SEVENTH FRAMEWORK PROGRAMME

CloudSpaces

(FP7-ICT-2011-8)

Open Service Platform for the Next Generation of Personal Clouds

D2.2 Draft architecture specifications, use case scenarios and benchmarking framework

Due date of deliverable: 30-11-2013
Actual submission date: 12-11-2013

Start date of project: 01-10-2012 Duration: 36 months

Summary of the document

Document Type	Deliverable
Dissemination level	Public
State	Final
Number of pages	85
WP/Task related to this document	WP2
WP/Task responsible	URV
Author(s)	Refer to contributors list
Partner(s) Contributing	URV, EPFL, EUR, TST, EOS, CNC
Document ID	CLOUDSPACES_D2.2_131112_Public.pdf
Abstract	Draft architecture specifications, use case scenarios and benchmarking framework: Architecture specifications of the CloudSpaces toolkit. First draft of platform APIs: Store, Proximity, Share, Export, Privacy, Logging, Persistence. Use cases, demonstration scenarios.
Keywords	Cloud storage, synchronization, sharing, interoperability, Personal Cloud

Contributors

Name	Last name	Affiliation	Email
Adrián	Moreno Martínez	URV	adrian.moreno@urv.cat
Pedro	García López	URV	pedro.garcia@urv.cat
Hamza	Harkous	EPFL	hamza.harkous@epfl.ch
Rameez	Rahman	EPFL	rameez.rahman@epfl.ch
Marko	Vukolic	EUR	Marko.Vukolic@eurecom.fr
José Miguel	García López	TST	jmgarcia@tissat.es
John	Lenton	CNC	john.lenton@canonical.com

Table of Contents

1 Executive summary

The purpose of the present CloudSpaces project's deliverable D2.2 (Draft architecture specifications, use case scenarios and benchmarking framework) is to define the design principles for open personal clouds, the CloudSpaces draft architecture, the guidelines on adaptive synchronization and replication schemes, guidelines on privacy-aware data sharing, and the semantic and syntactic interoperability of personal data.

In Section 3, we first introduce StackSync, a scalable open source software framework that implements the basic components of a scalable synchronization tool. StackSync will be used to design, implement, and validate essential contributions of the project such as the Personal Cloud interoperability, privacy-aware data sharing or achieving an adaptive personal storage, among others. We explain the different components of StackSync, which are the clients, the synchronization service, the storage service, and the communication middleware.

Afterwards, we explore an adaptive replication and synchronization for personal storage infrastructure that will integrate different user and Cloud storage resources and avoid data lock-in. Our focal use case is specifically SME that wishes to use (personal) cloud services, but wants more control over its own data. Such a SME, desiring to use public cloud service, typically owns some computation and storage resources in a private infrastructure. Our approach is to leverage private portion of these resources to store personal cloud metadata, to give a user control over data stored in a public cloud. We refer to this approach as Hybrid Cloud Storage (HCS).

Moreover, in Section 4 we also detail a novel adaptive content distribution design for Cloud systems that is able to accommodate a sudden demand of data, i.e., when a large number of requests are received in short period of time. Our approach relies on Bittorrent to offload storage by monitoring the activity of users and upon detection of a certain critical mass, it transparently generates a torrent file to seamlessly switch to BitTorrent and reduce its bandwidth consumption.

In Section 5 we address the privacy problem in the personal cloud from the perspective of an end user, whose sensitive data needs to be protected and who aims to use the cloud services whenever suitable. We tackle this problem as a privacy risk management process, conducted in two steps: risk estimation and risk mitigation. We attempt at solving the former by quantifying the risk of data sharing, via a mechanism that relies on users' sharing policies, works for general data types, and accounts for the sharing semantics. Building on our techniques for quantifying the privacy risks, we aim at designing a suite of applications for mitigating those risks. Our target behind that is to provide users with mechanisms for smoothly managing the trade-off between the privacy risk and the services desired.

An overview of the service platform is provided in Section 6, with a coherent, integrated, and high-level interface to all platform functionalities including the storage, sharing, persistence, proximity and portability approaches.

In the Benchmarking section we provide a description of the expected evaluation, including guidelines to validate the system's effectiveness in terms of privacy and performance. Using real-world datasets, we will evaluate the privacy considering the time it takes for determining the sensitivity of an item and evaluating the privacy risk. We will also attempt at

seeing the rate at which the privacy risk can be decreased via various mitigation techniques. The performance will be evaluated using a series of QoS benchmarks, and the interoperability assessed with conformance tests, guaranteeing that implementations are compliant with the defined interoperability and storage standards.

Next, we define a set of simulation/experimentation tools and testbeds that will facilitate the overall validation of the project results. In addition, we offer guidelines for industrial partners to help prepare the traces. This traces will benefit partners, as well as the whole project, in a twofold manner: i) academic partners will work with real-world data, giving to their research technical strength and impact, and ii) the advances achieved using a specific dataset are applicable to the service where such a dataset comes from. This represents that many potentially beneficial and novel techniques will be ready for exploration to industry partners during the development of the project.

Finally, in Section 7.5.3 we detail the demonstration scenarios for the different contributions of the CloudSpaces project. We explain the use cases that will be presented in the different demonstration as well as the expected outcome.

2 Introduction

CloudSpaces aims to create the next generation of Personal Clouds, namely Personal Cloud 2.0, offering advanced issues like interoperability, advanced privacy and access control, and scalable data management of heterogeneous storage resources. Furthermore, it will offer an open service platform for third-party applications leveraging the capabilities of the Open Personal Cloud. The CloudSpaces project has three main building blocks: CloudSpaces Share, CloudSpaces Storage and CloudSpaces Services.

CloudSpaces Share will deal with interoperability and privacy issues. The infrastructure must ensure privacy-aware data sharing and trustworthy assessment from other Personal Clouds. It will also overcome existing vendor lock-in risks thanks to open APIs, metadata standards, personal data ontologies, and portability guarantees.

CloudSpaces Storage takes care of scalable data management of heterogeneous storage resources. In particular, users retaking control of their information implies control over data management. This new scenario clearly requires novel adaptive replication and synchronization schemes dealing with aspects like load, failures, network heterogeneity and desired consistency levels.

Finally, CloudSpaces Services provides a high level service infrastructure for third-party applications that can benefit from the Personal Cloud model. It will offer data management (3S: Store, Sync, Share), data-application interfaces, and a persistence service to heterogeneous applications with different degrees of consistency and synchronization.

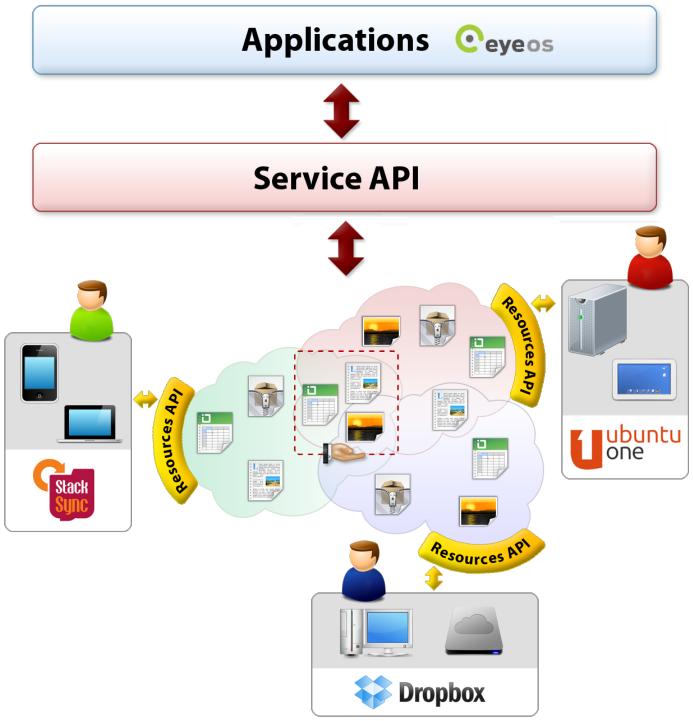


Figure 1: Big picture of the service platform

In Figure 1 we observe the picture that captures the main contribution of the project. On the bottom part we observe how heterogenous Personal Clouds are able to share specific files, so that users on any of the participant Personal Clouds can access files hosted on other services seamlessly. This transparency when accessing different Personal Clouds is possible thanks our service platform, which is a common interface that all participants must implement. Likewise, the interoperability protocol allows Personal Clouds to establish a sharing agreement between users while preserving their privacy, allowing horitzonal interoperability.

In the upper part, we observe the *Service API*, which allows third-party application to access the CloudSpaces infrastructure by offering a collection of services. These services include access to Personal Clouds data in terms of storage, synchronization and data sharing, allowing vertical interoperability between services and applications.

The project will demonstrate interoperability between StackSync and Ubuntu One, and complete integration of eyeOS on top of any of them thanks to the *Service API*.

3 The StackSync framework

In this section we will introduce StackSync, a scalable open source software framework that implements the basic components of a scalable synchronization tool.

3.1 Why StackSync?

In the last few years, we have seen how the market of cloud storage is growing rapidly. Despite the rush to simplify our digital lives, many of the commercial Personal Clouds in operation today like Dropbox are *proprietary*, and rely on algorithms that are *invisible* to the research community, and what is even worse, existing open source alternatives fall short of addressing all the requirements of the Personal Cloud. Next we discuss the existing open source solutions for the Personal Cloud, namely SparkleShare, ownCloud and Syncany.

SparkleShare[1] is built on top of Git, using it as both its storage and syncing back-end. SparkleShare clients use push notification to receive changes, and maintain a direct connection with the server over SSH to exchange file data. When a client is started, it connects to a notification server. The notification server tells the other clients subscribed to that folder that they can pull new changes from the repository after a user change.

Using Git as the storage back-end is a double-edged sword. While Git implements an efficient request method to download changes from the server (git pull command), avoiding massive metadata exchanges between server and clients, it is not prepared to process large binary files. Also, this architecture tied to the Git protocol is also difficult to scale and deploy in cloud environments.

ownCloud[2] is the most famous open source Personal Cloud and they have an active community. We refer here to the Community Edition of ownCloud, since their enterprise edition is not available to the public. In ownCloud, clients communicate with the server following a *pull* strategy, i.e. clients ask periodically to the server for new changes. There are two types of data traffic: data and metadata. For data exchange, ownCloud uses a REST API; however, metadata traffic is transferred using the WebDAV protocol. Because both types of traffic are processed by the same server, data and metadata traffic are completely coupled.

Unfortunately, ownCloud is not an extensible and modular framework like StackSync. In this line, the ownCloud developer community is mainly working around the web front-end. Although we will devote a subsection to ownCloud in the validation, we can advance that their inefficient pull strategy with massive control overheads is not scalable. Furthermore, their sync flows and data flows are tightly coupled, and they do not even provide basic chunking or deduplication mechanisms.

Syncany[3] is an open source Personal Cloud developed by Philip Heckel in Java. It is a client-side Java application that can synchronise against a variety of storage back-ends thanks to their extensible plugin model. They also provide extensible mechanisms for chunking and their architecture is elegant, clean and modular.

[1] `http://sparkleshare.org`
[2] `http://owncloud.org`
[3] `http://syncany.org`

Although we give much credit to Syncany, indeed the StackSync client is a branch project that evolved from Syncany, it presents a number of drawbacks that made us evolve towards the current StackSync architecture. The major shortcoming is the lack of scalability of Syncany due to its heavy pull strategy with metadata and data flows heavily coupled. To support versioning and resolve conflicts, Syncany relies on a metadata file that contains the complete history of each individual file, and that is stored in the storage back-end as a regular file. To determine the most recent version of a file, the Syncany client needs first to download this metadata file, which grows with each new file modification, severely limiting scalability.

For these reasons, we decided to create StackSync, an open framework for Personal Cloud systems. Its architecture is highly modular, with each module represented by a well-defined API, allowing third parties to replace components for innovation in versioning, deduplication, live synchronization or continuous reconciliation, among other relevant topics. StackSync will be used in the CloudSpaces to design, implement, and validate essential contributions of the project such as Personal Cloud interoperability, privacy-aware data sharing or achieving an adaptive personal storage.

3.2 Understanding Personal Clouds

Personal Clouds are complex infrastructures involving a variety of distributed components. Even for setting up and running a basic service, engineers in charge of building it need to implement many processes. The lack of a reference open source implementation complicates even more the design of these systems.

To put in perspective, we describe next the typical interactions among the different blocks of a Personal Cloud architecture. Among the three key services, here we will focus only on the synchronization service. The sharing service can be considered as a particular case of file synchronization where users set sharing controls at the account level to grant access of shared folders and links to other users. On the other hand, the storage service must offer a clean and simple file-system interface for archiving, backup, and even primary storage of files, abstracting away the complexities of direct hardware management. At the same time, the storage service must guarantee the availability of data files stored by users, which is achieved by adding redundancy to multiple servers. As the implementation of the storage service is more related to hardware issues like redundancy management, and it is in general not architecturally relevant, discussion on the storage service has been skipped due to space constraints.

It must be noted that Personal Clouds usually treat differently desktop clients in PCs and laptops from mobile and Web clients. Whereas the former maintain a copy of the information and check for changes, mobile clients retrieve the information on demand. We will focus only on the desktop scenario.

3.3 Architecture overview

In general terms, StackSync can be divided into four main blocks: clients, synchronization service, storage back-end, and communication middleware. An overview of the architecture

with the main components and their interaction is shown in Figure 2. The StackSync client and the sync service interact through the communication middleware called ObjectMQ. The sync service interacts with the metadata database. The StackSync client directly interacts with the storage back-end to upload and download files.

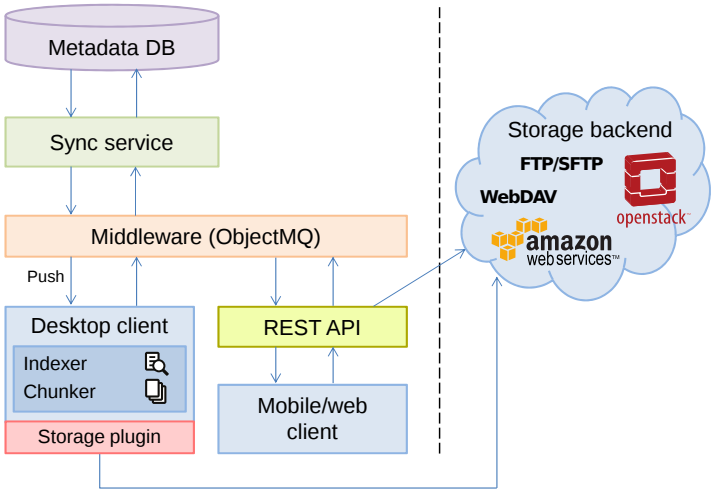


Figure 2: StackSync architecture overview

As we can see in the figure above, the storage back-end is separated from the rest of the architecture by a line. This means that StackSync can be used with external storage back-ends such as Amazon S3 or RackSpace. This enables StackSync to fit different organization requirements and be offered in three different configurations:

- **Public Cloud.** Data and metadata is stored in a public storage provider such as Amazon or Rackspace.
- **Private Cloud.** StackSync is installed on-premise. Data and metadata is stored on the company's infrastructure.
- **Hybrid Cloud.** Data is stored in a public storage provider and metadata is kept inside the company's infrastructure. This allows organizations that sensible information is stored on-premise, while raw data is stored encrypted on a third-party storage service.

3.4 Desktop client

The main StackSync client is a application that monitors local folders and synchronizes them with a remote repository. The StackSync client interacts with both the synchronization and storage services. The first one handles metadata communication, such as time stamps or file names, and the second one is in charge of raw data.

This decoupling of sync control flows from data flows implies a user-centric design where the client directly controls its digital locker or storage container. The synchronization protocol have been designed to put the load in the client side, whereas the synchronization service just checks if the change is consistent, and then applies all changes proposed by clients.

The desktop client must meet some essential requirements:

- **Detect file changes.** Every time a user modifies a file inside the synchronized folder, the client must be aware of the event and process it in order to be synchronized with StackSync.
- **Handle conflicts.** Conflicts may occur on StackSync due to offline or concurrent operations from different locations. For instance, two users sharing a folder may edit the same file at the same time, so the client must be prepared to handle file conflicts.
- **Automatic updates.** As the application will be on constant improvement, it must be able to transparently upgrade to newer version.
- **Error reporting.** The application must log every error occurred and report it to the server in order to identify the problem and get it fixed.
- **Visual information.** The application must be integrated with the operating system shell to show information about the state of files. This information will be displayed to users in the form of an overlay icon that will change from one icon to another depending on the state. In Figure 3 we observe the icons that will be displayed when a file is being synchronized, is already synchronized, or cannot be synchronized due to an exceeded quota or other errors.

Figure 3: Overlay icons

3.4.1 File processing

The desktop client must integrate with the system in the sense that, when there is a change in the synchronized folder, the application receives a notification to process the event. Once the application identifies which file or files have been modified or created, it will proceed to store it in StackSync.

First, it will extract all needed metadata about the file (e.g. file name, size, modification date, etc.). Next, it will store it in a database that will be used to keep the information about different version of files. Finally, the file will be split into small pieces called chunks. Each chunk is treated independently and is identified by its hash value. To reconstruct the original file, chunks must be joined in the correct order.

The chunking provides StackSync with the following advantages:

1. **Optimize the bandwidth usage.** When a file is modified, it is processed again and new chunks are generated. But only those parts of the file that have been modified will generate different chunks compared to the previous version. The client will be aware of this and will only send the chunks that differ from the previous version.

2. **Optimize the storage usage.** Without chunking, the smallest modification of a file would force the client to upload the file again, and therefore, using more storage space. Using chunking we significantly reduce the storage usage.

3. **Improve big files transfers.** Syncing a $3GB$ file is a costly operation for both the client and the server. Chunks normally use much less space (between $32KB$ and $1MB$), so they can be transferred faster. In addition, if the connection is lost while synchronizing, the client may resume the process from the last uploaded chunk and avoid uploading the entire file again.

When the chunking process finishes, the client proceeds to upload the chunks to the storage service, storing the chunks in the user's private space.

Once the unique chunks are successfully submitted to the `Storage back-end`, the `Indexer` will communicate with the `SyncService` to commit the changes to the `Metadata back-end`, which is the component responsible for keeping versioning information. The `Metadata back-end` may be a relational database like MySQL or a non-relational data store like Cassandra[4] or Riak[5], now frequently called NoSQL databases. Irrespective of the chosen database technology, the `SyncService` needs to provide a consistent view of the synced files. Allowing new commit requests to see uncommitted changes may result in unwanted conflicts. As soon as more than one user works with the same file, there is a good chance that they accidentally update their local working copies at the same time. It is therefore critical that metadata is consistent at all times to establish not only the most recent version of each individual file but to record its (entire) version history. Although relational databases process data slower than NoSQL databases, NoSQL databases do not natively support ACID transactions, which could compromise consistency, unless additional complex programming is performed. Since the nature of the `metadata back-end` strongly determines both the scalability and complexity of the synchronization logic, an open modular architecture like StackSync can reduce the cost of innovation, adding a great flexibility to meet changing needs.

Finally, when the `SyncService` finishes the commit operation, it will then notify of the last changes to all out-of-sync devices. The device that originally modified the local working copy of the file will just update the `Indexer` upon the arrival of the confirmation from the `SyncService`. The other devices will both update their local databases and download the new chunks from the `Storage back-end`. Here we are assuming that an efficient communication middleware mediates between devices and the `SyncService`. This middleware should support efficient marshaling and message compression to reduce traffic overhead. Very importantly, it should support scalable change notification to a high number of entities, using either pull or push strategies. To deduplicate files and offer continuous reconciliation [1], recall that the local database at the `Indexer` must be in sync with the `Metadata back-end`, for which notification must be performed fast.

Figure 4 illustrates the interaction between all the components of a file sync engine. Observe that not all the different components described in this section are present in the architecture of a Personal Cloud. In some architectures, our overall model could be simplified and one component could be responsible for many tasks. Some architectures can even lack some components. For example, ownCloud does not provide deduplication and chunking.

[4] `http://cassandra.apache.org/`
[5] `http://basho.com/riak/`

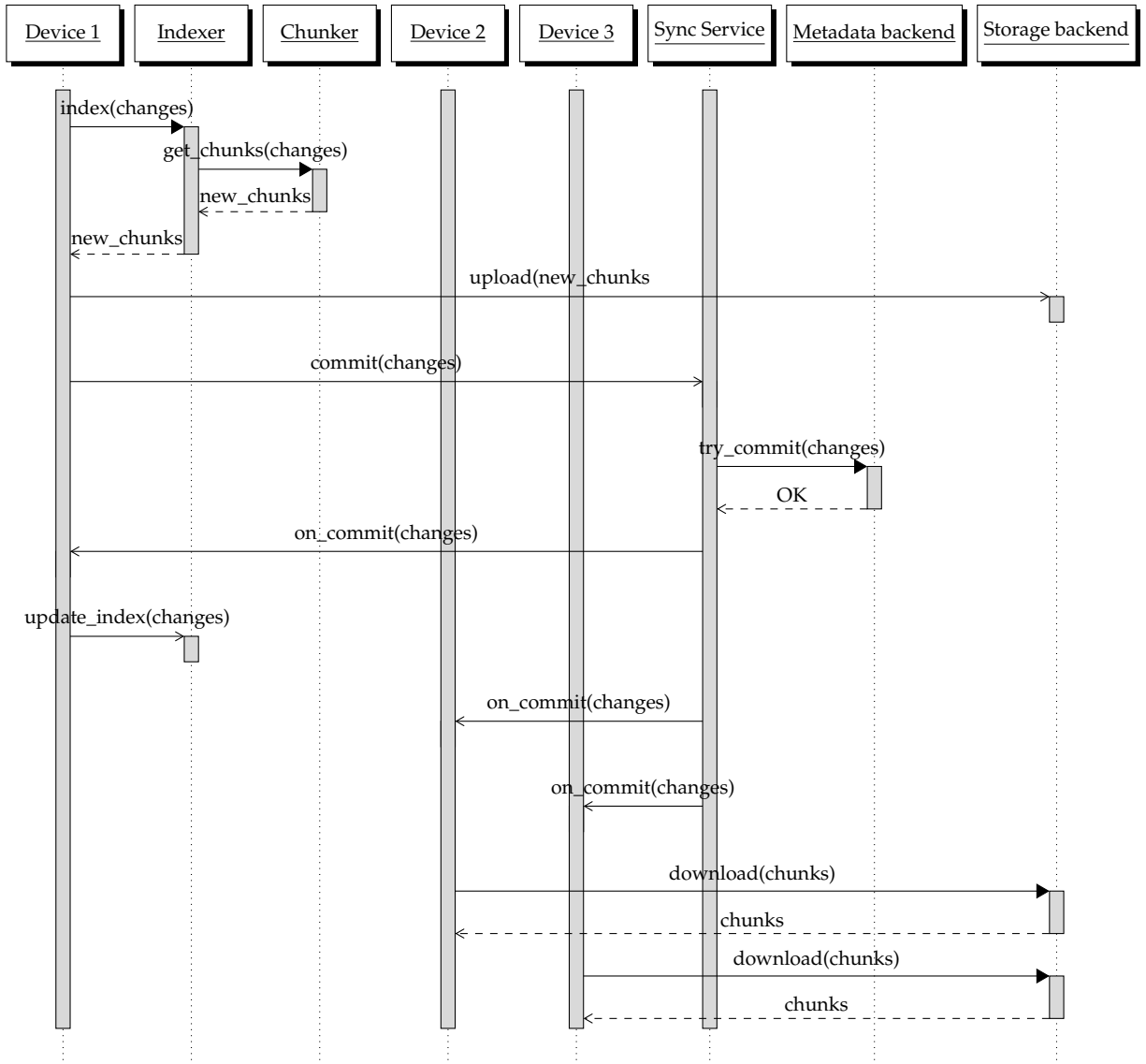


Figure 4: Interaction between the components of sync engine for a Personal Cloud.

3.4.2 Conflicts

As in any Personal Cloud system, synchronization conflicts are likely to occur. As it not possible to avoid these errors, we need to implement a strategy to deal with and resolve them. For example, it can happen in a situation where two or more users modify the same file simultaneously.

It can happen that, while working with a file, we lose the Internet connection. Clients must be prepared to handle these situations. While the client has no Internet connection be, it cannot notify any updates to the server, so the client must keep all changes in the internal database until the connection is reestablished and they can be submitted to the server.

Imagine a user that modified some files in his personal computer with no Internet connection, i.e., changes were not committed to the server. Afterwards, he went to work and modified the same files. Changes were successfully send to the server. Then, when he returns to his personal computer, now connected to Internet, first of all StackSync will retrieve the remote changes and will realize that some remote files are in conflict with the local ones.

StackSync will rename the local files adding "Conflicted copy" to the name and treating them as new files, letting the user decide which file is the correct one.

3.4.3 Architecture

In Figure 5 we can observe the architecture of the desktop client. Next, we will detail each element and explain the relation between them.

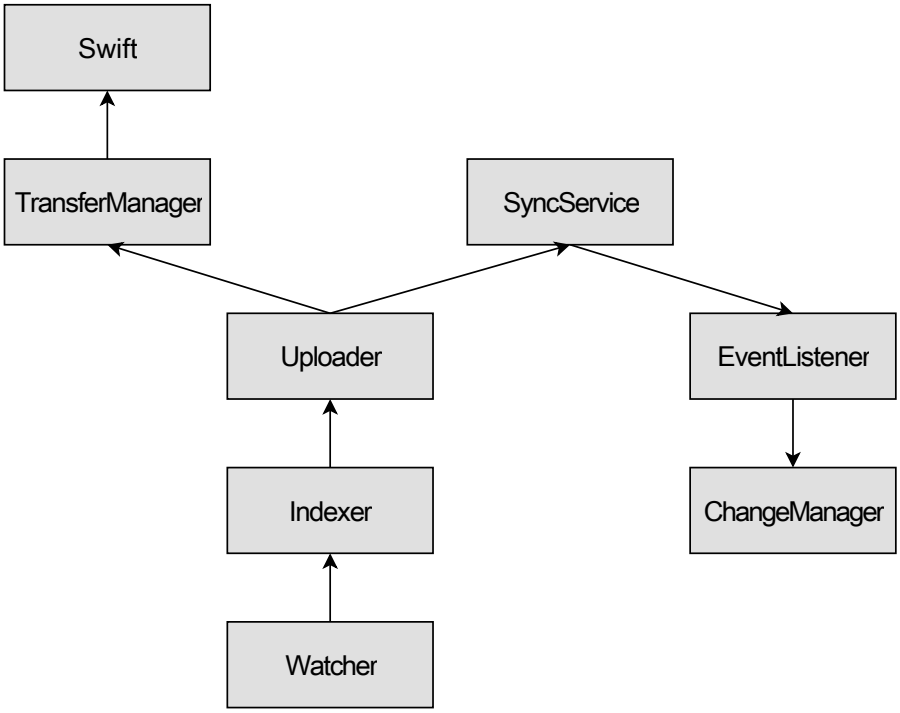


Figure 5: Desktop client architecture.

- **Watcher.** Receives modification events suffered by files located in the synchronization folder. Any action performed by the user (e.g., create a file or folder, delete them, etc.) is captured by the `Watcher` and notified to the `Indexer`.
- **Indexer.** It must process all events received from the `Watcher`. It updates the internal database and creates the chunks from the modified file.
- **Uploader.** It is charge of communicating with `OpenStack Swift` and the `SyncService`. First, it uploads the chunks to Swift through the `TransferManager` interface. Then, it notifies the `SyncService` about the committed operation.
- **EventListener.** It awaits events from the `SyncService`. When receiving events, it queues them in order for the `ChangeManager` to sequentially process them.
- **ChangeManager.** Processes all events coming from the `SyncService`. It contains the synchronization algorithm that ensures consistency in the client side, which can be observed in Algorithm 1.

Algorithm 1 Pseudocode of the synchronization algorithm

```
1:  function ProcessEvent(event)
2:      file_metadata = event.get_file_metadata
3:      if file_ID_in_DB(file_metadata) then
4:          if correct_version then
5:              apply_update
6:          else
7:              apply_conflict
8:          end if
9:      else
10:         if exist_file_locally then
11:             apply_conflict
12:         else
13:             apply_new
14:         end if
15:     end if
16: end function
```

3.4.4 Communication

As we have mentioned before, clients are in constant communication with both, the metadata and storage services. Communication with the metadata service is bidirectional, that is, clients notify updates about local file changes to the server at any time, and receive events about remote modifications.

In order for the clients to keep updated, they need to be aware of all changes made in their remote repository. This can be achieved in two different ways: periodically asking the metadata service if there is any change (i.e. Pull strategy); or the metadata service to notify clients when there is a change (i.e. Push strategy). These two strategies have been compared in many research reports [2, 3, 4]. Next, we will provide a brief description of their advantages and disadvantages.

- **Pull.** In this strategy, the information is available on the server and clients are asking periodically for changes. If there were many clients connected simultaneously, the server could collapse when receiving too many requests. It would create many request that most times would be unnecessary. Thus, as we pretend to create a scalable system to be able to accommodate a large number of users, this strategy is not suitable.
- **Push.** It is the opposite case, clients are kept waiting for the server to notify them about changes, saving many request to the server. But has the drawback that clients have to maintain an open connection to receive notifications.

As we prioritized the server bandwidth and load, we opted for a push-based communication between desktop clients and servers.

Clients can perform two types of server calls: synchronous and asynchronous.

Synchronous calls

There are some requests that clients must perform in order for them to be initialized. These type of calls are blocking, i.e., when the client makes the request, it is blocked until the server responds.

Each time a client is started, it needs to apply all changes made since the last time it was running. As notifications follow a push strategy, if they are not processed by the time the server sends them, they are lost. Therefore, clients must ask the server for the current state of the files to check themselves what changes have occurred.

The fact that these type of calls are blocking makes the time to process it a critical issue. I the server does not respond within a time period, more request could accumulate, specially at peak hours.

Asynchronous calls

On the other hand, some other requests require significant processing time to ensure data consistency or are not time-dependent. Decoupling the request from the response in these scenarios is vital to ensure the system's scalability.

An example of asynchronous call is when the client wants to upload the metadata of a new version. When the client finishes uploading a file to the data server, it sends a request to provide the metadata with all the necessary information. The server, after analyzing the data consistency, sends an event to all interested devices.

From the time the message is sent to the server, until notification is received, the client can perform other tasks.

3.4.5 Chunking

In the early versions of StackSync, the client implemented a static chunking algorithm, which works well for a system with a small number of users. But as we want to create a truly scalable system, we were forced to use a dynamic chunking.

A content-based dynamic chunking can significantly reduce the amount of space used by users in the storage service. Unlike the static chunking, when the user modifies a file, the chunking algorithm detects what parts of the file have not been modified and do not transfer them again. On the other hand, the computational cost of applying this algorithm is more intensive. But as it is not a time-critical operation and the storage savings are significant, we implemented it.

The content-based chunking to be implemented was the TTTD (Two Thresholds Two Divisors). This algorithm must specify the minimum and maximum sizes of chunks (two limits). From the first byte of the file, the algorithm reads byte by byte until it reaches the minimum size of the chunk. Then, it computes the hash of the chunk and calculates the module of the hash with a main and a secondary value (two dividers). If the result is equal to the first divisor, it means that the chunk must begin there.

Figure 6 we can see the worst case scenario for a static chunking. Add a byte to the beginning of the file will cause all chunks to be different from the previous version. In contrast to the dynamic chunking, the only chunk that would be affected would be the first one, once you find the intersection point with the divisors, all other chunks will be the same and will not upload them.

In the Algorithm 2 we provide the pseudocode for the TTTD content-based chunking.

Algorithm 2 Pseudocode of the TTTD content-based chunking

```
1:  function TTTD(input)
2:      p = 0
3:      l = 0
4:      backupBreak = 0
5:      while not EOF(input) do
6:          c = getNextByte(input)
7:          hash = updateHash(c)
8:
9:          if p − 1 < Tmin then
10:             //Not at minimum size yet
11:             continue
12:         end if
13:
14:         if (hash%Ddash) == (Ddash − 1) then
15:             //Possible backup break
16:             backupBreak = p
17:         end if
18:
19:         if (hash%D) == (D − 1) then
20:             //Found a breakpoint
21:             //before the maximum threshold
22:             addBreakpoint(p)
23:             backupBreak = 0
24:             l = p
25:             continue
26:         end if
27:         if (p − 1) < Tmax then
28:             //Fail to find a breakpoint,
29:             //but we are not at the maximum yet
30:             continue
31:         end if
32:
33:         //When we reach here, we have not found a breakpoint with
34:         //the main divisor, and we are at the threshold. If there
35:         //is a backup breakpoint use it. Otherwise impose a hard threshold.
36:         if backupBreak! = 0 then
37:             addBreakpoint(backupBreak)
38:             l = backupBreak
39:             backupBreak = 0
40:         else
41:             addBreakpoint(p)
42:             l = p
43:             backupBreak = 0
44:         end if
45:         p = p + 1
46:     end while
47: end function
```

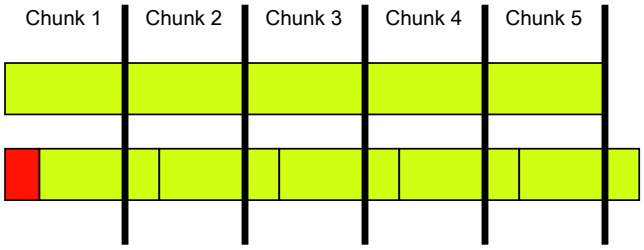


Figure 6: Static chunking example.

3.4.6 Internal database

The client has to maintain internal database to keep track of the files and versions it has in the local repository. In Figure 7 we can observe the database model used in the desktop client.

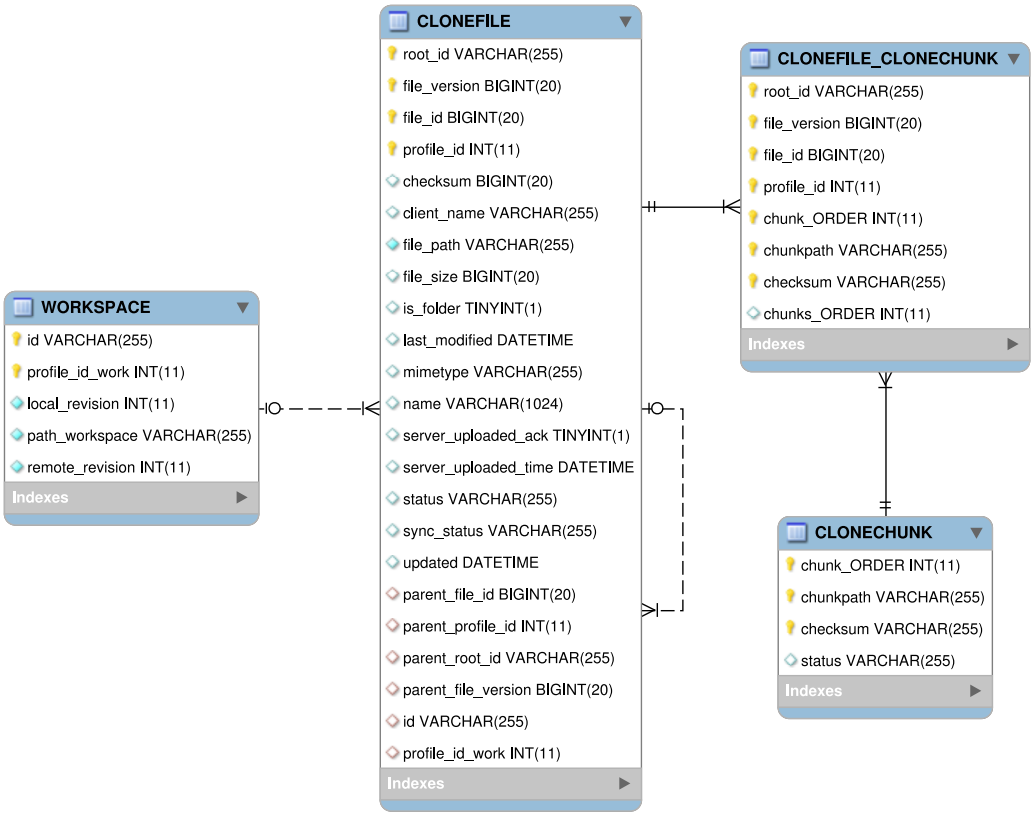


Figure 7: Database model of the StackSync desktop client

To achieve the data persistence we use JPA (Java Persistence API), so that classes are directly mapped into the database.

- **CloneChunk.** It keeps track of the chunks created by the client. The `path` indicates the location in the file system and the `checksum` is calculated over the chunk data.
- **CloneFile.** This table contains information about files located in the synchronization folder, including the path, checksum of the file, internal identifier, size, an flag to indicate if it's a file or a folder, the synchronization status, etc.

- **CloneFile_CloneChunk.** This table contains the relation between files and chunks. We need to know what chunks form each file and their order to be able to recover the file later.
- **Workspace.** It contains information about the workspaces to which the user is allowed to access.

3.5 Synchronization service

The synchronization service, namely `SyncService`, is the most important and critical part of the synchronization architecture. It is charge of managing the metadata in order to achieve data synchronization. Desktop clients communicate with the SyncService for two mainly reasons: to obtain the changes occurred when they were offline; and to commit new versions of a files.

When a client connects to the system, the first thing it does is asking the SyncService for changes that were made during the offline time period. This is very common situation for users working with two different computers, e.g., home and work. Assuming the home computer is off, if the user modifies some files while at work, the home computer will not realize about these changes until the user turns on the computer. At this time, the client will ask the SyncService and update to apply the changes made at work.

When the server receives a commit operation of a file, it must first check that the metadata received is consistent. If the metadata is correct, it proceeds to save it to a database. Afterwards, a notification is sent to all devices owned by the user reporting the file update. This provides StackSync with real-time in all devices as they receive notifications of updates and are always synchronized. However, if the metadata is incorrect, an notification is sent to the client to fix the error.

3.5.1 Metadata persistence

All metadata is stored in a database. We have created the following models:

- **User.** It corresponds to a user in the StackSync system. It contains information related to accounts: email, storage quota, etc.
- **Device.** A physical device owned by a user in which StackSync has been installed. A user may have more than one device, e.g., the home and work computers, a laptop, a mobile phone.
- **Workspace.** By default, each user has one workspace, which corresponds with the root synchronized folder. When a user shares a folder with other users, a new workspace is created in the scope of the shared folder.
- **Object.** A folder or a file. A workspace contains objects. It contains information such as the file name, path, type.

- **Version.** An object has different versions that are created as users modify files and folders. Versions are submitted by devices. It contains information about the specific version such as the size, checksum, modification date.
- **Chunk.** A version is composed by a set of chunks that form a specific object. A version will have one or more chunks if it is a file, or none if it is a folder.

With this information we have created the database model observed in Figure 8.

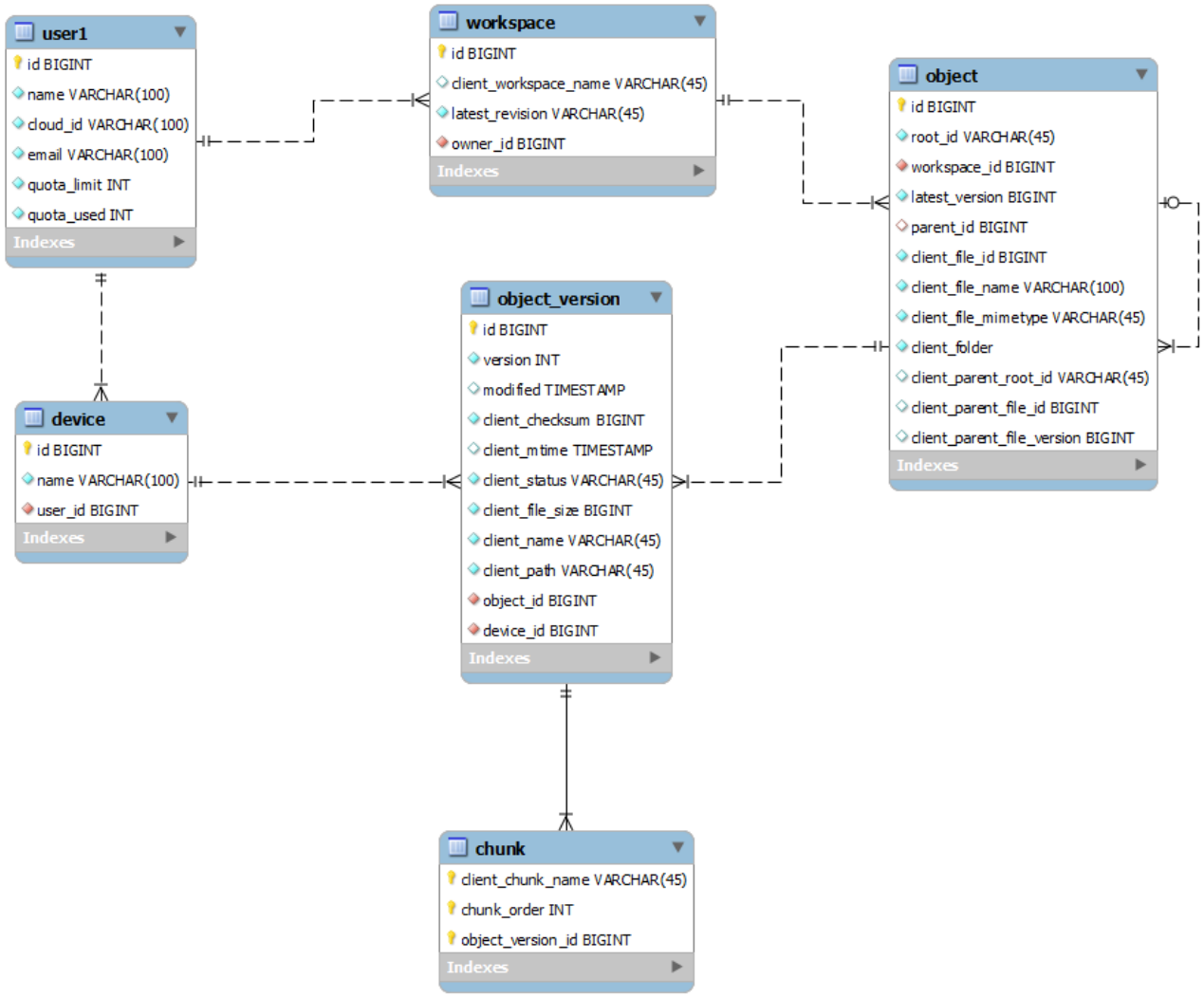


Figure 8: SyncService database model.

To access or modify values stored in the database from the code we implemented the design pattern DAO (Data Access Object). With this pattern we are able to create a common interface between the application and the database. Furthermore, it also creates a factory in which we specify what kind of database we want to use (e.g., PostgreSQL, MySQL). This provides us with a modular persistence system, in which we can add different databases just implementing the DAO interfaces without modifying the code.

In Figure 9 we can observe the class diagram that corresponds to the database DAO.

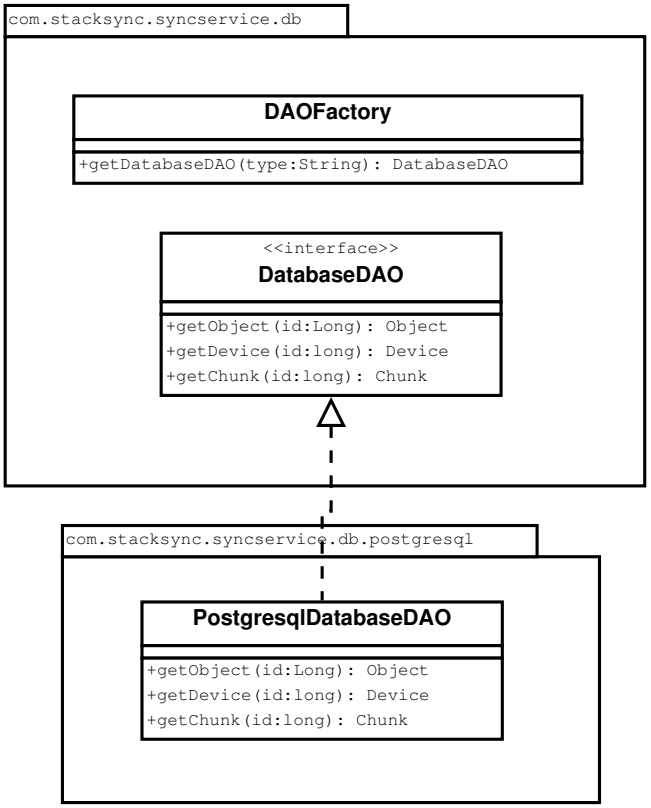


Figure 9: Class diagram of the database DAO in the SyncService.

3.5.2 Metadata processing

When the SyncService receives a commit operation, it must process the received metadata in order to check if the version is correct and there is no conflict. Algorithm 3 reports the pseudocode of the `commitRequest` operation. When a `commitRequest` message is received in the global request queue, the ObjectMQ middleware will invoke the appropriate `commitRequest` method in the SyncService.

This method then receives a proposed list of change operations in a concrete Workspace. For every change operation, it will then check if the current version of the object in the `Metadata backend` precedes the change proposed by the client. In this case, the changes are (transactionally) stored in the `Metadata backend` and confirmed in the `CommitEvent`. If there is a conflict with versions, the `commitRequest` is set as failed and information about the current object version is added to the `CommitEvent`. The reason for adding the current object version to the `CommitEvent` is to piggyback the information about the "differences" between the two versions, such that the "losing" client can identify the missing chunks and reconstruct the object to the current version. As usual, in StackSync, a conflict occurs when two users change a file at the same time. This implies that the two clients will propose a list of changes over the same version of the file. The first `commitRequest` to be processed will increase the version number by one, but the second `commitRequest` will inevitably propose a list of changes over a preceding version, resulting in a conflict.

To resolve the conflict, the `SyncService` adopts the simplest policy in this case, which is to consider as the "winner" the client whose `commitRequest` was processed first. This way,

Algorithm 3 Pseudocode of the commitRequest function in the SyncService

```
1:  function COMMITREQUEST(workspace, List < ObjectMetadata > objects_changed)
2:      commit_event ← new instance of CommitEvent
3:      for new_object in objects_changed do
4:          server_object ← metadata_backend.get_current_version(new_object.id)
5:          if not exists server_object then          ▷ To commit the first version of the new object
6:              metadata_backend.store_new_object(new_object)
7:              commit_event.add(new_object, confirmed = True)
8:          else if server_object.version precedes new_object.version then
9:                                                    ▷ No conflict, committing the new version
10:             metadata_backend.store_new_version(new_object)
11:             commit_event.add(new_object, confirmed = True)
12:         else
13:                                                   ▷ Conflict detected, the current object metadata is returned
14:             commit_event.add(new_object, confirmed = False, server_object)
15:         end if
16:     end for
17:     trigger_event(workspace, commit_event)
18: end function
```

the `SyncService` avoids rolling back any update to the `Metadata back-end`, saving time and increasing scalability. At the client, the conflict is resolved by renaming the "losing" version of the file to "...(Conflicted copy)".

In Figure 10 we can observe a conflict scenario. In this case, two different users try to upload a different version of the same file. The SyncService sequentially receives their `commitRequest` and returns a positive response to the first processed request (User 2) and a negative response to the second (User 1). User 1 will have to rename its version and download the correct one from the server.

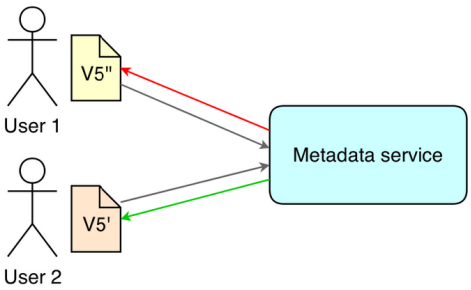


Figure 10: File conflict

Finally, the `CommitEvent` will be triggered to the Workspace AMQP Exchange, and it will be received by all interested devices in their incoming event queues.

As just elaborated above, note that the `CommitRequest` is an important operation in the sync service since it has to provide *scalable request processing, consistency, and scalable change notification*. Scalable request processing is achieved because the method is *asynchronous* and *stateless*. Multiple `SyncService` instances can listen from the global request queue and the message broker will transparently balance their load. Consistency is achieved using the transactional ACID model of the underlying `Metadata back-end`. Finally, scalable change notification to the interested parties is achieved using one-to-many push notifications (@event).

The SyncService interacts with the database using an extensible DAO explained in the previous section. Our reference implementation is based on a relational database although the system is modular and may be replaced easily.

3.5.3 Communication

An important design decision of our reference implementation was to rely on a messaging middleware for communication. Since Personal Cloud storage services exhibit significant read-write ratios [5], we decided that the sync engine should support *persistent client connections, push-based notifications, and asynchronous and stateless interactions*. A message-oriented middleware fits well with these requirements, because of its support to *loosely coupled communication* among distributed components thanks to *asynchronous message-passing*.

The communication system must guarantee safety, speed, and above all, scalability, because the server must attend many requests. For this reason, we decided to use a message-oriented middleware called ObjectMQ. Developed at the URV, ObjectMQ is a lightweight remote object layer constructed on top of a messaging middleware compatible with AMQP. In our case, it is running on top of RabbitMQ.

ObjectMQ is using a global request queue for the SyncService, a response queue for each device (SyncService Proxy), and a fan-out Exchange for each workspace. Each device will bind its request queue to the appropriate workspace Exchange to receive notification changes in this workspace. In any case, queue message programming is abstracted thanks to ObjectMQ, so that the protocol will be defined in terms of RPCs or method calls.

```
@RemoteInterface
public interface SyncService extends Remote {

    @SyncMethod(retry = 5, timeout = 1500)
    public List<ObjectMetadata> getChanges(Workspace workspace);

    @SyncMethod(retry = 5, timeout = 1500)
    public List<Workspace> getWorkspaces();

    @AsyncMethod
    public void commitRequest(Workspace workspace, List<ObjectMetadata> objectsChanged);

}
@event
public interface CommitEvent extends Event {

    public List<ObjectMetadata> objectsChanged getChanges();
}
```

Figure 11: SyncService interface

In Fig. 11 we can see the interface definition of the SyncService. Clients can request the list of Workspaces they have access to with the *getWorkspaces* operation. Once the client obtains the list of Workspaces, it can then perform two main operations: *getChanges* and *commitRequest*. Furthermore, the client will be notified of changes by means of the event *CommitEvent*.

getChanges is a synchronous operation (@sync) that StackSync clients perform on startup. This is a costly operation for the SyncService as it returns the current state of a Workspace.

Once the client receives this information, it registers its interest in receiving committed updates, i.e., *CommitEvent*s (@event) for this Workspace. From that point on, any change occurring on this Workspace will be notified to the client in a push style.

`commitRequest` is an asynchronous operation (@async) that clients employ to inform the `SyncService` about detected file changes in their Workspaces. This is a costly operation since it must guarantee the consistency of data after the new changes.

`CommitEvent` is triggered by the `SyncService` in an asynchronous one-to-many operation (@event) to all out-of-sync devices in the specified Workspace. This operation is only launched by the `SyncService` once the changes has been correctly stored in the `Metadata back-end`.

3.6 Storage API

Including an API to StackSync will provide us with many benefits. Otherwise, the access to the StackSync system would be limited to our only desktop client. The only way a user could access their content is installing our application on its computer and synchronize its whole repository, which could take some time depending on the amount of files.

By creating an API we pretend to have a wider reach and allow a third-parties to create new presentation layers like an application, a website, or a widget that interacts with the StackSync system. Therefore, an API is meant to distribute services and information to new audiences that can be customized to provide a new user experience.

Furthermore, an API would also allow StackSync content to be integrated or embedded with other services or applications. Thus, ensuring a smooth and integrated user experience, and relevant and up-to-date information, for the user. The information is delivered wherever it can be useful to them. In addition, as everything is changing so quickly, an API would help us to support unanticipated future uses. Making data available via API can support faster and easier data migration and improved data quality review and cleanup. All in all, an API provides a set of benefits, some of them are not even expected, that would increase the value of our product.

Next we will provide explain how the API should be built and what requirements have to be met.

3.6.1 Design

Following the REST principles, the API will be organized by resources. Therefore we needed to identify what kind of entities should be represented in the API. The most important entity in our system are *Objects*, which can represent files and folders. Each *Object* will have a unique identifier that can be used to access through the API.

We will make a differentiation between data and metadata. Depending on the scenario, we may need to display the information of a file (e.g. file name, size, type) or the actual file content (i.e. a byte array). Because of this, we require to create two different resources called *Metadata* and *Content*.

We have identified the following resources:

- **Account.** A user resource represents information about a StackSync user.
- **Metadata.** A Metadata resource represents the information of a specific file or folder. It will contain the following fields:
- **Content.** A `Content` resource represents the data in a specific file, that is, the actual bytes in the file.
- **Version.** A `Versions` resource represents a specific version of a file. A file version contains the same fields as specified in the `Metadata` resource.

Depending on the resource and the HTTP method, we can perform a series of actions. In Figure 12 we observe a use case diagram representing the actions that a user can make to the API.

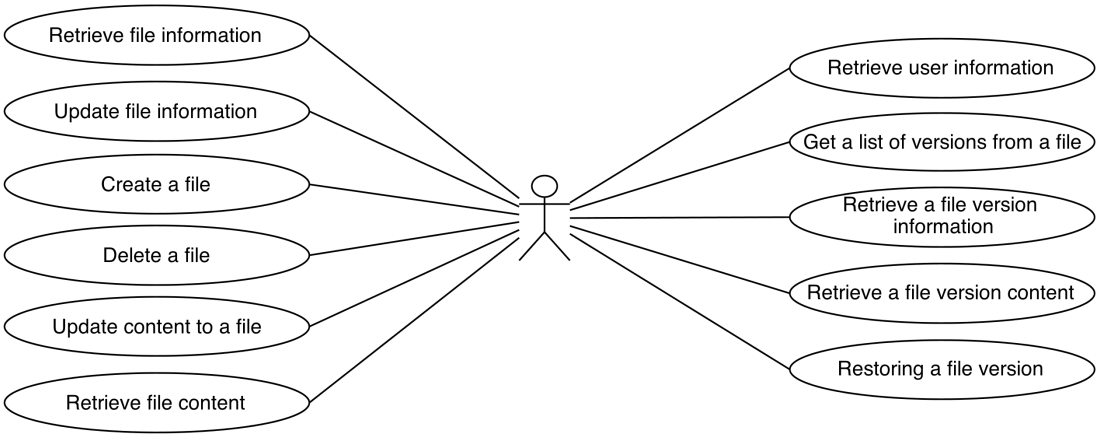


Figure 12: Storage API use cases diagram

Now that we have defined the API it is time to study how can it be integrated with StackSync. Considering the StackSync's architecture we realize that the API can be placed either on the *Metadata* side or on the *Data* side. Therefore, we are going to examine the pros and cons of each one.

On one hand, integrating the API on the *Metadata* side would imply to create a new service. This new service would have to handle all requests to the API, so in order not to be a bottleneck for the system we would have to provide some tools to allow scalability, adding an extra cost for hardware and maintenance. In addition, as the API has to handle not only metadata, but data as well, communication between Swift and the API would be very costly, increasing the bandwidth usage and probably congesting the network.

On the other hand, integrating the API on OpenStack Swift would allow us to include it as a plugin and add it to the Swift's pipeline. Including the API as a plugin to OpenStack Swift would give us some benefits. First, as it is integrated with Swift, we don't need any extra hardware, Swift will take care of our plugin and it will scale if it is necessary. Second, data send to the API will not be redirected anywhere else because the API is already inside Swift, saving important bandwidth. However, metadata needs still to be transmitted to the

synchronization service. But as metadata is negligible compared to the data itself and the synchronization service relies on a queuing system which allows it to scale, we finally opt for including the API as a plugin on OpenStack Swift.

So, let's know a little bit more about Swift.

As we can observe in Figure 13, OpenStack Swift is composed by at least one `Proxy` node and a set of `Storage nodes`. The proxy node is the entry point to Swift, so it is responsible of handling all incoming requests. Once the proxy receives a request it has to contact the `Ring`, which is the responsible for determining where data should reside inside the cluster of `Storage nodes`. The proxy also coordinates responses, handles failures and coordinates timestamps. A minimum of two proxies are recommended for redundancy, so in case of one server fails, the other one will take over.

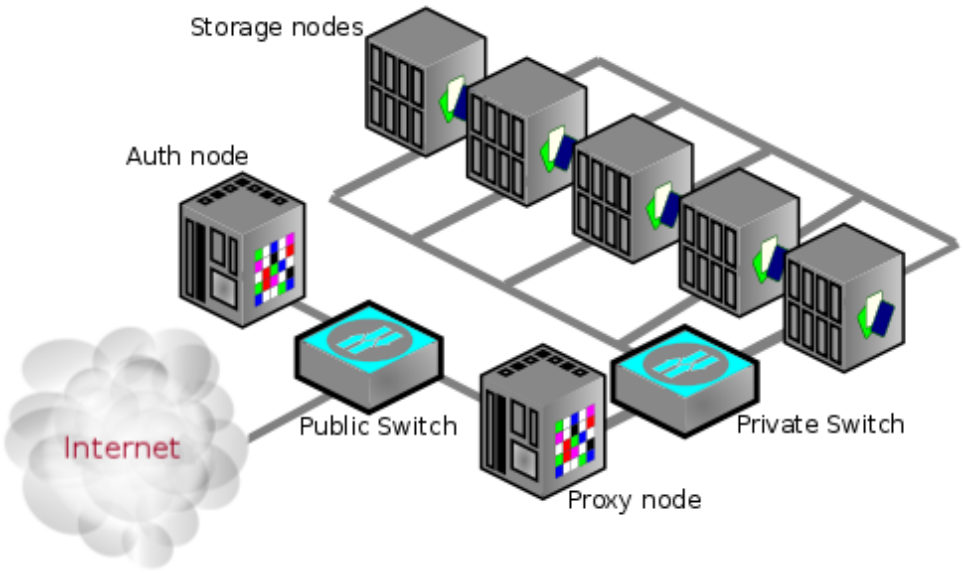


Figure 13: OpenStack Swift architecture

The `Ring` represents a mapping between the names of entities stored on disk and their physical location. When other components need to perform any operation they need to interact with the `Ring` to determine its location in the cluster. Each partition in the ring is replicated, by default, 3 times across the cluster. When a new storage node is added or removed from the ring it has to redistribute the load between the available storage nodes, ensuring that partitions are equally divided among them.

The `Storage nodes` are responsible for actually storing the data objects. It can perform operations to store, retrieve and delete objects on their local devices.

In OpenStack Swift, modules can be added as plugins in the `Proxy` node's pipeline. The pipeline is a list of middleware modules that filter the request before it is processed by Swift. By default, Swift comes with modules to provide services like basic authentication, data cache and logging. Each module can modify the request (e.g. adding a flag to indicate that the request is authenticated) and decide whether the request can continue its way to the next module or not, before it even gets to the `Proxy`.

```
[pipeline:main]
pipeline = [...] tempauth proxy-logging cache proxy-server
```

In our particular case, integrating our Storage API as a Swift module fulfills all requirements and facilitates the task of integrating it to StackSync.

3.7 Mobile client

Nowadays people are on the move and almost every one has a smartphone with Internet connection capable of accessing any kind of information from anywhere. That's why mobile apps are so important in today's market, and before releasing StackSync we need to provide a mobile app at least for one platform.

According to IDC [6], between Android and iOS add up to 92.5% market share on smartphone platforms. Concretely, Android surged to 79.3% and iOS is down to 13.2%. Therefore, it seems logical that the first mobile application should be done for Android.

3.7.1 Limitations

Mobile platforms come with a new set of interaction patterns and limitations that are not present when designing an application for PC or the web. Here we list the main limitations that we have to take into account when designing the mobile application for StackSync.

- **Storage space.** Although storage space in mobile phones is increasing in the last few years, it is still very limited if we compare it to a personal computer. StackSync account quotas are set arbitrarily by administrators. A standard account may have a 5 GB limit (as of Google Drive's accounts). Many mobile phones could not have enough free space to keep a copy of all user's files, in case the quota is used at 100%. And others simply may not be willing to dedicate so much space to files that are already safely stored in StackSync.
- **Network.** As mobile phones are, by definition, mobile, meaning that network signal can vary from one extreme to another, or even lose the complete connection, depending on the location. Moreover, smartphones still have lower network bandwidth as compared to the typical corporate network and high-speed Internet connections. Furthermore, charges may apply for people using the 3G network connection. For this reason, bandwidth usage must be as low as possible and the application must overcome sudden network disconnections and anomalies.
- **Performance.** Again, computational performance in mobile phones have been increased at a fast pace in the last years. But, as the CPU performance increases, so does the energy consumption. It is known that intense CPU usage decreases battery life drastically. Therefore, it is essential that our application makes an efficient use of the CPU to maximize battery life.
- **Screen.** A mobile phone screen is way much small than a monitor. This reduced space must be optimized to show concrete and clear information to the user. Moreover,

elements displayed in the screen must be large enough for the user to reliably touch the right element.

3.7.2 Requirements

The application will be only usable by users previously registered in StackSync and having a valid account. A registered user must be able to introduce its credentials to the application and start using it. The next time the user opens the application he must not be asked to introduce his credentials again. The user must be able to log out and unlink the application from the device deliberately. By doing this, the application must remove all remaining information about the user, including files, metadata and any other data.

The application must show a screen containing a list with the user's files and folders stored in StackSync. Icons must indicate whether it is a file or a folder. It is also recommended to visually distinguish different file types (e.g. images, music or documents). A title containing the name of the current folder must be shown on top of the application, this title will be changing as we move from one folder to another.

When entering a folder, the screen must be cleared and repopulated with the new folder's content. To go back, the user only has to push the back button. If a folder has no content, a text indicating that the folder is empty must be displayed. The user must be able to refresh the state of a folder on demand.

Users must be able upload their local files, download remote files into the smartphone, create folders, and delete files and folders. This actions must be done in an intuitive way and in the less steps as possible.

The application must save the state of visited folders for the user to examine it even if the device is offline. The user must also choose whether he wants to have a cache to save already opened files or not. In an affirmative case, he must indicate how many space is allocated to the cache.

3.7.3 Design

Unlike the desktop client, the mobile app will not synchronize a local folder into a remote repository. Synchronization would require the application to keep a local copy of the repository in the local file system, which is not feasible due to the limitations present on mobile platforms stated in Section 3.7.1.

The application will make use of the Authentication and Storage APIs documented in the previous sections in order to interact with StackSync. In Figure 14 we observe the interactions between the mobile application and other components. First, the application will obtain the Access Token and Token Secret from the authentication service. Afterwards, the application will communicate with the Storage API to obtain the metadata of the root folder.

In Figure 15 we observe an upload request for a new file to the Storage API service. In this case, the service must replicate some actions that we previously explained in the desktop client.

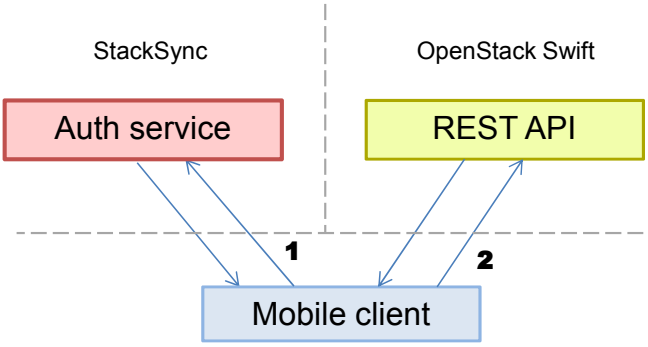


Figure 14: Mobile client interactions with auth service and storage API.

The Storage Service must ask the Chunker to split the new file. It must check with the SyncService for new chunks (user or global level), and it will then upload these new chunks to the Storage Back-end. It will also communicate with the SyncService to perform the commit operation. And this commit operation will trigger the notification of changes to all interested Desktop devices so that they can download the new chunks and obtain the current state.

Confidential information such as the user name, tokens or URLs will be stored in the `Shared Preferences`, which is used to store private data in key-value pairs. Android guarantees that this information is only accessible by the application. This allows the application not to ask for user credentials on every start. When a user logs out, the Shared Preferences is cleared.

The application will use a `SQLite` database to store information about file metadata received from the server. Each time the application downloads a file, whatever the purpose is, or gets a folder's metadata, the application will store the metadata in a cache table. In Figure 16 we can observe the database model.

Files and folders will be identified by a `node_id`. As files and folders can change over type, we also store the `version`. In order to retrieve a cached file or metadata, both the `node_id` and `version` must be the same. The `metadata` field will contain the actual metadata in JSON format, just like the API returns it. In case the user enables the data cache, the `local_path` will contain the path to the downloaded file in the local file system. The `size` is used to keep a record on the current cache usage and set limits.

The user will be able to set a limit to the cache to ensure that the cache does not grow indefinitely. When the user downloads a file and the cache is full, the first cached file is removed from the cache to make room for the newly downloaded file. If the size of the downloaded file is bigger than the one removed from the cache, the application will remove files in a FIFO (First in, first out) order until the file can be fitted in the cache without surpassing the limit.

Tasks that require considerable amount of time are performed in asynchronous tasks. Android has a special class called `AsyncTask` that will allow us to launch a task and receive a callback when its done. In Figure 17 we show a class diagram of the tasks that will inherit from `AsyncTask`.

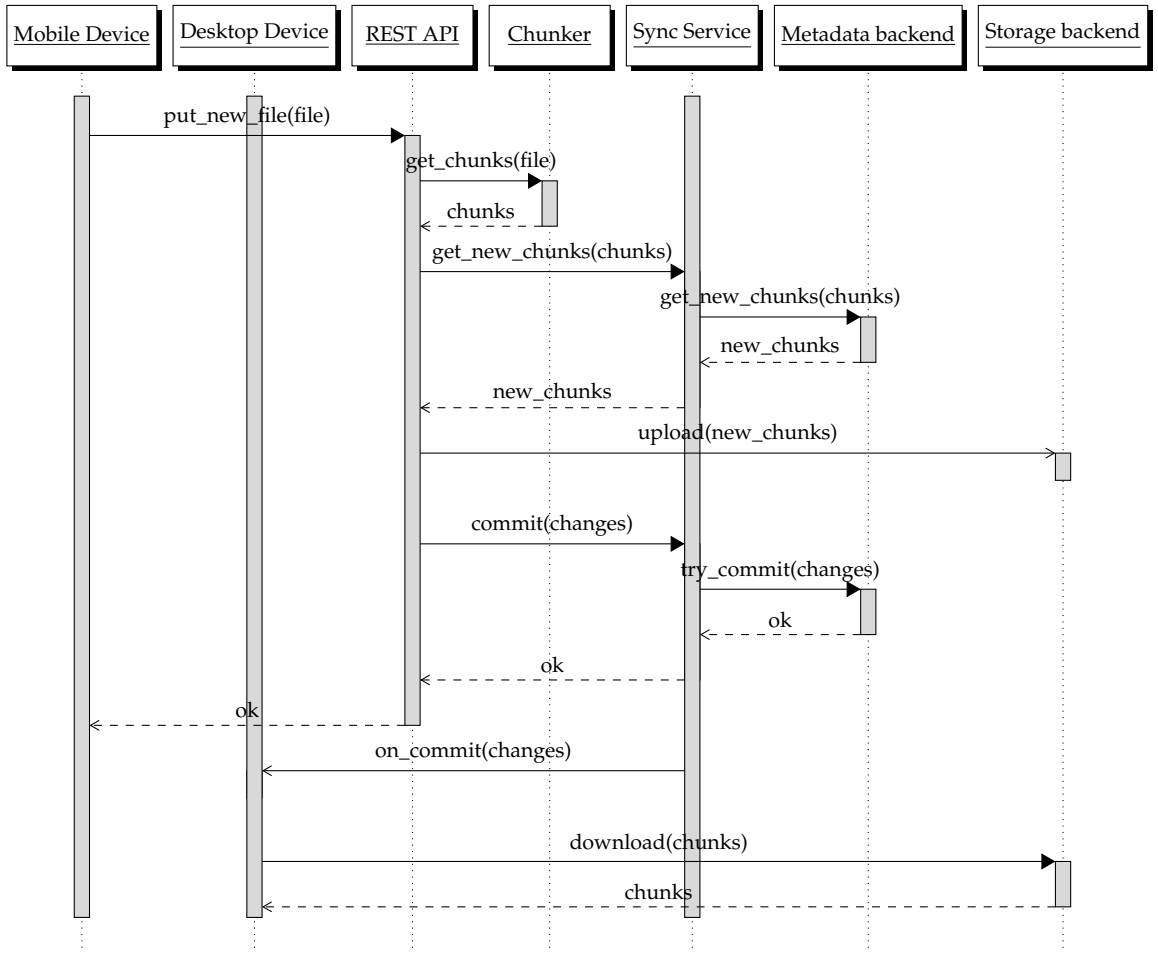


Figure 15: Mobile Client using the Storage Service Cloud.

These tasks return a response which can vary depending on the nature of the request. Most of the time we will only need to know whether the request has been completed or not, but on other cases we will also want to know the reason why it failed. In addition, other tasks return special values, such as authentication tokens or metadata. In Figure 18 we observe a `GenericResponse` class which has common attributes present in all responses, and specific attributes for special responses returned when logging in and listing folder's metadata.

We have identified three main user interfaces. Each user interface will translate to an Android `Activity`, which is focused on a thing that a user can do. An Activity interacts with the user through a UI.

As we observer on Figure 19, there will be three main UI. On the left we observer an `Activity` whose main focus will be to show the user a login form for him to provide his credentials.

On the center, and once the user is logged in, the main `Activity` will present the user a list of his files and folders synced with StackSync, entering and leaving folders will not change the `Activity`, instead the list will be repopulated to show the new content. Pressing on a folder will enter the folder. Pressing on a file will download and open the file. Long-pressing on a folder or file will show a list of additional options (e.g. sharing, exporting and

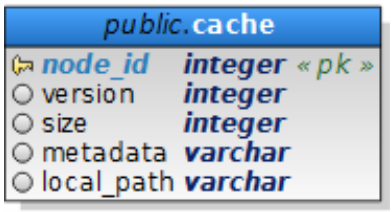


Figure 16: Cache database model.

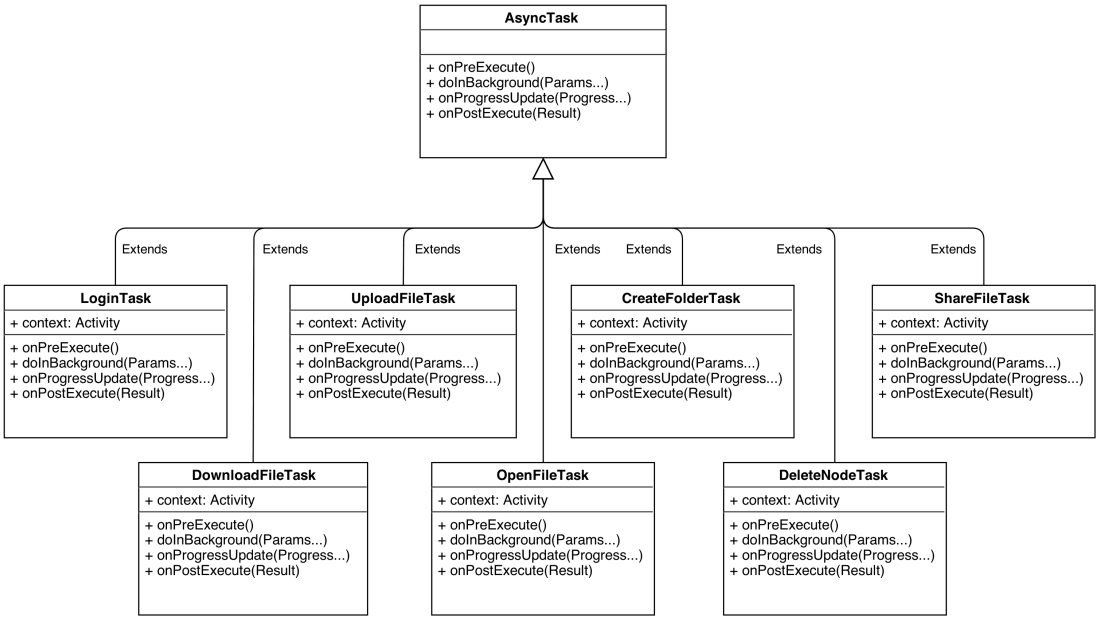


Figure 17: AsyncTask inheritance class diagram.

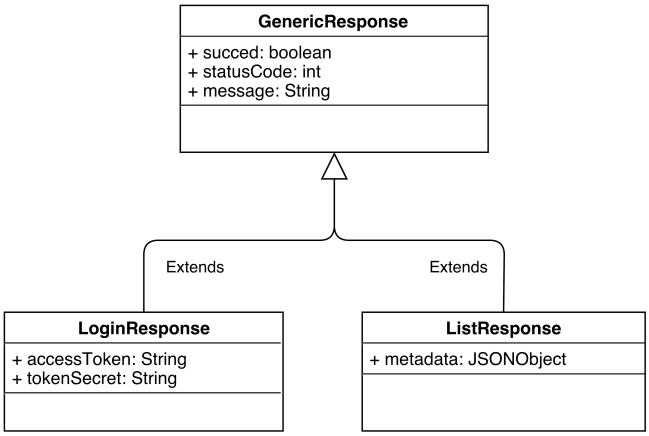


Figure 18: Response types class diagram.

deleting a file). To go back to the parent folder, the user can either press the back button or press the arrow on the top left of the screen. For the time being, uploading a file will use an external application to select the file.

On the right, the last `Activity` will be used to display and modify the application settings such as the cache or the user information.

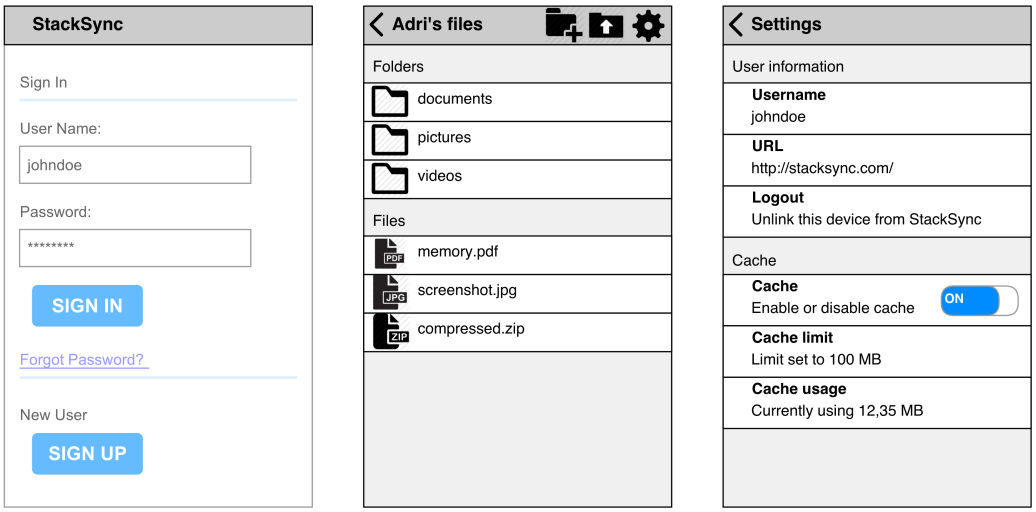


Figure 19: Design of user interfaces.

All three activities will inherit from the `Activity` class as show in Figure 20.

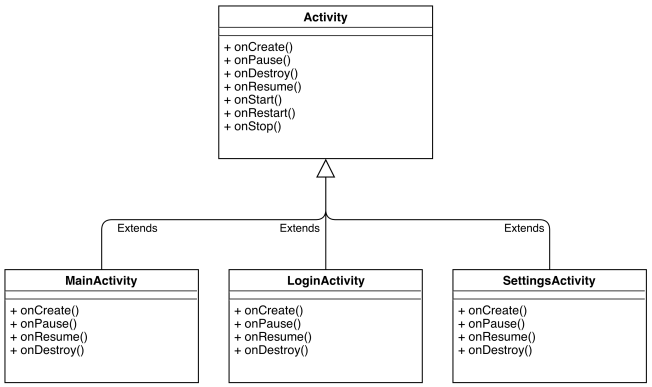


Figure 20: Activity inheritance class diagram.

4 Adaptive Personal Storage

4.1 Hybrid Personal Storage

4.1.1 Motivation and Benefits

Most of the personal cloud storage providers such as Dropbox, SygarSync, Box or others, ultimately store users' data on proprietary commodity cloud storage services such as Amazon S3. In this way, users are given no control whatsoever over where their data is stored and have no efficient means to migrate their data to a different commodity cloud service other than change the personal cloud storage provider altogether. Even then, chances are data are going to end up at the same commodity cloud storage provider anyway, causing data gravity and provider lock-in.

Clearly, data gravity and provider lock-in have serious implications for privacy, availability, performance and cost of using personal cloud services. To this end in WP3 of CloudSpaces (Cloudspaces Storage) we will focus on ways to provide adaptive replication and synchronization for personal storage infrastructure that will integrate different user and Cloud storage resources and avoid data lock-in. Our focal use case is specifically that of an SME that wishes to use (personal) cloud services, but wants more control over its own data. Such an SME, desiring to use public cloud service, typically owns some computation and storage resources in a private infrastructure. Our approach is to leverage private portion of these resources to store personal cloud metadata, to give a user control over data stored in a public cloud. We refer to this approach as Hybrid Cloud Storage (HCS).

Compared to existing cloud storage services, HCS will provide the better reliability, performance (e.g., latency), consistency, as well as inherent tolerance of untrusted cloud repositories. HCS will also allow dynamic cloud reconfiguration inherently fighting data gravity and provider lock-in. HCS will adapt to heterogeneous networks and topologies, leveraging geographical distribution of public commodity storage clouds.

4.1.2 Draft design and specification

High-level design of our Hybrid Cloud Storage (HCS) is given in Figure 21. HCS will export a key value store API [7], which will allow seemless integration with other Cloudspaces components, notably the StackSync personal storage client. The design sharply separates metadata from data. HCS metadata consists of least of the following: a) key version number, b) hash of the data, c) the set of clouds that store the latest version of data, d) the identifier of the replication/dispersal protocol used to store data across clouds in c). On the other hand, HCS will store data across (multiple) commodity storage clouds.

HCS metadata will be stored in a reliable fashion, without any single point of failure in a Replicated Metadata Service (RMDS). In our implementations RMDS will be stored on private and trusted resources (premises) of a user. Since HCS metadata contains critical information, it must be reliably replicated to avoid any loss of metadata. To this end our implementation will leverage Apache Zookeeper.[6] Our design reuses Zookeeper since it is

[6] http://zookeeper.apache.org.

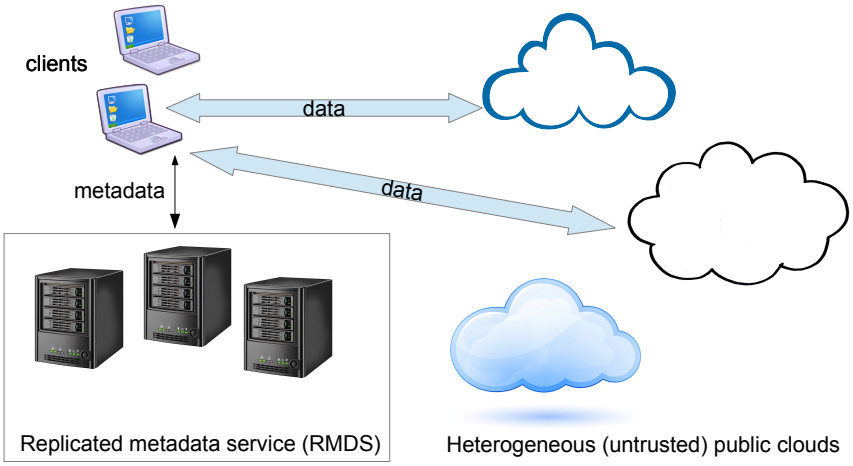


Figure 21: High-level design of Hybrid Cloud Storage (HCS). Metadata is separated from data and replicated within a replicated metadata service (RMDS). RMDS is envisioned to run on trusted premises (private cloud), although this is not required by the design. Data is in turn stored on public, potentially untrusted, clouds.

a stable open-source readily available in open source Hadoop distributions.

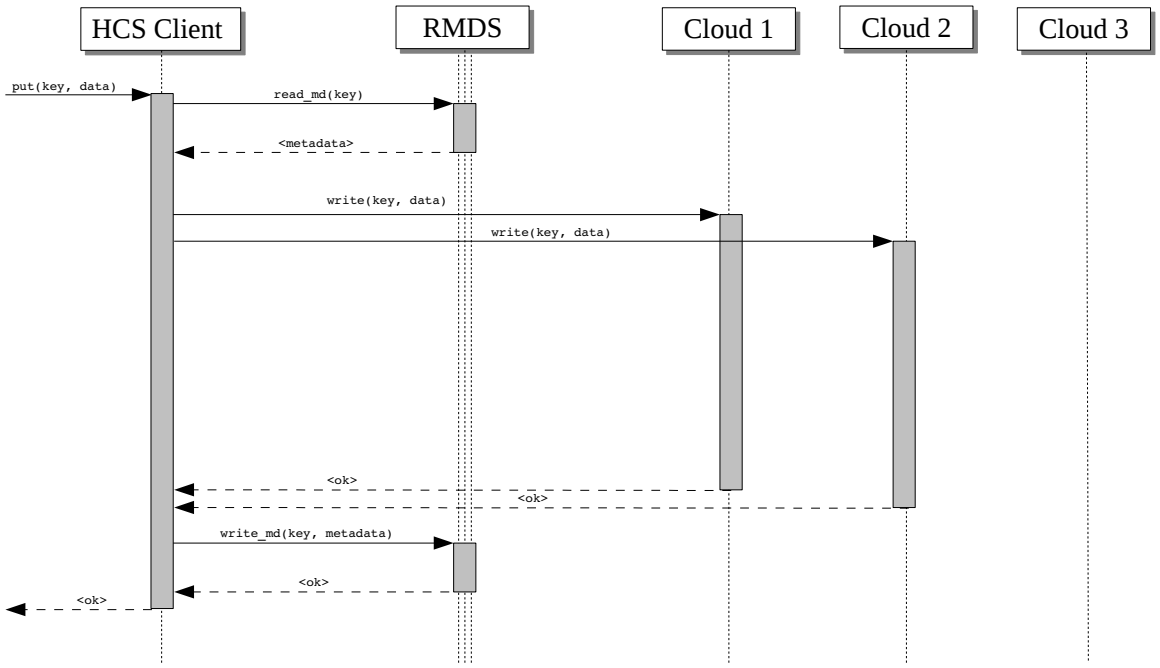
As an illustration, the sequence diagrams of HCS are depicted in Figure 22.

Recently, and in the context of the Cloudspaces project, we have shown [8] that this design has enormous potential when it comes to tolerating untrusted data repositories (more specifically, these are in the case of HCS, public clouds). Namely, HCS design that separates data from metadata offers reliable storage despite data integrity violations of untrusted repositories at a fraction of the cost born by similar state-of-the-art solutions [9] in terms of cryptographic overhead, performance and storage cost. Essentially, the design allows tolerating untrusted repositories virtually for free (compared to assuming trusted data repositories), i.e., by only paying a price of cryptographic hashes which are anyway practically always performed on cloud data for checks on inadvertent data corruptions.

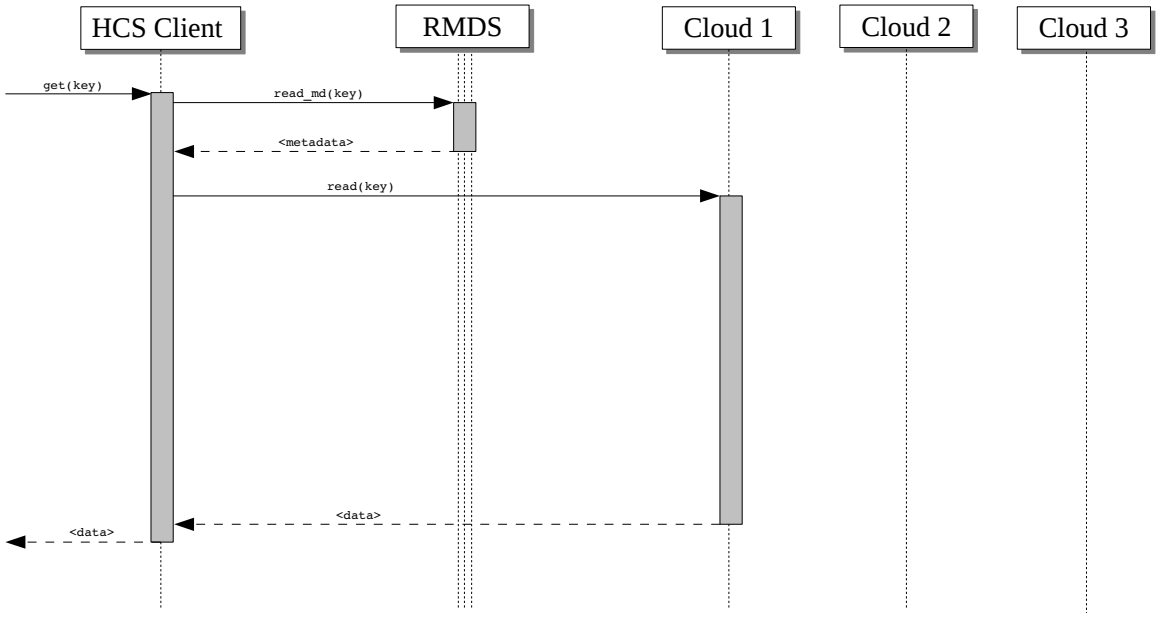
Our HCS design does not strictly require RMDS to be run on trusted premises [8]. Yet, we believe that it is realistic to assume that resources on private premises (as in our SME use case, which is our focal use case) are trusted. In case this assumption is not realistic and RMDS should be deployed in untrusted environment, we have developed a separate strongly consistent protocol that can be used for the purpose of HCS metadata storage [10].

4.2 Adaptive content distribution

Most of the cloud storage providers such as Dropbox, Amazon S3 or Box use HTTP as the transfer protocol to upload or download files. As it is being demonstrated, HTTP protocol and REST APIs are working quite well and are being useful to create scalable architectures. But, what if hundreds of clients want to download the same file? This file has to be uploaded as many times as the number of clients and this could become a bottleneck for the service provider.



(a) HCS put sequence diagram.



(b) HCS get sequence diagram.

Figure 22: Sequence diagram of Hybrid Cloud Storage (HCS) put and get methods. RMDS is replicated, yet appears to the client as a single, centralized metadata service.

Here will take the opposite tack and use BitTorrent to offload storage servers from doing much of the serving. The idea is that the storage service monitors the activity of users and upon detection of a certain critical mass, it transparently generates a torrent file to switch to BitTorrent and reduce its bandwidth contribution. Everything without user intervention.

With this approach, we can minimize the bandwidth consumption on the cloud provider's side while ensuring a relatively good download speed for the clients.

4.2.1 Architecture diagram

Our architecture contains three well differenced parts, two of them are in the cloud part and the other is in the client side:

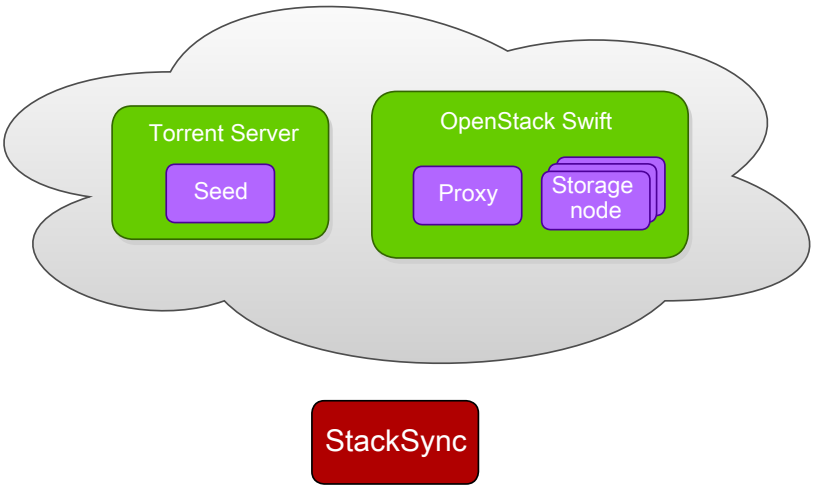


Figure 23: Architecture diagrams.

StackSync StackSync is a client that provides storage, syncing and sharing capabilities among devices and users in an easy and transparent way. StackSync clients' files are stored in a remote storage service and they can be accessed from any device with a StackSync client. By default, the HTTP protocol is used to upload or retrieve files. But, StackSync also includes a BitTorrent client to be used when a popular file is being downloaded.

OpenStack Swift Swift is a highly available, distributed and scalable object store. Apart from using it as the storage service, we have developed a new module that monitors the amount of clients requesting the same file at the same time. If there are few users downloading the file, Swift proceeds with the file transfer using the HTTP protocol. Otherwise, if the number of clients is bigger than a threshold, Swift automatically decides to switch to the BitTorrent protocol to offload itself from doing all the serving.

Torrent Server The role of this server is to create a torrent file that will be distributed to the interested clients and launch a false seed to provide pieces to the clients (peers). Initially,

the seed will have no pieces and as peers perform their requests, it will download the pieces on demand from Swift. Once a piece is served, the seed can save it on the disk to avoid requesting it again from the cloud in case that piece is requested again from another peer.

4.2.2 Download scenario

All the components of the architecture exchange information between them. Fig. 24 illustrates the sequence of actions.

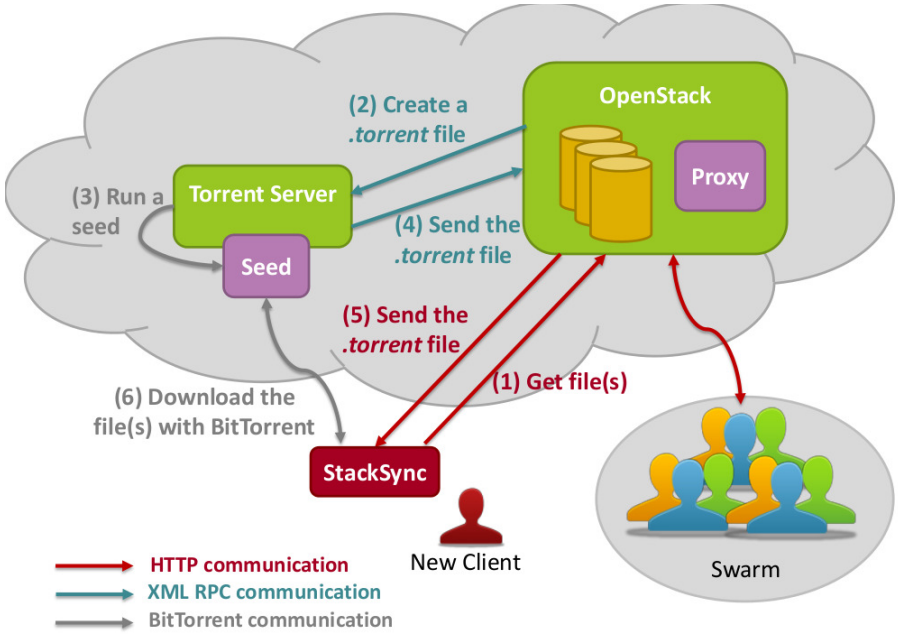


Figure 24: Download scenario.

First of all, it is important to take a look at the different types of communications that the architecture involves:

- **HTTP**: Used between StackSync and OpenStack Swift to exchange files.
- **RPC**: The BitTorrent module of Swift uses a RPC to communicate with the Torrent Server.
- **BitTorrent**: Used in the communications between StackSync BitTorrent client and the Torrent Server, more specifically between the client and the false seed.

To explain all the sequence of actions performed when a change to BitTorrent protocol is needed, it is necessary to imagine a scenario where there are N-1 clients (where N is the threshold needed to switch to the BitTorrent protocol) downloading the same file at the same time. Each user requests pieces using the REST API, so the server is responding almost the same data to all the clients.

In a certain moment, a new user requests the same file that the rest of the clients are downloading (1). The BitTorrent module hosted in the proxy, then, realizes that the threshold has been reached and asks the Torrent Server RPC to create the torrent file (2). After the

Torrent server getting all the metadata about the file and creating a corresponding .torrent file, the Torrent server runs a seed ready to serve StackSync BitTorrent peers (3). Later, the .torrent is returned to Swift module (4) and transmitted to the client (5). The HTTP response that the client receives notifies him to switch to the BitTorrent protocol. The client launches the BitTorrent client and reads the torrent file from the body of the HTTP response (6).

Finally, when another client tries to download another piece of the file, Swift will notify him that file is going to be downloaded using BitTorrent instead of HTTP.

4.2.3 Validation and tests

The aim of this architecture is to improve content distribution in different ways:

- Reduce cloud traffic and costs.
- Improve download time if possible.

We will create two different scenarios to test the architecture performance. In the first one we will use pure HTTP download and in the second one our novel BitTorrent implementation. In both test we will analyse the client and the server behaviour, this implies to capture the bandwidth and the download times.

In order to understand how our architecture behave in different situations, we will run all the tests with different parameters. The aim is to get the optimal values for the best performance. Some of these parameters are presented below:

- **Threshold**: Depending on the cloud performance, the number of simultaneous clients downloading the same file using HTTP protocol could vary. The key is to find the optimal moment to switch to BitTorrent without worsen the download speed.
- **File sizes**: It is necessary to test which is the minimum file size to use the BitTorrent protocol.
- **Clients** upload and download **limitations**: Not all the clients have an unlimited bandwidth. We will test what will happen with different clients limitations.
- The **clients' arrival model** (Flash-crowd and Poisson-attenuated distributions): The clients' arrival mode is an important parameter to consider in order to test the behaviour of the system. On the one hand, the flash-crowd is widely used to simulate the clients' connection behaviour in a typical BitTorrent swarm when there is a popular content. On the other hand, Poisson distribution could be another useful connection behaviour.
- **Different disconnection times** for the clients when they finish the download (seeding time): It is not the same if a client disconnects when has already downloaded the file or if he seeds the file for an interval of time.

Once all the data has been acquired, the results will be analysed to understand how the system evolves and which are the best parameters in each case.

5 Privacy-aware data sharing

5.1 Introduction

Cloud computing has become an essential part of people's electronic life. Services such as online file storage, collaborative document editing, music streaming, and photo browsing are just examples of what the traditional users are utilizing in their everyday life for personal or professional purposes. With the increased dependency on the cloud as a medium for storing and managing the data users share, the cloud has become a target for collecting users' private information. Several cloud providers consider this data as a monetization tool, producing revenue through targeted advertisements' provision. Governmental agencies consider such citizens' information as a valuable means of surveillance. Recently, news were leaked about PRISM, a secrete comprehensive surveillance program operated by the National Security Agency (NSA) in the United States [11]. PRISM allows the NSA to have access to specific users' data hosted by major computing companies, such as Google, Yahoo, Facebook, etc. Accordingly, users' data is no longer sought exclusively by external hackers or disgruntled employees; entities who were expected to be guarding this data are now its primary seekers!

5.2 Challenges

In particular, we notice three major stumbling blocks towards privacy provision in the cloud:

a) Privacy vs Services Dilemma: Making data items private implies that certain clouds services become unavailable. Put another way, a naive solution in which all data items are encrypted is not suitable.

b) Effort required in manually specifying each data item's privacy level: Users cannot be expected to manually specify privacy settings for all items. A user has too many data items that she would like to share on the cloud, and it is unrealistic to suppose that she has the time, resources and expertise to individually assess each data item's privacy level.

c) General lack of awareness about privacy: This includes limited notions about privacy being restricted to hiding 'important' content, such as personal identification numbers, credit card details, etc. Often context (semantics) can be more important than the content itself. E.g., metadata associated with the data item, the location and device from which the item is shared, the entity with whom the data is shared–all this information can be equally important as the content of the data itself. Furthermore, regardless of context and content, people are generally not aware of privacy risks posed by their online data sharing activities [12].

In order to overcome the above hurdles, we present a conceptual framework for privacy risk evaluation that includes: **1)** Risk evaluation based on the sharing context (and not just content) of the data items; **2)** Aggregation (crowd-sourcing) based techniques to inform individual ignorance; and finally **3)** An automated solution that determines privacy risk scores based on (1) and (2) using psychometric models.

5.3 Conceptual Approach

We present a centralized solution that can be implemented separately or by the cloud provider. The first major feature in our framework is *Semantic Vocabulary*. We propose that all items that are to be shared on a given cloud platform can be represented by a Semantic Vocabulary. This vocabulary ideally encapsulates the entire context in which an item is shared. For instance, an item could be an *Excel Document* made at *Work* authored by *Me* and shared with *Family*. Here the words in italics define the context of this item. Concentrating on semantics/context is important because as has been noted [12], privacy of the same item (in syntactic terms) might vary in different semantic contexts. As a trivial example, a word document, which is made at work carries a different privacy risk when shared with colleague as compared to when the same document is shared with a friend who is not a colleague.

The second feature is *Crowd Sourcing Data Items and Policies*. People can share different data items (as defined by the semantic vocabulary) with different policies, where a policy is in the range **[0, 1]**.

The final feature of our framework is *Automated risk evaluation using psychometric functions*. On top of the aggregated crowd-sourced data, we apply psychometric functions to automatically calculate users privacy attitudes, item sensitivity and privacy risk.

Architecturally, our framework can be divided into two components, which we describe next.

Semantic-Policy Crowd Sourcer: This component is straightforward. We define a data item by what we call its semantic *vocabulary*. The vocabulary can differ depending on the type of cloud platform and setting. For example, for a personal cloud platform like Dropbox, the semantics vocabulary could include the following fields: *created_by, file_type, device_from_which_item_is_shared, shared_with* etc. An example of such a vocabulary can be seen in Figure 25 [7]. Reliance on context leads to a richer model of privacy risk.

As stated earlier, people can share different data items with different policies, where a policy is in the range **[0, 1]**. Here, **0** signifies no privacy and **1** signifies full privacy (as in encryption). A value between **0** and **1** would result when a user decides to make *some* of the features of an item private[8].

We assume that each time a user shares a data item with a certain policy in the cloud, this piece of information gets stored in the *Semantic-Policy Crowd Sourcer*.

Risk Evaluation Component: We apply psychometric models on top of the data in *Semantic Policy Crowd Sourcer* to accurately determine the sensitivity of data items and users attitudes towards privacy. Psychometric functions map psychological states to entities in the outside world. Any psychometric function can be used to determine these features. In our approach, we shall use item response theory. Item Response Theory has been fruitfully used in many fields including educational testing [13].

Risk can be evaluated based solely on the sensitivity of data items, or can also include other metrics such as Trust scores of people with whom data is being shared, etc. This

[7] The semantic vocabulary would differ depending on cloud type, e.g., it would be different based on organizational needs as in the case of private cloud(s) within organization(s)

[8] E.g., hiding *creation_date* out of five possible metadata fields would give a policy value of **1/5 = 0.2**

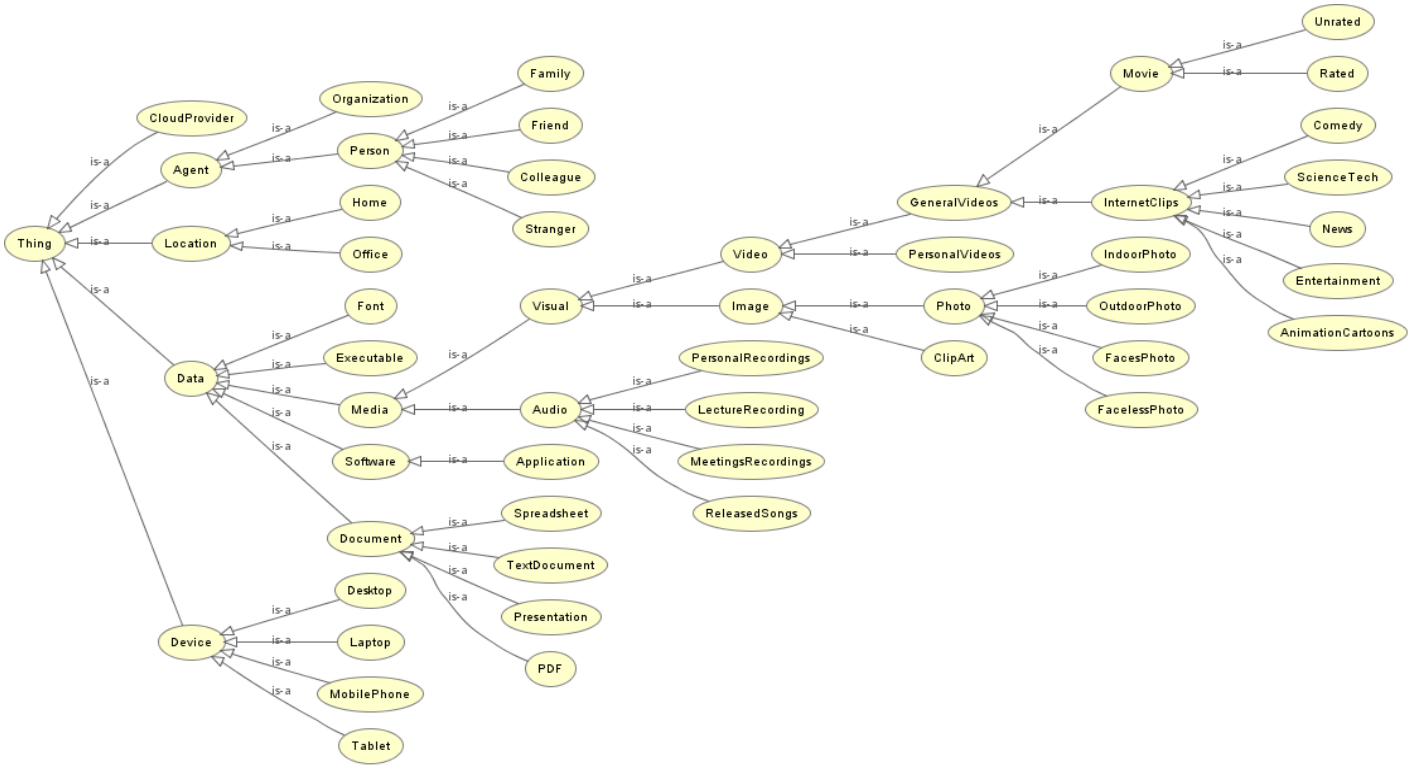


Figure 25: An example vocabulary for data sharing in the personal cloud

framework is depicted in Figure 26.

We shall undertake the following major tasks:

- We shall use a realistic vocabulary such as one depicted in Figure 25 for a personal cloud, and use it to create 'Human Intelligence Tasks' on the *Amazon Mechnical Turk*. We shall measure peoples responses against the 'Item Response Theory' and find a good fit. We hope to thereby demonstrate that Item Response Theory, a well-used psychometric model for educational purposes, can be used fruitfully in the cloud scenario.
- In our scheme, items are defined by their context. As we collect the contexts associated with shared items for evaluating privacy risk, we also have to ensure the privacy of this context information. We will use a scheme based on k-anonymity to anonymise a shared data item's context.
- We shall apply Item Response Theory to the cloud scenario to calculate privacy risk associated with sharing data items.
- We shall use the ENRON email dataset for our experiments. This dataset can give us a good model of users sharing activities and the diversity of data items (and their contexts)
- We shall also perform a set of experiments using synthetic data, with different graphs for user activities. Under both datasets (real and synthetic), we shall aim to show that our scheme bootstraps quickly and provides accurate privacy scores.
- We aim to identify risk mitigation scenarios for our prototype. See Section 5.4.4.

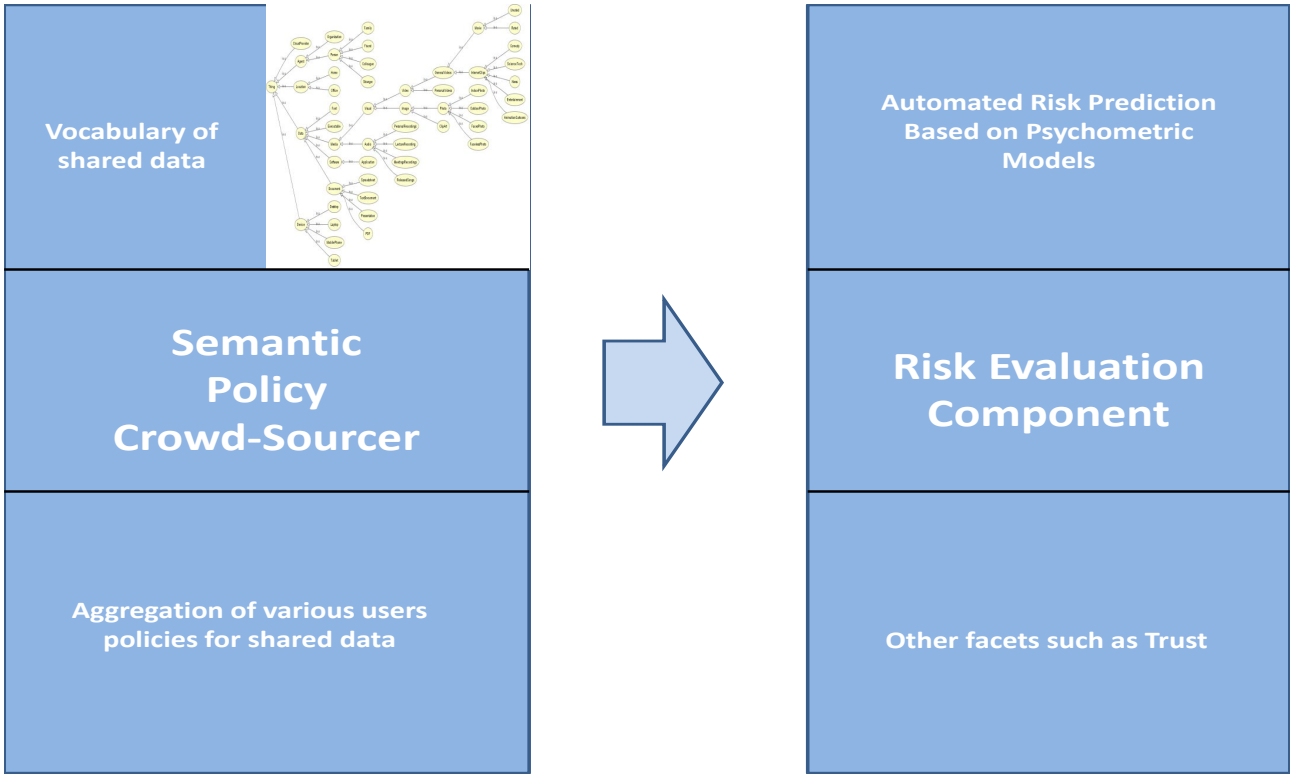


Figure 26: A componential view of our framework

5.4 System Architecture

5.4.1 Interacting Entities

We consider a system involving interactions between two types of entities: *end-users* and *cloud service providers (CSPs)*. A schematic of such system appears in Figure 27. The end-user can play one of two roles: *data sharer* or *data observer* while the cloud provider can only be a data observer. A data sharer is an end-user who decides to share *data items* he possesses. A data observer is any entity that is given access by the data sharer to observe the shared items. We assume that the user sends his data to a single CSP, called the *intermediary* that acts as the repository for this user's data. The user can select to give other CSPs access to his data through the intermediary. The interaction between these two types of entities is in the form of data sharing operations. Each such operation is initiated by an end-user u_0 who intentionally shares a data item d with a CSP c or another user u_1. Nevertheless, the data always flows from the data sharer through the intermediary CSP to the data observers. Additionally, the data sharer intends from the sharing operation to obtain a certain service of interest from the CSP, such as music streaming, document viewing, file syncing, etc.

The network is dynamic, in the sense that these entities can enter and leave the network, and the user items can be shared over time, not necessarily concurrently.

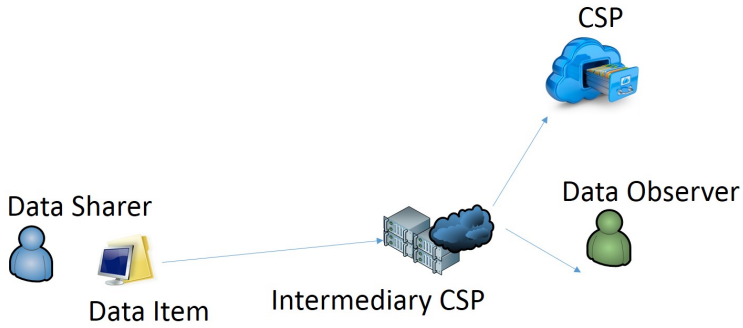


Figure 27: Interacting entities

5.4.2 Threat Model

We assume that the user is interested in hiding his sensitive data from the CSPs. Unlike existing approaches, we do not set an a priori definition of sensitive information due to the heterogeneity of the shared data items we consider. Existing privacy threat models consider an adversary who attempts at discovering specific personally identifiable sensitive information, such as location, browsing activity, credit card information, etc. In this work, we consider that the user can share any type of data with the CSP, where different privacy definitions hold for each type. To target that, we developed a context-based sensitivity definition, where the value of sensitivity of a certain item is calculated based on the sharing behaviours of multiple sharers. In this sense, sensitivity is derived based on users' sharing reactions to the data item.

5.4.3 Assumptions

We assume that, at each user's device, there is available information about the user's relationships with other users (possibly derived from online social networks or input by the user himself). Additionally, upon deciding to share a certain item, it is possible to locally derive various context information and metadata about the item, such as device information, file extension, file author, etc., which might vary according to the type of shared data.

We further assume that each entity k described in section 5.4.1 is associated with a trust score $T_{j,k}$ with respect to a user j, where $T_{j,k} \in [1, T_{max}]$. These trust values are assigned via techniques orthogonal to our system. For example, for end-users, the trust score can benefit from the social network of the user. For CSPs, the trust score can be built using reputation based techniques.

5.4.4 Problem Definition and Solution Sequence

We consider the privacy of end-users' data shared in the personal cloud scenario described above. In particular, we target solving two problems respectively: privacy risk estimation and privacy risk mitigation. In the first, we aim at quantifying the privacy risk numerically on a scale allowing comparing the risk of different sharing operations. By mitigating the

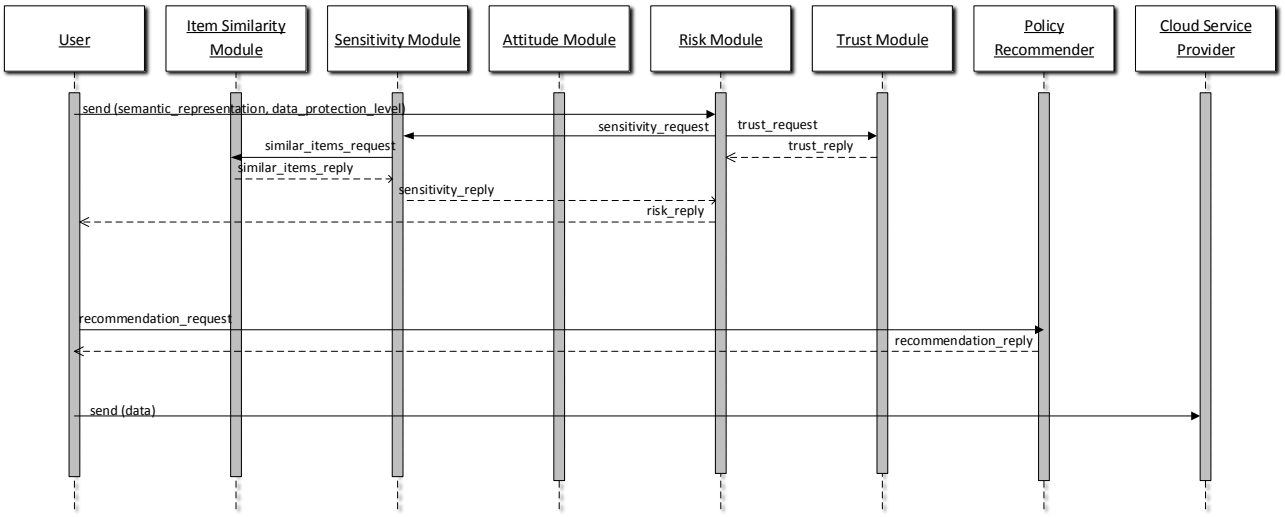


Figure 28: Sequence diagram of the system

risk, we mean designing a suite of methods for helping the user to guard his sensitive data. We also target designing techniques that require minimal user expertise and intervention so that the general user can easily benefit from them.

In sum, we aim at aiding the user to control the trade-off between minimizing exposed data on one hand and achieving the required cloud services on the other hand, without demanding a significant user expertise or effort.

In Figure 28, we show a sequence diagram summarizing the steps taken in one run of the system, including the privacy risk estimation and the policy recommendation steps. At the beginning, the user sends the semantic representation of the item to the risk module along with the needed auxiliary information, such as the data protection level. The risk modules queries the sensitivity module for the value of the sensitivity of the current item and the trust module for calculating the probability of disclosure. Then the item similarity module is queried by the sensitivity module, which replies by the items of similar semantic definition. After calculating the sensitivity, a reply is sent to the risk module, and the latter sends the computed risk value back to the user side. Afterwards, the users' device requests a recommended policy from the policy recommender, which replies directly to the user and allows him to send his data to the cloud service provider taking this policy into account.

6 Service platform

In this section we will provide an overview of the service platform of the CloudSpaces project, with a coherent, integrated, and high-level interface to all platform functionalities including authentication, storage, sharing, persistence, proximity and portability approaches. For detailed information about the platform, please refer to the deliverable document D5.1, which includes guidelines, open specifications, and documented best-practices for achieving syntactic interoperability of Personal Clouds, as well as the services to ensure both horizontal and vertical interoperability.

6.1 Authentication

Nowadays, the Internet is a very hazardous place where the minimum mistake will be exploited without hesitation, and CloudSpaces is no exception. Thus, before releasing the service platform it is mandatory to address the matter of security by means of an authentication mechanism. First of all, let's point out the differences between the concepts of authentication and authorization, concepts that will appear in the following paragraphs and are likely to cause misunderstanding.

On one hand, Authentication verifies who you are. A simple example is when you login into a website, you are proving that you are who you say you are because you, and only you, know the password. The basic idea is asking the user for some sort of secret information known only to the user. It may be as simple as a password, public key authentication, or other advanced systems based on tokens.

On the other hand, authorization verifies what are you authorized to do. For example, in a file system context, authorization can be controlled by an ACL (Access Control List), which is a list of permissions that specifies which users are granted access to objects. In this line, a user may be authorized to browse the folder */data* but not to access the folder */system*. Authorization must always occur upon successful authentication, otherwise it does not make sense.

Lacking an authentication mechanisms would mean that a malicious user could easily impersonate to any other existing user, gaining access to all its content stored in the system and compromising its privacy and, hence, the privacy of the whole system. For this very reason, adding an authentication mechanism will be crucial to preserve users' privacy.

Next we are going to examine the requirements for our authentication system.

- **Cross platform / cross language.** We need a global authentication service to be compatible with our current system specifications so we don't have to rebuild everything from scratch. The authentication service must integrate with the synchronization service, the newly created API, OpenStack Swift, desktop clients, and future mobile clients and third-party applications and services. Due to the heterogeneity of services that will interact with it, we need something language and platform agnostic so we don't depend on a specific technology. Some examples could be the HTTP or XML standards.

- **Transport-independent.** The platform must be installed in a large variety of hardware and network settings. These configurations may have some limitations that StackSync must overcome without losing neither security nor functionality. Therefore, an important requirement is that the authentication service must completely support an insecure channel such as HTTP.
- **User-agent.** It must allow other application and services to access Personal Clouds on behalf of a user, of course, with previous authorization from the user part. This would permit third-party developers to create all sort of applications (web-based, mobile, or desktop) on top of Personal Clouds.
- **User control.** The user must be aware what applications and services have been granted to access its files. The user must be able to revoke access to individual application without affecting the rest. A revoked application must immediately lose access to the user's resources.
- **Token-based.** As the authentication service can run over a insecure channel, we cannot send over plain passwords. Even on secure channels like SSL, it is not a good practice to send login credentials as username and password any time we make a request. Therefore, we require a system based on tokens and signatures that contain information to identify a particular user.
- **Simplicity.** By simplicity we mean that the authentication system should not be a fuzzy scheme hard to manipulate and integrate. We seek something clear, well-documented and adaptable to our needs. In general terms, an authentication system that wouldn't cost a headache to someone trying to understand how it all works.

Before designing an authentication mechanism from scratch, we reviewed some of the most important authentication protocols available to see if they fitted our requirements.

Basic authentication

This authentication method is the simplest one. Plus, it is the oldest and most common format for web authentication. It is what appears when you visit a web site and a window pops up requesting user's login and password. Every known web browser supports basic authentication. In basic authentication user's login and password is sent with every request, therefore, they must rely on SSL to create an encrypted tunnel between the client and server. Otherwise, if using a plain channel like HTTP, anyone could easily sniff the connection and obtain user's credentials.

As user's login and password are required in every request, client applications must store these credentials. So developers take on additional responsibilities for the secure storage of credentials. The potential harm to users if login credentials are leaked or abused is very high. Because many users use the same password across many sites, the potential for damage does not necessarily stop with their Personal Cloud account. Basic authentication has not ability to restrict a particular application from accessing after obtaining user credentials.

As explicitly stated in the protocol's name, basic authentication is a very basic protocol that does not meet most of our requirements. Therefore, it is discarded as an option for our authentication service.

OAuth 1.0a

OAuth authentication is a bit more complex than Basic authentication, though it solves the main problem of Basic authentication: in OAuth, user's login and password are never shared with any other application or service. It allows users to approve applications to access their content on their behalf without giving away their password. It solves the problem through a secure handshake.

In OAuth 1.0a there are three main players: the user, the consumer, and the service provider. A consumer can be an external application that wants to access the service provider. The consumer and service provider share a token, which is like a username, and a secret, which acts like a password. On every API call, the consumer must generate a signature with a set of parameters using the secret. The service provider must generate the same signature with the same secret, which is known only by the consumer and the service provider. Anyone sniffing the network could not obtain the secret because it is always encrypted. So it can work properly over a insecure channel like HTTP without compromising anything and, therefore, meeting our requirement of being a transport-independent protocol.

Moreover, the user can revoke access to authorized applications at any time. So far, OAuth 1.0a meets all our requirements, but let's check a newer version of OAuth.

OAuth 2.0

OAuth 2.0 was created to simplify some complex parts of the OAuth 1.0a protocol like signatures. Signatures where created in OAuth 1.0 to allow the protocol to be used over insecure channels. So, in OAuth 2.0 they removed signatures and forced all communication to be done over a secure channel like SSL. Limiting the network scenarios on which OAuth can be used.

As there are no signatures, the token secret is no longer needed and only one security token is used in OAuth 2.0. Access tokens expire in this version of OAuth, so eventually one request will fail due to token expiration and the consumer would have to renew it every time period.

All in all, the main advantage of OAuth 2 over OAuth 1 is in reduced complexity. Many widely known services (e.g. Dropbox, LinkedIn, or Twitter, to name a few) that have adopted OAuth 2, still support OAuth 1 due to its versatility and robustness. Let's not forget that OAuth 2 is still a draft specification and is subject to frequent changes. Therefore, OAuth 2 may be suitable for those service that can assure SSL communication and are willing to often modify their implementation, but as the service platform is meant to be transport-independent and we seek something stable and robust, it does not fit on our requirements.

Therefore, the only one meeting all our requirements is OAuth 1.0a.

In this part we will explain in detail how the OAuth 1.0a will work in the CloudSpaces platform and then provide some real use case scenarios that will be faced and addressed by the authentication service.

Authentication process

We will use StackSync as an example of Personal Cloud. As explained before, the OAuth protocol will enable applications and services (i.e. consumers) to access protected resources from StackSync (i.e. the service provider), without requiring users to disclose their StackSync credentials to consumers. A consumer can be either a third-party tool, or a StackSync's native application such as a desktop or mobile client.

Figure 29 shows the StackSync's authentication flow. The process can be roughly divided into the following three steps.

- **Step 1.** Getting the Request Token.
- **Step 2.** Getting user's authorization.
- **Step 3.** Exchanging the Request Token for an Access Token.

The first step to obtain authorization for a user is to get a Request Token using your Consumer Key. This is a temporary unauthorized token that will be used to authenticate the user to StackSync. This token/secret pair is meant to be used with in the next steps to complete the authentication process and cannot be used for any other API calls.

After getting the Request Token, an application needs to present the authorization page to the user, where they will be asked to give permission to the application to access their data. The authorization page will present the user with a list of permissions that the application is requesting.

When authorization is complete, StackSync will redirect the user back to the application using the `oauth_callback` specified with the Request Token. If the `oauth_callback` parameter is set to `oob`, the application must find some other way of determining when the authorization step is complete. For example, the application can have the user explicitly indicate to it that this step is complete, but this flow may be less intuitive for users than the redirect flow

If the user denies permission to the application, the Request Token will be automatically cancelled. A cancelled Request Token cannot be reused, so the application will have to obtain a new Request Token to get the authorization of the next user.

Once the user has authorized the application, the application can now exchange the approved Request Token for an Access Token. This Access Token should be stored by the application, and used to make requests to protected resources located in StackSync.

The Request Token and Token Secret must be exchanged for an Access Token and Token Secret. To request an Access Token, the Consumer makes an HTTP request to StackSync's Access Token URL.

After proper validation from the server, StackSync generates an Access Token and Token Secret and returns them in the HTTP response body. The Access Token and Token Secret are stored by the Consumer and used when signing Protected Resources requests.

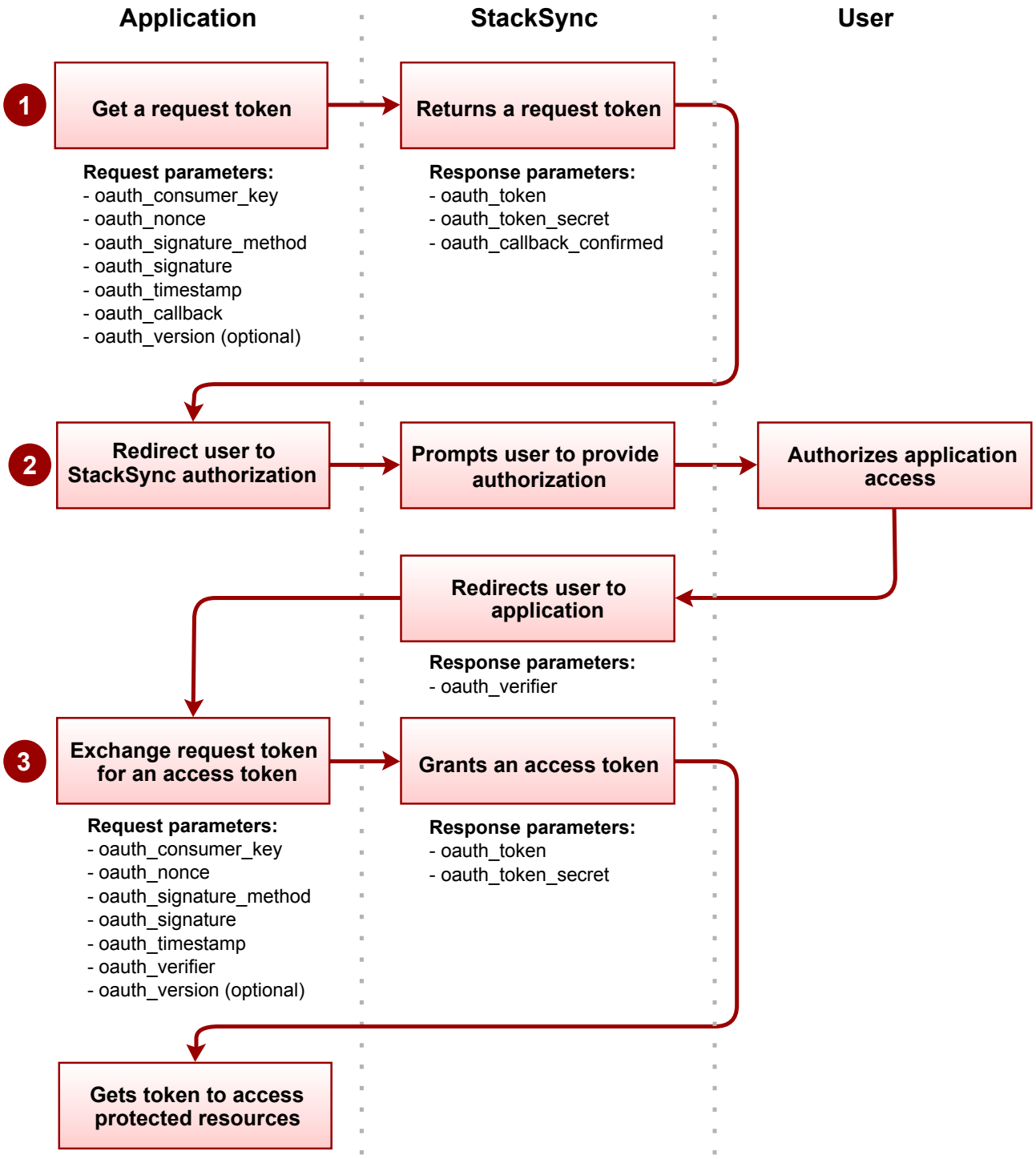


Figure 29: StackSync's authentication flow

Security considerations

There are some security issues that must be address to prevent illegitimate use of the authentication service.

For example, timing attacks are a real threat that all public service have to face. Failing to account for timing attacks could enable attackers to enumerate tokens and successfully guess HMAC secrets. For this reason, our implementation should keep the round-trip times (RTT) to the same length no matter whether the request is processed correctly or not. Dummy values will be used in the verification process for the running time to remain equivalent to the running time when given valid values.

Another essential security measure is the nonce/timestamp pair. A nonce is a random string, uniquely generated by the client to allow the server to verify that a request has never been made before and helps prevent replay attacks when requests are made over a non-secure channel. The nonce value must be unique across all requests with the same timestamp, client credentials, and token combinations. If we detect that a nonce/timestamp pair has been used before, we may just have detected a replay attack. Therefore it is an essential part of OAuth security that we do not allow nonce/timestamp reuse.

URL redirection is also a point that has to be taken into account. When consumers ask for a request token, they provide a callback URL for the user to be redirected upon successful authorization. StackSync requires all callback URIs to be registered before use, restricting which URIs may be submitted and making validation simpler and safer. Registration will be done in the Service Provider application registration web site.

Use cases

Third-party application/service use case

The authentication service will allow external applications and services to access the Service Provider. Here is an example of how this interaction will go. Let's say that Bob is the user, SuperStore is an application acting as the consumer, and StackSync is the service provider.

1. **SuperStore registers at StackSync as a service.** After successfully registering, SuperStore obtains a `consumer key` and a `consumer secret` that identify SuperStore on StackSync. It also obtains some other important information such as signature types and protocol endpoints supported by StackSync.

2. **Bob wants to use SuperStore with his StackSync's account.** Bob has a StackSync account with some personal files and he wants to access his content from the SuperStore application. SuperStore tries to access StackSync and gets an HTTP 401 Unauthorized access because it is private. SuperStore asks for a `request token` providing its consumer credentials. StackSync recognizes SuperStore as a valid registered application and returns a request token and token secret.

3. **Bob authorizes SuperStore.** SuperStore directs Bob to the StackSync's authorization page. In this page, Bob is asked if he is willing to grant SuperStore access to his per-

sonal files stored in StackSync. Once he accepts, Bob is automatically redirected to a callback site where SuperStore realizes that Bob has authorized the application.

4. **SuperStore gets an access token.** SuperStore has now an authorized Request Token that can be exchanged for an Access Token. SuperStore refers to the Access Token resource and obtains a valid Access Token and Token Secret.

5. **SuperStore access Bob's files**. SuperStore can now access to Bob's files using the Access Token and Secret. Bob will no longer need to provide his credentials unless he revokes access to SuperStore through the StackSync web site.

Native application use case

The authentication service will also be used in StackSync's native applications and services. Applications such as the desktop synchronization tool, the mobile application or the future web application. To this example, Bob is the user, the Desktop Client is a native application, and StackSync is the service provider.

1. **The Desktop Client is already registered in StackSync.** Unlike a third-party application, a native application is registered in StackSync and have a special Consumer Key to be recognized as a native app.

2. **Bob wants to keep his files synced with StackSync using the Desktop Client.** The Desktop Client wants to communicate with the synchronization tool in the name of Bob. The Desktop App asks for a `request token` providing its special consumer credentials. StackSync recognizes the Desktop Client as a valid registered application and returns a Request Token and Token Secret.

3. **Bob provides his credentials in the Desktop Client.** As a native app, the Desktop Client shows Bob a login form to provide his username and password. Once validated, the Desktop Client is automatically authorized.

4. **The Desktop Client gets an Access Token.** The Desktop Client has now an authorized Request Token that can be exchanged for an Access Token. The Desktop Client refers to the Access Token resource and obtains a valid access token and token secret.

5. **The Desktop Client communicates in the name of Bob**. The Desktop Client can now communicate with the synchronization service with the Access Token and Secret. Bob will no longer need to provide his credentials unless he revokes access to the Desktop Client through the StackSync web site.

6.2 Store

The store API is meant to consolidate a standard among Personal Clouds to achieve an easier interoperability and facilitate access to third parties. We will specify the different actions and resources available as well as the needed parameters to perform queries and the distinct error codes available.

This API will provide unified methods for call semantics including error handling, and the basic set of file and folder methods: retrieving file and folder metadata, retrieving file contents, creating new files and folders, deleting existing files and folders, and modifying existing files.

The store API will thus facilitate (perhaps even enable) the creation of cross-cloud clients. In particular it aims to facilitate:

Mobile clients Making an API that can be used from mobile devices limits some of the technology choices, as mobile clients will often have limits to their available runtime resources, local storage accessible to the application, and background processing time.

It is worth pointing out that we do not seek to enable *notifications* to be delivered to mobile devices, as there is not yet a single unified method for doing that (and is thus beyond the scope of this project).

Web clients We seek to enable people to easily create a web portal to multiple personal cloud providers. Here the limits are more related to trust and authentication, rather than resources. It is worth pointing out that the case we are considering is for the client software that would run on the server, with the resulting data provided as a service to the user; we are not considering a browser-side client at this time.

Desktop clients The desktop client is perhaps the least demanding target, because a desktop application can usually leverage all the facilities developed for web and mobile access. Ensuring that the authentication layer works well on both web and desktop clients is perhaps the component that needs the most attention, but our technology choice for authentication already ensures this.

As an example, a desktop client could be written that synchronises file and folder data to multiple cloud providers simultaneously. This would grant the user some redundancy, at the cost of additional bandwidth; a more interesting example, then, would be a desktop client and server-side client written to work in cooperation such that file data is sent to the server-side component once, and it acts as a middleware to the different personal cloud providers.

6.3 Share

The persistance API will provide a unified method of accessing structured data in personal clouds that is aimed at enabling application writers to store structured data and synchronise it between machines and devices, irrespective of the personal cloud provider.

As an example of why this is useful, consider an application writer that is writing a recipe application. Recipes can be created on any device, and viewed on any device. Rather than forcing the application writer to implement their own server-side storage for the recipes (which is often not of interest to the application writer), we enable them to use a unified way of persisting the recipes that will work on all devices and leverage whatever technologies are made available on those devices to satisfy the persistance requirements.

One particularly noteworth aspect of this is that applications are often written to be (mostly) browser-side, so one of the targets of the persistance API is to be usable in that environment.

6.4 Proximity

Proximity sync can be achieved between multiple devices of the same user, or multiple users that are sharing content. It involves three phases: *discovery*, *trust* and *lookup*.

Proximity sync discovery is performed via mDNS/DNS-SD. Establishing trust may be performed through a number of ways, although we propose a bluetooth-like pin exchange as being the least cumbersome to users. Once a native client has found a trusted proximity sync service, when the client needs to perform a content download it will first look up the content in the proximity sync service, using server-provided metadata to that end. It may then choose to transfer the content from locally rather than from the server.

Multiple native clients running proximity sync lookup servers should, provided trust can be established, arrange themselves into a distributed content mesh network to be resilient against partitions and other networking woes.

6.5 Portability

Synchronization between devices running different file and operating systems presents a challenge that needs addressing in a uniform way to avoid data loss.

Nowadays, file systems present in devices have different limits on acceptable file names (e.g.: in the characters that constitute a valid file name, the encoding the character is serialized in, or the length of the file name, to name a few). Moreover, some operating systems do not differentiate between upper-case and lower-case file names, and some operating systems do not distinguish between different Unicode normalizations; others do.

Native clients must, therefore, be resilient to the fact that it might not be technically feasible for all files in a personal cloud to be synced to a particular device. Furthermore, a system intended for interoperability must work with the native clients' limitations in this regard. Otherwise data loss will occur.

6.6 Persistence

A cross-platform, cross-device, language-agnostic, structured data service is a cornerstone of personal cloud services, complementary and orthogonal to file synchronization. U1DB is one such service.

U1DB is a cross-platform, cross-device, syncable database API. In order to be this way, there's a philosophy behind it. Key to this philosophy is that U1DB can be implemented in many languages and on top of many back ends: this means that the API needs to be, as much as possible, portable between very different languages. Each implementation should

implement The high-level API in the way appropriate to that language (Python uses tuples all over the place, Vala/C use a Document object for most things, and so on), but it's important that an implementation not diverge from the API. Because U1DB is a syncable database, it's quite likely that an app developer using it will be building their app on multiple platforms at once. Knowledge that an app developer has from having built a U1DB app on one platform should be transferable to another platform. This means that querying is the same across platforms; storing and retrieving docs is the same across platforms; syncing is the same across platforms. U1DB is also syncable to Ubuntu One, which is a very large server installation; the API needs to be suitable to run at scales from a mobile phone up to a large server installation.

For similar reasons, U1DB is schemaless. Documents stored in U1DB do not need to contain any pre-defined list of fields; this way, an application can store whatever it wants, however it wants; development is faster and changing how data is stored is simpler.

What this means is that U1DB is for user-specific data. A desktop app or a mobile app storing data for a user is the ideal use case. A web app which holds data for many users should be using and syncing a separate U1DB for each user. U1DB isn't designed to be the backend database for the next Facebook.

To this end, there are a few guidelines. Primarily, the guideline the U1DB team used for the largest U1DB is somewhere around 10,000 documents. It's important to note that this is not an enforced limit; a user can store any number of documents in a U1DB if they desire. However, implementations are allowed to assume that there are not infinite documents; in particular, suggestions for API changes which make things more aggravating for a 1,000 documents use-case in order to help with a zillion documents are not likely to be adopted.

Similarly, suggested changes to the high-level API which are very difficult to implement in static languages like C are also unlikely to be adopted, in order to maintain the goal of knowledge on one platform transferring to another.

U1DB is designed so that implementations are built by creating small layers on top of existing storage solutions. It isn't a database in itself; it's an API layer which sits on top of a native database to that platform. This means that the platform provides the actual database functionality and U1DB takes advantage of it. SQLite where available, localStorage for JavaScript in the web browser; U1DB should work with the platform, not be ported to it.

It should be easy to sync a U1DB from place to place. There is a direct server HTTP API, which allows an app to work with a U1DB on the server without any locally-stored data. However, this is not the preferred way to work; to edit data in a U1DB, the easiest course should be to sync that database, edit it, and then sync it back. If this is not the easiest course, then that's a bug.

7 Benchmarking

7.1 Expected Evaluation

Evaluating the effectiveness of the system in guaranteeing privacy requires a prototype to be ready for people to test. However, before that, we plan to evaluate the above mechanism from a performance perspective. We aim at evaluating the time it takes for determining the sensitivity of an item and evaluating the privacy risk. We also will attempt at seeing the rate at which the privacy risk can be decreased via various mitigation techniques.

For that, we will be utilizing the following real-world datasets:

- **Enron Email Dataset [14, 15]:** Sharing data of various types with certain entities in the cloud is analogous to sharing attachments via emails. Hence, testing the performance of our schemes with the latter can give an insight that applies to the former. Accordingly, we will be using the Enron email corpus, which has data from around 150 senior management executives of Enron, which was originally made public during the Federal Energy Regulatory Commission's investigation of Enron. The corpus contains roughly 500,000 messages.
- **Ubuntu One Dataset** In our project, we will be having access to a dataset obtained from Ubuntu One cloud, which contains metadata about files shared in the cloud. We will be exploiting this dataset for the performance testing too.
- **EyeOS Dataset** EyeOS already has its own community with thousands of users. The project will benefit from this situation by obtaining a valuable dataset.
- **StackSync Dataset** We plan to release StackSync and gather information from a large number of students. We will create a dataset with information such as availability patterns, metadata about files and other application usages.

To evaluate our interoperability efforts, we plan to develop a set of conformance tests that will determine whether an API meets our protocol specification. With this, we will be able to easily validate any API implementation, guaranteeing the compliance of the defined interoperability and storage standards.

Furthermore, it will also be able to validate the Quality of Service offered by an API. The test will run a set of benchmarks to assess the relative performance of an implementation. It will measure metrics such as the latency while we increase the throughput. In addition, we will use Yahoo Cloud Serving Benchmark (YCSB) [16], which is a tool used to evaluate the performance of different key-value and cloud services.

The establishment of a methodology is the first step for benchmarking.

From a technical point of view, we will implement a technical benchmarking, in order to achieve optimal performances at the Openstack platform.

At last phases of the project we will add another interesting benchmarking like:

- **Energy benchmarking.** Analysing energy proficiency of the platform.

- **Strategy benchmarking.** When the project objectives were selected, a strategic benchmarking about similar technologies were performed. The evolution and apparition of new products, will be necessary also at the final phases of the project.

The CloudSpaces project will develop a Technical benchmarking, as a technique used to compare Openstack against the technical criteria that are tested.

Benchmarks may be employed at various stages during system development to ensure that resource utilization remains within the system requirements limits.

The configuration to be tested is conformed by the different test platforms of the project. Some of them are used for Openstack version testing, while other platforms will be centred at proficiency and the use of the sharing platform.

We will identify the variables (e.g., the disk drive use, common sizes for upload or download , number of concurrent users...).

At last, we will select the final platforms candidates, to be benchmarked.

At this moment, partners are defining the most relevant items for being included at the benchmark.

7.2 Testbed description

7.2.1 URV's Testbed

The URV has deployed a testbed platform for testing and development purposes. The main requirement of this platform is to simulate a highly available and scalable platform without the necessity of a cloud provider for our development and tests. This platform will be used in the open Internet experiments with real users. It will also interoperate with other platforms in Canonical, Tissat, and eyeOS.

In particular, the URV is modifying OpenStack Swift and this requires a local deployment in our University. We have deployed the OpenStack Swift Grizzly version. Following the Swift architecture we have to differentiate between two different types of nodes: proxies and storage nodes. As shown in Figure 30, our platform is composed by four storage nodes, two proxies and a private switch:

The hardware specifications of the proxies and the storage nodes are almost the same:

- Proxy node hardware:

  CPU: Intel Xeon X5-2407 2.20GHz

  RAM: 12 GB 1333 MHz (3x4GB)

  Primary HDD: 500 GB 7.2K RPM

- Storage node hardware

  CPU: Inel Xeon X5-2403 1.80GHz

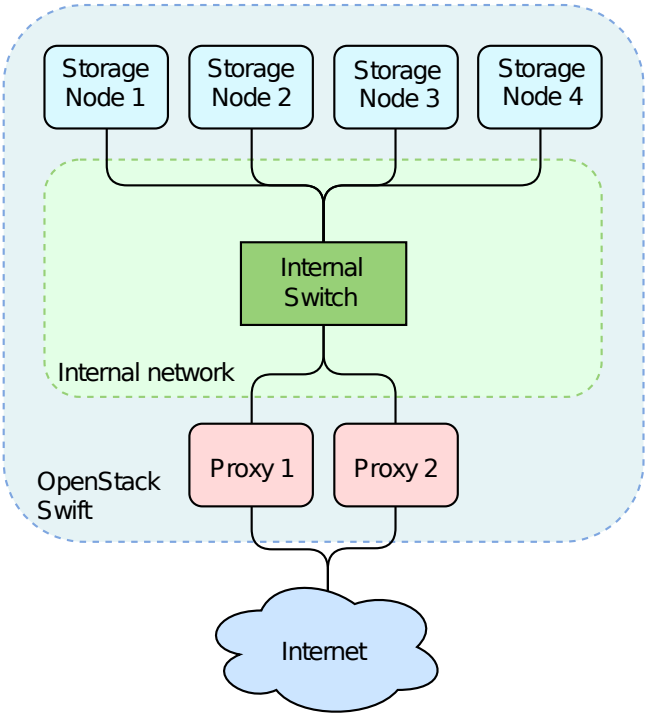


Figure 30: URV's testbed

RAM: 8GB 1333MHz (2x4GB)
Primary HDD: 500 GB 7.2K RPM
Storage HDD: 2x 2TB 7.2K RPM

The hardware differences between both types of nodes are the amount of RAM in the proxies which is higher than the amount used in storage nodes and also the proxy CPU, which has better performance.

We also need some open ports to allow connections from Internet clients. The ports needed are explained below:

- **5000**: Used by Keystone for Swift authorization and authentication.
- **8080**: Swift uses it for the clients requests.
- **6881**: BitTorrent port.
- **80**: Apache
- **5672/5673**: RabbitMQ. Used to communicate clients with the metadata server.
- **22**: SSH for remote connection.

The rack has been placed at the URV's server room, guaranteeing an adequate temperature and and tolerance for power outages. Since the bandwidth is also provided by URV we assume it will be enough to perform our tests and provide a good quality of service to the rest of partners.

7.2.2 Tissat's Testbed

The hardware platform has been designed so that it can evolve according to the needs of growth and use of the project, which may include different virtual platforms for development and testing, thus let us to maximize capability and resource utilization.

Hardware

Tissat has installed the following equipment for CloudSpaces project:

Servers	1
Use	Development platform
Configuration	Keystone modules, Proxy and OpenStack Swift Folsom Version
Description	Develop testing
Servers	1
Use	Test platform
Configuration	Keystone modules, Proxy and OpenStack Swift Folsom Version
Description	Open functionality testing

Servers	1
Use	Swift test platform
Configuration	Keystone modules, Proxy and OpenStack Swift Folsom Version
Description	StackSync test storage platform with URV SyncService
Servers	7
Use	Pretesting platform
Server 1 configuration	1 server, operating system Ubuntu Server 12.04.3 LTS, with swift modules installed
Server 1 description	swift-client: This server is used for testing at storage platform
Server 2 configuration	1 server, operating system Ubuntu Server 12.04.3 LTS, with OpenStack Keystone module installed
Server 2 description	swift-k1-pre: This server is used to validate user operations in the Swift storage platform
Server 3 configuration	1 server, operating system Ubuntu Server 12.04.3 LTS, with OpenStack Proxy module
Server 3 description	swift-p1-pre: This server manages data transfers between users and storage nodes
Server 4-7 configuration	4 servers, operating system Ubuntu Server 12.04.3 LTS, with modules and storage Swift
Server 4 -7 description	swift-pre-n1, n2-swift-pre, pre swift-n3-n4-and swift-pre: These servers manage storage Swift platform, and include replicates information to ensure availability at all times, and no data loss

Servers	14
Use	Stable OpenStack Swift Platform, Folsom Version, consisting of 12 physical servers and 2 virtual servers
Server 1 configuration	1 virtual server, operating system Ubuntu Server 12.04.3 LTS, with Openstack Keystone module installed
Server 1 description	swift-k1: This server is used to validate user operations in the Swift storage platform
Server 2 Configuration	1 physical server, operating system Ubuntu Server 12.04.3 LTS, with OpenStack Proxy module installed
Server 2 description	swift-p1. This server manages data transfers between users and storage nodes
Server 3 configuration	1 virtual server, operating system Ubuntu Server 12.04.3 LTS, with OpenStack Proxy module installed
Server 3 description	It is a virtual proxy server used for backup in case of any contingency with physical Proxy Server
Server 4-7 configuration	1 Blade four-servers Dell, with 3 2TB disks of storage per server. Each server runs on Ubuntu Server 12.04.3 operating system and have installed storage modules and OpenStack Swift
Server 4-7 description	rc-s1, rc-s2, rc-s3 and rc-s4: These servers are used to manage the storage at Swift Platform, through replication between themselves, in order to ensure its accessibility, and avoiding data leaks
Server 8-10 configuration	3 physical servers Dell R520 with 8 2TB disks, which increases the total storage capacity of the platform.Each server runs on Ubuntu Server 12.04.3 operating system and have installed storage modules and OpenStack Swift
Server 8-10 description	swift-s5, swift-s6 and swift-s7: These servers are used to manage the storage at Swift Platform, through replication between themselves, in order to ensure its accessibility, and avoiding data leaks
Server 11-14 configuration	1 Blade four-servers Dell , with 3 1TB disks of storage per server. In 3 servers are installed Swift and Nova OpenStack modules, version Folsom
Server 11-14 Description	rc-c1,-c2 rc, rc-rc-c3 and c4: Servers are used for testing purposes, with Nova modules of OpenStack platform. One of the blade servers has been used to install and test Swift and Nova OpenStack modules, at its most recent version: Grizzly

Location

Almost all nodes will be hosted in Walhalla, DataCenter located in Castellón, there being a backup node in another DataCenter located in Valencia, both of them property of Tissat.

Walhalla is one of 8 global DataCenters that are certified as Tier IV, which ensures a reliability of 99.995% per year. It has been designed focusing on efficiency:

- Cloud Computing. Designed to host cloud computing services.

- Green Energy. Trigeneration energy production. It also has a quintuple redundant system.
- Intelligent control. Integration between IT management systems and energy infrastructure.
- Sustainable building. Power Usage efectiveness, PUE 1.1.
- Tier IV Certification. Walhalla is the first Tier IV DataCenter certificated by Uptime Institute in Southern Europe.
- Datacenter Leaders Awards. Awarded by "Datacenter Leaders Awards 2010" for First European Medium-Sized Datacenter.

Because of these characteristics, it offers the highest quality platform and possible stability for installation and deployment of the nodes needed for the project.

Monitoring of Swift Stable Platform

Through the Nagios software, we are continuously monitoring servers of Swift Stable Platform.

Specific periodic checks for OpenStack are used in order to receive information continuously in assuring that the platform is working properly.

In addition, has been implemented different monitors of critical processes at the different OpenStack modules.

It is also being monitored the status, at operating system level, for all servers included at the platform, providing continuous information about connectivity, CPU load, the load factor of the different file systems, and the use of RAM and swap.

This allows us to run the entire platform in real time and detect any errors or failure, to address it as soon as possible, minimizing the impact on active services...

Service Status Details For Host 'swift-k1'

Host	Service	Status	Last Check	Duration	Attempt	Status Information
swift-k1	Carga Cpu	OK	16-10-2013 11:23:52	56d 5h 57m 2s	1/2	OK - load average: 0.00, 0.01, 0.05
	Check_Keystone	OK	16-10-2013 11:24:14	15d 16h 53m 9s	1/2	Got token 785419e4080f48c39d24e45db7507e0f for user 7e6bb66beb e8a9e4b953284c54a07032b831a35978
	Check_Swift	OK	16-10-2013 11:21:52	6d 2h 24m 8s	1/2	Upload+download+delete of 732 KiB file in container check_swift
	Filesystem /	OK	16-10-2013 11:22:47	43d 22h 48m 22s	1/2	DISK OK - free space: / 25007 MB (90% inode=96%):
	Ocupacion_Memoria	OK	16-10-2013 11:23:42	15d 15h 43m 34s	1/2	OK - 63.6% (648296 kB) free.
	Proceso keystone-all	OK	16-10-2013 11:22:14	56d 5h 56m 42s	1/2	PROCS OK: 1 process with command name 'keystone-all'
	Proceso mysqld	OK	16-10-2013 11:23:11	15d 18h 25m 31s	1/2	PROCS OK: 1 process with command name 'mysqld'

Service Status Details For Host 'swift-p1'

Host	Service	Status	Last Check	Duration	Attempt	Status Information
swift-p1	Carga Cpu	OK	16-10-2013 11:21:14	6d 2h 23m 21s	1/2	OK - load average: 0.01, 0.03, 0.05
	Filesystem /	OK	16-10-2013 11:23:15	6d 2h 23m 17s	1/2	DISK OK - free space: / 47478 MB (95% inode=98%):
	Ocupacion_Memoria	OK	16-10-2013 11:21:38	6d 2h 22m 3s	1/2	OK - 98.0% (16040480 kB) free.
	Proceso memcached	OK	16-10-2013 11:22:51	6d 2h 22m 44s	1/2	PROCS OK: 1 process with command name 'memcached'
	Proceso swift-proxy-server	OK	16-10-2013 11:21:27	6d 2h 22m 49s	1/2	PROCS OK: 2 processes with command name 'swift-proxy-ser'

Web Management interface

To create this interface we have retied on the following technologies:

- **Openstack Swift.** This service provides cloud storage services. This software allows us to create containers, upload, download and delete files and folders. It also provides security directives for privacy.
- **Stacksync REST API.** It's an Openstack Swift extension that includes easy mechanisms for allowing communication between software and Openstack Swift. There are also included other mechanisms in order to improve data transfer, and to simplify the creation of metadata.
- **Symfony.** It is an MVC framework for PHP that allows developing of code in a more simple and structured way, the main goal using this framework for the project is to make use of the best software engineering practices.
- JQuery **Mobile.** It is a tool that allows us to create web applications optimized for mobile phones, it allow us to show an appearance of native application when you are actually using the web, by the use of techniques of AJAX and HTML 5.

In this framework we have had as objectives to create a web environment where a user can register and have private storage space.

To do this the user can upload files to the server, download and delete them as you wish. Also included managing folders for higher user comfort.

During this project we have prioritized issues such as ease of use for the user, through a web interface compatible with all browsers and devices, including PCs, mobiles, tablets, etc.

We also examined the possible performance improvements, that could be achieved through different settings for the storage server of the application.

Finally, there has been a security analysis of the application, and the potential authentication problems and / or soft passwords.

7.3 StackSync application tests

First of all, a base form for global testing of the platform has been fixed, even before of the develop of all the interfaces. The objective for this dot is to uniformize the ' "check form" during all the project.

Next, we describe the check base form StackSync Application Tests.

Application Version:

Test start date:

Test end date:

General tests

Objective	Works (Y/N)	Description
Stacksync Server Installation		
web-api Installation		
Secure mode Configuration https		
Configuration for public access		
Safety Audit		
Testing Windows client		
Testing Web Browser client application		
Testing Linux client		
Testing Android Client		
Testing IOS Client		
Load test		
Testing Manage Module		
Availability of Installation Manuals		
Availability of Installation Manuals		

Testing Windows Client

Objective	Works (Y/N)	Description
Client installation		
Access to the application		
ipload a fUle		
Download a file		
Modify a file		
Remove a file		
Fzle synchroniiation		

Testing Web Browser Client

Objective	Works (Y/N)	Description
Access to the application		
Upload a file		
Download a file		
Modify a file		
Remove a file		
File synchronization		

Testing Linux Client

Objectvie	Works (Y/N)	Description
Client installation		
Access to the application		
Upload a file		
Download a file		
Modify a file		
Remove a file		
File synchronization		

Testing Android Client

Objective	Works (Y/N)	Description
Client installation		
Access to the application		
Upload a file		
Download a file		
Modify a file		
Remove a File		
File synchronization		

Testing IOS Client

Objective	Works (Y/N)	Description
Client installation		
Access to the application		
Upload a file		
Download a file		
Modify a file		
Remove a file		
File synchronization		

In addition, specific templates for intensive testing, has been used.

Several Openstack platforms has been tested at Tissat during all this time, including several functionalities:

- Testing of Proxy
- Testing of virtual proxy
- Testing of dual-proxy
- Testing of Swift Folsom Releases:
  - Folsom 2012.2.1
  - Folsom 2012.2.2
  - Folsom 2012.2.3

It implies an intensive use of our platform, having to create, test all functionalities, and also, to check completely and with every release, the availability of the Openstack access interfaces.

7.3.1 Canonical's Testbed

Canonical has deployed a testbed comprising of an additional instance of the full Ubuntu One production infrastructure, with a few exceptions[9].

This testbed can be used from the standard Ubuntu One clients by configuring them appropriately. The website can be seen at `https://one.ubuntone.info/`.

This deployment uses the *staging* Ubuntu One Accounts server; in order to create an account you must first launch the Ubuntu One Accounts backend helper with two environment variables set, prior to launching the desktop client (`syncdaemon`) for the first time. In the `sh` family of shells and on Ubuntu that would be

```
$ export SSO_AUTH_BASE_URL=https://login.staging.ubuntu.com
$ export SSO_UONE_BASE_URL=https://one.ubuntone.info
$ /usr/lib/ubuntu-sso-client/ubuntu-sso-login
```

Once `ubuntu-sso-login` is running, `syncdaemon` needs to be configured to not look up the SRV record via DNS but to connect to host `fs.ubuntone.info` on port `11001`. For example, via commandline arguments on Ubuntu,

```
$ /usr/lib/ubuntuone-client/ubuntuone-syncdaemon --dns_srv=: \
  --host=fs.ubuntone.info --port=11001
```

[9] specifically, photo album creation and the music store and streaming are unsupported in the Canonical Cloudspaces Testbed.

These same options can be configured per host or per user using the usual XDG mechanism.

> These steps assume the testbed account is the only Ubuntu One account on a particular login; while `syncdaemon` is able to handle multiple accounts if configured appropriately, such usage is unsupported. Further information can be found in the commandline documentation.

7.4 Guidelines for the preparation of traces

7.4.1 Introduction and Objective

Increasingly, the role of production datasets in today's research in Cloud computing is gaining importance. Such valuable information, coming from the real use of widely deployed Cloud services, constitutes a solid ground for researchers to devise novel techniques and advances.

More technically, using real-world datasets in any piece of research avoids (at least partially) the introduction of synthetic conditions and risky assumptions during the research process, which may not be realistic or accurate in some cases. In this sense, the lack of real data translates into uncertainty, which may in turn affect the quality and applicability of the targeted solution for a specific Cloud setting or application.

The CloudSpaces project aims to benefit from the cooperation between industry partners and universities to produce high-quality research. As a pillar of this cooperation, industry partners will collaborate with universities by providing some information of their production Cloud services. This will benefit partners, as well as the whole project, in a twofold manner: i) academic partners will work with real-world data, giving to their research technical strength and impact, and ii) the advances achieved using a specific dataset are applicable to the service where such a dataset comes from. This represents that many potentially beneficial and novel techniques will be ready for exploration to industry partners during the development of the project.

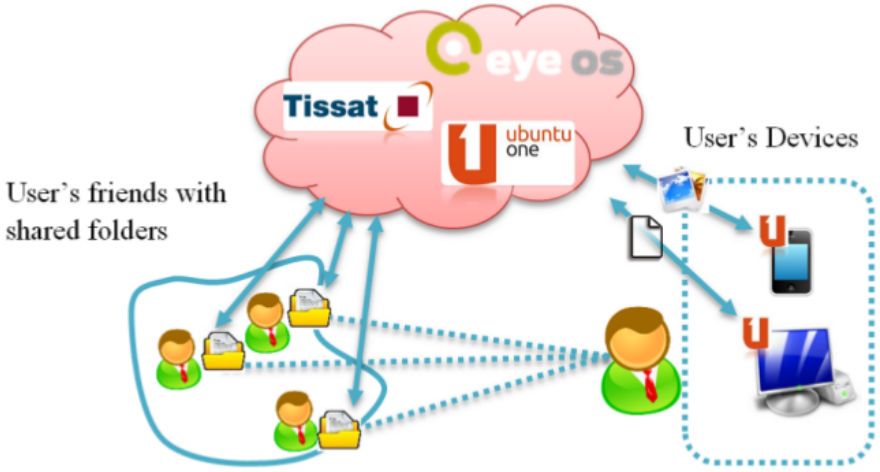
Therefore, we foresee a fruitful collaboration between industry partners (Canonical, eyeOS, Tissat) and academic partners (EPFL, Eurecom, URV) in the context of the CloudSpaces project.

7.4.2 Dataset Definition

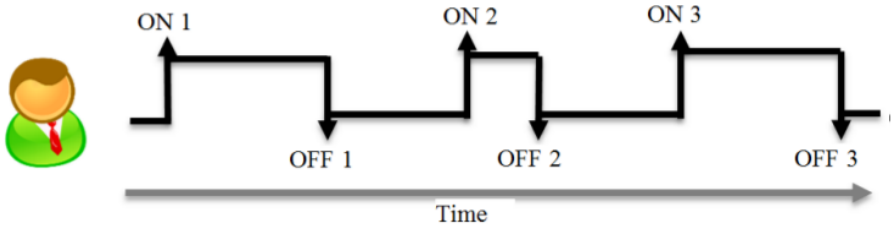
Before creating a dataset from a production service, we must specify which data will be gathered and an appropriate format to express this information. To this end, we address a detailed specification for the industry partners.

There are three main characteristics that we should capture from each user in a Cloud service: user availability, storage use and social interactions. These characteristics can be appreciated in the picture below. A single user is connected to the Cloud service during certain periods irrespective of the device and location (user availability). In this sense, that

user is managing his data utilizing the service. Thus, he can be uploading files or downloading pictures to see them from his smartphone (storage use). Finally, the Cloud services of this project also enable sharing capabilities (Personal Cloud, Cloud Web Desktop). This means that every user will probably share content with other users in a single service (social interactions).

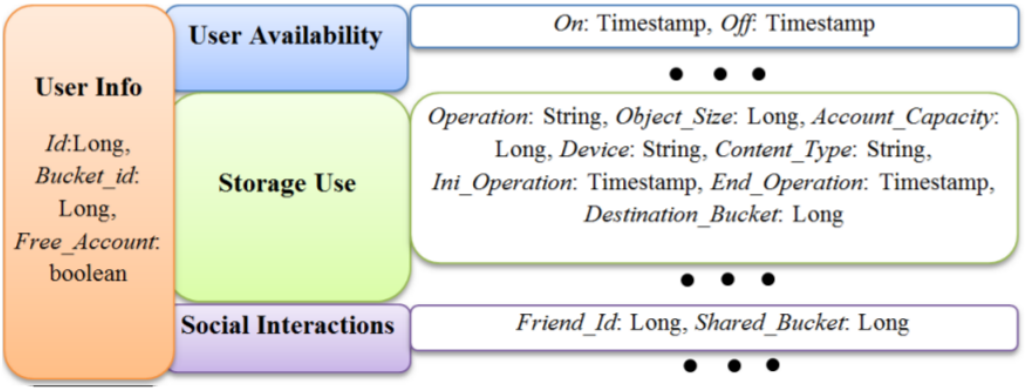


First, we refer to user availability as the online/offline behaviour of a user. This information is of particular interest to understand the dynamics of users and the potential periods of peak workloads in a Cloud service. Armed with this information, we can provide better resource provisioning in the datacenter side taking into account the availability patterns of users. The availability of a user can be observed in the next figure:



Second, for storage use we mean the utilization of a user of a storage service, in terms of storage capacity employed, number of transactions and bandwidth consumption. Finally, an important aspect of a personal storage service refers to the social interactions, that is, the activity of a single user with a set of other users in that service. Specifically, we refer to the file sharing activity among group of users.

Once defined the main aspects that we want capture in the datasets, we proceed to describe their format in depth. Thus, we consider a dataset as a set of records (e.g. database records, lines in a log file, etc.) where each record represents the behaviour of a single user. In that case, the high level description of a record is as follows:



Next, we proceed to describe each field in terms of objectives and type of information that they will contain.

User Info

Field	Type	Description
Id	Long	Unique number that identifies a single user.
BucketId[10]	Long	Unique number that identifies the bucket of a user. That is, we want to relate a bucket with a user to study the use that a user makes of its own storage space.
FreeAccount	Boolean	This field describes if the user employs a free account or not.

User Availability

Field	Type	Description
On	Timestamp	This timestamp represent the start of an online session of a user.
Off	Timestamp	This field represents the end of an online session of a user.

Note that **for every user record there will be many online sessions**. These online sessions, defined as tuples of timestamps (i.e. [ON, OFF]), represent the connectivity of a single user to the system during the data gathering period. Thus, every time a user connects to the system a new [ON, OFF] tuple should be created for that user, filling the On field with the timestamp of the initial connection instant. The Off field should be filled when that user does not appear as connected (e.g. no transactions, no keepalives, etc.) for a certain period of time (e.g. 5 minutes).

It is important to notice that here we assume that the client software maintains a sort of heartbeat mechanism with the Cloud service. This provides an easy way of tracking the availability of users.

Storage Use

Field	Type	Description
Operation	String	Type of action performed against the server (read, write, etc).
ObjectSize	Long	Size in bytes of the object manipulated by the user. For example, the size of an uploaded file.
AccountCapacity	Long	For each storage operation, we want to register the account capacity of a user's account. Basically, this means to register the number of bytes stored that account at the moment of the storage operation.
Device	String	Type of user's device that is accessing to the service (software client, web browser, mobile phone, etc).
ContentType	String	Type of file involved in the storage operation (e.g. image, text file, video). Probably registering the extension of the file is enough for our purposes.
IniOperation	Timestamp	Initial instant of the storage operation.
EndOperation	Timestamp	Final instant of the storage operation.
Bucket	Long	Identifier of the destination bucket of the current operation. Recall that this bucket might be the user's bucket or a shared bucket with a social contact.

As in the previous case, a single user register will contain many storage operations.

Social Interactions

Field	Type	Description
FriendId	Long	Identifier of a social contact of the current user.
SharedBucket	Long	Bucket shared among both users.

Note that depending on the internal architecture of the Cloud service, two users can share many folders that can be placed in several buckets. For this reason a single user can have various entries for a single friend, accounting for different buckets.

7.4.3 Further Considerations

We are conscious that collecting data from a widely deployed Cloud service poses non-negligible privacy issues and technical challenges.

First, datasets should characterize a service **without disclosing personal user information of any type**. This means that user records must be totally anonymous and no personal information of users should be recorded (e.g. email addresses, IPs). This principle is supported by the specification of the datasets, which does not include any type of sensitive user information.

Finally, this document defines an abstract specification of the structure of datasets for research purposes. However, gathering a certain kind of data may pose technical challenges

and difficulties. For this reason, academic and industry partners should reach consensus and solve these low-level problems to satisfy the requirements of the dataset while minimizing the effort during the data gathering process.

7.5 Measuring Personal Cloud Storage

Due to internal policy regulations and technical reasons form Canonical and eyeOS, the extraction of the traces have been delayed and, therefore, academic partners were not able to make an analysis of them. Instead, we present a measurement study of three major Personal Clouds: Dropbox, Box and SugarSync. We accessed to free accounts through their REST APIs, analysing important aspects to characterize their QoS, such as transfer speed, variability and failure rate. The measurement was conducted during two months, and we believe that it will be of interest to researchers and developers with diverse concerns about Cloud storage, since our observations can help them to understand and characterize the nature of these services.

We gathered information from more than 900,000 storage operations, transferring around 70TB of data. We analysed important aspects to characterize their QoS, such as in/out transfer speed, service variability and failure rate.

7.5.1 Measurement Methodology

We installed several vantage points in the URV university network and PlanetLab [17] to measure the performance of three of the major Personal Cloud services in the market: Dropbox[11], Box[12] and SugarSync[13]. The measurement methodology was based on the REST interfaces that these three Personal Cloud storage services provide to developers.

Personal Clouds provide REST APIs, along with their client implementations, to make it possible for developers to create novel applications. These APIs incorporate authorization mechanisms (OAuth [18]) to manage the credentials and tokens that grant access to the files stored in user accounts. A developer first registers an application in the Cloud provider website and obtains several tokens. As a result of this process, and once the user has authorized that application to access his storage space, the Personal Cloud storage service gives to the developer an *access token*. Including this *access token* in each API call, the application can operate on the user data.

There are two types of API calls: *metadata* and *data* calls. The former type refers to those calls that retrieve information about the state of the account (i.e., storage load, filenames), whereas the latter are those calls targeted at managing the stored files in the account. We will analyse the performance of the most important data management calls: `PUT` and `GET`, which serve to store and retrieve files.

[11] `http://www.dropbox.com`
[12] `http://www.box.net`
[13] `http://www.sugarsync.com`

Location	Op. Type	Operations	Transferred Data
University Labs	GET	168,396	13.509 TB
	PUT	247,210	15.945 TB
PlanetLab	GET	354,909	31.751 TB
	PUT	129,716	9.803 TB

Table 7.5a: Summary of Measurement Data (May 10 − July 15)

Measurement Platform

We employed two different platforms to execute our tests: URV laboratories and PlanetLab. The reason behind this is that our labs contain *homogeneous and dedicated machines* that are under our control, while PlanetLab allows the analysis of each service from *different geographic locations*.

URV laboratories. We gathered 30 machines belonging to the same laboratory to perform the measurement. These machines were Intel Core2 Duo equipped with 4GB DDR2 RAM. The employed operating system was a Debian Linux distribution. Machines were internally connected to the same switch via a 100Mbps Ethernet links.

PlanetLab: We collected 40 PlanetLab nodes divided into two geographic regions: Western Europe and North America. This platform is constituted by heterogeneous (bandwidth, CPU) machines from several universities and research institutes. Moreover, there were two points to consider when analyzing data coming from PlanetLab nodes: i) Machines might be concurrently used by other processes and users, and ii) The quota system of these machines limited the amount of in/out data transferred daily.

Specifically, we used the PlanetLab infrastructure for a high-level assessment of Personal Clouds depending on the client's geographic location. However, the mechanisms to enforce bandwidth quotas in PlanetLab nodes may induce the appearance of artifacts in bandwidth traces. This made PlanetLab not suitable for a fine-grained analysis in our context.

Workload Model

Usually, Personal Cloud services impose file size limitations to their REST interfaces, for we used only files of four sizes to facilitate comparison: 25MB, 50MB, 100MB and 150MB[14]. This approach provides an appropriate substrate to compare all providers with a large amount of samples of equal-size files. Thanks to this, we could observe performance variations of a single provider managing files of the same size.

We executed the following workloads:

Up/Down Workload. The objective of this workload was twofold: Measuring the maximum up/down transfer speed of operations and detecting correlations between the transfer speed and the load of an account. Intuitively, the first objective was achieved by alternating upload and download operations, since the provider only needed to handle one operation

[14]Although the official limitation in some cases is fixed to 300MB per file, we empirically proved that uploading files larger than 200MB is highly difficult. In case of Box this limitation is 100MB.

per account at a time. We achieved the second point by acquiring information about the load of an account in each API call.

The execution of this workload was continuously performed at each node as follows: First, a node created synthetic files of a size chosen at random from the aforementioned set of sizes. That node uploaded files until the capacity of the account was full. At this point, that node downloaded all the files also in random order. After each download, the file was deleted.

Service Variability Workload. This workload maintained in every node a nearly continuous upload and download transfer flow to analyze the performance variability of the service over time. This workload provides an appropriate substrate to elaborate a time-series analysis of these services.

The procedure was as follows: The upload process first created files corresponding to each defined file size which were labeled as "reserved", since they were not deleted from the account. By doing this we assured that the download process was never interrupted, since at least the reserved files were always ready for being downloaded. Then, the upload process started uploading synthetic random files until the account was full. When the account was full, this process deleted all files with the exception of the reserved ones to continue uploading files. In parallel, the download process was continuously downloading random files stored in the account.

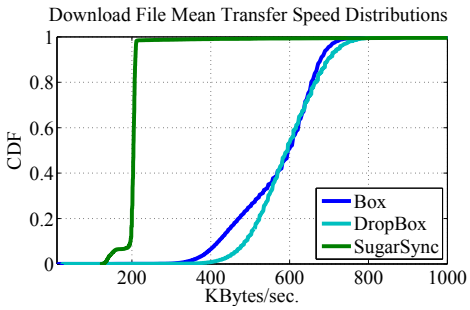
Finally, we executed the experiments in different ways depending on the chosen platform. In the case of PlanetLab, we employed the *same machines in each test*, and therefore, we needed to sequentially execute all the combinations of workloads and providers. This minimized the impact of hardware and network heterogeneity, since all the experiments were executed in the same conditions. On the contrary, in our labs we executed in parallel a certain workload for all providers (i.e. assigning 10 machines per provider). This provided two main advantages: The measurement process was substantially faster, and fair comparison of the three services was possible for the same period of time.

We depict in Table 7.5a the total number of storage operations performed during the measurement period.

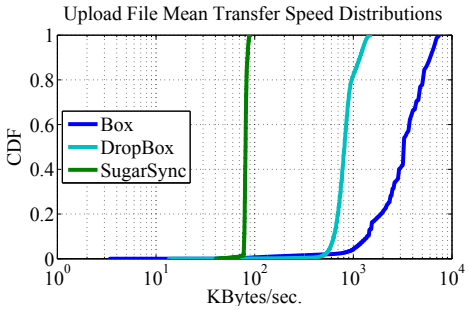
Setup, Software and Data Collection

Before starting our experiment, we created around 150 new user free accounts from the targeted Personal Clouds. That is 120 new accounts for PlanetLab experiments (40 nodes $\times$ 3 Personal Clouds), and 30 accounts for the experiments in our labs (10 accounts per Personal Cloud deployed in 30 machines). We also registered as developers 35 applications to access the storage space of user accounts via REST APIs, obtaining the necessary tokens to authenticate requests. We assigned to every node a single new free account with access permission to the corresponding application. The information of these accounts was stored in a database hosted in our research servers. Thus, nodes executing the measurement process were able to access the account information remotely.

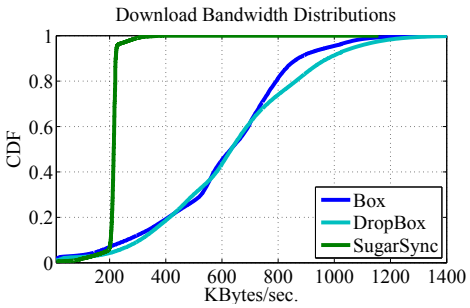
Measurement processes were implemented as Unix and Python scripts that ran in every node. These scripts employed third party tools during their execution. For instance, to



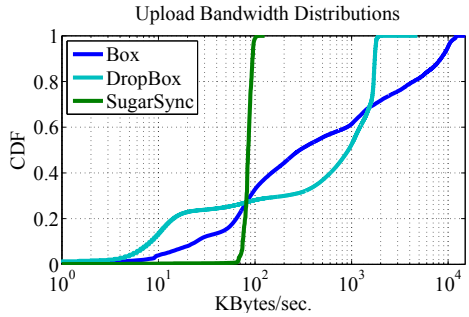
(a) Download file MTS distributions



(b) Upload file MTS distributions



(c) Download bandwidth distributions



(d) Upload bandwidth distributions

Figure 31: Transfer capacity of Box, Dropbox and SugarSync free account REST API services. The data represented in these figures corresponds to the aggregation of the up/down and service variability workloads during 10 days (June/July 2012) in our university laboratories.

synchronize tasks, such as logging and starting/finishing experiments, we used the `cron` time-based job scheduler. To gather bandwidth information we used `vnstat`, a tool that keeps a log of network traffic for a selected interface. Nodes performed storage operations against Personal Clouds thanks to the API implementations released in their websites.

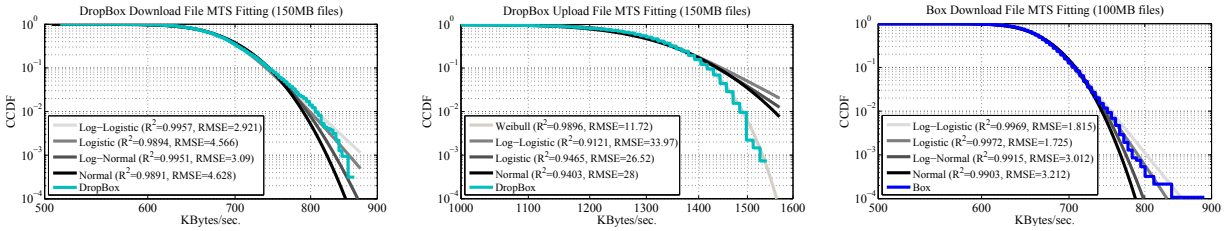
The measurement information collected in each storage operation was sent periodically from every node to a database hosted in our research servers. This automatic process facilitated the posterior data processing and exploration. The measurement information that nodes sent to the database describes several aspects of the service performance: operation type, bandwidth trace, file size, start/end time instants, time zone, capacity and load of the account, and failure information.

7.5.2 Measuring Personal Cloud REST APIs

Transfer Capacity of Personal Clouds

The transfer capacity of Box, Dropbox and SugarSync is characterized using the following indicators:

- *File Mean Transfer Speed (MTS).* This metric is defined as *the ratio of the size of a file, S, to the time, T, that was spent to transfer it: $MTS = S/T$ (KBytes/sec).*



(a) Dropbox down. file MTS fitting (150MB files).

(b) Dropbox up. file MTS fitting (150MB files).

(c) Box down. file MTS fitting (100MB files).

Figure 32: Distribution fittings of upload/download file mean transfer speeds (MTS) of the examined Personal Clouds (up/down workload, university labs).

- *Bandwidth Distributions*. We define as a *bandwidth trace* the set of values that reflects the transfer speed of a file at regular intervals of 2 secs. To obtain a single empirical distribution, we aggregated the bandwidth traces of all the transfers separated by uploads and downloads. We refer to the resulting empirical distribution as the *aggregated bandwidth distribution*.

Transfer speeds. Fig. 31 reports these metrics for both workloads (up/down and service variability) executed in our university labs during 10 days. First, Fig. 31 evidences an interesting fact: *Personal Clouds are heterogeneous in terms of transfer speed*. For instance, Fig. 31b shows that Box and Dropbox present an upload MTS several times faster than SugarSync. The same observation holds for downloads. Moreover, the heterogeneity of these services also depends on the *traffic type* (in/out). This can be appreciated by comparing Fig. 31a with Fig. 31b: *Dropbox exhibits the best download MTS while Box presents the fastest uploads.*

This proves that *the transfer performance of these services greatly varies among providers*, and consequently, developers should be aware of this in order to select an adequate provider.

Among the examined Personal Clouds, Dropbox and SugarSync are *resellers* of major Cloud storage providers (Amazon S3 and Carpathia Hosting, respectively). On the other hand, Box claims to be owner of several datacenters. In our view, it is interesting to analyze this Cloud ecosystem and the possible implications to the service delivered to end-users.

In this sense, in Fig. 31 we observe that Personal Clouds apply *distinct internal control policies* to the inbound/outbound bandwidth provided to users. To wit, both Dropbox and Box exhibit an *upload transfer capacity remarkably better than the download capacity*. This means that the datacenter outgoing traffic is more *controlled and restricted* than the incoming traffic. This agrees well with the current pricing policies of major Cloud providers (Amazon S3, Google Storage) which do not charge inbound traffic whereas the outbound traffic is subject to specific rates (see `http://aws.amazon.com/en/s3/pricing/`).

In SugarSync, both the upload and download transfer speeds are constant and low. Interestingly, SugarSync presents slightly faster downloads than uploads, though only a small fraction of downloads (less than 1%) exhibits a much higher transfer speed than the rest. These observations are also supported by Fig. 31c and Fig. 31d: the captured download bandwidth values fall into a small range $[200, 1300]$ KB/sec. Also, the shape of these distributions are not steep, which reflects that there is a strong control in the download bandwidth. On the contrary, upload bandwidth distributions present more irregular shapes and

they cover a wider range of values, specially for Box. As a possible explanation to this behavior, the experiments of Fig. 31 were executed from our university labs (Spain) to exclude the impact of geographic heterogeneity. Considering the fact that the majority of Personal Cloud datacenters are located in USA [19], this may have implications in the cost of the traffic sent to Europe. This could motivate the enforcement of more restrictive bandwidth control policies to the outbound traffic.

As seen in Fig. 32a and 32c, both Dropbox and Box download file MTS *can be approximated using log-logistic or logistic distributions, respectively*. This argument is supported by the coefficient of determination, R^2, which in the case of Box is $R^2 = 0.9972$, and for Dropbox is $R^2 = 0.9957$. However, we observe that these fittings differ from the empirical data in the tails of highest transfer speed values. Further, we performed fittings depending on the file size, obtaining closer fittings as the file size grew. The heavier tails found in empirical data but not captured well in the fittings led the KS test to reject the null hypothesis at significance level $\alpha = 0.05$, although in the case of Dropbox, this rejection is borderline (KS-test=0.0269, critical value=0.0240, p-value=0.197).

Regarding uploads, we find that Dropbox file MTS *can be modelled by a Weibull distribution* with shape parameter $\mu = 1339.827$ and scale parameter $\sigma = 14.379$ (Fig. 32b). In addition to the high $R^2 = 0.9896$, the KS test *accepted the null hypothesis* at significance level $\alpha = 0.05$ (KS-test=0.0351, critical value=0.0367, p-value=0.0025).

Due to the high variability, we found that Box uploads do not follow any standard distribution. The implications of these observations are relevant. With this knowledge, researchers can model the transfer speed of Personal Cloud services employing specific statistical distributions.

Transfers & geographic location. Next, we analyse transfer speeds depending on the geographic location of vantage points.

In Fig. 33, we illustrate the file MTS obtained from executing the up/down workload during 3 weeks in PlanetLab. As can be seen in the figure, *Personal Clouds provide a much greater QoS in North American countries than in European countries*. Intuitively, the location of the datacenter plays a critical role in the performance of the service delivered to users. Observe that this phenomenon is orthogonal to all the examined vendors.

Finally, we quantify the relative download/upload transfer performance delivered by each service as a function of the geographic location of users. To this end, we used a simple metric, what we call the *download/upload ratio* (D/U), which is the result of dividing the download and upload transfer speeds of a certain vendor. In Table 7.5b, we calculated this ratio over the mean ($\bar{U}, \bar{D}$) and median ($\tilde{U}, \tilde{D}$) values of the file MTS distributions of each provider depending on the geographic location of nodes.

In line with results obtained in our university labs, *European nodes receive a much higher transfer speed when uploading than when downloading* ($D/U < 1$). However, contrary to conventional wisdom, *North American nodes exhibit just the opposite behaviour*. This is clearly visible in Dropbox and Box. However, this ratio is constant in SugarSync, irrespective of the geographic location.

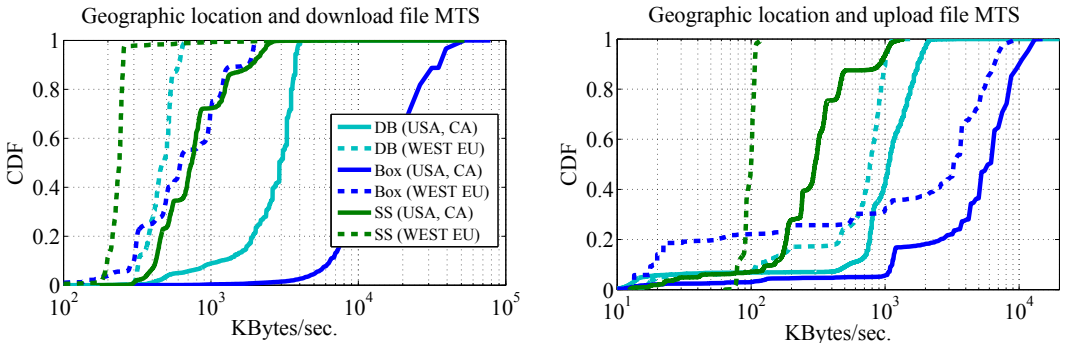


Figure 33: File MTS distributions of PlanetLab nodes from June 22 to July 15 2012 depending on their geographic location (up/down workload). Clearly, USA and Canada nodes exhibit faster transfers than European nodes.

Geo. Location	*Metric*	Box	Dropbox	SugarSync
USA & CA	$\bar{D}_{MTS}/\bar{U}_{MTS}$	3.198	2.482	2.522
	$\tilde{D}_{MTS}/\tilde{U}_{MTS}$	2.550	2.722	2.500
WEST EU	$\bar{D}_{MTS}/\bar{U}_{MTS}$	0.255	0.681	2.589
	$\tilde{D}_{MTS}/\tilde{U}_{MTS}$	0.190	0.682	2.387

Table 7.5b: Download/Upload transfer speed ratio of Personal Clouds depending on the client's geographic location.

Variability of Transfer Performance

In this section, we analyse which factors can contribute to the variance in transfer speed observed in Personal Clouds. We study three potential factors, which are *the size of file transfers*; *the load of accounts*; and *time-of-day effects*.

Variability over file size. We first investigate the role that file size plays on *transfer times* and *transfer speeds*. Fig. 34 and Table 7.5c report the results for both metrics as function of file size, respectively. Unless otherwise stated, results reported in this subsection are based on executing the up/down workload in our university labs during 5 days.

Fig. 34 plots the transfer time distribution for all the evaluated Personal Clouds. As shown in the figure, for the same provider, all the distributions present a similar shape, which suggests that *the size of file transfers is not a source of variability*. As expected, the only difference is that the distributions for large file sizes are shifted to the right towards longer time values. Significant or abnormal differences were not observed when transferring large files compared to small data files. This observation is applicable to all evaluated Personal Clouds. This leads us to the conclusion that these Personal Clouds *do not perform aggressive bandwidth throttling policies to large files*.

An interesting fact appreciable in Table 7.5c is that *managing larger files report better transfer speeds than in case of small files*. Usually, these improvements are slight or moderate (0.5% to 25% higher MTS); however, uploading 100MB files to Box exhibits a MTS 48% higher than uploading 25MB files to this service. In our view, this phenomena is due to the variability in the incoming bandwidth supplied by Box, and the TCP slow start mechanism, which makes difficult for small file transfers to attain high performance [20].

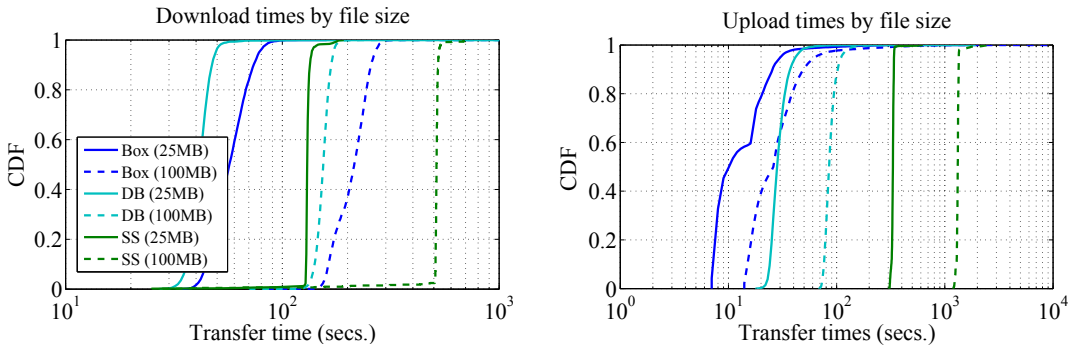


Figure 34: Transfer times distributions by file size.

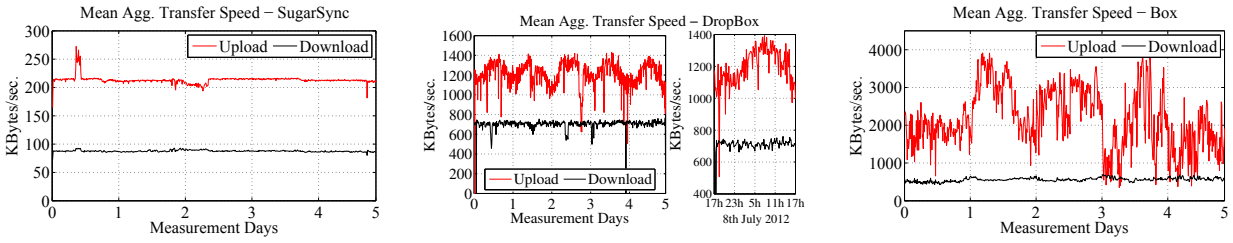
		Upload File MTS Distribution (KBps)								Download File MTS Distribution (KBps)							
Size	Provider	Min.	Q1	Median	Q3	Max.	Mean (μ)	Std. Dev. (σ)	CV (σ/μ)	Min.	Q1	Median	Q3	Max.	Mean (μ)	Std. Dev. (σ)	CV (σ/μ)
25MB	Dropbox	13.54	819.20	903.94	1008.24	1456.36	896.28	151.56	0.1691	24.89	582.54	624.152	672.16	970.90	626.94	71.23	0.1136
	Box	14.70	1379.71	2383.13	3276.80	3744.91	2271, 29	973.06	0.3963	163.84	397.19	459.90	534.99	794.38	463.72	87.76	0.0837
	SugarSync	41.87	78.25	78.96	80.17	86.23	79.26	2.82	0.0356	136.53	198.59	200.11	201.65	1048.57	201.35	37.89	0.1882
50MB	Dropbox	213.99	970.90	1092.27	1191.56	1497.97	1069.12	152.23	0.1424	210.56	624.15	663.66	699.05	888.62	661.55	58.02	0.0877
	Box	5.26	2496.61	4369.07	4766.25	5825.42	3721.12	1357.18	0.3647	14.15	623.16	647.26	672.16	887.42	646.22	44.33	0.0686
	SugarSync	40.27	78.72	79.44	80.41	86.95	79.59	3.08	0.0387	144.43	200.88	202.43	204.00	2496.61	216.57	149.28	0.6893
100MB	Dropbox	250.26	1127.50	1219.27	1310.72	1519.66	1205.69	143.05	0.1186	25.09	647.27	676.50	708.49	1497.97	680.32	50.94	0.0749
	Box	4.71	2912.71	3883.61	6168.09	7489.83	4350.37	1797.32	0.3252	14.43	436.91	487.71	579.32	1233.62	507.82	89.36	0.0539
	SugarSync	42.23	78.96	79.62	80.66	87.31	79.64	3.74	0.0470	145.64	202.03	204.00	205.20	3744.91	223.49	219.50	0.9822

Table 7.5c: Summary of file MTS distributions by file size.

Further, we found that all the measured Cloud vendors tend to perform a more restrictive bandwidth control to outgoing traffic. This can be easily confirmed by inspecting the obtained standard deviations σ of file MTS listed in Table 7.5c. Clearly, *the inbound traffic in Dropbox and Box is much more variable than the outbound traffic*. On the contrary, despite its limited capacity, the source of highest transfer variability in SugarSync is in the outbound traffic, which a clear proof of the existing heterogeneity in Personal Clouds.

Variability over load account. Next we explore if Personal Clouds perform any control to the transfer speed supplied to users based on the amount of data that users have in their accounts. To reveal any existing correlation, dispersion graphs were utilized to plot the relationship between the MTS and the load of an account at the instant of the storage operation.

As shown in Fig. 36, *we were unable to find any correlation between the file MTS and the load of an account* in any of the measured Personal Clouds. This suggests that the transfer speed delivered to users remains the same irrespective of the current amount of data stored in an



(a) SugarSync does not present important changes in both in/out traffic speed over time.

(b) We observe daily patterns in the Dropbox upload transfer speed. Download transfer speed remains stable.

(c) Box upload transfers are highly variable and probably affected by daily patterns.

Figure 35: Evolution of Personal Clouds upload/download transfer speed during 5 days. We plotted in a time-series fashion the mean aggregated bandwidth of all nodes (600 secs. time-slots) executing the service variability workload in our university laboratories (3rd−8th July 2012).

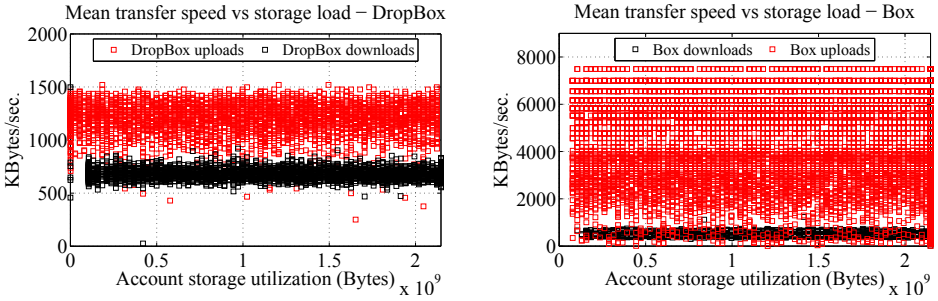


Figure 36: Relationship between file MTS and the storage load of an account.

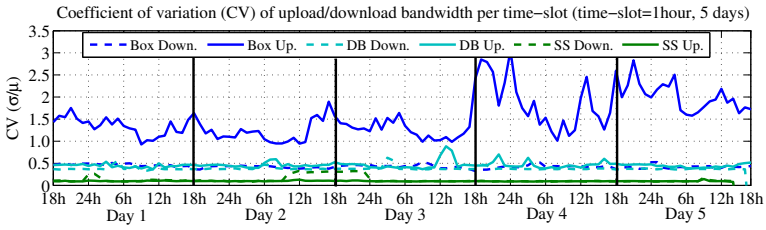


Figure 37: Evolution of transfer speed variability over time (service variability workload, university labs).

account. This conclusion is important to characterize which types of control mechanisms are actually applied to these storage services.

Variability over time. We now analyse how the transfer speed varies over time. To better capture these variations, we used the data from the *service variability workload*, which was aimed to maintain a constant transfer flow and was executed at our university labs. The results are shown in Fig. 35 where the *mean aggregated bandwidth* of all nodes as a whole is plotted in time intervals of 600 seconds. As expected, we found that the transfer speed of these services behave differently depending on the provider. To wit, while SugarSync exhibits a *stable service for both uploads and downloads*, at the price of a modest transfer capacity (Fig. 35a), the upload transfer speed varies significantly over time for Dropbox and Box.

Appreciably, *Dropbox exhibits appreciable daily upload speed patterns* (Fig. 35b). Data represented in Fig. 35 was gathered between July 3, 6:00p.m. and July 8, 3:00p.m. Clearly, during night hours (1 a.m.−10 a.m.), transfer speed was between 15% to 35% higher than during diurnal hours. This phenomenon has been also detected in the experiments performed in PlanetLab, thereby discarding any artificial usage pattern induced by our university network. Moreover, considering that Dropbox uses Amazon S3 as storage backend, our results are consistent with other recent works [21] that observed similar patterns in other Amazon services.

Further, we found that Box upload service may be subjected to high variability over time. Indeed, we observed *differences in upload transfer speed by a factor of* 5 *along the same day*. This observation is consistent with the analysis of the file MTS distribution where significant heterogeneity was present. More interestingly, Box uploads appear to be also affected by *daily patterns*. Concretely, the periods of highest upload speed occurred during the nights, whereas the lowest upload speeds were observed during the afternoons (3 p.m. −10 p.m.). Due to the huge variability of this service, a long-term measurement is needed to provide a solid proof of this phenomenon, though.

With respect to downloads, we observed no important speed changes over time in any system. This suggests that *downloads are more reliable and predictable, probably due to a more*

Downloads	Dropbox	Box	SugarSync
25MB	0.047%($\frac{5}{10,503}$)	0.572%($\frac{68}{11,878}$)	0.115%($\frac{2}{1,740}$)
50MB	0.082%($\frac{8}{9,745}$)	0.698%($\frac{80}{11,445}$)	0.057%($\frac{1}{1,727}$)
100MB	0.044%($\frac{4}{9,026}$)	0.716%($\frac{80}{11,169}$)	0.059%($\frac{1}{1,691}$)
150MB	0.042%($\frac{3}{7,136}$)	–	0.076%($\frac{1}{1,359}$)
Uploads	Dropbox	Box	SugarSync
25MB	0.384%($\frac{41}{10,689}$)	0.566%($\frac{227}{40,043}$)	0.889%($\frac{8}{899}$)
50MB	0.450%($\frac{48}{10,663}$)	1.019%($\frac{405}{39,719}$)	1.079%($\frac{10}{926}$)
100MB	0.502%($\frac{54}{10,740}$)	2.097%($\frac{836}{39,875}$)	1.988%($\frac{18}{905}$)
150MB	1.459%($\frac{58}{3,974}$)	–	3.712%($\frac{33}{889}$)

Table 7.5d: Server-side failures of API operations ($3_{rd} - 8_{th}$ July 2012).

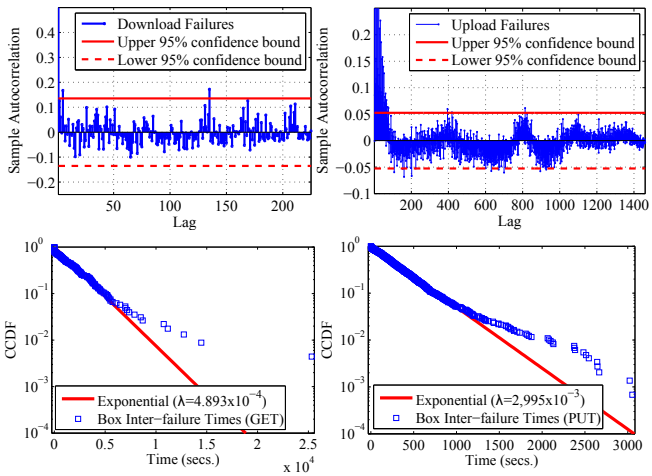


Figure 38: Failure interarrival times autocorrelation (upper graphics) and exponential fitting of failure interarrival times (lower graphics) for Box.

intense control of this type of traffic by the datacenter.

To specifically compare the variability among services over time, we made use of the *Coefficient of Variation* (CV), which is a dimensionless and normalized measure of dispersion of a probability distribution, specifically designed to compare data sets with different scales or different means. The CV is defined as:

$$CV = \frac{1}{\bar{x}} \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} (x_i - \bar{x})^2},$$

where N is the number of measurements; x_1, .., x_N are the measured results; and $\bar{x}$ is the mean of those measurements.

Fig. 37 depicts the CV in 1-hour time slots of the *aggregated bandwidth* provided by each Cloud vendor. Clearly, it can be observed important differences across the vendors. Concretely, SugarSync experiences low variability with a CV of only 10%. Dropbox with a CV around 50%, however, exhibits a much higher variability than SugarSync, including isolated spikes in the upload bandwidth that reach a CV of 90%. In this sense, the Box download bandwidth capacity exhibits a similar trend. Finally, the highest observed variability was for Box uploads. In the first 3 days of the experiment, Box exhibited a mean CV of 125% approx. However, in the last part of the experiment some spikes reached a CV of 300%, suggesting that it is really hard to predict the behaviour of this service.

Service Failures and Breakdowns

Another important aspect of any Cloud storage service is *at what rate* users experience failures, and *whether the pattern of failures can be characterized by a simple failure process like a Poisson process*, which allows researchers to develop tractable analytical models for Cloud storage.

For this analysis, any event *server-side* notification signaling that a storage operation did not finish *successfully* was counted as a *failure*, thereby excluding any failure, where abnormal or degraded service was observed[15]. Table 7.5d summarizes the server-side failures observed during a 5-day measurement based on the variability workload run at our labs.

Failure rates. Table 7.5d illustrates a clear trend: *in general, uploads are less reliable than downloads*. This phenomenon is present in all the Personal Clouds measured and becomes *more important for larger files*. As can be observed, downloads are up to 20X more reliable than uploads (Dropbox, SugarSync), which is an important characteristic of the service delivered to users. In this sense, although failures among uploads and downloads are not so high, Box seems to provide the least reliable service. Anyway, failure rates are generally below 1%, which suggests that these *free storage services are reliable*.

Poissonity of failures. Now we study whether service failures appear Poisson or not, because Poisson failures allow for easy mathematical tractability. Poisson failures are characterized by interarrival times which are independent of one another and are distributed exponentially [22], and for which the failure rate is constant. In this case, we focused only on Box, since it was the only service for which enough observations were available for the statistical analysis to be significant.

To verify whether failures are independent, we calculated the autocorrelation function (ACF) for consecutive failures in the time series and depicted it in Fig. 38[16]. When the failures are completely uncorrelated, the sample ACF is approximately normally distributed with mean 0 and variance $1/N$, where N is the number of samples. The 95% confidence limits for ACF can then be approximated to $0 \frac{2}{\sqrt{N}}$. As shown in Fig. 38, in the case of download failures, autocorrelation coefficients for most lags lie within 95% confidence interval, which demonstrates that failure interarrival times are independent of one another. However, uploads failures are not independent, since the first lags exhibit high ACF values, which indicates short-term correlation, with alternating positive and negative ACF trends.

To conclude Poissonity for failures, failure interarrival times must be exponentially distributed, for we report the *coefficient of determination*, R^2, after performing linear regression on the distribution $log_{10}(1 - \{Pr\{X < x\})$, where $Pr\{X < x\}$ is the empirical failure interarrival time distribution obtained for Box. In the case of downloads $R^2 = 0.9788$ whereas for uploads $R^2 = 0.9735$. This means that failure interarrival times *approximately follow an exponential distribution*, which is evidenced in Fig. 38, where most of the samples match the exponential fitting, with the exception of those at the end of the tail. Hence, Box download failures can be *safely considered as being Poisson*. Although upload interarrival times can be

[15]We filtered the logged error messages depending on their causes as detailed in the API specifications. We considered as errors most of the responses with $5XX$ HTTP status codes as well as other specific errors related with timed out or closed connections in the server side.

[16]Due to lack of space, we refer the reader to [22] for a technical description in depth of this methodology to assess Poissonity.

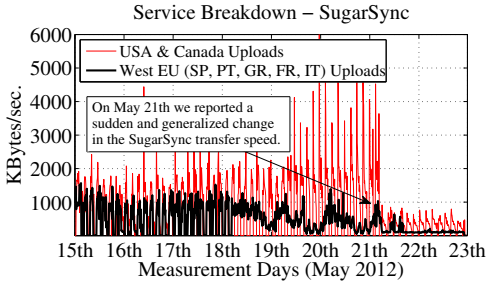


Figure 39: We observe a radical change in the upload transfer speed of SugarSync from May 21 onwards. After May 21 all the tests performed against SugarSync reported very low transfer speeds. This reflects a change in the QoS provisioned to the REST APIs of free accounts.

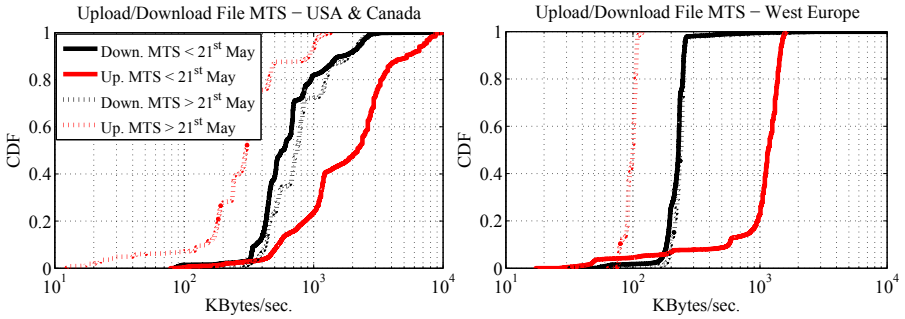


Figure 40: PlantLab experiments against SugarSync before and after the service breakdown reported in May 21. We observe an important service degradation for uploads, whereas the download service remains unaltered.

well fitted by the exponential distribution, they are not independent and further analysis is needed to their characterization.

Service breakdowns. Apart from the "hard" failures, there are other types of "soft" failures related with the deterioration of the QoS. And indeed, we captured a strong evidence of this in late May 2012 (Fig. 39).

In Fig. 39 we present a time-series plot of the aggregated upload MTS of PlanetLab nodes against SugarSync. This information is divided for those nodes located in West Europe and USA & Canada[17].

Clearly, the behavior of the upload speed of SugarSync changed radically from May 21 onwards (Fig. 39). Before that date, SugarSync provided high transfer upload speed, comparable to current performance of Box. However, in May 21 SugarSync bandwidth provisioning policies changed dramatically; the upload MTS was reduced from 1,200KBps to 80KBps in Western Europe — a similar trend can be observed in USA and Canada. Note that we accessed to the SugarSync service from a variety of nodes and accounts, discarding thus the possibility of IP filtering and account banning.

In this sense, Fig. 40 shows the upload/download MTS distributions for measurements performed before and after the service breakdown —executing the same workload (up/-down workload) over the same nodes. Clearly, the change in the transfer speed of SugarSync was focused on uploads, that previously exhibited a good performance. On the other hand, we see that the download service was almost unaltered after May 21. These observations ap-

[17] Spikes present in Fig. 39 are due to the PlanetLab quota system, which limits the amount of data that users can transfer daily.

ply to both geographic regions. This means that Personal Clouds may change their *freemium* QoS unexpectedly, due to internal policy changes.

7.5.3 Conclusions

To conclude, we summarize the most relevant technical observations obtained from this measurement:

Characterization of transfers. In some cases, we observed that transfer time distributions can be characterized by known statistical distributions like the *log-logistic* and the *logistic* for downloads in Dropbox and Box, respectively. We also found that upload transfer times are Weibull distributed in Dropbox. In SugarSync, we observed a constant and very limited transfer performance. This characterization opens the door to create Personal Cloud modeling and simulation environments.

High service variability. The variability of Personal Cloud services is significant and induced by many factors. To wit, we discovered that uploading to Dropbox is substantially faster at nights ($15\% - 35\%$), which proves the presence of daily usage patterns. We also found that the magnitude of the variation is not constant over time. An example of this is Box. While Box uploads exhibited a mean variability of 125% at the beginning of our experiment, the CoV reached 300% at the end. Further, we found that *uploads are more variable than downloads*.

Reliability and Poissonity of failures. In general, we found that Personal Clouds are *reliable*, exhibiting failure rates below 1%. We also found that for Box, failure interarrival times approximately follow an exponential distribution. Moreover, Box download failures can be modeled as a Poisson process, which is analytically simple.

QoS changes and data lock-in. We found that SugarSync changed its *freemium* QoS unexpectedly. Concretely, the mean upload speed delivered by SugarSync suddenly dropped from $1,200$ KBps to 80 KBps in EU. This emphasizes the relevance of the *data lock-in* problem, where a customer gets trapped in a provider whose service is unsatisfactory but cannot move to a new one because of the amount of data stored in it.

In this measurement, we have examined central aspects of Personal Cloud storage services to characterize their performance, with emphasis put on the data transfers. We have found interesting insights such as the high variability in transfer performance depending on the geographic location, the type of traffic, namely inbound or outbound, the file size, and the hour of the day. We have examined their failure patterns and found that these services are reliable, but susceptible to unpredictable changes in their quality of service as that witnessed in SugarSync.

8 Demonstration scenarios and use cases

8.1 StackSync demo

We will show how StackSync synchronizes files between different devices. We will put files and folder into the synchronization folder and observe how the overlay icons change as the state of files goes from synchronizing to synchronized. Once a file is synchronized, we will look at another device to see how changes are quickly propagated among all synchronized devices thanks to our push notifications.

Furthermore, we will use the mobile application to assess that modifications in the file system are also perceived by the API. We will modify the synchronized files through the mobile application by uploading and deleting files, and creating new folders to see how they affect other synchronized devices.

Finally, we will disconnect Internet to demonstrate how StackSync can handle offline modifications and how they are treated once the connection is recovered. We will also cause synchronization by modifying files from two devices at the same time in order to inspect the behaviour of clients when conflicts are detected.

8.2 Horizontal interoperability

The horizontal interoperability demonstration will show two heterogeneous Personal Clouds are able to interoperate by sharing a folder between distinct users and seamlessly accessing to the folder. StackSync and Ubuntu One will act as the heterogeneous Personal Clouds implementing our service platforms to demonstrate the feasibility of our solution.

A user in StackSync will create a sharing proposal. To do that, it will go to the StackSync administration website and share one of its files with an external user, indicating the email of the addressee. The addressee will click on the link in the email and will be redirected to the StackSync website, where he will select Ubuntu One from a list of compatible Personal Clouds. Next, the addressee will be redirected to the Ubuntu One website where he will explicitely authorize the sharing proposal by providing his user credentials. Next, Ubuntu One will inform StackSync the result of the proposal and StackSync will hand over the access credentials to Ubuntu One.

From that moment, these two users will be able to share a folder. We will demonstate how modifications can be done either from StackSync and Ubuntu One, and how modifications made in one Personal Cloud are reflected on the other Personal Cloud.

Finally, we will show how users can cancel the sharing agreement at any time, and access to the shared folder is denied immediately.

8.3 Vertical interoperability

The vertical interoperability demonstration how external applications can make use of our service platform to create new services or applications on top of CloudSpaces. We will show

how eyeOS, and the applications inside it, will communicate transparently with a storage layer that is not homogeneous. As a proof-of-concept, eyeOS will take advantage of the service platform developed within the frame of this project and access to the services offered by StackSync and Ubuntu One.

We will show how the eyeOS file manager (eyeSync) will navigate through files and folders located seamlessly in any participant Personal Cloud, transparently accessing files through the storage API developed within the project. Furthermore, eyeOS will show how the eyeOS calendar (eyeCalendar) makes use of our persistence API as a backend to store user information and enable sharing functionalities. In addition, eyeDocs will be show as a proof-of-concept of group editing and collaboration running on top of Personal Clouds.

8.4 Privacy demo

In this section, we describe a prototype application intended to demonstrate part of the privacy aware data sharing techniques we are developing in CloudSpaces. We start by a typical usage scenario before moving to the different components we envision in this application.

8.4.1 Scenario

A user has a set of personal files on his desktop. These files are of various types and are of different importance with respect to him. The user can select to share a subset of these files with other parties through the cloud in a certain context (e.g. a via specific device at a specific time, location, etc.). Associated with this sharing operation is a privacy risk function, evaluated based on the sharing context. The system prototype recommends, for each data observer, the data sharing format (i.e. encrypted, anonymised, plaintext, etc.), which is determined based on the privacy risk.

8.4.2 Overview

An overview of the prototype is presented in Figure 41. It includes a decision making module at its core that gives the sharing recommendation. It takes as an input the sharing context and the privacy risk value. The latter is in turn a function of the sharing context itself. The output is a decision about data sharing and manipulation for each party with which the data is being shared.

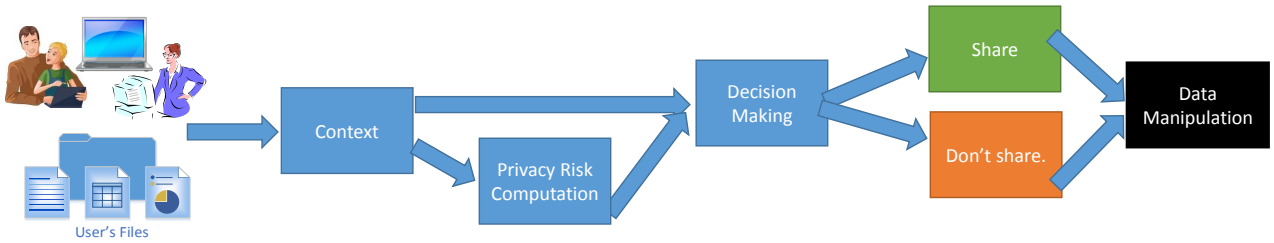


Figure 41: Prototype Components

8.4.3 Inputs

We begin by specifying the inputs that this module takes, which are the sharing context and the privacy risk value.

Sharing Context The sharing context shall be based on the contents and semantics of the data. The semantics can include the metadata associated with the data item and also other information such as the device from which the data item is being shared, the type of data observer with whom the data is being shared (e.g., friend, family, colleague), etc.

Privacy Risk Evaluation The risk is a function of the sharing context. It quantifies the value of the privacy loss from sharing the data in that context. Its output is specific to a certain party with which the data is being shared.

8.4.4 Decision Making

This module decides on the optimal privacy policy to apply given the data, the context, and the privacy risk. The policy is realized through two decisions:

1. **Data Sharing:** The module decides on whether the data should be shared with the entity being considered.
2. **Data Manipulation:** This is a decision about the form in which the data should be sent to the cloud. This could take at least three forms: unencrypted (non-edited) data, encrypted data, and data modified via a privacy enhancing technique.

8.5 Adaptive Storage

8.5.1 Hybrid storage demo

By the end of the first year, we will show how the preliminary HCS implementation with a RMDS deployed on a single Zookeeper server and with data replicated across several clouds. Fetching updates will be done from the closest cloud that stores a valid data item - this will demonstrate good performance of HCS. Additionally we will demonstrate how data corruption at a single untrusted cloud is masked in HCS by the existence of other non-corrupted clouds, entirely transparently to the user. Data will be stored across actual commercial cloud providers such as Amazon S3, Windows Azure, Rackspace, etc.

Additionally to the demo, we will provide benchmark results to state of the art reliable cloud storage solutions that show superiority of our Hybrid cloud storage. To this end we will use established cloud benchmarks such as Yahoo Cloud Serving Benchmark (YCSB), among others.

Eventually, towards the end of the project, the goal is to integrate HCS with StackSync and demo the entire system using the workloads provided by Ubuntu One.

8.5.2 Adaptive content distribution demo

By the end of the first year, we will show how the architecture reacts when many clients try to download the same file. To show in real time how the system evolves, we have created a real-time monitoring tool.

This tool will show some charts separated in three different sections: global information, cloud related information and a peers graph.

In the first section, it will show in a chart the requests done to the server. This will differentiate between HTTP and BitTorrent requests. It will also draw some charts comparing the amount of pieces transferred with both protocols. Thus, when the demonstration begins, the clients will connect to the service in a flash-crowd or a Poisson distribution. At the same time that clients are connecting and syncing data, a real-time plot will show the amount of data transferred. During the initial moments, all the bandwidth will be covered by the HTTP protocol, but when the clients change to the BitTorrent protocol, it will be possible to see in real-time how the BitTorrent contribution becomes more important.

Another important information that will be shown in this chart is the cloud contribution against the clients one. When all clients finish downloading, their contribution will be greater than the cloud contribution since all of them possess all pieces and the rest of peers can ask them for any piece. Clients will always try to get data from other clients to minimize the amount of data transferred from the cloud.

We will also monitor the number of requests the initial seed has to make to the cloud in order to obtain the pieces. There will be a chart comparing the pieces obtained from OpenStack and the ones obtained from the seed cache.

The last feature of the monitoring tool is a real-time peers graph where it will draw the swarm evolution: peers connected and their relation to other peers.

At the end of the demonstration we will have a set of clients with the file synced ready to be used by the user.

In order to validate our architecture, we developed a special tool that can monitor the traffic between the components of the system and give real time information about the status of the transfers. Our tool offers the following features:

- Real-time measurements of different events and physical parameters of the system;
- Message queueing service between the different parts of the system and the graphic components;
- Dynamic charts to visualize the download progress.

These features are further detailed in the following sections.

Real-time measurements

To assess the performance of our model, we provide mechanisms for real-time measurement of the parameters of our system. These parameters can be physical (e.g. available

bandwidth, consumed bandwidth) or they can be in the form of execution events (e.g. a peer joining or leaving the swarm, sending a piece to another peer)

Message-queueing system

To capture the previously stated parameters, we set up different sensors for each component of the system. The captured events are stored in a queue and consumed by the monitor responsible for updating the related graphical components.

Dynamic charts

Different statistical charts are included in our monitor in order to give us a real time projection of all the execution states. These charts are used to measure the evolution of different parameters both on the Cloud's and the peer's sides.

Cloud-Related Charts In order to measure the contribution of our model in offloading the Cloud's serving, we include a line chart representing the evolution of the cumulative number of clients and the number of chunk-requests addressed to the Cloud over time. This chart is used to show that with increasing number of clients, the number of chunks requested from the Cloud is decreasing as the clients will be exchanging chunks among one-another. We measure also the contribution of an eventual local cache system integrated within the BitTorrent server. The cache's role is to save the chunks already requested from OpenStack Swift's storage nodes in a local memory to avoid repeated requests of the same chunk. We measure the percentage of the requests addressed to the cache in comparison with those from the the OpenStack Swift's storage nodes and we plot it in a dynamic pie-like chart.

The Network's Topology Graph To be able to track the evolution of the transfers in the system, our monitor offers a real-time view of all the peers in the network in the form of a directed graph whose vertices are the nodes of the system and whose edges represent transfers in the form of arrows directed from the uploader to the downloader.

Global Charts In order to measure the contribution of each component in the download process, we represent also the evolution of the all the chunks' requests sent through the system depending on their nature: HTTP or BitTorrent, and in the BitTorrent's case whether they are addressed to the Cloud seed or are the result of peers exchange.

References

[1] S. Balasubramaniam and B. C. Pierce, "What is a file synchronizer?" in Proc. of ACM/IEEE MobiCom, 1998, pp. 98–108.

[2] D. Aksoy and M. S.-F. Leung, "Pull vs push: a quantitative comparison for data broadcast." in GLOBECOM. IEEE, 2004, pp. 1464–1468.

[3] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik, "Broadcast disks: data management for asymmetric communication environments," in Proceedings of the 1995 ACM SIGMOD international conference on Management of data, ser. SIGMOD '95. New York, NY, USA: ACM, 1995, pp. 199–210.

[4] S. Acharya, M. Franklin, and S. Zdonik, "Balancing push and pull for data broadcast," in Proceedings of the 1997 ACM SIGMOD international conference on Management of data, ser. SIGMOD '97. New York, NY, USA: ACM, 1997, pp. 183–194.

[5] "How we've scaled dropbox," http://www.youtube.com/watch?v=PE4gwstWhmc.

[6] "Idc - press release. apple cedes market share in smartphone operating system market as android surges and windows phone gains," http://www.idc.com/getdoc.jsp?containerId=prUS24257413.

[7] C. Basescu, C. Cachin, I. Eyal, R. Haas, A. Sorniotti, M. Vukolić, and I. Zachevsky, "Robust data sharing with key-value stores," in DSN, 2012, pp. 1–12.

[8] C. Cachin, D. Dobre, and M. Vukolić, "Brief announcement: Bft storage with 2t+1 data replicas," in DISC, 2013, to appear. Also available as arXiv report abs/1305.4868 (http://arxiv.org/abs/1305.4868).

[9] A. N. Bessani, M. P. Correia, B. Quaresma, F. André, and P. Sousa, "Depsky: dependable and secure storage in a cloud-of-clouds," in EuroSys, 2011, pp. 31–46.

[10] D. Dobre, G. Karame, W. Li, M. Majuntke, N. Suri, and M. Vukolić, "Powerstore: Proofs of writing for efficient and robust storage," in ACM CCS, 2013.

[11] J. Ball, "NSA's prism surveillance program: how it works and what it can do," The Guardian, Jun. 2013. [Online]. Available: http://www.guardian.co.uk/world/2013/jun/08/nsa-prism-server-collection-facebook-google

[12] B. Krishnamurthy, "Privacy and online social networks: Can colorless green ideas sleep furiously?" IEEE Security & Privacy, vol. 11, no. 3, pp. 14–20, 2013.

[13] F. Drasgow and C. L. Hulin, "Item response theory," Handbook of industrial and organizational psychology, vol. 1, pp. 577–636, 1990.

[14] B. Klimt and Y. Yang, "Introducing the enron corpus." in CEAS, 2004, cited by 0263. [Online]. Available: ftp://ftp.research.microsoft.com/users/joshuago/conference/papers-2004/168.pdf

[15] "NUIX - EDRM enron data set," cited by 0000. [Online]. Available: http://www.nuix.com/enron

[16] "Yahoo cloud serving benchmark." [Online]. Available: http://labs.yahoo.com/news/yahoo-cloud-serving-benchmark/

[17] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," SIGCOMM Comput. Commun. Rev., vol. 33, no. 3, pp. 3–12, 2003.

[18] E. Hammer-Lahav, "The OAuth 1.0 Protocol," http://tools.ietf.org/html/rfc5849, 2010.

[19] I. Drago, M. Mellia, M. M. Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside dropbox: understanding personal cloud storage services," in ACM IMC'12, 2012, pp. 481–494.

[20] M. Allman, V. Paxson, W. Stevens et al., "Tcp congestion control," 1999.

[21] A. Iosup, N. Yigitbasi, and D. Epema, "On the performance variability of production cloud services," in CCGRID'11, 2011, pp. 104–113.

[22] M. S. Artigas and E. Fernández-Casado, "Evaluation of p2p systems under different churn models: Why we should bother," in Euro-Par, 2011, pp. 541–553.