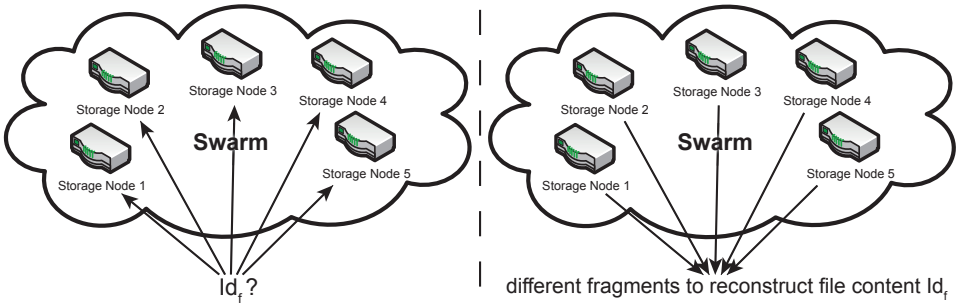


Figure 3.5: Addressing Fragments Corresponding to a Single File Content Using their Common File Content Identifier

Each storage node stores a different fragment of a file content so that, using their common file content identifier, we are able to address different fragments in a swarm (as illustrated in Figure 3.5). All these fragments correspond to a single file content and contribute to its recovery.

In the following, we provide more details about how we organize data in our system.

Back-Ups as Evolving Streams

We design our system so that it is able to reflect the time component of back-ups: Over time, we need to store new snapshots, which typically include new files. These new files occupy additional storage space and, in consequence, storage space requirements in the system increase over time. As storage space on gateways is, however, a limited resource, we need to consider that we can only store a limited number of snapshots. The fact that we need to comply with a quota equally applies to both, on-site snapshots and off-site snapshots.

We illustrate the development of snapshots by an example, as shown in Figure 3.6. In step 1 we store a snapshot v_1, which references five different file contents. At a later point in time, we add snapshot v_2, as can be seen in step 2. Snapshot v_2 has three files in common with snapshot v_1, but also adds two files and releases two files. Subsequently, in step 3, we store snapshot v_3, which includes another new file. From this point onwards there is not enough storage space available to store a new snapshot v_4. For this reason, we delete the oldest snapshot v_1 as shown in step 4, which enables us to free the storage space for files solely used by snapshot v_1. As shown from step 5 to 6, we now have enough space to store a new snapshot v_4.

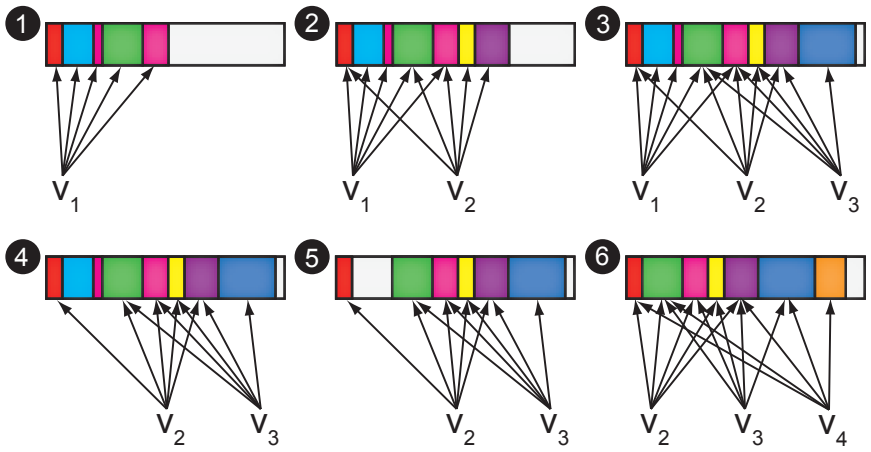


Figure 3.6: Example of the Evolution of Storage Space Usage Over Time

We see that the storage space usage is allowed to change over time due to both, added files and obsolete files. In order to emphasize this behaviour, we denote the data stored for all snapshots as a back-up **stream**. In fact, such a back-up stream involves the file set of all unique file contents UF of snapshot v_x to snapshot v_y:

$$UF = \bigcup_{t=x}^{y} FS_t$$

We are going to address quota compliance in more detail after having explained how we spread a back-up stream over all storage nodes in a swarm.

Creation of Substreams

Figure 3.7 illustrates our policy to store all file contents $f \in UF$ in a swarm. In the first two steps, we split each file f into k $(10 \leq k \leq 200)$ equally sized **transmission blocks** $T_{f,i}$ with the **index** $i \in \{1, \ldots, k\}$. As a result, each transmission block of a file with size S_f has a size of $\lceil S_f/k \rceil$. Using erasure codes (such as Reed-Solomon [WB99], we generate h additional transmission blocks $T_{f,i}, i \in \{k+1, \ldots, k+h\}$ (as shown in step 3). As a property of erasure codes, any k out of the $n = k + h$ different transmission blocks will be sufficient to reconstruct the original file content later on.

For data placement on one storage node, we group the transmission blocks of all files by their index i to a **substream** SS_i:

$$SS_i = T_{1,i}, T_{2,i}, \ldots, T_{f,i}$$

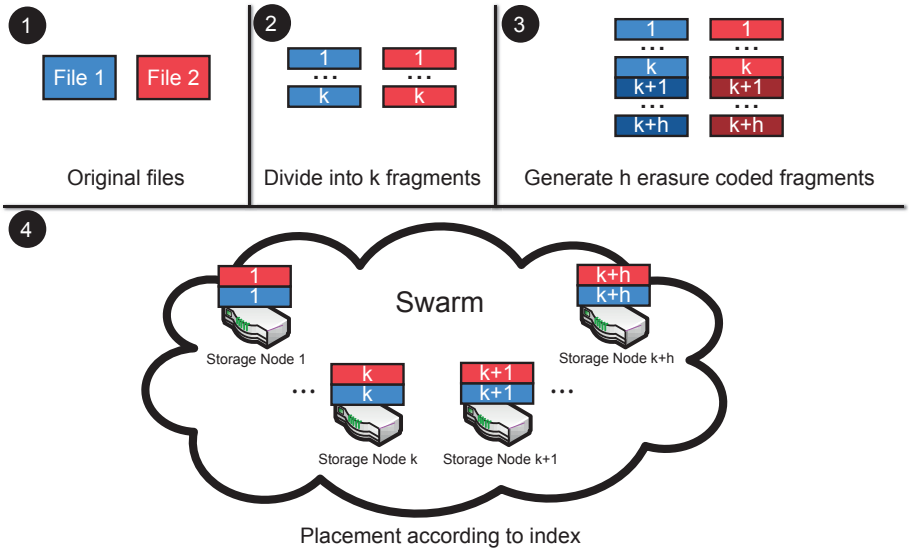


Figure 3.7: Placing Substreams on Storage Nodes

As a consequence, the extent of such a substream evolves whenever the set of unique files UF changes. The overall size of such substream depends on the total amount of data S_t used for unique files across all snapshots and is defined by S_t/k.

Assuming a swarm to store a snapshot with file set FS_1, for a new snapshot with a file set FS_2 we can skip all files $FS_1 \cap FS_2$ since $FS_1 \cap FS_2 \subseteq FS_1$ and $FS_1 \cap FS_2 \subseteq FS_2$, and therefore

$$\begin{aligned} FS_1 \cup FS_2 &= FS_1 \cup ((FS_1 \cap FS_2) \cup (FS_2 \setminus FS_1)) \\ &= FS_1 \cup (FS_2 \setminus FS_1). \end{aligned}$$

We place a distinct substream on each storage node within a swarm (shown in Figure 3.7 step 4). Further, we assign each storage node the index used for erasure coding when the substream was created so that we are able to reconstruct the data later on.

Depending on whether a storage node holds the latest transmission blocks of a substream or not, we speak of the storage node being in a **synchronized** or an **unsynchronized state**, respectively.

Quota Management

A gateway in our system cannot offer more than its local hard disk space to the system. Because of this, we also face limited amount of storage space in our distributed system.

To cope with this restriction, we introduce storage space quotas on a gateway. Given the total size of hard disk space Q_h on a gateway, we divide this space into two quotas, as shown in Figure 3.8.

Figure 3.8: Quota Restrictions on a Gateway

The **on-site snapshot quota** Q_{on} restricts the storage space used for on-site snapshots. This quota requires us to eventually remove old snapshots and reclaim storage space in order to store more recent snapshots, on-site as well as off-site ones. The **off-site snapshot quota** Q_{off} is the storage space offered to other participants in the federated network. It allows foreign swarm leaders to store data and therefore accounts for the gateway's role as a storage node. In general, we want that a gateway is able to store all its current on-site snapshots in its swarm. In this context, since we use redundancy in our system, we need to consider the resulting storage overhead. The size of the off-site snapshot quota thus depends on Q_{on} and the storage overhead $r = \frac{k+h}{k}$ and can be approximated by $r \cdot Q_{on}$[2].

The quota Q_h therefore is defined as $Q_{on} + Q_{off} = Q_{on} + r \cdot Q_{on}$ and allows us to infer Q_{on} in dependence of Q_h:

$$Q_{on} = \frac{Q_h}{r+1}$$

We further introduce substream quotas within the off-site snapshot quota. Such a substream quota Q_{ss} ensures that participants do not use more storage space on storage nodes than intended. A storage node keeps up to n substreams from different swarm leaders. Hence, we have n substream quotas on a storage node, each with a size of $\frac{Q_{off}}{n} = \frac{Q_{on}}{k}$. Whenever such quota exceeds its upper limit, a storage node does not accept further transmission blocks from the corresponding swarm leader, forcing it to free storage space first.

Quota Pools

Users in our system may have different needs for the amount of back-up storage. This can be the case because either more data needs to be stored or a high

[2] In practice, this can be slightly more, due to the inaccuracy of the failure detector, as we explain later.

number of snapshots is desired. We also expect that gateways are equipped with hard disk drives of different size. Further, the storage requirement of a user can also change over time because larger amounts of data need to be stored.

For this reason, we consider the concept of **quota pools** in our system. We divide the federated network into these pools by grouping gateways according to the storage quota Q_{off} they offer to the network.

The tracker only assigns gateways within the same quota pool to each other, so that a swarm leader only has storage nodes in his swarm that share the same amount of storage space with the network.

To allow a gateway to switch to another quota pool, a swarm leader successively moves its substreams from one storage node in the old quota pool to another storage node in the new quota pool. This way the swarm size remains equal over time and deters swarm leaders from abusing the temporary membership in two quota pools in order to store more data than permitted.

3.4.3 Data Kept on the Tracker

In this section we overview the data we need to store on the central tracker. This includes in particular all information related to the tracking of gateways and the back-up of a swarm leader's swarm set. We summarize the data in Figure 3.9 and subsequently explain the records in detail.

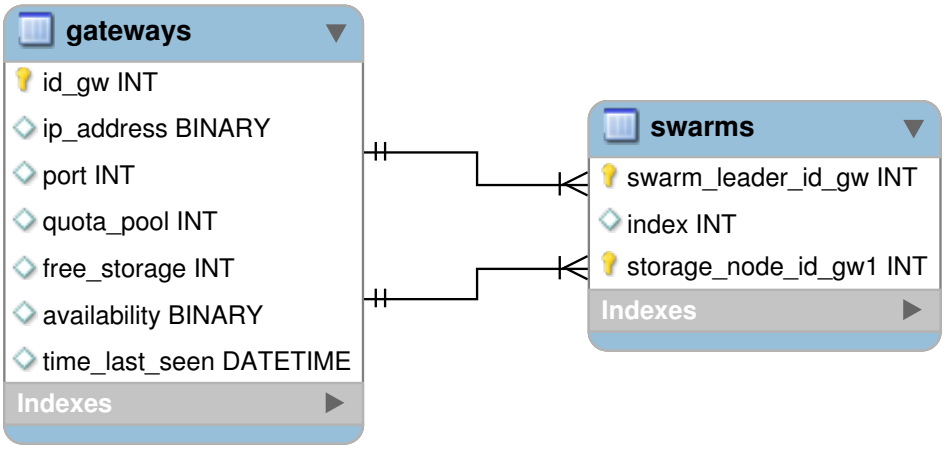


Figure 3.9: Data Stored on the Tracker

Tracking of Gateways

We keep the following gateway specific information on the tracker:

- The **gateway identifier** Id_{gw} which uniquely identifies a gateway in the federated network.
- The current **IP address** of a gateway.
- The **port** the application listens on.
- The **quota pool**, in which the gateway is a member.
- The amount of **free storage** a gateway can offer to others.
- A bitmap which allows us to approximate the **availability** of a gateway. We shift the bitmap each time a ping is sent to the gateway and set the new bit in accordance to its on-line state.
- The **last time** t_l a gateway was **seen** in the federated network.

The storage space complexity for this data is given by $O(N)$, so that the storage space requirement grows linearly with the number of gateways in the federated network.

Back-Up of the Swarm Set

We need to allow a swarm leader to recover its swarm set in case of local data loss. For this reason we keep a copy of a swarm leader's swarm set on the tracker, which involves the relation $(\text{swarm leader}, \text{index}, \text{storage node})$. Storing the erasure coding index i of a substream (introduced in Section 3.4.2) for the relationship between swarm leader and storage node further prevents deniability in the system; a storage node cannot claim to have different data from those that have been uploaded before. Since every swarm leader stores its data on a limited number of storage nodes, the storage space complexity for this data also grows linearly by $O(N)$.

3.4.4 Implications

The introduced data placement policy has several implications on the architecture:

1. The architecture uses file level access. In consequence, when we need to reclaim storage space occupied by unreferenced files, we are able to

free their storage space directly. We do not, in contrast to work done in [HMD05, Ke00, TDM10], keep[3] data in so-called aggregates, which bundle multiple files into a single object. These aggregates require additional metadata for bookkeeping and hinder direct file deletion in the system. Without overhead due to block alignment, a file cannot be deleted directly within such an aggregate because of the erasure coding and encryption used. This overhead increases especially when data is split and distributed to a higher number of storage nodes. To perform maintenance (we introduce our maintenance procedure in Section 3.5) for these aggregates using the on-site copy on the gateway, we would also need to keep files that in fact have already been deleted, but would still be required to regenerate the binary representation of the original aggregate. The only way to instantly free storage space of files within an aggregate is to upload a new aggregate without the corresponding file before deletion of the former aggregate.

2. Any k out of $k + h$ storage nodes are sufficient to recover all files stored in a swarm and, thus, all snapshots stored. This eases failure detection since we do not need to check the availability of individual transmission blocks but only the availability of the storage nodes in the swarm set. A swarm set is only a rather small subset of all storage nodes in the federated network so that we significantly reduce messaging overhead for heartbeat messages. As this allows fast and cheap reintegration of reappearing redundancy, our maintenance procedure benefits from this property as well.

3. We save storage space required for metadata on the storage node. Instead of storing the full relation $(Id_f,\ i,\ T_{f,i})$ for each transmission block, which we require for recovery, we only need the relation $(Id_{gw},\ i)$ once per storage node and the relation $(Id_f,\ T_{f,i})$ for each transmission block. According to the source of a downloaded transmission block, we can recover the index i used for erasure coding. For smaller files the amount of metadata required to store them is generally more decisive in terms of storage space efficiency than for files of big size. This approach therefore increases storage space efficiency especially for smaller files in our distributed system.

4. Since all storage nodes in the swarm hold a transmission block of each file, we spare additional look-ups to relocate transmission blocks corresponding to particular files. The knowledge of a single file content identifier is sufficient to download all transmission blocks necessary to recover a corresponding file content. We further require no interaction with the tracker

[3] However, we use aggregates for the *transmission* of smaller transmission blocks (explained in Section 4.2.2).

to store or retrieve files from the system. The amount of data stored on the tracker is independent of the amount of back-up data, which has a positive impact on scalability.

5. The loss of a single storage node leads to the loss of a whole substream and therefore affects the redundancy level of *all* files stored by a swarm leader. In consequence, a swarm leader needs to upload a new substream of size S_t/k, which may result in a longer lasting transfer. This renders the system vulnerable to successive failures within a short time period. We examine this issue more closely in Chapter 5.

In the following section we focus on the procedure of refreshing the amount of redundancy stored in a swarm.

3.5 Maintenance Procedure

In our system, we place data related to our off-site back-up on foreign storage nodes. We rely on these storage nodes to serve enough transmission blocks needed to recover the back-up in case of swarm leader failure. However, we also need to consider failures of storage nodes. To address these failures, we regularly perform the process referred to as *maintenance*, which is the process of responding to data loss in a swarm by uploading additional substreams.

In this section we first explain how we detect failures in our system and how we react to such failures. We then provide some implications and look at the potential bandwidth costs of the approach used.

3.5.1 Failure Detection

In general, failures can be divided into two categories. If a storage node is only temporarily unavailable for other participants, such a failure is denoted as a **transient failure**. The storage node will eventually return to the system so that the data stored on it can contribute to recovery again. For a **permanent failure**, however, the storage node has left the system forever. In the latter case, the data stored on the storage node is lost permanently.

We illustrate this behaviour in Figure 3.10. As long as a gateway is *alive*, it is either in the *on-line* or *off-line* state. From the on-line state, it changes into the off-line state (equivalent to a transient failure) with probability $\frac{1}{t_{on}}$, where t_{on} is the average duration a gateway is in the on-line state. By analogy, t_{off} refers to the average duration a gateway is in the off-line state, so that it changes back to the on-line state with a probability of $\frac{1}{t_{off}}$.

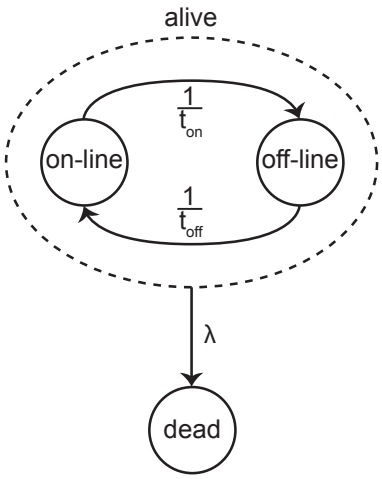


Figure 3.10: Model of the Gateway Behavior

The outer process is absorbing, which means as soon as a gateway is once in the *dead* state (equivalent to a permanent failure), it cannot return to the *alive* state anymore. Gateways with a mean lifetime τ turn into the dead state with probability $\lambda = \frac{1}{\tau}$.

Unfortunately, it is impossible to distinguish these states from outside: using remote network measurements, transient and permanent failures both have the same characteristics, while it is unknown how long a transient failure of a storage node lasts. However, in contrast to transient failures, permanent failures entail data loss in a swarm and, consequently, require the upload of additional redundancy.

In practice, distributed systems use **failure detectors** [CT96] to cope with this problem. Failure detectors are modules that regularly send heartbeat messages and suspect a participant to be failed when there is no reply within a timeout period t_o. Since we are interested in permanent failures in particular, t_o will typically be a large period (e.g., several weeks).

In our system, we record a timestamp t_l which indicates the last point in time a gateway was seen in the federated network. Due to its high availability, we choose the tracker to collect this data. At the current point in time t_c we are therefore able to declare a gateway to be in the **alive** state in case $t_l \geq t_c - t_o$, while it is in the **dead** state for $t_l < t_c - t_o$.

In practice, failure detectors are *complete*, which means they are able to eventually detect all permanent failures. On the other hand, they are not *accurate*, so that a suspected storage node might return at a later point in time and, thus, proves to be still alive.

3.5.2 Repair

For general on-line storage, the purpose of repair is to ensure *durability*, which is the property of data to survive permanently, and *availability*, which ensures that data can be accessed at any point in time. As data availability implies data durability [DBEN07], achieving high data availability results in high costs for maintenance in the system [TCDM12, PJGL10]. In the case of back-up systems, back-up recovery is a longer lasting process. We do not need to be able to access the whole back-up at any point in time, we simply need the download process to finish eventually. For this reason, maintenance costs can be lowered by focusing on data durability instead of data availability. Toka et al. [TCDM12] even go further and weaken the definition for durability, which we also adopt in this work:

Definition: Data durability d is the probability to be able to access data after a time window t_{iso}, during which no maintenance operations can be executed.

In our system, we rely on the on-site copy to generate new substreams. This restricts us to perform maintenance only when the swarm leader is on-line. The redundancy level in the swarm therefore decreases whenever the swarm leader suffers a transient or permanent failure. In case of a transient failure, the swarm leader is still alive and, therefore, able to increase the redundancy level as soon as it reconnects to the system again. When the swarm leader suffers a permanent failure, it is not only unable to perform further maintenance, it has also lost the local copy.

Since, in turn, data loss causally leads to a restore operation by the user, we expect the download of the back-up to start within a period we denote as **time to replace** t_r. Within this typically longer period, we expect a user to notice the local data loss, install a new device, and set it back to an operational state.

Further, the **time to download** t_d is the period required to download the back-up from the swarm. In case of asynchronous links, the upload speed of a storage node is typically lower than the download speed of the back-up owner. However, since we can perform multiple concurrent transfers from different storage nodes at the same time, we expect the download speed of the back-up owner to determine the time required for this operation. For this, the data availability within the period t_d needs to be high enough to finish the download, as also discussed in more detail in [TCDM12].

Up to now we have taken t_r and t_d into account, which both focus on providing a user enough time to recover his back-up. In addition, however, we need to consider the **timeout period** t_o used by the failure detector. We tolerate storage nodes to be off-line within this period without considering them dead. As a consequence, in case a storage node permanently fails, it is not considered

dead before the period t_o has passed. This also implies that the lost data due to the undetected permanent failure is missing in the swarm. For this reason, our maintenance procedure can only take into account the redundancy level detected at time $t_c - t_o$, and needs to expect data loss during the period t_o.

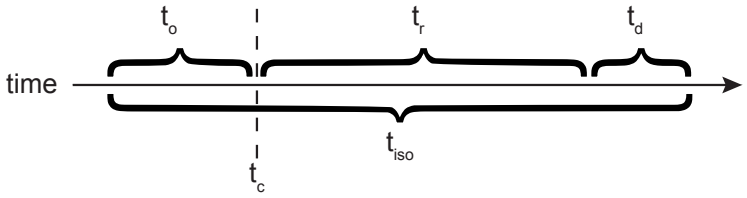


Figure 3.11: Periods Relevant for the Maintenance Procedure

Figure 3.11 provides an overview over the time periods t_o, t_r, and t_d, which in sum result in t_{iso}, which is the total time we want a back-up to survive in isolation from the swarm leader.

By assuming node lifetime values to follow a Poisson process, we can compute the durability resulting from a certain redundancy level and time t_{iso} as follows [TCDM12]:

$$d = \sum_{i=k}^{k+h} \binom{k+h}{i} (e^{-t_{iso}/\tau})^i (1 - e^{-t_{iso}/\tau})^{(k+h)-i} \tag{3.1}$$

Figure 3.12 and Figure 3.13 show the redundancy level $r = \frac{k+h}{k}$ required for a target durability of $d = 0.999999$ and different values for the mean lifetime τ of storage nodes and the number of fragments k in which we divide our back-up.

While in Figure 3.12 we target a period t_{iso} of half a year, we see that the necessary redundancy increases for a t_{iso} of one year, as we show in Figure 3.13. We further see that the redundancy factor in both cases is low especially when $\tau \gg t_{iso}$. High values for parameter k also reduce the redundancy needed in the system, although this effect is negligible for values higher than 100. We discuss the impact of this particular parameter separately in Section 3.6.

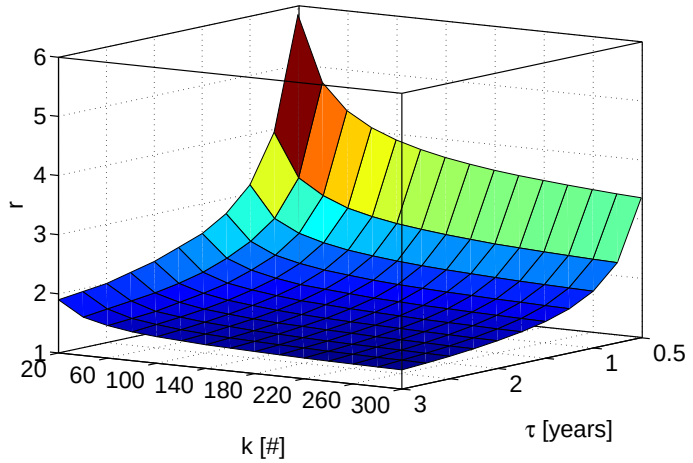


Figure 3.12: Necessary Redundancy r Depending on τ and k for $t_{iso} = 1/2$ year

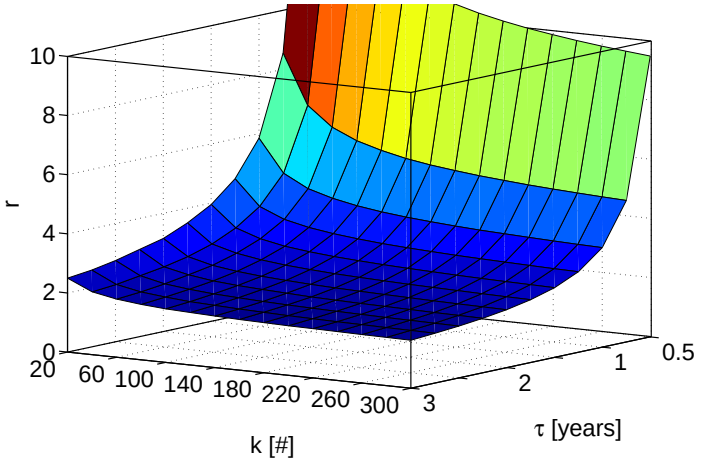


Figure 3.13: Necessary Redundancy r Depending on τ and k for $t_{iso} = 1$ year

In our system, we further use the following approaches to keep the effort for repairs low:

- **Cheap eager and reactive repairs:**
  We use eager and reactive repairs so that we react immediately whenever we observe too little redundancy in the swarm. By holding an on-site copy on the swarm leader, we are able to generate additional redundancy without any prior download from the swarm. This reduces bandwidth costs for repairs to the amount of lost data in the swarm, which is the absolute minimum possible.

- **Benefit from reintegration:**
  When storage nodes transiently leave the system, they still hold their data unless they permanently fail. Weatherspoon et al. [CDH+06] show that the reintegration of reappearing nodes significantly reduces the maintenance costs of the system. To benefit from reintegration in our system, we create substreams with indexes that are new to the system and upload them to new storage nodes instead of re-uploading substreams already known to the system. This also allows us to reintegrate storage nodes that are wrongly suspected to be dead by the failure detector. Thus, in the long term, only permanent failures trigger an upload of additional redundancy.

Finally, we need an algorithm that responds to data loss in the system by the upload of new redundancy. Our algorithm regularly (e.g., once every few hours) monitors the status of storage nodes in a swarm and **adds new storage nodes whenever we observe fewer than n storage nodes that are alive and synchronized**. We explain in the following why we only consider storage nodes in these states.

According to our data placement policy, when a storage node is dead, we are able to exactly determine the lost data by design: when a swarm leader suspects a storage node to be dead, which holds a substream with erasure coding index i, we know that all transmission blocks $T_{f,i} \mid f \in UF$ are missing, which concerns all files stored by the swarm leader. Therefore, instead of monitoring single blocks or files, in our system, it is sufficient to monitor storage nodes and only consider storage nodes that are alive.

In addition, there is the constraint that our repair process needs to consider only storage nodes that are in the *synchronized* state, meaning they hold a transmission block for each file $f \in UF$ stored in the swarm (cf. Section 3.4). We cannot take unsynchronized storage nodes into account because transmission blocks for some files might be missing.

To keep the option for later reintegration of this data, we add a new storage node and use it to store a new substream SS_i with an erasure coding index i that is not yet present in the swarm.

Further, it may happen that a storage node goes off-line while we upload a substream, leading to an interrupted transfer. To facilitate the overall progress in the upload of substreams, we therefore keep a minimum of three parallel uploads active at the same time.

Bandwidth Costs for Repair

The amount of data we need to upload to a new storage node is given by the size of one substream $\frac{S_t}{k}$. Since our maintenance algorithm keeps n alive storage nodes in a swarm, we lose storage nodes with a rate of $\frac{n}{\tau} = n\lambda$. Hence, we can estimate the **average repair bandwidth** BW_r required by a swarm leader:

$$BW_r = \frac{S_t}{k} n\lambda = \frac{S_t}{k} rk\lambda = S_t r\lambda \tag{3.2}$$

Further, given a swarm leader with availability α_1, a storage node with availability α_2, and their common transmission rate BW_c, the expected **time required for a single repair** t_{sr} is:

$$t_{sr} = \frac{\frac{S_t}{k}}{BW_c \cdot \alpha_1 \cdot \alpha_2} \tag{3.3}$$

Bandwidth Costs for Heartbeat Messages

We want the tracker to gather information about the availability of participating gateways (cf. Section 3.2). For this, a gateway sends heartbeat messages of size S_b to the tracker at rate h_t (e.g., once per hour). For N total gateways in the federated network this results in an incoming bandwidth of $BW_t = NS_b h_t$ at the tracker. The corresponding outgoing bandwidth for a swarm leader is $BW_h = S_b h_t$.

In addition, the swarm leader queries the tracker for information about the state of the storage nodes in its swarm. For these queries of size S_q at rate h_q (e.g., once per day) we need an incoming bandwidth of $BW_q = NS_q h_q$ at the tracker. The outgoing bandwidth for responses of size S_r is $BW_o = NS_r h_q$.

3.6 Influence of the Number of Original Fragments

The parameter k determines into how many fragments we divide a file. In this section we discuss the impact of this parameter on our system and give hints for a possible choice.

3.6.1 Storage Overhead

As we see in Figure 3.12 and Figure 3.13, a higher value for k decreases the redundancy factor r. In consequence, we need to store less redundancy on storage nodes for a single back-up, while we meet the same requirements concerning durability. Considering the back-ups of all swarm leaders in the federated network, this leads to an reduction of the overall storage space occupied in the system.

However, high values for k increase the level of data fragmentation. In our scenario, this means we have to distribute a single file among more storage nodes. Each storage node needs to store a file content identifier of fixed size, while the size of the corresponding transmission block decreases. Consequently, higher k increases metadata overhead in the system.

Depending on the particular erasure code used, a higher value for k can also increase the memory consumption for generating erasure coded data. Today, due to decreasing costs for memory, this is less of a problem but can still be an issue in constrained hardware environments.

3.6.2 Data Rates

The parameter k influences several data rates in our system. In the first place, it affects the required bandwidth for repair (see Equation 3.2), which depends on the redundancy factor. Since higher values for k allow us to reduce the redundancy in our system, the necessary repair bandwidth decreases as well.

Further, with increasing k, the size $\frac{S_t}{k}$ of a substream is smaller (cf. Section 3.4). This reduces the amount of data we need to transfer to a single storage node that is added to our swarm. Hence, the duration to transfer a substream at constant bandwidth b is $\frac{S_t}{kb}$ and decreases with higher k.

However, since we have more storage nodes in a swarm for higher k, we also have more storage nodes potentially leaving the system. Therefore, the average rate $n\lambda = (k+h)\lambda$ at which storage nodes leave the swarm increases with increasing k. As in the long term only storage nodes leaving the swarm trigger a repair in our system (cf. Section 3.5), this also causes more frequent maintenance operations.

Another rate influenced by the number of original fragments can be the transmission rate for substreams. Since we address each transmission block by using a file content identifier, each single request for receiving or storing a certain transmission block would require a full round-trip time. However, we show in Section 4.2.2 how we cope with this problem by using aggregates.

Before we upload substreams, in order to ensure consistency (as we explain in Section 4.3.2), we first write them on the swarm leader's hard disk drive. For

higher k, we face increasing delays due to a higher number of local disk seeks for the generation of substreams. Especially for small files this can significantly slow down the process of substream generation. We analyze this problem in Section 4.3 and show how we can prevent low rates via embedding small files.

3.6.3 Bandwidth Saturation

With higher k we increase the number $n = k + h$ of storage nodes a swarm leader maintains in a swarm. Considering a mean availability α of storage nodes, there are $n \cdot \alpha$ different storage nodes on-line on average. A higher number of storage nodes therefore improves flexibility in storage node selection: If certain transfers from or to storage nodes are interrupted, e.g., because they are temporarily unavailable, we can replace these transfers with new ones.

We also illustrate this by means of an example in which we need to download from four different storage nodes in order to recover a back-up. We see in Figure 3.14 that due to the presence of storage node 5 and 6 we can finish the overall transfer within the first two time slots, while we would need four time slots without them.

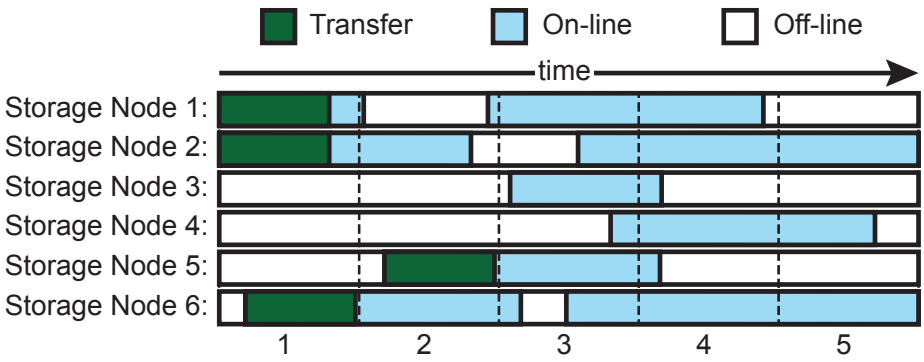


Figure 3.14: Scenario Representing a Back-Up Download

We see that a higher number of storage nodes improves bandwidth saturation and therefore contributes to achieve an overall increase of transfer speed. This is crucial especially when links are asynchronous so that we benefit from performing transfers in parallel [Li13]. Given an average download bandwidth b_d and an average upload bandwidth b_u of gateways in our system, approximately $\frac{b_d}{b_u} \cdot \frac{1}{\alpha}$ storage nodes saturate the download bandwidth in the case of a back-up download. Assuming we achieve the download link of a gateway to be constantly saturated, the time required to download a back-up is $\frac{S_t}{b_d}$, which is the possible minimum. To the end of a back-up download, however, there are typically fewer sources involved in the transfer. Since we need data from distinct storage

nodes in order to recover, the number of eligible sources decreases during the download progress. Therefore, in practice, full bandwidth saturation is difficult to achieve over time, but improves with an increasing number of storage nodes in a swarm. In consequence, we recommend to consider the available bandwidth on gateways for the choice of parameter k.

3.6.4 Effect of Correlated Failures

Our failure model generally assumes permanent failures to occur independently and that inter failure times are exponentially distributed (see Section 3.5.2). In a real world environment, however, this assumption does not necessarily hold.

We divide the possible effect of the parameter k under correlated failures into two cases:

- **Swarm leader is alive**
  In the case the swarm leader is still alive, a higher value for k decreases the effort for repairs under correlated failures. This is due to the fine granularity of the back-up, since it is split into more substreams. When we lose a percentage p_c of storage nodes in the swarm, we eventually upload $\lceil (k+h) \cdot p_c \rceil$ new substreams, each of size $\frac{S_t}{k}$. Since the number of missing substreams rounds up to integers, on average, we save upload bandwidth to compensate for correlated failures when k is higher. In the worst case, a high number of correlated failures results in the situation where the swarm leader cannot complete the maintenance procedure for a long period because the upload bandwidth capacity is fully occupied.

- **Swarm leader is dead**
  As soon as the swarm leader is dead, by design, our system cannot guarantee durability of data under correlated failures. However, a higher redundancy factor r generally increases the share p_c of storage nodes we are able to lose due to a correlated failure, and yet, still have more than k storage nodes left to recover the back-up. Therefore, although our system allows to use a lower redundancy factor r when k is higher, it can be advantageous to use a higher redundancy level in order to tolerate a certain degree of correlated failures.

In Chapter 5 we further analyze the presence of correlated failures in an environment that is close to the one we focus on.

3.6.5 Load on the Tracker

The tracker receives heartbeat messages at a constant rate per on-line gateway. The effort for processing these increases linearly with the number of gateways.

Further, for each swarm leader we store a copy of its swarm set on the tracker. Since the size of the swarm set depends on k, this parameter also influences the storage space required on the tracker.

Each update of the swarm set requires an update on the tracker. The frequency for such updates is given by $N(k+h)\lambda$. For each update the tracker needs to select a new storage node, return it to the swarm leader, and update the existing swarm set. This workload involves several disk seeks and messaging effort.

Numerous disk seeks can lead to lower throughput and higher queuing delay and, hence, restrict the scalability of a system [MDWS10]. Since this is crucial for our tracker, we separately address its scalability in Section 4.4.2.

3.7 Encryption

When users perform an off-site back-up, data security is a strong requirement. Participants in the network are required not to have access to the data of others. The following cryptographic concepts [FSK10] help achieve this requirement:

- **Authentication**
  It is required to ensure that participants in the network do not impersonate others. We need to make sure that only data owners are allowed to modify their data. In general, a public-key infrastructure can meet this requirement.

- **Data Confidentiality**
  Data stored on a gateway is private to the user and is required to remain private. Therefore, encryption can be used to achieve data confidentiality.

- **Data Integrity**
  This denotes the property of stored data to be consistent over time. Unauthorized modification of stored data must be detected. However, stored data can alter due to malicious attacks or hardware failures. Typically, hashes are used to perform integrity checks.

In order to create and reconstruct off-site back-ups in our system, a user needs to keep two pieces of information:

- The public gateway identifier Id_{gw}, provided by the tracker when joining the federated network.

- A secure [DMR10b] and confidential password, chosen by the user.

In addition, we store a randomly generated bit sequence, referred to as a *salt*, on the tracker. The gateway uses this salt for hash computation so that no other participant produces an equal result for hashing the same input. This hardens computed hashes against attacks, e.g., using so-called rainbow tables, which hold precalculated hashes for certain input data [TP11].

Below we explain how our system leverages the aforementioned cryptographic concepts to achieve our requirements on the system.

3.7.1 Authentication

For authentication, we use a Public-Key Infrastructure (PKI) [FSK10] with a Certification Authority (CA). To increase modularity and lighten its load, we allow this CA to be physically separated from the tracker. We show an overview of the resulting setup in Figure 3.15, which we further explain in the following.

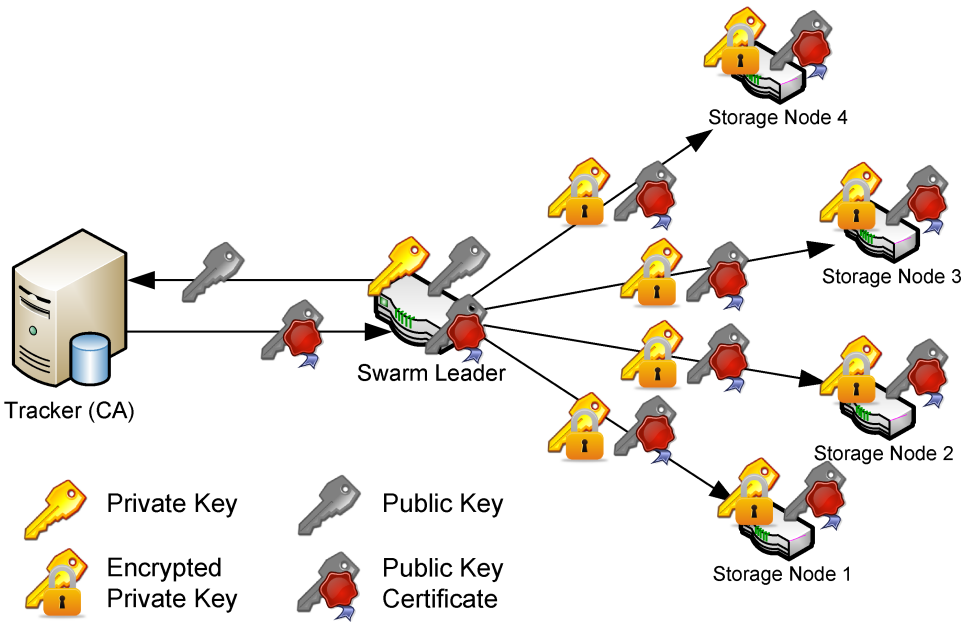


Figure 3.15: Handling of a Swarm Leader's Asymmetric Keys

The swarm leader creates an asymmetric key pair, which consists of a **private key** K_s^{SL} and a **public key** K_p^{SL}. As typical for asymmetric encryption, the private key is required to remain secret, known only to the owner, while the public key is publicly known. Because of this, we upload the public key to the CA, which for its part also has an asymmetric key pair consisting of the private key K_s^{CA} and the public key K_p^{CA}.

The CA approves that the gateway identifier Id_{gw} of the swarm leader is bound to the given public key by creating a **public key certificate**. This certificate contains the public key K_p^{SL}, the gateway identifier Id_{gw}, and a signature $Sig(K_p^{SL}, Id_{gw})$. The CA determines this signature by encrypting the hash of the gateway identifier and the public key using its private key:

$$Sig(K_p^{SL}, Id_{gw}) = \{H(Id_{gw}, K_p^{SL})\}_{K_s^{CA}} \tag{3.4}$$

Since only the CA can generate this signature, this proves that the CA is aware that the particular gateway operates in the federated network using a particular gateway identifier. This empowers the CA with global knowledge of participants in the federated network and allows it to revoke certificates so that participants can be excluded from the system. In contrast to an authentication protocol such as Kerberos [SNS88], which relies on continuous availability of an authentication server, we are able to tolerate the CA to be unavailable. In such case already known gateways can still operate in the federated network, while new gateways cannot join anymore. Key revocation in our system requires the CA to actively distribute signed notifications to storage nodes in the swarm concerned by the revocation. This, in contrast to the protocol used in Kerberos, introduces the possibility for a man-in-the-middle attack, where an attacker could simply drop such notifications. Due to the numerous storage nodes in a swarm and their geographical distribution, however, we consider such scenario to be rather unattractive for an attacker.

After creation of the public key certificate, the CA sends it to the swarm leader, which hereupon distributes it to storage nodes on its own. The fact that we do not require the CA to distribute public key certificates reduces messaging complexity at the CA from $O(Nn)$ to $O(N)$.

In consequence, a storage node can check the certificate of a swarm leader before accepting any requests. By encrypting all further communication (see Section 4.2.1 for details), we ensure confidentiality, integrity and authenticity for communication within the federated network [All10]. This prevents man-in-the-middle attacks, where an attacker relays and potentially modifies communication between two participants who believe to communicate directly with each other over a private connection.

However, when a swarm leader permanently fails and suffers data loss, the private key on the swarm leader will be lost as well. In consequence, we need to make sure a user can recover the private key from a different place. Hence, we use a Password-Based Key Derivation Function (PBKDF) [RSA13] to derive a **symmetric key** K_{symm}^{SL} from the password a user needs to memorize. To avoid password collisions for different users we add a random salt which we associate with Id_{gw}. We use K_{symm}^{SL} as input for a symmetric encryption algorithm in order to encrypt the private key K_s^{SL}.

Thus the **encrypted private key** K_e^{SL} is derived as follows:

$$\begin{aligned} K_{symm}^{SL} &= PBKDF(password, salt) \\ K_e^{SL} &= \{K_s^{SL}\}_{K_{symm}^{SL}} \end{aligned} \tag{3.5}$$

Finally, we store K_e^{SL} on each storage node in relation to the public gateway identifier together with the random salt.

Henceforth, the download of K_e^{SL} is possible to everybody without any restriction. Only via knowledge of the user password, however, is it possible to decrypt the private key, and thus, recover a previous identity in the federated network again. The choice of a secure user password therefore is crucial in our system.

3.7.2 Data Encryption

We use a symmetric-key algorithm for file content encryption so that keys for encryption and decryption are the same. In order to decrease the overall number of keys, we encrypt files before splitting them into fragments. This also improves performance since the encryption algorithm, which needs to initialize first, runs on a larger portion of data. We further differentiate two different cases regarding encryption:

- **Encryption of files in the file set**
  For file contents referenced by a snapshot we use the file contents hash to derive a **file encryption key** $K^f = H(f)$. This procedure is known as convergent encryption [SGLM08] and allows us to obtain the same binary representation of the cipher text when two files that have identical content are encrypted. This allows us to deduplicate file contents within single snapshots as well as over different snapshots. We keep all encryption keys to decrypt referenced files in the index file. As already mentioned before, the encryption key also allows us to derive the file content identifier by calculating its hash $Id_f = H(K^f)$.
- **Encryption of the index file**
  We use a key derivation function to derive a symmetric key from a gateway's private key. We use this symmetric key to encrypt the index files. In consequence, the index file can only be recovered with knowledge of the private key K_s^{SL}.

Using this approach, we achieve data confidentiality for index files and all files in the file sets. We require the knowledge of the password to decrypt an index file, which further includes all keys to decrypt referenced files.

Further, a single storage node only receives a share of the encrypted data so that it never possesses the whole encrypted back-up. Even if an encryption standard

should become insecure in the future, the data stored on a single storage node will not be sufficient to recover parts of the original data.

3.7.3 Integrity Checks

Data on storage nodes may change at any time, either due to malicious behaviour or hardware errors[4]. Further, it is not easy to locate these changes because they can be distributed over the whole data set.

However, since the swarm leader holds an on-site copy of all files, we can use a standard challenge-response protocol as described in [DQ04]. As challenge, the swarm leader sends a random string to the storage node. The storage node computes the response as the hash over the challenge concatenated with a transmission block's content. If a storage node does not have the correct binary representation of a transmission block available, it is not able to answer the challenge by the swarm leader correctly.

Using the hashes we store in an index file as encryption keys, we are also able to check for corrupted files on the gateway. For this, we compute the hash over the file and compare it to the hash stored in the index file. If they differ, either the index file or the local file is corrupted. We can easily identify the corruption by downloading the affected files from the swarm again.

The hashes in the index file are also important to confirm the integrity of a file downloaded when it comes to back-up recovery. A storage node could modify the binary representation of transmission blocks, but it cannot update the hashes stored within the encrypted index file. Therefore we can use a trial and error approach that excludes a particular storage node from the recovery process until the integrity of the restored data can be confirmed. Such an approach was also used in the distributed version of Wuala [TMEBPM12].

3.8 Conclusion

In this chapter we introduced our distributed back-up architecture. In our scenario, we benefit from an on-site copy that allows us to add additional redundancy at low cost. Further, we reintegrate reappearing storage nodes so that in the long term the system only suffers from permanent failures. We introduce swarms, which are easy to monitor: swarms allow swarm leaders to act in isolation of others so that we support concurrent access on the storage provided by

[4] In 2008, for example, the Amazon S3 storage service was down due to a single bit error propagating in the system, cf. the report under http://status.aws.amazon.com/s3-20080720.html.

participants. In combination with our data placement policy, we reduce metadata necessary for data localization to a very low level. In fact, once an index file is recovered, no additional lookups are needed to locate all referenced files.

We use a tracker to keep condensed information about the state of the system. However, we only impose little management effort on this central instance. Especially information about specific files is completely hidden to the tracker in order to reduce its load. We show that the usage of such central node entails several advantages over a decentralized approach using a DHT among participants. It supports us to enforce incentives in the system, provides an efficient way for authentication, and improves performance since we spare expensive protocols for consensus among participants. However, we do not rely on the tracker to store data which could undermine data confidentiality of data stored by a user.

We further introduce index files, which are, to the best of our knowledge, the first approach for an encapsulated data structure that supports the creation of distributed back-ups with support for snapshots. These index files allow us to consistently represent snapshots from local to distributed environment. With respect to data confidentiality index files also allow us to embed the keys used for encryption.

With this architecture in mind, we provide more details about our implementation in the following chapter.

Chapter 4

Implementation

> "If it is good for everything, it is good for nothing."
>
> Michael Hammer,
> Computer Industry Consultant

4.1 Introduction

In this chapter we provide more details on our implementation. This comprises the communication between participants as well as key features of the entities in our system. Our implementation is publicly available [Tho14], including its Java source code.

In the following we first show how participants communicate and how we transfer transmission blocks between gateways. Then we explain why the modularization of the swarm leader is key to a reliable back-up processing in case of an interrupted execution. Since files in our system require different handling, we also introduce different classes of files and explain their characteristics. We also illustrate how the tracker can still be operational despite single failures and how to replace it in case of total failure. Regarding storage nodes we show how storage space is freed without additional communication overhead. Finally, we look at some incentives for participants in order to support correct behaviour in the federated network.

4.2 Communication

In this section we focus on the protocols we use for communication between participants in the federated network. This includes taking a close look at the communication from storage nodes to the tracker and the communication between storage nodes.

4.2.1 RESTful Architecture

We deploy our system by using a RESTful architecture. This means participants communicate in the federated network by performing CRUD[1] operations on resources that are either located on storage nodes or on the tracker. RESTful services use the HTTP to send PUT, POST, GET, and DELETE requests on resources which are identified by a Uniform Resource Identifier (URI). Like HTTP, RESTful services are stateless, so that no state maintenance across request-response pairs is required [KR01]. This improves scalability and is the reason why we choose RESTful services instead of stateful web services[2] based on Remote Procedure Call (RPC), such as Simple Object Access Protocol (SOAP).

The HTTP methods PUT, GET, and DELETE are designed to be *idempotent*, so that they can be applied more than once and yet always lead to the same result for a given input [All10]. Idempotence further requires the implementation of a function to be *reentrant* [SR05], which allows the function to be interrupted and a new instance to be executed without being affected by a previous execution. Reentrancy can be achieved by meeting the following requirements [Cor13]:

1. A reentrant function does not hold static data over successive calls, nor does it return a pointer to static data.
2. All data is provided by the caller of the function.
3. A reentrant function must not call non-reentrant functions.

When idempotence is combined with retry, a workflow based on a fault tolerant composition of atomic actions can be built. Such workflow does not require [RV13] distributed coordination so that we can avoid overhead for distributed algorithms such as two-phase commit[SK09]. As a result, whenever, e.g., a swarm leader temporarily fails, it can resume its execution by restarting open tasks again.

Because HTTP relies on the Transmission Control Protocol (TCP) for data transmission, our application does not need to provide a reliable, error-checked,

[1] CRUD refers to the basic functions of persistent storage: create, retrieve, update, and delete.

[2] A more detailed comparison can be found in [FSF09].

OSI Layer	Protocol
7 - Application	HTTP
6 - Presentation	TLS
5 - Session	TCP
4 - Transport	TCP

Figure 4.1: Communication Layers According to the OSI Model

and ordered delivery for data transfers on its own (in contrast to the hybrid version of Wuala, as shown in the following box). In addition, we layer HTTP requests over the Transport Layer Security (TLS) protocol [IET13] to encrypt communication in the federated network (cf. Section 3.7.1). In Figure 4.1 we show the host layers of our environment according to the Open Systems Interconnection (OSI) model [Sta97]. In our back-up system, we store all back-up data on storage nodes within the swarm leader's swarm set, which is independent of the number of files and only involves a limited subset of the federated network. This property enables us to reuse established TCP connections for the transfer of transmission blocks belonging to different files[3]. Connection re-usage also allows to reuse temporary encryption keys for successive requests. Key negotiation using, e.g., Diffie-Hellman [FSK10] leads to additional overhead for opening new connections and can be reduced this way.

[3]By using persistent connections via the keep-alive keyword of HTTP 1.1 [KR01].

Side Glance: Why the Hybrid Version of Wuala Used UDP

The hybrid version of Wuala [TMEBPM12] leveraged storage provided by participating peers, as we do in our system. However, in contrast to our system, Wuala was never designed just to be a distributed back-up system. It also supports file sharing and live streaming of video data. As another difference to our system, data of each file in the Wuala system was stored on a different set of storage nodes.

In an environment in which storage nodes may disappear at any time, these features benefit from a quick alternation of sources for transfers and, thus, from low latency in particular. Low latency is important to achieve continuous video streams and fast consecutive transfers of multiple files from different sources.

For this reason, Wuala used an own pull-based protocol for data transfer, which is based on User Datagram Protocol (UDP). UDP per se does neither require handshakes, nor acknowledgements, nor does it use retransmissions like TCP and, therefore, is generally better suited for features like video streaming.

When downloading bigger files, the Wuala client initiates more than 100 transfers from different sources in parallel. This requires congestion and flow control, which is supported by Wuala's protocol implementation. Wuala used the size of 1024 bytes for one transmission object, called a coding fragment. Such a coding fragment is delivered in a single datagram with the "don't fragment" Internet Protocol (IP) flag [IET14b] enabled. In consequence, the delivery of coding fragments is atomic. A receiver indicates missing coding fragments of a file to the sender by using a bitmap in combination with an offset. This allows to request specific coding fragments, regardless of whether the sender has already sent them.

In contrast to Wuala, for our back-up system as such, it is not important that a file can be reconstructed on the fly while it is being transferred. We therefore do not target transfers from a high number of storage nodes in parallel. Further, we store data for all files on the same storage nodes so that, in general, transfers between participants in our system are longer lasting than transfers in Wuala. It is because of these differences that the use of UDP is less attractive in our system.

4.2.2 Aggregates

As described in Section 3.4, we divide a file of size S_f in our system into k transmission blocks of size $\lceil S_f/k \rceil$. Subsequently, we transfer these transmission blocks to storage nodes. However, for files of small size, transmission blocks may only have the size of few bytes. Sending an HTTP request for each transmission block of such limited size is inefficient because of, e.g., the HTTP header information which is sent each time. Thus, apart from a different treatment for small files (which we will introduce in Section 4.3.3), we want to decrease the protocol overhead for small transmission blocks.

For this reason, we group several transmission blocks into aggregates, as suggested in [All10]. We assemble an aggregate on the sender side and transfer it via a single HTTP request to the receiver. For each transmission block within an aggregate, we store, in binary form, its corresponding file content identifier, the length of the transmission block in bytes, and its content. We put transmission blocks with a size smaller than 256 KiB into an aggregate. Below this size, data can typically be stored more storage space efficient in a BLOB (such as our aggregates) than compared to the file system [SVIG06]. Since a storage node buffers aggregates on its local hard drive before transmission, this approach saves disk space. As soon as the size of an aggregate exceeds 1 MiB, we start a new aggregate. This way we limit the number of files that are affected in case an aggregate cannot be reconstructed (e.g., due to corrupted data on a storage node). We illustrate the resulting structure of aggregates in Figure 4.2.

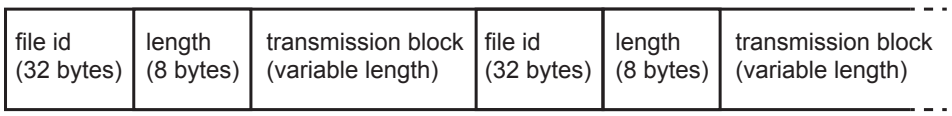


Figure 4.2: Aggregate Structure

At reception, the receiver disassembles the aggregate again. Since the content is a field of variable length, the receiver uses the length information to delimit a transmission block's content within the aggregate. As a result, the receiver can separate each transmission block and store it in relation to its file content identifier.

4.2.3 Partial Transfers

We provide a resumable data transfer after a communication failure has interrupted the data flow. Especially for the longer lasting transfers of bigger transmission blocks, this feature is useful since the likelihood increases that one

of the participants goes off-line. By resuming transfers, we profit from already transferred data and save time and bandwidth for retransmission.

The possibility of partial transfers also allows us to selectively request missing data when it comes to file reconstruction. In Section A we will introduce the interleaving scheme we use to create transmission blocks for bigger files. The interleaving scheme allows us to already reconstruct parts of a file before we have received k complete transmission blocks. In consequence, when a storage node turns off-line while it is sending a bigger transmission block, the data already transferred can still be used for file reconstruction. Another storage node which replaces the former source only has to send the missing part of its transmission block to recover the file.

We implement the partial transfer using the `Content-Range` header field[4] supported by HTTP 1.1. The field allows us to specify a start and end position in bytes in order to indicate that only data within this range is transferred.

4.3 Swarm Leader

In general, when organizing computer systems, modularity is an important property to reduce system complexity. It leads to less time spent debugging an implementation and facilitates system improvement and evolution over time [SK09]. Therefore, in this section, we describe how we modularize the swarm leader, which is responsible for both, on-site and off-site back-up creation. Consequently, we divide its functionality into two main modules, one for the **on-site back-up** and another for the **off-site back-up**. We isolate the functionality of these modules from each other and limit their interaction on snapshots which we store on the local file system of the gateway. Figure 4.3 illustrates the data flow between the two modules.

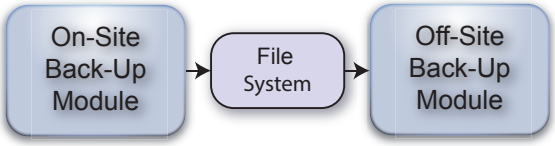


Figure 4.3: Data Flow Between Modules for On-Site and Off-Site Back-Up

Both these modules store files on the gateway and use the following folders for dedicated purposes:

- **On-Site Copy folder (`on-site copy`)**
  Here we keep a single folder for each snapshot, named by its time of cre-

[4] See RFC 2616 (`http://www.ietf.org/rfc/rfc2616.txt`) for more details.

ation. Within such a folder, we store the whole folder structure and file contents of the snapshot so that we obtain the structure for on-site back-ups, as introduced in Section 3.3.1. Data within this folder is immutable.

- **Index Files folder (`index files`)**
  In this folder we keep one immutable index file per snapshot.

- **Temporary folder (`temporary`)**
  This folder contains work in progress. This includes index files or transmission blocks that are currently in creation.

- **Upload folder (`upload`)**
  The upload folder holds immutable transmission blocks which are scheduled for upload to storage nodes.

Within the off-site and on-site back-up modules we have further sub-modules which we illustrate in the following sections.

4.3.1 Modules for On-Site Back-Up

In this section we describe how the on-site back-up is modularized and implemented. The goal of this module is to create an on-site back-up on the gateway within the home network. To achieve this, we have to access files stored on user devices so that we can finally synchronize them to the gateway. We divide the functionality of the on-site back-up module into two sub-modules which we further explain in more detail.

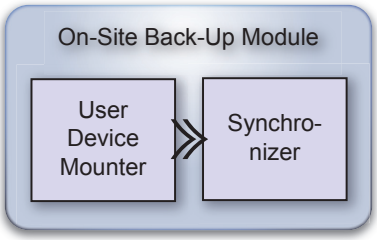


Figure 4.4: Sub-Modules for On-Site Back-Up

User Device Mounter Module

The data we want to back-up is stored on heterogeneous user devices within the home network. We need to expect that these devices offer different interfaces

for file access by using protocols such as NFS, Server Message Block (SMB), File Transmission Protocol (FTP), HTTP, or others.

However, we want to avoid that users need additional software on each device in order to create back-ups. This is why we put all functionality concerning data access on the gateway, as proposed by Defrance et al. [DGLR+11]. They propose to use a Virtual File System (VFS) [Bar01] as an abstraction layer on the gateway. The VFS provides a unified interface for file access on devices within the home network, while device dependent protocols are hidden to the application. The resulting setup is shown in Figure 4.5.

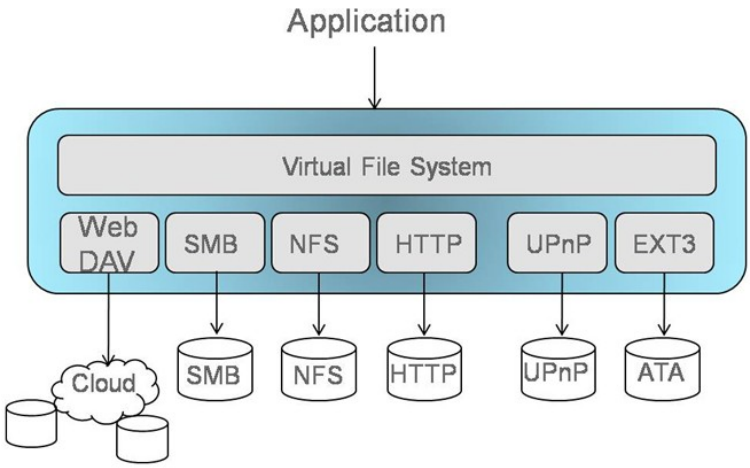


Figure 4.5: Virtual File System for Unified File Access [DGLR+11]

The user device mounter is also in charge of detecting user devices to join or leave the home network. Depending on this, it performs a mounting or unmounting operation, respectively.

Synchronizer Module

After the gateway mounts a user device, we are able to access and transfer files from it. We use the file-copying tool Rsync [Tri96] to create our on-site copy on the gateway. Rsync is famous for its delta-transfer algorithm, which allows to reduce the amount of transferred data to the difference between source files and existing files at the destination[5].

Before Rsync copies a file, it checks whether it is already present in the previous

[5]The man page of Rsync is available at http://www.samba.org/ftp/rsync/rsync.html

snapshot. Given this is the case, it creates a hard link[6] to the file already stored on the gateway and skips the file transfer[7].

Therefore, instead of transferring and repeatedly storing all file contents, we use **hard links to deduplicate data** between the new snapshot and the last snapshot already present on the gateway.

We call the following command to transfer changed data to the gateway:

```
rsync -rpEgotc --link-dest=$PREV $SRC $DEST
```

where `$PREV` is the directory of the last snapshot previously taken, `$SRC` is the directory on the user device, and `$DEST` is the destination for the new snapshot.

As destination we use the `temporary` folder on the gateway. It may happen that the synchronization is interrupted, e.g., because a user device disconnects from the home network before the process completes. In this case we can restart the synchronization at a later time without side effects. Only after the synchronization has finished successfully, do we move the snapshot to an individual folder on the gateway, named after the current timestamp.

Hereupon, the snapshot is completely synchronized with the gateway and available for on-site recovery in case a user device fails.

If in addition we want to schedule the current snapshot for off-site back-up, we move it in an atomic operation to the `on-site copy` folder using the UNIX `mv` command[8]. As a result, we never see partial snapshots in the `on-site copy` folder, which thus contains only **immutable** data. We therefore use the `on-site copy` folder as an interface to push snapshots from the on-site back-up module to the off-site back-up module and, thus, decouple their functionality.

The output of Rsync provides a log concerning files which have changed for the new snapshot. From this log we can infer files which are used by the previous snapshot but not by the new snapshot anymore, as well as files which are added by the new snapshot. We keep this log for later usage when it comes to off-site back-up creation.

4.3.2 Modules for Off-Site Back-Up

In the previous section we have shown how snapshots are passed in an atomic operation to the off-site back-up module. Starting from the moment a new snapshot is visible to the off-site back-up module, its initial upload to storage nodes as well as its future maintenance need to be ensured.

[6] Hard links are supported by POSIX compliant (or partly compliant) operating systems and can be used to create new directory entries for already existing files. See man page, available at http://www.unix.com/man-page/POSIX/3/link/.

[7] Some file systems, such as Btrfs, offer additional deduplication on block granularity, cf. https://github.com/g2p/bedup.

[8] For this operation we require both directories to be located on the same file system.

In fact we have two situations in which the off-site back-up module needs to create new redundancy. Given we have a swarm which already stores a set of transmission blocks for its back-up, this set extends in the following cases:

- **New Substreams**
  We create transmission blocks for each file already present in the system.
- **New Snapshots**
  We create transmission blocks for all new files referenced by new snapshots.

Therefore, in the off-site back-up module we have **two triggers** which lead to the creation of new transmission blocks. The first trigger is the **maintenance module**, which creates a complete substream for a new storage node in the swarm. The **snapshot creator module**, on the other hand, triggers the creation of new transmission blocks when a swarm leader adds a new snapshot.

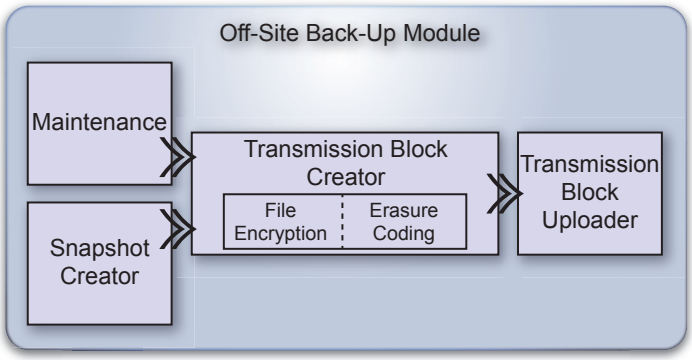


Figure 4.6: Sub-Modules for Off-Site Back-Up

Since both these modules launch the creation of transmission blocks, we use a uniform **transmission block generator module**. This module internally performs encryption and erasure coding and passes created transmission blocks to the **transmission block uploader module**. We outline this processing chain in Figure 4.6.

The fact that we have two triggers operating on a common data set leads to a situation in which we need to address concurrency. As we illustrate in Figure 4.7, a concurrent execution of both modules may lead to transmission blocks missing in the intersection set of their output.

To cope with this problem, we can either use interprocess communication or make sure that both modules never run at the same time. In our implementation we choose the latter since both modules work on the same hard disk drive but on different files, so that concurrent access leads to more disk seeks, and thus,

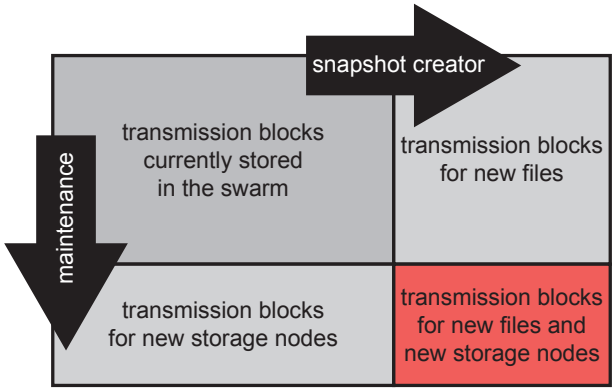


Figure 4.7: Overlap in Creation of Transmission Blocks

to an overall slowdown of the creation process of transmission blocks. As we see later in Section 4.3.2, disk I/O is an important performance factor concerning the creation of transmission blocks. Besides, avoiding concurrent execution of both modules reduces software complexity and, in practice, can improve software reliability due to fewer bugs caused by concurrent execution [FLSR10].

Referential Integrity for Snapshots

We need to expect a gateway to transiently fail at any point in time due to hardware failures, power cuts, or comparable events which lead to an interruption of the swarm leader. However, it is important that failures of pending processes do not lead to inconsistency in the long term. Otherwise, interruptions could lead to the following problems:

- **Incomplete snapshots**
  The process creating new snapshots is interrupted, so that transmission blocks for added files are missing without notice. When it comes to snapshot recovery, certain files referenced by such snapshots can be impossible to recover. Further, the tracker needs to hold a reference to uploaded snapshots so that the recovery process can find a corresponding index file.
- **Incomplete maintenance**
  The maintenance process is interrupted and does not successfully terminate. This could lead to storage nodes lacking transmission blocks of already uploaded snapshots. In consequence, files of several snapshots could be impossible to recover. Additionally, the tracker needs to know about new storage nodes used, so that the swarm list is up to date when it comes to recovery.

In the area of database system design, the property that references only point to existing data is referred to as *referential integrity* [EN03] and guaranteed by the consistency property of databases[9]. For this purpose, database systems allow to define *integrity constraints* for data sets. Whenever a transaction, which consists of several operations, succeeds, these integrity constraints are guaranteed to be fulfilled by the system. However, such constraints typically entail high costs for cross checking and, hence, do not scale [Ora13].

Considering our system, the answer to the problem is to divide our workflows for snapshot upload and maintenance into a fault tolerant composition of single atomic actions. Each such atomic action is applied at-least-once [SK09] so that we have to make sure the repetition of a request does not result in side effects. When combined with retry, such workflow also allows to avoid [RV13] distributed coordination protocols such as for example two-phase commit [SK09]. Further, since our architecture operates on *immutable files*, we do not face problems related to read/write coherence. Read/write coherence means that the result of a read of a particular data object is always the same as the most recent write [SK09]. As in our system we derive all file content identifiers from the files' content, we never access different content under a particular file content identifier.

Finally, to support resuming the workflow at any point in time, we use a composition of atomic actions, each of which needs to comply with the following scheme:

- **Precondition**
  A certain precondition needs to be met. Only in this case the action is triggered.
- **Action**
  The particular action to be performed. We require the action to allow being applied at-least-once without any occuring side effects.
- **Postcondition**
  After the action has been performed at-least-once, eventually, the postcondition is met. This includes that the precondition is not fulfilled anymore.

Subsequently, we provide more details on the individual sub-modules for the off-site back-up. We explain the resulting composition of atomic actions, realized by using RESTful web services. Since we process the upload of a new snapshot and the maintenance procedure as such a composition, referential integrity can be transiently violated. However, due to the retries we use, references eventually point to existing data only.

[9] As part of the ACID properties: Atomicity, Consistency, Isolation, Durability; cf. [EN03]

Maintenance Module

When the redundancy level in the swarm drops below a threshold, the maintenance module is in charge of repair. As mentioned before, the maintenance module runs only once at the same time and never concurrent to the snapshot creator module.

To support resuming an interrupted execution of the maintenance module, we divide its task into three phases in accordance to the scheme introduced in Section 4.3.2. Each phase supports reentrancy and ends with an atomic operation[10] to prove its termination. The detailed algorithms for these phases are illustrated in the Appendix, Section B.1. In the following we outline their basic functionality.

Phase 1: Generate Unassigned Substreams

In this phase we generate a new substream in a temporary folder on the swarm leader. For this, we create a transmission block for each unique file content by passing through all the snapshots already stored in the swarm. The maintenance algorithm (introduced in Section 3.5) may require to add multiple substreams at once. Hence, we initialize the erasure coding with at least one index in order to create a corresponding transmission block per file content. The process of substream creation can be time-consuming. Therefore, in case this phase turns out to be a bottleneck, we can trade local storage space and run the procedure in the background before the swarm leader is in need for a new substream.

Phase 2: Assign Substreams to Storage Nodes

We need to assign the previously generated substreams to new storage nodes. On request, the tracker provides an identifier of a new storage node within the federated network. The swarm leader persists this information by atomically renaming the folder of the corresponding substream. Hence, this phase only succeeds once in assigning a substream. This is important, since the tracker returns a random result each time we request a new storage node.

Phase 3: Include Storage Nodes in Swarm List

After a substream is locally assigned to a storage node, we update the swarm set stored on the tracker. Only after successful synchronization do we move the substream into the upload folder so that the upload to the storage node begins. In consequence, no data is stored on storage nodes before the tracker keeps a copy of the swarm set, which can be recovered in case of data loss on the gateway.

After the last phase terminates with success, the transmission blocks in the upload folder eventually are uploaded to the targeted storage nodes.

[10] Using the ATOMIC_MOVE argument for the move command provided by JDK 7, see http://docs.oracle.com/javase/tutorial/essential/io/move.html for more details.

Snapshot Creator Module

The snapshot creator module is in charge of creating a new snapshot to be stored in the swarm. This includes two main tasks:

1. The creation of an index file which holds references to all the file contents referenced by the snapshot.
2. The generation of transmission blocks for file contents referenced by the new snapshot but no previous snapshot stored in the swarm. This also includes the transmission blocks for the index file.

We keep all relevant information of an index file (cf. Section 3.3.2) in a `tar`-file. This increases interoperability since an index file can also be read outside our application in case of a software error. Originally used for tape back-ups, the `tar` file format[11] allows to pack any number of files into a continuous stream of bytes.

According to the POSIX compliant format from 1988, each file content is preceded by a header record of 512 bytes, which allows to keep file metadata like the following stored in a file system[12]:

- file name (including path to reconstruct folder structure)
- access permissions
- file owner id
- file group id
- last modification date
- link information (for hard links or soft links)

The `tar` file format is designed to hold file contents as well. As we show in Section 4.3.3, we use this possibility to store files of small size close to their metadata. In accordance with our general data placement policy (Section 3.4), however, we generally want to store file contents in a distributed way. For this reason, instead of storing the file content in the `tar` file, we store the data listed in the following:

[11] The `tar` archive format is described under `http://www.freebsd.org/cgi/man.cgi?query=tar&sektion=5`.

[12] More recent but not POSIX compliant versions such as the GNU tar format also allow to keep the time when a file was last accessed (`atime`) and the file creation time (`ctime`).

- **file encryption key (32 bytes)**
  In order to decrypt a particular file, we need to know the key K^f used for encryption. Further, by hashing the file encryption key we can derive the file content identifier $Id_f = H(H(f)) = H(K^f)$, which is needed to query the fragments of the particular file.

- **file length (8 bytes)**
  For erasure coding and encryption we need to pad a file until we reach a multiple of the block size used. In order to crop the file again, we need to know its original size.

To keep this metadata apart from real file content, we check entries within the `tar` archive for having the size of $32 + 8 = 40$ bytes, and whether the resulting file content identifier can be used to download transmission blocks from the swarm. If neither is the case, we interpret the content within the `tar` archive as file content.

As a result, an index file contains all data necessary to recover the original state of a file tree, covering file metadata and file contents.

After creation of an index file, we compress it using `gzip`[13] to reduce its size. Because the `tar` format pads empty fields with null values, the compression significantly reduces the size of our index files (typically to a size smaller than 10% of its original size).

In the following we describe the phases we use to create an index file. Each of these phases ends with an atomic action to indicate its termination. Thus, we can restart each of these phases individually. For further details on these algorithms we refer to the Appendix, Section B.2.

Phase 1: Parse New Snapshot

This phase starts as soon as a snapshot is passed from the on-site back-up module to the off-site back-up module. We generate the index file by traversing the file tree of the snapshot. For each file we check whether it is already present in a previous snapshot and, if this is not the case, create corresponding transmission blocks for it. We compress an index file after creation to reduce its size.

Phase 2: Generate Transmission Blocks for Index File

The index file created in the previous phase needs to be stored in the swarm as well. Therefore, in this phase, we create transmission blocks for the index file. In case the index file is too small to be uploaded into the swarm (we will explain this case in Section 4.3.3), we pad it to the minimum size.

[13] See the `gzip` man page under `http://www.freebsd.org/cgi/man.cgi?query=gzip`.

Phase 3: Update Snapshot Metadata on Storage Nodes In this phase we send metadata about the snapshot to every storage node in the swarm. This metadata includes the following:

- a unique **snapshot identifier** v_t, indicating its **creation time**.
- the **file content identifier** Id_f of the **index file**.
- all **unused file content identifiers** to file contents $OF = FS_{t-1} \setminus FS_t = \{f \in FS_{t-1} \mid f \notin FS_t\}$ used by the previous snapshot v_{t-1}, but not by the current snapshot v_t anymore.
- all **new file content identifiers** to file contents $NF = FS_t \setminus FS_{t-1} = \{f \in FS_t \mid f \notin FS_{t-1}\}$ used by the new snapshot v_t, but not by the previous snapshot v_{t-1} before.

The first two items are required in case of recovery, while the latter are used for storage reclamation on the storage node.

In case a storage node is not available, we retry the submission at a later point in time, until success.

Transmission Block Creator Module

The transmission block creator module is either triggered by the maintenance module or the snapshot creator module to generate transmission blocks for a particular file. Within this module we first encrypt the file content of the on-site copy and, thereafter, perform erasure coding. Finally, we atomically move finished transmission blocks into the folder structure used by the transmission block uploader module (which we show in Figure 4.9). We illustrate the parameters and the data flow for transmission block generation in Figure 4.8 and describe its composition in the following.

File Encryption In our implementation for file content encryption, we use state of the art cryptographic algorithms [FSK10]. This is Advanced Encryption Standard (AES) with 256 bit keys for file content encryption and the Secure Hash Algorithm (SHA) with 256 bits as cryptographic hash function[14]. As described in Section 3.7.2, we derive the keys used for symmetric encryption either from the file content (in case of a file within a snapshot) or from the gateway's private key (in case of an index file). Since AES operates on blocks, it requires input data to be aligned to a multiple of 16 bytes. Further, we operate AES in Cipher-Block Chaining (CBC) mode, which connects a previous

[14] In fact, we always compute SHA-256(SHA-256(x)) instead of SHA-256(x) to prevent length extension attacks as described in [PO95].

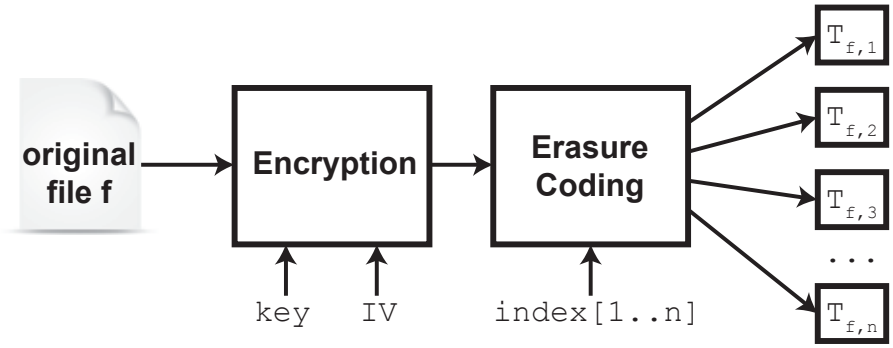


Figure 4.8: Data Flow for the Creation of Transmission Blocks

cipher block via XOR with the following clear text block so that equal blocks within one file do not result in equal cipher text. CBC requires an initialization vector (IV) to start with the first block. As initialization vector we use the first 16 bytes of the file content identifier, which in turn is derived from the whole file content. This procedure assures that different files with the same first block still result in different cipher text for the first block.

After encrypting a block, we pass the buffer holding the cipher text to the erasure coding algorithm.

Erasure Coding Our implementation uses Reed-Solomon coding in order to generate redundancy. We favor Reed-Solomon coding over fountain codes because of its properties as a MDS code: in our scenario it requires exactly k transmission blocks for decoding, while fountain codes require some additional data, which typically results in an overhead of 5-10% [Mac05]. In literature [Lub02, Sho06, Mac05] it is mentioned that fountain codes offer better performance for encoding and decoding operations. However, performance evaluations [PLS+09] and projects like Wuala [TMEBPM12] and Tahoe-LAFS [WOW08] show that in practice the performance for Reed-Solomon coding turns out to be sufficiently fast. The costly multiplication operations in the Galois field can be reduced to a set of table lookups and an addition[15], so that encoding results in less computational costs than encryption, as observed in [WOW08].

We use the Onionnetworks FEC[16] Java library, which is based on the imple-

[15] Described in more detail at http://www.randombit.net/bitbashing/2009/01/19/forward_error_correction_using_simd.html.

[16] The Onionnetworks FEC library is also used by Wuala [TMEBPM12] and Freenet[CMH+02]. It is available at https://bitbucket.org/onionnetworks/fec. While

mentation of Luigi Rizzo[17] and offers a Java Native Interface (JNI) wrapper to an implementation in C. The memory space required by the lookup table grows according to $O(kn)$. Its preparation takes up to several seconds but has to be done only once when the application launches. We parametrize the encoder by "overshooting" the coding rate as suggested in [DMTC14]. This means instead of initializing the code with (k, n), we use (k, n') with $n' > n$. When n' is sufficiently high, we can increase the redundancy level in our system over time, as required whenever a new storage node needs to be added to a swarm.

Besides a buffer holding the encrypted file content, the erasure coding procedure needs an **index field** as input (cf. Figure 4.8). This index field specifies for which indexes of the erasure code a transmission block needs to be created. Depending on whether the snapshot creator module or the maintenance module calls the transmission block creator module, we face two different scenarios:

- **Generate transmission blocks for added files**
  When we create a new snapshot, we need to generate transmission blocks for each new file content and for each storage node within the swarm. The number of new file contents for a new snapshot is limited in relation to the overall number of files, since over time only a smaller portion of files change [LPGM08]. Therefore, this workload typically affects only a smaller subset of files. Since we create a transmission block for each storage node in the swarm, we initialize the index field with all the indexes already used in the swarm. In consequence, once a file is read from disk and its content is encrypted, we generate multiple transmission blocks for different indexes. The ratio of data output against data read from disk for one file therefore is $\frac{\frac{S_f}{k}n}{S_f} = 1 + \frac{h}{k} = r > 1$.
- **Generate transmission blocks for a new storage node**
  Whenever needed, our maintenance module adds a new storage node to our swarm. For this new storage node, we need to generate one substream, which involves one transmission block for each unique file content over all snapshots of a swarm leader. In contrast to the previous workload, this workload requires to touch a high number of files on disk. Furthermore, we generate only one transmission block per file content, but still have to read the whole file from disk and perform encryption. Therefore, the ratio of data output against data read from disk is only $\frac{\frac{S_f}{k}}{S_f} = \frac{1}{k}$, with $\frac{1}{k} < 1$.

Since $\frac{1}{k} < 1 < r$, we infer that the latter case is more likely to result in a bottleneck for transmission block generation. If this is the case, it is advisable to create multiple substreams at once. The creation of multiple substreams increases the

this work was written, another comprehensive library [PMG+13] has been released.

[17] Available at http://info.iet.unipi.it/~luigi/fec.html.

share of output data per input data so that the overall data throughput of the transmission block creator module increases. The first approach to achieve this is to precalculate transmission blocks for substreams before they are needed in the system. For keeping substreams on stock we also profit from a higher value for k since they occupy less storage space on the gateway. Alternatively, we can expand the interval in which we perform maintenance. In consequence, it is more likely that multiple substreams are required at once.

Transmission Block Uploader Module

We run the transmission block uploader module completely decoupled from any other module within a separate thread. This allows uploads to happen concurrent to the generation of transmission blocks.

We organize the upload folder structure so that each storage node $sn \in SW$ has its own folder, named after its identifier. Within such a folder, we place all immutable transmission blocks scheduled for upload to the particular storage node. As file name for a transmission block we use its corresponding file content identifier. We illustrate this structure in Figure 4.9.

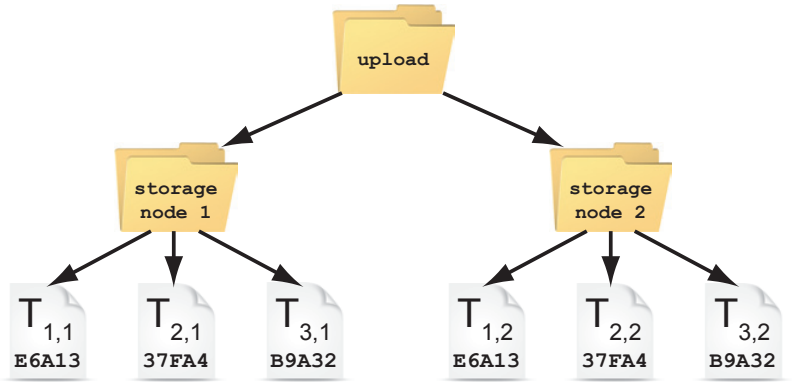


Figure 4.9: Transmission Blocks for Three Files Scheduled for Upload to Two Storage Nodes

We periodically check the upload folder for available transmission blocks. When transmission blocks are available, we start up to three concurrent uploads to ensure that the line is saturated. Only after the success of the upload to the storage node do we delete a transmission block from the upload folder. This ensures retries of failed uploads until they succeed. Further, it allows the swarm leader to detect whether a storage node is in the *synchronized* state (see our data placement strategy in Section 3.4.1 and our algorithm for maintenance in Section 3.5): only if the folder corresponding to a storage node is empty, the storage node is classified as *synchronized*.

4.3.3 Different Ways of Storing Files in a Swarm

We store files in our system in different ways, depending on their size. Our measurement study on Wuala [TMEBPM12] has shown that this strategy can be used to improve efficiency when fragments of files are distributed over a high number of storage nodes. However, the way files of a certain size should be stored primarily depends on the architecture of the system and on requirements concerning latency and storage space efficiency. In Wuala we have seen that the parameter k increases with file size in three steps. To achieve lower latency, smaller files are stored on servers only, starting with $k = 1$ (which is replication) up to $k = 10$, while bigger files are split into $k = 100$ fragments to save storage space.

In our system, however, we do not store files on servers. Instead, we could introduce different types of swarms, each of them using a parameter k which best matches the file size of files stored in it. However, this strategy unfortunately involves higher monitoring costs since each swarm needs to be observed separately. Further, it requires the tracker to hold and update several swarm sets for each participant and, thus, increases its load.

As already mentioned in Section 4.2, latency is less important in our scenario since we focus on a pure back-up service. For this reason, we choose a single value $k = 100$ which favors storage space efficiency. In the following we introduce three different ways of storing files that we use in this environment. Figure 4.10 also depicts an overview of this scheme.

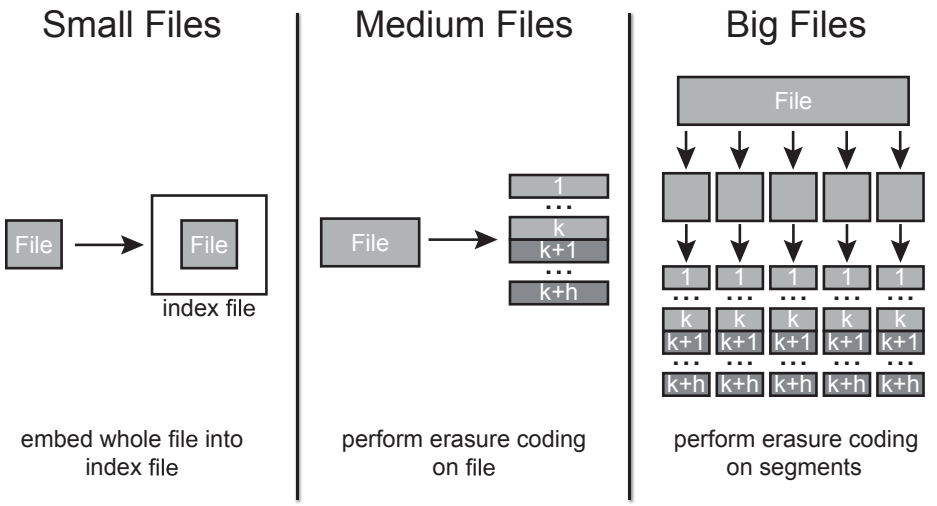


Figure 4.10: Three Ways to Store Files

Small Files (smaller than 16 KiB)

We store *small files* with a size smaller than 16 KiB in the index file. This means we only read them once to embed them into the `tar` file where they are stored close to their metadata. Since we encrypt the whole index file before we upload it, we can skip the encryption procedure for small files. Using this procedure, we avoid low data rates caused by disk seeks for small files during the creation of new substreams.

However, as a drawback, this approach prevents us from deduplicating small files over different snapshots. Same file contents therefore can be included in several index files. Due to the small amount of storage they occupy, we perceive this as justifiable (we discuss this more closely in Section 4.3.3 and Section 4.3.3, which follow hereafter).

As small files are hidden to a storage node, we thus also reduce the managing costs for storage reclamation on a storage node (we introduce storage reclamation in Section 4.5.2). As a further consequence we cannot store an index file as a small file. Therefore, we need to pad an index file to a size of at least 16 KiB so that it can be uploaded as a medium file.

Medium Files (16 KiB - 1 MiB)

We denote files with a size between 16 KiB and 1 MiB as *medium files*. After encryption we encode medium files in one pass by loading the whole file content into the data buffer of the erasure coding module. This results in transmission blocks of size $\lceil S_f/k \rceil$, which entails a maximum overhead of $k-1$ bytes per file.

While this procedure entails only negligible storage overhead, it requires more memory with increasing file size. Since we perform encryption and erasure coding in memory, we need buffers with a total size of $2 \cdot S_f + n \cdot S_f/k$. Further, in order to recover a medium file, a receiver can only use fully delivered transmission blocks. This can lead to situations in which lots of data received from a storage node cannot be used for file reconstruction only because few bytes are missing.

In order to limit the memory footprint and to keep the effect of lost connections low, we use a different method to store files bigger than 1 MiB, which we explain in the following.

Big Files (bigger than 1 MiB)

Since erasure coding is performed in memory, we cannot perform it on files with arbitrary size. This is why for files bigger than 1 MiB we switch to an

interleaving scheme [PS07], which allows us to keep the memory consumption independent of the size of a file, as also seen in Wuala [TMEBPM12] and explained in the following.

We partition a file of size S_f into $j = \lceil S_f/b \rceil$ **segments** of $b = 100$ KiB size. We choose 100 KiB because it offers good performance and requires only a small memory footprint. Each of these segments is then encoded independently using erasure coding. For this purpose, a segment is broken into $k = 100$ original fragments of 1 KiB that are encoded to obtain $n = k + h$ **coding fragments** of 1 KiB each. In this way, segment $s, s \in \{1, \ldots, j\}$ results in n coding fragments $C_{s,1}, ..., C_{s,n}$. For file f and substream SS_i we then group all coding fragments $C_{1,i}, ..., C_{s,i}$ into one transmission block $T_{f,i}$, which can be sent over the network. In Figure 4.11 we illustrate this procedure in detail.

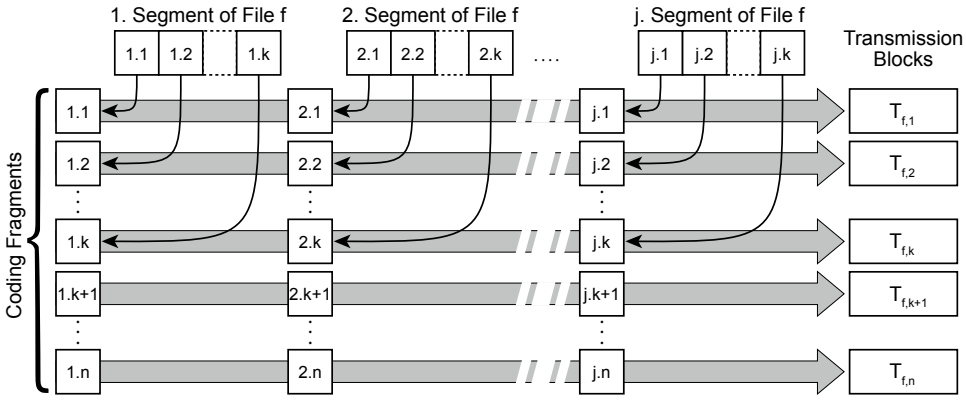


Figure 4.11: Interleaving Scheme Operating on Segments

We obtain transmission blocks with a size of $\lceil S_f/b \rceil \cdot b/k$, which entails a maximum overhead of size b per file. For $b = 100$ KiB this entails up to 10% overhead for files slightly bigger than 1 MiB. In fact, if memory consumption on the gateway is no issue, we can consider to raise the upper bound of medium files in order to lower the overhead on smaller files handled by the interleaving scheme. However, the amount of overhead becomes negligible for larger files, which typically account for most of the total storage space in storage systems (as we show in Section 4.3.3).

When it comes to recovery, in contrast to the scheme used for medium files, the interleaving scheme allows us to benefit from data of transmission blocks that are not fully downloaded. In case a storage node goes off-line during the recovery process, we are able to choose another storage node and still use the data that was already transferred. In order to recover a segment s_1, we require coding fragments for this segment from k different transmission blocks. However, to recover another segment s_2 of the same file, a different set of transmission blocks

can be used instead. Since our system supports partial transfers (introduced in Section 4.2.3), we can request particular coding fragments from storage nodes and are not dependent on the full transfer of a transmission block corresponding to a big file.

On Embedding Small Files

In Section 4.3.3 we have decided to embed small files in the index file and, therefore, we only store medium and big files individually in the swarm.

This implies that we cannot perform deduplication of small files with same file content over different snapshots. Every time the same small file is included by a snapshot, it will be embedded into the index file again. However, as shown in [ABDL07] and [TMEBPM12], it can be advantageous for system performance to store files of small size close to their metadata, which in our case is the index file.

In the following we discuss the effect of embedding small files more closely.

Storage Space Efficiency In order to determine a file size at which a file should be embedded in the index file, as a first step, we only consider storage space efficiency. For this purpose, we define functions for the storage costs on a single storage node for placing a single file of size S_f. Analyzing the storage space on a single storage node allows us to disregard the redundancy factor r in the system.

For an embedded file, these costs depend on the number of snapshots v in which the same file content is used and, therefore, is replicated over different index files. In addition we have a constant overhead S_{to} within the index file. This constant overhead is used for storing an additional file entry with all file attributes. The storage costs S_{emb} for an embedded file on one storage node are defined as follows:

$$S_{emb} = v \cdot \frac{S_f + S_{to}}{k} \tag{4.1}$$

When we store a file individually in the swarm, the costs are divided into two parts.

The first part are the costs for storing a transmission block of the file on the storage node. These costs include a constant share for metadata S_m, which involves the file content identifier and storage reclamation. Additionally, there is some constant overhead S_{so}, depending on the storage engine used on storage node side[18].

[18] In case the transmission block is stored in the file system, this includes overhead due to block alignment and inodes. For databases this is typically overhead for index structures.

The second part are the costs attributed to the index file. This includes the metadata for file attributes represented by S_{to} and the costs for metadata S_r necessary to request and reconstruct the file (as introduced in Section 4.3.2).

So the final costs for an individually stored file are:

$$\begin{aligned} S_{ind} = \frac{S_f}{k} + S_m + S_{so} \\ + v \ \cdot \ \frac{S_r + S_{to}}{k} \end{aligned} \tag{4.2}$$

To find the break even point for the costs of embedded respectively individual files, we identify the costs for storing an embedded file with the costs for storing an individual file and solve for S_f:

$$\begin{aligned} S_{emb} &= S_{ind} \\ v \ \cdot \ \frac{S_f + S_{to}}{k} &= \frac{S_f}{k} + S_m + S_{so} + v \ \cdot \ \frac{S_r + S_{to}}{k} \end{aligned} \tag{4.3}$$

$$S_f = \frac{k \cdot (S_m + S_{so}) + S_r \cdot v}{v - 1} \tag{4.4}$$

We see that the parameter k influences the break even point so that for higher k we should embed more files. This is because of the metadata required to manage additional transmission blocks on a storage node. We also see that the break even point depends on the number of snapshots that reference a file content. Because this depends on the amount of data a user changes over time, we face a feature that is individual for each user.

As most data in file systems is not touched again [LPGM08], we further look at the extreme case of an unlimited number of references to files:

$$\lim_{v \to \infty} \frac{k \cdot (S_m + S_{so}) + S_r \cdot v}{v - 1} = S_r \tag{4.5}$$

In this case, we see that k has no influence anymore, so that a file size of $S_f = S_r$ is a lower bound regarding storage space efficiency. However, since storage space in our scenario is limited by a quota, we definitely reach a limited number of snapshots. Splitting files of small size leads to very poor I/O performance due to an increase in disk head movements. At worst, this can lead to bottlenecks in the creation of new redundancy, leading to data loss [VI12, GMP09]. Consequently, especially for small files, a system design may need to disregard strict storage space efficiency and, instead, embed more files.

Impact of File Size Distributions Up to now we have focused on how to store a single file in a storage efficient way. However, we need to consider that the total storage overhead in the system depends on how much storage space is occupied by smaller files in general.

Related to this, Agrawal et al. [ABDL07] published a five-year study about file system metadata. Over the years 2000 to 2004 they collected file system metadata from over 60,000 Windows PC file systems. This involved both, files stored by a user and files for the operating system[19]. Their study shows that typically, file systems contain a lot of small files, but the majority of stored bytes are found in increasingly larger files [ABDL07]. In Figure 4.12 we see the CDF of used storage space by file size, as observed in the five year study. We see that

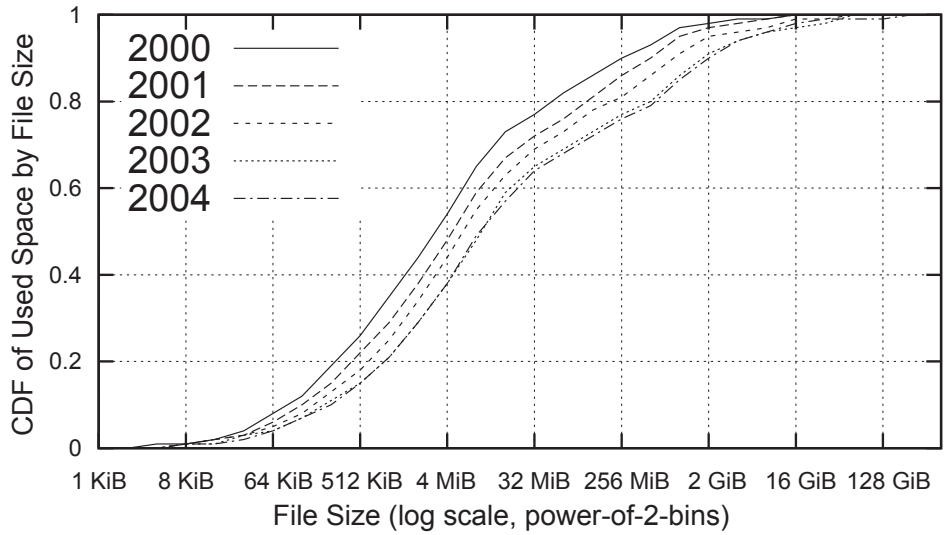


Figure 4.12: CDF of Used Space by File Size; Five Year Study of File-System Metadata [ABDL07]

smaller files do not account for a lot of used storage space. Files smaller than 16 KiB make up for only about 1% of the storage space used. During the course of the study there is also a trend towards a higher percentage of bytes stored in bigger files. In fact, the study determines that the mean file size in file systems grows by roughly 15% each year. A similar observation about increasing file sizes over time was made in a study from 1999 [Vog99]. Unfortunately, there is no comprehensive study that is more recent.

For this reason we asked the authors of Wuala [LaC13] for statistics on the file

[19]We analyzed the file size distribution of files only used by the operating systems Fedora 11 and Windows 2000 and found more bytes in small files compared to the results of the five-year study. By trend, bytes related to operating systems therefore seem to be contained in smaller files.

size distribution in their widely used cloud storage system. We received statistics on all incoming files for the month of October, 2012. Fig. 4.13 shows the corresponding CDF of used storage space by file size. Similar to the observations

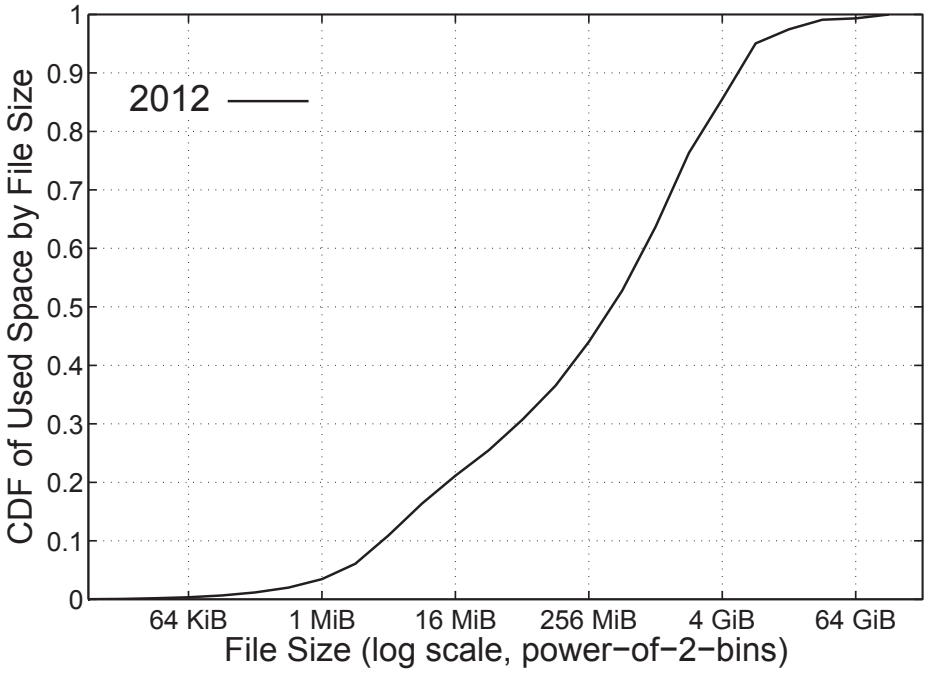


Figure 4.13: CDF of Used Space by File Size; Wuala Cloud Storage System

in the five-year study, we see that the amount of storage space used for smaller files is fairly low. In fact, all files smaller than 128 KiB account for less than 1% of the total storage space used, while files up to 16 KiB occupy less than 0.1%.

We see two reasons why we observe a larger portion of data in big files for the Wuala file size distribution. First, the data is more recent. According to the five-year study, the mean file size in 2012 should be more than twice the mean file size observed in 2005. In addition, we assume users of Wuala to store mainly personal files in the Wuala system, rather than files related to an operating system, which tend to have more bytes stored in smaller files.

Conclusion We saw that by strictly focusing on storage space efficiency, embedding small files can be disadvantageous. On the other hand, dealing with files of very small size can lead to bad system performance.

Considering the fact that file systems tend to contain most bytes in bigger files, however, embedding small files comes with a storage overhead which is barely reflected in the overall storage consumption. According to the recent file

size distribution we saw for Wuala, this storage overhead is less than 0.1% for embedding files smaller than 16 KiB, which is the size we use in our system.

Certainly, this procedure may lead to poor storage space efficiency in case of extreme workloads, i.e., when a high portion of files are smaller than 16 KiB. Such inefficiencies for small files, however, are also known[20] for other storage systems such as Hadoop Distributed File System (HDFS) [SKRC10]. In practice, however, they show only little impact. Further, in view of rising mean file sizes of about 15% per year, we expect this issue to have even less importance in future.

4.4 Tracker

This section outlines how we implement the centralized tracker. The implementation focuses on reliability so that single hardware failures withing the data center do not lead to an interruption of the service. We consider the scalability of the service so that outages due to the growth of the network can be prevented. We also show how participating gateways can switch to another tracker in case the currently used one leaves the system.

4.4.1 Resolution of Gateway Identifiers

In our system we identify gateways by using their constant gateway identifier Id_{gw}. However, in order to open a connection, we need to know the IP address related to a gateway identifier. This is why we use the Domain Name System (DNS) to map gateway identifiers to corresponding IP addresses. The tracker therefore runs an authoritative name server [IET14a] under a domain name we refer to as *primary domain*.

Every gateway in the federated network has its own dedicated subdomain within the primary domain, which results in hostnames of the form *"gatewayidentifier.primarydomain"*. When a gateway opens a connection to another gateway, it uses its corresponding hostname. Consequently, the authoritative name server returns a DNS address record [IET14a], which contains the IP address under which a gateway can be reached.

Gateways in our system generally get their IP addresses assigned either statically or dynamically by their Internet Service Provider (ISP). In case a gateway has a dynamic IP address, its IP address changes regularly. Whenever a gateway detects such change, it needs to update its IP address on the authoritative name server.

[20] Storing small files in HDFS also results in inefficiency, as explained in http://blog.cloudera.com/blog/2009/02/the-small-files-problem/

Using DNS for the mapping of gateway identifiers to IP addresses allows us to profit from the wide deployment of DNS and its caching mechanism. DNS allows to set a Time To Live (TTL) for records, which allows other DNS servers to cache records for the time specified. For gateways using a static IP address we therefore set a long TTL in contrast to gateways using a dynamic IP address, which need a shorter update period. As a result, DNS helps us to decentralize requests for name resolution from the tracker and distribute the load over the Internet. Further, DNS provides mechanisms to enhance reliability for name resolution. By specifying secondary DNS servers [IET14c], we can publish alternative servers which take over the service in case of primary server failure.

4.4.2 Internal Tracker Structure

Since all gateways in the network rely on the service provided by the tracker, it is important that it remains in an operational state. In this section we focus on the following two scenarios, which potentially undermine the service of the tracker. Subsequently we describe a possible implementation which addresses these issues.

Reliability Despite Hardware Failures

Although the lifetime of single hardware components typically spans several years, computer systems that consist of many of such components need to consider hardware failures rather to be the norm than an exception [VN10]. Hardware failures can lead to unavailability of the system and in the worst case to data loss. In general, redundancy and failover mechanisms are used to cope with hardware failures in data centers. Any possible single point of failure should be avoided.

Scaling the Tracker in Case of Increased Resource Needs

When the number of users demanding a service increases, the increase in load first leads to a slow down of the service and eventually results in an overall unavailability of the service due to overload. The property of a system to be scalable therefore ensures that an increasing demand of resources can be countered by increasing the system's resources. This can be done in two different ways:

- **Horizontal Scaling**
  Scaling a system horizontally means to add more nodes whenever more resources are necessary. This typically means that a new computer is integrated into the system.

- **Vertical Scaling**
  Vertical scaling denotes the procedure of adding more resources to already existent nodes in the system. This, for example, comprises increasing the amount of memory of a single computer.

Nowadays, data centers are often scaled horizontally to save costs [Ama14a, GGL03]. Instead of using expensive high-performance hardware, horizontal scaling allows to use cheap commodity computers.

Separation of Application Logic from Data Representation

We separate the functionality of the tracker into two layers, which can be independently scaled according to the demands of the service. We show an overview of this architecture in Figure 4.14 and further describe how each of these layers can provide reliability and scalability.

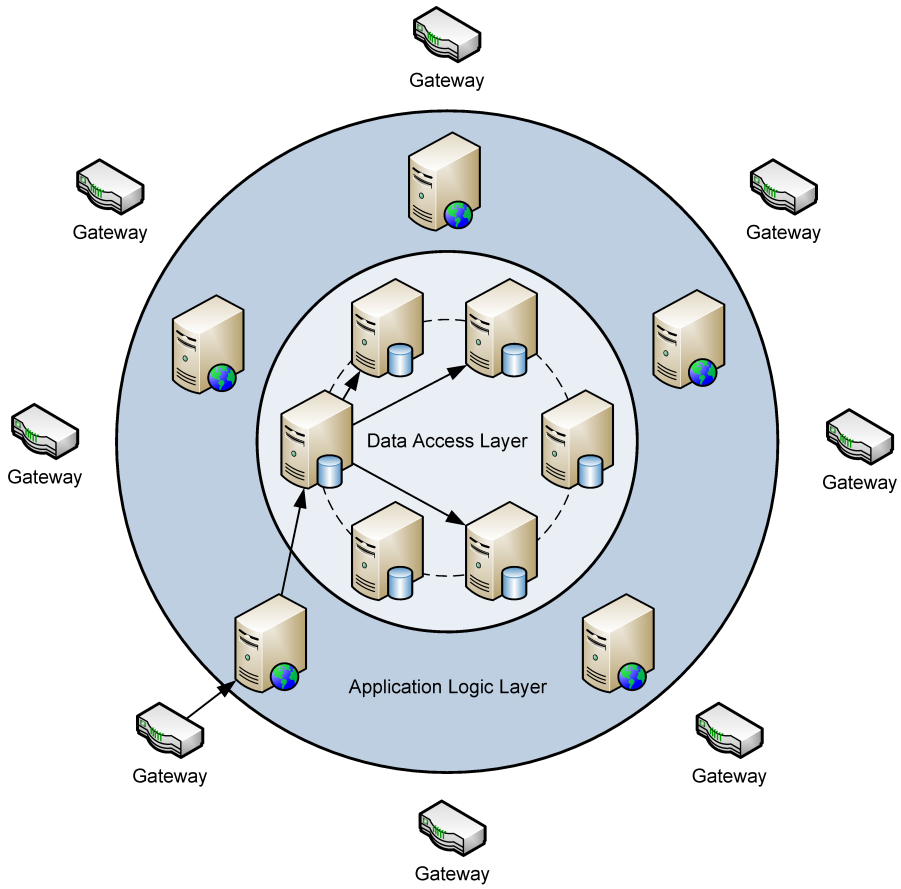


Figure 4.14: Internal Architecture of the Tracker

Application Logic Layer The application logic layer consists of application servers, which receive requests from gateways in the federated network. We distribute incoming requests among the application servers by using round-robin DNS. When a gateway contacts the tracker by using its hostname, the DNS resolves this hostname to a list of IP addresses, each of them belonging to a different application server. Since this list is rotated via round-robin, gateways send their requests to different application servers, which avoids a bottleneck in the communication between gateways and application servers.

The application servers are in charge of executing the application logic, including e.g., validation of constraints concerning input data from gateways.

Whenever an application server fails to process a request, a gateway resubmits the same request to another application server on a different IP address. Each application server can process every possible request so that the functionality of application servers is redundant. There is, thus, no single point of failure within the application logic layer.

As mentioned in Section 4.2, we generally use a stateless RESTful architecture to improve scalability. We keep no data in the application logic layer. Instead, application servers perform read and write operations on the underlying data access layer, which we describe in the following.

Data Access Layer The data access layer is responsible for storing data on persistent storage. Typically, the data we store on the tracker (summarized in Figure 3.9) are only partially accessed. All queries by gateways only affect data related to their role as a swarm leader or a storage node, which strongly limits the scope of requests in relation to the data stored for the whole federated network. In fact, we perform all look-ups and writes using a gateway identifier as a key. Hence, we encourage the use of a state-of-the-art distributed key-value store such as Apache Cassandra [LM10, DHJ+07].

Apache Cassandra internally uses a DHT among nodes so that they are organized in a decentralized way, avoiding a single point of failure. Every node in the data access layer can likewise process a read or write request. Writes are performed to a configurable number of different nodes in order to achieve fault-tolerance. Further, each node in Cassandra is only responsible for a certain portion of the data set so that load is distributed among all nodes.

Consistency of Data Stored on the Tracker

The CAP theorem by Lynch et al. [GL02] states that for distributed storage it is only possible to provide two out of the following properties:

- **Consistency**
  All participants see the same state of data at every point in time. This is also referred to as *strict* or *strong* consistency.

- **Availability**
  Every request receives a response about whether it was successful or not.

- **Partition-tolerance**
  The system still continues to operate in spite of lost messages and is fault tolerant for parts of the system.

At this point, the definition of consistency slightly differs from the definition used in database system design. While database consistency refers to the property that transactions transform a database from one valid state to another, Lynch et al. refer to a single atomic operation of a single request/response [GL02]. Further, Fekete et al. [FGL+98] suggest relaxing the property of consistency to *eventual consistency*, which allows transient inconsistencies to occur as long as they are resolved eventually. According to Lynch et al., this relaxation allows to fulfill the properties availability and partition-tolerance at the same time.

In our system, we require the tracker to be partition-tolerant because in practice partitions are unavoidable due to component failures. Hence, according to the CAP theorem, we can choose between strict consistency or availability.

Writes on the tracker concern updates for the swarm set. If such information is not available on the tracker, there is the risk that a recently added storage node could be missing for recovery. We see that writes on the tracker should always be accepted and, hence, that the tracker's availability is crucial. In contrast, when a gateway fails, it is less important that data on the tracker is consistent at this particular point in time. We assume that before a user replaces the gateway and starts the recovery process, the tracker has enough time to resolve inconsistencies in the data related to the concerned back-up.

The fact that the classic Relational Database Management System (RDBMS) focuses on strict consistency, supports us in our decision to use Cassandra, which is designed to support partition-tolerance and availability [Apa14].

4.4.3 Total Tracker Outage

Up to now we have ensured that single hardware failures do not have an impact on the service provided by the tracker. There is, however, still the risk of losing the tracker, e.g., when the operator stops all services. In the following we analyze this scenario more closely and show how the federated network can return into an operational state.

Impact on the Federated Network

A complete tracker outage has a strong impact on the functionality of our back-up system, which is, henceforth, reduced to the remaining gateways in the system.

These remaining gateways face the following consequences and restrictions:

- Swarm leaders cannot request new storage nodes from the tracker. Performing maintenance is not supported so that the redundancy provided by a swarm decreases over time.
- The CA does not issue new public key certificates so that new gateways cannot join the system anymore.
- Fairness in the system is not globally observed by the tracker.
- The copy of the swarm set on the tracker is inaccessible. In case of data loss on a gateway we need an alternative way to recover the swarm set.
- Gateways cannot update their DNS address records on the authorative name servers anymore. Gateway identifiers therefore cannot be resolved to IP addresses so that especially the connectivity to gateways with dynamic IP addresses is limited.
- In case of local data loss a user has *no entry point* to the federated network anymore. It is therefore impossible to catch up with other participants and access a previously stored back-up.

We see that especially the last point is crucial for our system. A back-up system lacking the possibility to recover from data loss is of little value. However, not even decentralized approaches help in this scenario. In order to join a DHT, a node also requires [MM02, SMLN+03, RFH+01, RD01] an entry point, which is typically an IP address of a node that is already participating in the DHT. In the following we show a best effort approach how the system can still operate to some extent without the tracker.

Fallback Mode and Regathering

In case a gateway cannot reach the tracker for a predefined period (e.g., a couple of days), it switches into a *fallback mode*. In this mode the behaviour of gateways changes as follows:

- **Domain Name Alternation**
  Over time the gateway alters the domain name under which it tries to contact the tracker. For this we build a candidate set of possible alternative

domain names, using a deterministic function. We seed this function by the current date on the gateway so that each day all gateways try to reach the same domain names in the DNS.

Such procedure is known as Domain Generation Algorithm (DGA) and became popular due to their usage in recent botnets such as Conficker [PSY09]. Similar to our scenario using a tracker, botnet operators try to be resilient against outages of their central command and control server. DGA therefore allows us to create a high number of rendezvous points we can use in future to gather previous gateways again.

It is sufficient to register one of the domain names in the candidate set of a given date and run an authoritative name server. This authoritative name server only needs to provide the service for the resolution of gateway identifiers as previously described in Section 4.4.1. Henceforth, gateways are able to connect to each other again. In case an attacker registers one of the potential domain names without providing the intended service, the gateways still try to reach other authorative name servers on different domain names.

- **Heartbeats to Back-Up Owners**
  Since we consider the swarm set on the tracker as lost, in case of local data loss on a gateway, we need to recover the swarm set using a best effort approach. To achieve this, we change the origin for establishing contact: instead of the back-up owner contacting its storage nodes in order to recover, the storage nodes try to contact the back-up owner. This is possible since storage nodes know the back-up owner corresponding to each substream stored on it. A storage node therefore regularly sends heartbeat messages to a back-up owner.

  As soon as after data loss the back-up owner reconnects using its previous gateway identifier, it receives these heartbeat messages and, thus, can determine potential sources to start back-up recovery.

  In case a back-up owner uses a static IP address, this procedure does not even need to rely on the resolution of gateway identifiers: the back-up owner is likely to return with the same IP address and therefore can be contacted directly.

Whenever gateways detect that the tracker reappears (possibly on one of the rendezvous points), they check its authenticity via its public key. If the authenticity is confirmed, gateways return to the normal operation mode and, thus, continue to perform maintenance again.

4.5 Storage Node

In this section we focus on the functionality implemented on a storage node.

4.5.1 Storing Transmission Blocks

A storage node in our system is responsible for holding $(key, value)$ pairs, where key is the file content identifier of fixed size and $value$ represents the transmission block related to the file content identifier. When a storage node receives an aggregate, we extract all contained transmission blocks and store them individually.

As a result from the way we store files, our storage nodes receive transmission blocks of variable length. This includes transmission blocks worth a small sequence of bytes, up to a size which is only limited by the underlying file system on the storage node.

Storing small sequences of bytes in file systems tends to waste storage space. This is caused by **internal fragmentation** due to the block alignment in file systems. When a file system uses a fixed block size of, e.g., 4 KiB, and one stores a file worth 1 KiB, this leads to a storage overhead of 3 KiB since the whole 4 KiB are allocated and cannot be used anymore.

Databases typically store sequences of variable length in a storage efficient way so that no storage space is wasted due to block alignment [SZT+08]. Further, they achieve higher throughput for smaller objects, e.g., because database queries are faster than file opens [SVIG06]. Hence, we place transmission blocks up to a size of 256 KiB [SVIG06] in a local lightweight database[21], using the file content identifier as key.

For sequential access on bigger transmission blocks, file systems typically provide better performance [SVIG06]. This is why we store transmission blocks larger than 256 KiB in the file system. We use the file identifier as file name to locate the corresponding transmission block.

In turn, on deleting transmission blocks we face **external fragmentation**, in which allocated data is separated by freed regions. Making use of these regions requires ad hoc handling, e.g., by checking for the availability of such regions before a new file is stored. While modern file systems address this issue[22], we occasionally need to execute a compaction[23] procedure on the database.

[21] We use Apache Derby because of its small footprint. Derby is available at https://db.apache.org/derby/.

[22] Modern file systems, like ext3, do not require defragmentation, as explained in http://www.tldp.org/LDP/sag/html/filesystems.html.

[23] As explained in http://db.apache.org/derby/docs/10.1/ref/rrefaltertablecompress.html.

4.5.2 Storage Reclamation

Storage reclamation generally refers to the process of reclaiming allocated storage space for data that is not used anymore by a system. Storage allocation and reclamation techniques are well-studied areas in computer science [WJNB95, Wil92] and, e.g., find their use in modern garbage collectors for automatic memory management [JHM11].

As storage space provided by a swarm is limited in our system, we also consider a method for storage reclamation. This way we can delete previous snapshots and store more recent ones. Our off-site snapshot representation (introduced in Section 4.3.2) uses a flat hierarchy with only one level of references to file contents. We neither need to follow these references, nor do we encounter cycles in the graph. This allows us to use straightforward methods such as reference counting or Least Recently Used (LRU). For reference counting, an object is deleted when there exists no more reference to the object. LRU, on the other hand, is typically used as a replacement policy in caching [SK09], where a cheap implementation is crucial. It tags all objects with a single value indicating their last usage, which in our case is the last time a file is referenced by a snapshot. When it comes to storage reclamation, it deletes all objects assigned with the oldest value. In our implementation we choose LRU because of its simplicity and performance.

The storage reclamation procedure can be executed either on the swarm leader or on the storage nodes. Since the responsibility for off-site snapshot quota management is generally incumbent upon the storage node, we perform storage reclamation on the storage node side as well[24].

In the following we explain how a storage node maintains LRU information in a *tagging phase* by using the supplied metadata conjoined with a snapshot. We further use this information to delete unused data in the *sweeping phase*.

Tagging Phase

To support LRU in our system, a storage node keeps track of file content usage in order to determine which file contents are not referenced by recent snapshots anymore. For this, we tag each file content identifier with a LRU value, by indicating the last snapshot that references this particular file content.

Since in our system the creation time of a snapshot is used as its identifier v_t, the sequence $v_1 \dots v_t$ is strictly increasing. Hence, for each file content f we keep a tuple (f, v) in our local database[25].

[24] By shifting the task to the swarm leader, however, it is possible to integrate common web services e.g., Amazon S3 to function as a storage node.

[25] In fact, our system supports intermediate deletion of snapshots, e.g., after one week. Due

As we mentioned before in Section 4.3.2, a swarm leader submits a new snapshot v_t accompanied by metadata, which includes, in particular, the *unused file content identifiers* $OF = FS_{t-1} \setminus FS_t$ and the *new file content identifiers* $NF = FS_t \setminus FS_{t-1}$. We use these sets to update the tuples (f, v) according to a new snapshot. In order to update all identifiers f, which are used by both, v_t and v_{t-1}, we update all tuples $\{(f, v) \mid f \in FS_{t-1} \wedge f \notin OF\}$ to the new value v_t by using a single transaction in the database. This procedure allows us to update the set $FS_{t-1} \cap FS_t$ without having to transfer all the affected identifiers for each new snapshot. As files in the set NF are possibly referenced by previous file sets $FS_1 \ldots FS_{t-2}$, we use a single transaction to either update existing tuples or otherwise insert new ones.

Sweeping Phase

In case the substream quota of a swarm leader is exhausted, a storage node starts the sweeping phase.

In this phase we successively delete old snapshots in order to free storage for a new snapshot. In order to delete a snapshot v_d this means we delete all transmission blocks related to file content identifiers exclusively referenced by this particular snapshot. This includes transmission blocks of both, referenced file contents and the index file. We use the tuples maintained by the tagging phase to narrow down these file identifiers. All files with identifiers $\{f_i \mid (f_i, v_j) = (f_i, v_d)\}$ are not referenced by any more recent snapshot; we therefore can delete corresponding transmission blocks without impact on remaining snapshots. After this is done, a storage node deletes the metadata concerning the snapshot in order to finalize the deletion of the snapshot.

We terminate the sweeping phase after a snapshot is fully removed and enough storage space for a new snapshot is available. Alternatively, e.g., when a minimum number of snapshots in the swarm is reached, we cannot remove further snapshots and need to move the back-up into another quota pool, as discussed in Section 3.4.2.

4.6 Incentives

In our system gateways do not store their data in a reciprocative way; it is a unidirectional relation where a swarm leader stores its data (as we justified in Section 3.4.1). Therefore, we do not use algorithms such as tit-for-tat to reward fair behaviour[26]. For back-ups, such fairness can also be difficult to

to this we keep a list of the snapshots referencing a file content within this period and reduce it to a single LRU value afterwards.

[26] Like e.g., in BitTorrent [Coh03], where nodes assist each other to download a file.

measure, e.g., some participants may have more frequent changes in their data and, thus, need to transfer more data than others. However, since we encounter an asymmetry of interest, we need to make sure a malicious participant can only harm the system to a limited extent.

As suggested in [BLV05], we use a reputation system to record misbehaviour of participants. We assign this reputation system to the CA. Whenever a swarm leader or a storage node observes misbehaviour, it will send a report to the CA. Such reports should be the exception rather than the rule, so that we do not expect heavy load on our central instance caused by report messages.

We tolerate false reports by waiting for several independent reports before a participant is declared as malicious by the CA. Such declaration results in the revocation of the public key certificate and a notification sent to all storage nodes of the concerned swarm leader. As a result, the stored back-up is erased and the swarm leader is excluded from the system.

In this context, we also need to consider collusions, where several federated participants send malicious reports to harm a particular participant. It is, however, expensive to form collusions in our system. Reports can only affect participants which stand in a relationship via a swarm. Since our tracker assigns new storage nodes by random, attackers cannot chose their target at their own will. When colluding nodes want to control a portion p of the possible reports for a particular node, they need to control $c = p \cdot n$ nodes within a swarm. However, in order to infiltrate c nodes into a particular swarm, in average, $\frac{c \cdot N}{n}$ nodes are required within the federated network. From this follows that with increasing network size also a higher number of colluding nodes are required.

Which factors we take into account concerning the malicious behaviour in our system is explained in more detail below, where we intend to show how to detect such behaviour and point to possible countermeasures.

- **Incentive to Keep Data of Others**
  Keeping data from other swarm leaders increases storage costs on a storage node. Therefore a storage node may be tempted to drop received data instead of allocating resources for them. A swarm leader, however, detects such behaviour at low bandwidth costs by using the hash-based integrity checks. True hardware errors are unlikely to affect a huge portion of the data set. Hence, given a swarm leader receives false responses for several challenges concerning different transmission blocks, it sends a report to the CA.

- **Limiting the Amount of Redundancy**
  The amount of redundancy injected into a swarm basically depends on the average life time τ of gateways and the time t_{iso} we want a back-up to survive without further maintenance (cf. Figure 3.12 and Figure 3.13).

Further, in our system with globally fixed parameter k, more redundancy results in a higher number of storage nodes in a swarm, which need to store a full substream. A swarm leader could calculate the required amount of redundancy using different parameters so that its back-up survives a longer period of time than the back-up of others; this would undermine fairness in the system. Unfortunately, a storage node with its limited view cannot detect such misbehaviour so that we need the tracker to perform this observation. Therefore, the tracker occasionally checks for swarms having outlying swarm sizes.

- **Motivation to Stay Connected**
  Since a swarm leader is responsible for maintaining its own back-up by uploading new substreams into the system, it is already in the interest of the user that the gateway is connected to the federated network most of the time. A possible extension to this approach is, however, that the tracker introduces a bias for picking new storage nodes with a similar availability as the swarm leader exhibits to have.

- **Quota Compliance**
  A swarm leader needs to adhere to the storage quota provided by the storage nodes in the swarm. In our architecture, a storage node can attribute the occupied storage space to the corresponding swarm leader. In consequence, it refuses to store new transmission blocks as long as the quota is exceeded.

- **Incentive to Serve Data**
  The fact that a storage node keeps the data for a swarm leader does not necessarily mean the data is served on request as well. Serving the actual data leads to higher bandwidth costs than answering challenge-response requests, so that such requests may be dropped by a storage node. Given a swarm leader does not receive requested data during the recovery phase, although the storage node is on-line, the swarm leader sends a report to the tracker.

Overall, we see that a combination of measures implemented on the tracker and participants can lead to higher fairness in the system. Especially the tracker as a central instance can contribute to establish a fair usage of the system due to its global knowledge. Its usage, however, needs to be weighted carefully against the resulting costs.

4.7 Conclusion

This chapter outlined the key features of the implementation of our back-up system.

The implementation relies on compositions of single atomic actions without requiring any locks. Hence, the result is a high level of concurrency, which allows the system to scale well and yet ensure referential integrity.

Although in literature fountain codes are often preferred due to their higher performance, we saw that classical MDS erasure codes perform well enough and help us to avoid storage overhead. In fact, file encryption and disk I/O turn out to be limiting factors when it comes to substream creation. Holding additional substreams on a swarm leader therefore shows to be a good idea in case the creation of substreams is slow. We further saw that files of different size should be handled differently. For small files we save metadata and performance by embedding them into index files. According to the Wuala file size distribution, this results in a storage overhead of 0.1% per snapshot, which we consider as reasonable. For the future we can expect this portion to decrease even further. For big files, on the other hand, we profit from an interleaving scheme, which reduces the memory footprint for erasure coding and allows us to profit from partial transfers when storage nodes turn off-line during the recovery process.

We further illustrated how we use state-of-the-art technologies in order to include a central instance that is highly scalable, fault-tolerant, and is even replaceable in the worst case. On the storage node side we illustrated how to free storage space without additional communication overhead. Combined with occasional checks by a trusted CA, we see that our service can enforce correct behaviour in the federated network.

Chapter 5

Impact of Correlated Failures

"Failure is a part of success."

Hank Aaron

5.1 Introduction

In our system, in accordance with the work by Toka et al. [TCDM12], we assume that node lifetime values follow a poisson process. In fact this assumption means we expect permanent failures to occur both independently and exponential distributed. This is important for our repair policy for lost data as introduced in Section 3.5.2. It concerns Equation 3.1 in particular, which ensures data durability in our system over a predefined period with a certain probability.

In this chapter, we first analyze whether our assumption of lifetime values that are exponentially distributed is correct in a real world scenario. We use statistical methods to examine a trace collected in an environment that is similar to the one our system focuses on.

Subsequently, we discuss the geographical scope of failures, the resulting implications on our system and provide an outlook for a deployment on global scale.

We finally perform tests to check whether the durability of back-ups stored in our system can be assured as intended.

5.2 Suitability of the Markovian Assumption

We discuss the poission assumption in the following by analyzing real world traces of residential gateways.

5.2.1 Real World Traces

There exist numerous studies [DMR10a, RP06, DBEN07, NYGS06] that analyze metrics about the on-line behaviour of participants in P2P systems and the resulting impact on properties like the availability of stored data. These studies often rely on publicly available traces collected by observing particular networks over a longer period. The Failure Trace Archive [KJIE10, EIG+14] provides a collection of traces, captured in different environments. The traces published in the Failure Trace Archive, however, only focus on either servers or user devices, which, e.g., act as peers in P2P systems.

Servers, in contrast to our gateways, are typically hosted in air-conditioned data centers. Such an environment improves the operational conditions and reduces the risk of hardware damage to occur [BSSP10]. In addition, data centers are usually equipped with a redundant power supply and redundany Internet access. These factors enhance the reliability of the service provided.

Although we expect the gateways in our scenario to be on-line most of the time, they lack redundant connectivity and, thus, are more likely affected by power cutoffs or loss of connectivity to the Internet. User devices, on the other hand, fit our domestic environment but are typically turned off when they are not in use.

5.2.2 Traces Matching Our Environment

The only study we found that matches our targeted environment is a study by Defrance et al., published in [DKM+11]. They observe the availability of residential gateways of the French ISP Free [Fre13]. The gateways are operated at home and are connected to the Internet via ADSL links so that the environment fits to the scenario in our system.

In their experiment, they periodically ping-ed a set of 24,781 IP addresses within the address range of Free. Free assigns static IP addresses. Hence, a gateway does not reappear in the data set under a different IP address due to dynamic assignment. The experiment covers a period of 7.5 months in total. The result is a file providing a list for each gateway with timestamps when a gateway turns on-line or off-line. We further refer to these traces as *Free-traces*.

Figure 5.1 shows the availability of the observed gateways over the whole period. With an average availability of 86%, the gateways show a rather high availability

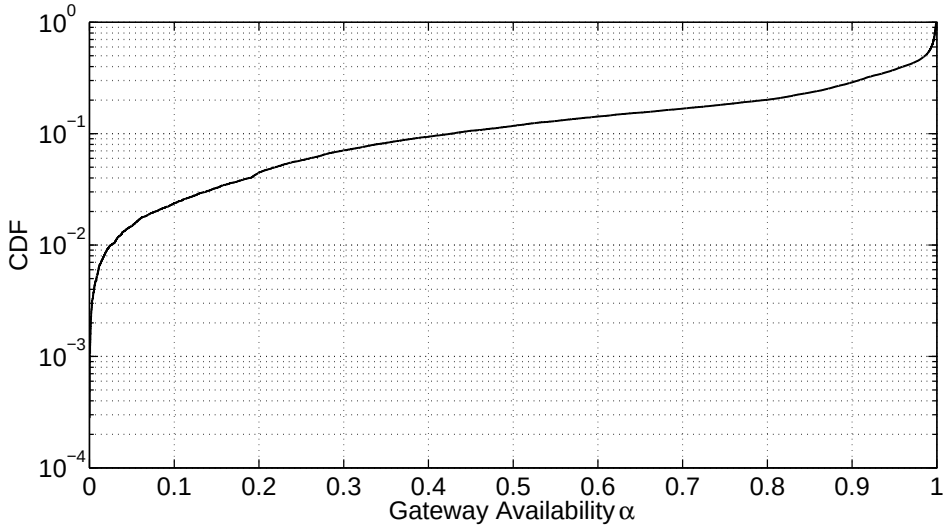


Figure 5.1: Gateway Availabilities in the Free-traces

but, as expected, cannot compete with the availability provided by servers, which ISPs typically state to be about 99%. In Figure 5.2 we see the CDF of the downtime duration of gateways.

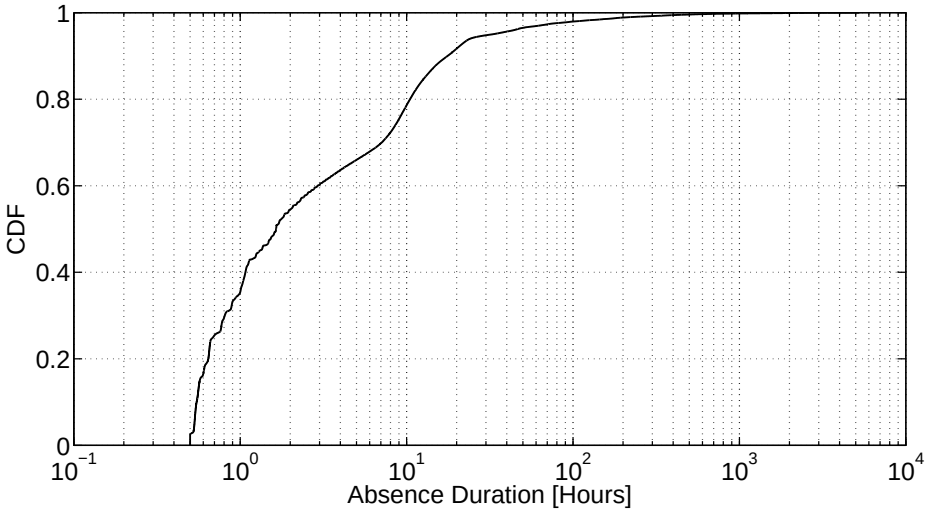


Figure 5.2: CDF of Downtime Duration in the Free-traces

A gateway could leave the system for a longer duration due to several reasons, such as permanent device failure, change of the ISP, or a user turning off the

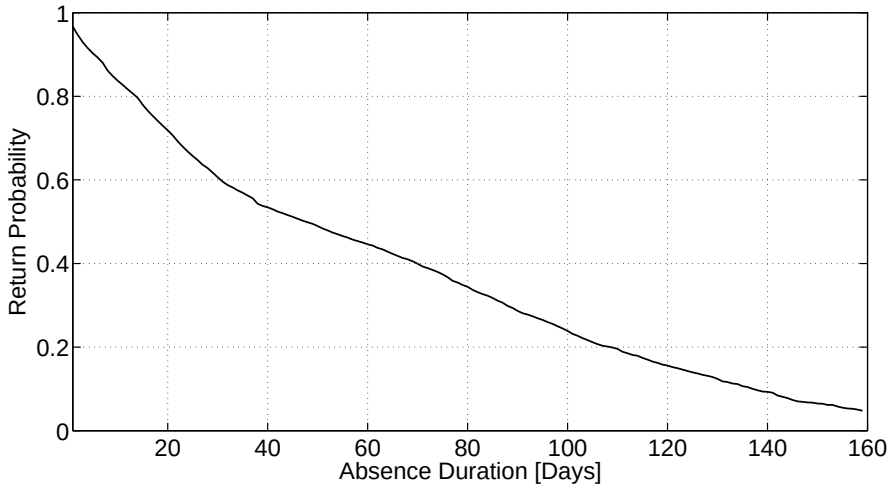


Figure 5.3: Return Probability Depending on Absence Duration

gateway to save energy. Since we cannot differentiate between these events, we assume that gateways with an off-line period of more than 10 weeks to the end of the data set to have left the system and consider them as *permanent failures*. In fact, this procedure involves inaccuracy since some gateways still reappear in the system at a later point in time, as we see in Figure 5.3. The limited observation period of the Free-traces, however, does not allow us to detect permanent failures in a more reliable way. In our experiments we therefore overestimate the number of permanent failures in the system.

On this basis, we calculate the annual failure rate, which results in $\lambda = 0.1345$. This is equivalent to an average life time of $\tau = \frac{1}{\lambda} = 7.43$ years.

5.2.3 Independence of Permanent Failure Events

We assume permanent failures to occur independently. This is crucial, since a high number of correlated permanent failures within a short time period could lead to loss of back-ups, even before the time expires that is provided by our system to the user in order to recover. In order to analyze permanent failures more closely, we first plot a histogram of permanent failures per day (cf. Figure 5.4). We see peaks for some days with up to 25 gateways leaving the system. Since we still have 23,634 active gateways after the end of the observation, this equals less than 0.11% of total gateways. The storage system Glacier [HMD05] focuses on tolerating a fixed number of nodes leaving the system within a short period of time. However, in [NYGS06] the authors show that this approach does not lead to satisfying results.

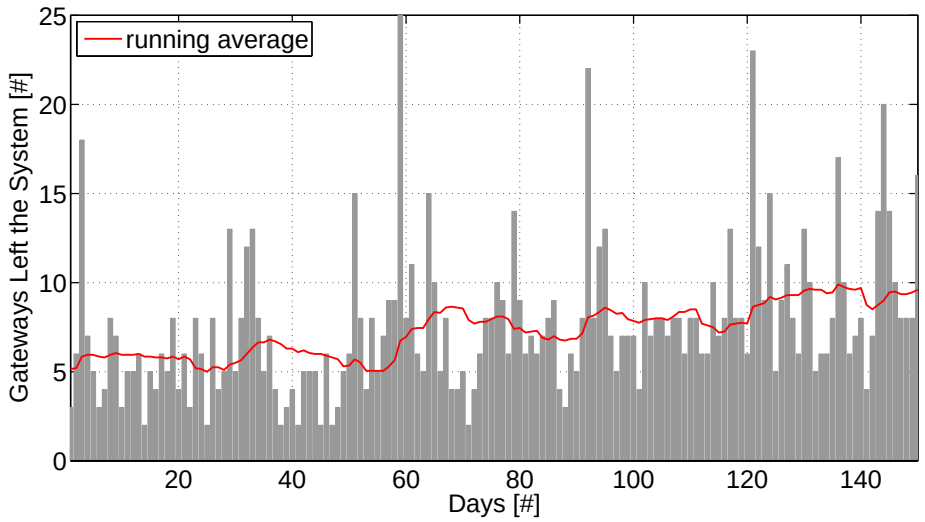


Figure 5.4: Number of Gateways Leaving the System per Day

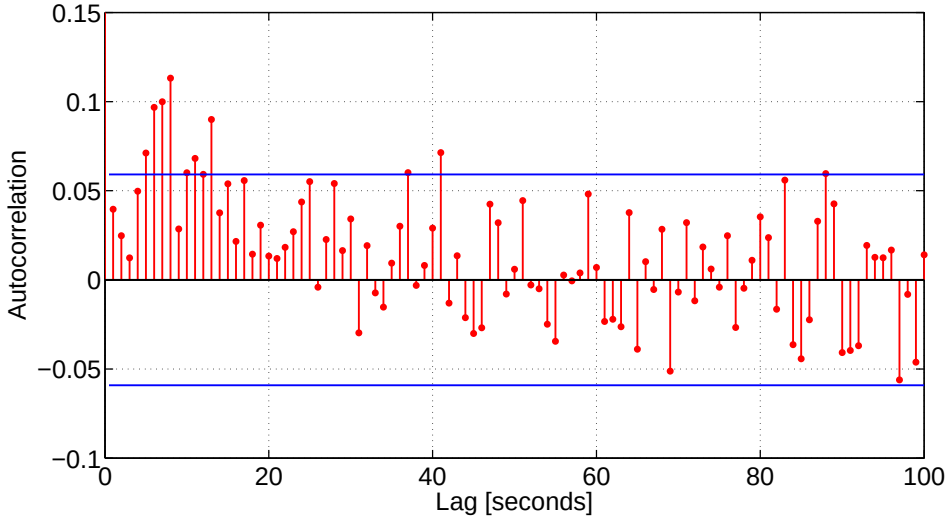
In order to test the Free-traces for independent permanent failures, we perform a Chi-Square-Test [GN96]. We examine permanent failures per day as well as permanent failures per week. For both tests we cannot reject the hypothesis that the distribution of permanent failures is independent at a significance level of 0.05. As we observe insufficient permanent failures on an hourly basis, we cannot perform the Chi-Square-Test for lower time intervals.

For this reason we create autocorrelation plots as suggested in [LB10]. An autocorrelation plot shows the correlation between a random variable with itself at different time lags. Given time-ordered measurements $Y_1, Y_2, \ldots, Y_N$ with mean $\overline{Y}$, the autocorrelation coefficient for time lag l is defined as [BJR94]:

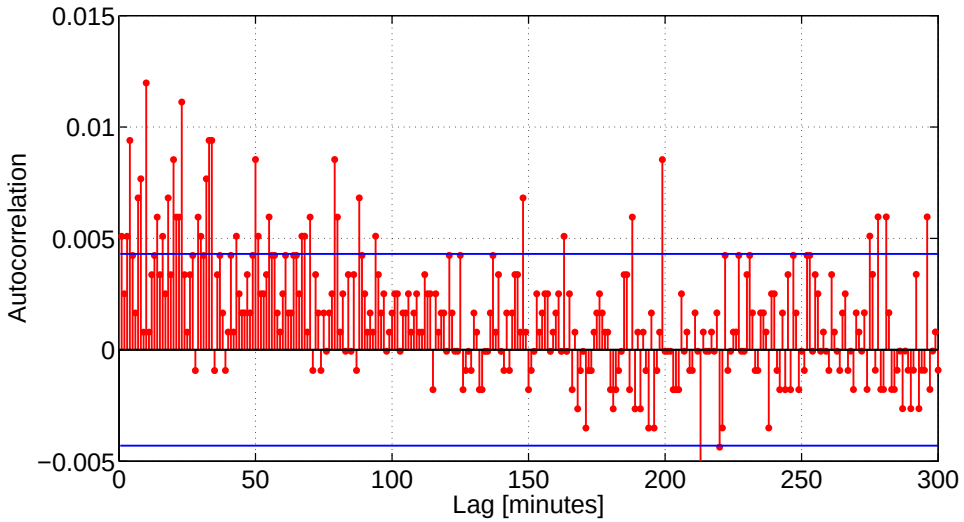
$$c_l = \frac{1}{N} \sum_{t=1}^{N-l} (Y_t - \overline{Y})(Y_{t+l} - \overline{Y}) \tag{5.1}$$

In a first step, we sort all the different failure timings increasing over time. Subsequently, we calculate the difference in seconds between consecutive permanent failures. Figure 5.5(a) shows the autocorrelation coefficients of the resulting data together with the 95% confidence interval which is approximated by $\pm 2/\sqrt{N}$. We indicate the confidence interval in the figures by using two horizontal lines.

We see that not all coefficients fall into the confidence interval, yet, some are slightly positively correlated. In order to determine the failures per minute we count the total number of failures in bins of one minute over the whole data



(a) Permanent Failures per Second



(b) Permanent Failures per Minute

Figure 5.5: Autocorrelation Plots Showing Correlation Coefficients for Permanent Failures per Second and Minute for Different Lags

set. As can be seen in Figure 5.5(b) the autocorrelation plot for the failures per minute shows a similar pattern as the failures per second. While for failures on a weekly basis the plot shows no significant correlation at all, we see slightly higher correlation coefficients for failures per hour and failures per day. We show the latter autocorrelation plots in Figure 5.6(a) and Figure 5.6(b).

In summary, we see that permanent failures in the analyzed data set show no strong correlation.

5.2.4 Exponential Distribution of Permanent Failures

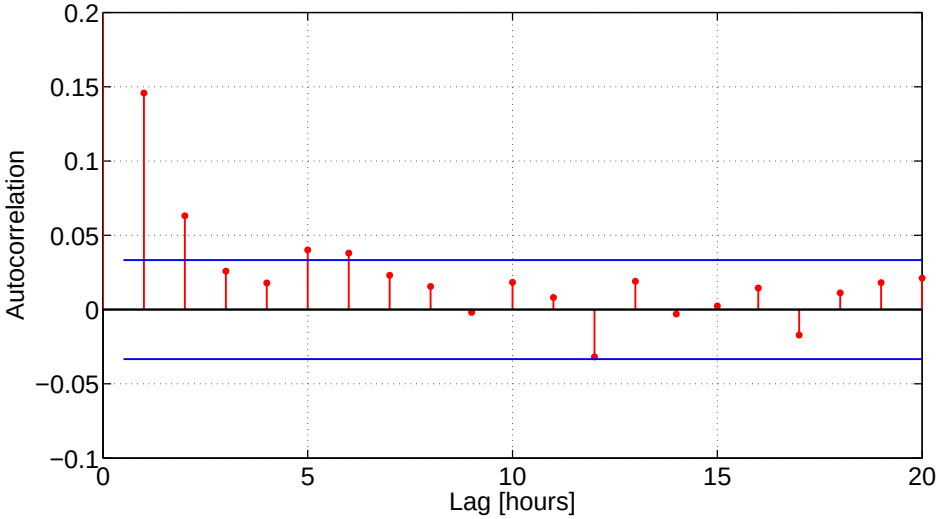
An exponential distribution of the time between failures is often used in reliability theory to model the overall lifetime of a system. It assumes failures to occur at a constant rate and to be memory-less, so that previous failures do not affect the likelihood of future failures. However, in reality, this assumption does not hold in most cases. Failures of entities such as electrical devices usually follow the bathtub curve [YS99], which divides the failure rates into three parts: an early failure rate, a random failure rate, and a wear out failure rate. While the early failure rate decreases over time, the wear out failure rate increases. Only the random failure rate stays constant over time and, thus, corresponds to an exponential distribution.

We see in Figure 5.4 that the failure rate is not constant over time in the analyzed data set. We use a Kolmogorov-Smirnov test (K-S test) to determine whether the empirical distribution of permanent failures in the Free-traces corresponds to a theoretical exponential distribution. The null hypothesis of the K-S test is that both samples are from the same continuous distribution, while the alternative hypothesis states that they are from different continuous distributions.

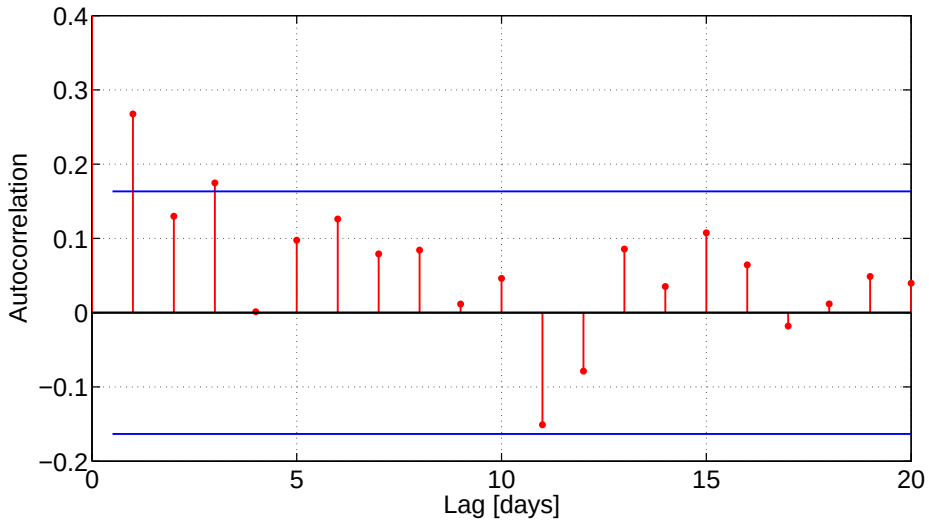
The Kolmogorov-Smirnov statistics is defined as [LB10]:

$$T = \sup_x |F_1(x) - F_2(x)| \tag{5.2}$$

where F_1 and F_2 are the empirical distribution functions of both samples. For a large number of observations n_1 and n_2 the null hypothesis is rejected if $T > c(\alpha)\sqrt{\frac{n_1+n_2}{n_1 n_2}}$ where $c(\alpha)$ is a constant according to the significance level α. This constant is provided by most statistical software packages. According to this test, with a significance level of 0.05, we have to reject that permanent failures in the Free-traces follow an exponential distribution.



(a) Permanent Failures per Hour



(b) Permanent Failures per Day

Figure 5.6: Autocorrelation Plots Showing Correlation Coefficients for Permanent Failures per Hours and Days for Different Lags

5.3 Discussion

From the tests we performed above, we summarize that permanent failures in the Free-traces do not occur independently and the time between failures is not exponentially distributed at a significance level of 0.05. In consequence, the exponential distribution can only be an approximation for the analyzed data and has to be used with caution.

In fact, understanding the reasons why correlated failures occur is a crucial step in order to infer their potential dimensions and the resulting impact on the system. Further, depending on the causes of correlated failures, the possibility to employ countermeasures might be limited as well.

We therefore identify the following triggers for correlated failures in our distributed system:

- **Destructive Events**
  Destructive events that lead to correlated failures can either be naturally caused or human made. Naturally caused are events such as fires, lightning strikes, floods, storms, and earthquakes [KV10, IBe99]. Human made events include for example accidents, warfare, and theft.

- **Change in Operational Environment**
  Extreme conditions due to the operational environment can lead to an increase of correlated failures. This is valid especially for periods of extreme temperatures [SSVG13], but also for increased dust contamination [Zha07].

- **Permanent Infrastructure Issues**
  It is possible that multiple participants in the system are permanently disconnected from the system, e.g., due to an ISP closedown or nationwide connectivity restrictions. However, such issues do not lead to device failures so that there is still the chance for reintegration.

- **Attacks on Gateways**
  When gateways are victims of a computer virus attack, they potentially lose all data. This can affect any gateway at any location.

We find that the first three points show a **geographically-related** property: destructive events, the operational environment and the infrastructure of ISPs or countries have a limited geographical scope. We therefore first focus on the impact of geographical proximity of correlated failures. Afterwards, we discuss characteristics of **geographically-diverse** correlated failures.

5.3.1 Geographically-Related Correlated Failures

Since in our system we select storage nodes at random, we generally store data in a geographically distributed way, potentially across the whole world. In contrast to the Free-traces, we are neither restricted to a single ISP, nor to the geographical area of a single country. From such a global scale deployment we therefore expect less impact [KV10] from geographically-related correlated failures.

Assuming a maximum of C storage nodes to fail simultaneously in the federated network of N gateways due to a geographically related event, we can calculate the probability for having at most x failures within a swarm of n gateways by using the hypergeometric distribution:

$$P(X \leq x) = \sum_{y=0}^{x} \frac{\binom{C}{y}\binom{N-C}{n-y}}{\binom{N}{n}} \tag{5.3}$$

ISPs typically record information about causes for problems in their infrastructure plus the number of affected customers by such events. Unfortunately, since it is not in their interest to share such information with the public[1], we cannot further analyze the underlying reasons and extents of geographically-related correlated failures.

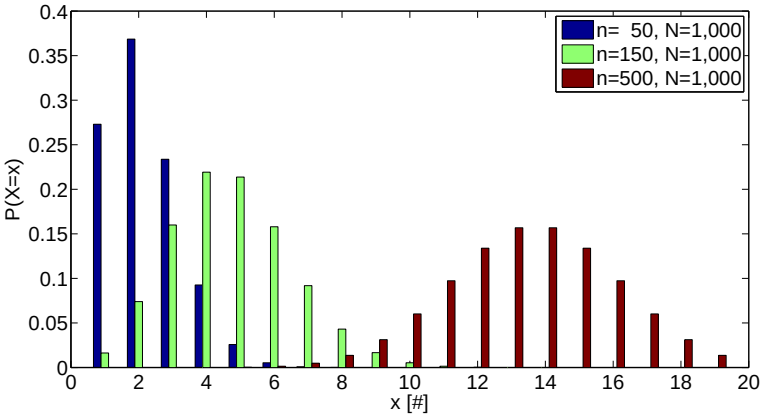
In Figure 5.7(a) we therefore show an example in which we assume to suffer x correlated failures out of $C = 25$ for a varying number of storage nodes in the swarm. We see that for higher n, by trend, more storage nodes within a swarm are affected by correlated failures.

Following from the mean of the hypergeometric distribution, the mean of affected storage nodes within a swarm is defined by $E(X) = \frac{nC}{N}$. Hence, when the size of the federated network increases while C remains constant due to the geographical scope, we observe less impact for single swarms per geographical event. This is illustrated by the example shown in Figure 5.7(b).

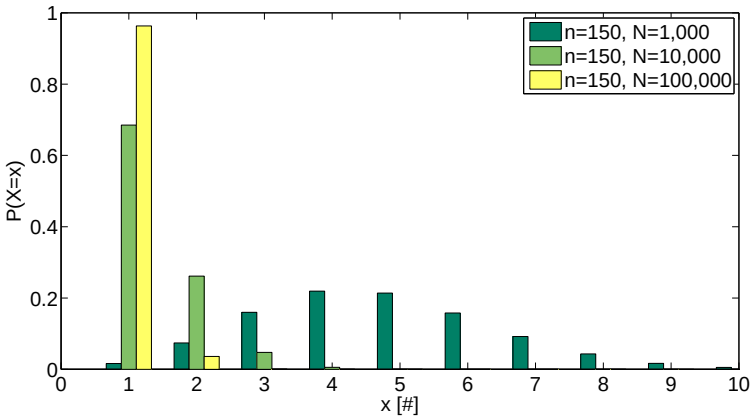
We therefore expect that on a global scale geographically-related correlated failures tend to show less impact on swarms and that failures occur closer to an independent and exponential distribution than we observed in the Free-traces.

However, in order to analyze this, a more comprehensive trace is necessary, involving gateways distributed over several continents and preferably collected over a longer time period.

[1] We asked for anonymized data from a big German ISP about correlated failures in their infrastructure.



(a) Increasing n, Constant N, C=25



(b) Constant n, Increasing N, C=25

Figure 5.7: Probability For Observing x Correlated Failures Within a Swarm

5.3.2 Geographically-Diverse Correlated Failures

Above we have identified that malicious attacks like the use of computer viruses result in failures that are unrelated to a geographical origin. Additionally, attacks via viruses show a very crucial characteristic: Not only can viruses delete substreams stored on other storages nodes, they can also delete or corrupt the on-site copy on a swarm leader. While hashes can be used to detect modifications by viruses on storage nodes (as explained in Section 3.7.3), there is no countermeasure against modifications when a virus infects a swarm leader. Further, correlated failures due to attacks can potentially affect the whole federated network. The potential scope of such attacks basically only depends on the implementation of the virus.

As pointed out in [TSH+05], the only way to securely prevent changes to a

back-up in presence of viruses is to use immutable storage. This especially includes Write-Once-Read-Many (WORM) drives [Sto90], which allow data to be written only once on the physical layer. Most popular for this purpose are optical disks such as CD-R, DVD-R, or BD-R. Once data are stored using a WORM drive, they cannot be modified or erased any more, neither by attacks nor by accident.

This underlines that storing a back-up in a distributed system does not fulfill all criteria to achieve a fully reliable back-up.

5.4 Testing Back-Up Durability

We have seen in the previous sections that the assumption of failures to occur independently and the time between failures to be exponentially distributed does not really hold. For our system it is crucial to support a target durability d for the period t_{iso}, in which no maintenance is performed (as explained in Section 3.5.2). Since we assume failures to occur according to the Poisson distribution, we might encounter problems in our system. For this reason we analyze whether the correlation of failures in the Free-traces is decisive for the target durability provided by our system.

In the following we first explain the experimental setup used to test the durability of back-ups stored on gateways. Thereupon, we present the results and discuss their implications.

5.4.1 Experimental Setup

In our system we split our back-up over $n = k + h$ storage nodes. We choose h such that after the time period t_{iso}, with probability d, we still have k alive storage nodes left. Having k alive storage nodes after t_{iso} means we have at least k alive storage nodes within t_{iso} in order to recover the back-up.

In order to achieve a target durability of $d = 0.999999$, we need to find a value h which satisfies the following formula:

$$\sum_{i=k}^{k+h} \binom{k+h}{i} (e^{-t_{iso}/\tau})^i (1 - e^{-t_{iso}/\tau})^{(k+h)-i} > 0.999999 \tag{5.4}$$

We take the average life time of gateways in the Free-traces of $\tau = 7.43$ years, which we have calculated in the previous section. Since the Free-traces span an observation period of 7.5 months, we limit our experiment to $t_{iso} = 182$ days,

which is about half a year. With these parameters we use a binary search to probe formula 5.4 for the lowest value h which still supports our target durability.

In the following table we show the necessary value of h depending on different values k. We further provide the redundancy factor $r = \frac{k+h}{k}$ and the resulting theoretical durability d.

k	h	r	d
10	8	1.80	0.99999942
20	11	1.55	0.99999976
50	16	1.32	0.99999926
100	24	1.24	0.99999963
200	36	1.18	0.99999900
500	68	1.13	0.99999900

Table 5.1: Different Values for k and Their Corresponding Values for h, r, and d.

We simulate a single back-up by drawing at random $k + h$ different gateways of the Free-traces. These gateways represent the storage nodes that make up a single swarm in our architecture. We do not add additional gateways to a swarm during an experiment because we want to test its survivability without maintenance. We further assume that all $k + h$ gateways already hold a sub-stream at the beginning of the experiment $t = 0$ so that we do not simulate transfers to gateways.

We use the timestamps in the Free-traces in order to determine whether a gateway is alive or not. Only if a chosen gateway is on-line at an arbitrary point in time after t_{iso} do we consider it to be alive after the time period t_{iso}, as shown in Figure 5.8.

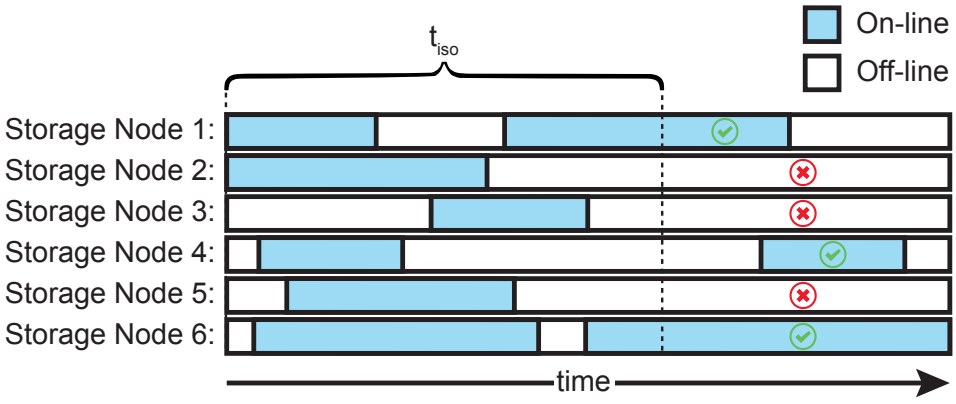


Figure 5.8: Determining Alive Storage Nodes

Given at least k gateways are alive after t_{iso}, the back-up can be reconstructed

within t_{iso} and therefore the back-up has survived. For a single experiment we formalize this as follows:

$$X = \begin{cases} 1 & \text{if } |\text{alive gateways}| \geq k \\ 0 & \text{if } |\text{alive gateways}| < k \end{cases} \tag{5.5}$$

According to the law of large numbers [Bil95] the arithmetic mean $\overline{X}_q = \frac{\sum_{i=1}^{q} X_i}{q}$ of the results obtained by a large number q of experiments converges to the expected value, which in our case is the durability d. Hence, for any positive number ε applies:

$$\lim_{q \to \infty} \mathrm{P}(|\overline{X}_q - d| > \varepsilon) = 0 \tag{5.6}$$

In total, we simulate $q = 10^8$ experiments for each value of k shown in Table 5.1. We keep track of the outcomes of the experiments in order to determine the arithmetic mean $\overline{X}_q$.

5.4.2 Results and Conclusion

In Table 5.2 we show the result of our experiments. The number of lost back-ups is the number of experiments in which we observe less than k alive gateways after t_{iso}.

k	d	# back-ups lost	$\overline{X}_q$
10	0.99999942	24	0.99999976
20	0.99999976	12	0.99999988
50	0.99999926	28	0.99999972
100	0.99999963	40	0.99999960
200	0.99999900	20	0.99999980
500	0.99999900	12	0.99999988

Table 5.2: Number of Back-ups Lost and the Resulting Artithmetic Mean $\overline{X}_q$ for $q = 10^8$ experiments

We see that the number of experiments for which we have less than k alive storage nodes after the period t_{iso} is very close to the expected values according to d shown in Table 5.1. In fact, the expected values for d are slightly lower than the mean values observed in our experiments. As explained in Section 5.2.2, we tend to count more permanent failures to determine τ than actually occur. With higher precision for determining permanent failures, it is possible that a

lower redundancy rate can be used for back-ups in the Free-traces. In this case we also expect to see some impact on the durability due to correlated failures. Traces collected over a longer observation period therefore lead to more accurate results than the Free-traces. To the best of our knowledge there is currently no trace in a similar environment over a longer duration. Hence, we leave a more accurate analysis by using the same methodology as future work.

From the experiment performed we can, however, conclude that by using the given redundancy levels, the simulated back-ups effectively show the desired number of alive storage nodes after the time span t_{iso} with high probability. This is the case even despite the slight correlation of failures, which we have determined in the beginning of this chapter. We therefore conclude that the environment observed in the Free-traces allows us to rely on our proposed maintenance mechanism, which focuses on durability over a given time span.

We also see that the required redundancy level is lower than 1.5 for higher values of k. In contrast, storage systems that focus on data availability typically require higher redundancy levels [HMD05, Ke00, BR03] and additional bandwidth for downloads from the system before new redundancy can be uploaded.

Chapter 6

Back-Up Simulation

> "A mind is a simulation that simulates itself.."
>
> Erol Ozan,
> Professor at ECU

6.1 Introduction

Our distributed architecture involves several participants in the network, which interact with each other. In this chapter we perform simulations to better understand the characteristics resulting from this architecture.

First, we want to compare our swarm-based architecture with a cloud-based back-up solution, such as services provided by Dropbox [Dro14a] or Amazon S3 [Ama14a]. We use cloud-based services as a basis for comparison since they represent the optimal but costly solution with the minimum possible requirements concerning the resources of gateways. We further analyze the swarm-based back-up for some characteristics such as the time required for back-up creation and the resulting swarm sizes, which in our architecture also imply the amount of uploaded data. We also look closer at the timeout period used by our failure detector and its impact on the total data uploaded into the system. At last we observe the bandwidth usage over time for different values for the number of original fragments k.

Throughout this chapter, we use the Free-traces to determine the on-line states of gateways and simulate the evolution of back-ups over time.

6.2 Time Required for Back-Up Creation

In this section we compare the performance for back-up creation of our swarm-based architecture to a cloud-based architecture. Cloud storage manages redundancy internally within the data center so that the client only needs to upload the amount of data to store. In contrast, in our swarm architecture we inject additional redundancy by the swarm leader and, thus, need to upload more data. Thus, it is clear from the outset that the time until the back-up completes cannot compete with the cloud-based solution.

In the following, we denote both, the client in the cloud scenario and the swarm leader in the swarm-based scenario, as back-up source.

In order to show the difference between the two approaches, we simulate multiple back-up uploads in the following.

6.2.1 Common Back-Up Scenario

For both, the cloud-based and the swarm-based back-up scenario, we simulate a back-up that is worth $S_t = 100$ GiB of data. We only consider a single snapshot so that no additional data is added over time. We assume an upload data rate at the back-up source of 512 kbit/s, which is a common up-link capacity for ADSL connections offered by ISPs in 2014.

The metric for comparing the performance of both scenarios is the proportion of incomplete back-up simulation passes to the total number of complete simulation passes at a particular point in time.

For the cloud-based back-up, this is the fraction of simulation passes that have not already completed the upload of the full 100 GiB. For the swarm-based back-up, this is the fraction of simulations with currently less than $k + h$ alive gateways, each of them holding a complete substream.

For the simulations of transfers we simulate progress only when both participants are on-line at the same time. We continuously keep track of the amount of data that is scheduled for transfer to a dedicated sink.

In order to determine the impact of the back-up source's availability on the upload progress, we choose the back-up source according to its availability value α_s. We measure this availability over the whole simulation period. As gateways with very low availability do not overcome the initial upload phase of the back-up, we start with $\alpha_s \geq 0.2$.

For both, the cloud-based and the swarm-based back-up, we simulate 10,000 back-up creation processes. This high number allows us to identify general trends in the simulation.

6.2.2 Cloud-Based Back-Up

In the following we explain the setup and the results for the cloud-based back-up scenario.

Simulation Setup

For the cloud-based simulator we choose a single gateway from the real world traces in order to simulate the client uploading its back-up. For each of the simulated back-up creation processes we randomly choose a different gateway from the Free-traces.

We take a single endpoint as the recipient of the back-up. This endpoint is never off-line, thus it represents the cloud with its high availability. An interrupted data transfer to the cloud can be resumed after the client is on-line again, so that no transferred data is lost.

Simulation Results

Figure 6.1 shows the ratio of incomplete back-ups over time. Due to the logarithmic scale, the end of a line indicates that all back-ups are completed.

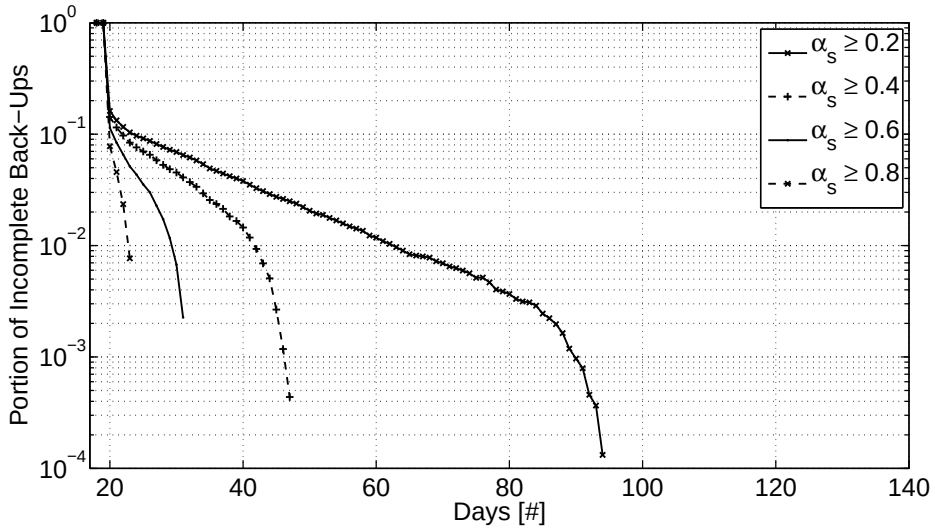


Figure 6.1: Back-Up to the Cloud with Different Availability α_s of the Back-Up Source; $S_t = 100$ GiB at 512 kbit/s

We see that the first back-ups complete after $\frac{100 \text{ GiB}}{512 \text{ kbit/s}} \approx 19$ days, which is a lower bound when data are transferred without interruption. After 20 days we

already observe 90% of the back-ups to be completed if the client has at least an availability of $\alpha_s = 0.8$. Only four days later, all simulated back-ups for this group are completed.

All clients with an availability of 20% or more finish their back-ups after 94 days.

6.2.3 Swarm-Based Back-Up

We now have a closer look at how the back-up upload performs in the scenario for our swarm-based back-up.

Simulation Setup

For the swarm-based back-up, we need to consider the maintenance procedure described in Section 3.5. The swarm leader needs to initiate new uploads to storage nodes as long as fewer than $k + h$ alive storage nodes holding a redundancy stream are observed. We choose storage nodes from the Free-traces by random without restriction with respect to their availability. We do, however, verify that they are on-line at the point in time they are requested. This way we enable swarm leaders to start transfers right away since both participants are on-line at the same time. Data transfers are simulated over time, taking off-line periods of both participants into account. While a swarm leader is uploading, it keeps three uploads open at the same time to cope with temporary unavailability of receivers. Similarly to the cloud-based back-up, we resume interrupted uploads.

In this simulation, we use the parameter $k = 100$, as recommended in Section 4.3.3. Consequently, a single transfer to a storage node is worth the size of one substream, which is $\frac{S_t}{k} = \frac{100 \text{ GiB}}{100} = 1$ GiB in our case. We use $t_o = 1$ day for the failure detection and target $t_{iso} = 182$ days. Throughout this chapter we further use the average gateway lifetime τ we have determined for the Free-traces in Section 5.2.2. As a result, the maintenance procedure targets $n = 124$ substreams in the swarm, which results in a redundancy factor of $r = 1.24$.

Simulation Results

For the swarm-based back-up simulation we see the resulting portion of incomplete back-ups over time in Figure 6.2.

Compared to the cloud-based back-up, the minimum duration for back-up creation increases linearly with the redundancy factor and equals $\frac{1.24 \cdot 100 \cdot 1 \text{ GiB}}{512 \text{ kbit/s}} \approx 24$ days. For a minimum swarm leader availability of 80%, already 90% of the swarms complete their back-ups after 25 days, while after 30 days the back-ups

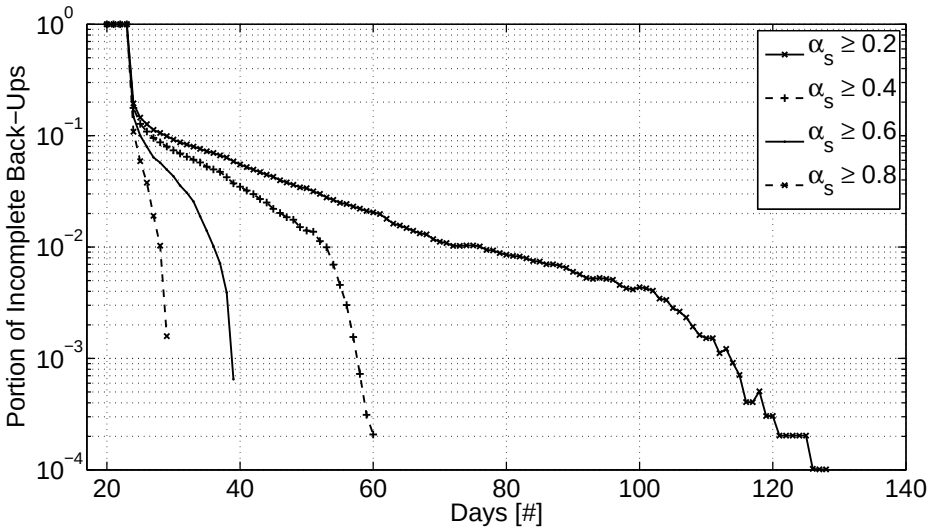


Figure 6.2: Back-Up Using Swarms with Different Availability α_s of the Swarm Leader; $S_t = 100$ GiB at 512 kbit/s, $k = 100$, $h = 24$

complete for all of them. For the swarm leaders with lower availability, it takes up to 128 days to reach the point until all back-ups are successfully completed.

In contrast to the cloud-based back-up, we observe that the portion of incomplete back-ups is not monotonically decreasing over time. This is because in the swarm-based back-up scenario gateways also leave the system. Therefore, unless the maintenance algorithm finishes the upload of additional data, the targeted redundancy level is not present in the swarm. We further see that the resulting local peaks only have a low amplitude. Correlated failures (as analyzed in Chapter 5) therefore only show little impact on the portion of incomplete back-ups.

In Figure 6.3 we show how the average swarm size of all simulations evolves over time. We see that as soon as the initial upload completes, the maintenance process reaches a steady state and, henceforth, barely adds new storage nodes. As mentioned in Section 3.5, this is only necessary for permanent failures in the long term due to the reintegration of reappearing storage nodes. We see that in the long term the availability of the swarm leader does not influence the swarm size: independent of the swarm leaders' availability, the swarms converge to the same swarm sizes in the long term.

In Figure 6.4 we see that swarm leaders with higher availability already achieve a higher portion of complete back-ups with fewer storage nodes in a swarm. This can be explained by the fact that for progress in transfers we need both participants to be on-line, the storage node as well as the swarm leader. Since

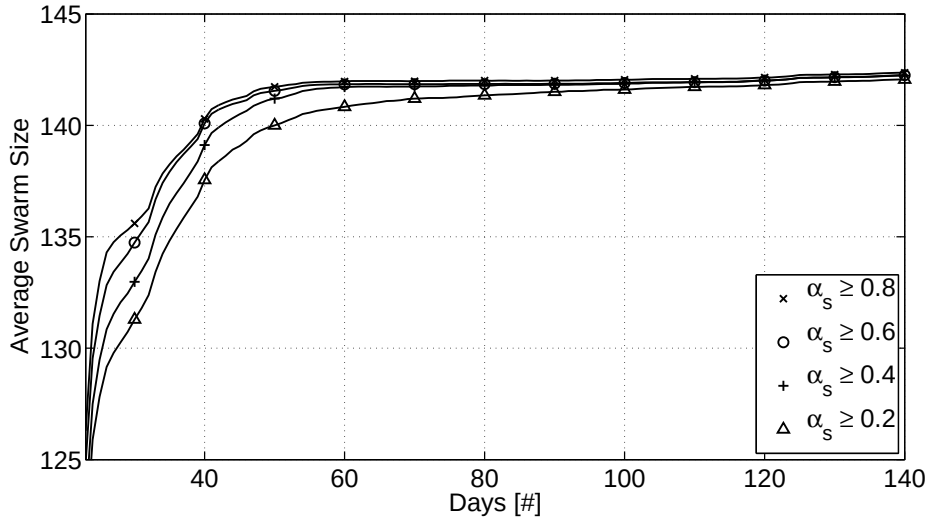


Figure 6.3: Average Swarm Size over Time with Different Availability α_s of the Swarm Leader; $S_t = 100$ GiB at 512 kbit/s, $k = 100$, $h = 24$

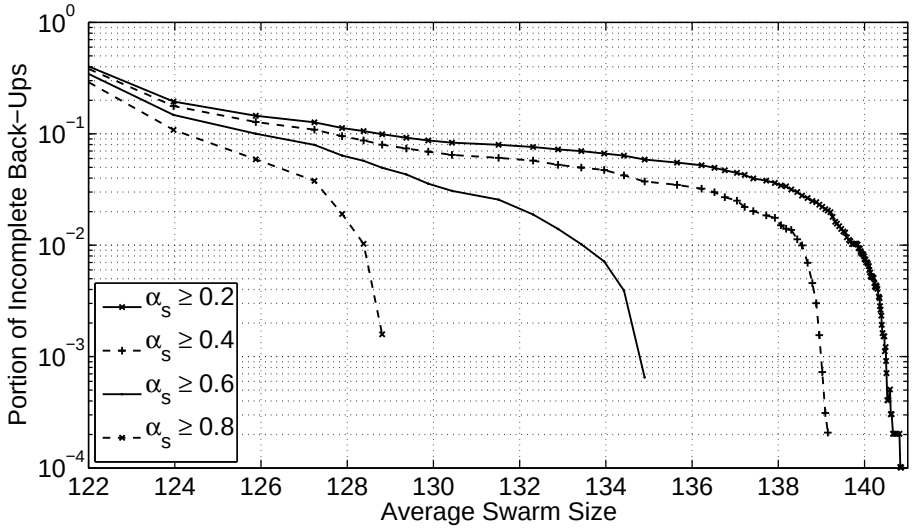


Figure 6.4: Failed Swarms Depending on Swarm Size with Different Availability α_s of the Swarm Leader; $S_t = 100$ GiB at 512 kbit/s, $k = 100$, $h = 24$

a swarm leader with low availability is more likely to be off-line, this results in more interrupted transfers. In order to facilitate progress, our maintenance procedure (explained in Section 3.5) adds new storage nodes in case of interrupted transfers. A swarm leader with low availability therefore tends to add more storage nodes in its swarm while the maintenance process is running.

Over time, however, as mentioned before in the context of Figure 6.3, the average swarm size converges to the same level.

6.2.4 Conclusion

From the simulations above we draw several conclusions. In both, the cloud-based and the swarm-based back-up, a higher availability of the back-up source is important to achieve a fast back-up. Obviously, whenever the back-up source is off-line, there cannot be progress in the back-up upload. On the other hand, we see that the availability of the storage nodes in the swarm-based back-up shows only little relevance. Although in our simulation we have no restrictions on the availability of storage nodes, we constantly see progress in the back-up creation. This is due to the flexibility of our parallel uploads: whenever a storage node is temporarily off-line, we can proceed with the upload to another storage node.

Finally, the major reason why the required time for back-up creation is higher for the swarm-based back-up is that we need to upload additional redundancy. In fact, the minimum time required to complete the swarm-based back-up is close to the minimum time required for the cloud-based back-up multiplied by the redundancy factor.

6.3 Influence of the Timeout Period

In this section we take a closer look at the effect of the timeout period t_o, which we have introduced in Section 3.5 for our maintenance procedure.

6.3.1 Costs Separated into Two Components

The timeout t_o decides how efficiently our maintenance procedure operates. The choice over its duration entails the costs, which can be separated into different components, as also described in [Wea06]. In our system, we face the following costs:

- **Costs due to transient failures**
  When t_o is low, we declare storage nodes to be in the dead state and consider their data to be lost at an earlier point in time. We generally react to lost data by triggering the upload of additional redundancy in order to achieve the targeted redundancy level again. However, although a storage node may have left the system for several days, there might still be the chance that it reconnects to the system so that its data can be reintegrated. In consequence, the earlier we declare storage nodes dead, the more do we face increasing costs for maintenance due to transient failures.

- **Costs due to permanent failures**
  As t_o is a component of t_{iso} in our system, by increasing t_o we also increase t_{iso}. For higher t_{iso}, according to Equation 3.1, we further need a higher redundancy level in order to provide the user with a certain degree of durability for a targeted period. A higher redundancy level in turn means we include more storage nodes into a swarm. Since these storage nodes eventually fail permanently, for high t_o, we face increasing costs due to permanent failures. As in our case we can only add a discrete number of storage nodes in order to achieve an increased redundancy level, the costs due to permanent failures increase stepwise with respect to increasing t_o.

In order to minimize the overall maintenance costs, we need to find a timeout t_o which minimizes the sum of both above-mentioned costs.

The optimal value for t_o therefore depends on session durations and the average life time of participants in the system.

6.3.2 Optimal Value for the Free-Traces

We further analyze the above-mentioned costs on the Free-traces by using a simulation.

Again we break the back-up into $k = 100$ substreams and use the same parameters as mentioned above. This time, however, we use a varying value for t_o so that t_{iso} increases. Since we still target a durability of 0.999999 over t_{iso}, we need to increase the number of additional substreams k accordingly.

In order to determine the optimal period t_o, we measure the final swarm size after 180 days. Since the swarm size indicates how much data has been uploaded to storage nodes, it allows us to infer the overall costs for maintenance. In order to smooth the result over the data set, we calculate the average swarm size over 10,000 simulations.

Figure 6.5 shows the resulting average swarm sizes. We see that in the beginning

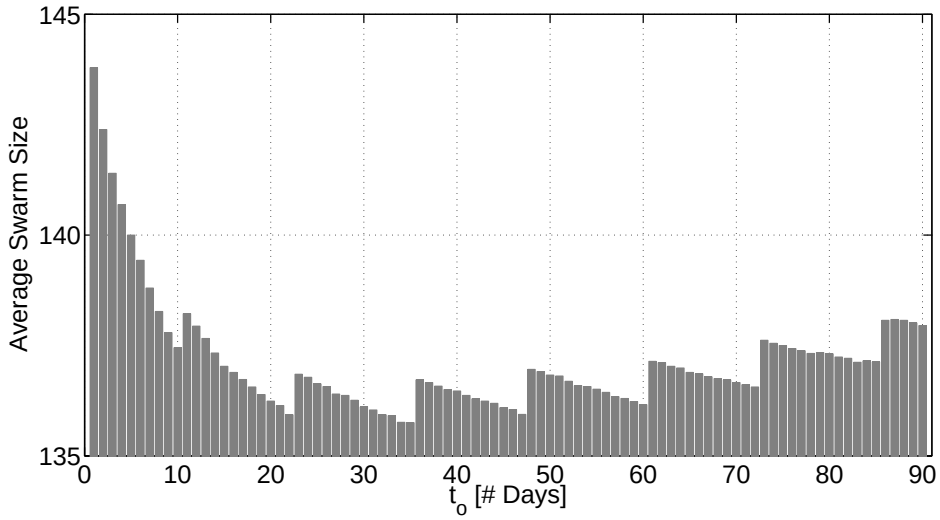


Figure 6.5: Influence of Observation Period on the Average Swarm Size After 180 days

the overall costs decrease quite rapidly with increasing t_o. This is because with higher t_o we can reintegrate more storage nodes that transiently left the system.

Additionally, the graph shows characteristics resulting from the stepwise increase of the redundancy level according to t_o. In our case, we need to increase the redundancy level the first time by using a timeout period of more than $t_o = 10$ days.

The graph therefore shows both of the aforementioned costs: the costs due to transient failures, which decrease with increasing t_o, and the costs due to permanent failures, which increase stepwise with t_o.

For our scenario with gateways from the Free-traces, we find the global optimum for a timeout period of $t_o = 35$ days. This means that for running our back-up system in the same environment as for the Free-traces, the maintenance costs

are optimal once a storage node is declared *dead* after it has been continuously off-line for 35 days.

6.4 Visualization of Bandwidth Usage

In the following we present exemplary simulations of single back-up creation processes. We keep track of the bandwidth usage over time and discuss the activity over time. Since we use different values for k we require different redundancy factors for the back-ups and have different substream sizes, which are defined by $\frac{S_t}{k}$. The size of a substream decides how much time is required to finish the maintenance process.

First we explain the experimental setup used and subsequently present our observations.

6.4.1 Simulation Setup

In this simulation we record the upload bandwidth usage on a swarm leader for uploading and maintaining a back-up worth $S_t = 100$ GiB. We use a period of 35 days for t_o. In order to guarantee a period $t_r + t_d$ of half a year for downloading the back-up, we use $t_{iso} = 217$ days.

For this experiment we use an artificial swarm leader that is always on-line. In consequence, off-line periods of the swarm leader do not hinder progress in back-up creation and maintenance. This way, we further increase the comparability of the different simulations.

According to our maintenance procedure (see Section 3.5) we add storage nodes to the swarm as long as the maintenance procedure is running and we observe fewer than three active uploads to storage nodes. As all storage nodes in the swarm are eventually synchronized, we expect to observe transfers worth a total of at least $k + h + 2$ substreams for the initial upload.

We perform four different simulations, for which we gradually increase the parameter k to the values 10, 50, 100, and 200.

6.4.2 Results

Below we present the results for using the four different values for k, which imply a different substream size for each simulation.

Simulation 1

For the first simulation we set $k = 10$ so that our maintenance procedure targets $k + h = 18$ substreams on different storage nodes in order to provide durability over the period t_{iso}. For $k = 10$ one substream is worth $\frac{S_t}{k} = 10$ GiB of data.

In Figure 6.6 we see the upload data rate per hour over a period of 180 days. In this pass we observe that 21 substreams are uploaded within 40 days for the initial back-up. After 150 days a maintenance process is triggered, resulting in the upload of 4 substreams to new storage nodes. It takes about 8 days to finish these transfers.

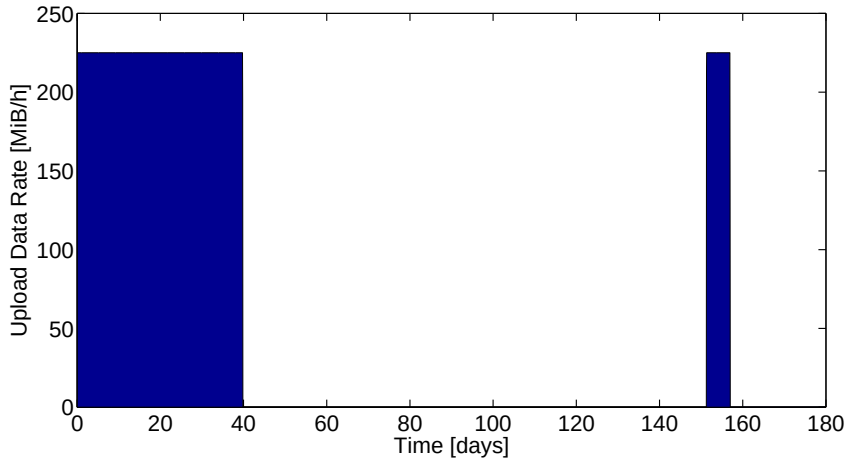


Figure 6.6: Simulation 1: Upload Data Rate Over Time for $k = 10$, $h = 8$

Simulation 2

With $k = 50$ we can operate with a lower redundancy factor than before. We target 68 substreams in the system, each with a size of 2 GiB.

As shown in Figure 6.7, the initial upload now only takes 27 days due to the lower redundancy level. The upload process ends up with a total of 71 substreams in the system. After the initial upload, we observe 3 transfers of durations shorter than a couple of hours. These transfers fully synchronize storage nodes that are already members of the swarm but left the system temporarily. After about 130 days two maintenance processes are triggered. Each of these maintenance processes takes about 27 hours for the upload of three substreams.

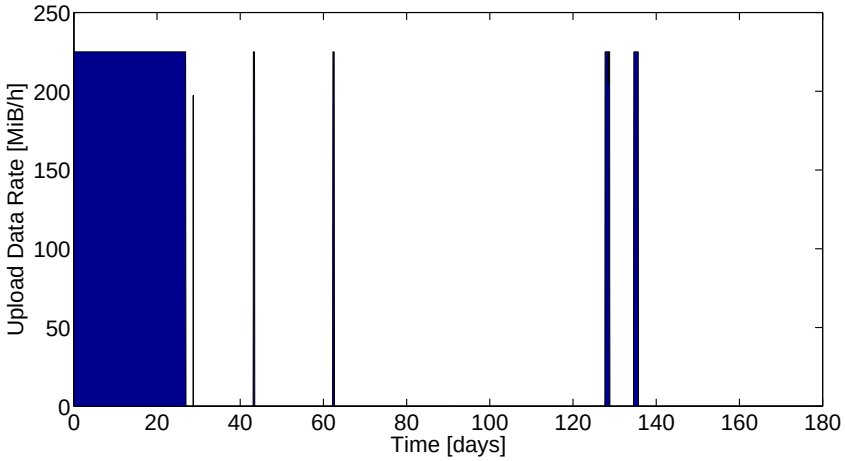


Figure 6.7: Simulation 2: Upload Data Rate Over Time for $k = 50$, $h = 18$

Simulation 3

For $k = 100$ the maintenance procedure focuses on 126 substreams in the system, which again results in a lower redundancy factor than for lower k. One substream has a size of 1 GiB so that it takes about 4.5 hours for the transfer to a storage node. We show the results of this simulation in Figure 6.8. The

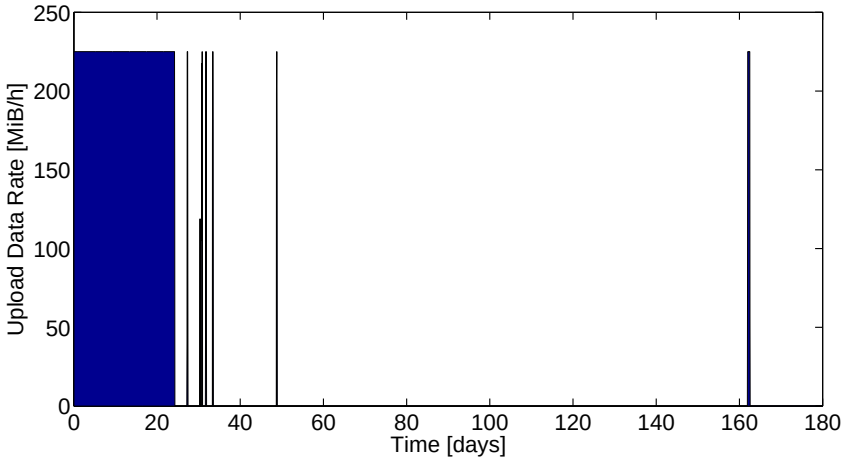


Figure 6.8: Simulation 3: Upload Data Rate Over Time for $k = 100$, $h = 26$

initial upload finishes after about 24 days and is followed by six consecutive transfers in order to fully synchronize storage nodes that turned off-line during

the transfer before. Due to these consecutive transfers, there is more redundancy in the network than necessary. In consequence, only after about 160 days do we need to upload additional substreams, which takes less than 14 hours for three substreams in this case.

Simulation 4

In the last scenario we choose $k = 200$ so that the maintenance procedure focuses on 240 substreams in the system. In consequence, one substream has a size of 0.5 GiB.

We are able to complete the initial upload after 23 days, which is one day less than for the previous simulation. After the initial upload, as illustrated in Figure 6.9, we observe eight subsequent transfers to finish the synchronization to storage nodes. Apart from these short transfers with a duration of about two hours, we observe two maintenance processes, one at day 88 and one at day 142. Both involve the upload to three new storage nodes and finish within 7 hours.

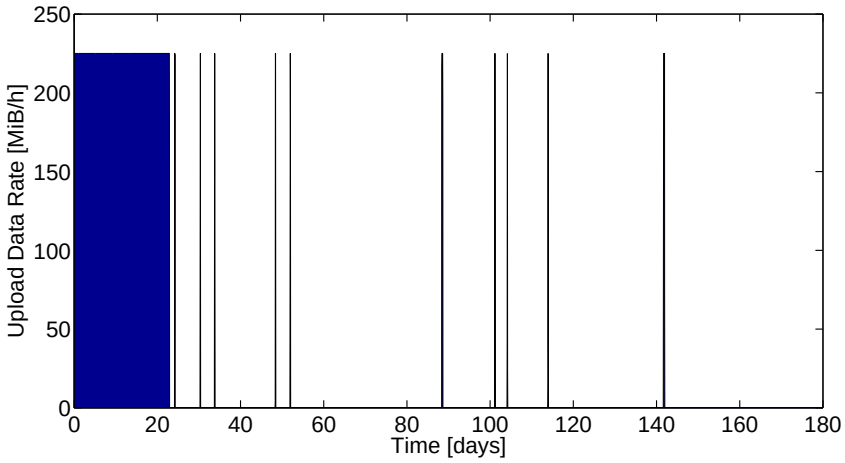


Figure 6.9: Simulation 4: Upload Data Rate Over Time for $k = 200$, $h = 40$

Generally we see that with higher k the initial upload finishes earlier due to the lower redundancy factor. This effect, however, becomes less decisive with k higher than 100 because of the high number of involved storage nodes. This increases the number of storage nodes that are unsynchronized after the initial upload finishes. Over time we try to synchronize these storage nodes so that the initial upload results in more data transferred than required.

6.4.3 Conclusion

As mentioned in Section 3.6, for higher k we should also face more frequent maintenance over time. Our simulations do not show this trend due to the additional storage nodes included for the initial upload. However, we see that as far as the duration of the maintenance procedure is concerned, it is advantageous to have smaller substreams. While the maintenance using high k can be finished relatively fast, the maintenance for low k requires a period of about one week. This can be of interest to schedule maintenance to periods in which bandwidth on the gateway is typically available, so that maintenance is, e.g., performed over night. However, a very high k also leads to the situation in which we face a higher number of unsynchronized storage nodes after the initial upload. Due to this, we upload more substreams than actually are required. This effect is influenced by the availability of storage nodes in a swarm so that its impact is dependent on the individual characteristics of the network.

Chapter 7

Conclusion and Perspective

In this chapter we first synthesize and conclude our work by focusing on its contributions. Finally we propose potential future research directions.

7.1 Conclusion

This work introduces a back-up service that allows a user to store a snapshot-based back-up in a distributed way. We profit from the common interest of participants in storing back-ups by mainly leveraging the resources provided by the users themselves. However, a trusted central instance such as our tracker still turns out to be a good way to reduce communication overhead and vulnerability, e.g., to Sybil attacks. With current failure-tolerant architectures from the area of cloud computing, we can even design a scalable tracker without introducing a single point-of-failure into the system.

The main contribution of this work is a proof of concept that confirms the feasibility of such a service. Another contribution is the concept of index files, which are a new way of distributed data organization with respect to the alternation of file systems over time. Further, the division of the federated network into swarms eases the monitoring of gateways and reduces the metadata on the tracker required for data localization to a very low grade. In fact, the load on the tracker is independent of the amount of data stored in the system, which is a desirable property, especially in the context of increasing amounts of data in the future.

The architecture comes along with a key management that allows full data encryption on the user side in order to ensure data confidentiality, even against the people who provide the service of the tracker. We further show that it is beneficial to handle files of different sizes in a different way. Embedding small files into index files increases the overall efficiency in our system with only minor

additional storage space requirements. According to the trend we see in recent file size distributions, which tend to store more and more bytes in big files, this approach shows good prospects for the future. For big files we consider an interleaving scheme, which allows us to benefit from partial transfers and limits the required memory footprint.

With our system we also question the established approach to require data in the system to last forever. Instead, we focus on enabling a user to recover all data in case of local data loss. We underpin this strategy by an analysis based on traces collected in a similar environment. Although the underlying failure model is an approximation for the occurrence of failures in the system, we see that recovery is still possible at low redundancy levels. Consequently, we can reduce the bandwidth requirements in the system to a very low level, which in the long term is equivalent to the rate at which we lose data in the system.

The result is an architecture that supports automated and snapshot-based back-up creation; first from user devices to the gateway and finally to the federated network. Data loss due to events such as fire and flooding can therefore be avoided. We are strongly confident that our system can be deployed in the real world and contribute to achieve an affordable and confidentiality-conserving back-up for everyone.

However, for an advanced back-up practice, we still recommend to create occasional back-ups on immutable storage such as optical devices. As long as storage systems are connected to the Internet, they always remain vulnerable to computer viruses. Relying on multiple methods for back-up creation always improves chances that data can be recovered safely.

7.2 Perspective and Future Work

Our work can be extended in various ways, which we depict in the following.

First, in case the tracker leaves the system, our best-effort approach to interconnect gateways could be extended by a flooding mechanism as used in Gnutella [PSAS01]. That way, dynamic IP addresses can be replicated in the federated network without relying on any instance in the DNS system. Such an approach is known to be effective against network partitioning, yet it also entails a high messaging complexity that can potentially overload the system.

Second, as we have seen in Chapter 6, a high number of original fragments for a file increases the number of unsynchronized storage nodes after the initial upload of a snapshot. When we upload subsequent snapshots, this potentially leads to the inclusion of further storage nodes. Therefore, it may be interesting to analyze how the average gateway availability α and the timeout t_o influence the size of a swarm in the long term.

Third, we designed our system to support convergent encryption. This is a promising technique to reduce the overall storage space required by the system, since equal file content of different users can be reduced to a single instance of the file content. Our presented system design uses this way of encryption but does not match equal contents of different users since they are stored in different swarms. In this scenario the matching procedure requires knowledge over different swarms, which is an expensive task in practice [ZLL13]. However, regarding the trend that most bytes are stored in big files, it seems a good approach to perform deduplication on at least those. The tracker therefore could determine the observed frequency of big files and announce that certain files are not required to be uploaded into swarms. In this case, the number of replicas in the federated network is high enough so that file recovery can be assured by simply using one of the many already existing replicas.

Appendix A

Synthèse en francais

Introduction

Dans ce chapitre, nous introduisons, de manière générale, la création de sauvegardes, ainsi que les raisons pour lequelles il est désirable de stocker ces sauvegardes de manière distribuée. Nous présentons aussi le scénario qui est à la base de notre architecture, à savoir les passerelles résidentielles. Finalement, nous donnons une vue d'ensemble des contributions de cette thèse.

Un besoin de sauvegardes

Il est généralement convenu que les données informatiques représentent une partie de plus en plus importante dans notre vie quotidienne. Pour cette raison, ces données deviennent de plus en plus importante à nos yeux: perdre toutes les photos de familles et autres documents importants est un scénario auquel personne ne souhaite être confronté.

Malheureusement, les équipements électroniques, comme les disques durs, n'ont pas une durée de vie infinie. Des données stockées peuvent devenir partiellement, ou même complètement, inaccessibles sans signes précurseurs. Dans de nombreux cas, les données enregistrées sur ces périphériques peuvent être récupérées par des sociétés spécialisées dans la récupération de données [The14a, Kro14]. La procédure est, cependant, très couteuse [The14b] et peut prendre un temps important.

Il est donc nécessaire de trouver des techniques qui permettent aux utilisateurs de récupérer leurs données de manière autonome. Une approche manuelle est de créer des volumes de sauvegarde en utilisant, par exemple, des CD-ROM ou des disques durs externes. Cependant, cette approche est fastidieuse et peut résulter en perte de données. En effet, les sauvegardes doivent être créées de

manière régulière et non automatisée. L'utilisateur peut donc oublier de créer une nouvelle sauvegarde. Un autre facteur de risque est le vol, ou les catastrophes naturelles impliquent une perte de données, car celles-ci ne sont stockées qu'à un seul endroit. De plus, les utilisateurs possèdent souvent plusieurs ordinateurs sur un même réseau résidentiel, et il est essentiel d'établire une technique plus complexe afin de préserver les données de toutes ces machines.

Pour ces raisons, une technique qui récupère localement les données et les distribue à différents endroits est utile. Biensûr, cette méthode doit prendre en compte la confidentialité des données: stocker les données d'un utilisateur à un emplacement distant ne doit pas impliquer que ces données soient accessible à des personnes tierce.

Pourquoi ne pas utiliser le cloud?

Actuellement, le cloud est de plus en plus utilisé comme service centralisé de stockage de données. Par exemple, les services Dropbox [Dro14a] et Amazon S3 [Ama14a] sont très populaires. Les données à sauvegarder sont transférées vers ces services, et le système interne au service garanti que les données ne seront pas perdues. Etant donné que la croissance de la densité des données [Wal05] résulte en une diminution du prix des unités de stockage (voir Figure A.1), stocker une sauvegarde dans le cloud parraît être souhaitable.

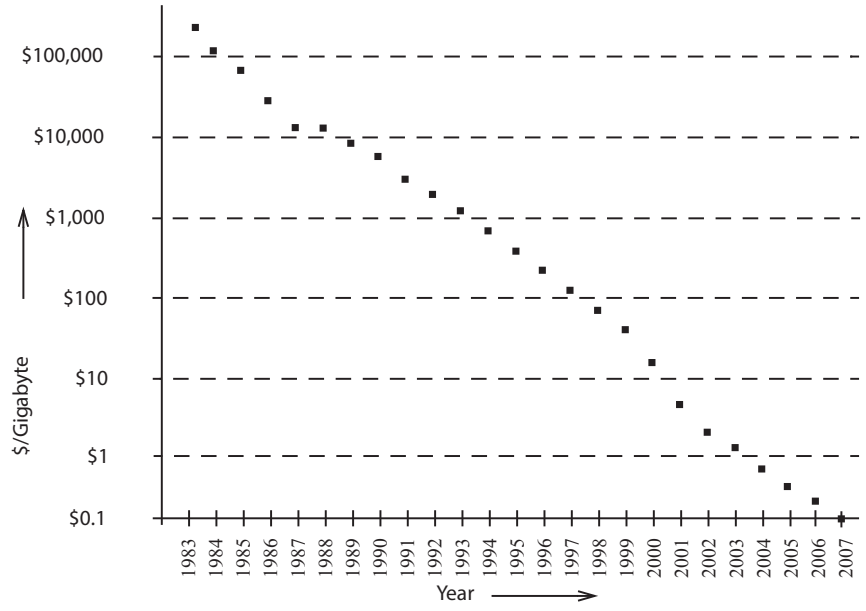


Figure A.1: Evolution du prix d'un disque magnétique; source [SK09]

Cependant, Amazon, par exemple, a modifié sa structure de prix [Ama14b] afin de mieux prendre en compte les coûts générés, qui incluent le prix de l'énergie nécesaire à l'accès et au transfert des données. Ces couts énergétiques ont globallement augmentés ces dernières années [U.S14], et sont devenus un facteur de coût très important pour les sociétés de cloud. En plus, sauvegarder des données dans le cloud rend ces données accessibles aux personnes tierces, par exemple à des employés de la société, voire même au public global dans certains cas [Dro14b], ce qui est, bien évidemment, contraire aux intérêt des utilisateurs.

En revanche, nous pouvons tirer parti de l'intérêt mutuel des utilisateurs de stocker leur sauvegarde dans un endroit géographique différent. Utiliser les ressources d'autres utilisateurs pour stocker des sauvegardes ne génère aucun coût supplémentaire si ces ressources sont déjà disponible chez les utilisateurs. Ainsi, chaque utilisateur peut créer une sauvegarde, et ce gratuitement, sans se confronter à un éventuel enfermement propriétaire [RKYG13] (qui, par exemple, ralenti un utilisateur souhaitant changer de service suivant un changement de conditions d'utilisation). En chiffrant continuellement les données avant de les transmettre vers un autre endroit, nous pouvons garantir la confidentialité de ces données. Il est, dès lors, possible de conserver les intérêts des utilisateurs au premier plan, et d'offrir un service qui va plus loin que ce que les services cloud permettent.

Réseau fédéré de passerelles réseau

Dans ce travail, nous nous concentrons sur une architecture dans laquelle les paserelles résidentielles jouent un rôle majeur. Nous les utilisons pour interconnecter différents réseaux résidentiels via Internet, afin d'assurer notre service distribué. En effet, ce travail a été réalisé dans le cadre du projet Européen FP7, FIGARO [FIG14], qui s'appliquait à un tel environement.

La Figure A.2 illustre l'architecture globale. Elle est composée des éléments suivants:

- **Machines des utilisateurs**
  Les machines des utilisateurs sont des équipements hétérogènes, comme ordinateurs de bureaux, ordinateurs portables, téléphones, tablettes. Un utilisateur y stocke des données et les connecte au réseau résidentiel.

- **Réseau résidentiel**
  Un réseau résidentiel est un réseau local (LAN), tel qu'il est généralement déployé chez monsieur tout-le-monde. Il est connecté à toutes les machines utilisateurs, soit de manière cablée (Ethernet), soit sans fils (Wi-Fi).

- **Réseau fédéré**
  Le réseau fédéré comprend toutes les passerelles qui font partie du service

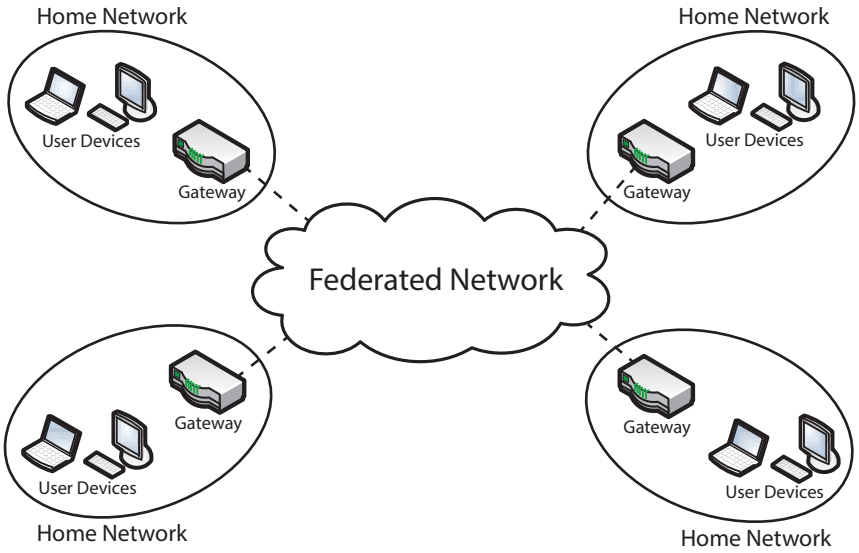


Figure A.2: Réseau fédéré

de sauvegarde distribué. Chaque paserelle est atteignable via Internet par les autres passerelles.

Dans ce scénario, nous avons deux vitesses réseaux différentes. Ceci est un facteur déterminant dans notre architecture. Les transferts de données à l'intérieur des réseaux résidentiels sont généralement rapides. Cependant, la vitesse de transmission de données vers Internet est généralement moindre. Notre architecture prendra donc en compte ces restrictions.

Axe de développement et contributions de cette thèse

Dans cette thèse, nous nous concentrons sur la faisabilité d'un système de sauvegarde distribué. Cela implique plusieurs challenges, comme l'extensibilité du système, sa résistance aux pannes et aux attaques, ainsi qu'une stratégie adéquate de localisation des données. Etant donné que nous stockons des données sur les paserelles participantes, nous devons faire face avec un stockage peu disponible, et avons besoin de techniques supplémentaires afin de garantir la confidentialité des données.

Les contributions de cette thèse sont les suivantes:

1. Nous donnons la preuve-de-concept d'un système de sauvegarde distribué, étant capable d'instantanés, et qui permet à un utilisateur de restaurer l'entièreté de son système de fichiers à un instant donné. Ce système comprend aussi l'administration et le respect des quotat de stockage.

2. Nous illustrons une façon d'utiliser les techniques de pointe afin d'include une instance centrale qui coordonne le placement de données dans le réseau. Cette instance est facilement extensible, résistante aux pannnes, et remplaçable, si elle quitte le système. De plus, elle n'est exposée qu'à une charge de travaille moyenne.

3. Nous créons une architecture en swarms, qui opère directement sur les fichiers, et qui limite la création de méta-données additionnelles autant que possible. De plus, elle nous permet de monitorer les données stockées dans une swarm de manière simple et efficace.

4. Nous proposons des moyens de transférer et stocker des fichiers de tailles distinctes dans un tel système.

5. Nous analysons le comportement de notre système en utilisant des traces provenant du monde réel. De plus, nous étudions l'impact des paramètres du système et comparons sa performance par rapport aux services cloud.

Présentation de l'architecture à base de swarms

Dans cette section, nous présentons un aperçu sur notre architecture générale du système. Nous expliquons comment les appareils dans notre scénario de réseaux fédérés interagir avec les autres et de définir leurs rôles et leurs fonctions. En plus de dispositifs d'utilisateur et les passerelles, nous introduisons une instance centralisée, dénommé le *tracker*. Nous organisons le réseau fédéré FN en *swarms* $SW \subset FN$, qui sont des ensembles individuels de nœuds de stockage pour chaque contrôleur du swarm pour stocker son back-up. Figure A.3 illustrates this architecture from the point of view of a single swarm leader. We subsequently discuss the shown entities in detail.

La Figure A.3 illustre cette architecture du point de vue d'un contrôleur du swarm unique. Nous discutons ensuite les entités représentés en détail.

La passerelle

Une passerelle est l'intermédiaire pour les deux réseaux, le réseau résidentiel et le réseau fédéré. Chaque passerelle a un nom d'hôte unique qui peut être résolu en une adresse IP en utilisant le DNS. Il stocke toutes les données à sauvegarder dans un réseau résidentiel et est en charge de mettre en ligne un back-up externe à d'autres passerelles qui font partie du réseau fédéré. D'autre part, il reçoit de tels fragments de données d'autres passerelles et est obligé de les tenir. Chaque passerelles joue donc le rôle d'un contrôleur du swarm et le rôle d'un nœud de stockage, comme expliqué ci-après.

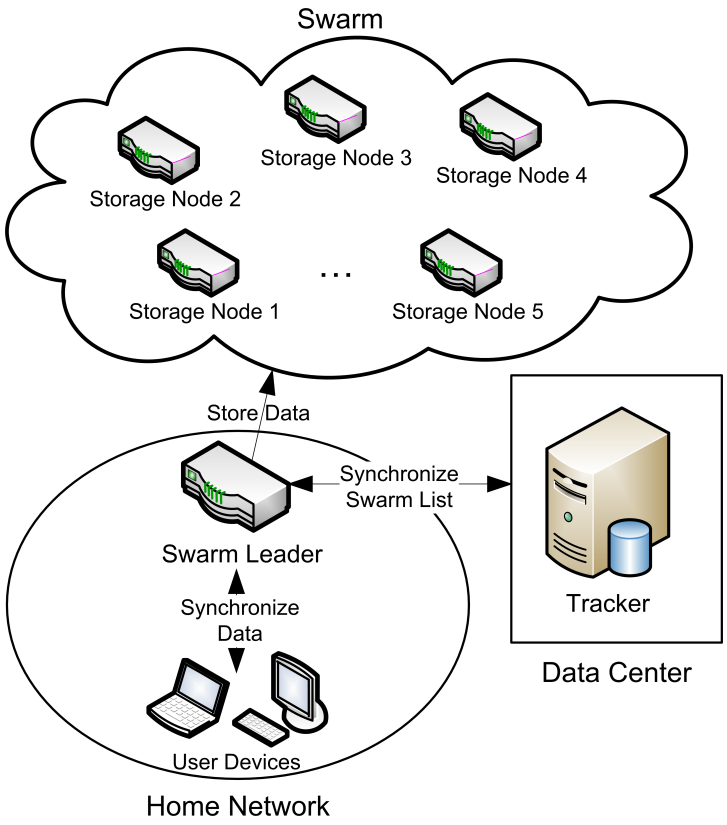


Figure A.3: General Architecture

Le contrôleur du swarm

Le contrôleur du swarm conserve une copie de toutes les données d'appareils au réseau résidentiel. Il le fait en synchronisant régulièrement avec les machines des utilisateurs, de telle sorte qu'une sauvegarde dans le réseau résidentiel est disponible. Chaque fois qu'un dispositif tombe en panne, ce **copie sur place** peut être utilisée pour récupérer une sauvegarde à la vitesse du réseau à domicile.

En outre, le contrôleur du swarm utilise la copie locale sur place pour créer des fragments de données à être transféré à d'autres passerelles. De cette façon, nous soulager les machines des utilisateurs de rester connecté à Internet afin de créer un back-up hors-site. Ce back-up hors-site est géré par le contrôleur du swarm, dont nous attendons pour être en ligne la plupart du temps. Outre le téléchargement de la sauvegarde initiale, le contrôleur du swarm est également en charge de la **maintenance** du back-up hors-site. Comme le stockage sur d'autres passerelles n'est pas fiable, nous devons considérer les comportements malveillants, qui peuvent corrompre notre back-up. Par conséquent,

un contrôleur du swarm régulièrement **vérifie l'intégrité** de du back-up hors-site. En conséquence, en synchronisant toutes les données précieuses des machines des utilisateurs au contrôleur du swarm, nous obtenons un back-up sur place rapide et confier la tâche plus vaste de la création d'un back-up hors-site pour le contrôleur du swarm. Une passerelle seuls ajouts fragments de sa propre back-up hors-site et prend donc le rôle d'un contrôleur du swarm exactement une fois.

Nœud de stockage

Un nœud de stockage $sn \in SW$ **stocke des fragments de données** liée à back-up d'un contrôleur du swarm étrangère. Pour cela, il propose un stockage clé-valeur qui accepte les données jusqu'à un certain quota est atteint. En outre, il fournit une méthode pour la remise en état de stockage, ainsi que les anciennes back-ups peuvent être remplacés par de nouveaux.

Nous considérons que le stockage offert par un nœud de stockage à ne pas être fiable en termes de confidentialité et d'intégrité. Par conséquent, avant d'envoyer des fragments de données à un nœud de stockage, un contrôleur du swarm crypte les données de sorte qu'un nœud de stockage ne peut pas voir les données d'origine d'un back-up. Pour un contrôleur du swarm, afin de récupérer un back-up, plusieurs nœuds de stockage doivent être contactés. En retour, une passerelle a le rôle d'un nœud de stockage à plusieurs reprises pour différents contrôleurs des swarms.

Swarm

Un swarm $SW = \{sn_1, sn_2, \ldots, sn_i\}$ est un ensemble de nœuds de stockage choisis au hasard. Chaque contrôleur du swarm sauvegarde toutes ses données sur un tel swarm individu. La taille de l'ensemble des swarms est flexible dans le temps, par conséquent, les nœuds de stockage peuvent être ajoutés et supprimés par le contrôleur du swarm. Depuis le contrôleur du swarm est la seule source de données stockées dans un swarm, sa présence a un impact direct sur la quantité de redondance disponible. Par conséquent, un swarm peut être dans l'un des états suivants:

- **Swarm intact:**
  Un contrôleur du swarm effectue généralement la maintenance. Comme les nœuds de stockage dans le swarm quittent le système, le contrôleur du swarm ajoute de nouveaux nœuds de stockage et ajouts des fragments de données supplémentaires pour eux. Un swarm est considérée comme intacte, même si un contrôleur du swarm peut être hors ligne pendant une courte période (par exemple plusieurs jours).

- **Swarm isolé:**
  Le contrôleur du swarm est hors ligne pendant une longue période (plusieurs semaines par exemple). Il n'effectue pas de maintenance, et donc, la redondance présente dans le swarm diminue au fil du temps. Nous concevons le niveau de redondance dans un swarm à être suffisamment élevée pour qu'un swarm peut être à l'état isolé pour une période prédéfinie jusqu'à plusieurs années. Durant cette période, le back-up peut être restauré à l'aide des données disponibles dans le swarm.

Après cette période, la redondance disponible dans le swarm descend en dessous d'un seuil de sorte que la perte de données peut se produire. C'est seulement dans ce cas que l'utilisateur ne peut plus télécharger son back-up de le swarm. Par conséquent, avant d'arriver à ce point, un tiers, par exemple un service au sein d'un centre de données, peut télécharger et reconstruire tous les fichiers chiffrés et les tenir prêt pour le téléchargement pour le propriétaire de back-up.

Tracker

Le tracker est un intermédiaire confidentiel et une instance centrale au sein d'un centre de données.

Pour rejoindre le réseau fédéré, une passerelle contacts le tracker. Le tracker fournit des informations sur les états des autres passerelles. En particulier, le tracker remplit les fonctions suivantes:

- **Back-Up de la liste du swarm**
  Quand une passerelle échoue, il perd non seulement la copie sur place, mais également les informations sur l'endroit où les instantanés précédents sont stockés dans le réseau fédéré. Pour cette raison, le tracker conserve une copie de la liste du swarm, qui, en cas d'échec, doit être demandé pour commencer la récupération.

- **Suivre les passerelles**
  Le tracker conserve la trace des passerelles dans le réseau fédéré. Par conséquent, les passerelles envoient de temps en temps des messages de pulsation au tracker. Le tracker met à jour un horodatage t_l de la passerelle particulier à l'heure actuelle chaque fois qu'un tel message de pulsation est reçu.

- **Observer l'équité dans le système**
  Le tracker reçoit les rapports des passerelles sur une éventuelle mauvaise conduite des autres passerelles. Il regroupe régulièrement ces informations et exclut certaines passerelles du réseau fédéré par la suite. Cette mauvaise conduite, par exemple la suppression illégitime de fragments comme

un nœud de stockage ou la création de trop de redondance comme un contrôleur du swarm, peut être détecté par le tracker en raison de sa connaissance globale sur le réseau fédéré.

- **Autorité de certification**
  Pour chaque passerelle dans le réseau fédéré le tracker conserve un certificat, contraignant l'identité d'une passerelle à une clé publique correspondante. De cette façon, les participants peuvent valider chaque identité et d'autres, en cas de mauvaise conduite, nous sommes en mesure de révoquer une autorisation de passerelles pour participer au réseau fédéré.

De toute évidence, une instance centrale augmente le risque de créer un goulot d'étranglement, empêchant éventuellement un système à l'échelle mondiale [HAY+05]. En fait, les fonctionnalités offertes par le tracker peut également être réalisé en utilisant une DHT entre les passerelles. Dans les systèmes P2P pur, cependant, il est difficile de créer un consensus sur l'état du système. Le problème des généraux byzantins par Lamport et al. [LSP82] est souvent utilisé pour illustrer ce problème. Il indique que, pour décider de l'état actuel d'un système de partis non fiables, au moins $3c + 1$ nombre total de participants sont nécessaires pour tolérer c participants défectueux ou de mauvaise conduite. En conséquence, pour parvenir à un consensus, beaucoup des messages sont nécessaire, et pourtant, le système peut être compromise, par exemple, en créant un grand nombre de pseudonymes identités connues, comme une attaque Sybil [Dou02].

En revanche, en utilisant le tracker comme une instance unique avec un savoir global, ces décisions sont prises par le tracker et les participants tout simplement s'y adapter. Cela peut conduire à charge plus élevée sur le tracker de sorte que son évolutivité est très important et l'utilisation excessive doit être évitée par la conception. En fait, le tracker peut être constituée de plusieurs machines de telle sorte qu'une panne matérielle unique n'entraîne pas de perte de données ou une interruption de service.

Notre architecture nécessite un utilisateur de faire confiance à l'instance centrale de fournir un soutien à la création et le maintien de ses sauvegardes. Pour la procédure de récupération du back-up la présence du tracker n'est pas nécessaire. Nous aussi explicitement ne comptons pas sur le tracker en termes de confidentialité des données. Fragments de données dans notre système exclusivement contiennent des données chiffrées que seules le propriétaire des données est en mesure de déchiffrer.

Représentation pour les back-ups hors site

Pour créer des back-ups hors site, nous utilisons des réplicas hébergés par le gateway qui, par conséquent, ne nécessitent pas l'accès aux appareils des util-

isateurs.

Nous introduision un niveau de contournement qui nous permet de référencer le contenu des fichiers à partir de leur contenu. Pour ce faire nous introduison un identifiant de fichier, Id_f qui est calculé de manière determininste selon la formule suivante: $Id_f = H(H(f))$, où H est une fonction de hashage appliqué sur le hash du contenu du fichier[1].

Cette procédure nous assure que l'identifiant du contenu d'un fichier ne sera pas modifié lors de futures sauvegardes, ceci même lorsque certaines pertes peuvent avoir lieu au niveau de la passerelle. D'autre part, si l'identifiant d'un contenu de fichier existe déjà dans le swarm, cela siginifie que le contenu lui même existe déjà et son transfert peut donc être ignoré.

Lors de chaque sauvegarde un **fichier d'indexe** est créé qui contient toutes les meta données nécessaires pour reconstruire le système de fichier dans son état initial. Ces méta données incluent des informations sur les noms de fichier, les dossiers utilisés, les propriétés et droits d'accès ainsi que les dates où l'on a accédé à ce fichier. Enfin cette métadonnée garde des référence sur le contenu des fichiers mentionés précédemment ainsi que les clés nécessaire à la description de ces contenus.

Après sa création le fichier d'indexe est stocké de la même manière qu'on le ferait avec un fichier ordinaire, au sein du swarm. Etant donné que le fichier d'indexe est le point de départ de toute remise en état à partir d'un fichier de sauvegarde, il est nécessaire de pouvoir le localiser suite à la survenue d'une erreur. Par conséquent, chaque nœud de stockage au sein du swarm garde une liste des identifiants de contenus de fichiers ainsi de leur dates de création.

La Figure A.4 présente la solution proposée à travers un exemple utilisant deux sauvegardes. Étant que nous utilisons des identifiant de fichiers, nous gardons ces références au sein du fichier d'indexes. Ce fichier contient également toutes les autres métadonnées de la sauvegarde.

Stratégie de placement de données

Dans cette section, nous expliquons comment nous distribuons données entre les passerelles du réseau fédéré et le tracker central.

Dans notre système, le tracker affecte de nouveaux nœuds de stockage à un swarm uniformément au hasard.

Cela correspond à une politique globale de placement de données, dans lequel pratiquement n'importe quel des N nœuds de stockage totale du réseau fédéré

[1] Le résultat d'une seule fonction de hashage est utilisé pour construire la clée d'encryption d'un fichier.

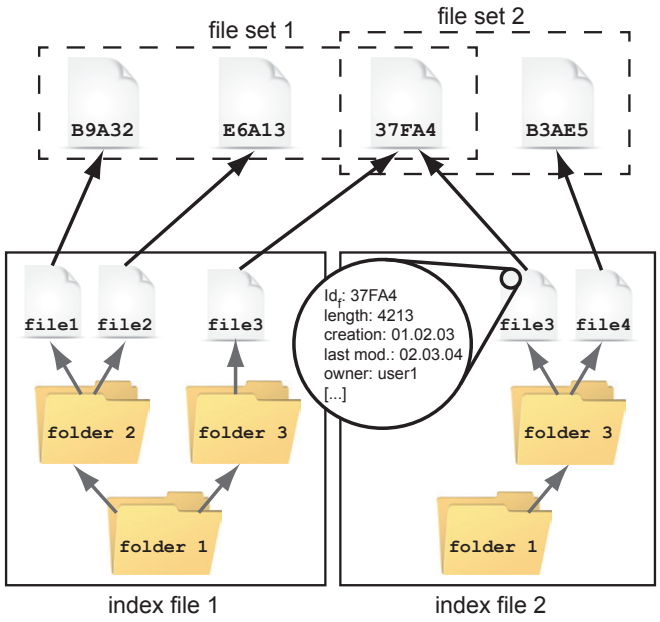


Figure A.4: Informations dans les fichiers d'indexe pour les back-ups hors site

peut être sélectionné pour les fragments de données d'un contrôleur du swarm. Cette politique de placement conduit à une répartition égale de la charge dans le réseau fédéré.

La Figure A.5 illustre notre politique de stocker tous les contenus de fichiers dans un swarm.

Dans les deux premières étapes, nous séparons chaque fichier f en k ($10 \leq k \leq 200$) **blocs de transmission** $T_{f,i}$ de taille égale avec l'**index** $i \in \{1, \ldots, k\}$.

En conséquence, chaque bloc de transmission d'un fichier de taille S_f a une taille de $\lceil S_f/k \rceil$.

Utilisant des codes d'effacement (comme Reed-Solomon [WB99], nous générons h des blocs de transmission supplémentaires $T_{f,i}, i \in \{k+1, \ldots, k+h\}$ (comme indiqué à l'étape 3). Comme une propriété de codes d'effacement, des k sur les $n = k + h$ différents blocs de transmission seront suffisantes pour reconstruire le contenu du fichier d'origine par la suite.

Pour le placement des données sur un nœud de stockage, nous regroupons les blocs de transmission de tous les fichiers par leur indice i à un **substream** SS_i:

$$SS_i = T_{1,i}, T_{2,i}, \ldots, T_{f,i}$$

La taille globale de ces sous-flux dépend de la quantité totale de données S_t

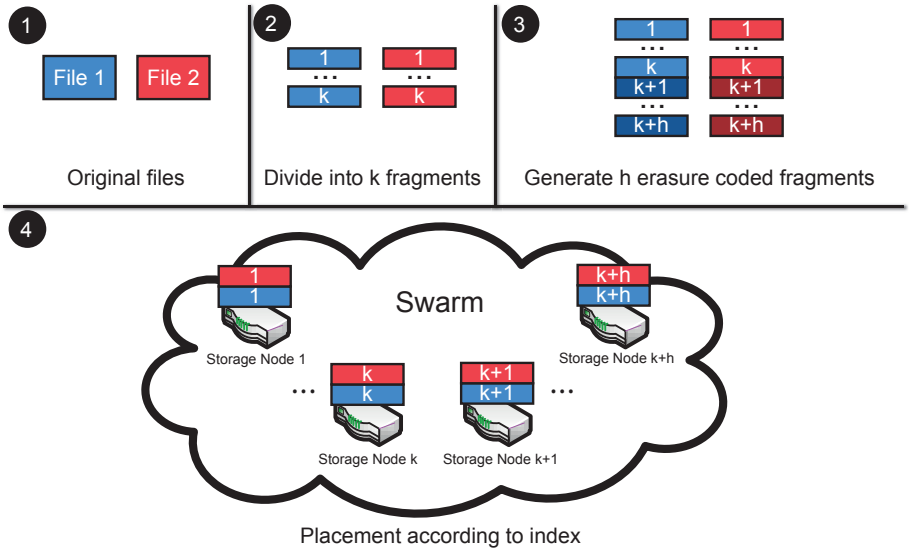


Figure A.5: Placer les substreams sur les nœuds de stockage

utilisé pour les fichiers uniques à travers tous les snapshots et est définie par S_t/k.

Nous mettons un substream distinct sur chaque nœud de stockage au sein d'un swarm (voir Figure A.5 étape 4). En outre, nous attribuons à chaque nœud de stockage l'indice utilisé pour le codage d'effacement de sorte que nous sommes en mesure de reconstruire les données plus tard et les données sont stockées de manière déterministe.

En conséquence de cette stratégie de placement nous transférons la propriété de codes d'effacement sur les nœuds de stockage: tout k nœuds de stockage d'un swarm sont suffisantes pour reconstruire un snapshot.

Réparation

Avec le temps, les nœuds de sauvegarde au sein d'un même swarm peuvent disparaitre en raison d'une panne. On peut classer ces pannes en deux catégories. Si un nœud de sauvegarde est indisponible pour les autres participants d'une manière temporaire uniquement, alors cette panne peut être désignée comme une **panne transitoire** : le nœud de sauvegarde pourra retourner éventuellement dans le système de manière à ce que les données sauvegardées sur ce dernier puissent permettre à nouveau une récupération du système. En cas de **panne permanente** cependant, le nœud de sauvegarde reste indisponible au système de manière permanente. Dans ce dernier cas, les donnéees sauvegardées sur le

nœud sont perdues définitivement et ont besoin d'être remplacées par le chargement d'une nouvelle redondance. Cependant, puisqu'il nous est impossible de différencier une panne transitoire d'une panne permanente vu de l'extérieur du système, un maintien du nombre de nœuds de sauvegarde au sein du swarm est réalisé quel que soit le type de panne encourue. Pour permettre cela, nous incluons de nouveaux nœuds de sauvegarde au sein du swarm et plaçons un nouveau flux de données auquel on attribue un indice incrémenté.

Au cas où un nœud de sauvegarde est de nouveau accessible par la suite, il peut alors être réintégré au swarm sans coût additionnel. A long terme cela revient à effectuer des réparations déclenchées uniquement lors de pannes permanentes [CDH+06].

Niveau de redondance requis

Dans notre système, nous comptons sur la possible copie sur place des données afin de générer de nouveau flux de données. Cela nous restreint à n'effectuer des maintenances que lorsque le contrôleur du swarm est en ligne. De plus nous adoptons la définition de durabilité des données introduite par Toka et al. [TCDM12]:

Definition: La durabilité des données d est la probabilité d'être capable d'accéder des données après une fenêtre temporelle t_{iso}, durant laquelle aucune opération de maintenance ne peut être effectuée.

Dans le cas d'une panne transitoire, le contrôleur du swarm détient toujours une copie des données et peut ainsi augmenter le niveau de redondance aussitôt qu'il peut se reconnecter à nouveau au système. En revanche, lors d'une panne permanente, aucune réparation n'est alors possible. Dans ce cas, puisque les données locales sont perdues, cela mène à une récupération du système par l'utilisateur, nous nous attendons à démarrer le téléchargement du back-up au bout d'une période que nous nommons **temps de remplacement** t_r. Au cours de cette période généralement longue, nous nous attendons à ce que l'utilisateur remarque la perte des données locales, installe un nouveau device et le ramène à un état opérationnel.

Ainsi le **temps de téléchargement** t_d est la période requise pour télécharger le back-up du swarm. Dans le cas de liens asynchrones, la vitesse de chargement des nœuds de sauvegarde est en général plus basse que la vitesse de téléchargement du propriétaire du back-up. Puisqu'il nous est possible d'effectuer de multiples transferts concurrents de plusieurs nœuds de sauvegarde en même temps, nous nous attendons à ce que la vitesse de téléchargement du propriétaire de la back-up détermine le temps requis pour l'opération. Pour cela, la

période de disponibilité des données t_d a besoin d'être suffisamment longue afin de premettre de finir le téléchargement des données comme discuté en détails dans [TCDM12].

Mais encore, nous avons besoin de considérer la **période de timeout** t_o utilisée pour différencier les pannes temporaires des pannes permanentes. Cette période implique que les données perdues du à une potentielle panne permanente non détectée sont manquantes dans le swarm. Pour ces raisons, la procédure de maintenance ne peut tenir compte que du niveau de redondance détecté à $t_c - t_o$, où t_c est le temps en cours.

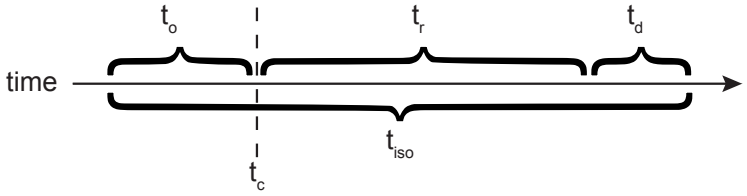


Figure A.6: Périodes Pertinentes pour la Procédure de Maintenance

La Figure A.6 offre un aperçu selon les périodes t_o, t_r, et t_d, dont la somme est égale à t_{iso}, qui correspond au temps total de survie de la back-up, une fois isolée du contrôleur de swarm.

En faisant l'hypothèse que la durée de vie des nœuds est régie par un processus de Poisson, on peut calciler la durabilité résultante d'un certain niveau de redondance et le temps t_{iso} comme décrit dans [TCDM12]:

$$d = \sum_{i=k}^{k+h} \binom{k+h}{i} (e^{-t_{iso}/\tau})^i (1 - e^{-t_{iso}/\tau})^{(k+h)-i} \tag{A.1}$$

La Figure A.7 et la Figure A.8 montrent le niveau de redondance $r = \frac{k+h}{k}$ requis pour une durabilité de $d = 0.999999$ et différentes valeurs de temps moyen de durée de vie τ des nœuds de sauvegarde et du nombre de fragments k en combien notre back-up est divisée.

Alors que dans la Figure A.7 nous ciblons une période t_{iso} d'une demi-années, nous voyons que la redondance nécessaire augmente pour un t_{iso} d'une année, comme nous l'avons montré dans la Figure A.8. Nous voyons de plus que la facteur de redondance dans les deux cas est bas, principalement quand $\tau \gg t_{iso}$. Des valeurs élevées pour le paramètre k réduisent également la redondance nécessaire au système, même si cet effet devient négligeable pour des valeurs au-delà de 100.

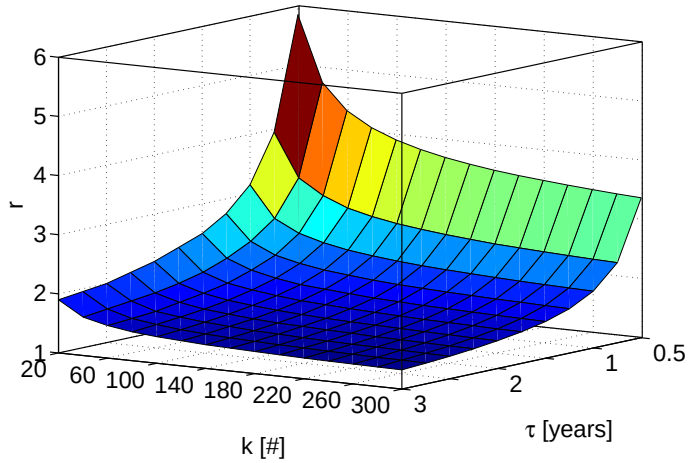


Figure A.7: Redondance nécessaire r en fonction de τ et k pour $t_{iso} = 1/2$ années

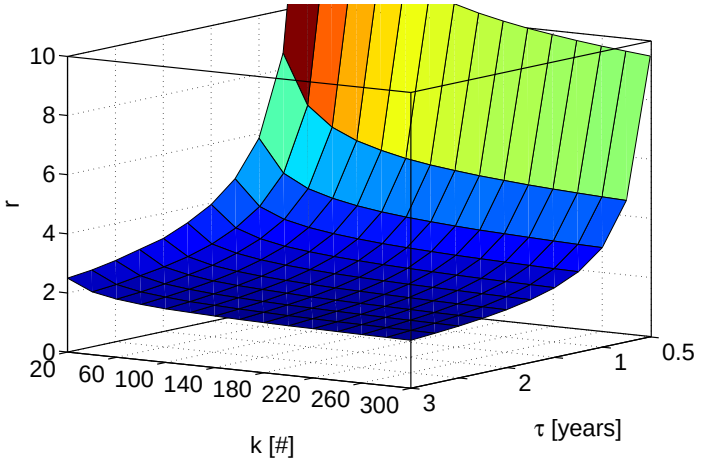


Figure A.8: Redondance nécessaire r en fonction de τ et k pour $t_{iso} = 1$ années

Différents moyens de sauvegarder des fichiers dans un swarm

Dans cette section nous introduisons trois techniques de sauvegarde de fichiers dans le but d'améliorer les capacités de nos systèmes en termes d'espace de stockage et de latence.

Petits fichiers (plus petits que 16 KiB)

Nous sauvegardons les *petits fichiers* dont la taille est plus petite que 16 KiB dans le fichier d'indexes. Cela revient à ne les lire qu'une seule fois afin de les sauver dans un fichier `tar` où ils sont sauvegardés proches de leurs métadonnées. Puisque nous cryptons le fichier d'indexes avant de le charger, nous pouvons nous affranchir de la procédure de cryptage pour les petits fichiers.

En utilisant cette procédure, nous empêchons un faible taux d'accès aux données causé par les recherches disques pour les petits fichiers lors de la création des nouveaux flux de données.

Cependant, un des désavantage de cette approche c'est qu'elle ne permet pas d'empêcher le duplicata de petits fichiers au cours de différents snapshots. Dans le système de sauvegarde Wuala [LaC13] tous les fichiers plus petits que 16 KiB occupent moins de 0.1% de la sauvegarde totale utilisée (voir également la Figure A.9). Il nous semble ainsi totalement justifié d'inclure les contenus des petits fichiers au sein de multiple fichiers d'indexes.

Comme les petits fichiers sont cachés via un nœud de sauvegarde, nous réduisons ainsi le coût de gestion lors de la récupération des sauvegardes au sein d'un nœud de sauvegarde. En conséquence on ne peut pas sauvegarder un fichier d'indexe comme un petit fichier. Aussi nous avons besoin de remplir un fichier d'indexe pour atteindre une taille d'au moins 16 KiB afin qu'il soit chargé comme un fichier de taille moyenne.

Fichiers de taille moyenne (16 KiB - 1 MiB)

On dénote par fichiers de *taille moyenne* les fichiers dont la taille est comprise entre 16 KiB et 1 MiB. Après le cryptage les fichiers de taille moyenne sont encodés en une seule passe en chargeant le contenu complet du fichier dans un buffer de données du module de codage effaçage. Les résultats des blocs de transmission de taille $\lceil S_f/k \rceil$, lesquels impliquent un maximum plafonné de $k-1$ d'octets par fichier.

Alors que cette procédure implique uniquement des sauvegardes plafonnées négligeables, elle requiert davantage de mémoire avec des fichiers de tailles plus

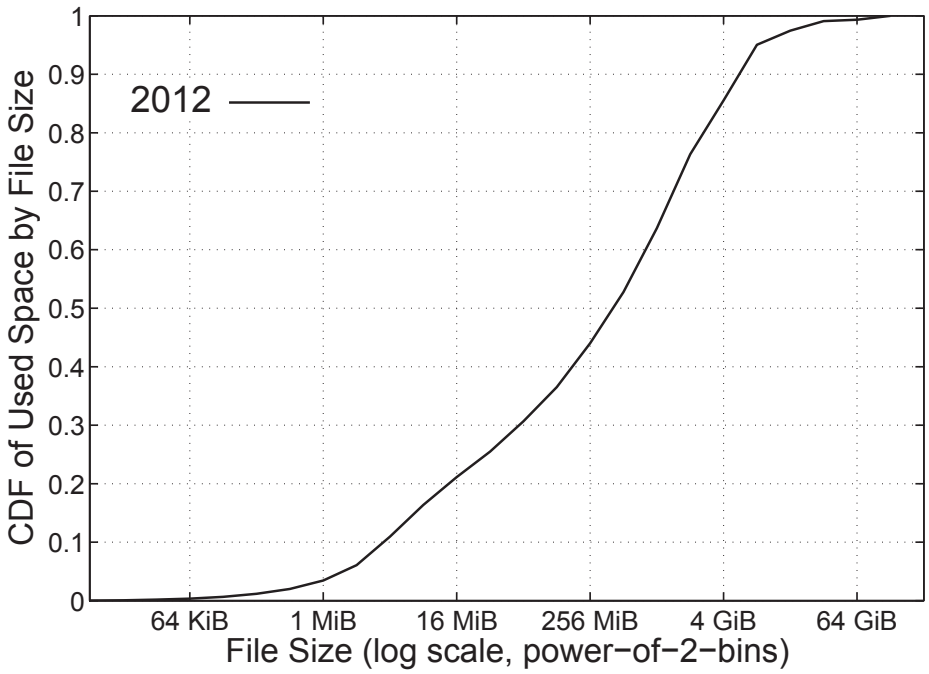


Figure A.9: CDF de l'espace de stockage utilisé par taille de fichier; Système cloud storage Wuala

importantes. Puisque nous effectuons les cryptage et le codage d'effacement en mémoire, nous avons besoin de buffers avec un espace total de $2 \cdot S_f + n \cdot S_f/k$.

De plus, afin de récupérer un fichier moyen, un receveur ne peut utiliser que des blocs entièrement transmis. Ceci peut mener à des situations dans lesquelles de grande quantité de données reçues d'un nœud de sauvegarde ne peuvent être utilisées pour la reconstruction de fichiers car seuls quelques octets sont manquants.

Dans le but de limiter l'impact en termes de mémoire et minimiser les effets liés aux pertes de connectivité, nous utilisons une méthode différente pour sauvegarder les fichiers plus gros que 1 MiB, lesquelle est introduite dans le paragraphe suivant.

Gros Fichiers (Plus gros que 1 MiB)

Puisque le codage d'effacement est effectué en mémoire, nous ne pouvons pas l'effectuer sur des fichiers de tailles arbitraires. Aussi pour les fichiers dont la taille excède 1 MiB nous utilisons un **schéma d'entrelacement** [PS07], lequel nous permet de conserver une consommation mémoire indépendante de la taille

du fichier comme montré dans Wuala [TMEBPM12] et expliqué dans ce qui suit.

On partitionne un fichier de taille S_f en $j = \lceil S_f/b \rceil$ **segments** de taille $b = 100$ KiB. Nous choisissonns 100 KiB parce que cette valeur nous a donné de bonne performances et n'entraine qu'un faible impacte mémoire.

Chacun de ces segments sont alors encodés indépendamment en utilisant un code d'effacage. Dans ce but, un segment est divisé en $k = 100$ fragments originaux de 1 KiB lesquels sont encodés afin d'obtenir $n = k + h$ **fragments de codage** de 1 KiB chacun. Ainsi, le segment $s, s \in \{1, \ldots, j\}$ donne n fragments codés $C_{s,1}, ..., C_{s,n}$. Pour les fichiers f et les flux SS_i on groupe tous les fragments codés $C_{1,i}, ..., C_{s,i}$ en un bloc de transmission $T_{f,i}$, lequel peut être envoyé sur le réseau. Sur la Figure A.10 nous illustrons cette procédure en détails.

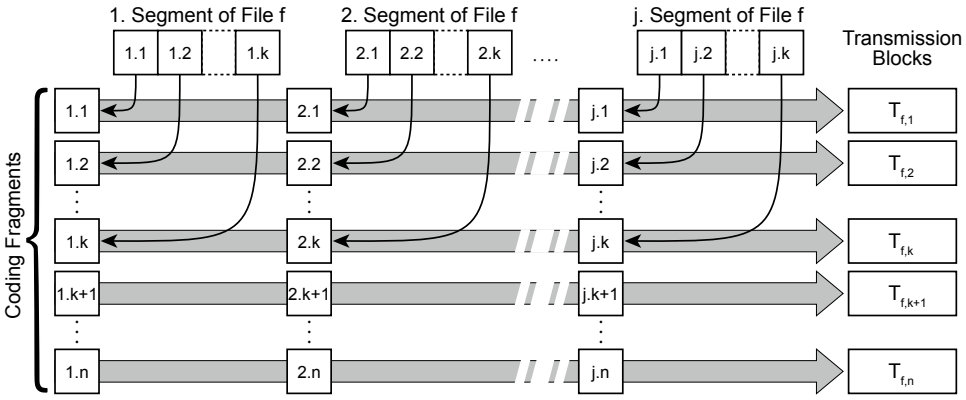


Figure A.10: Schéma d'entrelacement opérant sur des segments

Nous obtenons des blocs de transmission avec une taille de $\lceil S_f/b \rceil \cdot b/k$, lesquels impliquent une taille maximum de b par fichier. En effet, si la consommation mémoire sur la gateway n'est par un problème, on peut envisager d'augmenter le seuil maximum des fichiers de taille moyenne afin de minimiser l'impact sur les petits fichiers manipulés selon le schéma d'entrelacement.

Lorsqu'une récupération est nécessaire, contrairement au schéma utilisé pour les fichiers de taille moyenne, le schéma d'entrelacement nous permet de tirer le bénéfice des données des blocs de transmission qui ne sont pas téléchargés entièrement. Dans ce cas un nœud de sauvegarde est mis hors ligne durant la récupération des fichiers, nous sommes alors capables de choisir un autre nœud et de toujours continuer à utiliser les fragments codés du fichier en cours.

Panne totale du tracker

Dans notre système, nous avons le risque de perdre le tracker, par exemple, lorsque l' opérateur du tracker arrête tous les services. Dans ce qui suit, nous analysons le scénario sans le service du tracker de plus près et nous montrons comment le réseau fédéré peut retourner dans un état opérationnel.

Impact sur le réseau fédéré

Une panne totale du tracker a un fort impact sur la fonctionnalité de notre service de sauvegarde distribué qui est désormais réduite à des passerelles qui subsistent dans le système.

Ces passerelles restent confrontés aux conséquences et restrictions suivantes:

- Le contrôleur du swarm ne peuvent pas demander des nouveaux nœuds du tracker. De réaliser la réparation n'est pas supportée de sorte que la redondance fournie par un swarm diminue au fil du temps.
- La CA ne délivre pas de nouveaux certificats de clés publiques afin que de nouvelles passerelles ne peuvent plus adhérer au système.
- L'équité dans le système n'est pas observée globalement par le tracker.
- La copie de la liste du swarm situé sur le tracker est inaccessible. En cas de perte de données sur une passerelle nous avons besoin d'une autre façon de récupérer la liste du swarm.
- En cas de perte de données locales un utilisateur n'a *pas encore un point d'entrée* au réseau fédéré. Les nœuds de stockage restant dans le réseau fédéré sont donc la seule instance détenant des informations sur l'emplacement des sauvegardes stockées.

Dans ce qui suit, nous montrons une approche comment le système peut continuer à fonctionner sans le tracker.

Mode de repli et réunion

Si une passerelle ne peut pas atteindre le tracker pour une période prédéfinie (par exemple un couple de jours), il passe en *mode de repli*. Dans ce mode, le comportement de passerelles change comme suit:

- **Alternance du nom de domaine**
  Au fil du temps la passerelle modifie le nom de domaine sous lequel il tente

de communiquer avec le tracker. Pour cela, nous construisons une candidat un ensemble de noms de domaine possibles de substitution, à l'aide d'une fonction déterministe. Nous semons cette fonction par la date du jour sur la passerelle de sorte que chaque jour toutes les passerelles tentent d'atteindre les mêmes noms de domaine dans le DNS.

Cette procédure est connue sous le nom de *Domain Generation Algorithm* (DGA) et est devenu populaire en raison de leur utilisation dans les réseaux de machines zombies récents tels que Conficker [PSY09].

Semblable à notre scénario en utilisant un tracker, les opérateurs de réseaux de zombies tentent de résister contre les pannes de leur serveur central commandement et contrôle. DGA nous permet donc de créer un grand nombre de points de rendez-vous que nous pouvons utiliser à l'avenir pour établir un nouveau tracker et de recueillir à nouveau les passerelles précédentes.

- **Des messages de pulsation aux contrôleurs du swarms**
  Puisque nous considérons la liste du swarm situé sur le tracker comme perdu, en cas de perte de données locale sur une passerelle, nous avons besoin de récupérer la liste du swarm. Pour ce faire, nous changeons l'origine pour établir le contact: Dans cette situation, le propriétaire de back-up ne peut pas communiquer avec les nœuds de stockage. C'est pourquoi les nœuds de stockage essaient de contacter le propriétaire de back-up. Ceci est possible car le stockage nœuds connaissent le propriétaire de back-up correspondant à chaque substream. Conséquent, un nœud de stockage envoie régulièrement des messages de pulsation à un propriétaire de back-up. Dès après la perte de données le propriétaire de back-up se reconnecte en utilisant son nom d'hôte précédent, il reçoit ces messages de pulsation et, par conséquent, peut déterminer les sources potentielles de commencer back-up récupération.

Quand les passerelles détectent que le tracker réapparaît (éventuellement sur l'un des points de rendez-vous), ils vérifient son authenticité au moyen de sa clé publique.

Si l'authenticité est confirmée, les passerelles revenir au mode de fonctionnement normal et, ainsi, continuer à exécuter la réparation. Dans le cas où aucun tracker réapparaît la période t'_{iso} est le temps qu'un utilisateur dispose afin de passer à un autre système de back-up.

Conclusion

Ce travail présente un système de sauvegarde distribué qui permet à un utilisateur de stocker un back-up en manière de snapshots. Nous profitons de l'intérêt commun des participants dans le stockage des sauvegardes en misant principalement sur les ressources fournies par les utilisateurs eux-mêmes.

La principale contribution de ce travail est une preuve de concept qui confirme la faisabilité d'un tel service. Une autre contribution est le concept de fichiers d'index, qui sont une nouvelle façon pour un système distribué pour représenter l'alternance des systèmes de fichiers au fil du temps. La division du réseau fédéré en swarms facilite le suivi des passerelles et réduit les métadonnées sur le tracker nécessaire pour la localisation des données à un très faible niveau. En fait, la charge sur le tracker est indépendante de la quantité de données stockées dans le système, ce qui est une propriété souhaitable, en particulier dans le contexte de l'augmentation des quantités de données dans le futur.

Nous nous occupons de fichiers de différentes tailles d'une manière différente afin d'accroître l'efficacité globale de notre système. Avec notre système, nous remettons aussi en question l'approche établie d'exiger des données dans le système pour durer éternellement. Au lieu de cela, nous nous concentrons sur permettant à un utilisateur de récupérer toutes les données en cas de perte de données locale dans un délai prédéfini. Durant cette période, nous pouvons également tolérer le tracker de quitter le système.

Le résultat est une architecture qui supporte la creation de back-up automatiquement et en manière de snapshots: d'abord de dispositifs d'utilisateur à la passerelle et enfin au réseau fédéré. La perte de données due à des événements tels que les incendies et les inondations peuvent donc être évitée. Nous sommes fermement convaincus que notre système peut être déployé dans le monde réel et contribuer à atteindre un back-up à prix abordable qui conserve la confidentialité des données.

Appendix B

Additional Implementation Details

B.1 Maintenance Module

Phase 1: Generate Unassigned Substreams

Precondition: The maintenance algorithm requires $a > 0$ new substreams in the swarm.

Operation:

Procedure Generate Unassigned Substreams

```
1  referenced := {};
2  for current snapshot := oldest snapshot to newest snapshot do
3  |  foreach file ∈ current snapshot do
4  |  |  if file ∉ referenced then
5  |  |  |  referenced := referenced ∪ {file};
6  |  |  |  for index := last index to (last index + a) do
7  |  |  |  |  generate transmission block T_{file,index};
8  |  |  |  |  store T_{file,index} in /temporary/inprocess/$index/;
9  |  |  |  end
10 |  |  end
11 |  end
12 end
13 for index := last index to (last index + a)) do
14 |  atomically move /temporary/inprocess/$index/ to
   |  /temporary/unassigned/$index/;
15 end
```

Postcondition: There are a unassigned substreams available in the folder `/temporary/unassigned/`. The creation procedure assures that unassigned substreams are *complete*, meaning that they contain a transmission block for each unique file content and that transmission blocks are fully written to disk.

Phase 2: Assign Substreams to Storage Nodes

Precondition: Substream(s) present in `/temporary/unassigned/`.

Operation:

Procedure Assign Substreams

```
1 foreach unassigned substream do
2 |   send RESTful GET request for a new storage node to tracker;
3 |   atomically rename /temporary/unassigned/$index/ to
  |   /temporary/assigned/$node_id/$index/;
4 end
```

Postcondition: Substream(s) assigned to storage node(s). Therefore, folder `/temporary/unassigned/` is empty.

Phase 3: Include Storage Nodes in Swarm List

Precondition: Substream(s) assigned to storage node(s) in folder `/temporary/assigned/`.

Operation:

Procedure Include Storage Nodes in Swarm List

```
1  foreach assigned substream do
2  |   store swarm set on tracker via RESTful PUT request with node id;
3  |   if response is HTTP Status 200-OK then
4  |   |   include storage node in local swarm list;
5  |   |   atomically move /temporary/assigned/$node_id/$index/ to
   |   |   /upload/$node_id/;
6  |   end
7  |   else if response is HTTP Status 409-Conflict then
8  |   |   undo possible insertion of storage node into local swarm list;
9  |   |   atomically rename /temporary/assigned/$node_id/$index/
   |   |   to /temporary/unassigned/$index/;
10 |   end
11 end
```

Postcondition: No substream in `/temporary/assigned/`.

B.2 Snapshot Creator Module

Phase 1: Parse New Snapshot

Precondition: New snapshot in folder `/on-site copy/` but no corresponding index file in `/temporary/new_snapshot/generate_transmission_blocks/`, `/temporary/new_snapshot/upload_index_file`, or `/index files/`.

Operation:

Procedure Parse New Snapshot

```
1  references := load previous reference set;
2  create index file
   /temporary/new_snapshot/generate_index_file/$snapshot_id;
3  foreach folder in new snapshot do
4      create folder in index file;
5      foreach file in folder do
6          calculate file hash to generate $file_id;
7          insert file metadata into index file;
8          if ∃(a, b) ∈ references | a = $file_id then
9              references := references ∪ ($file_id, $snapshot_id);
10         end
11         else
12             foreach $node_id ∈ swarm do
13                 generate transmission block in
                   /temporary/inprocess/$node_id/;
14                 atomically move completed transmission block to
                   /upload/$node_id/;
15                 references := references ∪ ($file_id, $snapshot_id);
16             end
17         end
18     end
19 end
20 compress index file;
21 atomically move completed index file to
   /temporary/new_snapshot/generate_transmission_blocks/;
```

Postcondition: Complete index file in `/temporary/new_snapshot/generate_transmission_blocks/`.

Phase 2: Generate Transmission Blocks for Index File

Precondition: Existing index file in `/temporary/new_snapshot/upload_index_file/`.

Operation:

Procedure Generate Transmission Blocks for Index File

```
1 foreach $node_id ∈ swarm do
2 |   generate transmission block for index file in
  |   /temporary/new_snapshot/upload_index_file/$node_id/;
3 |   atomically move completed transmission block to
  |   /upload/$node_id/;
4 end
5 atomically move index file to
  /temporary/new_snapshot/update_metadata/;
```

Postcondition: Index file in `/temporary/new_snapshot/update_metadata/`.

Phase 3: Update Metadata on Storage Nodes

Precondition: Existing index file in `/temporary/new_snapshot/update_metadata/`.

Operation:

Procedure Update Storage Nodes

```
1 foreach $node_id ∈ swarm do
2 |   store metadata for index file in
  |   /temporary/new_snapshot/upload_metadata/$node_id/;
3 |   atomically move metadata to /upload/$node_id/;
4 end
```

Postcondition: No index file in `/temporary/new_snapshot/update_metadata/`.

Appendix C

Glossary

C.1 Acronyms and Abbreviations

P2P Peer-to-Peer, decentralized network in which participants not only consume but also offer resources to the network

DHT Distributed Hash Table, decentralized and fault tolerant hash table storing key value pairs.

LRU Least Recently Used, replacement policy to discard the least recently used items at first in order to free storage space

CA Certification Authority, architecture used to distribute asymmetric keys.

PKI Public-Key Infrastructure, an infrastructure used for key and certificate management.

VFS Virtual File System, abstraction layer to uniformly access data from different underlaying file systems.

NAS Network-Attached Storage, a service used to store and share files in a network.

NFS Network File System, a protocoll used to access a file system over a network.

SMB Server Message Block, a network protocol used for file, printer sharing, and serial ports.

FTP File Transmission Protocol, a protocol used for file transmission over a network.

BLOB Binary Large Object, a large binary block of data stored in a database.

HTTP Hypertext Transfer Protocol, a protocol used in the World Wide Web for content transmission.

URI Uniform Resource Identifier, an identifier for web resources.

TCP Transmission Control Protocol, a protocol for ordered and reliable transmission.

UDP User Datagram Protocol, a leightweight protocol used for transmission of single datagrams without guaranteed delivery.

TLS Transport Layer Security, a protocol used for encrypted data transmission.

OSI Open Systems Interconnection, standardized reference model for interoperability in network communication.

RPC Remote Procedure Call, an technique for inter-process communication over a network.

SOAP Simple Object Access Protocol, a protocol for the exchange of structured data over a network.

IP Internet Protocol, a protocol used for data transmission over a network.

CBC Cipher-Block Chaining, an encryption technique that uses the previous ciphertext block as input for the encryption of the succeeding one.

AES Advanced Encryption Standard, a symmetric-key algorithm for data encryption.

SHA Secure Hash Algorithm, a set of cryptographic hash functions.

JNI Java Native Interface, an interface allowing java applications to invoke code in libraries that are written in other programming languages.

MDS Maximum Distance Separable, a linear code that allows maximum error detection and correction for a given amount of redundancy.

HDFS Hadoop Distributed File System, a distributed file system designed for high data throughput and fault tolerance.

GFS Google File System, a distributed file system designed by Google Inc. for high data throughput and fault tolerance.

RDBMS Relational Database Management System, a database operating on tables and records.

PBKDF Password-Based Key Derivation Function, a function that allows to derive a key from a password.

DNS Domain Name System, a network system used to resolve names into IP addresses.

ISP Internet Service Provider, an organization that provides Internet services.

TTL Time To Live, a technique to limit the lifetime of data in a network.

DGA Domain Generation Algorithm, an algorithm originally used in botnets to generate alternative domain names in order to increase resistance to attacks.

WORM Write-Once-Read-Many, a device that does not support modifications on written data.

ADSL Asymmetric Digital Subscriber Line, a technology for data transmission over copper lines.

RAID Redundant Array of Independent Disks, a composition of multiple hard drives in order to improve reliability or performance.

List of Figures

List of Tables

Bibliography

[ABC+02] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer, *Farsite: Federated, available, and reliable storage for an incompletely trusted environment*, Proceedings of the 5th Symposium on Operating Systems Design and Implementation, OSDI '02, ACM, 2002, pp. 1–14. (Cited on page 12.)

[ABDL07] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch, *A five-year study of file-system metadata*, Trans. Storage **3** (2007), no. 3. (Cited on pages 77, 79, and 162.)

[All10] Subbu Allamaraju, *Restful web services cookbook*, O'Reilly Media, 2010. (Cited on pages 51, 56, and 59.)

[Ama14a] Amazon Web Services, *Amazon s3*, online, http://aws.amazon.com/s3/, 2014. (Cited on pages 2, 83, 111, and 130.)

[Ama14b] ———, *Amazon s3 pricing*, online, https://aws.amazon.com/s3/pricing/, 2014. (Cited on pages 3 and 131.)

[Apa14] Apache Software Foundation, *Cassandra architecture*, online, https://wiki.apache.org/cassandra/ArchitectureOverview, 2014. (Cited on page 85.)

[App14] Apple Inc., *Time machine*, online, http://www.apple.com/support/timemachine/, 2014. (Cited on page 14.)

[Bar01] Moshe Bar, *Linux File Systems*, McGraw-Hill, 2001. (Cited on page 62.)

[BBST02] Christopher Batten, Kenneth Barr, Arvind Saraf, and Stanley Trepetin, *pStore: A secure peer-to-peer backup system*, Tech. report, Massachusetts Institute of Technology Laboratory for Computer Science, October 2002. (Cited on page 17.)

[Bil95] Patrick Billingsley, *Probability and measure*, 3 ed., Wiley-Interscience, 1995. (Cited on page 108.)

[BJR94] George E. P. Box, Gwilym M. Jenkins, and Gregory C. Reinsel, *Time series analysis: forecasting and control (third ed)*, Prentice-Hall, 1994. (Cited on page 99.)

[BLV05] Alberto Blanc, Yi-Kai Liu, and Amin Vahdat, *Designing incentives for peer-to-peer routing*, Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2005), vol. 1, 2005, pp. 374–385 vol. 1. (Cited on page 91.)

[BPS05] Jean-Michel Busca, Fabio Picconi, and Pierre Sens, *Pastis: A highly-scalable multi-user peer-to-peer file system*, in Euro-Par 2005, 2005. (Cited on page 12.)

[BR03] Charles Blake and Rodrigo Rodrigues, *High availability, scalable storage, dynamic peer networks: Pick two*, Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9 (Berkeley, CA, USA), HOTOS'03, USENIX Association, 2003, pp. 1–1. (Cited on page 109.)

[BSSP10] Cullen Bash, Rocky Shih, Amip. Shah, and Chandrakant Patel, *Data center damage boundaries*, IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm), June 2010, pp. 1–9. (Cited on page 96.)

[BTC+04] Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey M. Voelker, *Total recall: system support for automated availability management*, USENIX NSDI 2004, 2004. (Cited on pages 12 and 17.)

[CDH+06] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, Frans Kaashoek, John Kubiatowicz, and Robert Morris, *Efficient replica maintenance for distributed storage systems*, NSDI 2006, 2006. (Cited on pages 13, 44, and 141.)

[CLG+94] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson, *Raid: High-performance, reliable secondary storage*, ACM Comput. Surv. **26** (1994), no. 2, 145–185. (Cited on page 12.)

[CMH+02] Ian Clarke, Scott Miller, Theodore Hong, Oskar Sandberg, and Brandon Wiley, *Protecting free expression online with freenet*, Internet Computing, IEEE **6** (2002), no. 1, 40–49. (Cited on page 71.)

[CMN02] Landon P. Cox, Christopher D. Murray, and Brian D. Noble, *Pastiche: Making backup cheap and easy*, Proceedings of OSDI, ACM, 2002. (Cited on page 18.)

[CN03] Landon P. Cox and Brian D. Noble, *Samsara: Honor among thieves in peer-to-peer storage*, Proceedings of the Symposium on Operating Systems Principles, ACM, 2003. (Cited on page 18.)

[Coh03] Bram Cohen, *Incentives build robustness in bittorrent*, 2003. (Cited on page 90.)

[Cor13] IBM Corp., *Writing reentrant and threadsafe code*, online, http://publib.boulder.ibm.com/infocenter/aix/v6r1/index.jsp?topic=/com.ibm.aix.genprogc/doc/genprogc/writing_reentrant_thread_safe_code.htm, 2013. (Cited on page 56.)

[CT96] Tushar Deepak Chandra and Sam Toueg, *Unreliable failure detectors for reliable distributed systems*, J. ACM (1996), 225–267. (Cited on page 40.)

[CWO+11] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, and et al., *Windows azure storage: A highly available cloud storage service with strong consistency*, Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11, ACM, 2011, pp. 143–157. (Cited on page 12.)

[DBEN07] Alessandro Duminuco, Ernst Biersack, and Taoufik En-Najjary, *Proactive replication in distributed storage systems using machine availability estimation*, Proceedings of the 2007 ACM CoNEXT Conference, CoNEXT '07, ACM, 2007, pp. 27:1–27:12. (Cited on pages 13, 41, and 96.)

[DG04] Jeffrey Dean and Sanjay Ghemawat, *Mapreduce: Simplified data processing on large clusters*, Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation, OSDI'04, USENIX Association, 2004, pp. 10–10. (Cited on page 12.)

[DGLR+11] Serge Defrance, Rémy Gendrot, Jean Le Roux, Gilles Straub, and Thierry Tapie, *Home networking as a distributed file system view*, Proceedings of the SIGCOMM Workshop on Home Networks (HomeNets'11), ACM, 2011, pp. 67–72. (Cited on pages 62 and 162.)

[DGW+07] Alexandros Dimakis, Brighten Godfrey, Yunnan Wu, Martin Wainwright, and Kannan Ramchandran, *Network coding for distributed storage systems*, IEEE INFOCOM 2007, 2007. (Cited on page 13.)

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels, *Dynamo: Amazon's highly available key-value store*, Proceedings of Twenty-first ACM SIGOPS, ACM, 2007, pp. 205–220. (Cited on page 84.)

[DKK+01] Frank Dabek, Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica, *Wide-area cooperative storage with cfs*, In SOSP, 2001. (Cited on pages 12 and 13.)

[DKM+11] Serge Defrance, Anne-Marie Kermarrec, Erwan Le Merrer, Nicolas Le Scouarnec, Gilles Straub, and Alexandre van Kempen, *Efficient peer-to-peer backup services through buffering at the edge.*, Peer-to-Peer Computing, IEEE, 2011. (Cited on page 96.)

[DMR10a] Matteo Dell'Amico, Pietro Michiardi, and Yves Roudier, *Back to the future: On predicting user uptime*, CoRR **abs/1010.0626** (2010). (Cited on page 96.)

[DMR10b] Matteo Dell'Amico, Pietro Michiardi, and Yves Roudier, *Password strength: an empirical analysis*, IEEE INFOCOM 2010, 2010. (Cited on page 49.)

[DMTC14] Matteo Dell'Amico, Pietro Michiardi, Laszlo Toka, and Pasquale Cataldi, *Adaptive redundancy management for durable p2p backup*, online, http://arxiv.org/abs/1201.2360v2, 2014. (Cited on page 72.)

[Dou02] John R. Douceur, *The sybil attack*, Revised Papers from the First International Workshop on Peer-to-Peer Systems, IPTPS '01, Springer-Verlag, 2002, pp. 251–260. (Cited on pages 26 and 137.)

[DQ04] Yves Deswarte and Jean-Jacques Quisquater, *Remote Integrity Checking*, Sixth Working Conference on Integrity and Internal Control in Information Systems (IICIS), Kluwer Academic Publishers, 2004. (Cited on page 53.)

[DR01] Peter Druschel and Antony Rowstron, *Past: A large-scale, persistent peer-to-peer storage utility*, In HotOS VIII, 2001, pp. 75–80. (Cited on pages 12 and 13.)

[Dro11] Dropbox Inc., *Authentication bug in dropbox*, online, http://blog.dropbox.com/?p=821, 2011. (Cited on page 3.)

[Dro14a] _____, *Dropbox*, online, http://www.dropbox.com/, 2014. (Cited on pages 2, 111, and 130.)

[Dro14b] _____, *How secure is dropbox?*, online, https://www.dropbox.com/help/27/, 2014. (Cited on pages 3 and 131.)

[EIG+14] Dick Epema, Alexandru Iosup, Matthieu Gallet, Emmanuel Jeannot, Derrick Kondo, Bahman Javadi, and Artur Andrzejak, *Failure trace archive*, online, http://fta.scem.uws.edu.au/, 2014. (Cited on page 96.)

[EN03] Ramez Elmasri and Shamkant Navathe, *Fundamentals of Database Systems*, Addison Wesley, 2003. (Cited on page 66.)

[FGL+98] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman, *Eventually-serializable data services*, 1998. (Cited on page 85.)

[FIG14] FIGARO Consortium, *Figaro project website*, online, http://www.ict-figaro.eu/, 2014. (Cited on pages 4 and 131.)

[FLSR10] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues, *A study of the internal and external effects of concurrency bugs*, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2010, pp. 221–230. (Cited on page 65.)

[Fre13] Free SAS, *Freebox revolution*, online, http://www.free.fr/adsl/freebox-revolution.html, 2013. (Cited on page 96.)

[FSF09] Xinyang Feng, Jianjing Shen, and Ying Fan, *Rest: An alternative to rpc for web services architecture*, Future Information Networks, 2009. ICFIN 2009. First International Conference on, 2009, pp. 7–10. (Cited on page 56.)

[FSK10] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno, *Cryptography engineering - design principles and practical applications*, Wiley, 2010. (Cited on pages 49, 50, 57, and 70.)

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, *The google file system*, Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (New York, NY, USA), SOSP '03, ACM, 2003, pp. 29–43. (Cited on pages 12 and 83.)

[GL02] Seth Gilbert and Nancy Lynch, *Brewer's conjecture and the feasibility of consistent available partition-tolerant web services*, In ACM SIGACT News, 2002. (Cited on pages 84 and 85.)

[GMP09] Frederic Giroire, Julian Monteiro, and Stephane Perennes, *P2P storage systems: How much locality can they tolerate?*, IEEE LCN 2009, October 2009. (Cited on pages 30 and 78.)

[GN96] Priscilla E. Greenwood and Michael S. Nikulin, *A Guide to Chi-Squared Testing*, 1 ed., Wiley, 1996. (Cited on page 99.)

[HAY+05] Ragib Hasan, Zahid Anwar, William Yurcik, Larry Brumbaugh, and Roy Campbell, *A survey of peer-to-peer storage techniques for distributed file systems*, ITCC 2005., 2005. (Cited on pages 13, 26, and 137.)

[HMD05] Andreas Haeberlen, Alan Mislove, and Peter Druschel, *Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures*, Networked Systems Design and Implementation, 2005, pp. 143–158. (Cited on pages 16, 38, 98, and 109.)

[HSX+12] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin, *Erasure coding in windows azure storage*, Proceedings of the Conference on Annual Technical Conference, USENIX Association, 2012, pp. 2–2. (Cited on page 12.)

[IBe99] Alaoui Ismaili, Pierre Bernard, and et al., *Estimating the probability of failure of equipment as a result of direct lightning strikes on transmission lines*, Power Delivery, IEEE Transactions on **14** (1999), no. 4, 1394–1400. (Cited on page 103.)

[IBM06] IBM Corp., *Understanding and exploiting snapshot technology for data protection.*, online, http://www.ibm.com/developerworks/tivoli/library/t-snaptsm1/index.html, 2006. (Cited on page 5.)

[IET13] IETF, *Rfc 5246*, online, http://tools.ietf.org/html/rfc5246, 2013. (Cited on page 57.)

[IET14a] ———, *Rfc 1035*, online, https://tools.ietf.org/html/rfc1035, 2014. (Cited on page 81.)

[IET14b] ———, *Rfc 1191*, online, http://www.ietf.org/rfc/rfc1191.txt, 2014. (Cited on page 58.)

[IET14c] ———, *Rfc 2182*, online, https://tools.ietf.org/html/rfc2182, 2014. (Cited on page 82.)

[Int12] International Data Corporation (IDC), *Digital universe study*, online, http://idcdocserv.com/1414, 2012. (Cited on pages 1, 2, and 161.)

[JHM11] Richard Jones, Antony Hosking, and Eliot Moss, *The garbage collection handbook: The art of automatic memory management*, 1st ed., Chapman & Hall/CRC, 2011. (Cited on page 89.)

[Ke00] John Kubiatowicz and et al., *Oceanstore: an architecture for global-scale persistent storage*, SIGPLAN Not. (2000), 190–201. (Cited on pages 18, 38, and 109.)

[KG94] Rom-Shen Kao and Vickie Gibbs, *A fast reed-solomon and cyclic redundancy check encoding algorithm for optical disk error control*, ASIC Conference and Exhibit, 1994. Proceedings., Seventh Annual IEEE International, Sep 1994, pp. 250–253. (Cited on page 11.)

[KJIE10] Derrick Kondo, Bahman Javadi, Alexander Iosup, and Dick Epema, *The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems*, IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid), May 2010, pp. 398–407. (Cited on page 96.)

[KR01] Balachander Krishnamurthy and Jennifer Rexford, *Web protocols and practice*, Addison Wesley, 2001. (Cited on pages 56 and 57.)

[Kro14] Kroll Ontrack, *Kroll ontrack data recovery service*, online, http://www.ontrack.fr, 2014. (Cited on pages 1 and 129.)

[KV10] Kyungbaek Kim and Nalini Venkatasubramanian, *Assessing the impact of geographically correlated failures on overlay-based data dissemination*, Global Telecommunications Conference (IEEE GLOBECOM 2010), Dec 2010, pp. 1–5. (Cited on pages 103 and 104.)

[LaC13] LaCie AG, *Wuala cloud storage service*, online, http://www.wuala.com, 2013. (Cited on pages 79 and 144.)

[LB10] Jean-Yves Le Boudec, *Performance evaluation of computer and communication systems*, EPFL Press, Lausanne, Switzerland, 2010. (Cited on pages 99 and 101.)

[LF04] Geoffrey Lefebvre and Michael J. Feeley, *Separating durability and availability in self-managed storage*, Proceedings of the 11th Workshop on ACM SIGOPS European Workshop (New York, NY, USA), EW 11, ACM, 2004. (Cited on page 12.)

[LGZ+09] Yongmei Liu, Yong Guan, Jie Zhang, Guohui Wang, and Yan Zhang, *Reed-solomon codes for satellite communications*, Control, Automation and Systems Engineering, 2009. CASE 2009. IITA International Conference on, July 2009, pp. 246–249. (Cited on page 11.)

[Li13] Keqin Li, *Parallel file download in peer-to-peer networks with random service capacities*, Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International, 2013, pp. 677–686. (Cited on page 47.)

[LM10] Avinash Lakshman and Prashant Malik, *Cassandra: A decentralized structured storage system*, SIGOPS Oper. Syst. Rev. **44** (2010), no. 2, 35–40. (Cited on page 84.)

[LPGM08] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller, *Measurement and analysis of large-scale network file system workloads*, USENIX ATC (Berkeley, CA, USA), USENIX Association, 2008. (Cited on pages 7, 25, 27, 72, and 78.)

[LS01] Nicholas Laneman and Carl-Erik Sundberg, *Reed-solomon decoding algorithms for digital audio broadcasting in the am band*, Broadcasting, IEEE Transactions on **47** (2001), no. 2, 115–122. (Cited on page 11.)

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease, *The byzantine generals problem*, ACM Transactions on Programming Languages and Systems **4** (1982), 382–401. (Cited on pages 26 and 137.)

[Lub02] Michael Luby, *Lt codes*, Foundations of Computer Science, 2002. Proceedings. The 43rd Annual IEEE Symposium on, 2002. (Cited on pages 12, 13, and 71.)

[LZT04] Martin Landers, Han Zhang, and Kian-Lee Tan, *Peerstore: Better performance by relaxing in peer-to-peer backup*, Proceedings on Peer-to-Peer Computing, 2004. (Cited on page 19.)

[Mac05] David JC MacKay, *Fountain codes*, IEEE Communications **152** (2005), 1062–1068. (Cited on page 71.)

[Mas81] James Massey, *Capacity, cutoff rate, and coding for a direct-detection optical channel*, Communications, IEEE Transactions on **29** (1981), no. 11, 1615–1621. (Cited on page 11.)

[May02] Petar Maymounkov, *Online codes*, Tech. report, New York University, 2002. (Cited on page 12.)

[MB09] Dirk Meister and André Brinkmann, *Multi-level comparison of data deduplication in a backup scenario*, Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference, ACM, 2009, pp. 8:1–8:12. (Cited on page 17.)

[MCL+06] Faruck Morcos, Thidapat Chantem, Philip Little, Tiago Gasiba, and Douglas Thain, *idibs: an improved distributed backup system*, International Conference on Parallel and Distributed Systems, 2006. (Cited on page 19.)

[MCM01] Athicha Muthitacharoen, Benjie Chen, and David Mazières, *A low-bandwidth network file system*, Proceedings of Symposium on Operating Systems Principles, ACM, 2001. (Cited on page 18.)

[MDWS10] John C. McCullough, John Dunagan, Alec Wolman, and Alex C. Snoeren, *Stout: An adaptive interface to scalable cloud storage*, Proceedings of the 2010 USENIX Annual Technical Conference, USENIX Association, 2010, pp. 4–4. (Cited on page 49.)

[Mic06] Microsoft Corp., *Description of full, incremental, and differential backups.*, online, http://support.microsoft.com/kb/136621, 2006. (Cited on page 5.)

[MM02] Petar Maymounkov and David Mazières, *Kademlia: A peer-to-peer information system based on the xor metric*, IPTPS '01, Springer-Verlag, 2002, pp. 53–65. (Cited on page 86.)

[NYGS06] Suman Nath, Haifeng Yu, Phillip B. Gibbons, and Srinivasan Seshan, *Subtleties in tolerating correlated failures in wide-area storage systems*, Proceedings of NSDI'06 (Berkeley, CA, USA), USENIX Association, 2006, pp. 17–17. (Cited on pages 96 and 98.)

[Ope14] Open Source Project, *Dibs: Distributed internet backup system*, online, http://sourceforge.net/projects/dibs/, 2014. (Cited on page 19.)

[Ora13] Oracle Corp., *Maintaining data integrity through constraints*, online, http://docs.oracle.com/cd/B12037_01/appdev.101/b10795/adfns_co.htm, 2013. (Cited on page 66.)

[PJB11] Lluis Pamies-Juarez and Ernst Biersack, *Cost analysis of redundancy schemes for distributed storage systems*, CoRR (2011). (Cited on page 13.)

[PJGL10] Lluis Pamies-Juarez and Pedro Garcia-Lopez, *Maintaining data reliability without availability in p2p storage systems*, Proceedings of the 2010 ACM Symposium on Applied Computing (New York, NY, USA), ACM, 2010. (Cited on pages 13 and 41.)

[PLS+09] James S. Plank, Jianqiang Luo, Catherine D. Schuman, Lihao Xu, and Zooko Wilcox-O'Hearn, *A performance evaluation and examination of open-source erasure coding libraries for storage*, Proccedings of FAST'09, USENIX Association, 2009, pp. 253–265. (Cited on page 71.)

[PMG+13] J. S. Plank, E. L. Miller, K. M. Greenan, B. A. Arnold, J. A. Burnum, A. W. Disney, and A. C. McBride, *GF-Complete: A comprehensive open source library for Galois Field arithmetic. version 1.0*, Tech. Report UT-CS-13-716, University of Tennessee, September 2013. (Cited on page 72.)

[PO95] Bart Prenel and Paul C. van Oorschot, *Mdx-mac and building fast macs from hash functions*, Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology (London, UK, UK), CRYPTO '95, Springer-Verlag, 1995. (Cited on page 70.)

[PPT09] Giorgos Papastergiou, Ioannis Psaras, and Vassilis Tsaoussidis, *Deep-space transport protocol: A novel transport scheme for space dtns*, Comput. Commun. **32** (2009), no. 16, 1757–1767. (Cited on page 11.)

[PS07] John Proakis and Masoud Salehi, *Fundamentals of communication systems*, Pearson Education, 2007. (Cited on pages 76 and 145.)

[PSAS01] Marius Portmann, Pipat Sookavatana, Sëbastien Ardon, and Aruna Seneviratne, *The cost of peer discovery and searching in the gnutella peer-to-peer file sharing protocol*, Proceedings of the 9th IEEE International Conference on Networks, 2001, pp. 263–268. (Cited on page 126.)

[PSY09] Phillip Porras, Hassen Saïdi, and Vinod Yegneswaran, *A foray into conficker's logic and rendezvous points*, Proceedings of the 2Nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More (Berkeley, CA, USA), LEET'09, USENIX Association, 2009, pp. 7–7. (Cited on pages 87 and 148.)

[Rab81] Michael Rabin, *Fingerprinting by random polynomials*, Tech. report, Harvard University, 1981. (Cited on page 18.)

[RD01] Antony I. T. Rowstron and Peter Druschel, *Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems*, Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Springer-Verlag, 2001, pp. 329–350. (Cited on pages 18 and 86.)

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker, *A scalable content-addressable network*, Proceedings of SIGCOMM '01, ACM, 2001, pp. 161–172. (Cited on page 86.)

[RKYG13] Seyed Majid Razavian, Hadi Khani, Nasser Yazdani, and Fatemeh Ghassemi, *An analysis of vendor lock-in problem in cloud storage*, International eConference on Computer and Knowledge Engineering (ICCKE), 2013, pp. 331–335. (Cited on pages 3 and 131.)

[RP06] Sriram Ramabhadran and Joseph Pasquale, *Analysis of long-running replicated systems*, Proceedings of INFOCOM 2006, April 2006, pp. 1–9. (Cited on page 96.)

[RSA13] RSA Laboratories, *Pkcs #5: Password-based cryptography standard.*, online, http://www.rsa.com/rsalabs/node.asp?id=2127, 2013. (Cited on page 51.)

[RV13] Ganesan Ramalingam and Kapil Vaswani, *Fault tolerance via idempotence*, Proceedings of ACM SIGPLAN-SIGACT, POPL '13, ACM, 2013, pp. 249–262. (Cited on pages 56 and 66.)

[SGLM08] Mark Storer, Kevin Greenan, Darrell Long, and Ethan Miller, *Secure data deduplication*, Proceedings of the 4th ACM international workshop on Storage security and survivability, StorageSS '08, ACM, 2008. (Cited on pages 18 and 52.)

[Sho06] Amin Shokrollahi, *Raptor codes*, IEEE Transactions on Information Theory **52** (2006), no. 6, 2551 –2567. (Cited on pages 12, 13, and 71.)

[SK09] Jerome H. Saltzer and M. Frans Kaashoek, *Principles of computer system design: An introduction*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009. (Cited on pages 3, 11, 21, 22, 29, 56, 60, 66, 89, 130, 161, and 163.)

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler, *The hadoop distributed file system*, Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on, May 2010, pp. 1–10. (Cited on page 81.)

[SMLN+03] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan, *Chord: a scalable peer-to-peer lookup protocol for internet applications*, IEEE/ACM Trans. Netw. **11** (2003), no. 1, 17–32. (Cited on pages 17 and 86.)

[SNS88] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller, *Kerberos: An authentication service for open network systems*, Usenix Conference Proceedings, 1988. (Cited on page 51.)

[SR05] Richard Stevens and Richard Rago, *Advanced programming in the unix environment*, Addison-Wesley professional computing series, Addison-Wesley, 2005. (Cited on page 56.)

[SSVG13] Sriram Sankar, Mark Shaw, Kushagra Vaid, and Sudhanva Gurumurthi, *Datacenter scale evaluation of the impact of temperature on hard disk drive failures*, Trans. Storage **9** (2013), no. 2, 6:1–6:24. (Cited on page 103.)

[Sta97] William Stallings, *Data and computer communications - fifth edition*, Prentice-Hall India, 1997. (Cited on page 57.)

[Sto90] Philip Storey, *Worm disk drive systems*, Data Storage Technology, IEE Colloquium on, Feb 1990, pp. 6/1–6/3. (Cited on page 106.)

[SVIG06] Russell Sears, Catharine Van Ingen, and Jim Gray, *To BLOB or not to BLOB: large object storage in a database or a filesystem*, Tech. report, Microsoft Research, 2006. (Cited on pages 59 and 88.)

[SZT+08] Baron Schwartz, Peter Zaitsev, Vadim Tkachenko, Jeremy Zawodny, Arjen Lentz, and Derek J. Balling, *High performance mysql, 2nd edition*, second ed., O'Reilly, 2008. (Cited on page 88.)

[TCDM12] Laszlo Toka, Pasquale Cataldi, Matteo Dell'Amico, and Pietro Michiardi, *Redundancy management for P2P backup*, IEEE INFOCOM 2012, 2012. (Cited on pages 13, 41, 42, 95, 141, and 142.)

[TDM10] Laszlo Toka, Matteo Dell'Amico, and Pietro Michiardi, *Online data backup: A peer-assisted approach*, Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on, 2010, pp. 1–10. (Cited on page 38.)

[The14a] The Data Rescue Center, *Data recovery service*, online, http://www.thedatarescuecenter.com, 2014. (Cited on pages 1 and 129.)

[The14b] _____, *Data recovery service, pricing*, online, http://www.thedatarescuecenter.com/pricing.html, 2014. (Cited on pages 1 and 129.)

[The14c] The GnuPG Project, *Gnu privacy guard*, online, https://www.gnupg.org/, 2014. (Cited on page 19.)

[Tho14] Thomas Mager, *Implementation of distback*, online, http://www.eurecom.fr/~mager/DistBack/, 2014. (Cited on page 55.)

[TMEBPM12] Thomas Mager, Ernst Biersack, and Pietro Michiardi, *A measurement study of the Wuala on-line storage service*, IEEE P2P 2012, 2012. (Cited on pages 16, 53, 58, 71, 74, 76, 77, and 146.)

[TP11] Chad Teat and Svetlana Peltsverger, *The security of cryptographic hashes*, Proceedings of the 49th Annual Southeast Regional Conference (New York, NY, USA), ACM-SE '11, ACM, 2011, pp. 103–108. (Cited on page 50.)

[Tri96] Andrew Tridgell, *The rsync algorithm*, Ph.D. thesis, Australian National University, 1996. (Cited on page 62.)

[TSH+05] Joseph Tucek, Paul Stanton, Elizabeth Haubert, Ragib Hasan, Larry Brumbaugh, and William Yurcik, *Trade-offs in protecting storage: a meta-data comparison of cryptographic, backup/versioning, immutable/tamper-proof, and redundant storage solutions*, Proceedings of Mass Storage Systems and Technologies. 22nd IEEE / 13th NASA Goddard Conference, April 2005, pp. 329–340. (Cited on page 105.)

[TYP13] Burcu Tepekule, Utku Yavuz, and Ali Pusane, *On the use of modern coding techniques in qr applications*, Signal Processing and Communications Applications Conference (SIU), 2013 21st, April 2013, pp. 1–4. (Cited on page 11.)

[U.S14] U.S. Energy Information Administration, *Short-term energy and summer fuels outlook*, online, http://www.eia.gov/forecasts/steo/report/electricity.cfm, 2014. (Cited on pages 3 and 131.)

[VI12] Vinodh Venkatesan and Ilias Iliadis, *Effect of codeword placement on the reliability of erasure coded data storage systems*, Tech. report, IBM Reseach, 2012. (Cited on pages 30 and 78.)

[VN10] Kashi Venkatesh Vishwanath and Nachiappan Nagappan, *Characterizing cloud computing hardware reliability*, Proceedings of the 1st ACM Symposium on Cloud Computing (New York, NY, USA), SoCC '10, ACM, 2010, pp. 193–204. (Cited on page 82.)

[Vog99] Werner Vogels, *File system usage in windows nt 4.0*, Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (New York, NY, USA), SOSP '99, ACM, 1999, pp. 93–109. (Cited on pages 7 and 79.)

[Vog09] ———, *Eventually consistent*, Commun. ACM **52** (2009), no. 1, 40–44. (Cited on page 22.)

[Wal05] Chip Walter, *Kryder's Law*, Scientific American **293** (2005), 32–33. (Cited on pages 2 and 130.)

[WB99] Stephen B. Wicker and Vijay K. Bhargava, *Reed-solomon codes and their applications*, Wiley-IEEE Press, 1999. (Cited on pages 12, 13, 33, and 139.)

[Wea06] Hakim Weatherspoon, *Design and evaluation of distributed wide-area on-line archival storage systems*, Ph.D. thesis, EECS Department, University of California, Berkeley, Oct 2006. (Cited on page 118.)

[Wil92] Paul R. Wilson, *Uniprocessor garbage collection techniques*, Proceedings of the International Workshop on Memory Management (London, UK, UK), IWMM '92, Springer-Verlag, 1992, pp. 1–42. (Cited on page 89.)

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles, *Dynamic storage allocation: A survey and critical review*, Proceedings of the International Workshop on Memory Management, IWMM '95, Springer-Verlag, 1995, pp. 1–116. (Cited on page 89.)

[WOW08] Zooko Wilcox-O'Hearn and Brian Warner, *Tahoe: The least-authority filesystem*, Proceedings of the 4th ACM International Workshop on Storage Security and Survivability (New York, NY, USA), StorageSS '08, ACM, 2008, pp. 21–26. (Cited on pages 15 and 71.)

[YKMI88] Toshio Yamada, Hisakazu Kotani, Junko Matsushima, and M. Inoue, *A 4-mbit dram with 16-bit concurrent ecc*, Solid-State Circuits, IEEE Journal of **23** (1988), no. 1, 20–26. (Cited on page 12.)

[YS99] Jimmy Yang and Feng-Bin Sun, *A comprehensive review of hard-disk drive reliability*, Reliability and Maintainability Symposium, 1999, pp. 403–409. (Cited on page 101.)

[Zha07] Ji Gao Zhang, *Effect of dust contamination on electrical contact failure*, IEEE Holm Conference on Electrical Contacts, Sept 2007, pp. xxi–xxx. (Cited on page 103.)

[ZLL13] Ruijin Zhou, Ming Liu, and Tao Li, *Characterizing the efficiency of data deduplication for big data storage management*, IEEE International Symposium on Workload Characterization (IISWC), 2013, pp. 98–108. (Cited on page 127.)