cloudspaces

SEVENTH FRAMEWORK PROGRAMME

SEVENTH FRAMEWORK PROGRAMME

CloudSpaces

(FP7-ICT-2011-8)

Open Service Platform for the Next Generation of Personal Clouds

D2.4 Reference implementation of architectural building blocks

Due date of deliverable: 30-09-2015
Actual submission date: 06-10-2015

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	65
WP/Task related to this document	WP3
WP/Task responsible	URV
Author(s)	Pedro García López, Cristian Cotes González, Raúl Gracia Tinedo
Partner(s) Contributing	URV
Document ID	CLOUDSPACES_D2.4_141031_Public.pdf
Abstract	In this document, we present the final StackSync architecture with all its components and we explain how to deploy and manage them in a production environment. Furthermore, we present an analysis of the Ubuntu One (U1) trace, providing interesting insight on the functioning and dynamics of the U1 service.
Keywords	Cloud storage, synchronization, sharing, interoperability, Personal Cloud

Table of Contents

1 Executive summary

The purpose of the present CloudSpaces project's deliverable D2.4 (Reference implementation of architectural building blocks) is twofold. Firstly, we describe the final architecture of StackSync. Secondly, we present an analysis of the Ubuntu One (U1) trace.

In Section 3, we explain in detail all the basic components of the final StackSync architecture released. During the project, StackSync has been used to validate essential contributions such as the Personal Cloud interoperability between StackSync and NEC, the privacy-aware data sharing developed by EPFL or the adaptative personal storage created by Institute Eurecom. In this section we also review the changes that we introduce in the architecture during the project.

In Section 4 there is a complete guide to install the basic components of StackSync, while in Section 5 we explain how to install and configure the StackSync website and the REST API.

Finally, the analysis of the U1 trace is described in Section 7. In this section, we first explain in detail the U1 architecture. Next, we deeply analyse the storage workload, the users' behaviour and the metadata back-end performance.

2 Introduction

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.

Although we give much credit to 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

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

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 StackSync reference architecture

In general terms, StackSync can be divided into five main blocks: clients, synchronization service, quota service, storage back-end, and communication middleware. An overview of the architecture with the main components and their interaction is shown in Figure 1. 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.

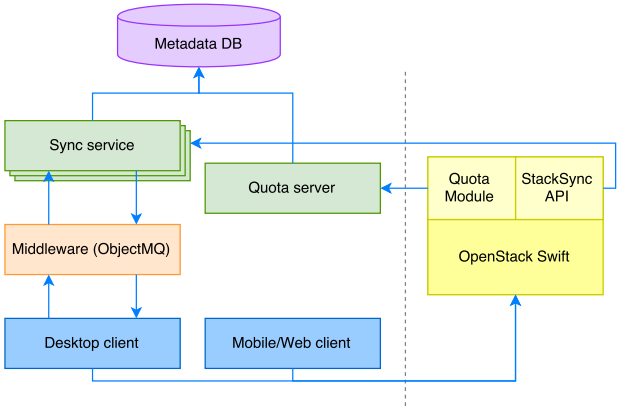


Figure 1: StackSync architecture overview

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.

3.1 Storage back-end: OpenStack Swift

Decoupling metadata from raw data allows us to use different storage back-end solutions. For our reference architecture we decided to use OpenStack Swift, an open-source highly available and scalable object storage.

Swift architecture is composed of proxies and storage nodes. The Proxy Server is responsible for tying together the rest of the Swift architecture. For each request, it will look up the

location of the object in the ring and route the request accordingly. The Storage nodes are responsible for running the object, container and account servers.

Object server The Object Server is a very simple blob storage server that can store, retrieve and delete objects stored on local devices. Objects are stored as binary files on the filesystem with metadata stored in the file's extended attributes (xattrs). This requires that the underlying filesystem choice for object servers support xattrs on files.

Container server The Container Server's primary job is to handle listings of objects. It doesn't know where those object's are, just what objects are in a specific container. The listings are stored as sqlite database files, and replicated across the cluster similar to how objects are. Statistics are also tracked that include the total number of objects, and total storage usage for that container.

Account server The Account Server is very similar to the Container Server, excepting that it is responsible for listings of containers rather than objects.

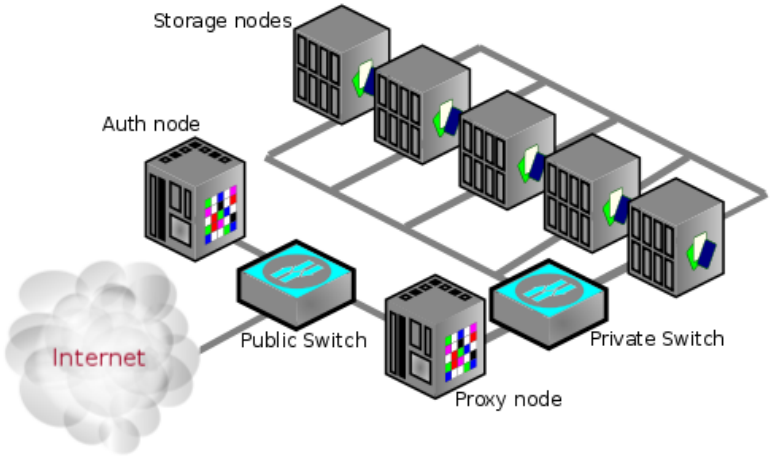


Figure 2: OpenStack Swift architecture

As we can observe in Figure 2, 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.

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.

3.2 Synchronization server

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

When a client connects to the system, the first thing it does is to ask 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 become aware of 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 notifications and all devices are always synchronized. However, if the metadata is incorrect, a notification is sent to the client to fix the error.

3.2.1 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 etc.
- **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 3.

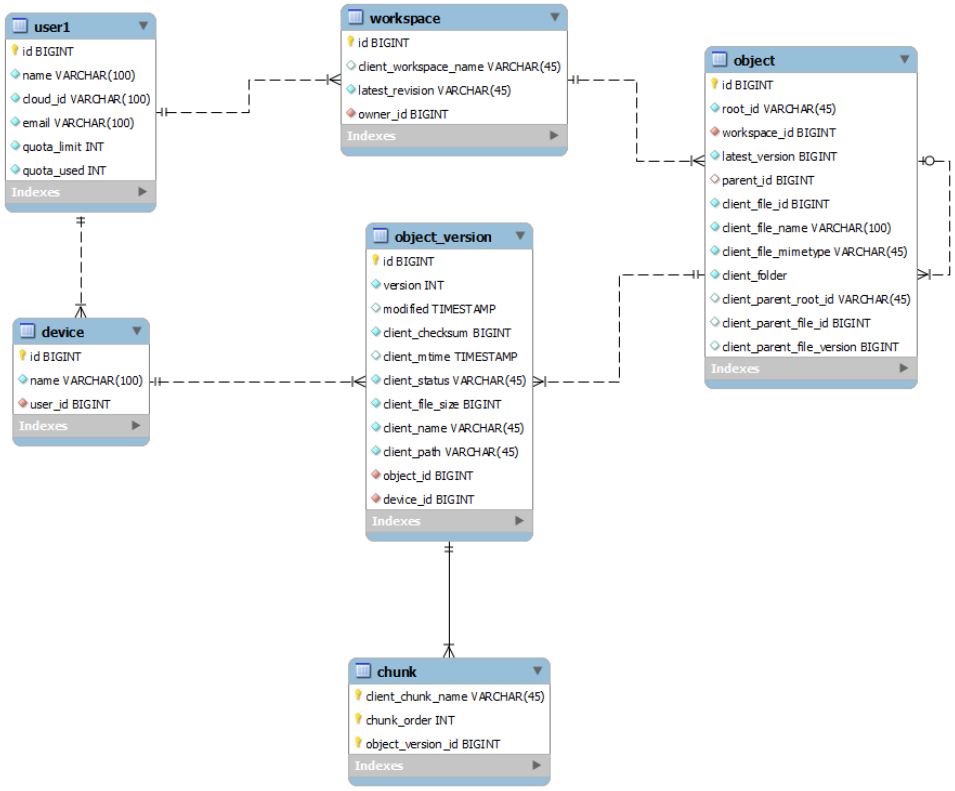


Figure 3: SyncService database model.

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 4 we can observe the class diagram that corresponds to the database DAO.

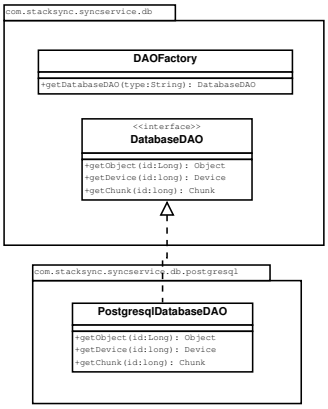


Figure 4: Class diagram of the database DAO in the SyncService.

3.2.2 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 1 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.

Algorithm 1 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
```

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, 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 5 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

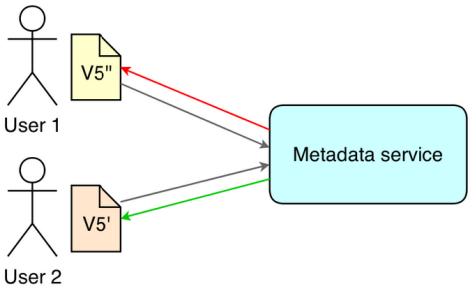


Figure 5: File conflict

a negative response to the second (User 1). User 1 will have to rename its version and download the correct one from the server.

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.

3.2.3 Communication: ObjectMQ

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 [1], 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 design a message-oriented middleware called ObjectMQ. 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 6: SyncService interface

In Fig. 6 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`.

3.3 Communication middleware

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 Quota server

The quota server is responsible to manage StackSync users' quota. When creating a new user, the administrator decides a storage quota limit. This quota delimits the maximum amount of storage that the user can use in the storage back-end.

The quota server is composed by two parts: the Swift module and a standalone server. Swift module intercepts the requests from the StackSync clients and checks if they have enough free storage in order to process them or not. Swift module communicates with the quota server to update and get quota values. Finally, quota changes are persisted in the StackSync metadata database.

If a user reaches the storage limit, he or she won't be able to upload new data until some files are removed and some storage is freed.

3.5 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.

3.5.1 Design

Following the REST principles, the API will be organized by resources [5]. 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 7 we observe a use case diagram representing the actions that a user can make to the API.

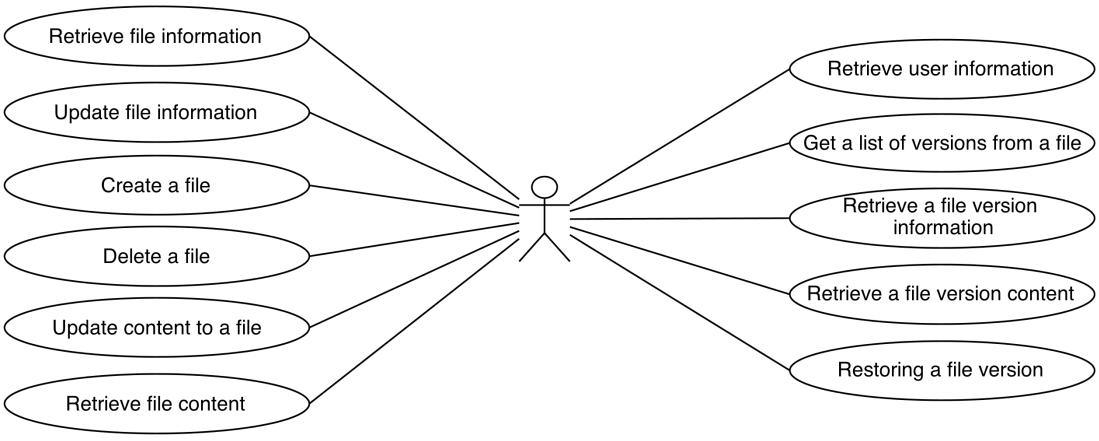


Figure 7: Storage API use cases diagram

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.

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.

3.6 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.
- **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 8 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 8: Overlay icons.

As the client is mostly developed in Java, it can be easily executed in multiple OS. However, there are some parts that are slightly different in each operating system, as it is the case of the overlays. Overlay icons are drawn in file explorers, and each file explorer has its own way to show them. For this propose, we rely on third party libraries and integrate them with the StackSync client. At this moment we have installers for the three major OS: Windows 7 (Explorer), Debian (Nautilus) and MacOS X (Finder).

3.6.1 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 transfers of big files.** 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`

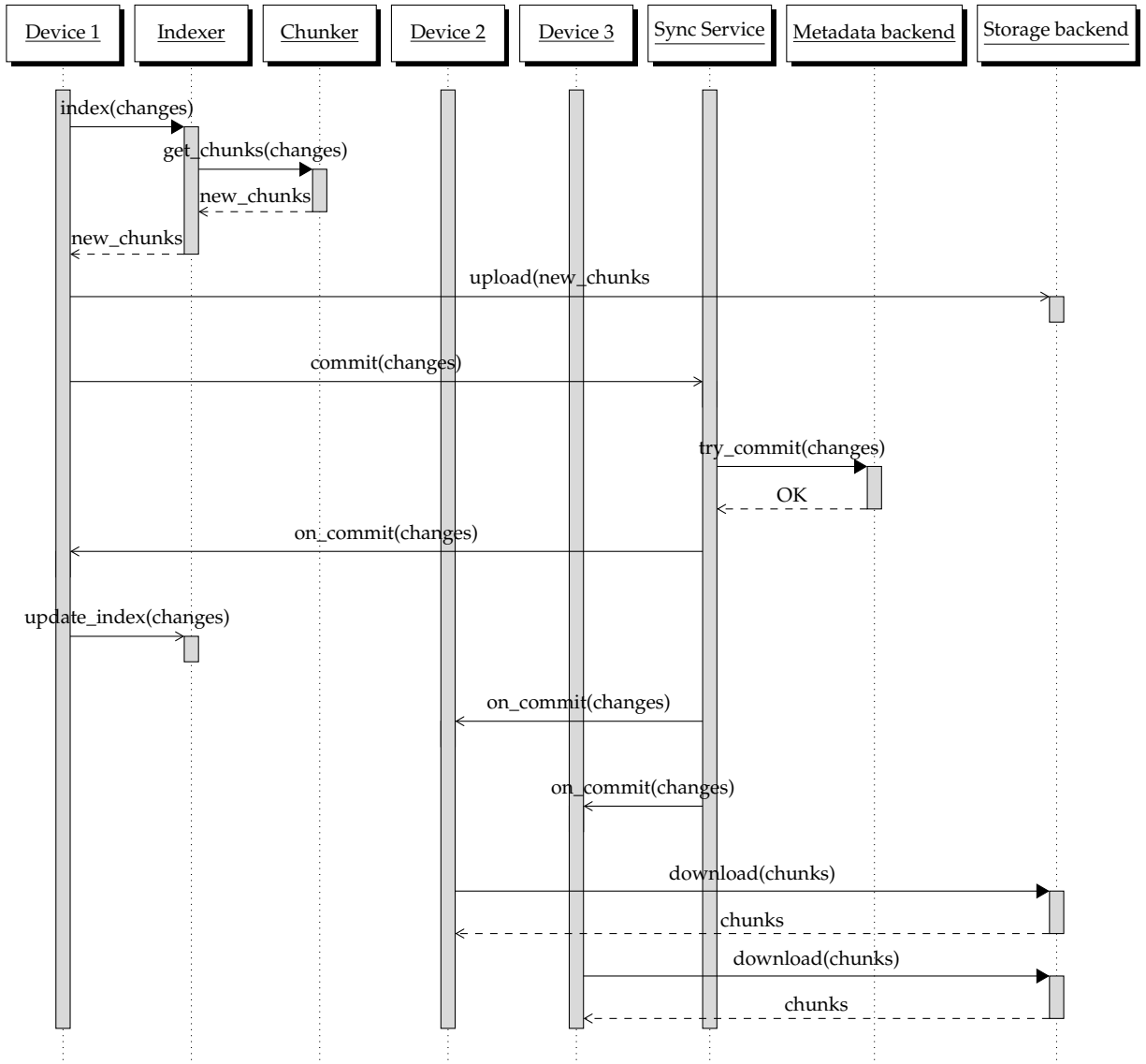


Figure 9: Interaction between the components of sync engine for a Personal Cloud.

`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.

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

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 [6], 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 9 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.

3.6.2 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, 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.

3.6.3 Architecture

In Figure 10 we can observe the architecture of the desktop client. Next, we will detail each element and explain the relation between them.

- **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.

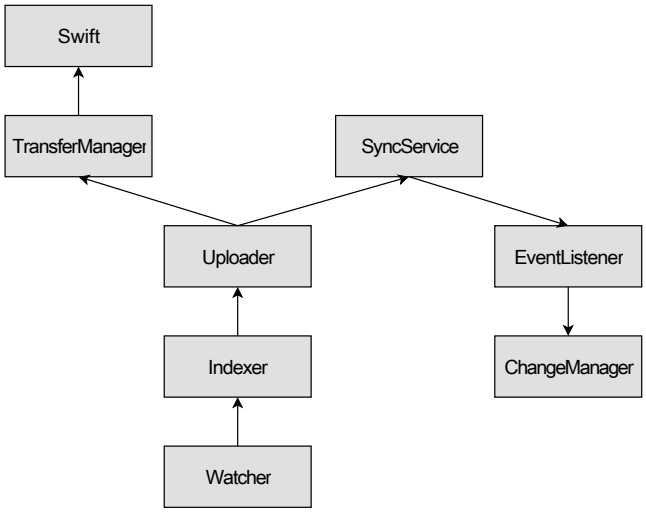


Figure 10: Desktop client architecture.

- **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 2.

Algorithm 2 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
```

3.6.4 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 higher. 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) [7]. 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 11 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.

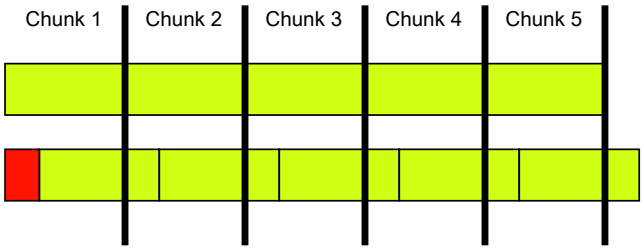


Figure 11: Static chunking example.

In the Algorithm 3 we provide the pseudocode for the TTTD content-based chunking.

Algorithm 3 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
```

3.6.5 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 12 we can observe the database model used in the desktop client.

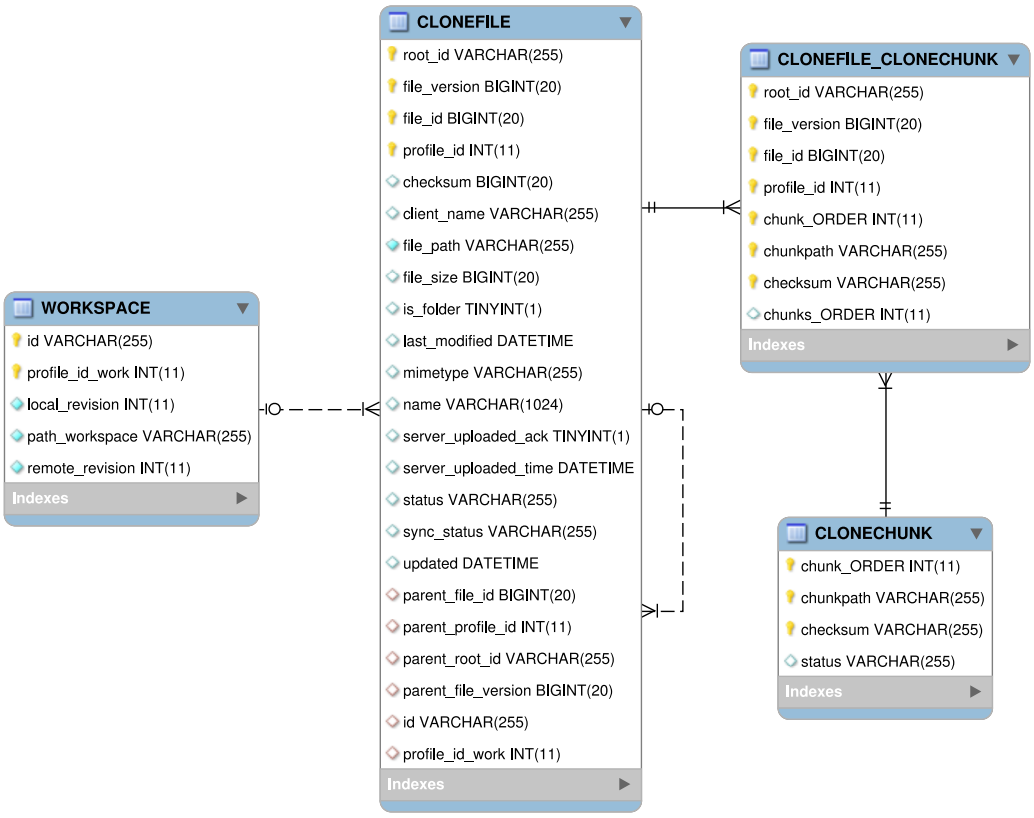


Figure 12: Database model of the StackSync desktop client

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, 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 which the user is allowed to access.

3.7 Mobile clients

Nowadays people are on the move and almost everyone 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.

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 storage and network limitations present on mobile platforms.

In Figure 13 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.

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.

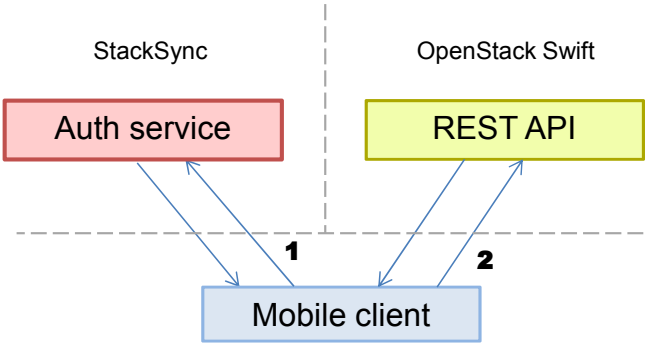


Figure 13: Mobile client interactions with auth service and storage API.

The application will make use of the Authentication and Storage APIs documented in the previous sections in order to interact with StackSync. In Figure 13 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.

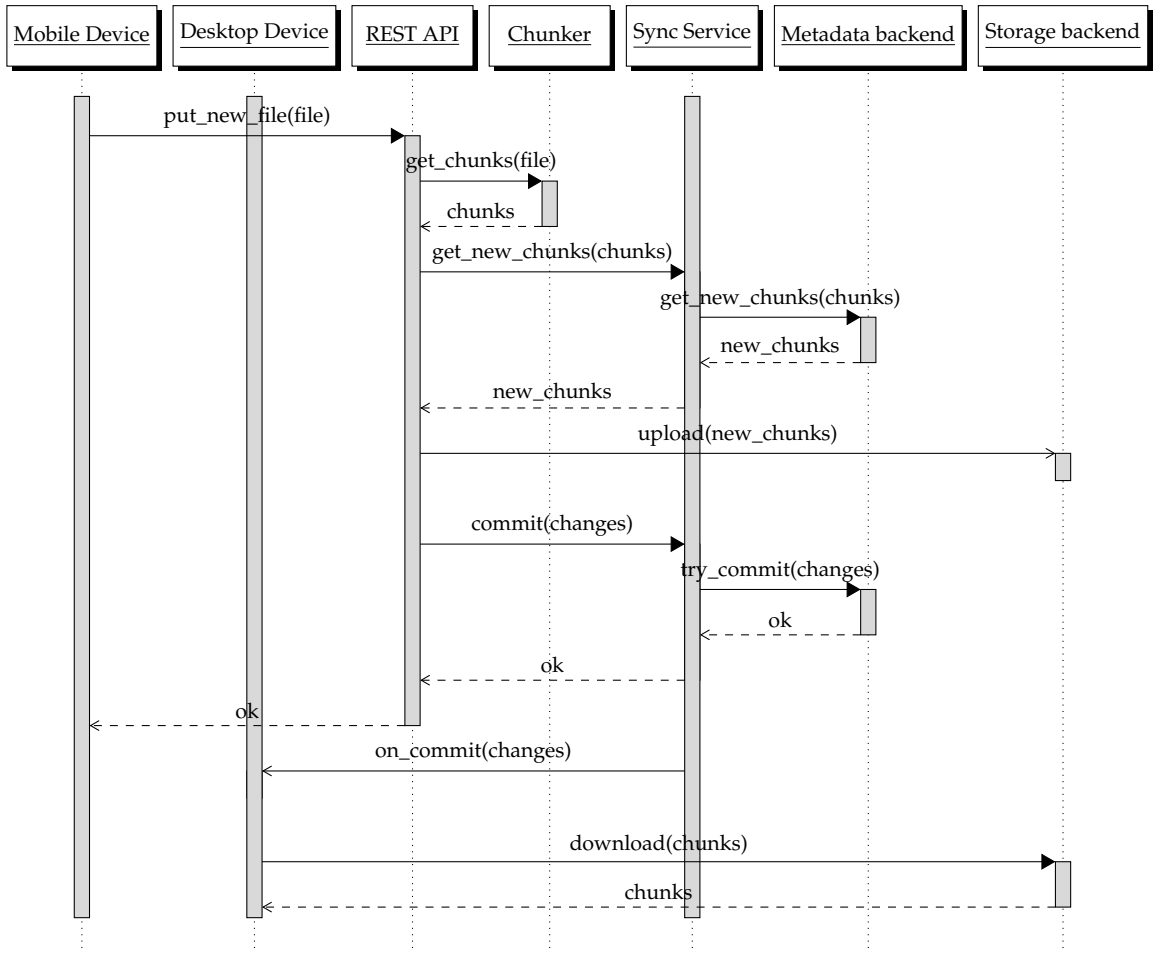


Figure 14: Mobile Client using the Storage Service Cloud.

In Figure 14 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.

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.

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 15 we show a class diagram of the tasks that will inherit from `AsyncTask`.

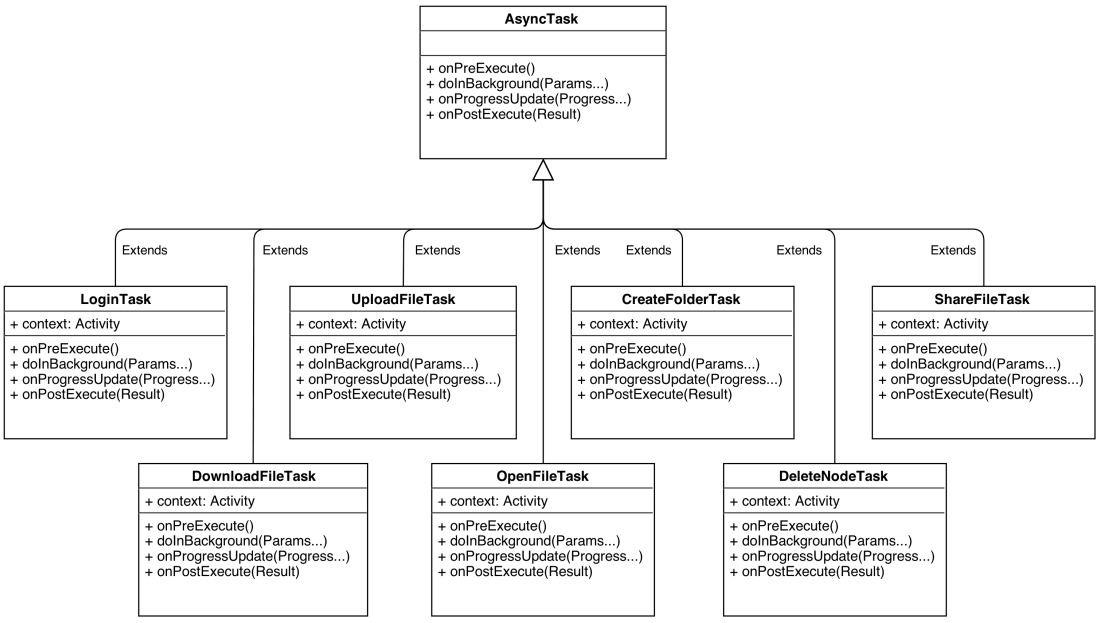


Figure 15: AsyncTask inheritance class diagram.

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 16 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.

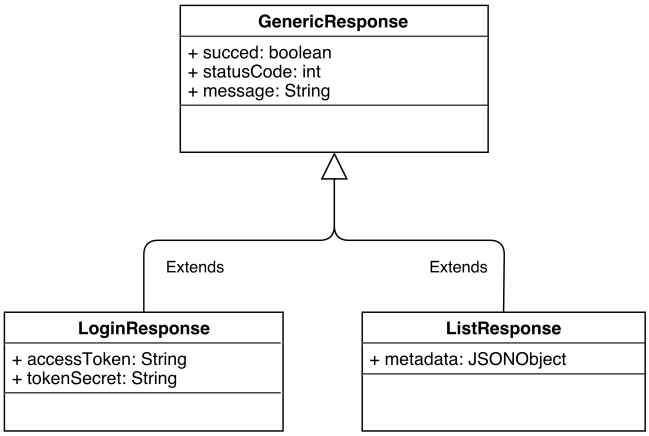


Figure 16: Response types class diagram.

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 user interface (UI).

As we observe on Figure 17, there will be three main UI. On the left we observe an `Activity` whose main focus will be to show the user a login form for him to provide his credentials.

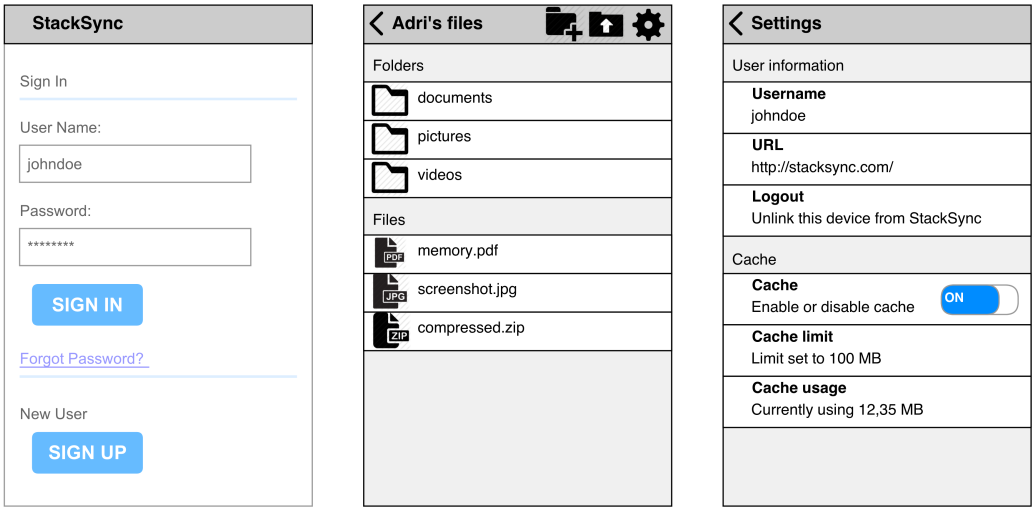


Figure 17: Design of user interfaces.

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 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.

All three activities will inherit from the `Activity` class as show in Figure 18.

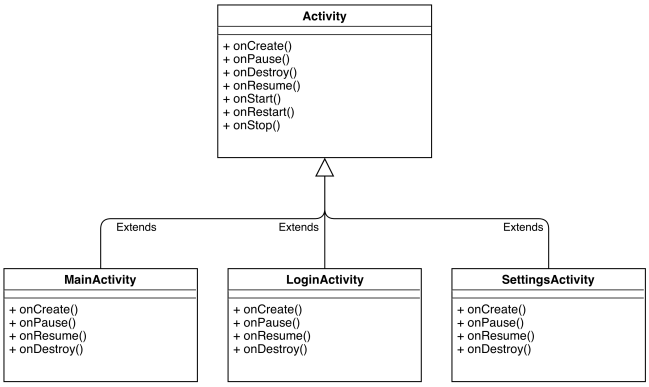


Figure 18: Activity inheritance class diagram.

4 StackSync Deployment essentials

In this section we explain how to deploy StackSync from scratch. First, we need a OpenStack Swift platform for the storage back-end. In this manual we assume that there is an OpenStack Swift installation ready to use. If this is not the case, please proceed to the official Swift documentation to deploy the latest available version.

Once the storage back-end is ready-to-use, it is necessary to decide how many servers we need to install StackSync. In this tutorial we will install all the services (SyncServer, QuotaServer, ObjectMQ and the Database) in the same server, but they can be installed in different servers independently.

4.1 SyncService

4.1.1 Requirements

RabbitMQ

As ObjectMQ is built on top of RabbitMQ, we need a RabbitMQ server. Since this service has to be accessed by desktop clients, it has to be deployed in a server with, at least, one public IP. To install it, execute the following command:

```
$ sudo apt-get install rabbitmq-server
```

Database initialization

The current version of StackSync uses PostgreSQL as the metadata database. It requires PostgreSQL version 9.1 or higher.

```
$ sudo apt-get install postgresql postgresql-contrib
```

In order to initialize the database we need to create the database and the user and execute the script "setup_db.sql" located in "src/main/resources".

Enter in Postgres command line mode with the superuser account:

```
$ sudo -u postgres psql
```

Execute the commands below to create a user and the database, and enable de UUID extension.

```
postgres=$ create database db_name;
postgres=$ create user db_user with password 'mysecretpwd';
postgres=$ grant all privileges on database db_name to db_user;
postgres=$ \connect db_name
postgres=$ CREATE EXTENSION "uuid-ossp";
postgres=$ \q
```

Enter to the database with the user role created. Note that the first parameter is the host, the second is the database name and the last one is the username:

```
$ psql -h localhost db\_name db\_user
```

Finally, download the setip_db.sql script from `https://github.com/stacksync/sync-service/blob/master/src/main/resources/setup_db.sql` and execute it to initialize the database:

```
postgres=$ \i PATH_TO_SCRIPT/setup_db.sql
postgres=$ \q
```

4.1.2 Installation

The SyncServer has to be installed in any Linux distribution based on Debian. To install it, we need to download the deb file form `https://github.com/stacksync/sync-service/releases` and execute the following command:

```
$ sudo dpkg -i stacksync-server_X.X.X_all.deb
```

The StackSync Server has a dependency with the JSVC library, if you experience any problem while installing run the following command:

```
$ sudo apt-get -f install
```

Once the server is installed, you must modify the configuration file located under the folder /etc/stacksync-server/ to connect with the database, the messaging middleware, and OpenStack Swift. You need to specify a Keystone user capable of creating users and set up ACL on containers on the specific container configured in the file.

The init script assumes that you have a "JAVA_HOME" environment variable set up, if not, it will execute the java located in "/usr/lib/jvm/default-java". You can change the Java VM by setting up the "JAVA_HOME" environment or by modifying the script in */etc/init.d/stacksync-server*.

Once configured, just run the server.

```
$ sudo service stacksync-server start
```

If something went wrong, you can check the standard and error log files located in:

/var/log/stacksync-server/

4.2 Quota Server

The Quota Server has two components: the daemon that will run in a server and the Swift module. First we will install the deb package and later we explain the installation of the Swift module.

Like the SyncServer, the StackSync Quota Server has to be installed in a Debian-based Linux distribution. The .deb file can be downloaded from `https://github.com/stacksync/stacksync-quota-server/releases`. To install it execute the following command:

```
$ sudo dpkg -i stacksync-quota-server_X.X.X_all.deb
```

After installing it, the configuration file located in */etc/stacksync-quota-server/stacksync-quota-server.conf* has to be modified to connect to the StackSync database.

Finally, start the server:

```
$ sudo service stacksync-quota-server start
```

Now we need to install the Swift module that will control the amount of data uploaded by users from StackSync. First, download the module from github:

```
$ git clone https://github.com/stacksync/swift-stacksync-quota.git
```

Install it with the following command:

```
$ cd swift-stacksync-quota
$ sudo python setup.py install
```

Modify the */etc/swift/proxy-server.conf* file to add the module in the Swift pipeline:

```
pipeline = catch_errors ... proxy-logging stacksync_quota_swift
proxy-server
```

And add the filter at the end of the file. Take into account that you have to set the IP of *stacksync_quota_host* to the host in which you have installed the StackSync Quota Server.

```
[filter:stacksync_quota_swift]
use = egg:stacksync_quota#stacksync_quota
stacksync_quota_host = 127.0.0.1
stacksync_quota_port = 62345
```

Finally, restart the proxy:

```
$ sudo swift-init proxy restart
```

4.3 Management Interface

Once we have the StackSync platform ready, we have to create users. For this reason we will need to install the management interface. This web manager allows StackSync administrators to create users and manage their quota.

The management interface is a Django project. It can be downloaded from `https://github.com/stacksync/manager`.

Before running the manager, we have to install some dependencies:

```
$ sudo apt-get install libpq-dev
$ sudo apt-get install python-dev
$ pip install -r requirements.txt
```

Once the requirements are installed, we have to create database tables for the manager. Before to sync the database, you have to change the settings.py file located `https://github.com/stacksync/manager/blob/master/stacksync_manager/settings.py`. After that, you just need to run the following command:

```
sudo python manage.py syncdb
```

This command will create necessary tables in the StackSync database. Furthermore, if it is the first time that you execute it, you will have to create an admin user. This user will be used to access the manager interface and create other admins/groups.

To run the manager interface you can configure the django project on top of Apache or you can execute directly the following command:

```
$ sudo python manage.py runserver 0.0.0.0:80
```

When the manager is running, you can access it from the browser with the URL `http://127.0.0.1:80/admin`.

4.4 Desktop clients

We provide installers for all platforms, Windows, MacOS X and Linux. The installers can be downloaded from `https://github.com/stacksync/desktop/releases`.

5 Installing StackSync Web and API

In the previous version we installed the basic components of StackSync to run desktop clients and synchronize files. In this section we will install the required components to use StackSync mobile and web clients. To achieve this, we will need to install two modules in the Swift proxy: StackSync Auth and StackSync storage API. The former is used to authenticate and authorize API clients, while the second will process the requests from the clients.

5.1 StackSync Auth

5.1.1 Requirements

Before installing the StackSync authentication middleware you first need to install some requirements. Although they are included in the setup.py, we show you how to install them one by one.

PostgreSQL client library

First, you have to install the PostgreSQL client library to communicate with the database backend.

```
$ sudo apt-get install libpq-dev python-dev
```

Now, you have to install the psycopg2 Python package.

```
$ sudo easy_install psycopg2
```

SQLAlchemy

SQLAlchemy is used to map Python objects to the database.

```
$ sudo easy_install sqlalchemy
```

Jinja2

Jinja2 is used to render the OAuth's authorization page so that the user can grant or deny permissions to applications.

```
$ sudo easy_install Jinja2
```

StackSync OAuth Python library

This Swift module uses the StackSync OAuth library, which is the OAuth 1.0 implementation for StackSync. To install it, clone the project from GitHub and execute the following command:

```
$ sudo python setup.py install
```

5.1.2 Installation

To install the StackSync authentication middleware for Swift you first need to install the Python package.

```
$ sudo python setup.py install
```

This will install the package after checking that the previous requirements are satisfied.

Now we need to modify the proxy configuration. First, we add the filter to tell the proxy that the middleware should be loaded. Be sure you set up the database with the correct parameters.

```
[filter:stacksync-auth-swift]
use = egg:stacksync-auth-swift#stacksync_auth_swift
psql_host = localhost
psql_port = 5432
psql_dbname = stacksync
psql_user = stacksync_user
psql_password = stacksync
tenant = stacksync
keystone_host = localhost
keystone_post = 5000
keystone_version = 2.0
templates_path = /home/swift/stacksync-auth-swift/templates/
```

Next, we have to add the middleware to the proxy's pipeline.

```
pipeline = catch_errors healthcheck proxy-logging cache bulk slo ratelimit
crossdomain authtoken keystoneauth stacksync-auth-swift staticweb
container-quotas account-quotas proxy-logging proxy-server
```

Finally, if you use Keystone, you have to delay the auth decision so that it does not interfere with the StackSync auth middleware.

```
[filter:authtoken]
paste.filter_factory = keystoneclient.middleware.auth_token:filter_factory
...
delay_auth_decision = true
```

Now you can restart the proxy.

```
$ sudo swift-init proxy restart
```

5.2 StackSync API

First, the project has to be downloaded from the StackSync Github repository. Once downloaded, to install the module execute the command:

```
$ sudo python setup.py install
```

When the proxy is installed, it is necessary to modify the proxy-server.conf file from Swift (/etc/swift/proxy-server.conf) to enable the StackSync API module:

```
[pipeline:main]
pipeline = healthcheck cache authtoken keystone stacksync-api proxy-server
```

Add the WSGI filter:

```
[filter:stacksync-api]
use = egg:stacksync-api-swift#stacksync_api
stacksync_host = 127.0.0.1
stacksync_port = 61234
```

Finally, restart the proxy:

```
$ swift-init proxy restart
```

5.3 Web Client

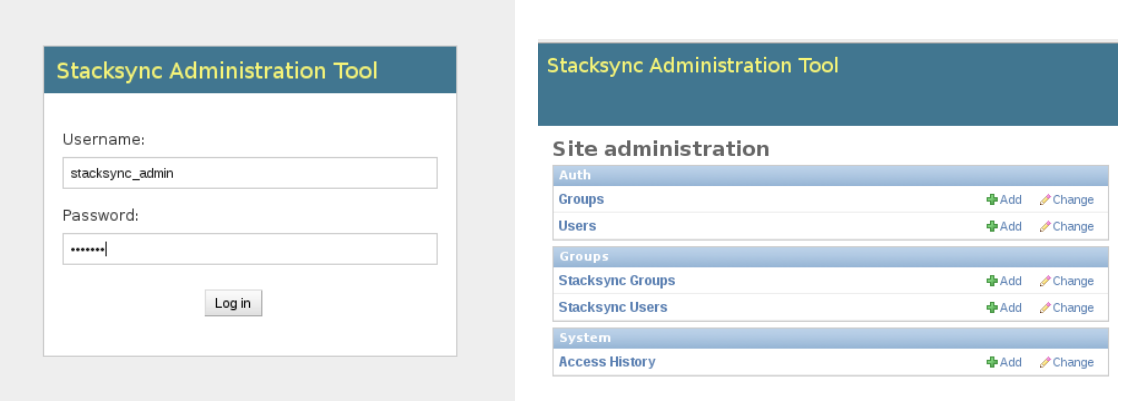
The web client will make use of the StackSync API and the StackSync Auth modules previously installed. This client will allow us to interact with our synced files from a browser. Like the management interface, the web client is a Django project. It can be downloaded from `https://github.com/stacksync/web`.

6 Management of the StackSync platform

6.1 Creating groups and administrators

In a StackSync deployment we can manage different groups. A group can be a company, institution or university. In this example, we will create the Universtitat Rovira i Virgili (urv) group.

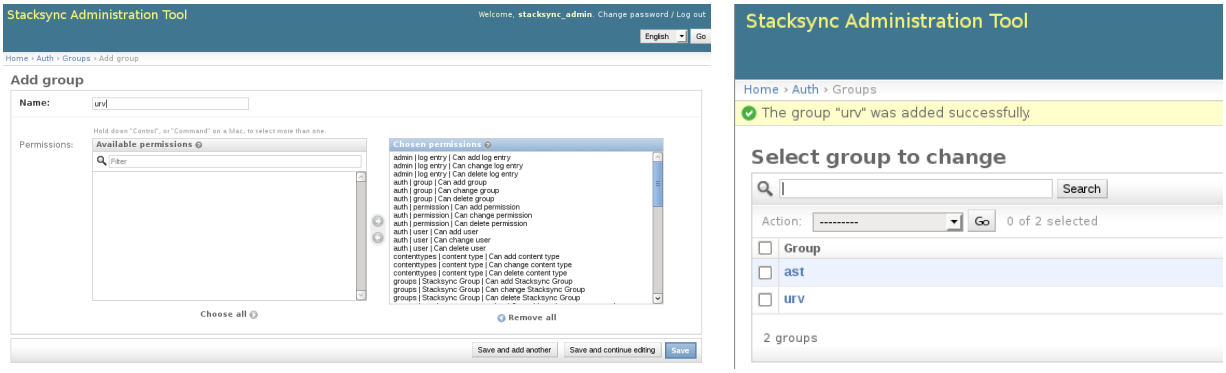
First of all the administrator of StackSync has to log in into the manager interface. Fig. 19 shows the login interface and the admin menu of the management web.



(a) Login interface. (b) First menu.

Figure 19

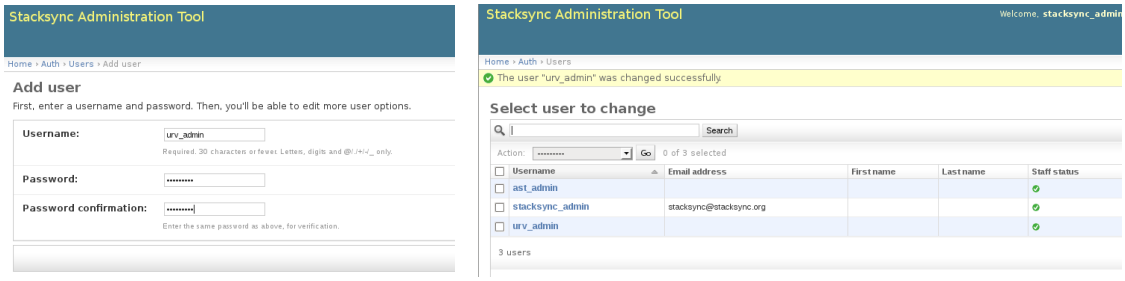
Once it is done, the admin has to create the URV group with the necessary permissions:



(a) Panel to add a new group and grant permissions. (b) When the group is created, it will be listed in the groups list.

Figure 20

Next step is to create an admin user for this new group. In our case, we will create the *urv_admin* user. This user will have the rights to create subgroups inside the *urv* group, i.e. departments, and create users and set their quota.



(a) Panel to create an admin user for a specific group.

(b) Admin user created.

Figure 21

It is also necessary to set the permissions to that admin user. It is important to select only the *urv* as the group to administrate. Otherwise, this user will have access to others groups in the platform.

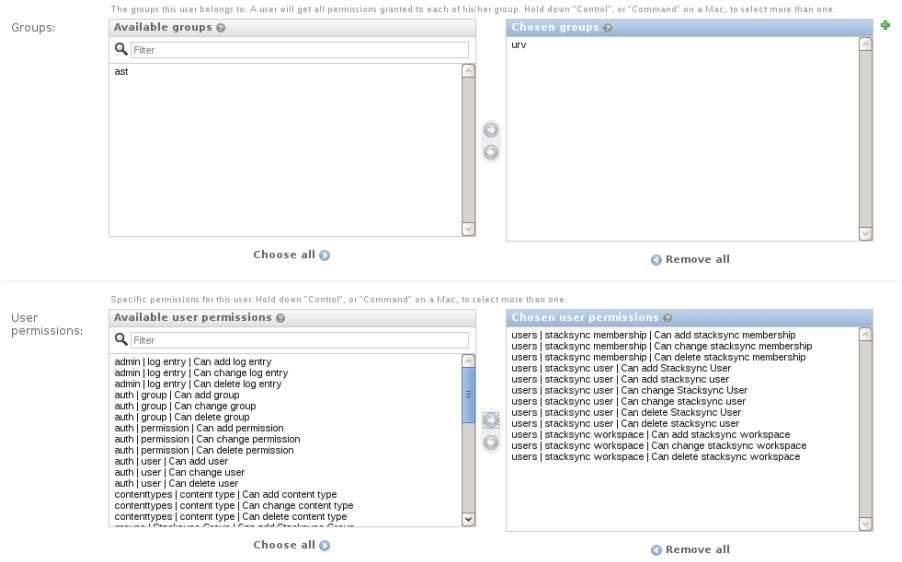
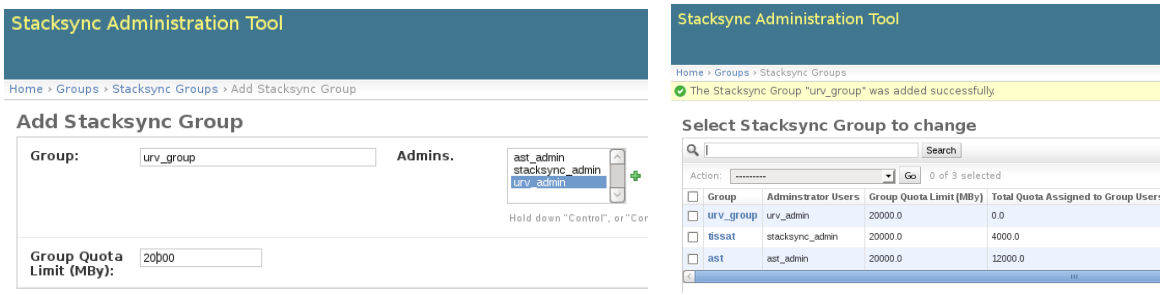


Figure 22: Grant permissions to the group.

6.2 Create subgroups and StackSync users

In the following images we will show how this admin user can create subgroups in the *urv* group. We can understand this groups as departments of a company.



(a) Panel to create StackSync subgroup. (b) Created groups.

Figure 23

As you can see in Fig. 23, the admin has to specify a total quota for that group. Next, we are going to create a user in this group.

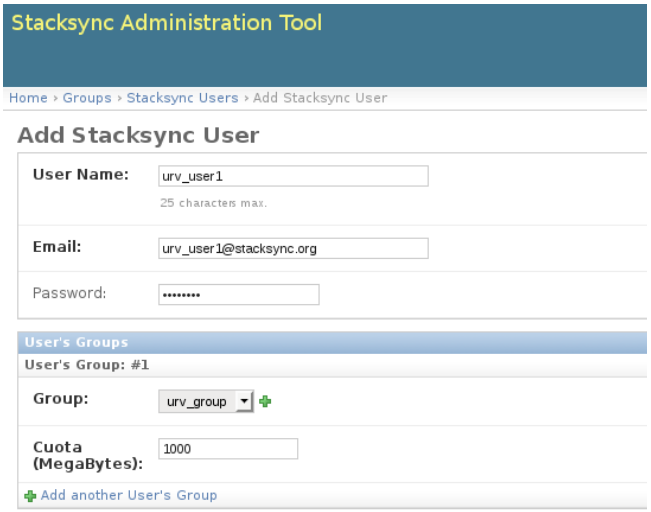


Figure 24: Panel to create a new user.

7 Analysis of U1 traces

7.1 Introduction

To optimize StackSync, we need to understand the nature of a real Personal Cloud workload in the wild. Unfortunately, due to the proprietary nature of these systems, very little is known about their performance and characteristics, including the workload they have to handle daily. And indeed, the few available studies have to rely on the so-called "black-box" approach, where traces are collected from a single or a limited number of measurement points, in order to infer their properties. This was the approach followed by the most complete analysis of a Personal Cloud to date, the measurement of Dropbox conducted by Drago et al. [9]. Although this work describes the overall service architecture, it provides no insights on the operation and infrastructure of the Dropbox's back-end. And also, it has the additional flaw that it only focuses on small and specific communities, like university campuses, which may breed false generalizations.

Similarly, several Personal Cloud services have been externally probed to infer their operational aspects, such as data reduction and management techniques [10, 11, 12], or even transfer performance [13, 14]. However, from external vantage points, it is impossible to fully understand the operation of these systems without fully reverse-engineering them.

We present results of our study of U1: the Personal Cloud of Canonical, integrated by default in Linux Ubuntu OS. Despite the shutdown of this service on July 2014, the distinguishing feature of our analysis is that it has been conducted using data collected by the provider itself. U1 provided service to several millions of users at the time of the study on January-February 2014, which constitutes the first complete analysis of the performance of a Personal Cloud in the wild. Such a unique data set has allowed us to reconfirm results from prior studies, like that of Drago et al. [9], which paves the way for a general characterization of these systems. But it has also permitted us to expand the knowledge base on these services, which now represent a considerable volume of the Internet traffic. According to Drago et al. [9], the total volume of Dropbox traffic accounted for a volume equivalent to around one third of the YouTube traffic on a campus network. Consequently, we believe that the results of our study can be useful for both researchers, ISPs and data center designers, giving hints on how to anticipate the impact of the growing adoption of these services. In summary, our contributions are the following:

Back-end architecture and operation of U1. This work provides a comprehensive description of the U1 architecture, being the first one to also describe the back-end infrastructure of a real-world vendor. Similarly to Dropbox [9], U1 decouples the storage of *file contents* (data) and *their logical representation* (metadata). Canonical only owns the infrastructure for the metadata service, whereas the actual file contents are stored separately in Amazon S3. Among other insights, we found that U1 API servers are characterized by long tail latencies and that a sharded database cluster is an effective way of storing metadata in these systems. Interestingly, these issues may arise in other systems that decouple data and metadata as U1 does [17].

Workload analysis and user behavior in U1. By tracing the U1 servers in the Canonical datacenter, we provide an extensive analysis of its *back-end activity* produced by the *entire user population* of U1 for one month (587K distinct users). Our analysis confirms already

Table 7.1a Summary of some of our most important findings and their implications.

UbuntuOne Analisys	**Finding**	**Implications and Opportunities**
Storage Workload (7.5)	90% of files are smaller than 1MByte (**P**).	Object storage services normally used as a cloud service are not optimized for managing small files [15].
	18.5% of the upload traffic is caused by file updates (**C**).	Changes in file metadata cause high overhead since the U1 client does not support delta updates (e.g. `.mp3` tags).
	We detected a deduplication ratio of 17% in one month (**C**).	File-based cross-user deduplication provides an attractive trade-off between complexity and performance [12].
	DDoS attacks against U1 are frequent (**N**).	Further research is needed regarding secure protocols and automatic countermeasures for Personal Clouds.
User Behavior (7.6)	1% of users generate 65% of the traffic (**P**).	Very active users may be treated in an optimized manner to reduce storage costs.
	Data management operations (e.g., uploads, file deletions) are normally executed in long sequences (**C**).	This correlated behavior can be exploited by caching and prefetching mechanisms in the server-side.
	User operations are *bursty*; users transition between long, idle periods and short, very active ones (**N**).	User behavior combined with the user per-shard data model impacts the metadata back-end load balancing.
Back-end Performance (7.8)	A 20-node database cluster provided service to 587K users without symptoms of congestion (**N**).	The *user-centric data model* of a Personal Cloud makes relational database clusters a simple yet effective approach to scale out metadata storage.
	RPCs service time distributions accessing the metadata store exhibit long tails (**N**).	Several factors at hardware, OS and application-level are responsible for poor tail latency in RPC servers [16].
	In short time windows, load values of API servers/DB shards are very far from the mean value (**N**).	Further research is needed to achieve better load balancing under this type of workload.

C: Confirms previous results, **P**: Partially aligned with previous observations, **N**: New observation

reported facts, like the execution of user operations in long sequences [9] and the potential waste that file updates may induce in the system [11, 12]. Moreover, we provide new observations, such as a taxonomy of files in the system, the modeling of burstiness in user operations or the detection of attacks to U1, among others. Table 7.1a summarizes some of our key findings.

Potential improvements to Personal Clouds. We suggest that a Personal Cloud should be aware of the *behavior of users* to optimize its operation. Given that, we discuss the implications of our findings to the operation of U1. For instance, despite U1 was frequently used for editing files, file updates were responsible for 18.5% of upload traffic mainly due to the lack of delta updates in the desktop client. Furthermore, we detected 3 DDoS attacks in one month, motivating the need for further research in automatic attack countermeasures in secure and dependable storage protocols. Although our observations may not apply to all existing services, we believe that our analysis can help to improve the next generation of Personal Clouds [17, 11].

Publicly available dataset. We contribute our dataset (773GB) to the community and it is available at `http://cloudspaces.eu/datasets/u1_measurement`. To our knowledge, this is the first dataset that contains the back-end activity of a large-scale Personal Cloud. We hope that our dataset provides new opportunities to researchers in further understanding the internal operation of Personal Clouds, promoting research and experimentation in this field.

Table 7.1b Description of the most relevant U1 API operations.

API Operation	*Related RPC*	*Description*
`ListVolumes`	`dal.list_volumes`	This operation is normally performed at the beginning of a session and lists all the volumes of a user (root, user-defined, shared).
`ListShares`	`dal.list_shares`	This operation lists all the volumes of a user that are of type *shared*. In this operation, ther field `shared by` is the owner of the volume and `shared to` is the user to which that volume was shared with. In this operation, the field `shares` represents the number of volumes type *shared* of this user.
`(Put/Get)Content`	see A	These operations are the actual file uploads and downloads, respectively. The notification goes to the U1 back-end but the actual data is stored in a separate service (Amazon S3). A special process is created to forward the data to Amazon S3. Since the upload management in U1 is complex, we refer the reader to A for a description in depth of upload transfers.
`Make`	`dal.make_dir` `dal.make_file`	This operation is equivalent to a "touch" operation in the U1 back-end. Basically, it creates a file node entry in the metadata store and normally precedes a file upload.
`Unlink`	`dal.unlink_node`	Delete a file or a directory from a volume.
`Move`	`dal.move`	Moves a file from one directory to another.
`CreateUDF`	`dal.create_udf`	Creates a user-defined volume.
`DeleteVolume`	`dal.delete_volume`	Deletes a volume and the contained nodes.
`GetDelta`	`dal.get_delta`	Get the differences between the server volume and the local one (generations).
`Authenticate`	`auth.get_user_id_from_token`	Operations managed by the servers to create sessions for users.

7.2 Background

A Personal Cloud can be loosely defined as a unified digital locker for users' personal data, offering at least three key services: *file storage*, *synchronization* and *sharing* [18]. Numerous services such as Dropbox, U1 and Box fall under this definition.

From an architectural viewpoint, a Personal Cloud exhibits a 3-tier architecture consisting of: (i) *clients*, (ii) *synchronization* or *metadata service* and (iii) *data store* [9, 17]. Thus, these systems explicitly decouple the management of file contents (data) and their logical representation (metadata). Companies like Dropbox and Canonical only own the infrastructure for the metadata service, which processes requests that affect the virtual organization of files in user volumes. The contents of file transfers are stored separately in Amazon S3. An advantage of this model is that the Personal Cloud can easily scale out storage capacity thanks to the "pay-as-you-go" cloud payment model, avoiding costly investments in storage resources.

In general, Personal Clouds provide clients with 3 main types of access to their service: Web/mobile access, Representational State Transfer (REST) APIs [14, 19] and *desktop clients*. Our measurements focus on the desktop client interactions with U1. Personal Cloud desktop clients are very popular among users since they provide automatic synchronization of user files across several devices (see Section 7.3.3). To achieve this, desktop clients and the server-side infrastructure communicate via a *storage protocol*. In most popular Personal Cloud services (e.g., Dropbox), such protocols are proprietary.

U1 Personal Cloud was a suite of online services offered by Canonical that enabled users to store and sync files online and between computers, as well as to share files/folders with others using file synchronization. Until the service was discontinued in July 2014, U1 provided desktop and mobile clients and a Web front-end. U1 was integrated with other Ubuntu services, like Tomboy for notes and U1 Music Store for music streaming.

7.3 The U1 Personal Cloud

In this section, we first describe the U1 storage protocol used for communication between clients and the server-side infrastructure (Sec. 7.3.1). This will facilitate the understanding of the system architecture (Sec. 7.3.2). We then discuss the details of a U1 desktop client (Sec. 7.3.3). Finally, we give details behind the core component of U1, its metadata back-end (Sec. 7.3.4).

7.3.1 U1 Storage Protocol

U1 uses its own protocol (`ubuntuone-storageprotocol`) based on TCP and Google Protocol Buffers[6]. In contrast to most commercial solutions, the protocol specifications and client-side implementation are publicly available[7]. Here, we describe the protocol in the context of its *entities* and *operations*. Operations can be seen as end-user actions intended to manage one/many entities, such as a file or a directory.

Protocol Entities

In the following, we define the main entities in the protocol. Note that in our analysis, we characterize and identify the role of these entities in the operation of U1.

Node: Files and directories are *nodes* in U1. For files, U1 decouples their logical representation from their actual contents. Drawing a comparison to a file system, the inodes are stored in the metadata service and the extents are stored in Amazon S3. The protocol supports CRUD operations on nodes (e.g. list, delete, etc.). The protocol assigns Universal Unique Identifiers (UUIDs) to both node objects and their contents, which are generated in the back-end.

Volume: A volume is a container of node objects. During the installation of the U1 client, the client creates an initial volume to store files with id=0 (root). There are 3 types of volumes: i) *root/predefined*, ii) *user defined folder* (UDF), which is a volume created by the user, and iii) *shared* (sub-volume of another user to which the current user has access).

Session: A user interacts with the server in the context of a U1 storage protocol session (not HTTP or any other session type). This session is used to identify the requests of a single user during the session lifetime. Usually, sessions do not expire automatically. A client may disconnect, or a server process may go down, and that will end the session. For this reason, in parallel with a session, a user establishes a TCP connection with U1 that is used to detect these events. To create a new session, an OAuth [20] token is used to authenticate clients against U1. Tokens are stored separately in the Canonical authentication service (see 7.3.4).

API Operations

The U1 storage protocol offers an API consisting of the *data management* and *metadata operations* that can be executed by a client. Metadata operations are those operations that do not involve transfers to/from the data store (i.e., Amazon S3), such as listing or deleting

[6] `https://wiki.ubuntu.com/UbuntuOne`
[7] `https://launchpad.net/ubuntuone-storage-protocol`

files, and are entirely managed by the synchronization service. On the contrary, uploads and downloads are, for instance, typical examples of data management operations.

In Table 7.1b we describe the most important protocol operations between users and the server-side infrastructure. We traced these operations to quantify the system's workload and the behavior of users.

7.3.2 Architecture Overview

As mentioned before, U1 has a 3-tier architecture consisting of *clients*, *synchronization service* and the *data/metadata store*. Similarly to Dropbox [9], U1 *decouples* the storage of *file contents* (data) and *their logical representation* (metadata). Canonical only owns the infrastructure for the metadata service, which processes requests that affect the virtual organization of files in user volumes. The actual contents of file transfers are stored separately in Amazon S3.

However, U1 treats *client requests* differently from Dropbox. Namely, Dropbox enables clients to send requests either to the metadata or storage service depending on the request type. Therefore, the Dropbox infrastructure only processes metadata/control operations. The cloud storage service manages data transfers, which are normally orchestrated by computing instances (e.g. EC2).

In contrast, U1 receives both metadata requests and data transfers of clients. Internally, the U1 service discriminates client requests and redirects them either to the metadata store or the storage service, respectively. For each upload and download request, a new process is instantiated to manage the transfer between the client and S3 (see A). Therefore, the U1 model is simpler from a design perspective, yet this comes at the cost of delegating the responsibility of processing data transfers to the metadata back-end.

U1 Operation Workflow. Imagine a user that initiates the U1 desktop client (7.3.3). At this point, the client sends an `Authenticate` API call (see Table 7.1b) to U1, in order to establish a new session. An API server receives the request and contacts to the Canonical authentication service to verify the validity of that client (7.3.4). Once the client has been authenticated, a persistent TCP connection is established between the client and U1. Then, the client may send other management requests on user files and directories.

To understand the synchronization workflow, let us assume that two clients are online and work on a shared folder. Then, a client sends an `Unlink` API call to delete a file from the shared folder. Again, an API server receives this request, which is forwarded in form of RPC call to a RPC server (7.3.4). As we will see, RPC servers translate RPC calls into database query statements to access the correct metadata store shard (PostgreSQL cluster). Thus, the RPC server deletes the entry for that file from the metadata store.

When the query finishes, the result is sent back from the RPC server to the API server that responds to the client that performed the request. Moreover, the API server that handled the `Unlink` notifies the other API servers about this event that, in turn, is detected by the API server to which the second user is connected. This API server notifies via push to the second client, which deletes that file locally.

Next, we describe in depth the different elements involved in this example of operation: The desktop client, the U1 back-end infrastructure and other key back-end services to the

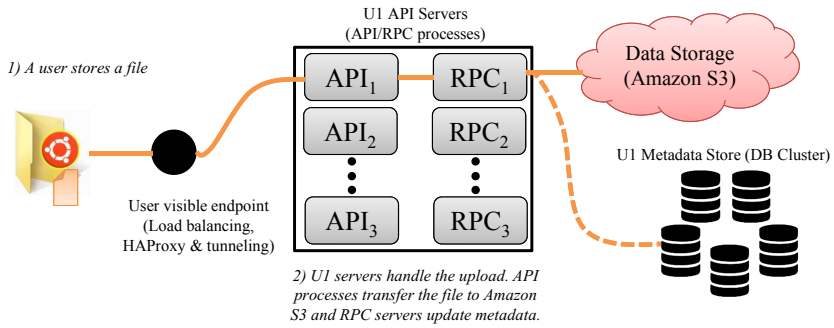


Figure 25: Architecture of U1 back-end.

operation of U1 (authentication and notifications).

7.3.3 U1 Desktop Client

U1 provides a user friendly desktop client, implemented in Python (GPLv3), with a graphical interface that enables users to manage files. It runs a daemon in the background that exposes a message bus (DBus) interface to handle events in U1 folders and make server notifications visible to the user through OS desktop. This daemon also does the work of deciding what to synchronize and in which direction to do so.

By default, one folder labeled `~/Ubuntu One/` is automatically created and configured for mirroring (root `volume`) during the client installation. Changes to this folder (and any others added) are watched using `inotify`. Synchronization metadata about directories being mirrored is stored in `~/.cache/ubuntuone`. When remote content changes, the client acts on the incoming unsolicited notification (push) sent by U1 service and starts the download. Push notifications are possible since clients establish a TCP connection with the metadata service that remains open while online.

In terms of data management, Dropbox desktop clients deduplicate data at chunk level [9]. In contrast, U1 resorts to file-based cross-user deduplication to reduce the waste of storing repeated files [12]. Thus, to detect duplicated files, U1 desktop clients provide to the server the SHA-1 hash of a file prior to the content upload. Subsequently, the system checks if the file to be uploaded already exists or not. In the affirmative case, the new file is *logically linked* to the existing content, and the client does not need to transfer data.

Finally, as observed in [12], the U1 client applies compression to uploaded files to optimize transfers. However, it does not perform advanced techniques, such as file bundling[8], delta updates and sync deferment, to buffer frequent changes to the same file, leading to potential inefficiencies.

[8]Li et al. [12] suggest that U1 may group small files together for upload (i.e. bundling), since they observed high efficiency uploading sets of small files. However, U1 does not bundle small files together. Instead, clients establish a TCP connection with the server that remains open during the session, avoiding the overhead of creating new connections.

7.3.4 U1 Metadata Back-end

The entire U1 back-end is all inside a single datacenter and its objective is to manage the metadata service. The back-end architecture appears in Fig. 25 and consists of *metadata servers* (API/RPC), *metadata store* and *data store*.

System gateway. The gateway to the back-end servers is the load balancer. The load balancer (HAProxy, ssl, etc.) is the visible endpoint for users and it is composed of two racked servers.

Metadata store. U1 stores metadata in a PostgreSQL database cluster composed of 20 large Dell racked servers, configured in 10 shards (master-slave). Internally, the system routes operations *by user identifier* to the appropriate shard. Thus, metadata of a user's files and folders reside always in the same shard. This data model *effectively* exploits sharding, since normally there is no need to lock more than one shard per operation (i.e. lockless). Only operations related to shared files/folders may require to involve more than one shard in the cluster.

API/RPC servers. Beyond the load balancer we find the API and RPC database processes that run on 6 separate racked servers. API servers receive commands from the user, perform authentication, and translate the commands into RPC calls. In turn, RPC database workers translate these RPC calls into database queries and route queries to the appropriate database shards. API/RPC processes are more numerous than physical machines (normally $8 - 16$ processes per physical machine), so that they can migrate among machines for load balancing. Internally, API and RPC servers, the load balancer and the metadata store are connected though a switched 1Gbit Ethernet network.

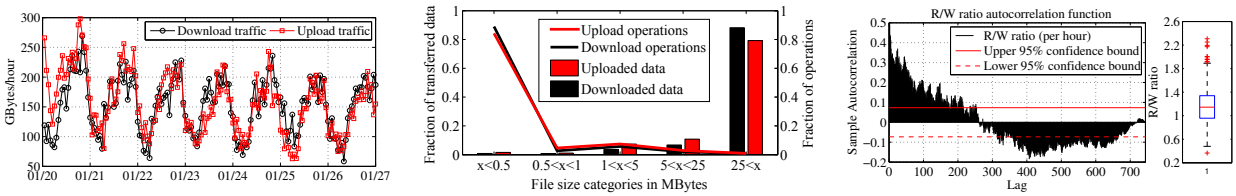
Data storage. Like other popular Personal Clouds, such as Dropbox or SugarSync, U1 stores user files in a separate cloud service. Concretely, U1 resorts to Amazon S3 (`us-east`) to store user data. This solution enables a service to rapidly scale out without a heavy investment in storage hardware. In its latests months of operation, U1 had a $\approx 20,000$\$ monthly bill in storage resources, being the most important Amazon S3 client in Europe.

With this infrastructure, U1 scaled up to 4 million registered users (587 thousand were traced in this measurement).

Authentication Service

The authentication service of U1 is shared with other Canonical services within the same datacenter and it is based on OAuth [20]. The first time a user interacts with U1, the desktop client requires him to introduce his credentials (email, password). The API server that handles the authentication request contacts the authentication service to generate a new token for this client. The created token is associated in the authentication service with a new user identifier. The desktop client also stores this token locally in order to avoid exposing user credentials in the future.

In the subsequent connections of that user, the authentication procedure is easier. Basically, the desktop client sends a connection request with the token to be authenticated. The U1 API server responsible for that requests asks the authentication service if the token does



(a) Transferred traffic time-series. (b) Traffic vs file size category. (c) R/W ratio analysis.

Figure 26: Macroscopic storage workload metrics of U1.

exist and has not expired. In the affirmative case, the authentication service retrieves the associated user identifier, and a new session is established. During the session, the token of that client is cached to avoid overloading the authentication service.

The authentication infrastructure consists of 1 database server with hot failover and 2 application servers configured with crossed stacks of Apache/Squid/HAProxy.

Notifications

Clients detect changes in their volumes by comparing their local state with the server side on every connection (generation point). However, if two related clients are online and their changes affect each other (e.g. updates to shares, new shares), API servers notify them directly (push). To this end, API servers resort to the TCP connection that clients establish with U1 in every session.

Internally, the system needs a way of notifying changes to API servers that are relevant to simultaneously connected clients. Concretely, U1 resorts to RabbitMQ (1 server) for communicating events between API servers[9], which are subscribed in the queue system to send and receive new events to be communicated to clients.

Next, we describe our measurement methodology to create the dataset used in our analysis.

7.4 Data Collection

We present a study of the U1 service back-end. In contrast to other Personal Cloud measurements [9, 14, 12], we did not deploy vantage points to analyze the service externally. Instead, we inspected directly the U1 metadata servers to measure the system. This has been done in collaboration with Canonical in the context of the FP7 CloudSpaces[10] project. Canonical anonimyzed sensitive information to build the trace, following strict ethical guidelines.

The traces are taken at both *API* and *RPC* server stages. In the former stage we collected important information about the storage workload and user behavior, whereas the second stage provided us with valuable information about the requests' life-cycle and the metadata store performance.

We built the trace capturing a series of service *logfiles*. Each logfile corresponds to the

[9] If connected clients are handled by the same API process, their notifications are sent immediately, i.e. there is no need for inter-process communication with RabbitMQ.

[10] `http://cloudspaces.eu`

Table 7.4a Summary of the trace.

Trace duration	30 days (01/11 - 02/10)
Trace size	773 GB (3,391M lines)
Back-end servers traced	6 servers (all)
Unique user IDs	587,602
Unique files	137.63M
User sessions	42.5M
Transfer operations	194.3M
Total upload traffic	105TB
Total download traffic	120TB

entire activity of a single API/RPC process in a machine for a period of time. Each logfile is within itself strictly *sequential and timestamped*. Thus, causal ordering is ensured for operations done for the same user. However, the timestamp between servers is not dependable, even though machines are synchronized with NTP (clock drift may be in the order of ms).

To gain better understanding on this, consider a line in the trace with this logname: `production-whitecurrant-23-20140128`. They will all be `production`, because we only looked at production servers. After that prefix is the name of the physical machine, followed by the number of the server process. The mapping between services and servers is dynamic within the time frame of analyzed logs, since they can migrate between servers to balance load. In any case, the identifier of the process is unique within a machine. After that is the date the logfile was "cut" (there is one log file per server/service and day).

Database sharding is in the metadata store back-end, so it is behind the point where traces were taken. This means that in these traces any combination of server/process can handle any user. To have a strictly sequential notion of the activity of a user we should take into account the U1 *session* and sort the trace by timestamp (a user may have more than one parallel connection). A session starts in the least loaded machine and lives in the same node until it finishes, making user events strictly sequential. Thanks to this information we can estimate system and user service times.

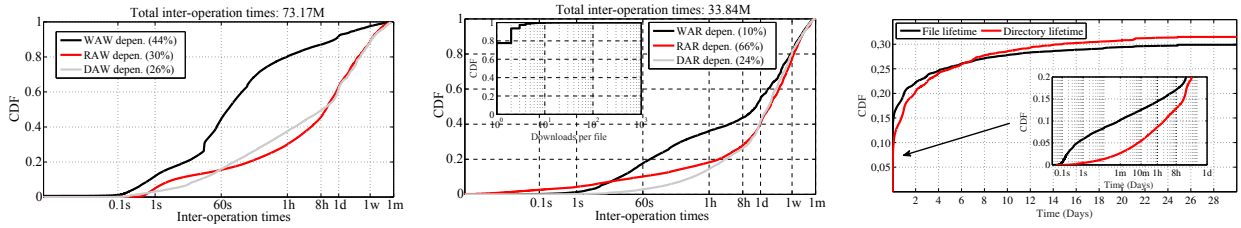
Approximately 1% of traces are not analyzed due to failures parsing of the logs.

7.4.1 Dataset

The trace is the result of merging all the logfiles (773GB of `.csv` text) of the U1 servers for 30 days (see Table 7.4a).

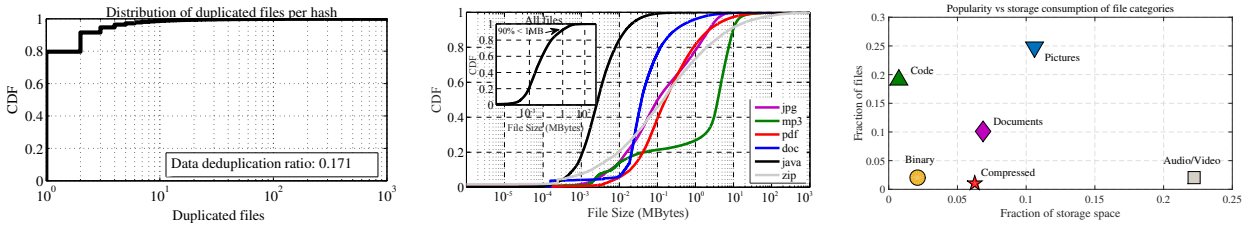
The trace contains the API operations (request type `storage/storage_done`) and their translation into RPC calls (request type `rpc`), as well as the session management of users (request type `session`). This provides different sources of valuable information. For instance, we can analyze the *storage workload* supported by a real-world cloud service (users, files, operations). Since we captured file properties such as file size and hash, we can study the storage system in high detail (contents are not disclosed).

Dataset limitations. We mentioned that timestamps among servers are not dependable since they may be different (in order of ms). Also, the dataset only includes events originating from desktop clients. Other sources, namely the web front-end and the mobile clients, are not included. This is because the different client types are handled by different software stacks that were not logged. Finally, we detected that sharing among users is limited.



(a) X-after-Write inter-arrival times. (b) X-after-Read inter-arrival times. (c) File/directory lifetime.

Figure 27: Usage and behavior of files in U1.



(a) Duplicated files per hash. (b) Size of files per extension. (c) Number/storage share of file types.

Figure 28: Characterization of files in U1.

7.5 Storage Workload

First, we quantify the storage workload supported by U1 for one month. Moreover, we pay special attention to the behavior of files in the system, to infer potential improvements.

7.5.1 Macroscopic Daily Usage

Storage traffic and operations. Fig. 26(a) provides a time-series view of the upload/download traffic of U1. We observe in Fig. 26(a) that U1 exhibits important *daily patterns*. To wit, the volume of uploaded GBytes per hour can be up to 10x higher in the central day hours compared to the nights. This observation is aligned with previous works, that detected time-based variability in both the usage and performance of Personal Cloud services [9, 14]. This effect is probably related to the working habits of users, since U1 desktop clients are by default initiated automatically when users turn on their machines.

Another aspect to explore is the relationship between file size and its impact in terms of upload/download traffic. To do so, in Fig. 26(b), we depict in relative terms the fraction of transferred data and storage operations for distinct file sizes. As can be observed, a very small amount of large files ($>$ 25MBytes) capitalizes 79.3% and 88.2% of upload and download traffic, respectively. Conversely, 84.3% and 89.0% of upload and download operations are related to small files ($<$ 0.5MBytes). As reported in other domains [21, 22, 23], we conclude that in U1 the workload in terms of *storage operations* is dominated by small files, whereas a small number of large files generate most of the *network traffic*.

For uploads, we found that 10.05% of total upload operations are *updates*, that is, an upload of an existing file that has distinct hash/size. However, in terms of traffic, file up-

dates represent 18.47% of the U1 upload traffic. This can be partly explained by the lack of delta updates in the U1 client and the heavy file-editing usage that many users exhibited (e.g., code developers). Particularly for media files, U1 engineers found that applications that modify the metadata of files (e.g., tagging `.mp3` songs) induced high upload traffic since the U1 client uploads again files upon metadata changes, as they are interpreted as regular updates.

To summarize, Personal Clouds tend to exhibit daily traffic patterns, and most of this traffic is caused by a small number of large files. Moreover, desktop clients should efficiently handle file updates to minimize traffic overhead.

R/W ratio. The read/write (R/W) ratio represents the relationship between the downloaded and uploaded data in the system for a certain period of time. Here we examine the variability of the R/W ratio in U1 (1-hour bins). The boxplot in Fig. 26(c) shows that the R/W ratio *variability can be important*, exhibiting differences of 8x within the same day. Moreover, the median (1.14) and mean (1.17) values of the R/W ratio distribution point out that the U1 workload is *slightly read-dominated*, but not as much as it has been observed in Dropbox [9]. This indicates that users mainly use U1 as a storage service, rather than for sharing content.

We also want to explore if the R/W ratios present patterns or dependencies along time due to the working habits of users. To verify whether R/W ratios are independent along time, we calculated the autocorrelation function (ACF) for each 1-hour sample (see Fig. 26(c)). To interpret Fig. 26(c), if R/W ratios 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 $\pm 2/\sqrt{N}$.

As shown in Fig. 26(c), R/W ratios are not independent, since most lags are outside 95% confidence intervals, which indicates long-term correlation with alternating positive and negative ACF trends. This evidences that the R/W ratios of U1 workload are not random and follow a pattern also guided by the working habits of users.

Concretely, averaging R/W ratios for the same hour along the whole trace, we found that from 6am to 3pm the R/W ratio shows a linear decay. This means that users download more content when they start the U1 client, whereas uploads are more frequent during the common working hours. For evenings and nights we found no clear R/W ratio trends.

We conclude that different Personal Clouds may exhibit disparate R/W ratios, mainly depending on the purpose and strengths of the service (e.g., sharing, content distribution). Moreover, R/W ratios exhibit patterns along time, which can be predicted in the server-side to optimize the service.

7.5.2 File-based Workload Analysis

File operation dependencies. Essentially, in U1 a file can be *downloaded (or read)* and *uploaded (or written)* multiple times, until it is eventually *deleted*. Next, we aim at inspecting the dependencies among file operations [24, 25], which can be *RAW* (Read-after-Write), *WAW* (Write-after-Write) or *DAW* (Delete-after-Write). Analogously, we have *WAR*, *RAR* and *DAR* for operations executed after a read.

First, we inspect file operations that occur after a write (Fig. 27(a)). We see that *WAW* dependencies are the most common ones (30.1% of 170.01M in total). This can be due to the fact that users *regularly update synchronized files*, such as documents of code files. This result is consistent with the results in [24] for personal workstations where block updates are common, but differs from other organizational storage systems in which files are almost immutable [25]. Furthermore, the 80% of WAW times are shorter than 1 hour, which seems reasonable since users may update a single text-like file various times within a short time lapse.

In this sense, Fig. 27(a) shows that *RAW* dependencies are also relevant. Two events can lead to this situation: (i) the system synchronizes a file to another device right after its creation, and (ii) downloads that occur after every file update. For the latter case, reads after successive writes can be optimized with sync deferment to reduce network overhead caused by synchronizing intermediate versions to multiple devices [12]. This has not been implemented in U1.

Second, we inspect the behavior of X-after-Read dependencies (Fig. 27(b)). As a consequence of active update patterns (i.e., write-to-write) and the absence of sync deferment, we see in Fig. 27(b) that *WAR* transitions also occur within reduced time frames compared to other transitions. Anyway, this dependency is the least popular one yielding that files that are *read tend not to be updated again*.

In Fig. 27(b), 40% of RAR times fall within 1 day. RAR times are shorter than the ones reported in [25], which can motivate the introduction of *caching mechanisms* in the U1 backend. Caching seems specially interesting observing the inner plot of Fig. 27(b) that reveals a long tail in the distributions of reads per file. This means that a small fraction of files is very popular and may be effectively cached.

By inspecting the Delete-after-X dependencies, we detected that around 12.5M files in U1 were *completely unused* for more than 1 day before their deletion (9.1% of all files). This simple observation on dying files evidences that *warm and/or cold data exists* in a Personal Cloud, which may motivate the involvement of warm/cold data systems in these services (e.g., Amazon Glacier, f4 [26]). To efficiently managing warm files in these services is object of current work.

Node lifetime. Now we focus on the lifetime of user files and directories (i.e., nodes). As shown in Fig. 27(c), 28.9% of the new files and and 31.5% of the recently created directories are deleted within one month. We also note that the lifetime distributions of *files and directories are very similar*, which can be explained by the fact that deleting a directory in U1 triggers the deletion of all the files it contains.

This figure also unveils that a large fraction of nodes are deleted within *few hours after their creation*, especially for files. Concretely, users delete 17.1% of files and 12.9% of directories within 8 hours after their creation time.

All in all, in U1 files exhibit similar lifetimes than files in local file systems. For instance, Agrawal et al. in [22] analyzed the lifetimes of files in corporative desktop computers for five years. They reported that around 20% to 30% of files (depending on the year) in desktop computers present a lifetime of one month, which agrees with our observations. This suggests that *users behave similarly deleting files* either in synchronized or local folders.

7.5.3 File Deduplication, Sizes and Types

File-based deduplication. The deduplication ratio (*dr*) is a metric to quantify the proportion of duplicated data. It takes real values in the interval $[0, 1)$, with 0 signaling no file deduplication at all, and 1 meaning full deduplication. It is expressed as $dr = 1 - (D_{unique}/D_{total})$, where D_{unique} is the amount of unique data, and D_{total} is equal to the total storage consumption.

We detected a *dr* of 0.171, meaning that the 17% of files in the trace can be deduplicated. This is slightly better than the deduplication ratio reported by Canonical ($\approx$ 11%), and similar (18%) to that given by the recent work of Li et al. [12]. This suggests that file-based cross-user deduplication could be a practical approach to reduce storage costs in U1.

Moreover, Fig. 28(a) demonstrates that the distribution of file objects w.r.t unique contents exhibits a long tail. This means that a small number of files accounts for a very large number of duplicates (e.g., popular songs), whereas 80% files present no duplicates. Hence, files with many duplicates represent a *hot spot* for the deduplication system, since a large number of logical links point to a single content.

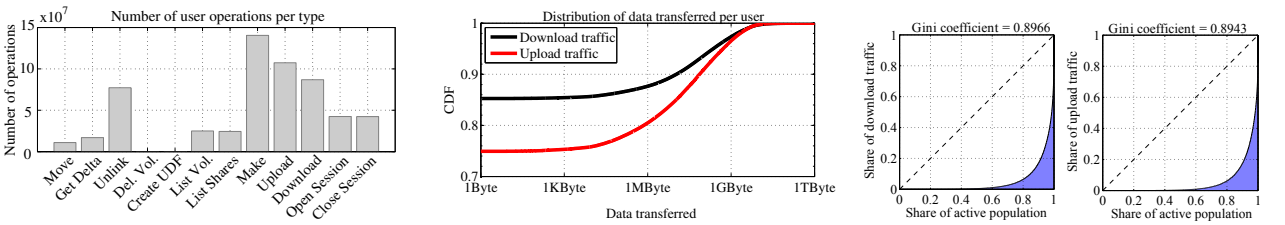
File size distribution. The inner plot of Fig. 28(b) illustrates the file size distribution of transferred files in the system. At first glance, we realize that the *vast majority of files are small* [21, 22, 23]. To wit, 90% of files are smaller than 1MByte. In our view, this can have important implications on the performance of the back-end storage system. The reason is that Personal Clouds like U1 use object storage services offered by cloud providers as data store, which has not been designed for storing very small files [15].

In this sense, Fig. 28(b) shows the file size distribution of the most popular file extensions in U1. Non-surprisingly, the distributions are very disparate, which can be used to model realistic workloads in Personal Cloud benchmarks [10]. It is worth to note that in general, incompressible files like zipped files or compressed media are larger than compressible files (docs, code). This observation indicates that compressing files *does not provide much benefits* in many cases.

File types: number vs storage space. We classified files belonging to the 55 most popular file extensions into 7 categories: Pics (`.jpg`, `.png`, `.gif`, etc.), Code (`.php`, `.c`, `.js`, etc.), Docs (`.pdf`, `.txt`, `.doc`, etc.), Audio/Video (`.mp3`, `.wav`, `.ogg`, etc.), Application/Binary (`.o`, `.msf`, `.jar`, etc.) and Compressed (`.gz`, `.zip`, etc.). Then, for each category, we calculated the ratio of the number of files to the total in the system. We did the same for the storage space. This captures the relative importance of each content type.

Fig. 28(c) reveals that Audio/Video category is one of the most relevant types of files regarding the share of consumed storage, despite the fraction of files belonging to this class is low. The reason is that U1 users stored `.mp3` files, which are usually larger than other popular text-based file types.

Further, the Code category contains the highest fraction of files, indicating that many U1 users are code developers who frequently update such files, despite the storage space required for this category is minimal. Docs are also popular (10.1%), subject to updates and hold 6.9% of the storage share. Since the U1 desktop client lacks delta updates and deferred sync, such frequent updates suppose a high stress for desktop clients and induce significant network overhead [12].



(a) Total number of requests per type.

(b) CDF of traffic across users.

(c) Lorenz curve of traffic vs active users.

Figure 29: User requests and consumed traffic in U1 for one month.

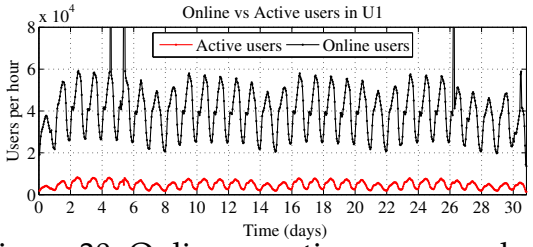


Figure 30: Online vs active users per hour.

7.6 Understanding User Behavior

Understanding the behavior of users is a key source of information to optimize large-scale systems. This section provides several insights about the behavior of users in U1.

7.7 Distinguishing Online from Active Users

Online and active users. We consider a user as *online* if his desktop client exhibits any form of interaction with the server. This includes automatic client requests involved in maintenance or notification tasks, for which the user is not responsible. Moreover, we consider a user as *active* if he performs data management operations on his volumes, such as uploading a file or creating a new directory.

Fig. 30 offers a time-series view of the number of online and active users in the system per hour. Clearly, *online users are more numerous than active users*: The percentage of active users ranges from 3.49% to 16.25% at any moment in the trace. This observation reveals that the actual storage workload that U1 supports is light compared to the potential usage of its user population, and gives a sense on the scale and costs of these services with respect to their popularity.

Frequency of user operations. Here we examine how frequent the protocol operations are in order to identify the hottest ones. Fig. 29(a) depicts the absolute number of each operation type. As shown in this figure, the most frequent operations correspond to *data management operations*, and in particular, those operations that relate to the download, upload and deletion of files.

This result is very interesting, because it proves that the U1 protocol scales well, since the operations that users issue to manage their sessions and are typically part of the session start up such as `ListVolumes` are significantly less frequent. And consequently, the major part of

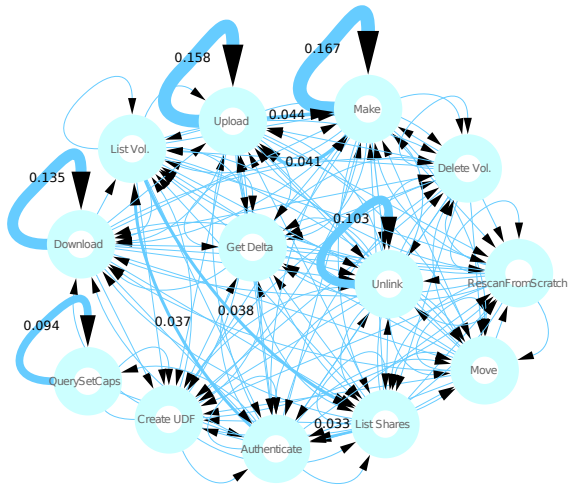


Figure 31: Desktop client transition graph through API operations. Global transition probabilities are provided for main edges.

the processing burden comes from active users as desired. This is essentially explained by the fact that the U1 desktop client does not need to regularly poll the server during idle times, thereby limiting the number of requests not linked to data management.

As we will see in 7.8, the frequency of API operations will have an immediate impact on the back-end performance.

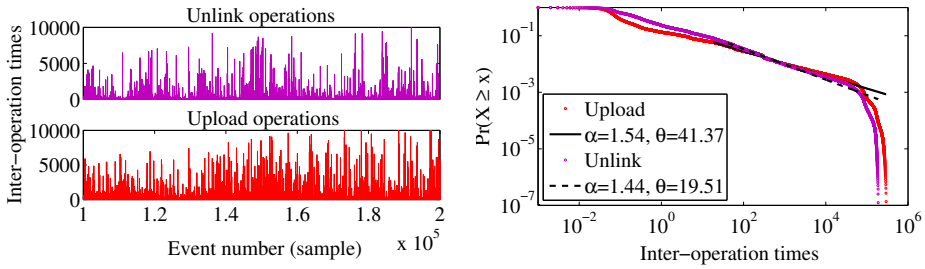
Traffic distribution across users. Now, we turn our attention to the distribution of consumed traffic across users. In Fig. 29(b) we observe an interesting fact: in one month, only 14% of users downloaded data from U1, while uploads represented 25%. This indicates that a minority of users are responsible for the storage workload of U1.

To better understand this, we measure how (un)equal the traffic distribution across active users is. To do so, we resort to the Lorenz curve and the Gini coefficient[11] as indicators of inequality. The Gini coefficient varies between 0, which reflects complete equality, and 1, which indicates complete inequality (i.e., only one user consumes all the traffic). The Lorenz curve plots the proportion of the total income of the population (y axis) that is cumulatively earned by the bottom x% of the population. The line at 45 degrees thus represents perfect equality of incomes.

Fig. 29(c) reports that the consumed traffic across active users is very unequal. That is, the Lorenz curve is very far from the diagonal line and the Gini coefficient is close to 1. The reason for this inequality is clear: 1% of users account for the 65.6% of the total traffic (147.52TB). Providers may benefit from this fact by identifying and treating these users more efficiently.

Types of user activity. To study the activity of users, we used the same user classification than Drago et al. in [9]. So we distinguished among *occasional*, *download/upload only* and *heavy* users. A user is *occasional* if he transfers less than 10KB of data. Users that exhibit more than three orders of magnitude of difference between upload and download (e.g., 1GB versus 1MB) traffic are classified as either *download-only* or *upload-only*. The rest of users are in the *heavy* group.

[11] `http://en.wikipedia.org/wiki/Gini_coefficient`



(a) (b)

Figure 32: Time-series user operations inter-arrival times and their approximation to a power-law.

Given that, we found that 71.95% of users are occasional, 13.94% upload-only, 4.24% download-only and 9.86% are heavy users. Our results clearly differ from the ones reported in [9], where users are 30% occasional, 7% upload-only, 26% download-only and 37% heavy. This may be explained by two reasons: (i) the usage of Dropbox is more extended that the usage of U1, and (ii) users in a university campus are more active than other types of users captured in our trace.

7.7.1 Characterizing User Interactions

User-centric request graph. To analyze how users interact with U1, Fig. 31 shows the sequence of operations that desktop clients issue to the server in form of a graph. Nodes represent the different protocol operations executed. And edges describe the transitions from one operation to another. The width of edges denotes the global frequency of a given transition. Note that this graph is user-centric, as it aggregates the different sequence of commands that every user executes, not the sequence of operations as they arrive to the metadata service.

Interestingly, we found that the *repetition of certain operations* becomes really frequent across clients. For instance, it is highly probable that when a client transfers a file, the next operation that he will issue is also another transfer —either upload or download.

This phenomenon can be partially explained by the fact that many times users synchronize data at *directory granularity*, which involves repeating several data management operations in cascade. File editing can be also a source of recurrent transfer operations. This behavior can be exploited by predictive data management techniques in the server side (e.g., download prefetching).

Other sequences of operations are also highlighted in the graph. For instance, once a user is authenticated, he usually performs a `ListVolumes` and `ListShares` operations. This is a regular initialization flow for desktop clients. We also observe that `Make` and `Upload` operations are quite mixed, evidencing that for uploading a file the client first needs to create the metadata entry for this file in U1.

Burstiness in user operations. Next, we analyze interarrival times between consecutive operations of the same user. We want to verify whether inter-operation times are Poissonian or not, which may have important implications to the back-end performance. To this end, we followed the same methodology proposed in [27, 28], and obtained a time-series view of `Unlink` and `Upload` inter-operation times and their approximation to a power-law

distribution in Fig. 32.

Fig. 32(a) exhibits large spikes for both `Unlink` and `Upload` operations, corresponding to *very long inter-operation times*. This is far from an exponential distribution, where long inter-operation times are negligible. This shows that the interactions of users with U1 are not Poissonian [27].

Now, we study if the `Unlink` and `Upload` inter-operation times exhibit *high variance*, which indicates *burstiness*. In all cases, while not strictly linear, these distributions show a downward trend over almost six orders of magnitude. This suggests that high variance of user inter-arrival operations is present in time scales ranging from seconds to several hours. Hence, users issue requests in a *bursty non-Poissonian way*: during a short period a user sends several operations in quick succession, followed by long periods of inactivity. A possible explanation to this is that users manage data at the *directory granularity*, thereby triggering multiples operations to keep the files inside each directory in sync.

Nevertheless, we cannot confirm the hypothesis that these distributions are heavy-tailed. Clearly, Fig. 32(b) visually confirms that the empirical distributions of user `Unlink` and `Upload` inter-arrivals can be only approximated with $P(x) \approx x^{-\alpha}, \forall x > \theta$, $1 < \alpha < 2$, for a central region of the domain.

We also found that metadata operations follow more closely a power-law distribution than data operations. The reason is that the behavior of metadata inter-operation times are not affected by the actual data transfers.

In conclusion, we can see that user operations are bursty, which has strong implications to the operation of the back-end servers (7.8).

7.7.2 Inspecting User Volumes

Volume contents. Fig. 33 illustrates the relationship between files and directories within user volumes. As usual, files are much more numerous than directories. And we have that over 60% of volumes have been associated with at least one file. For directories, this percentage is only of 32%, but there is a strong correlation between the number of files and directories within a volume: Pearson correlation coefficient is 0.998. What is relevant is, however, that a small fraction of volumes is heavy loaded: 5% of user volumes contain more than $1,000$ files.

Shared and user-defined volumes. At this point, we study the distribution of user-defined/shared volumes across users. As pointed out by Canonical engineers, sharing is not a popular feature of U1. Fig. 34 shows that only 1.8% of users exhibits at least one shared volume. On the contrary, we observe that user-defined volumes are much more popular; we detected user-defined volumes in 58% of users —the rest of users only use the root volume. This shows that the majority of users have some degree of expertise using U1.

Overall, these observations reveal that U1 was used more as a storage service rather than for collaborative work.

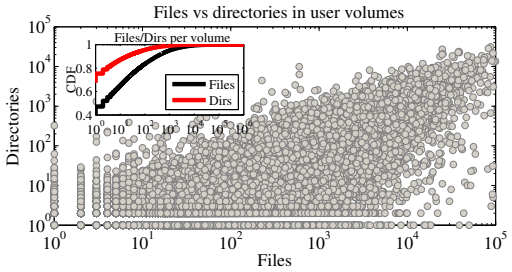


Figure 33: Files and directories per volume.

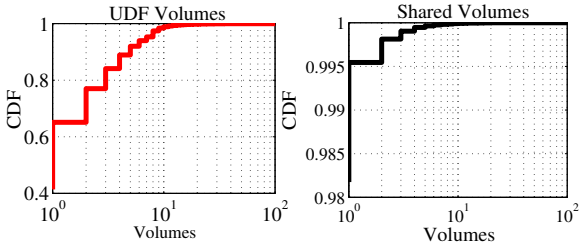


Figure 34: Distribution of shared/user-defined volumes across users.

7.8 Metadata Back-end Analysis

In this section, we focus on the interactions of RPC servers against the metadata store. We also quantify the role of the Canonical authentication service in U1.

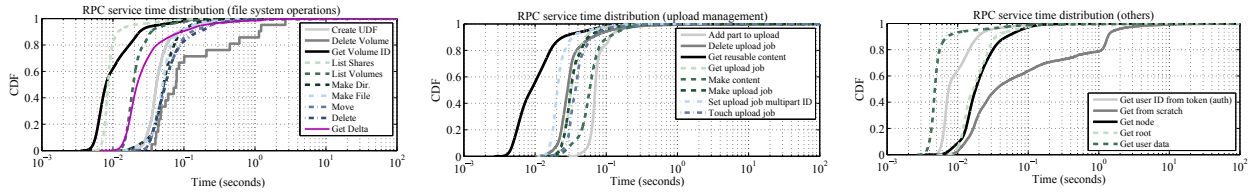
7.8.1 Performance of Metadata Operations

Here we analyze the performance of RPC operations that involve contacting the metadata store.

Fig. 35 illustrates the distribution of service times of the different RPC operations. As shown in the figure, all RPCs exhibit *long tails of service time distributions*: from 7% to 22% of RPC service times are very far from the median value. This issue can be caused by several factors, ranging from interference of background processes to CPU power saving mechanisms, as recently argued by Li et al. in [16].

Also useful is to understand the relationship between the service time and the frequency of each RPC operation. Fig. 36 presents a scatter plot relating RPC median service times with their frequency, depending upon whether RPCs are of type *read*, *write/update/delete* or *cascade*, i.e., whether other operations are involved. This figure confirms that the type of an RPC strongly determines its performance. First, *cascade* operations (`delete_volume` and `get_from_scratch`) are the slowest type of RPC —more than one order of magnitude slower compared to the fastest operation. Fortunately, they are relatively infrequent. Conversely, *read* RPCs, such as `list_volumes`, are the fastest ones. Basically, this is because *read* RPCs can exploit lockless and parallel access to the pairs of servers that form database shards.

Write/update/delete operations (e.g. `make_content`, or `make_file`) are slower than most *read* operations, but exhibiting comparable frequencies. This may represent a performance barrier for the metadata store in scenarios where users massively update metadata in their volumes or files.



(a) File system management RPCs. (b) Upload management RPCs. (c) Other read-only RPCs.

Figure 35: Distribution of RPC service times accessing to the metadata store.

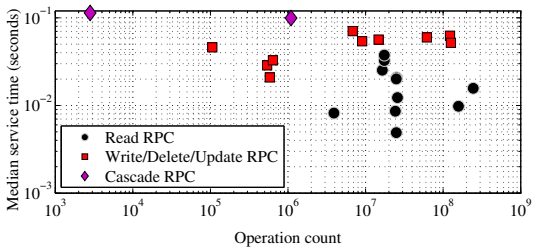


Figure 36: We classified all the U1 RPC calls into 3 categories, and every point in the plot represents a single RPC call. We show the median service time vs frequency of each RPC (1 month).

7.8.2 Load Balancing in U1 Back-end

We are interested in analyzing the internal load balancing of both API servers and shards in the metadata store. In the former case, we grouped the processed API operations by physical machine. In the latter, we distributed the RPC calls contacting the metadata store across 10 shards based on the user id, as U1 actually does. Results appear in Fig. 37, where bars are mean load values and error lines represent the standard deviation of load values across API servers and shards per hour and minute, respectively.

Fig. 37 shows that server load presents a *high variance across servers*, which is symptom of bad load balancing. This effect is present irrespective of the hour of the day and is more accentuated for the metadata store, for which the time granularity used is smaller. Thus, this phenomenon is visible in short or moderate periods of time. In the long term, the load balancing is adequate; the standard deviation across shards is only of 4.9% when the whole trace is taken.

Three particularities should be understood to explain the poor load balancing. First, user load is *uneven*, i.e., a small fraction of users is very active whereas most of them present low activity. Second, the cost of operations is *asymmetric*; for instance, there are metadata operations whose median service time is 10x higher than others. Third, users display a *bursty* behavior when interacting with the servers; for instance, they can synchronize an entire folder. So, operations arrive in a correlated manner.

We conclude that the load balancing in the U1 back-end can be significantly improved, which is object of future work.

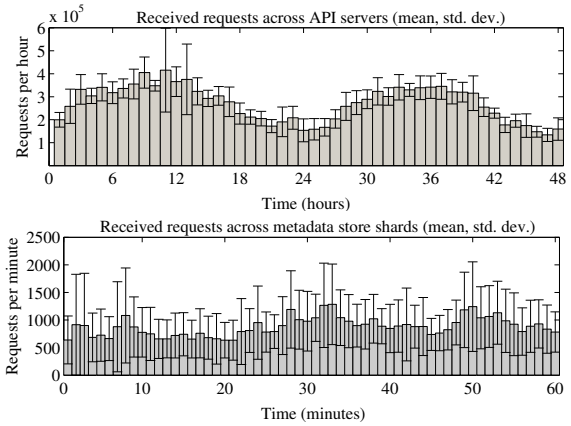


Figure 37: Load balancing of U1 API servers and metadata store shards.

7.8.3 Authentication Activity & User Sessions

Time-series analysis. Users accessing the U1 service should be authenticated prior to the establishment of a new session. To this end, U1 API servers should contact a separate and shared authentication service of Canonical.

Fig. 38 depicts a time-series view of the session management load that API servers support to create and destroy sessions, along with the corresponding activity of the authentication subsystem. In this figure, we clearly observe that the authentication and session management activity is closely related to the habits of users. In fact, daily patterns are evident. The authentication activity is 50% to 60% higher in the central hours of the day than during the night periods. This observation is also valid for week periods: on average, the maximum number of authentication requests is 15% higher on Mondays than on weekends. Moreover, we found that 2.76% of user authentication requests from API servers to the authentication service fail.

Session length. Upon a successful authentication process, a user's desktop client creates a new U1 session.

U1 sessions exhibit a similar behavior to Dropbox *home* users in [9] (Fig. 38). Concretely, 97% of sessions are shorter than 8 hours, which suggests a strong correlation with user working habits. Moreover, we also found that U1 exhibits a high fraction of very short-lived sessions (i.e. 32% shorter than 1s.), probably due to the operation of NAT and firewalls that normally mediate between clients and servers [29]. Overall, Fig. 38 suggests that *domestic users are more representative* than other specific profiles, such as university communities, for describing the connection habits of an entire Personal Cloud user population.

We are also interested in understanding the data management activity related to U1 sessions. To this end, we differentiate those sessions that exhibited any type of data management operation (e.g., upload, download, delete file, etc.) during their lifetime, namely, *active sessions*.

First, we observed that the majority of U1 sessions (and, therefore, TCP connections) do not involve any type of data management. That is, only 5.57% of connections in U1 are active (2.37M out of 42.5M), which, in turn, tend to be much longer than *cold* ones. From a back-end perspective, the unintended consequence is that a fraction of server resources is *wasted keeping alive TCP connections* of cold sessions.

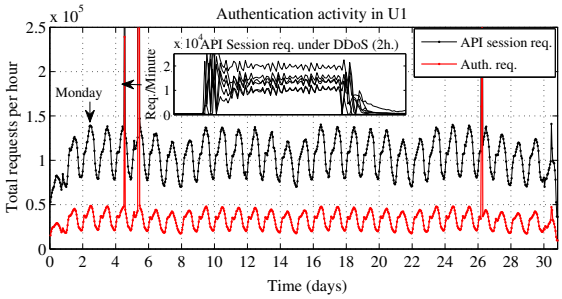


Figure 38: API session management operations and authentication service requests. The inner plot shows the number of session requests supported by API servers during a DDoS.

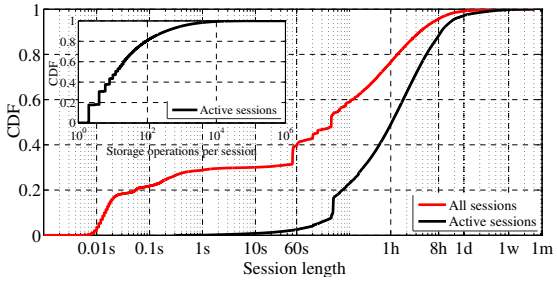


Figure 39: Distribution of session lengths and storage operations per session.

Moreover, similarly to the distribution of user activity, the inner plot of Fig. 38 shows that 80% of active sessions exhibited at most 92 storage operations, whereas the remaining 20% accounted for 96.7% of all data management operations. Therefore, the are sessions much more active than others.

A provider may benefit from these observations to optimize session management. That is, depending on a user's activity, the provider may wisely decide if a desktop client works in a *pull* (cold sessions) or *push* (active sessions) fashion to limit the number of open TCP connections [30].

7.9 Related Work

The performance evaluation of cloud storage services is an interesting topic inspiring several recent papers. Hill et al. in [31] provide a quantitative analysis of the performance of the Windows Azure Platform, including storage. Palankar et al. in [32] perform an extensive measurement against Amazon S3 to elucidate whether cloud storage is suitable for scientific Grids. Similarly, [33] presents a performance analysis of the Amazon Web Services. Liu et. al. [25] inspected in depth the workload patterns of users in the context of a storage system within a university campus. This work concentrates on macroscopic storage workload metrics and type of requests, as well as the differences in access patterns of personal and shared folders. Unfortunately, these papers provide no insights into Personal Clouds.

Perhaps surprisingly, despite their commercial popularity, only few research works have turned attention to analyze the performance of Personal Cloud storage services [9, 10, 12, 13, 14, 19, 34]. We classify them into two categories:

Active measurements. The first analysis of Personal Cloud storage services we are aware

is due to Hu et al. [13] that compared Dropbox, Mozy, Carbonite and CrashPlan storage services. However, their analysis was rather incomplete; the metrics provided in [13] are only backup/restore times depending on several types of backup contents. They also discuss potential privacy and security issues comparing these vendors. The work of Hu et al. [13] was complemented by Gracia-Tinedo et al. [14] that extensively studied the REST interfaces provided by three big players in the Personal Cloud arena, analyzing important aspects of their QoS and potential exploitability [19].

Recently, Drago et al. [10] presented a complete framework to benchmark Personal Cloud desktop clients. One of their valuable contributions is to set a benchmarking framework addressed to compare the different data reduction techniques implemented in desktop clients (e.g, file bundling).

Passive measurements. Drago et al. [9] presented an external measurement of Dropbox in both a university campus and residential networks. They analyzed and characterized the traffic generated by users, as well as the workflow and architecture of the service. [34] extended that measurement by modeling the behavior of Dropbox users. However, [9] does not provide insights on the metadata back-end of Dropbox, since this is not possible from external vantage points. Moreover, these works [9, 34] study the storage workload and user behavior on specific communities (e.g., university campus) that may lead to false generalizations. In contrast, we analyze the entire population of U1.

Furthermore, Li et al. [12] analyzed a group of Personal Cloud users in both university and corporate environments. Combined with numerous experiments, they studied the efficiency of file sync protocols, as well as the interplay between data reduction techniques integrated in desktop clients and users' workload (update frequency, file compressibility).

Key differences with prior work. The main difference with previous works is that we study in details the *metadata back-end servers* of a Personal Cloud, instead of simply measuring it from outside. The unique aspect of our work is that most of our insights could have been obtained only by taking a perspective from within a data center; this goes, for example, for the internal infrastructure of the service, or the performance of the metadata store, to name a few.

Furthermore and apart from guiding simulations and experiments, we believe that the present analysis will help researchers and practitioners optimize several aspects of these services, such as file synchronization [11, 17] and security [35], among others.

7.10 Discussion and Conclusions

We focus on understanding the nature of Personal Cloud services by presenting the internal structure and measurement study of UbuntuOne (U1). The objectives of our work are threefold: (i) to unveil the internal operation and infrastructure of a real-world provider, (ii) to reconfirm, expand and contribute observations on these systems to generalize their characteristics, and (iii) to propose potential improvements for these systems.

This work unveils several aspects that U1 *shares* with other large-scale Personal Clouds. For instance, U1 presents clear similarities with Dropbox regarding the way of decoupling data and metadata of users, which seems to be a standard design for these systems [17].

Also, we found characteristics in the U1 workload that reconfirm observations of prior works [9, 12] regarding the *relevance of file updates*, the *effectiveness of deduplication* or the execution of user operations in *long sequences*, among other aspects. Therefore, our analysis and the resulting dataset will enable researchers to get closer to the nature of a real-world Personal Cloud.

Thanks to the scale and back-end perspective of our study, we expanded and contributed insights on these services. That is, we observed that the distribution of activity across users in U1 is even *more skewed* than in Dropbox [9] or that the behavior of domestic users dominate session lengths in U1 compared to other user types (e.g., university). Among the novelties of this work, we modeled the *burstiness of user operations*, we analyzed the *behavior of files* in U1, we provided evidences of *DDoS attacks* to this service, and we illustrated the performance of the U1 *metadata back-end*.

An orthogonal conclusion that we extract from our study is that understanding *the behavior of users is essential* to adapt the system to its actual demands and reduce costs. In the following, we relate some of our insights to the running costs of U1 as well as potential optimizations, which may be of independent interest for other large-scale systems:

Optimizing storage matters. A key problem to the survival of U1 was the growing costs of outsourcing data storage [36], which is directly related to the data management techniques integrated in the system. For instance, the fact that file updates were responsible for 18.5% of upload traffic in U1, mainly due to the lack of delta updates in the desktop client, gives an idea of the margin of improvement (7.5.1). Actually, we confirmed that a simple optimization like file-based deduplication could readily save 17% of the storage costs. This calls to further research and the application of advanced data reduction techniques, both at the client and server sides.

Take care of user activity. This observation is actually very important, as we found that 1% of U1 users generated 65% of traffic (7.7), showing a weak form of the Pareto Principle. That is, a very small fraction of the users represented most of the OPEX for U1. A natural response may be to limit the activity of free accounts, or at least to treat active users in a more cost effective way. For instance, distributed caching systems like Memcached, data prefetching techniques, and advanced sync deferment techniques [12] could easily cut the operational costs down. On the other hand, U1 may benefit from cold/warm storage services (e.g., Amazon Glacier, f4 [26]) to limit the costs related to most inactive users.

Security is a big concern. Another source of expense for a Personal Cloud is related to its exploitation by malicious parties. In fact, we found that DDoS attacks aimed at sharing illegal content via U1 are indeed frequent. The risk that these attacks represent to U1 is in contrast to the limited automation of its countermeasures. We believe that further research is needed to integrate secure storage protocols and automated countermeasures for Personal Clouds. In fact, understanding the common behavior of users in a Personal Cloud (e.g., storage, content distribution) may provide clues to automatically detect anomalous activities.

A Upload Management in U1

The management of file uploads is one of the most complex parts in the U1 architecture[12]. Specifically, U1 resorts to the multipart upload API offered by Amazon S3[13]. The lifecycle of an upload is closely related to this API, where several U1 RPC calls are involved (see Table 7.1b).

Internally, U1 uses a persistent data structure called `uploadjob` that keeps the state of a multipart file transfer between the client and Amazon S3. The main objective of multipart uploads in U1 is to provide user with a way of interrupting/resuming large upload data transfers. `uploadjob` data structures are stored in the metadata store during their life-cycle. RPC operations during the multipart upload process guide the lifecycle of `uploadjobs` (see Fig. 40).

Upon the reception of an upload request, U1 first checks if the file content is already stored in the service, by means of a SHA-1 hash sent by the user. If deduplication is not applicable to the new file, a new upload begins. The API server that handles the upload sends an RPC to create an entry for the new file in the metadata store.

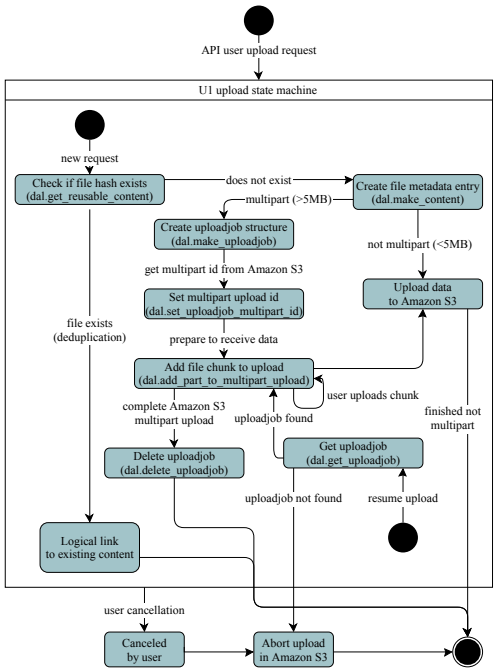


Figure 40: Upload state machine in U1.

In the case of a multipart upload, the API server creates a new `uploadjob` data structure to track the process. Subsequently, the API process requests a multipart id to Amazon S3 that will identify the current upload until its termination. Once the id is assigned to the `uploadjob`, the API server uploads to Amazon S3 the chunks of the file transferred by the user (5MB), updating the state of the `uploadjob`.

When the upload finishes, the API server deletes the `uploadjob` data structure from the metadata store and notifies Amazon S3 about the completion of the transfer.

[12]Downloads are simpler: API servers only perform a single request to Amazon S3 for forwarding the data to the client.

[13]`http://docs.aws.amazon.com/AmazonS3/latest/dev/UsingRESTAPImpUpload.html`

Finally, U1 also executes a periodic garbage-collection process on `uploadjob` data structures. U1 checks if an `uploadjob` is older than one week (`dal.touch_uploadjob`). In the affirmative case, U1 assumes that the user has canceled this multipart upload permanently and proceeds to delete the associated `uploadjob` from the metadata store.

References

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

[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] "Rest api principles," http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api.

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

[7] K. Eshghi and H. K. Tang, "A Framework for Analyzing and Improving Content-Based Chunking Algorithms," http://www.hpl.hp.com/techreports/2005/HPL-2005-30R1.pdf, 2005.

[8] F. Research, "The personal cloud: Transforming personal computing, mobile, and web markets." http://www.forrester.com, 2011.

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

[10] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras, "Benchmarking personal cloud storage," in ACM SIGCOMM IMC'13, 2013, pp. 205–212.

[11] Z. Li, C. Wilson, Z. Jiang, Y. Liu, B. Y. Zhao, C. Jin, Z.-L. Zhang, and Y. Dai, "Efficient batched synchronization in dropbox-like cloud storage services," in ACM/IFIP/USENIX Middleware'13, 2013, pp. 307–327.

[12] Z. Li, C. Jin, T. Xu, C. Wilson, Y. Liu, L. Cheng, Y. Liu, Y. Dai, and Z.-L. Zhang, "Towards network-level efficiency for cloud storage services," in ACM IMC'14, 2014.

[13] W. Hu, T. Yang, and J. Matthews, "The good, the bad and the ugly of consumer cloud storage," ACM SIGOPS Operating Systems Review, vol. 44, no. 3, pp. 110–115, 2010.

[14] R. Gracia-Tinedo, M. Sánchez-Artigas, A. Moreno-Martınez, C. Cotes, and P. Garcıa-López, "Actively measuring personal cloud storage," in IEEE CLOUD'13, 2013, pp. 301–308.

[15] R. Sears, C. Van Ingen, and J. Gray, "To blob or not to blob: Large object storage in a database or a filesystem?" Microsoft Research, Tech. Rep., 2007.

[16] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble, "Tales of the tail: Hardware, os, and application-level sources of tail latency," in ACM SoCC'14, 2014.

[17] P. García-López, S. Toda-Flores, C. Cotes-González, M. Sánchez-Artigas, and J. Lenton, "Stacksync: Bringing elasticity to dropbox-like file synchronization," in ACM/IFIP/USENIX Middleware'14, 2014, p. In press.

[18] "FP7 cloudspaces EU project," http://cloudspaces.eu.

[19] R. Gracia-Tinedo, M. Sánchez-Artigas, and P. García-López, "Cloud-as-a-gift: Effectively exploiting personal cloud free accounts via REST APIs," in IEEE CLOUD'13, 2013, pp. 621–628.

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

[21] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, "Measurements of a distributed file system," in ACM SIGOPS Operating Systems Review, vol. 25, no. 5, 1991, pp. 198–212.

[22] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch, "A five-year study of file-system metadata," ACM Transactions on Storage (TOS), vol. 3, no. 3, p. 9, 2007.

[23] A. W. Leung, S. Pasupathy, G. R. Goodson, and E. L. Miller, "Measurement and analysis of large-scale network file system workloads." in USENIX ATC'08, vol. 1, no. 2, 2008, pp. 213–226.

[24] W. Hsu and A. Smith, "Characteristics of i/o traffic in personal computer and server workloads," IBM Systems Journal, vol. 42, no. 2, pp. 347–372, 2003.

[25] S. Liu, X. Huang, H. Fu, and G. Yang, "Understanding data characteristics and access patterns in a cloud storage system," in IEEE/ACM CCGrid'13, 2013, pp. 327–334.

[26] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar, "f4: Facebook's warm blob storage system," in USENIX OSDI'14, 2014, pp. 383–398.

[27] A.-L. Barabasi, "The origin of bursts and heavy tails in human dynamics," Nature, vol. 435, no. 7039, pp. 207–211, 2005.

[28] M. E. Crovella and A. Bestavros, "Self-similarity in world wide web traffic: evidence and possible causes," IEEE/ACM Transactions on Networking, vol. 5, no. 6, pp. 835–846, 1997.

[29] S. Hätönen, A. Nyrhinen, L. Eggert, S. Strowes, P. Sarolahti, and M. Kojo, "An experimental study of home gateway characteristics," in ACM IMC'10, 2010, pp. 260–266.

[30] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy, "Adaptive push-pull: disseminating dynamic web data," in ACM WWW'01, 2001, pp. 265–274.

[31] Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez, and M. Humphrey, "Early observations on the performance of windows azure," in HPDC '10, 2010, pp. 367–376.

[32] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel, "Amazon s3 for science grids: a viable solution?" in DADC'08, 2008, pp. 55–64.

[33] A. Bergen, Y. Coady, and R. McGeer, "Client bandwidth: The forgotten metric of online storage providers," in IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, 2011, pp. 543–548.

[34] G. Gonçalves, I. Drago, A. P. C. da Silva, A. B. Vieira, and J. M. Almeida, "Modeling the dropbox client behavior," in Proceedings of the International Conference on Communications, ICC, vol. 14, 2014.

[35] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E.Weippl., "Dark clouds on the horizon: Using cloud storage as attack vector and online slack space," in USENIX Security, 2011, pp. 5–8.

[36] J. Silber, "Shutting down Ubuntu One file services," http://blog.canonical.com/2014/04/02/shutting- down-ubuntu-one-file-services/, April 2014.