# Combining Cloud Storage Systems

Bachelor-Thesis von Lennart Diedrich
September 2013

TECHNISCHE UNIVERSITÄT DARMSTADT

Department of Computer Science
P2P Networks Group

Combining Cloud Storage Systems

Vorgelegte Bachelor-Thesis von Lennart Diedrich

1. Gutachten: Prof. Dr. Thorsten Strufe
2. Gutachten: Benjamin Schiller

Tag der Einreichung:

**Erklärung zur Bachelor-Thesis**

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 30. September 2013

(Lennart Diedrich)

**Abstract**

With the advent of distributed platforms, cloud storage has become increasingly popular. Given this, recent years have seen a rather substantial surge in the number of providers; most of which offer a small, amount of free storage with the possibility for paid upgrades. Typically these providers are used to synchronize private files (i.e. documents and pictures) between multiple computers. In most instances, the price for said service is rather high and in terms of reliability regarding data access, when using a single provider the end user depends entirely on this vendor. A new approach is required, a system in which multiple providers work synergistically to produce both a reduced cost for end users as well as increased efficiency insofar as transfer rate and data reliability are concerned.

*Dropbox* as an example for this kind of cloud storage is one of the most popular providers on the market today; it saves one million files in three minutes and has more than 100 million user accounts. Moreover, the service offers two gigabytes of storage for free. Despite these facts, however, *Dropbox* is most certainly not without issues, larger media files tend to deplete storage space promptly. Furthermore, the platform has experienced two major service interruptions this past year in which access to the stored data was was not possible. A combined cloud storage system that transparently synchronizes files to multiple providers could have easily circumvented these issues.

In this thesis we therefore discuss different possibilities for to combining distinct cloud storage providers in order to build a combined system. We conduct two studies, one is aimed at measuring the file sizes stored in the cloud, the other is designed to measure the performance and transfer speeds of six different providers: namely, *Box.com*, *Dropbox*, *Google Drive*, *Skydrive*, *Sugarsync* and *Ubuntu One*. Through this endeavour, we develop strategies that improve the performance of the data transfer process within a combined system. Finally, we demonstrate that even the aforementioned preliminary strategies have a decent impact on transfer rate by developing a specialized simulation framework and performing an extensive simulation.

## Contents

## 1 Introduction

The terms *Cloud* and *Cloud Computing* have emerged since late 2007 and have grown in significance over the years. Now more than ever, the Cloud seems omnipresent. The National Institute of Standards and Technology defines cloud computing as *"a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."* [14]. In other words, it is a distributed computing architecture connected through the Internet. Modern cloud architectures can be divided into three service groups, IaaS, PaaS, and SaaS. IaaS (Infrastructure as a Service) architectures offers basic services like servers, virtual machines, or raw data storage. PaaS (Platform as a Service) architectures add an additional abstraction layer and offer a computing platform. These platforms often consists of databases, application environments or servers; Amazon EC2[1] or the Google App Engine[2] are examples for this kind. When using a SaaS (Software as a Service) architectures the provider offers a single application directly to the end-user (e.g., GMail[3], Microsoft Office 365[4]). This new computing paradigm has many advantages for the users: viz., its ease of use, scalability, reliability and flexibility. The user no longer needs to worry about many issues of the past; things such as updates for involved software, acquiring new hardware for growing demands, disaster recovery and accessibility are now all conveniently dealt with by a third-party provider.

Cloud storage is one of the primary uses of cloud-computing. Through it, digital storage is transferred directly into the cloud where it is distributed over multiple third-party servers. By making storage available in the cloud, the information quickly becomes ubiquitous; it is readily accessible from all over the world using the Internet (often at much lower cost than providing traditional data storage). Since the data is spread throughout the cloud it allows for an entire portion of new possibilities. Collaboration, joint editing, sharing and integration with other services are just a few examples. Furthermore, cloud storage is extremely scalable. It is possible to virtually obtain an almost unlimited amount of free space in different data center locations around the world. Cloud storage providers try to fulfill most of the essential requirements for maintaining data (including high availability, reliability, performance, replication and data consistency [21]). As a consequence, less effort is required on behalf of the users; they do not have to worry about data backups or disaster recovery. However, aside from these benefits, there are some concerns worth noting, especially when it comes to security and privacy. One has to keep in mind that all the data is in the hands of a third-party, which cannot be controlled directly, and stores information out of the owner's reach. When the service provider is down, intentionally or inadvertently, there is no access to the data. This can be even more severe when a service provider is shut down altogether, perhaps because of legal disputes as in the case of Megaupload, whose servers were seized due to alleged copyright infringement[5]. Another disadvantage is the required internet connection, not to mention the required bandwidth. Costs in both these areas may eventually grow, which might lead to a vendor lock-in, where the user is dependent on a vendor because of enormous switching costs. From a privacy perspective it is also hard to tell who has access to the data and how they are interacting with it, especially after the recent disclosure of the NSA's warrantless surveillance of internet traffic[6].

Cloud storage providers can be divided into three groups [1]:

1. **High-End Storage Provider:** High-end cloud storage providers are meant to primarily work as backend for other service providers or enterprise applications. They only provide the simple storage capability without any further function. Examples of this kind of storage provider are Amazon S3[7], Rackspace Cloud Files[8], and EMC Atmos[9]. Typically these providers charge per stored GB and per transferred GB. With them, it is usually possible to choose between different data center locations. Some providers even offer the possibility of choosing between different levels of data redundancy, more important files can be stored using a higher redundancy.

2. **File Storage Providers:** File storage providers like Dropbox, Google Drive, Ubuntu One, and Box.com specialize in storing files for the end-user. These providers use high-end cloud storage providers as backend storage while developing further functionality for their own services. Such services often include shared folders (publicly accessible folders) and joint editing or collaboration tools. Most of these providers ship their own synchronization

[1] http://aws.amazon.com/ec2/
[2] https://appengine.google.com
[3] http://mail.google.com
[4] http://office.microsoft.com/de-de/products
[5] http://www.wired.co.uk/news/archive/2013-06/20/dotcom-data-deletion
[6] http://www.reuters.com/article/2013/08/21/us-usa-security-nsa-idUSBRE97K02V20130821
[7] http://aws.amazon.com/s3/
[8] http://www.rackspace.com/cloud/files/
[9] http://www.emc.com/storage/atmos/atmos.htm

software which directly connects the end-users' computer with their storage backend and synchronizing specified folders with multiple devices. In an effort to reduce cost, these kinds of providers typically implement data deduplication [22] [18], which means that a file is only stored once although it has been uploaded from different user accounts. This is also the reason why most of these providers do not support client-side encryption, for if they did data deduplication would no longer be possible, since two equal files are different when they are encrypted.

3. **Backup Providers:** Backup storage providers offer personal file backup services, which target individuals or small businesses interested in having backup files online. Similar to file storage providers, backup storage providers ship with an individual backup software for the end-user. Examples for these providers are Mozy[10], Acronis Online Backup[11], Comodo System Online Backup Software[12] and CrashPlan[13].

| Provider | Vendor | API | Free Storage | Avg. Price / GB | Possible Upgrade Tiers |
|---|---|---|---|---|---|
| Box.com | Box Inc. | yes | 2 GB | $0.3997 | 25 GB / 50 GB |
| Dropbox | Dropbox Inc. | yes | 5 GB | $0.0999 | 50 GB / 100 GB |
| Google Drive | Google Inc. | yes | 5 GB | $0.0548 | 25 GB / 100 GB / 200 GB |
| SugarSync | SugarSync Inc. | yes | 5 GB | $0.1609 | 30 GB / 60 GB / 100 GB / 250 GB / 500 GB |
| SkyDrive | Microsoft Corp. | yes | 7 GB | $0.0416 | 20 GB / 50 GB / 100 GB |
| Ubuntu One | Canonical Ltd. | yes | 5 GB | $0.1495 | bookable packages with 20 GB |

**Table 1.1.:** Analyzed cloud storage providers as of May 2013

This thesis will mainly concentrate on file storage providers, since these providers offer a limited amount of storage for free and enable users to upgrade to a higher tier for a fee (cf. Table 1.1). The costs per GB ranges from $0.04 to $0.4. In order to use these providers conveniently it is mandatory to download the synchronization software for each provider and configure it accordingly, as long as one's platform is supported by the provider. In order to overcome these issues it seems obvious to implement a single, platform independent application that connects and combines multiple storage providers. In other words, a Combined Cloud Storage System. This system has many advantages over the traditional approach of using each provider separately:

- Using multiple providers combined in one application decreases the costs for the available storage, since both free and paid plans can then be combined. Furthermore, it is possible to create multiple accounts on each provider as long it is permitted by the initial terms of use.
- This custom combining application can add further functionality (e.g., a functionality similar to RAID (Redundant Array of Independent Disks)). In such a RAID setup, each provider represents a different hard drive. By using different RAID levels it is possible to improve file redundancy through mirroring files to different providers (RAID 1), increasing the combined storage capacity by using striping techniques (RAID 0), or even use striping with parity information (RAID 5) to compensate for the complete drop out of a cloud storage provider.
- Moreover, a custom application could extend the providers built-in features by security enhancements that increase the users' privacy. For example, data could be encrypted locally, or documents could be signed using a digital signature to verify the authenticity to collaborators.
- Since the providers handle uploaded files differently, it is possible to optimize the transfer speeds for each provider.
- Complete elimination of vendor lock-ins, since the data is distributed over several providers.

In this work we gain insights into how such a combined system can be built and what has to be considered while creating such an application. In Chapter 2 we show related work on this topic and then we discuss in Chapter 3 the goals and requirements one has keep in mind when building such a system. We will assert, that we need different strategies, which are based on distinct metrics. In Chapter 4 we will measure some of these metrics for different providers. In order to show, that the proposed metrics and strategies have an impact on the data transfer rate, we conduct a simulation in Chapter 5.

[10] http://mozy.com/
[11] http://www.acronis.com/articles/online-backup/
[12] http://backup.comodo.com/
[13] http://www.crashplan.com/

## 2 Related Work

The overall topic of cloud storage is currently under heavy research and in a constant development, which is one of the reasons why so many applications and cloud storage providers have entered the market during the last couple of years. A lot of research has been done in the field of cloud computing, despite the fact that cloud storage has been slightly neglected. Most of the work on this topic focuses on high-end providers like Amazon or Rackspace. In this Chapter we will therefore introduce some academic research alongside some interesting applications which treat the combination of online cloud storage using different techniques and approaches.

### 2.1 CloudRaid

CloudRaid [20] is a system which combines multiple cloud storage providers in an approach similar to RAID. It runs as a local web service and supports three different cloud storage providers (*Dropbox*, *Ubuntu One* and *SugarSync*). Multiple accounts can be created for each of the storage providers, which in turn are combined in an *Array*. The system supports two synchronization modes, which can be configured for each of the Arrays. The service listens for local file system changes within a configured directory and uploads the changed files according to the synchronization mode to its respective storage provider.

Similar to the RAID storage technology, the Arrays can be configured to use one of two synchronization modes: mirroring (similar to *RAID-1*) or striping (similar to *RAID-0*). The mirroring mode synchronizes the files to every storage provider selected in this Array. Therefore this mode increases the file redundancy and likewise data reliability and stability. However, when using this mode the size of the combined storage is determined by the smallest remote storage provider. Striping mode, on the other hand, uses multiple providers to increase the overall capacity of the system. Each new cloud storage provider adds new capacity to the entire system. The file distribution strategies is rather simple, it uploads the file to the first provider and switches to the next one, as soon as the first provider is operating at full capacity.

One if the disadvantages of CloudRaid is, that the local service does not listen for changes on the cloud storage providers. A user that changes files on another synchronized computer, or by using the web interface of the storage provider the data will not be noticed by the CloudRaid system.

### 2.2 Syncany

Syncany[1] is an open-source file synchronization application whose overall goal is to optimize remote storage usage. Moreover, it aims at utilizing the bandwidth and available resources by minimizing storage and synchronization time for Syncany clients [11]. A further note of interest is the author's attempt at discovering optimal deduplication algorithms. Once this has been done, a multichunk concept can then be introduced in order to reduce the data transfer time between the client and the remote storage. Syncany is able to store files remotely by using different kinds of storages and transfer protocols (like FTP, IMAP, WebDAV, SFTP). Furthermore, it provides clients with the option of using professional cloud storage providers (e.g. Amazon S3, Rackspace Cloud files, Box.com). Moreover, Syncany encrypts files locally before they are uploaded to the corresponding remote storage. Because of that, an attacker or even the cloud storage provider itself is not able to access the data that is stored using Syncany. In case of accidental data deletion, Syncany saves the entire history of a file, which enables the user to reconstruct old version of file from the remote storage.

### 2.3 RACS

Libdeh et al. describe in [1] a cloud storage proxy called RACS (Redundant Array of Cloud Storage) that transparently stripes data across multiple high-end cloud storage providers, called repositories (e.g. Amazon S3, Rackspace Cloud Files). RACS primarily deals with the economic loss and tries to estimate and reduce the cost of switching storage providers (in order to prevent vendor lock-ins). RACS also makes use of redundant striping with erasure coding in order to promote availability. RACS mimics the Amazon S3 interface to present itself as a proxy between the client application and the different providers. Similar to this work they describe "policy hints" which let a client specify their preferred repositories in order to exploit geographic proximity or pricing schemes. These policy hints are not implemented in the RACS prototype.

[1] http://www.syncany.org/

## 2.4 HAIL

HAIL, High-Availability and Integrity Layer for cloud storage, is a distributed cryptographic system that allows a set of servers to prove to a client that a stored file is intact and retrievable[4]. Its goal is to be resilient against a *mobile adversary* that is able to potentially corrupt all servers across the system. To achieve this goal the authors endeavour to replicate files onto different storage providers, and in so doing make use of cryptographic systems in order to prove the readability (POR) and the data possession (PDP).

## 2.5 Deltacloud

Deltacloud[2] is an open-source Apache Software Foundation project. It is aimed at enterprise applications, that use high-end storage providers as storage backend. Deltacloud defines an application programmable interface (API) in order to abstract between the different storage providers' APIs. It provides unified access using a RESTful API and maintains long-term stability, as well as backwards compatibility, for third party applications.

One of the main advantages is the possibility that enterprise source code can be tested against a dummy (or low-cost) storage provider using Deltacloud, since the individual providers' storage API is abstracted.

## 2.6 Otixo and CloudFuze

Otixo[3] and CloudFuze[4] are both web applications which provide an online file manager that combines multiple storage providers. They do not store the users' data, but rather provide a user interface to simplify file operations between the client and different providers. Both system provide their own synchronization software for mobile operating systems, but CloudFuze has also an own desktop application which can sync selected storage providers to the end-users' local hard drive. Both providers have their own mobile applications.

## 2.7 Owncloud

Owncloud[5] is an open-source web application that provides private cloud storage on a private server. It is written in PHP and uses the local storage to save the user files, calendar data, and contacts. Furthermore, it is possible to configure "Custom Mount Configurations", which integrate third party remote storage (FTP, SMB, WebDAV, Amazon S3, Dropbox, Google Drive, or OpenStack Swift) into its own filesystem tree. In terms of user authentication LDAP is supported, as well as server-side encryption to protect the data.

Ownloud is at the moment the only serious alternative, when private cloud storage is required. Since the data is located on the users' own server, they do not have to worry about unauthorized data access.

## 2.8 Bittorrent Sync

Bittorrent Sync[6] is a decentralized synchronization tool developed by BitTorrent Inc. in order to facilitate synchronizing personal files between multiple computers using peer-to-peer technology. On July 17th, 2013 it left the alpha phase[7] and is now publicly available. Bittorrent Sync does not synchronize to any centralized server in the cloud, and thus has no predetermined limitations regarding available space, bandwidth, or cost. Furthermore, it eliminates most of the privacy concerns since the data is never uploaded to a third party server and the communication between the clients is encrypted using a private key. Folders can be shared using a secret key with other clients. The system yields the peer-to-peer technologies in order to find and directly connect individual clients, that share the same secret.

2 http://deltacloud.apache.org
3 http://www.otixo.com/
4 http://www.cloudfuze.com/
5 http://owncloud.org/
6 http://labs.bittorrent.com/experiments/sync.html
7 http://blog.bittorrent.com/2013/07/17/now-in-beta-bittorrent-sync/

## 2.9 Others

**SparkleShare**

SparkleShare[8] is an open-source project that allows users to host their own cloud storage using Git as a backend. Since users run the servers themselves, there are no limitations regarding the amount of data a user can store (or the available bandwidth, for that matter). Moreover, the application is cross-platform compatible.

**FTPbox**

FTPbox[9], similar to SparkleShare, is an open-source software that allows its users to synchronize files using FTP. This application is in an early beta stadium.

**SharedSafe**

SharedSafe is a Windows desktop application that synchronizes files using IMAP, FTP or Dropbox. It has an integrated client side encryption function using AES.

[8] http://sparkleshare.org/

[9] http://ftpbox.org/

## 3 Combining Cloud Storage Systems

When implementing a combined system as mentioned in Chapter 1 there are some desirable goals an end-user must keep in mind (e.g., improved data transfer speed, safety and increased external storage). Some of these goals can be achieved easily. The external storage is increased by uploading files to the next cloud storage provider when the previous has no more free capacity. Safety, in respect to data resilience, can be improved easily by uploading files to multiple providers. However, there are also other goals which are harder to accomplish; in Section 3.1 we shed light on the goals that are essential when building a combined cloud storage system. Section 3.2 shows the requirements as well as a simple introduction to the most important RAID levels, while Section 3.3 sums up the conclusion yielded by our examination of requirements.

### 3.1 Goals

A combined cloud storage system should support the user when interacting with cloud storage. Moreover, it should also solve some of the deficiencies that occur when using only a single provider. In the best case all the goals (e.g., higher transfer speed, more storage, or improved data resilience) can be fulfilled by an application that combines multiple cloud storage providers.

Improving transfer speed when transferring files from, or to, the cloud storage providers is the most important goals. With improved data speed the end-user can be more productive, because it reduces the time it takes to download a file. The user has does not have to wait as long and has more time to work with the file itself. On the other hand files that can be uploaded faster are available earlier for others when working collaboratively with documents. Improved data transfer rate makes the system more responsive and facilitates all forms of work with respect to a given application for all users. When cloud storage is used as backup space, the remotely saved files can be retrieved faster, and so the time consumed for backing up files is measurably reduced. Thus, improving transfer speed is an essential goal; one that is much more easily met with the implementation of a combined provider.

Most cloud storage providers offer a limited amount of free storage (cf. Table 1.1) for each user account with the additional possibility of upgrading to a higher tier. These confined tiers have distinct storage limitations that range from 20 gigabytes (GB) to 200 GB, usually with a rather high cost (between $0.04 and $0.4 per GB). In order to minimalize the costs it is desirable to combine multiple free tiers and aggregate them into a combined, free system. Another advantage of this aggregated storage is the extra space gained without upgrading to a higher tier (something which enables the end-user to store more files in the cloud).

Another goal of a combined system is to improve the data safety. With traditional on-site data protection it is very hard to maintain disaster recovery plans that cover all foreseeable catastrophic risks (e.g., fire or flood–which creates the necessity for off-site backups [12]). Cloud storage providers can offer these off-site backups but [3] shows that cloud storage providers can also raise security and privacy concerns that the user is often not aware of. With the help of HoneyDocs[1], documents that can detect when they were opened, it was recently discovered that Dropbox opens the documents of their users, and Google has been in media because of leaking private documents[2]. A combined system can eventually reduce the security and privacy risks and increase the data safety. Furthermore, it is of initial importance to completely abolish the risk of a vendor lock-in. That is, when the user is entirely dependent on a vendor because of immense switching costs. High-end cloud storage providers charge the storage fee and an extra compensation for every transferred gigabyte from, or to, the provider's data center. When a high amount of data is stored on such a provider, the cost of downloading the complete information and upload it again to another data center are often not manageable. In a combined cloud storage system it would be easy to prevent a vendor lock-in by distributing files across multiple connected providers.

Moreover, stability is another goal that can be improved with a combined system. If a provider is unstable and often not reachable, the files stored on this provider are also not accessible. A combined system could classify providers regarding their stability and use the ones with lower stability as backup for others or even totally avoid it. On the other hand providers that are known to be very stable can be used the hold files that are accessed frequently.

### 3.2 Requirements

In order to achieve these goals, a combined cloud storage system aggregates multiple storage providers into a single system. These goals expose basic requirements which should be considered when building a combined system. Since

[1] https://www.honeydocs.com/

[2] http://www.itnews.com.au/News/139456,google-docs-leaks-private-data-online.aspx

combining different remote storage is very similar to combining local storage, there are a lot of requirements that are congruent with the requirements found in different RAID (Redundant Array of Independent Disks) setups. RAID is a technology that combines a disk array, which consists of multiple independent disks, into a large, high-performance logical disk [5]. The data is striped across the independent disks using different distribution strategies called RAID levels. The RAID storage technology is a mature technology that has solved a lot of difficulties and is now widely spread knowledge in order to create reliable, available and efficient storage capacity [16].

The RAID-0 level favors simple striping and does not employ redundancy at all, which results in high write performance, since there is no redundant information to store. Given this, a single disk failure will result in immediate data-loss [5]. This setup can easily be transferred to a combined cloud storage system that aggregates multiple cloud storage providers in a single drive. This drive's capacity will be the sum of the size offered by the individual provider. In the case that the end-user has a high bandwidth, or, equivalently, that the provider does not make the full bandwidth available for the end-user, such a system could lead to a noticeable performance push—since there are multiple storage backends to which the user can upload concurrently. Similar to the conventional system, a drop out of a single provider results in loss of the data stored by this particular provider, since there is no redundant information stored by other providers. Another drawback of this configuration is therefore that it is difficult to work with files collaboratively. Since each file can be uploaded to different providers, it is thus often out of the reach of the participating party.

The RAID-1 level emphasizes mirroring without parity information or striping. Every drive is mirrored to a second hard drive. This means that twice as many disks are needed to store the original data [5]. When one drive fails, every bit of information is redundantly stored on the other hard drive. The data can be read from any of these two devices which increases the reading performance of this setup. However, the system is not very storage efficient since half of the storage is dedicated to mirror the data. Furthermore, storage capacity is given away when, for instance, one hard disk is smaller than the other, since the storage capacity differs between both hard drives. Generally this RAID level is also easily adaptable to a combined cloud storage system. Every provider represents an independent disk and providers with equal size can be used as mirroring counterparts.

RAID-10 is a combination of the levels RAID-1 and RAID-0. It supports mirroring as well as striping. By combining the advantages from the RAID levels 0 and 1 to overcome their counterparts drawbacks, mirroring to improve redundancy and striping to increase performance and storage capacity. Thus, at least four hard drives are required; two hard drives to build the striping RAID-0 and another two hard drives to mirror each of the previous two used in the RAID-0. As this RAID level has been created by combining two other levels, the scheme can be transferred to a combined cloud storage system by combining the different approaches for each of the respective RAID levels.

The RAID-5 level combines reliability with storage efficiency by using a block-level striping functionality together with parity information. The user's data is divided into blocks and distributed over the available disks. The parity information is calculated by using the XOR operation over the data blocks and is also distributed uniformly over all of the available disks. This setup improves read activities and reliability since it allows all disks to participate in serving read operations. This composition can handle a complete failure of any hard disk since the stored data can be restored using the data and parity information stored on the other hard drives (which allows the lost parity to be calculated again). In order to build a RAID-5 array at least three hard drives are required. However, since the data is divided into blocks and stored on different providers, the files are not accessible using another interface than the combined system. A user that logs into the web interface of a provider will not be able to see or edit the files stored using the RAID-5 level.

All of the explanations regarding the RAID level imply a similarity of the characteristics of each cloud storage provider, as well as the comparability of hard disk and remote storage. However, there are a lot of differences between remote storage and local hard drives. One of the big differences is the higher latency when accessing files on remote storages. Since the cloud storage provider is connected using the Internet, the time to access and download files takes proportionally long. Thus, not only does it take longer, but the variance for time to execute the request is also higher. Moreover, the access time in a RAID setup is often nearly identical for files on different hard drives, but when using cloud storage one has to keep in mind that the characteristics of providers can differ severely (characteristics like transfer speed or rate-limiting). So, in order to improve a combined cloud storage system we need to be distinctly aware of these characteristics to make the correct decisions. By the time we know these provider specific characteristics, we can employ different *strategies* when distributing files over the individual cloud storage provider.

These strategies vary for the different RAID levels that are implemented by a combined system. Given a RAID-0 setup, the appropriate strategy has to decide, based on a predefined metric, which storage providers should be used to store the given file. The metric could be one of the described storage provider's characteristics, e.g., free capacity, the current utilization or even a combination of these characteristics (e.g., fastest provider with lowest utilization). Using a RAID-1 setup, the strategy has to choose, appropriately, which provider should mirrors the other. It is important that the mirror has at least as much capacity as the original provider, since the second provider has to handle all the files uploaded to the corresponding provider. When using providers with different storage capacities, it is even possible that the bigger provider acts as a mirror for multiple smaller storage providers in order to avoid unused remote storage. When a RAID-5 array is selected, the strategy has to split the data into blocks in order to calculate the parity information. This RAID

level however, has the disadvantage of using $n$ providers where the smallest provider offers the amount of $c$ storage. The maximum usable size is therefore calculated as $(n-1) \times c$, which means that the overall size is defined by the smallest provider. Providers with more capacity are thus unable to be utilized completely. When combining multiple RAID levels, the strategy must therefore guide the execution of tasks from both RAID levels in order to store files on the combined remote storage.

## 3.3 Summary

In this Chapter we showed that there are a lot of different details one must keep in mind when building a combined cloud storage system. Some of these details are predetermined, like the RAID level and the file distribution strategies specified by this level. Other details are provider specific and can be acquired using a simple measurement study. In order to improve the quality of a combined system it is therefore important to know details like the upload and download transfer rate, especially for different file sizes. It is also important to be aware of the capacity offered by the providers, the general stability and rate-limiting components that reduce the provided bandwidth or the amount of requests that can be conducted to the provider's API. All these characteristics need to be measured in order to build different strategies that take the currently best decision and upload the file to the most suitable provider. With respect to the bandwidth, it is important to measure the transfer rate during different times of the day in order to detect possible bandwidth utilization peaks [15].

## 4 Measurements

In order to improve the data transfer rate when uploading or downloading files from cloud storage providers, it is necessary to examine some characteristics of the different providers. The most important property of a provider is the maximum transfer bit rate, which depends on multiple factors (of which the proximity between the client and the server figures prominently). With a high proximity the transfer rate will decrease and the latency between the parties will increase. Given that most of the cloud storage providers are US-based, this is an important matter. Amazon, for example, maintains data centers in multiple locations around the world: in USA (Virginia, California, Oregon), Europe (Ireland), Asia (Singapore, Tokio), Australia (Sydney), and South America (Sao Paulo)[1]. Other important factors are the bandwidth of the providers and current utilization. When a lot of users are interacting concurrently with files the transfer rate will decrease significantly. Last but not least, many providers install rate-limiting functionality to protect their APIs, prevent denial of service (DoS) attacks and flash crowds (in which one or more links in the network become severely congested [13]).

To determine these characteristics we describe two measurement studies: the first to gain insight into exactly how cloud storage is used by the customer and how big the stored files are; the second to measure the transfer speeds of the different storage providers for different file sizes obtained in the first study.

### 4.1 File Size Distribution Measurement

To establish a cornerstone which we can use for the measurement of the storage providers performance in 4.2 we need to know how many files are stored on a typical computer (especially stored in the synchronized folders of the different cloud storage providers). The file-system contents have already been researched [7], especially the change in file size over time by Agrawal et al. [2], but to the best of the authors knowledge, not for the files synchronized to cloud storage providers.

The previously mentioned studies showed that the number of files as well as the file size have increased steadily in past years: between 2000 and 2004 the mean file size increased from 108 KB to 189 KB [2], which equates to a yearly growth of roughly 15 %. The median weighted file size increased even more drastically from 3 MB to 9 MB, which can be explained by the strong increase of audio files during this period. Interestingly, Evans et al. [7] predicted that video files will undergo the same growth in the future.

Previous work on this topic focused on analyzing the content of complete hard drives, which includes files only used by the operating system (e.g., libraries or configuration files). A lot of these files are rather small or even empty [2], something that could have an important impact on the file size distribution statistics. Furthermore, there are different usage patterns when comparing hard drives and cloud storage. Normal hard drives contain a lot of data, not to mention persistent and temporary files, whereas cloud storage is used to store documents and to share individual files with others, or to backup important files off-site.

#### 4.1.1 Data Collection

To collect the data we use a tool called FSstats[2] developed by Shobhit Dayal [6]. FSstatas is a program written in perl[3] that scans a given directory tree and creates statistics on file attributes (e.g., file EOF (file size), file capacity used, file positive/negative overhead (where file capacity used is more or less than file size), directory size in entries and in bytes, file name length, hard links, symbolic link length and file age [6]). This program emphasizes user privacy and anonymity in order to encourage participants within the study to submit their collected data–therefore promoting the studies' findings while ensuring that no sensitive information (such as file names, file types, extensions or any other information regarding content) is collected. The program is open source and released under the *GNU General Public License* (GPL).

We send the program and detailed instructions on how to gather the data to different persons of a mailing list of the department of computer science at the TU Darmstadt. We asked them to submit the statistics for each cloud storage provider's synchronized data directory as well as the statistics of their local documents directory.

[1] http://aws.amazon.com/s3/

[2] http://www.pdsi-scidac.org/fsstats/

[3] http://www.perl.org/

### 4.1.2 Results

We have received sixty-seven log files from different storage providers and local documents folders. FSstat scanned 2,052,710 files with an average file size of 493.87 KB, minimal file size of 0 KB and maximum file size of 11.17 GB. In comparison to Agrawal's study in 2007 [2] which measured an average file size of 189 KB and an annual growth rate of roughly 15 %, in our study we could only attest a file size growth of roughly 11 % annually.

FSstats groups files into buckets with file sizes of the power of two, i.e., each bucket contains the number of files with a file size between $2^n$ and $2^{n+1}$ (cf. Figure 4.1). The first bucket with file sizes between 0 KB and 2 KB contains 839,576 files which equates to 40.9 % of the total amount of files. These files only seize 0.00579 % of the total used space whereas the bucket containing the files from 4 GB to 8 GB contains 25 files and seizes 17.69 % of the total used space. 98.91 % of all the files only allocate 21.39 % of the total used space.

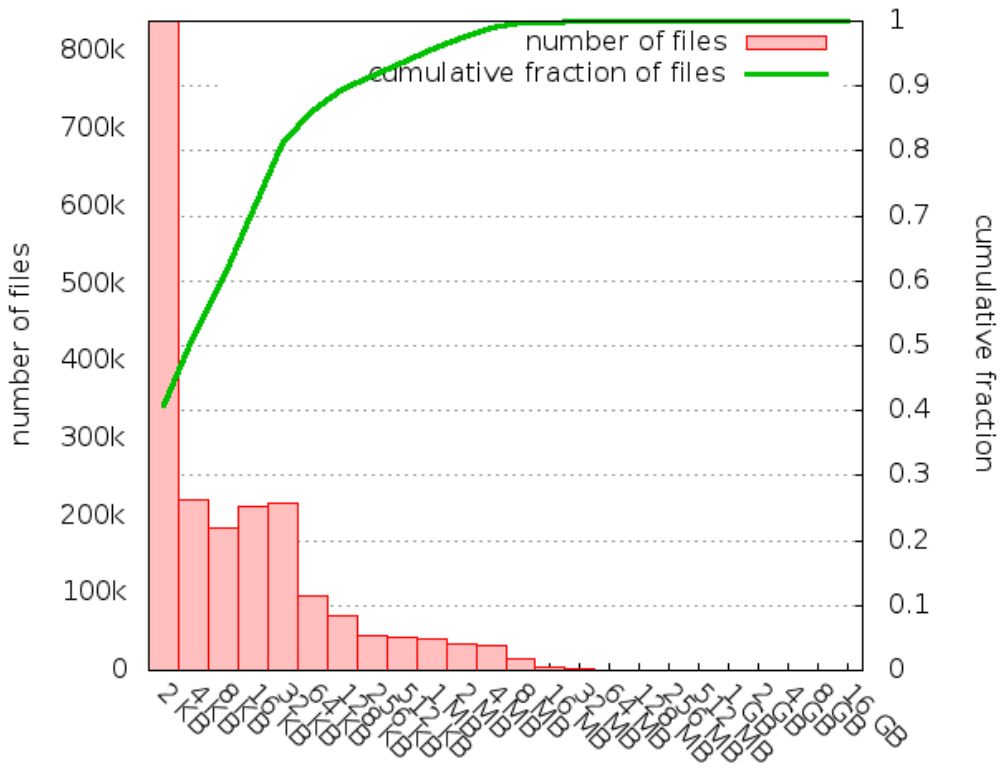



**Figure 4.1.:** Number of files with cumulated fraction

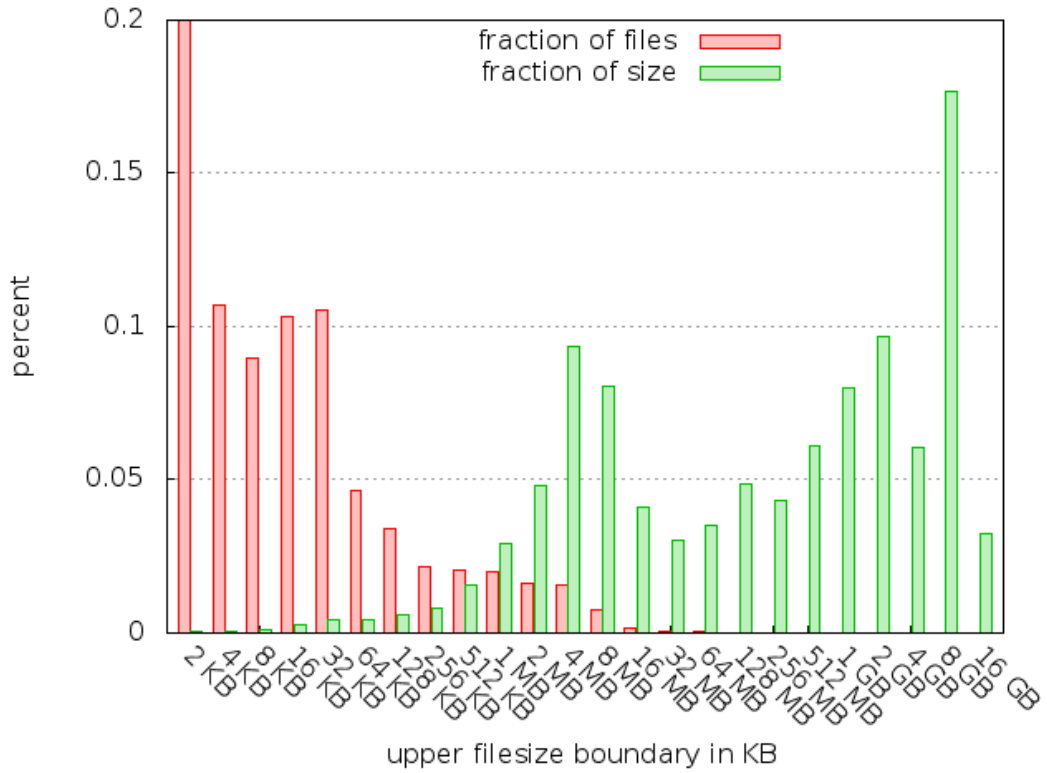



**Figure 4.2.:** Fraction of the number of files (red) and the space allocated by these files (green), trimmed at 20 %

## 4.2 Measurement Study of Cloud Storage Providers

In Chapter 3 we explained the goals and requirements in order to build a combined cloud storage system. We showed that it is necessary to know many details concerning the providers in order to build strategies that are able to distribute files according to distinct rules. The details we want to know concerning the different providers are the upload and download transfer rate as well as their overall performance throughout particular times of the day. Moreover, we desire to know the general long term stability of each system and possible rate-limiting components installed by the provider. We need to test the transfer rate during different times of the day in order to find possible bandwidth utilization peaks. Furthermore, this measurement study should also reveal information regarding the long term stability of each provider and show possible service disruptions. During such service disruptions, the files stored in the cloud are not accessible and thus outages should be avoided at all costs.

Supposing one provider has rate-limiting functionalities installed, then we should also determine whether these limitations are based on the IP address or the account session information of the end-user. With IP address rate-limiting, other users with the same originating IP address (e.g., when using Network Address Translation (NAT)) would then also be affected by this restriction.

To the best of the authors' knowledge this kind of measurement study for cloud storage providers has never been done before and will give new insights into the performance of cloud storage providers. Moreover, it will allow for the interesting possibility of comparison (insofar as performance is concerned) between the big providers.

### 4.2.1 Test Formalization

In order to test a cloud storage provider if necessary a clearly defined formalization which describes the current setup. This formalization should reflect every important aspect that has to be considered when dealing with bandwidth tests. Although the file size has no effect on the bandwidth directly, we must nevertheless add this attribute to our test formalization, since it is at least conceptually possible that different providers handle different files in distinct manner. Furthermore, it should be possible to test the storage providers from different networks with different Internet connections; it is important to reveal any rate-limiting systems, hence we need to add the user account as an attribute of our formalization.

In order to describe a test run formally, we propose a model in which a test combination is defined by a 4-tuple, e.g., $(10, 2, 1, 3)$; where 10 is the file size of the transferred file; 2 the amount of requests dispatched to the provider; 1 an identifier for the chosen network and 3 an identifier for the used user account. We define the mapping

$$TEST, BW : FILESIZE \times REQUESTS \times NETWORKS \times USER \mapsto \mathbb{R}^+$$

with $FILESIZE, REQUESTS, NETWORKS, USER \subseteq \mathbb{N}$ and $BW$ being a given and not further specified function to determine the bandwidth. Then the test function is defined as follows:

$$TEST(f, r, n, u) = \frac{1}{r} \sum_{i=1}^{r} BW(f, r, n, u)$$

This function defines the bandwidth for a provider by calculating the average bandwidth for the executed requests $r$. A possible test to determine the maximum bandwidth for a storage provider by using one account in one network could have these variables

$$f \in \{100\}, \quad r \in \{1\}, \quad n \in \{1\}, \quad u \in \{1\}$$

and would create the tuple $(100, 1, 1, 1)$, where one file of 100 Bytes would be uploaded using account 1 and network 1.

Another test to determine the maximum bandwidth for uploads from two different networks using one user account can be described as

$$f \in \{100\}, \quad r \in \{5\}, \quad n \in \{1, 2\}, \quad u \in \{1\}$$

and thus it creates the tuples $(100, 5, 1, 1)$ and $(100, 5, 2, 1)$. In this test case five files with a size of 100 Bytes are upload to the according providers using a single user account from two different networks. In order to check for rate limiting functionality it is possible to use multiple accounts

$$f \in \{10\}, \quad r \in \{5\}, \quad n \in \{1\}, \quad u \in \{1, 2\}.$$

### 4.2.2 Expected Test Results

In the worst case scenario the transfer speed varies during different times of the day; it may very well be that we hit our rate limit quite early, which may impede the entire measurement study. The best case scenario is one in which the complete opposite takes hold: every provider has consistent data transfer rates throughout the day and we do not cross any rate-limiting functionality. Most likely the results will rank in between these two described scenarios: the transfer rate will be consistent during the day because most of the providers serve international clients from all over the world with different timezones, but we may hit rate limiting constraints on some providers. Possible rate-limiting can be based on the user account or on the IP address and can be triggered after the transmission of a specified amount of data, or by hitting a maximum number of allowed requests. Typical rate-limiting concepts to avoid distributed denial of service (DDoS) attacks can throttle the bandwidth [19] to the originating IP address, but when dealing with heavy user accounts it is advisable to only limit the originating customer directly [17].

### 4.2.3 Implementation

In order to conduct this measurement study we require the possibility to upload and download files to the corresponding cloud storage provider. All of the providers offer a RESTful application programming interface (API), which stands for Representational State Transfer [8]. REST is a simple stateless architecture that can be accessed and modified using the Hyper Text Transfer Protocol (HTTP) operation. This architecture is very flexible since every resource can be accessed through an assigned, unique universal resource indicator (URI).

These APIs can be used to access the files on the different cloud storage providers without using the corresponding synchronization software. Hence, we have created two interfaces: one to extract the remote file system operations for our local application (called *StorageProvider*) and the other to authenticate the user and authorize each request executed by our application (called *Authenticator*). A *Connector* for a cloud storage provider has to implement these two interfaces in order to conduct our measurement study.

The whole application is based on the build automation tool Apache Maven[4]. It is divided into the following modules:

[4] http://maven.apache.org

- **CloudStorageSystems** - parent project containing all modules
  - **connectors** - containing all the connectors to the different cloud storage providers
    - **common** - exceptions and interfaces shared by all connector implementations
    - **box**
    - **dropbox**
    - **googledrive**
    - **skydrive**
    - **sugarsync**
    - **ubuntuone**
  - **service** - parent project containing all service related modules
    - **container** - the deployable service container
  - **Test** - parent package for all test related modules
    - **Test** - the actual testing environment
    - **Evaluation** - evaluated the test results
    - **Simulation** - simulation module

#### Connectors

The connector module is the parent package for all the connectors of the different storage provider. Each connector implements the REST API provided by the different cloud storage provider. The *common* package contains used exceptions (i.e., the *StorageException* and *AuthenticationException*), an HTTP Client with helper functions, simple objects, utility classes and interfaces shared by all the connector implementations. Every connector needs to implement the *Authenticator* and *StorageProvider* interface and thus depends on the commons package.

The Authenticator is used to authenticate the user against the storage provider and to sign HTTP requests with the credentials obtained by the authentication process. Furthermore this interface describes methods to store and load the user credentials as JSON to a file, so they can be used later. Since the application only has reached an early stage and has only been used to conduct this study, there are some drawbacks regarding the authentication. Most of the providers offer OAuth 2.0 as authentication protocol, therefore a callback address is required, to transfer the authorization code to the application. This part of the system has not been implemented yet, hence the code has to be entered manually during the first authentication process.

After the authentication is complete, you can store the obtained credentials using

```
authenticator.saveConfiguration(new File("provider.json"));
```

and reuse them at a later stage using

```
authenticator.loadConfiguration(new File("provider.json"));
```

The authenticator also stores the obtained lifetime of the access tokens and updates them automatically without any further ado.

The StorageProvider interface contains methods like *createFile*, *createFolder*, *deleteFile* and is used to directly communicate with the provider's REST API. The StorageProvider needs a provider-specific *Authenticator*, to sign its HTTP request with the obtained credentials.

```
File dropboxAuthFile = new File("dropbox.json");

// authentication process
DropboxAuthenticator dropboxAuth = new DropboxAuthenticator();
if (dropboxAuthFile.exists()) {
  dropboxAuth.loadConfiguration(dropboxAuthFile);
} else {
  dropboxAuth.authenticate();
  dropboxAuth.saveConfiguration(dropboxAuthFile);
}

// create StorageProvider connection
StorageProvider dropbox = new DropboxProvider(dropboxAuth);
```

Now the StorageProvider is authenticated and it is possible to transfer files:

```
FileObject file = box.createFile(null, new File("~/file.txt"));

// download remotely change files
file = dropbox.downloadFile(file);

// delete files again
dropbox.deleteFile(file);
```

**Service**

The whole system is designed to run as a service on the local machine, in order to expose a webDAV filesystem to other clients at a later stage. The container package is a web application service package that is deployable to any web application server, but it is not used at the moment.

**Test**

The *Test* module contains all the classes used to conduct this measurement study for cloud storage providers. Furthermore it contains all the source code to analyze and visualize the generated test reports. In addition to that, this package contains the the simulation framework, which we will introduce at a later stage.

### Google Drive

Google Drive has one of the most RESTful APIs from the analyzed cloud storage providers, every accessible information is a resource and has an assigned predictable URI. The API is protected against unauthorized access via an OAuth 2.0 authorization framework [10]. The API is documented[5] very well; Google even provides a software development kit (SDK) alongside many sample applications in different programming languages for interested developers. Nonetheless, the SDK had a limited use case for our application, because it contains a lot of classes and most of the provided features cannot be used. Furthermore, some of the class names were conflicting, at least partially, with the Java Software Development Kit's (JDK) native class names, and the whole SDK aggravated the debugging process of the source code.

In order to access the API a developer needs a client identification and a client secret, which can be requested directly from the *Google APIs console*[6]. The API has a courtesy limit from 10,000,000 requests per day.

### Dropbox

Dropbox provides a rather outdated SDK, which is not available through the central maven or any other artifact repositories. The API is well documented[7] and protected using the OAuth 1.0 [9] authorization protocol. Dropbox has recently been updated to support both OAuth 1.0 and OAuth 2.0. In order to access the API an *API Key* and an *API Secret* are needed, which can be obtained when registering an application.

### Box.com

Box.com's API is protected from unauthorized access using the OAuth 1.0, but they have recently released a new version of their API which also supports OAuth 2.0. In the future the support for the first version will be discontinued as it is preferred to use the newer version *v2*. There are also SDKs available for iOS, Android, Windows 8 and Java (solely for the second version of their API).

### SugarSync

The SugarSync's API is protected using a custom authorization algorithm. In order to access the API the application has to be registered to receive an *Access Key* and an *Access Key ID*. With these values it is possible to receive an access token, the user has to enter their username and password directly into the application. Admittedly, this could easily be used by a malicious application to store the user credentials and thus gain full access to the account. There are no SDKs available for SugarSync and some of the features are thinly documented.

[5] https://developers.google.com/drive/

[6] http://developers.google.com/console

[7] https://www.dropBox.com/developers/core/docs

#### Skydrive

Microsoft's Skydrive API is accessible by using OAuth 2.0 authorization for Microsoft Live Connect[8]. The developer has to register the application to receive the *client id* as well as the *client secret* which are required to access the OAuth endpoint. The scope transmitted while starting the authorization process has to contain the permission required by this application. The user has to review and grant this permission by entering her user credentials. Microsoft provides SDKs for the operating system Windows 8, Android, iOS and Windows Phone.

#### Ubuntu One

Ubuntu One's API is protected using a custom authentication combining a request to the single sign on framework and OAuth 1.0. The initial request is transferred over a secured channel using a SSL/TLS encryption, but it contains the user's credentials. In other words, the user has to enter his user credentials within the application, which obviously poses a security risk in light of the fact that malicious software could easily steal the user's credentials. Within this first request the application receives a *consumer key*, a *consumer secret*, and a *token secret*, which are used to start an OAuth 1.0 authentication process. The documentation for the API is not very comprehensive and the REST resources are not chosen conveniently. Ubuntu One has recently released a SDK, that is still in beta, for the Ubuntu operating system.

### 4.2.4 Test Setup and Results

We started the measurement study on January 18th, 2013, on a virtual private server within the HostEurope[9] network and a synchronous Internet connection of 100 MBit/s. This Internet connection does not reflect the typical Internet connection of an average end user in Germany, but we wanted to test the performance of the cloud storage providers and decided to exploit the maximum bandwidth possible. The measurement is scheduled to start eight times a day: beginning at midnight it recurs every three hours until 9pm. The initial setup had seven configurations, with each configuration representing a file size transferred to the servers of the providers. The amount of files in this configuration file were generated and filled with random data which was then uploaded to the storage provider in order to be downloaded and eventually deleted. This step was repeated for every storage provider. We then switched to the next configuration file (file size). This initial configuration (cf. Table 4.1a) had the purpose of evaluating how long the measurement takes and how many files can be uploaded to the storage providers within this three hour time frame.

Since one run for the seven configuration files took about 45 minutes, we increased the amount of files transferred to the storage providers on February 8th, and even added a second server with this new configuration (cf. Table 4.1b). The server resided within the network of the P2P department from the TU Darmstadt and was also connected to the Internet using a synchronous 100 MBit/s connection. This configuration took approximately two and a half hours and fitted our time frame perfectly.

On March 20th, two other servers within the network of the P2P department joined the measurement study. These servers were also connected to the Internet using a 100 MBit/s connection. Eventually we switched the configuration again (cf. Table 4.1c) in order to reflect the initial findings from the file size distribution study, which revealed that most of the users store smaller files in the cloud. We increased the amount of the smaller files (less than 1 MB) and removed files bigger than 10 MB. Although the transferred total file size dropped from 271 MB to 67 MB, this test also took approximately two and a half hours, because the amount of requests to be executed was increased.

The measurement study ended on June 9th for three of the four servers. It ran a total of three months and 22 days. We continued to gather data on one server residing in the network from the TU Darmstadt. This server extended the measurement study until August 15th in order to observe some long term issues regarding the providers.

The measurement study revealed that no provider has different transfer speeds at different times of the day. Figure 4.3 shows this result for Dropbox. The durations are nearly constant throughout the different time intervals measured during the day.

Regarding the download speed we discovered SugarSync and Dropbox to be the fastest providers for files smaller than 10 KB. For the bigger files, we found that Box.com (which only has average download speeds for smaller files), Google Drive and Dropbox to be the most advisable providers. SugarSync, one of the fastest providers for small files, is clearly one of the slowest providers for bigger files. Ubuntu One and Skydrive provided the slowest transfer speed across all file sizes (cf. Figure 4.4).

Regarding the upload speed we discovered Dropbox to be the fastest provider, even though it possessed decreasing rates down to an average value for bigger files. Similar to the download rates, SugarSync has an average rate for smaller files which reduces to the lowest transfer rate for bigger files. Skydrive and Ubuntu One provide again the worst

[8] http://msdn.microsoft.com/de-de/library/live/ff621314.aspx

[9] http://www.hosteurope.de/

| Amount of Files | File Size |
|---|---|
| 25 | 1 B |
| 25 | 10 B |
| 25 | 100 B |
| 15 | 1 KB |
| 5 | 10 MB |
| 1 | 50 MB |
| 1 | 100 MB |
| 97 | 200.017 MB |

**(a)** Initial configuration, started on January, 18th

| Amount of Files | File Size |
|---|---|
| 50 | 1 B |
| 50 | 10 B |
| 50 | 100 B |
| 25 | 1 KB |
| 20 | 10 KB |
| 10 | 100 KB |
| 20 | 1 MB |
| 10 | 10 MB |
| 1 | 50 MB |
| 1 | 100 MB |
| 237 | 271.23 MB |

**(b)** Second configuration, started on February, 8th

| Amount of Files | File Size |
|---|---|
| 75 | 1 B |
| 50 | 10 B |
| 50 | 100 B |
| 25 | 1 KB |
| 25 | 10 KB |
| 25 | 100 KB |
| 15 | 1 MB |
| 5 | 10 MB |
| 270 | 67.78 MB |

**(c)** Third configuration, started on March, 20th

**Table 4.1.:** Measurement Configurations

bandwidth. Ubuntu One, however, can increase the rate for files bigger than 1 MB to a more desirable average value (cf. Figure 4.5).

### Dropbox

Dropbox is one of the most stable providers in our measurement study. The duration required to upload and download the corresponding files was constant throughout the examination period; there was also no sign of a possible rate-limiting function. Dropbox provides a little bit more bandwidth when downloading files in comparison with the upload bandwidth. Overall Dropbox is one of the fastest providers.

### Google Drive

Google Drive is also one of the more stable providers in our measurement study. Regarding the transfer rate, Google Drive was always within the top three providers (especially the upload rate for 100 MB files is outstanding, it is with 6205 KB/s over six times faster than the third fastest provider Ubuntu One). Additionally we achieved a download rate for 50 MB files of 8187 KB/s, which is nearly the maximum bandwidth a 100 MBit/s Internet connection can provide. Google has a courtesy rate limit of 10 million free requests per day[10], but we were not able to reach this boundary in our study. Within our measurement study we generated approximately 30,000 requests a day for 270 files, eight times a day.

Although we could show quite good average transfer rates, we had some deviations for the two biggest file sizes (1 MB and 10 MB). Since the middle of April we were able to see some higher variances for the duration of the requests in comparison to the other file sizes, something which influenced the averages noticeably. Figure 4.6 shows the duration for the requests of the file size 1 MB, and Figure 4.7 shows the download duration for the file size of 100 Bytes. This influence disappeared on July 9th, when we ended the study for three of the four servers. These abnormalities could have been shown for the download, as well as the upload of the according file sizes, but not for any of the other sizes. This phenomenon could be a hint of some kind of different rate-limiting, in addition to the API request limit.

### Ubuntu One

In contrast to Dropbox or Google Drive, Ubuntu One is one of the unstable providers. A lot of requests failed and it was rarely possible to upload more than 10 MB using their REST API. The download transfer rates were generally one of the slowest. Disregarding file sizes greater than 1 MB (where Ubuntu One's transfer rates lie around the average) the upload rates are also very poor (cf. Figure 4.4). In addition, the duration to transfer the respective files increased throughout the whole study, a finding clearly supported by Figure 4.8. This could indicate a possible rate-limiting. This theory is supported by the frequent accumulations of points around the seven second mark, which means that some of the requests that, under normal circumstances, take about one or two seconds, now require seven seconds. This artificial delay may

[10] https://code.google.com/apis/console

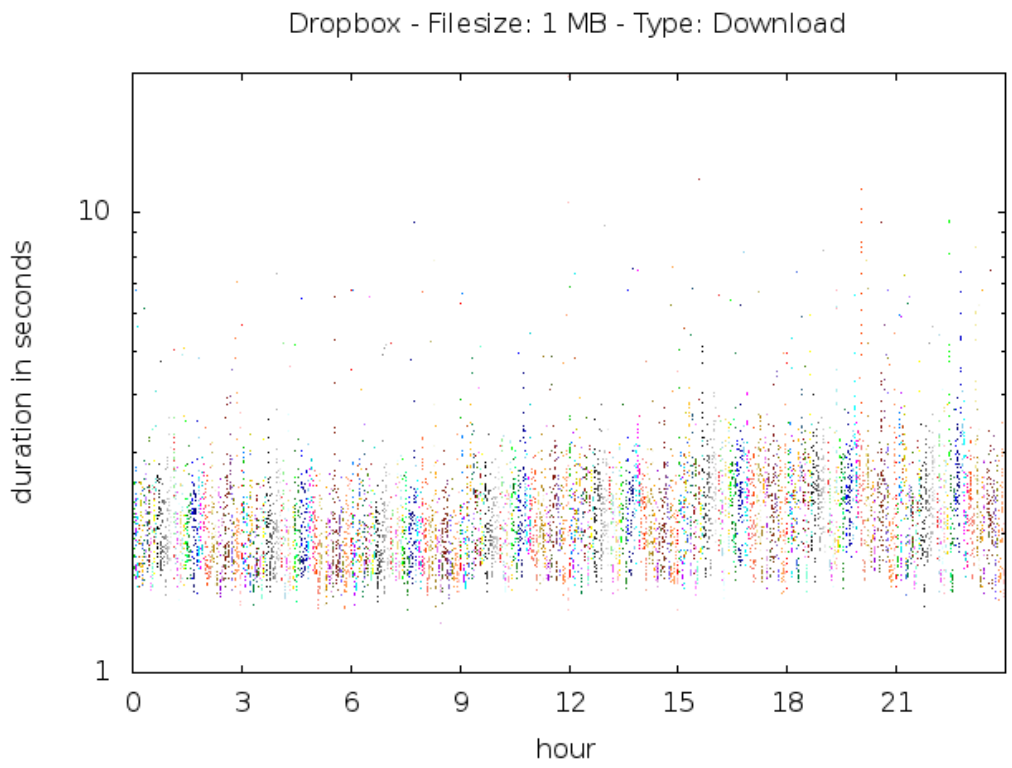



**Figure 4.3.:** Duration to download 1 MB files from provider Dropbox, 14 values between 31s and 53s have been cut off, which is 0.1439 % of all values.

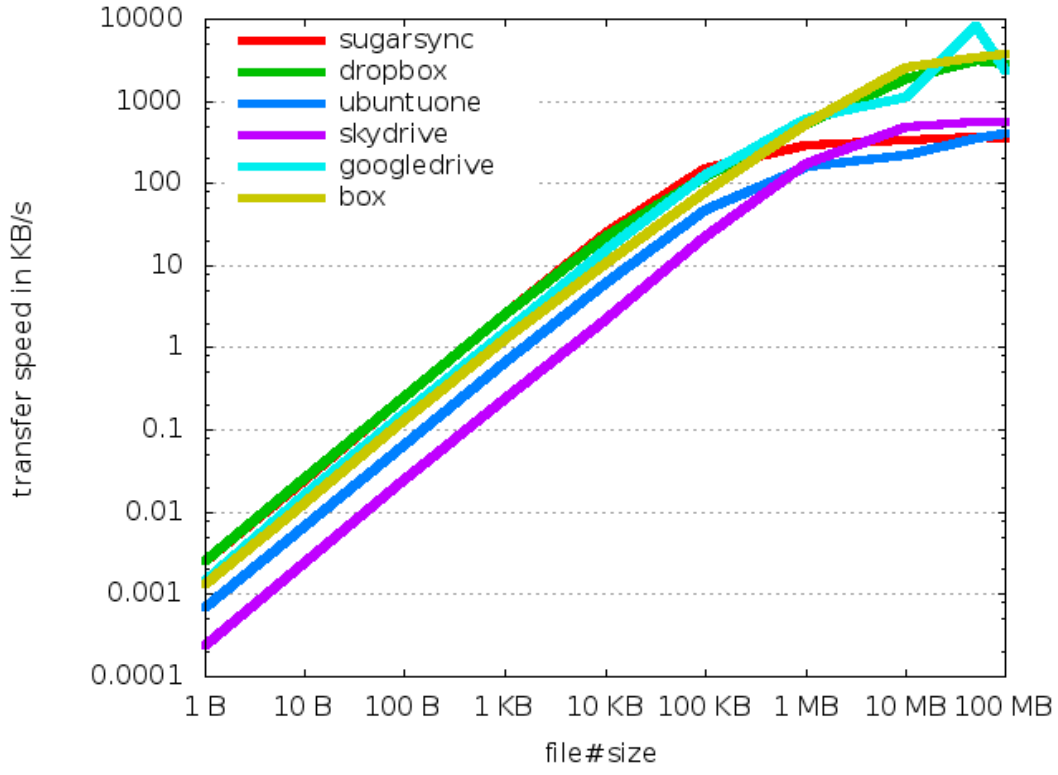



**Figure 4.4.:** Download Speeds for different file sizes.

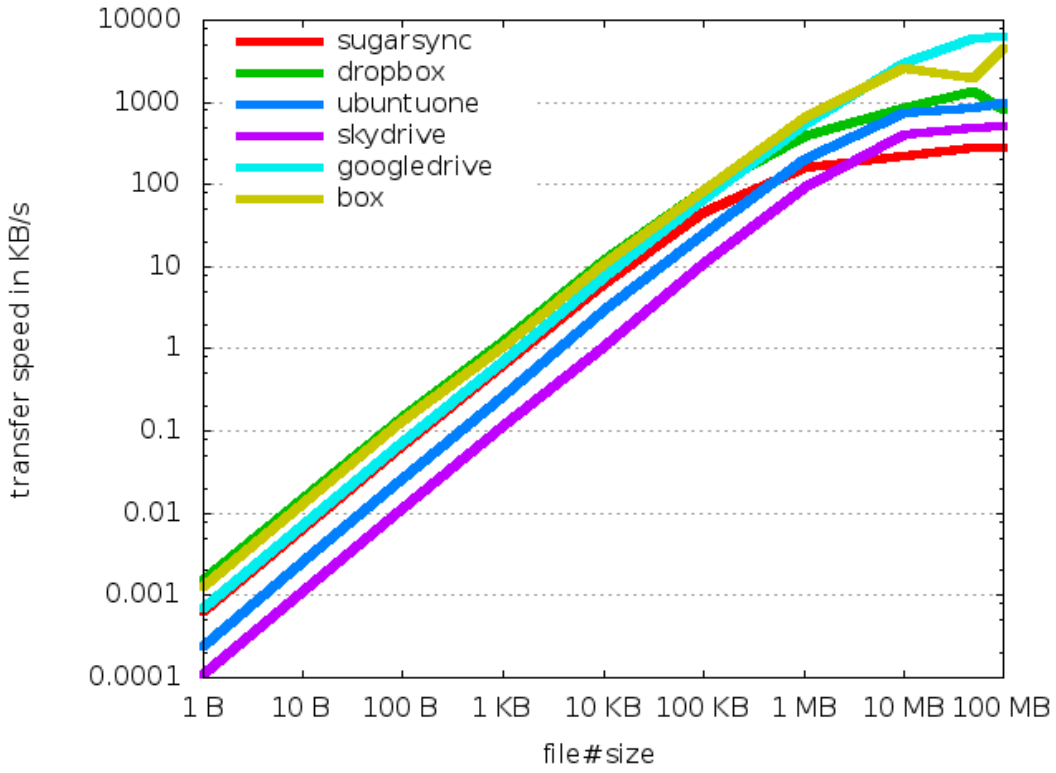



**Figure 4.5.:** Upload speeds for different file sizes.

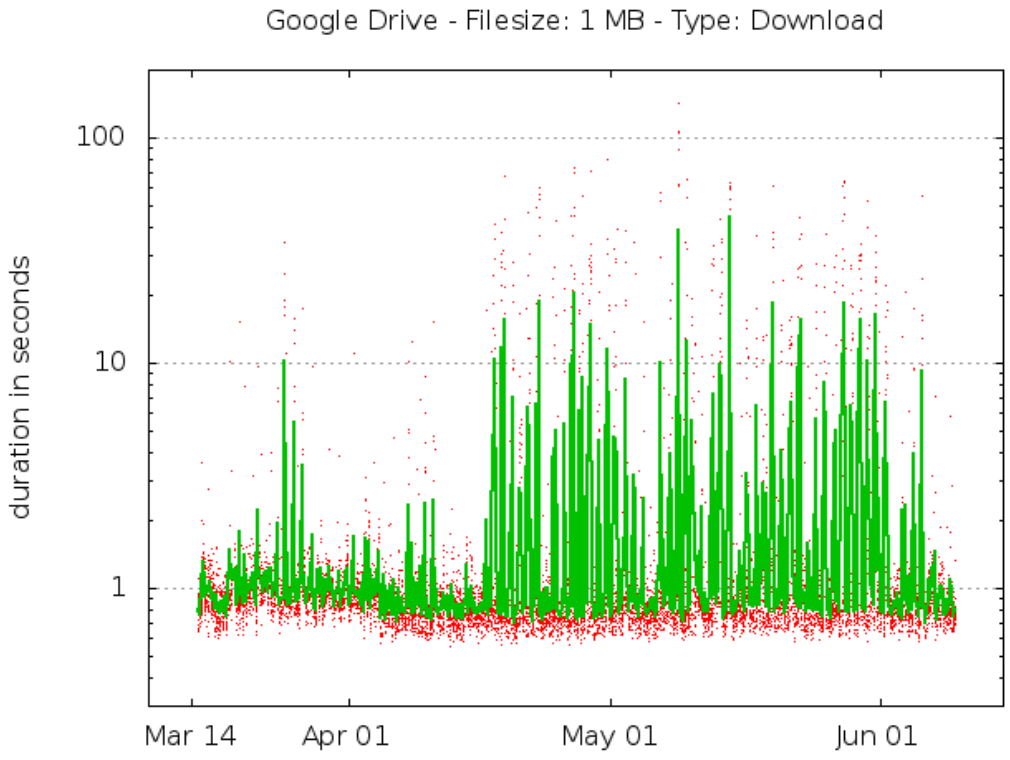



**Figure 4.6.:** Download durations for 1 MB from Google Drive. The red dots mark a successful transfer, the green line depicts the average value for each day.

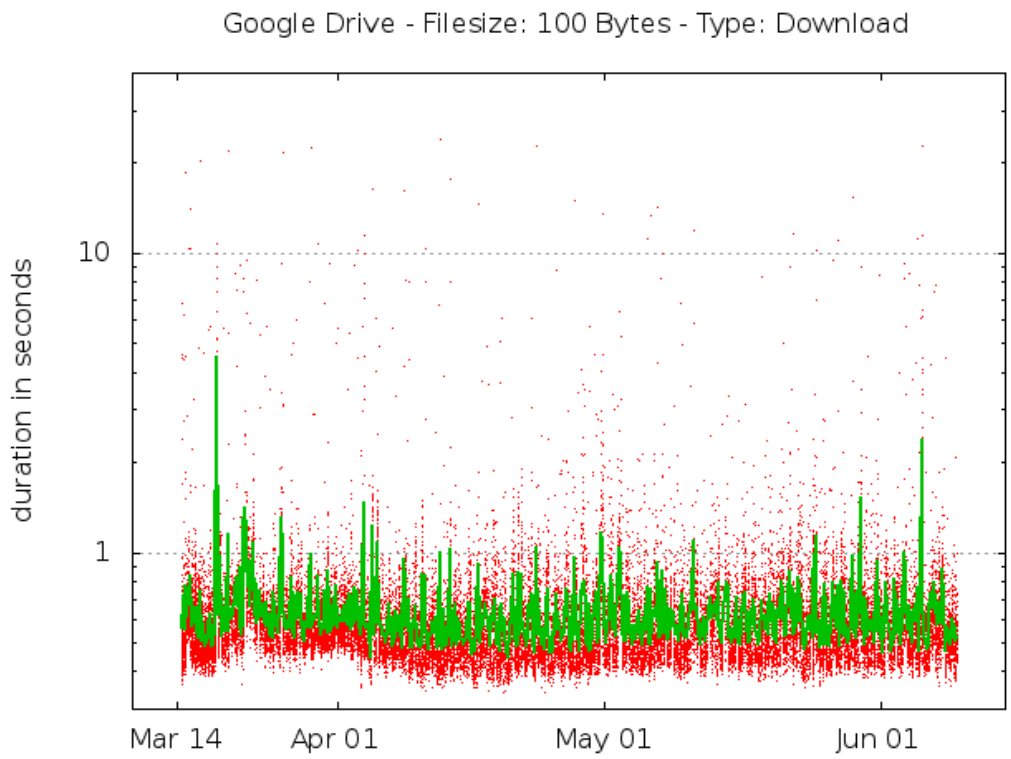



**Figure 4.7.:** Download durations for 100 B from Google Drive. The red dots mark a successful transfer, the green line depicts the average value for each day.

be caused by throttling of the bandwidth since the average transfer rate of the day seems to be bouncing between these two values (cf. Figure 4.9). This behaviour was also observed with all other file sizes, but not with any other tested cloud storage providers. As an interesting counter example, see Figure 4.7. The Figure 4.10 combined the results of all the servers and the described effect was mitigated when all servers conducted the measurement during March and June.

### Box.com

Box.com is one of the faster providers in terms of transfer rates. The download rate for 100 MB is with 3854 KB/s the fastest among all other providers. In terms of stability, Box.com is also one of the better providers, but shows similar behavior to Ubuntu One when it comes to accumulations of points (cf. Figure 4.11). This conspicuousness can be observed in all file sizes transferred to Box.com, most likely an indication for bandwidth throttling. This indication is confirmed by Box.com on their developer reference[11]. Box.com says that they will return an "HTTP/1.1 503 Service Unavailable" response when hitting the rate limit (we never received this status code during our measurement study).

### Skydrive

Skydrive is by far the slowest provider we have evaluated. We were never able to receive more than 575 KB/s. In addition Skydrive seems to throttle the bandwidth, which a Microsoft employee has confirmed with a post in Skydrive developers forum[12]. They do not publish the exact numbers and reserve the right to change them at any time. When hitting the rate-limit, Skydrive returns the HTTP status code 420 as response to inform the application that the rate-limit has been reached. We received this status code for the first time after the switch to the third configuration (cf. Table 4.1c), but then the very same code showed up on nearly every measurement run after 150 upload requests. In order to determine whether the rate limit is based on the user account or on the originating IP address, we have exchanged the authentication details from two of the servers on April 23rd. Given the case, that the rate-limiting is IP based, we should not be able to see a difference regarding the intensity of the throttling. When the throttling is bound to the

[11] http://developers.box.net/w/page/51585753/Rate%20Limiting

[12] http://social.msdn.microsoft.com/Forums/live/en-US/f10658d6-e995-41c0-9409-04de97d2b792/skydrive-api-limits

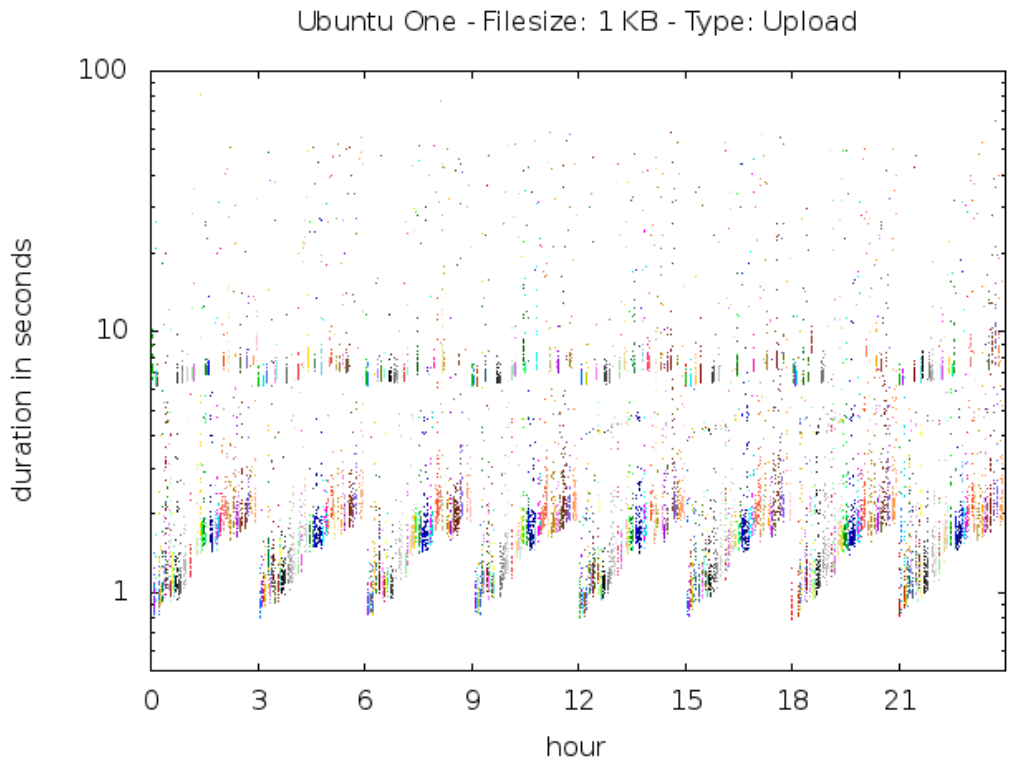



**Figure 4.8.:** Upload durations for 1 KB to Ubuntu One sorted by hour. Each dot marks a successful data transfer.

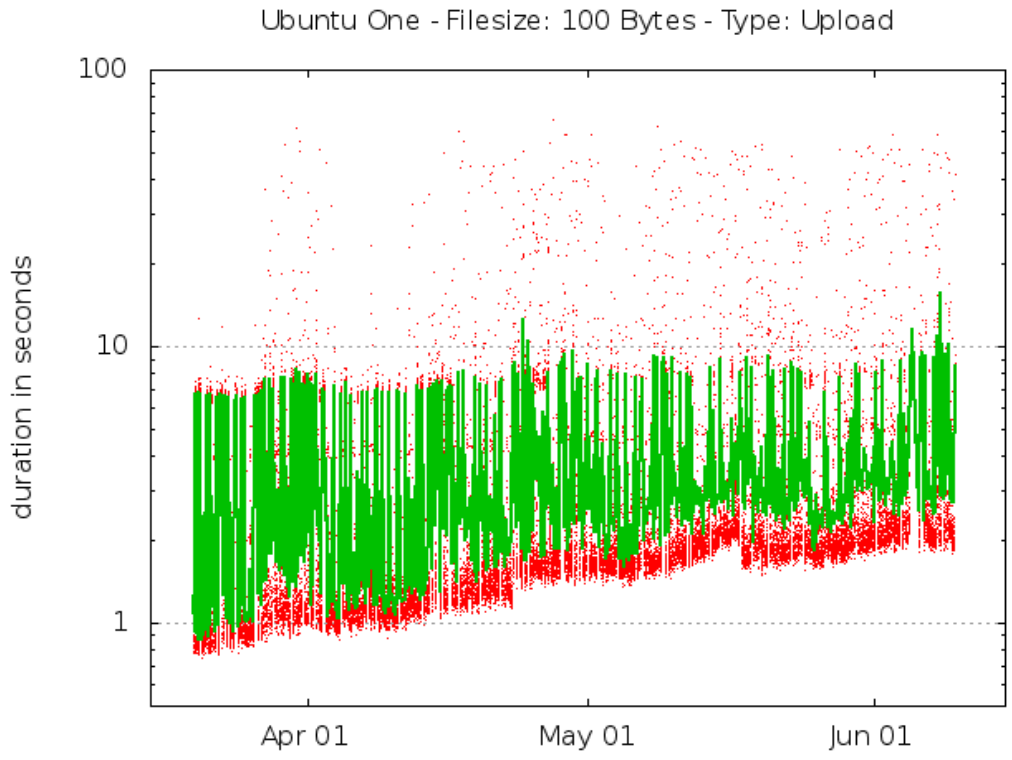



**Figure 4.9.:** Upload durations for 100 B to Ubuntu One. The red dots mark a successful transfer, the green line depicts the average value for each day.

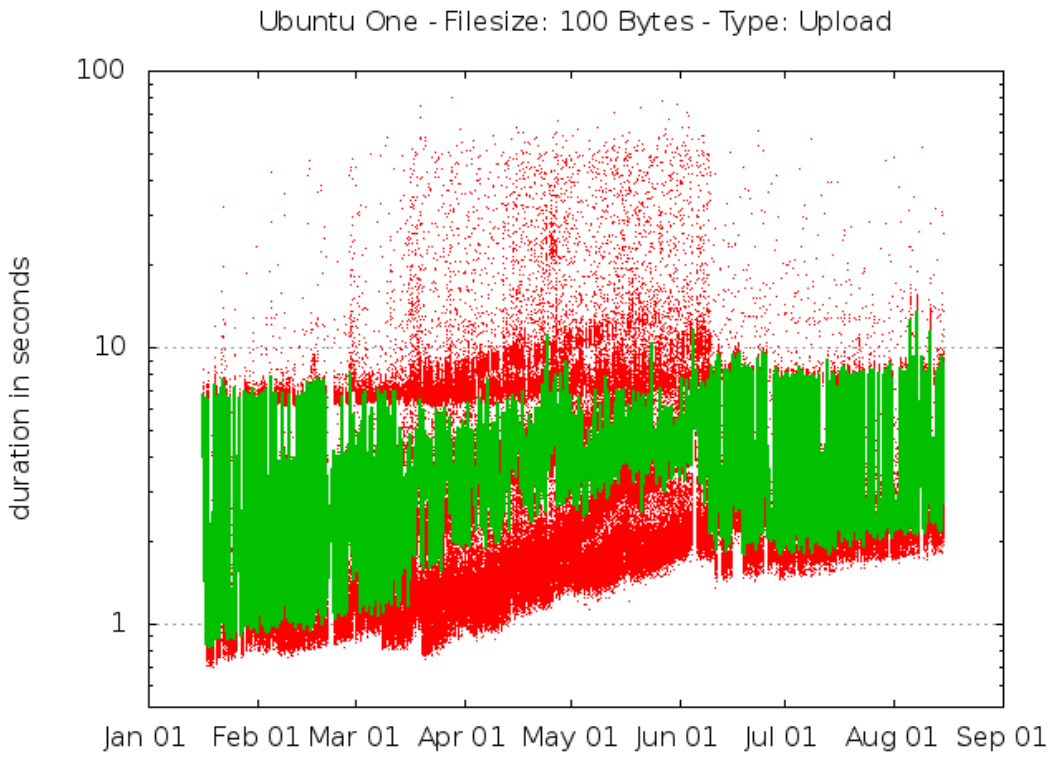



**Figure 4.10.:** Upload durations for 100 B to Ubuntu One from all servers combined. The red dots mark a successful transfer, the green line depicts the average value for each day.

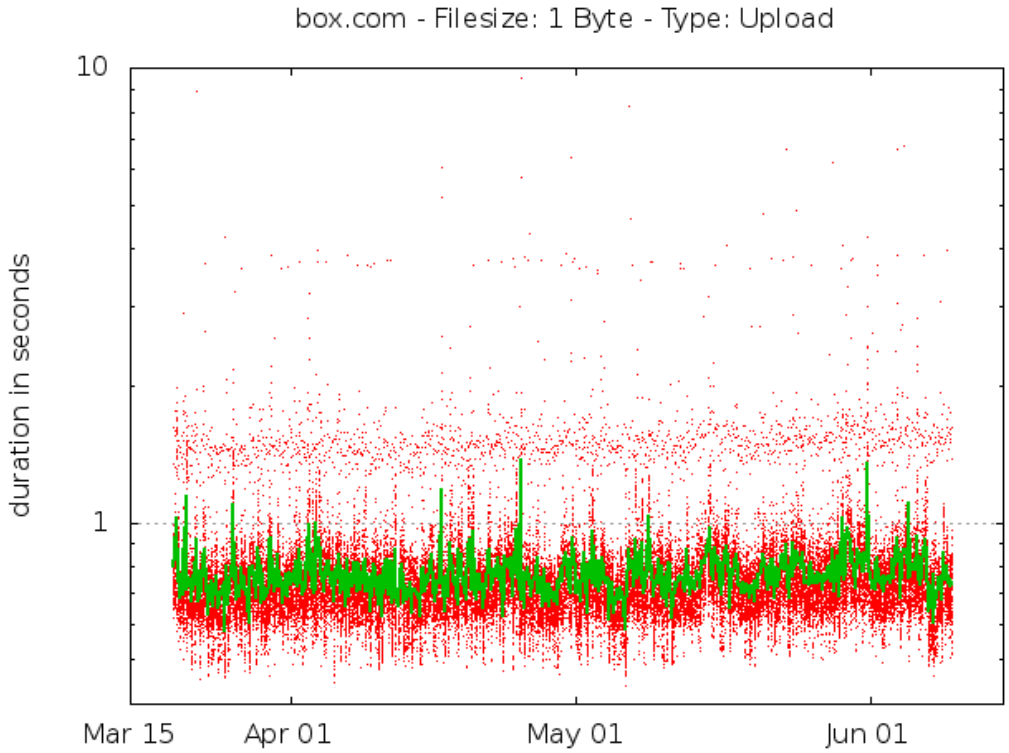
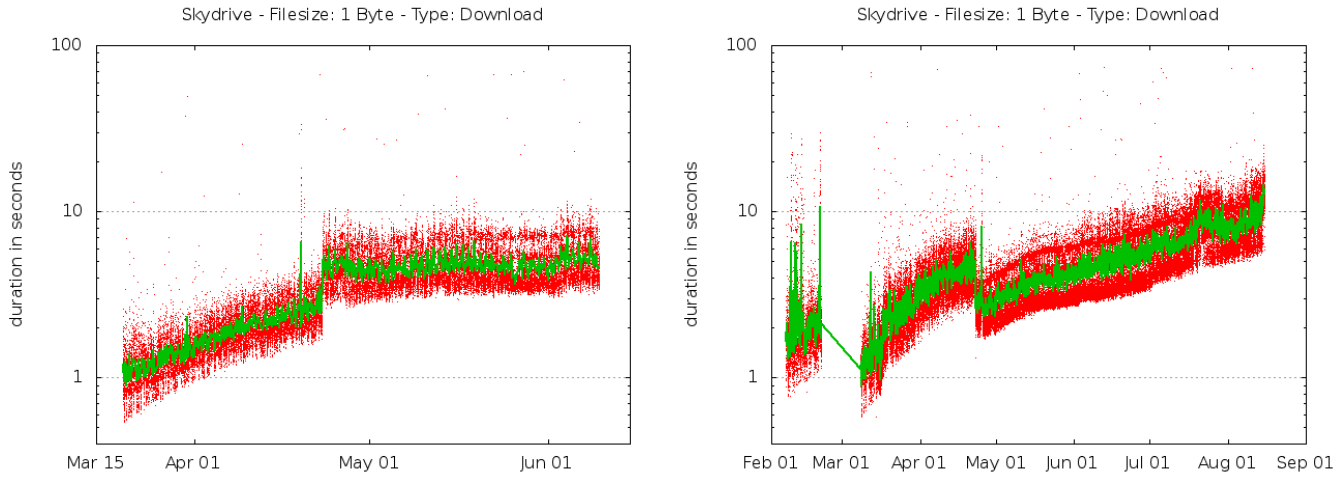



**Figure 4.11.:** Upload durations for 100 B to Box.com

user account information, we might see a change after the exchange of the authentication credentials, i.e., given the test combination $(f, r, n_1, u_1)$ running on the first server, and $(f, r, n_2, u_2)$ running on the other server, we obtain the combinations $(f, r, n_1, u_2)$ and $(f, r, n_2, u_1)$ by exchanging the user credentials. The two figures pictured in Figure 4.12 show the corresponding server during the measurement study. It is clearly to see, that intensity of the throttling increased on one of the server (cf. Figure 4.12a), and decreased on the other server (cf. Figure 4.12b). This is clear evidence, that the throttling of the bandwidth is based on the user accounts.




**(a)** Download durations for 1 Byte from Skydrive with authentication details exchanged on April, 23rd

**(b)** Durations for the other server, whose authentication credentials have been exchanged

**Figure 4.12.:** Exchanged user credentials on Skydrive.

### SugarSync

Sugarsync is a stable provider with higher variance of the durations in comparison to Dropbox or Google Drive (cf. Figure 4.13). The download transfer rate is average, but the upload rate is one of the fastest of the measured providers. Furthermore, the transfer durations did not increase during our measurement study, which seems to suggest an absence of rate-limiting. In terms of stability we have noticed that there has been a smaller service interruption between May 1st and May 15th. During this time all the requests executed to their API consumed more time than they used to before this period (cf. Figure 4.13).

### Summary

Although we tested a lot of very small requests, where the combined HTTP and IP header are considerably larger than the actual payload, the conducted measurement study showed decisively that there are some differences between the distinct providers. Most of the providers have some kind of rate-limiting functions installed: Box.com, Google Drive, and Skydrive note this characteristic on their website (although we were not able to hit the rate limit on Google Drive). Ubuntu One has no information regarding a rate-limit on their website, but the measurement indicates that there is in fact a rate-limiting function installed. Dropbox and Sugarsync were the providers for which we could find no indication of a rate limit.

With our measurement study we could also show that the cloud storage providers have different transfer bit rates for different file sizes. We are furthermore able to demonstrate that it is possible to improve a combined system's transfer rates by selecting a provider intelligently. Box.com was the only provider with similar upload and download rates. For all other providers the download rate is higher than the upload rate. Dropbox and Google Drive were the most stable and fastest providers in our study, Ubuntu One the most unstable (with many aborted transmissions) and Skydrive the slowest for most of the file sizes.

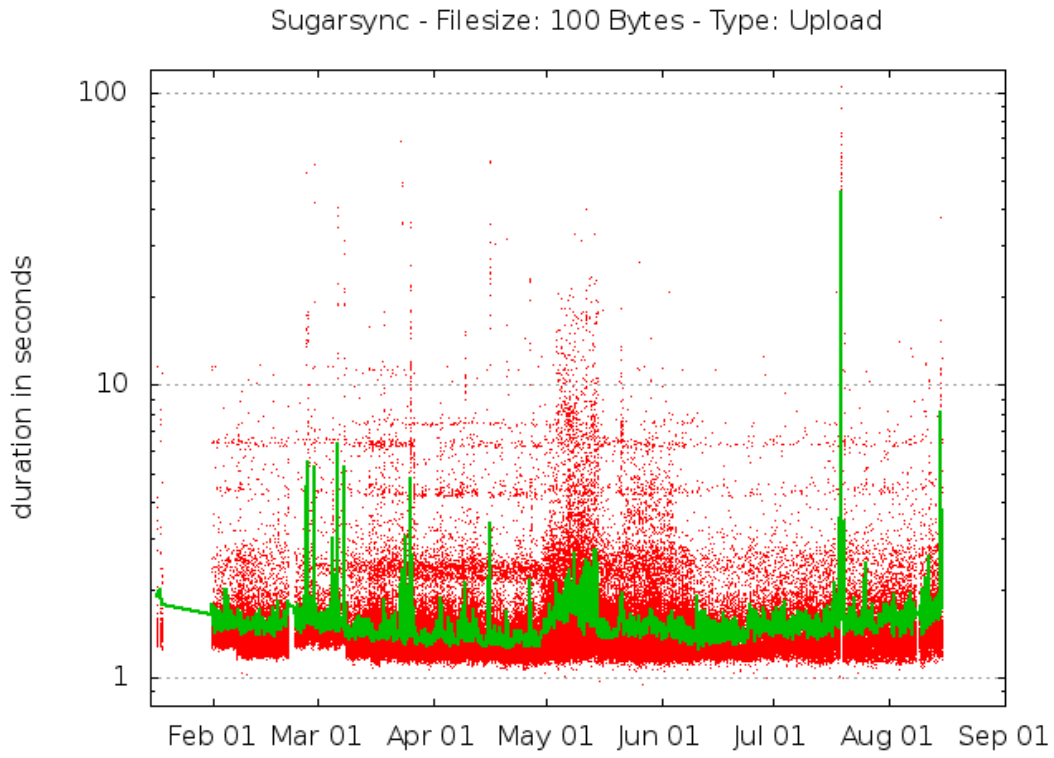



**Figure 4.13.:** Upload durations for 100 B to Sugarsync

# 5 Simulation

In the previous Chapter we analyzed different cloud storage providers in order to discover how each handles different file sizes as well as how well the data transfer rates are. In addition to that we conducted a study to reveal how large the files which are typically stored on the provider are, and how the file size is distributed. In Chapter 3 we discussed the different metrics that can be used to improve the providers performance (or the reliability of the data access). This information can be used now to develop distinct strategies in order to properly assess how the upload and download processes can be improved when using a combined cloud storage system. In order to compare the aforementioned strategies we built a framework that is able to simulate different scenarios with different configurations. These scenarios can vary the upload or download strategies (or amount of files that will be synchronized to the different providers) which enables us to compare the strategies with one another.

## 5.1 Simulation Model

In order to build a simulation framework that is capable of simulating the user's interaction with a cloud storage provider, it is essential to have a solid simulation model. In this Section we describe the core model and we propose for our simulation framework. After all, what we are after is a model which has the ability to simulate different scenarios in order to improve the upload and download processes for files of differing sizes.

### MockedFile

A *MockedFile* represents a file which can be stored on, or retrieved by, a cloud storage provider. It consists of a file name and size (which is represented in bytes), a directory and a provider. The directory is a *DirectoryTree* (cf. 5.1) and represents the folder in which the MockedFile is stored. The provider is an optional value; a reference to the corresponding *Provider* (cf. 5.1) where the file is stored. This reference is most definitely required, given that we need to know the file's origin (in terms of provider identity). A MockedFile stores neither content nor data. It is merely a mock-up of an actual file.

### Provider

In our simulation framework a *Provider* represents a cloud storage provider. It consists of the provider's name, the size of the storage offered by this vendor and a list of MockedFiles stored on this Provider. All tasks that can be performed on a real cloud storage provider can be performed on this created instance of a Provider. It is possible to upload files, download file, delete files, and update files. The transfer time depends on the size of the MockedFile (as is the case with a real provider); the bigger the file the more time required to transmit it. It also allows to search for a MockedFile by its file name, as well as list retrieval of all MockedFiles currently stored on the Provider. Furthermore, it is possible to retrieve information about the Provider and its current state. We can easily discover the remaining free capacity, the currently used storage in percent or even a statistic about the size of individual MockedFiles currently stored on the Provider.

### DirectoryTree

A *DirectoryTree* is a tree-like, recursive data structure that represents a filesystem tree. Each node incorporates a folder in the filesystem. An instance of the DirectoryTree has a name, a parent DirectoryTree, a list of child DirectoryTrees and a list of MockedFiles stored in the appropriate folder. Similar to a normal filesystem, it is possible to add a MockedFile or DirectoryTree as a subdirectory. Furthermore, it is possible to get the depth of this DirectoryTree (that is, the maximum amount of child levels the DirectoryTree contains) as well the count, the recursive amount of subdirectories stored in the DirectoryTree.

### FileDistribution

The *FileDistribution* is a simple algorithm that distributes objects, in particular MockedFiles, over the referenced DirectoryTree using a predefined distribution strategy called *DistributionRule*. The FileDistribution classifies each node of the tree as valid or invalid according to the current DistributionRule. Valid DistributionRules are **random**, **near**, and **far**. The random DistributionRule distributes objects randomly over the DirectoryTree. When using the DistributionRule near, the object are spread into folders within the configured distance of a randomly selected directory. The far DistributionRule distributes the objects in two distinct folders, that have at least a distance of the configured value.

### Batch

A *Batch* represents changes that should be applied to the filesystem. Therefore, it contains a list of *BatchEvents* that consist of a MockedFile and a *BatchEventType*. A BatchEventType is an enumeration of the type of changes that can

be applied to the remote filesystem (e.g., create a MockedFile, delete a MockedFile, retrieve a MockedFile or update a MockedFile). In addition to this, a Batch has an UploadStrategy (cf. 5.1 UploadStrategy), a download strategy and a FileDistribution (cf. 5.1 FileDistribution). When downloading multiple files, the download strategy decides the order that files are downloaded in; this may happen, for instance, in a per folder order or in a random selection.

#### UploadStrategy

An *UploadStrategy* decides to which Provider a given MockedFile should be uploaded. In order to make this decision, we pass the UploadStrategy the current state, the Batch containing the MockedFiles to be changed and the list of the Providers meant to handle these files. Given the previously discussed attributes in Chapter 3, the strategy decides which Provider is most suitable to handle the current MockedFile.

#### Change

*Change* is an abstract class that represents an actual change which can be applied to the Provider. It contains the time interval this change required to finish, the MockedFile and the Provider on which this operation is executed. Each storage provider operation (i.e., upload, download, delete and update) implements this class. An *UploadStrategy* converts all BatchEvents into Changes (i.e., it finds the most suitable provider for the MockedFile, it creates the Change object and it sets the Provider and the MockedFile). When the Change is applied at a later time, the MockedFile is actually uploaded to the provider and the time attribute of the Change object is set.

## 5.2 Evaluation Scenarios

In order to test multiple different configurations, we introduce *Scenarios* into our simulation framework. A Scenario consists of multiple core parameters that define a simulation. These parameters are a list of changes that will create Batch: the DistributionRule, the UploadStrategy, the filesystem tree depth, the amount of folder tested with this simulation, the list of different download strategies and the amount of iterations, this simulation should be repeated.

The changes described by a Scenario define how many Batches are created for this simulation run. Every entry of this list represents a distinct Batch that will be executed to the Provider. A Batch is here specified by an operation that can be created, deleted or updated, and the amount of affected files. The change “100,create” would create one hundred MockedFiles in a single Batch.

The tree depth, the folder amount and the DistributionRule are the three key indicators that determine how a filesystem tree is created. The DistributionRule values (random, near, and far) define how the MockedFiles should be distributed in the DirectoryTree; the folder amount determines how many folders should be created in our filesystem; the tree depth states how deep the created DirectoryTree should be. A tree depth of two means there can only be hierarchy level of two, i.e., one directory with one subdirectory.

Similarly, the download parameter defines how files should be downloaded. In contrast to the UploadStrategy it is not possible to configure the amount of files to download. When the download parameter is given, all files will be downloaded using the specified strategy. When using the download strategy *random* or *all* it is mandatory to specify the amount of files that should be downloaded using a single Batch. However, when more MockedFiles are stored on the provider, more Batches will be generated until all files have been downloaded.

The UploadStrategy is the most important parameter for the simulations, since it defines which UploadStrategy should be used when uploading the different files. This parameter has a noticeable effect on the whole simulation since it describes how files should be distributed over the different Providers.

We have created different Scenarios for each of the following UploadStrategies. In Section 5.4 we describe and discuss these Scenarios in detail.

#### MinTimeStrategy

The *MinTimeStrategy* is one of the more sophisticated strategies. It minimizes the upload time for the Batch by selecting the Provider with currently the least transfer time. In order to accomplish this it stores the transfer time for each Provider, and with every new MockedFile it determines the time it would take to upload the file to each Provider. This upload time is added to the current stored upload time. Now the provider with the smallest transfer time which can store the file is selected. The transfer time that has been added to the other providers is subtracted again.

This strategy has the advantage, that it does not matter how much files were already uploaded, or how big these files are. The only metric that is important is the time it takes to upload the files. Providers that have fast transmission rates for big files, but are slower when uploading smaller files, are preferred for the bigger files. It holds the overall upload time for each Provider as low as possible.

#### RandomStrategy

The *RandomStrategy* selects a random Provider from the list of available Providers and checks whether this Provider has enough space to store the file. When the Provider has enough space, the strategy selects it. Otherwise it randomly chooses another Provider from the list.

#### SequentialStrategy

The *SequentialStrategy* uploads each file to the first Provider available with enough free space. The order of the Provider list varies and can be changed during runtime, otherwise the list is sorted by the insertion time into the list.

#### SequentialSizeAscStrategy and SequentialSizeDescStrategy

The SequentialSize strategies sort the list of Providers by the available capacity (ascending or descending). The first Provider which can store the MockedFile successfully is selected. When using the SequentialSizeAscStrategy, the Provider with the least amount of storage space is preferred. When using the SequentialSizeDescStrategy the Provider with the most storage space is selected.

#### SpecifiedOrderStrategy

The *SpecifiedOrderStrategy* reads the order of the Providers for each file size from an external configuration file and returns the first one with enough space. This UploadStrategy can be used to presort the Providers based on a given metric (e.g., the transfer speeds for each Provider, personal preference or the amount of offered storage capacity). The priority of the Providers can be configured per file size, which means that it is possible to define a different order for different file sizes. Using this strategy it is possible to always select the fastest Provider for a given file size.

#### UtilizationStrategy

The *UtilizationStrategy* uploads the given MockedFile to the provider, with the lowest utilization in percentage. In so doing, this strategy guarantees that all Providers are utilized equally during an upload(ing) process.

## 5.3 Implementation

A simulation can be started by calling the main method within the **Main** class file. The main method requires a valid Scenario configuration file as parameter, but a second parameter may also specify the output file for this simulation. In addition to this, a configuration properties file is expected within the classpath of the program. This configuration only holds a few values, including the possible file sizes, the probability for these files and the distance required for the DistributionRule (cf. DistributionRule 5.1). The application parses the Scenario properties file and iterates in a loop over the amount of performed iterations. For every iteration a new DirectoryTree is generated and passed together with the DistributionRule and the UploadStrategy to a new **BatchGenerator** instance. For each configured Batch in the Scenario configuration the BatchGenerator stores a recipe on how to create the predefined Batch. When a download strategy has been specified, that strategy will also be added to the BatchGenerator. With this information the main method creates a new **Simulation** instance, adding all hard coded Providers and beginning the simulation.

The background information required to model a cloud storage provider were gained by the conducted measurement study in Chapter 4. For each of the providers we have gathered the durations required when uploading or download data of the predefined file sizes. With this data in hand, we can plot a histogram (cf. Figure 5.1) in order to identify the distribution of the data points. Within our simulation we created 100 buckets, ordered the measured values by size and distributed them equally over the buckets. In order to determine a representative transfer duration, we select a random number between zero and 100, and calculated the average value of the bucket with the randomly selected number.

The simulation starts by calling the getCurrentBatch method from the BatchGenerator. The generator will fetch the current recipe and create a corresponding Batch. If the current Batch is an upload Batch, the BatchGenerator will create the MockedFiles and distribute them directly over the created DirectoryTree. The file size of the MockedFiles is probabilistically chosen on the basis of our measurement study from Chapter 4.

The simulation retrieves the current Batch from the BatchGenerator and passes the current state of the simulation, viz., the information where each MockedFile is stored, the Batch and a list of the Providers, into the UploadStrategy's perform method. The UploadStrategy will decide, for each file in the Batch, which Provider the selected file should be uploaded to before returning a list of Changes (cf. paragraph Change in 5.1). The simulation will then process the Changes by calling the apply method on all of the Change objects. These Change objects will then upload the MockedFiles to the corresponding Provider. When the processing has finished, statistics about this Batch and the processing information will be printed on the output file. The concurrent data transfer process is simulated by cumulating the transfer times for each provider and determining the maximum of these values.

After all upload Batches have been processed, the simulation will start the download process according to the appropriate configuration. This procedure is similar to the upload process.

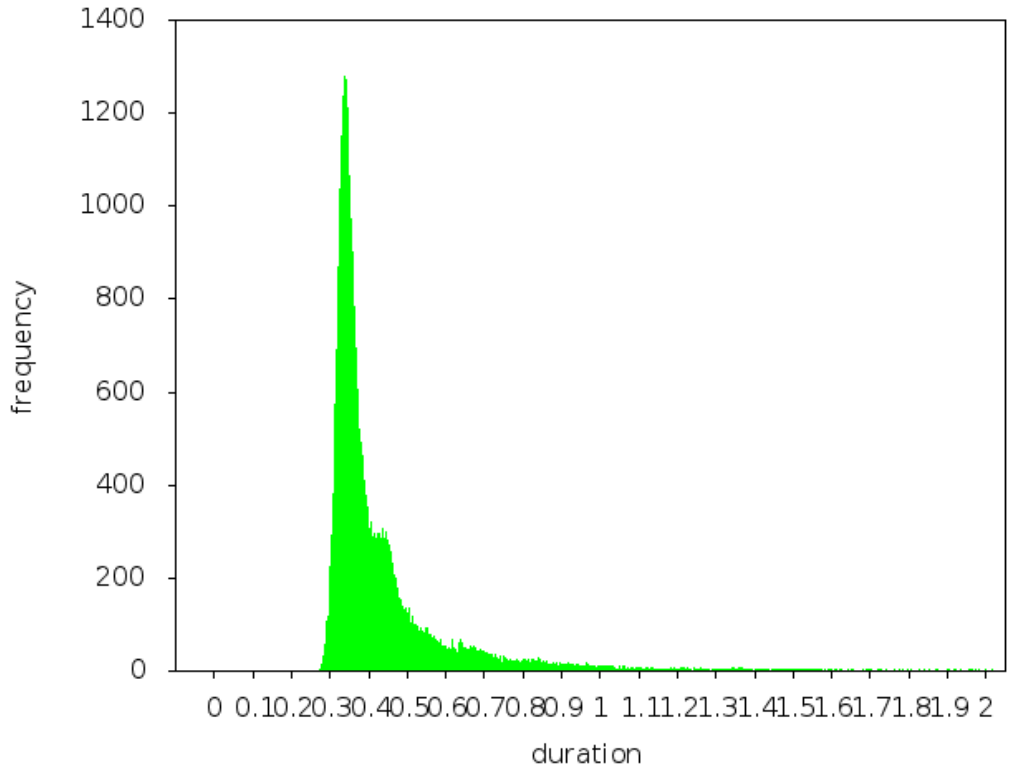



**Figure 5.1.:** Histogram for the durations required when downloading 10 KB from Dropbox; 630 (0.548 %) values were greater than two, and therefore cut-off.

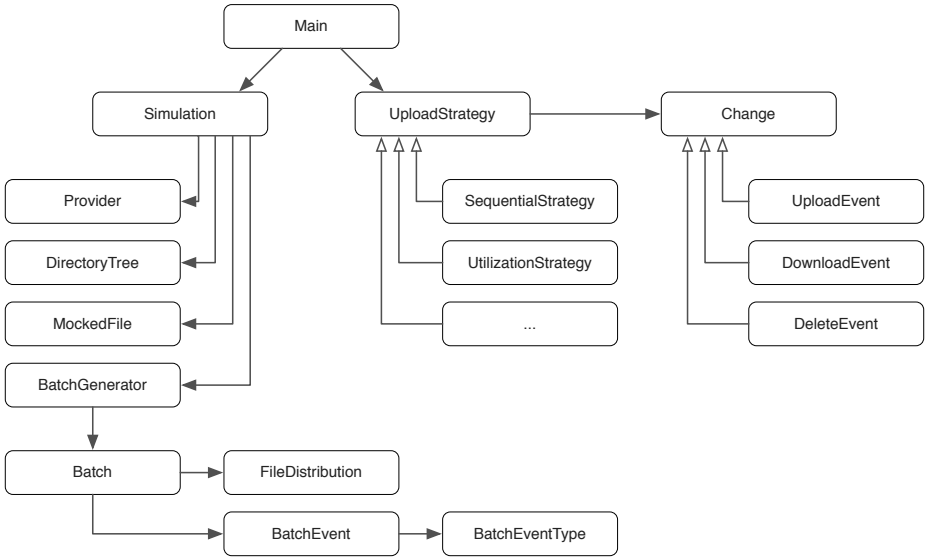



**Figure 5.2.:** A simple diagram illustrating components used by simulation framework.

## 5.4 Setup

We conducted our simulation in order to show that when building a combined cloud storage system the strategies which distribute the files across the cloud storage providers improve the performance of the system. For each UploadStrategy within each DistributionRule, we have created eleven different Scenarios. This results in 231 distinct Scenarios. Each Scenario uploads 30,000 files (cf. Table 5.1) to the different Providers and downloads all files seven times with different download strategies. Every Scenario is repeated 250 times.

| Scenario | Amount of Batches | Files per Batch | Amount of Files |
|---|---|---|---|
| Scenario 1 | 30,000 | 1 | 30,000 |
| Scenario 2 | 15,000 | 2 | 30,000 |
| Scenario 3 | 10,000 | 3 | 30,000 |
| Scenario 4 | 7,500 | 4 | 30,000 |
| Scenario 5 | 6,000 | 5 | 30,000 |
| Scenario 6 | 5,000 | 6 | 30,000 |
| Scenario 7 | 3,750 | 8 | 30,000 |
| Scenario 8 | 3,000 | 10 | 30,000 |
| Scenario 9 | 300 | 100 | 30,000 |
| Scenario 10 | 60 | 500 | 30,000 |
| Scenario 11 | 30 | 1000 | 30,000 |

**Table 5.1.:** Simulated upload scenarios.

The download is configured to perform the following: download all files in one Batch, create one Batch per folder and select the files randomly with a Batch size of 50, 100, 200, 500, and 1000.

The SpecifiedOrderStrategy is configured to upload the files according to the results acquired from the measurement study (cf. Table 5.2). The priorities based on the measurement are the following:

| File Size | Priority |
|---|---|
| 1 B | Dropbox, Box.com, Google Drive, Sugarsync, Ubuntu One, Skydrive |
| 10 B | Dropbox, Box.com, Google Drive, Sugarsync, Ubuntu One, Skydrive |
| 100 B | Dropbox, Box.com, Google Drive, Sugarsync, Ubuntu One, Skydrive |
| 1 KB | Dropbox, Box.com, Google Drive, Sugarsync, Ubuntu One, Skydrive |
| 10 KB | Dropbox, Box.com, Google Drive, Sugarsync, Ubuntu One, Skydrive |
| 100 KB | Box.com, Dropbox, Google Drive, Sugarsync, Ubuntu One, Skydrive |
| 1 MB | Box.com, Google Drive, Dropbox, Ubuntu One, Sugarsync, Skydrive |
| 10 MB | Google Drive, Box.com, Dropbox, Ubuntu One, Skydrive, Sugarsync |
| 50 MB | Google Drive, Box.com, Dropbox, Ubuntu One, Skydrive, Sugarsync |

**Table 5.2.:** Measured upload priorities for the SpecifiedOrderStrategy.

| File Size | Priority |
|---|---|
| 1 Box | Sugarsync, Dropbox, Google Drive, Box.com, Ubuntu One, Skydrive |
| 10 B | Sugarsync, Dropbox, Google Drive, Box.com, Ubuntu One, Skydrive |
| 100 B | Sugarsync, Dropbox, Google Drive, Box.com, Ubuntu One, Skydrive |
| 1 KB | Sugarsync, Dropbox, Google Drive, Box.com, Ubuntu One, Skydrive |
| 10 KB | Sugarsync, Dropbox, Google Drive, Box.com, Ubuntu One, Skydrive |
| 100 KB | Sugarsync, Google Drive, Dropbox, Box.com, Ubuntu One, Skydrive |
| 1 MB | Google Drive, Dropbox, Box.com, Sugarsync, Skydrive, Ubuntu One |
| 10 MB | Box.com, Dropbox, Google Drive, Skydrive, Sugarsync, Ubuntu One |
| 50 MB | Google Drive, Box.com, Dropbox, Skydrive, Sugarsync, Ubuntu One |

**Table 5.3.:** Measured download priorities

For our simulation we removed the file size of 100 MB, because this file size is very rarely uploaded to the storage providers. Furthermore, the amount of time required to upload the file is very high in comparison with other intervals (something which could easily have skewed our simulation results).

We chose three as a maximum tree depth for the DirectoryTree, since Agrawal et al. showed in [2] that 90 % of the user's folders had two or fewer subdirectories, and a total of thirty-five folders.

At the time when the simulation started, the providers offered the amount of free storage as illustrated in Table 1.1. These values have been adopted for the conducted simulation.

## 5.5 Simulation Results

Using the simulation it was possible to show that each of the seven UploadStrategies has a differing upload time for the individual scenarios. We were not able to show any correlation between the FileDistribution and the upload time. We expected this result, since the upload time is defined by the provider's transfer rate and not by the directory the files are stored in.

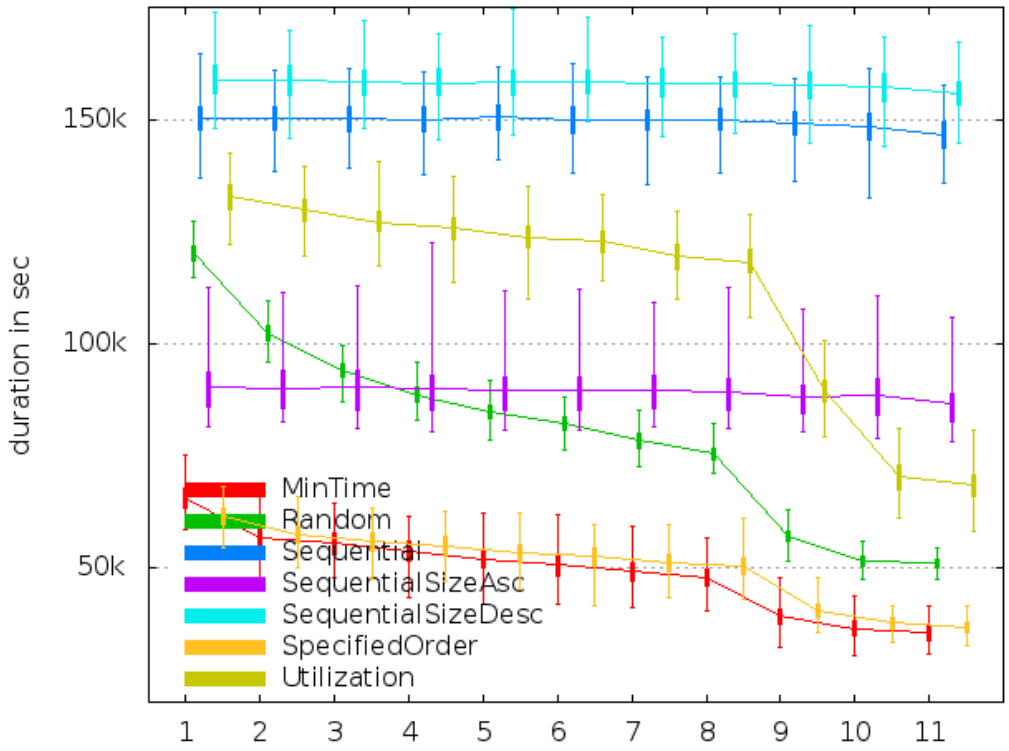


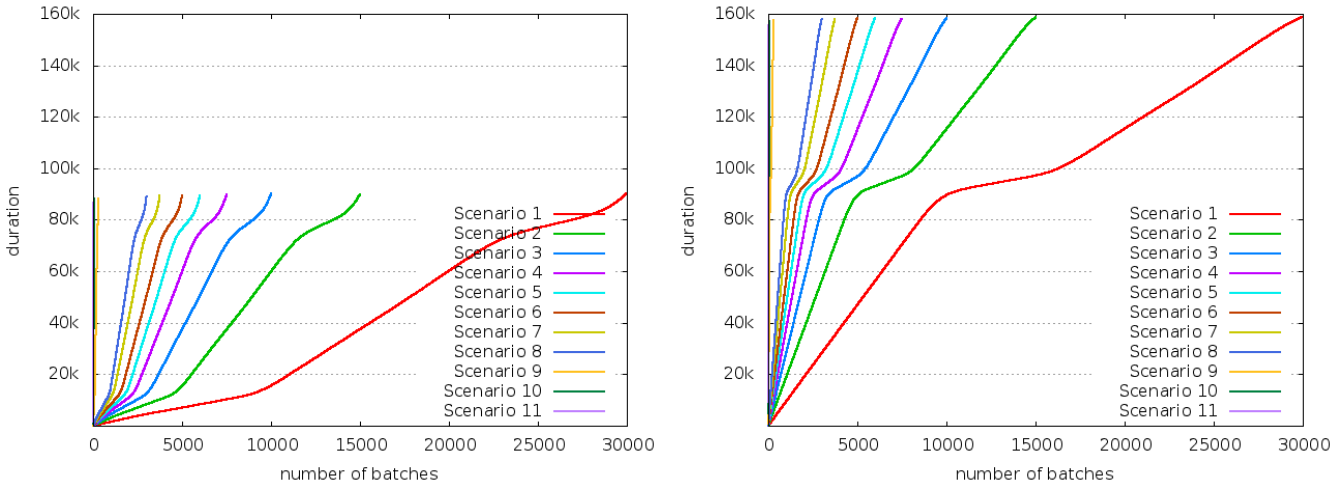

**Figure 5.3.:** Minimal, maximal, and lower and upper quartiles of the durations in seconds (x-axis) required by the different UploadStrategies for the distinct Scenarios (x-axis).

Figure 5.3 shows, that the SequentialStrategy (5.2) as well as the SequentialSizeDescStrategy (5.2) were the slowest strategies in our simulation, ranging between 145k and 160k seconds (roughly 1 day and 17 hours) for the different scenarios. The last two scenarios were a little bit faster than the previous. Both of the strategies, as well as the SequentialSizeAscStrategy (5.2), used a predefined order in which the providers are filled. The three strategies show all the same behavior throughout the scenarios. The transfer time of the data does not really change and is nearly constant throughout the simulation. Nonetheless, the SequentialSizeAscStrategy is nearly twice as fast as the SequentialSizeDescStrategy, which can be explained by that fact that the first selected provider is faster than the larger providers selected by the SequentialSizeDescStrategy.

The other strategies show different behavior. For all strategies, the upload time improves the more files a batch contains. The lower the amount of Batches the faster the upload process. This happens because the system can utilize concurrency when uploading files to the providers. For these strategies the UtilizationStrategy is the slowest, because this strategies' distribution method sends files across the entirety of the available providers (including the slower ones). The RandomStrategy selects a random provider and therefore also serves a slower provider, but in this occurrence there are more files within a Batch treated. This strategy can drastically improve the required upload time by transferring files concurrently.

In our simulation the MinTimeStrategy as well as the SpecifiedOrderStrategy were by far the fastest analyzed. These strategies directly take advantage of the measured performance results and therefore within the simulation configured transfer rates for the different providers. The required transfer time lies between 65,000 and 31,000 seconds, which equates to roughly 12 and-a-half-hours; roughly three times faster than the slowest strategy.

With respect to download strategies we also noticed that for the current combination of upload and download strategies the FileDistribution has no impact on the transfer time. Figure 5.5 shows for each UploadStrategy the minimal, maximal as well as lower and upper quartiles of the required duration for each download strategy. The download required the most time when the data was uploaded using the SequentialStrategy or the SequentialSizeDescStrategy. This




**(a)** Cumulative durations for the number of Batches of each Scenario transferred by the SequentialSizeAscStrategy.

**(b)** Cumulative durations for the number of Batches of each Scenario transferred by the SequentialSizeDescStrategy.

**Figure 5.4.:** Cumulative durations for the different Scenarios of the two SequentialSize strategies.

conclusion corresponds with the results gained for the upload strategies, wherein these two strategies have also been the slowest in terms of uploading data. The download transfer time is similar when the data has been uploaded by using the RandomStrategy or the SequentialSizeAscStrategy. When using the SpecifiedOrderStrategy to upload files the download is the fastest. In comparison with the upload transfer times, the SpecifiedOrderStrategy is better than the MinTimeStrategy. This may be due in part to the fact that the SpecifiedOrderStrategy always uses the fastest provider for a given file size, whereas the MinTimeStrategy makes use of slower providers whenever the fastest provider is too congested. Table 5.3 shows that when it comes to higher file sizes, the fastest provider when downloading files is often also one of the fastest provider when uploading files (cf. Table 5.2).

Regarding the distinct download strategies we can therefore conclude that the strategy that downloads every file stored on the remote provider in one Batch is continuously the slowest strategy in our simulation. This is followed by the “random 50” strategy, which selects 50 random files per Batch until every file has been downloaded. Interestingly, the “random 200” was invariably the fastest download strategy, followed by the “random 500” strategy (cf. 5.5).

For all the UploadStrategies we can safely state that the different upload Scenarios had no impact on the download (cf. Figure 5.6), except when using the MinTimeStrategy. When using this strategy we noticed that the required download time was worsening for scenarios with more files per upload Batch. This observation confirms our assumption that the MinTimeStrategy strategy is slower when downloading files, mainly because it transfers files to moderate fast providers when the initial provider is too congested. When using more files per upload Batch, as the latter scenarios do, the strategy has to upload more files in one run and, as a consequence, also has to distribute files to slower providers.

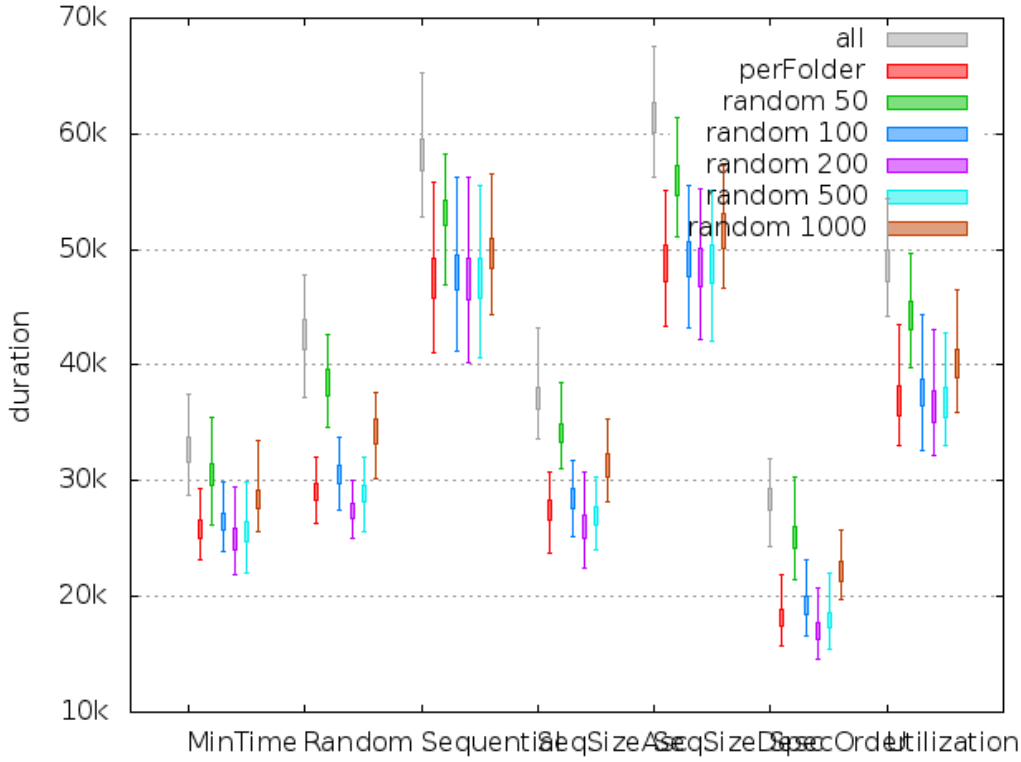



**Figure 5.5.:** Durations in seconds (y-axis) of the different download strategies in conjunction with the different Upload-Strategies (x-axis).

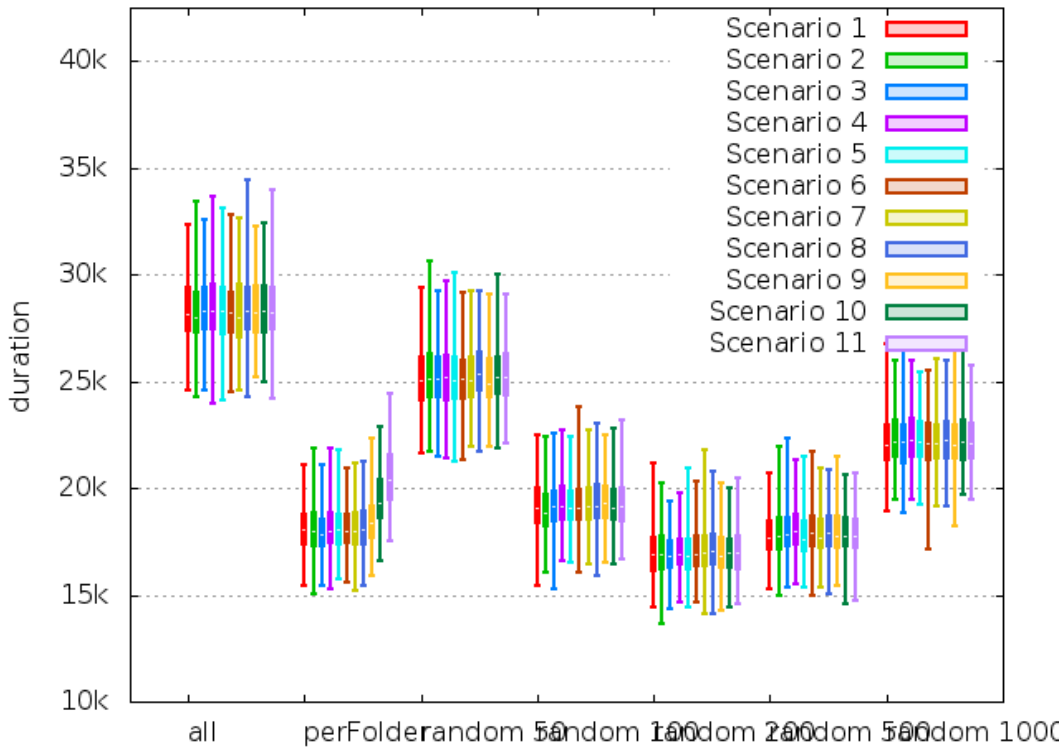



**Figure 5.6.:** Durations (y-axis) of the SpecifiedOrderStrategy for the different download strategies (x-axis) in correlation with the Scenarios.

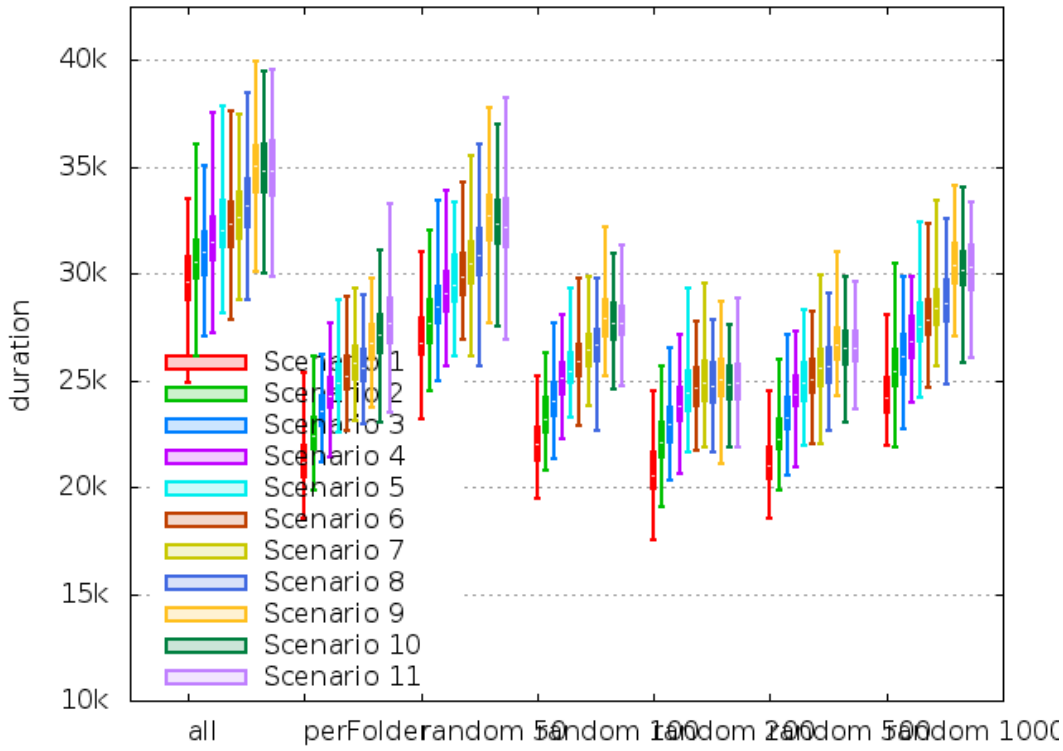



**Figure 5.7.:** Durations (y-axis) of the MinTimeStrategy for the different download strategies (x-axis) in correlation with the Scenarios.

## 6 Conclusion and Outlook

In this thesis we gave insights into the required metrics and considerations which are essential when building a combined cloud storage system, that transparently distributes files across different providers in order to improve the data transfer as well as the data reliabilty. We stated the involved goals and requirements, and introduced prevailing RAID levels in present environments. Furthermore, we demonstrated their application within a system that synergistically combines multiple cloud storage providers. After that we introduced different strategies that distribute files across the different providers based on distinct specifics in order to improve the data transfer process according to appropriate data.

Next we conducted two measurement studies: one aimed at measuring the representative file sizes and their distribution on the different cloud storage providers, and the other designed to measure the performance and transfer speeds of six contemporary providers. On the basis of the collected data we could show, that fourty-one percent of the files synchronized to cloud storage had a size of less than 2 KB and only one percent of the files were larger than 4096 KB. In the next step we developed connectors for each of the corresponding APIs and conducted an extensive measurement study for over three months. This study enables us to make a statement about the overall performance of the evaluated providers. In addition to this we were able to show, that the analyzed cloud storage providers have different transfer rates for the distinct file sizes.

In order to evaluate the proposed strategies, we built an adaptable simulation framework, implemented seven fundamental upload as well as three download strategies, and conducted a comprehensive simulation. This simulation used the information gained from the previous measurement studies in order to resemble the original cloud storage providers as well as possible. The simulation results depicted that our simple strategies already have a decent impact on the required time when transferring data to the cloud storage providers. Furthermore, we can point out that the strategy most suitable when uploading files is not ideal when it comes to reducing required download time. The fastest UploadStrategy is the MinTimeStrategy, that minimizes the upload time for a Batch by selecting the Provider with currently the least transfer time. The fastest download time could be showed when using the SpecifiedOrder UploadStrategy to upload files in combination with the “random 200” download strategy.

Although our efforts were able to yield definitive answers for many important questions, our work also created many new ones which prompt further investigation on the topic at hand. The measurement study to determine the performance for the different cloud storage providers could be extended to test new providers, since to the best of the author’s knowledge these kind of measurements have never been done before. Hence, they obviously represent an important criteria when making a decision of providers selection. Furthermore the measurement study should be expanded to test the data transfer rate and the rate limits when using multiple user accounts from a single machine with a distinct IP address; using $(f, r, n_1, u_1)$ and $(f, r, n_1, u_2)$ concurrently. In addition, it is also advisable to conduct the measurement at a greater extent and with more file sizes.

The entire application developed during this endeavour is divided into multiple modules in order to alleviate reuse and encourage further development of a single component. We established the basis for the previously mentioned, platform independent combined system, and thus it should be easily possible to build on our work when implementing a combined system.

The aforementioned simulation framework also resides in an early development state; something that could easily be matured with the addition of other key features. First of all the framework should be expanded to support the simulation of a striping functionality as mentioned in Chapter 3. This will give interesting insights that are very useful when implementing a combining system as mentioned before. Furthermore, the framework should be extended by new strategies that handle uploading and downloading of the data; for example, an UploadStrategy that distributes single directories to different cloud storage providers. It is even possible to realize an entire heuristic which decides, based on multiple characteristics, which provider should be used to store a file or a folder. Last but not least, it seems appropiate to conduct a study on how users interact with cloud storage providers. With this data in hand, it should be possible to model users behavior to improve the simulation and make it as realistic as possible.

## A List of Illustrations

### List of Figures



### List of Tables

## B Measurement Results

count:2052710
average:493.870005
min:0.000000
max:11171008.000000

| min size | max size | number | fraction | cum. fraction | size | fraction size | cum. fraction size |
|---|---|---|---|---|---|---|---|
| 0 | 2 | 839576 | 0.409009 | 0.409009 | 586748.093750 | 0.000579 | 0.000579 |
| 2 | 4 | 220128 | 0.107238 | 0.516246 | 634054.894532 | 0.000625 | 0.001204 |
| 4 | 8 | 183957 | 0.089617 | 0.605863 | 1027484.594726 | 0.001014 | 0.002218 |
| 8 | 16 | 211587 | 0.103077 | 0.708940 | 2569348.178714 | 0.002534 | 0.004752 |
| 16 | 32 | 216693 | 0.105564 | 0.814504 | 4639724.609375 | 0.004577 | 0.009329 |
| 32 | 64 | 95396 | 0.046473 | 0.860977 | 4429879.075197 | 0.004370 | 0.013699 |
| 64 | 128 | 70172 | 0.034185 | 0.895162 | 6194557.946288 | 0.006110 | 0.019809 |
| 128 | 256 | 44410 | 0.021635 | 0.916797 | 8067392.793943 | 0.007958 | 0.027767 |
| 256 | 512 | 41842 | 0.020384 | 0.937181 | 15719047.006833 | 0.015506 | 0.043272 |
| 512 | 1024 | 41001 | 0.019974 | 0.957155 | 29477656.557617 | 0.029077 | 0.072350 |
| 1024 | 2048 | 33817 | 0.016474 | 0.973629 | 48569570.707030 | 0.047910 | 0.120259 |
| 2048 | 4096 | 31869 | 0.015525 | 0.989155 | 95019277.241213 | 0.093728 | 0.213988 |
| 4096 | 8192 | 15314 | 0.007460 | 0.996615 | 81607997.163088 | 0.080499 | 0.294487 |
| 8192 | 16384 | 3649 | 0.001778 | 0.998393 | 41614728.337891 | 0.041049 | 0.335536 |
| 16384 | 32768 | 1346 | 0.000656 | 0.999049 | 30447984.530275 | 0.030034 | 0.365571 |
| 32768 | 65536 | 772 | 0.000376 | 0.999425 | 35668188.252928 | 0.035184 | 0.400754 |
| 65536 | 131072 | 550 | 0.000268 | 0.999693 | 49242183.766602 | 0.048573 | 0.449328 |
| 131072 | 262144 | 248 | 0.000121 | 0.999813 | 43938626.906250 | 0.043342 | 0.492669 |
| 262144 | 524288 | 167 | 0.000081 | 0.999895 | 61791145.666016 | 0.060952 | 0.553621 |
| 524288 | 1048576 | 101 | 0.000049 | 0.999944 | 81054193.215821 | 0.079953 | 0.633574 |
| 1048576 | 2097152 | 66 | 0.000032 | 0.999976 | 98038395.056640 | 0.096707 | 0.730281 |
| 2097152 | 4194304 | 21 | 0.000010 | 0.999986 | 61106547.950196 | 0.060276 | 0.790557 |
| 4194304 | 8388608 | 25 | 0.000012 | 0.999999 | 179349788.527344 | 0.176913 | 0.967471 |
| 8388608 | 16777216 | 3 | 0.000001 | 1.000000 | 32977376.000000 | 0.032529 | 1.000000 |

**Download in KB/s**

| | Sugarsync | Dropbox | Ubuntu One | Skydrive | Google Drive | Box.com |
|---|---|---|---|---|---|---|
| 1 B | 0.0025353826 | 0.0025316913 | 0.0006897638 | 0.0002392063 | 0.0014712794 | 0.0013201215 |
| 10 B | 0.025309894 | 0.0256675032 | 0.0067887545 | 0.0024425949 | 0.0163169057 | 0.0131734159 |
| 100 B | 0.2536431228 | 0.2528384261 | 0.0676325105 | 0.0246849774 | 0.1605210996 | 0.13235072 |
| 1 KB | 2.5305799483 | 2.5586109391 | 0.6896768531 | 0.2454582315 | 1.5644388765 | 1.2864934805 |
| 10 KB | 25.4452928044 | 23.4162978887 | 6.3563357486 | 2.262267521 | 16.0502362441 | 11.1009802997 |
| 100 KB | 150.9613621547 | 119.5843210981 | 47.1271140483 | 22.7023212484 | 129.8370958356 | 81.6738093864 |
| 1 MB | 297.0784494194 | 534.3330558095 | 158.6361727947 | 174.1945077874 | 617.1891002142 | 533.4672846979 |
| 10 MB | 337.4312628948 | 1863.9990950097 | 219.0463575171 | 483.4476564693 | 1148.1507944038 | 2664.3328852279 |
| 50 MB | 363.5965517817 | 3126.3048650329 | 358.8920007939 | 556.8968624437 | 8187.7903207935 | 3421.6767943565 |
| 100 MB | 351.9701310724 | 2870.2098148769 | 406.7225732297 | 575.3492383392 | 2352.4367370723 | 3854.7988576195 |

**Upload in KB/s**

| | Sugarsync | Dropbox | Ubuntu One | Skydrive | Google Drive | Box.com |
|---|---|---|---|---|---|---|
| 1 B | 0.000649041 | 0.0015126255 | 0.0002441403 | 0.0001088951 | 0.0007048755 | 0.0012908236 |
| 10 B | 0.0064346193 | 0.0150745758 | 0.0026131412 | 0.0011197618 | 0.0072832972 | 0.0128907318 |
| 100 B | 0.0651241175 | 0.1457450239 | 0.0259155726 | 0.0113790523 | 0.0723884302 | 0.1300204172 |
| 1 KB | 0.6473447897 | 1.2487066372 | 0.2646329489 | 0.113370978 | 0.7211591069 | 1.064399074 |
| 10 KB | 6.1209535987 | 1.9292954087 | 2.9307222764 | 1.0565295905 | 7.4232364122 | 10.7799666118 |
| 100 KB | 45.5314643399 | 5.6898450665 | 25.5675431243 | 10.8614334333 | 66.3267824967 | 82.6178067229 |
| 1 MB | 158.8204680968 | 8.3746301773 | 205.9207868547 | 94.2039822597 | 532.7260040249 | 657.4649427554 |
| 10 MB | 219.3926263627 | 8.0567278442 | 760.9491301269 | 408.7910814159 | 2985.5103433698 | 2625.4313440365 |
| 50 MB | 283.5544705285 | 3.8871638202 | 838.0628129727 | 492.7362792468 | 6108.196456698 | 2004.9554888599 |
| 100 MB | 275.2547776353 | 4.4029541854 | 987.8927596433 | 513.7173718818 | 6205.2257520365 | 4497.0669910512 |

# C Documentation

## Development and Build

### Sourcecode

The sourcecode is currently hosted in a private Github repository.

### Building the System

The system can be built using Apache Maven, enter the CloudStorageSystem folder and type the following command:

```
mvn package
```

This command will download the required dependencies and build the whole system directly. If you want to build just one module, enter the corresponding directory and execute the command above. The binaries of each module are in their respective *target* folders. A simple Mave introduction can be found here[1].

### Development using Eclipse

To contribute to the sourcecode using the IDE Eclipse you need to install the Eclipse Maven Plugin m2eclipse, which can be found here. After successful installation it is recommended to restart Eclipse. After that you can import the sourcecode:

1. Click on “File” → “Import...”
2. Expand “Maven” and select “Existing Maven Projects”
3. Select the root directory of the sourcecode
4. Expand “Advanced” and select “[groupId].[artifactId]” as Name template
5. Press “Finish”

Every connection of a storage provider’s API need to implement the *StorageProvider* interface, that creates the advantage that all the file operations are identical for every connected provider. The *StorageProvider* class has not been finished yet, so this interface may change during the next development iterations. This chapter briefly describes how the implemented file operations work at the moment.

## FileObject

Since storing files on a cloud storage provider produces metadata like hash sums, the providers id of the file or sharing information, an further data type has to be created. This data type is called *FileObject* and represents a normal file, but also stores the previously mentioned metadata and abstracts the provider specific file handling. A local FileObject incorporates a file or a folder stored on a remote cloud storage provider.

## Creating Files

Calling the *createFile(FileObject parent, File file)* method uploads the given file to the correspondent storage provider. When the parent FileObject is given, it is uploaded to the folder represented by the parent object, otherwise when the parent object is *null*, the file is uploaded to the root directory of the storage provider. The *createFile* method returns an FileObject which incorporates the original file.

[1] http://maven.apache.org/guides/getting-started/maven-in-five-minutes.html

### Updating Files

Updating files is slightly different to creating files, since a local FileObject already exists. Calling the *updateFile(FileObject file)* method just uploads the file referenced by the given FileObject. This method maintains the files parent folders and other metadata, it just updates the files contents and metadata depending on the content, like hashes.

### Downloading Files

In order to download a remote file pass the corresponding FileObject to the *downloadFile(FileObject fileObject)* method. The file will be downloaded and the FileObjects reference to the file will be updated.

### Creating Folders

The creation of a folder is similar to the creation of a file. Calling the *createFolder(FileObject parentFolder, String foldername)* method creates a folder with the name of the given *foldername* string. Similar to the file creation process, the folder is created in the root directory of the storage provider when *null* is passed as parent object, otherwise the folder is created as a subfolder of the given parent object. The *createFolder* method returns a FileObject which incorporates the original folder.

### Deleting Files

Files can easyily deleted using the *deleteFile(FileObject file)* method, which deletes the corresponding remote file. It must be pointed out, that this method does not delete the local file and its return type is void.

### Deleting Folders

Folders can easyily be deleted using the *deleteFolder(FileObject folder)* method, which deletes the corresponding remote folder. It must be pointed out, that this method does not delete the local folder and its return type is void.

### Example

This simple example shows, how an application could use CloudStorageSystems API Connectors:

```
import java.io.File;
import java.io.IOException;

import de.tudarmstadt.p2p.connectors.Box.BoxAuthenticator;
import de.tudarmstadt.p2p.connectors.Box.BoxProvider;
import de.tudarmstadt.p2p.connectors.Dropbox.DropboxAuthenticator;
import de.tudarmstadt.p2p.connectors.Dropbox.DropboxProvider;
import de.tudarmstadt.p2p.connectors.common.Exceptions.AuthenticationException;
import de.tudarmstadt.p2p.connectors.common.Exceptions.StorageException;
import de.tudarmstadt.p2p.connectors.common.interfaces.StorageProvider;
import de.tudarmstadt.p2p.connectors.common.objects.FileObject;


public class Main {

  public static void main(String[] args) throws AuthenticationException,
      IOException, StorageException {
    File dropboxAuthFile = new File("dropbox.json");
    File boxAuthFile = new File("box.json");

    // authentication process
    DropboxAuthenticator dropboxAuth = new DropboxAuthenticator();
    if (dropboxAuthFile.exists()) {
```

```
      dropboxAuth.loadConfiguration(dropboxAuthFile);
    } else {
      dropboxAuth.authenticate();
      dropboxAuth.saveConfiguration(dropboxAuthFile);
    }

    BoxAuthenticator boxAuth = new BoxAuthenticator();
    if (boxAuthFile.exists()) {
      boxAuth.loadConfiguration(boxAuthFile);
    } else {
      boxAuth.authenticate();
      boxAuth.saveConfiguration(boxAuthFile);
    }

    // create StorageProvider connection
    StorageProvider dropbox = new DropboxProvider(dropboxAuth);
    StorageProvider box = new BoxProvider(boxAuth);

    // upload files to the root directory
    FileObject file1 = dropbox.createFile(null, new File(
        "/home/diedrich/Downloads/file.txt"));
    FileObject file2 = box.createFile(null, new File(
        "/home/diedrich/Downloads/file2.txt"));

    // download remotely change files
    file1 = dropbox.downloadFile(file1);
    file2 = box.downloadFile(file2);

    // delete files again
    dropbox.deleteFile(file1);
    box.deleteFile(file2);

    // create directory containing a file
    FileObject folder = dropbox.createFolder(null, "myfolder");
    FileObject file3 = dropbox.createFile(folder, new File(
        "/home/diedrich/Downloads/file3.txt"));

    // delete file and folder
    dropbox.deleteFile(file3);
    dropbox.deleteFolder(folder);
  }

}
```

## D Bibliography


[1] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. RACS: A Case for Cloud Storage Diversity. 2010.

[2] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. *ACM Transactions on Storage*, 3(3):9–es, October 2007.

[3] Moritz Borgmann, Tobias Hahn, Michael Herfert, Thomas Kunz, Marcel Richter, Ursula Viebeg, and Sven Vowé. *On The Security of Cloud Storage Services*. 2012.

[4] Kevin D. Bowers, Ari Juels, and Alina Oprea. HAIL: A High-Availability and Integrity Layer for Cloud Storage. pages 187–198, 2009.

[5] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26:145–185, 1994.

[6] Shobhit Dayal. Characterizing HEC Storage Systems at Rest. *Parallel Data Lab, Carnegie Mellon University, Pittsburgh, PA, USA*, (May), 2008.

[7] Kylie M Evans and Geoffrey H Kuenning. A Study of Irregularities in File-Size Distributions. 2002.

[8] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[9] E. Hammer-Lahav. The OAuth 1.0 Protocol. RFC 5849 (Informational), April 2010. Obsoleted by RFC 6749.

[10] D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard), October 2012.

[11] Philipp C. Heckel. *Minimizing remote storage usage and synchronization time using deduplication and multichunking: Syncany as an example*. Masterthesis, University of Mannheim, 2012.

[12] Wenjin Hu, Tao Yang, and Jeanna N. Matthews. The good, the bad and the ugly of consumer cloud storage. *ACM SIGOPS Operating Systems Review*, 44(3):110, August 2010.

[13] Ratul Mahajan, Steven M. Bellovin, Sally Floyd, John Ioannidis, Vern Paxson, and Scott Shenker. Controlling high bandwidth aggregates in the network. *ACM SIGCOMM Computer Communication Review*, 32(3):62–73, July 2002.

[14] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing (Draft). 145:7, 2011.

[15] Andrew Odlyzko. The Internet and Other Networks: Utilization Rates and Their Implications. *DIMACS Technical-Report*, 07(February), 1999.

[16] David A. Patterson, Peter Chen, Garth A. Gibson, and Randy H. Katz. Introduction to Redundant Arrays of Inexpensive Disks (RAID). In *Proc. IEEE COMPCON*, pages 112 – 117, 1989.

[17] Rade Stanojevic and Robert Shorten. Generalized distributed rate limiting. *2009 17th International Workshop on Quality of Service*, (2):1–9, July 2009.

[18] Mark W. Storer, Kevin Greenan, Darrell D. E. Long, and Ethan L. Miller. Secure Data Deduplication. 2008.

[19] Chee Wei Tan, Dah-ming Chiu, John C. S. Lui, and David K. Y. Yau. A Distributed Throttling Approach for Handling High Bandwidth Aggregates. 18(7):983–995, 2007.

[20] Klaus Wilhelmi. *Ein sicherer Raid-Manager für freie Cloud Storages*. Bachelorthesis, TU Darmstadt, 2012.

[21] Jiyi Wu, Lingdi Ping, Xiaoping Ge, Ya Wang, and Jianqing Fu. Cloud Storage as the Infrastructure of Cloud Computing. *2010 International Conference on Intelligent Computing and Cognitive Informatics*, pages 380–383, June 2010.

[22] Wenying Zeng, Yuelong Zhao, Kairi Ou, and Wei Song. Research on cloud storage architecture and key technologies. *Proceedings of the 2nd International Conference on Interaction Sciences Information Technology, Culture and Human - ICIS '09*, pages 1044–1048, 2009.