SEVENTH FRAMEWORK PROGRAMME

CloudSpaces

(FP7-ICT-2011-8)

Open Service Platform for the Next Generation of Personal Clouds

D5.2 Service Platform reference prototype

Due date of deliverable: 31-10-2014
Actual submission date: 15-10-2014

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	54
WP/Task related to this document	WP5
WP/Task responsible	EOS
Author(s)	Adrián Moreno (URV), Pedro García (URV), Raquel Sánchez (EOS), Anastasio Illana (NEC), Alberto Gomez (NEC),
Partner(s) Contributing	URV, EOS, NEC, TST
Document ID	CLOUDSPACES_D5.2_141015_Public.pdf
Abstract	This report includes documentation, tutorials, and specifications of the interoperability protocol created by NEC and URV in their respective personal cloud solutions (i.e. Cloud Storage and StackSync). As well as proof-of-concept tools created by EyeOS and Tissat that make use of the service platform created in StackSync.
Keywords	Cloud storage, synchronization, sharing, interoperability, Personal Cloud

Table of Contents

1 Executive summary

This report includes documentation, tutorials, and specifications of the interoperability protocol created by NEC and URV in their respective personal cloud solutions (i.e. Cloud Storage and StackSync). As well as proof-of-concept tools created by EyeOS and Tissat that make use of the service platform created in StackSync.

First, the URV introduces StackSync and details the requirements to be able to comply with the interoperability protocol. Prior to implementing the interoperability protocol to allow sharing resources between heterogeneous personal clouds, StackSync needs to implement the sharing mechanism and be able to share files and folders among its own users. Afterwards, the URV details how StackSync has been modified to implement the protocol. Moreover, they present a novel architecture for file synchronization to tackle the issues present in synchronization services like StackSync.

Then, NEC presents a detailed guide about the interoperability protocol of CloudSpaces in NEC's personal cloud called Cloud Storage. It also describes the interoperability process through screenshots, examples and all necessary calls with its parameters for its proper operation.

EyeOS presents a set of tools that make use of the service platform created by StackSync. First, they detail the authentication protocol so that users can link their EyeOS accounts with StackSync. Then, it is shown how StackSync file system is integrated into EyeOS's virtual desktop, being able to seamlessly navigate between files and folder independently of their source (i.e. EyeOS or StackSync). EyeOS also shows two tools that use the persistence services: eyeOS comments, and eyeOS calendar. Both tools are synchronized using a U1DB service that incorporate value-added features to their platform. Finally, EyeOS details how they have integrated StackSync's sharing functionality into their application, allowing users to share folders with others.

Finally, Tissat presents its contributions to the CloudSpaces service platform which includes the management interface to enable administrators to set up quotas and have complete control over users. They also worked on security issues related to Keystone and deploying StackSync with SSL. Among other tasks, they developed and published the native StackSync application for iOS, leveraging iPhone users to access the platform.

2 StackSync

2.1 Introduction

StackSync is a scalable open source Personal Cloud that implements the basic components to create a synchronization tool. The StackSync framework is compounded by three main components: a synchronization server, OpenStack Swift, and desktop / mobile clients.

While synchronization servers are in charge of processing metadata and provide the logic, OpenStack Swift is focused on storing the raw chunked data. Users are able to keep their files synced by using the StackSync clients available for the main desktop and mobile operating systems.

StackSync is based on an advanced synchronization technology, similar to Dropbox, with data optimizations (chunking, compression and push mechanisms) that allows it to scale to thousands of users with an efficient use of cloud resources. It also provides data-sharing mechanisms to groups of users. A RESTful API has been built as a Swift module living at the proxies to allow the StackSync mobile apps and other third-party apps to interact with the available resources.

The separation between data and metadata allows StackSync to be deployed in different configurations depending on the needs. Using the private configuration, organizations can deploy it on-premise, as both OpenStack Swift and StackSync Server run on their facilities. The public configuration may be suitable for Cloud providers willing to offer a public synchronization service to their customers. Whereas the Hybrid configuration allows organizations to keep their metadata on-premise while storing the raw data at a public Cloud provider, retaining control over their information. Finally, as an open-source project, StackSync welcomes and encourages any kind of collaboration from the community.

2.2 Sharing

Prior to implementing the interoperability protocol to allow sharing resources between heterogeneous personal clouds, StackSync needs to be able to share files and folders among its own users. As seen in figure 1, each user in StackSync is assigned a logical workspace, which represents the relation between files and users. The workspace approach facilitates authorization of users, in contrast to a per-file authorization, and enables us to directly map workspaces into OpenStack Swift containers.

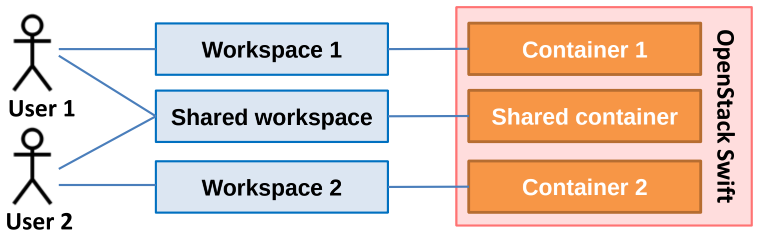


Figure 1: StackSync workspace / container relation

To guarantee that only the owner can access its physical data, we set up an access control list (ACL) in each OpenStack Swift container by using the `X-Container-Read` and `X-Container-Write` parameters. This solution works fine for users just wanting to synchronize and backup files across their devices, but limitations arise when it comes to sharing.

If we keep shared resources in the same workspace and container, we would need a fine-grained access control to ensure that only the right users access the information. In other words, a user may be allowed to access only a subset of files inside a container, complicating permission-handling tasks. Consequently, we would lose the coarse granularity offered by workspaces and containers that guaranteed us that if a user has access to a particular workspace/container, then it will also has access to all resource contained inside.

In order to keep the benefits of our permission model, we proposed an approach that consisted in addressing shared folders as independent workspaces and containers. In figure 1 we observe that when a user shares a folder, a new workspace is created and the folder and its content are moved to the new workspace. This new workspace will be accessible by the user that originated the order (owner) and users that were explicitly given access to the shared folder (invitees).

From the data point of view, a new container is created and all chunks belonging to the shared content are moved to the new location. OpenStack Swift provides a request to avoid downloading and reuploading the content to the new container, instead, we copy the objects using the `X-Copy-From` request, which does not incur in any data transference. After a successful copy operation, old chunks are deleted from the origin container.

Syncing protocol upgrade

Whenever the desktop client is ordered to share a folder, it sends a "createShareProposal" to the SyncService, providing the folder to be shared and a list of emails belonging to the invited users. This is a synchronous call that will return whether the sharing proposal was created successfully or not. Invitees will receive an email and an instant push notification informing them about the proposal. Once accepted, invitees will be added to the new workspace and will receive all future events related to the shared folder. Furthermore, they will be incorporated to the container ACL so that they can download and upload content.

API upgrade

In order to allow the sharing functionality on mobile devices, we upgraded the API by adding three new actions on the folder resource. First, the actions "share" and "unshare" let applications to create a sharing proposal and remove users from it. Moreover, applications can also list the members of a shared folder by using the "members" action on a folder. It will return all active participants that are granted access to a particular folder.

Share folder

An application can share a folder with other users by issuing an HTTP POST request to the URL that represents the folder resource (e.g. `/folder/214874/share`) . The app must provide a JSON object that represents the users that will be invited to the folder.

The request body must include an element "share_to" which will contain an array of emails representing the users that will be invited to the shared folder.

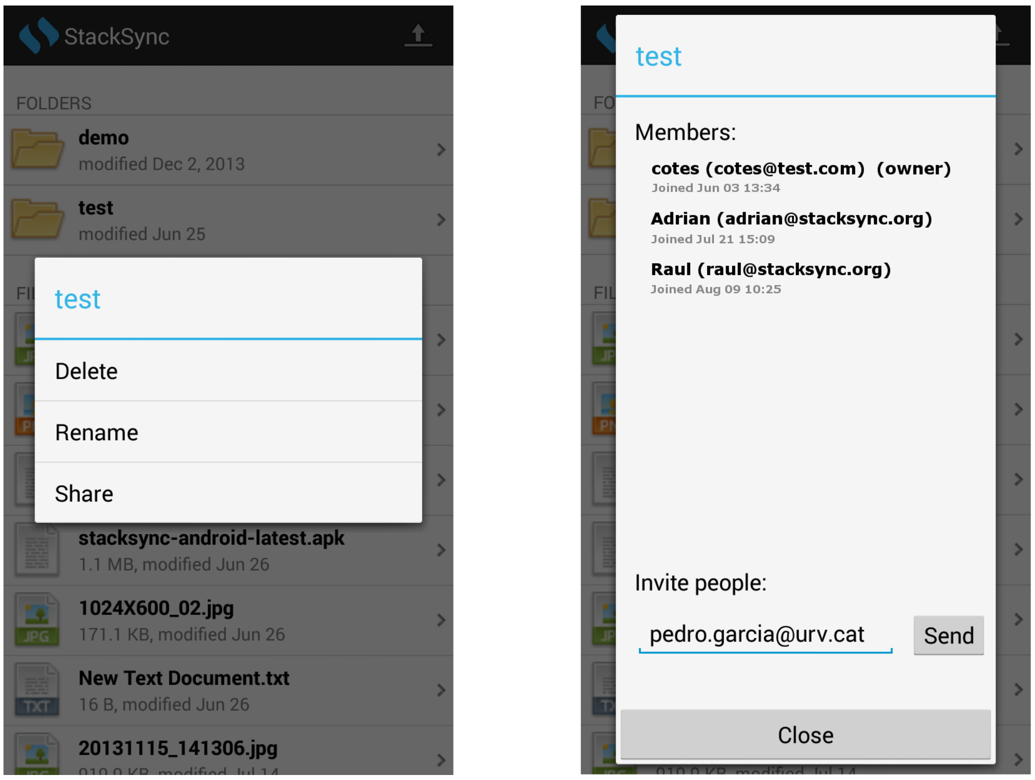


Figure 2: Sharing functionality in StackSync's Android app

Unshare folder

An application can stop sharing a folder at any time just issuing an HTTP POST request to the URL that represents the folder resource (e.g. `/folder/214874/unshare`). This action does not require a payload and will remove the authenticated user from the shared folder. However, if a payload is provided and the authenticated user is the owner of the shared folder, the request body may include an element "unshare_to" which will remove the given users from the shared folder.

Get folder members

To retrieve information about the users that have access to a folder, an application submits an HTTP GET request to the folder resource that represents the folder (e.g. `/folder/214874/members`).

The response body contains a JSON array enclosing dictionaries with the following keys:

Element	Description
name	The name of the user
email	The email of the user
joined_at	The date the user joined the folder
is_owner	Whether the user is the owner of the folder or not. Options are True or False

2.3 Interoperability

Now that StackSync enables to share folders among its users it is time to implement the interoperability protocol.

The interoperability protocol defines three endpoints: one to receive share proposals from external personal clouds; another to receive cancellations of already-established sharing agreements; and the last one to receive access credentials for accepted sharing proposals. To this end, we created a Django module implementing the interoperability protocol. This new module can be located either with the synchronization service or in an independent server.

The next figure shows how the module interacts with the rest of the architecture,

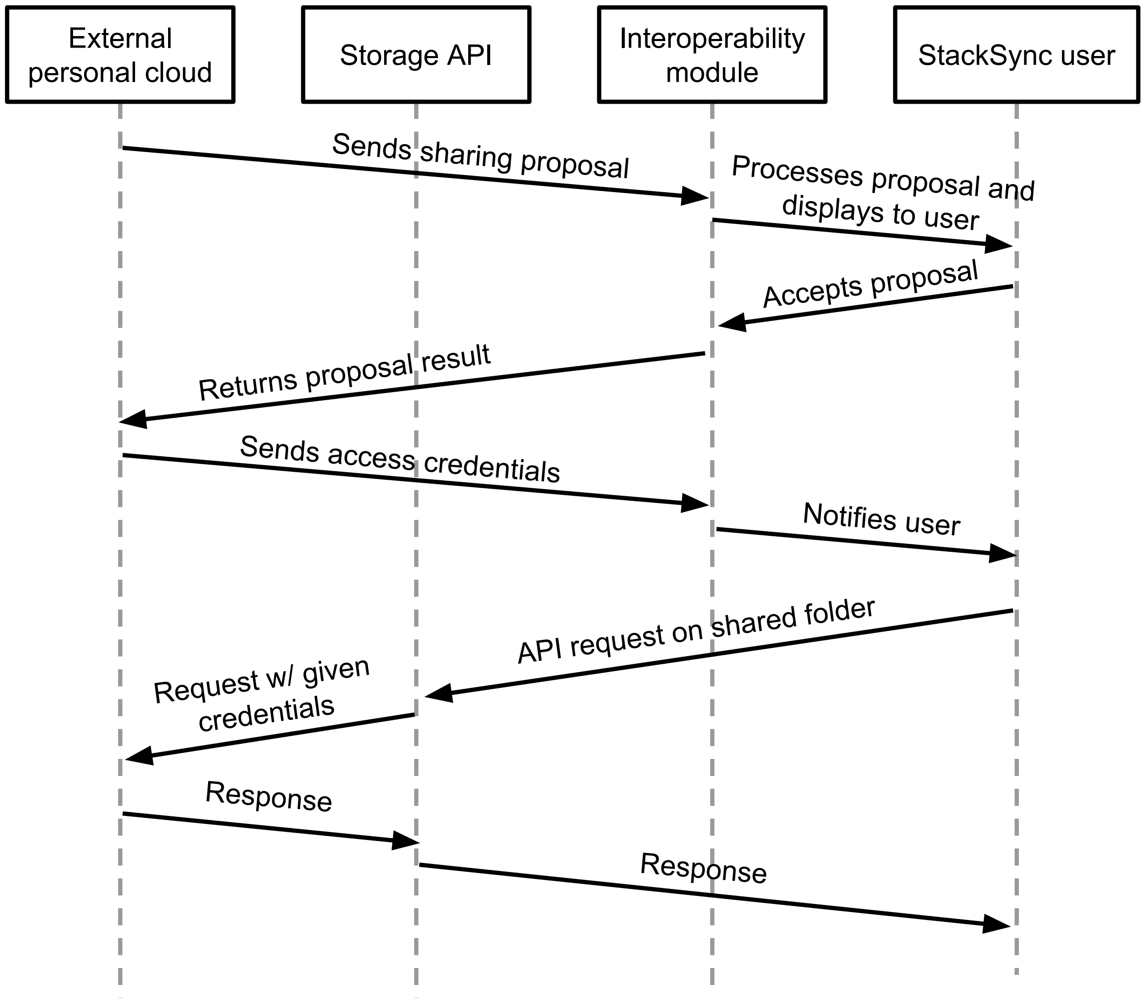
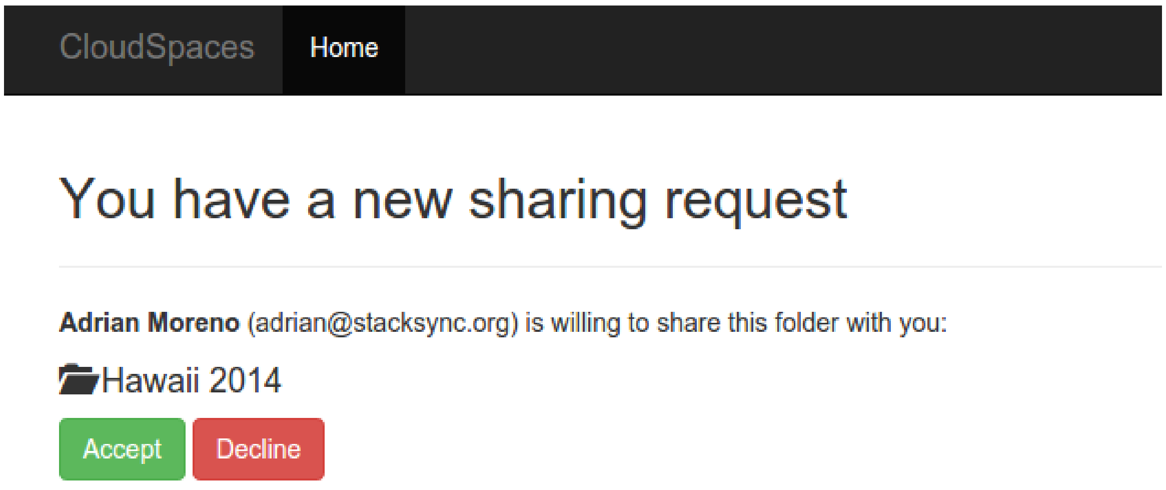


Figure 3: Interoperability protocol sequence diagram

The sequence above depicts a scenario where StackSync receives an external sharing proposal from a personal cloud that already implements the interoperability protocol. The entry point for the request is the interoperability module. It will receive the sharing proposal and check that it is well formed. Afterwards, the invitee, which is a StackSync user, will be shown the proposal details like shown in figure 4, where she will either accept it or decline it.



You have a new sharing request

Adrian Moreno (adrian@stacksync.org) is willing to share this folder with you:

Hawaii 2014

Figure 4: StackSync's share proposal permission screen

The result is then forwarded to the source personal cloud, which will reply with the access credentials in case the user accepted the proposal. These credentials are assigned to the recently shared folder and saved into the database. At this time, the interoperability process is completed and the StackSync user is notified about its newly shared folder. Whenever the user is to retrieve or upload data to the shared folder, the API will seamlessly detect the special condition of the folder and forward the request to the origin personal cloud with the given credentials. This process is performed in a fully transparent manner for the user.

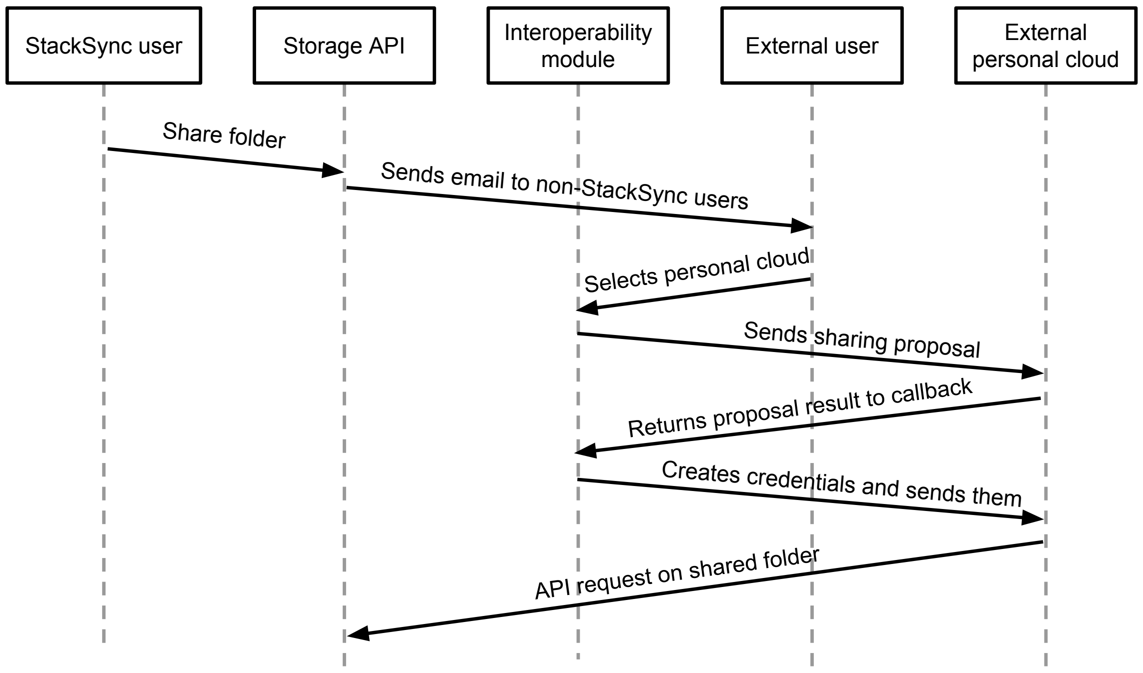


Figure 5: Interoperability protocol sequence diagram

Oppositely, when a sharing proposal is generated by a user in StackSync, the API receives the request and sends an email to the invitee, which is a user belonging to an external personal that also implements the interoperability protocol. The invitee is forwarded to a website located in StackSync where she will select her personal cloud from a list of all compatible services.

Once selected, she will be redirected to a website located on her personal cloud where she will be show the details of the folder and will be requested to accept or deny the proposal. StackSync will get notified of the user's choice and will hand the access credentials (OAuth tokens) to the other personal cloud. Afterwards, whenever the invitee wants to access to the shared folder, the StackSync API will receive the request and process it as any other API call.

2.4 Elasticity

In the last years, we have witnessed a rush of Personal Cloud Storage services offering file synchronization to millions of users. In this line, Dropbox [1] has achieved massive scalability thanks to a decoupled architecture that separates control flows (Dropbox sync servers) from data flows (Amazon S3 Object Storage).

While the elasticity of Cloud Object Storage Services like Amazon S3 is ensured, the design of elastic and scalable file synchronization protocols is complex [2]. Among the major challenges, we outline the following two issues: *fine-grained programmable elasticity* and *efficient change notification* to millions of users.

The first challenge is related to the observation that scaling up some types of cloud applications is not straightforward using traditional VM resource utilization metrics (CPU, RAM, etc.) [3], because, for instance, they are not CPU or memory intensive, but I/O bound, as is the case for file synchronization [2]. In those cases, it is better to rely on metrics such as the average and message handling response times exhibited by VM instances to cope with the varying demand. This implies that fine-grained elasticity management components must be built for the synchronization service as argued in the paper.

The other challenge is that the high read-write ratio of file syncing services makes it more suitable to make use of one-to-many push communication for rapid notification. Analogously, to efficiently maintain the consistency of files, any change performed elsewhere must be advertised as soon as they occur to reduce conflicts [1], in particular, when a file is susceptible to be modified by more than one client at the same time. This requires the file syncing service to operate as quickly as possible to commit changes, along with an efficient notification service to inform clients about file mutations.

To face the above challenges, we propose a novel architecture for elastic file synchronization. The major contributions of our work are:

1. ObjectMQ: a lightweight framework for providing programmatic elasticity to distributed objects using message queues as their underlying communication middleware. The efficient use of indirect communication in our middleware removes the need for preprocessing client stubs for scaling out and down, it provides transparent load balancing mechanisms based on queues, it simplifies one-to-many communications, and it enables flexible programmatic elasticity based on queue message processing.
2. StackSync: an elastic file synchronization architecture decoupling metadata and data flows in structured and object storage services. StackSync implements predictive and reactive provisioning policies on top of ObjectMQ that adapt to real traces from the Ubuntu One service. Furthermore, the ObjectMQ unicast and multicast communication primitives have considerably simplified the code of the synchronization protocol. It also enables efficient change notification in a transparent way on top of the underlying messaging service.
3. StackSync has been extensively tested using real traces from the Ubuntu One system to validate its elasticity and efficient use of resources. Furthermore, we extended an open benchmark [4] for Personal Clouds which provides trace generators and test scripts. Using this benchmark, we compared our service with Dropbox, Box, OneDrive, and Google Drive. StackSync is a stable open source project after two years of development that is being used in several public institutions and data centers.

2.4.1 ObjectMQ

ObjectMQ is a framework which provides programmatic elasticity to distributed objects using a message queue system. In figure 6, we can see the basic architecture of our middleware, from left to right:

- **Client Stub:** It allows to call a remote object by utilizing the MOM communication layer. To make a remote call, the stub sends a message to a queue where the remote object is subscribed. Further, every stub has its own queue to receive responses from the server side.

- **MOM System:** It is the communication layer between stubs and skeletons. Every stub has its own queue to receive replies from remote objects. Figure 6 shows the two types of queues a remote object is subscribed to. Specifically, the uppermost queue is a *global* queue shared among all the different remote objects. The lower queues correspond to the *private* queues where each individual object is listening to incoming calls.

- **Remote Objects:** They are remote objects that listen to the queues and execute RPCs. To add a remote object instance into the system, our middleware provides the method: *bind(oid, remoteObject)*, which binds a particular object instance with the identifier *oid*. Internally, ObjectMQ will create a queue called *oid* where the *remoteObject* will be able to listen for new RPCs. If the queue already exists, the new instance will be simply subscribed to the queue. This binding mechanism will help scale out the system by dynamically creating new objects and subscribe them to a particular named queue, with the MOM system providing automatic load-balancing to all the objects subscribed to the queue. Due to the fact that only one of the subscribed remote objects can consume a specific message, i.e., the same message is not delivered to any other object, a separate private queue for every object is needed to support multicast.

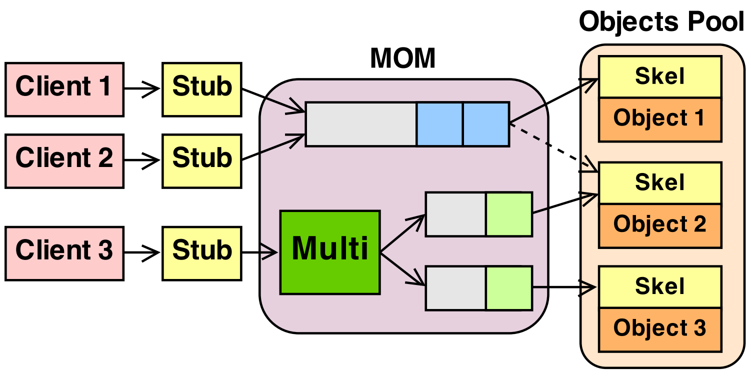


Figure 6: ObjectMQ architecture.

Figure 6 also illustrates the two types of remote invocations supported by ObjectMQ: *unicast* and *multicast*. Unicast invocations, issued by Client1 and Client2 in this example, are processed through the global queue. For this type of call, the MOM system will deliver the RPC to the first remote object that is idle. In multicast invocations, issued by Client3 in this example, the same RPC will be sent to all the private queues bound with the same *oid*, i.e., Multi(N queues, 1 object per queue). More technically, as our current implementation of ObjectMQ is built over the AMQP protocol [5], we simply use a type of *exchange*[1] called *fanout exchange* to support multicast. This type of exchange broadcasts all the messages it receives to all the queues that have been bound to a specified name *oid*.

The major building blocks of our architecture are: (i) a lightweight communication layer with small stubs and skeleton; (ii) novel communication primitives offering stateless one-to-one and one-to-many synchronous and asynchronous invocations, and (iii) an extensible provisioning model enabling third parties to create their own policies controlling the size of server object pools.

[1]In the AQMP parlance, an *exchange* can be viewed as a mailbox, distributing copies of the message to one or more queues according to specified bindings.

2.4.2 Communication Layer

Our major aim is to create a minimalist communication layer delegating complex communications to the messaging services. The programming model must be simple and it must completely hide queue and message management from developers. Our middleware avoids any stub compilation or preprocessing phase thanks to the use of dynamic stubs. Although we are inspired in Java RMI, we decided to create our own naming service and method decorators in order to simplify the overall communication model.

Our middleware delegates as much responsibilities as it can to the underlying MOM system. The underlying MOM system will be the responsible for balancing load while avoiding the loss of messages. Further, it will help us implement a naming service for the objects. Inspired by Java RMI, ObjectMQ also provides the methods *bind*, to bind a remote object to a specified name, and *lookup*, to return the remote reference bound to a given name. However, instead of using a centralized naming registry, we will use the queues to bind objects with their identifiers. As a result, whenever a stub wants to interact with a remote object, it will not need to look up the registry. Instead, it will suffice to know the name of the queue where it wants to send a message. Let us explain these functions specified in the `omq.Broker` class:

- *Broker.bind(oid, remoteObject)*: A call to this method binds a remote object with the identifier *oid*. Once done, the object will be ready to receive RPC requests. If necessary, a queue named *oid* will be created to receive unicast invocations. It will also create a unique private queue to receive multicast requests.
- *Broker.lookup(oid, aClass)*: An invocation to this primitive will generate a Proxy object for the class *aClass*. This Proxy will be used to submit messages to the queue named *oid* and receive responses in the private queue of the client.

Observe that binding more than one object with the same identifier also means that the load from the clients will be evenly distributed among multiple remote objects. This will help us to scale up and down the service by adding and removing remote object instances dynamically. In this case, there is no need to modify client stubs and they do not need to be aware of changes in the pool of remote objects offering a service.

Let us show a simple HelloWorld example:

```
@RemoteInterface
public interface HelloWorld extends Remote {
    @AsyncMethod
    public void helloWorld();

}

Broker broker = new Broker(environment);
helloServer = broker.bind("hello", new HelloServer());

helloClient = broker.lookup("hello");
helloClient.helloWorld();
```

Figure 7: ObjectMQ HelloWorld example.

As shown in the figure, developing remote objects using ObjectMQ is very simple. We omitted here the connection parameters of the Broker object referring to the location of the messaging service.

Since we aim to create a robust but very lightweight middleware, we do not provide shared state or consistency mechanisms between distributed objects. If many servers with the same identifier want to maintain consistent shared state, they should rely on a database or consistent data store. We consider that consistency is not responsibility of our middleware and that other services are ideally suited to this end. We also want to avoid any implicit or transparent state between servers and prefer to bet on a simple stateless model.

2.4.3 Communication Primitives

In the development of our communication abstractions, we decided to treat the local and remote entities separately, following the well known recommendation of Waldo et al. [6]. In ObjectMQ, remote object transparency is not desirable, because developers should be aware of when they are using remote or local entities to program in a way that reflects the indeterminacy and concurrency constraints inherent in the use of remote objects. For this reason, ObjectMQ offers explicit mechanisms using method decorators to define method invocation primitives. In particular, we offer three main invocation abstractions: *asynchronous*, *synchronous*, and *multi-calls*. Let us define the three calls:

- @AsyncMethod: This is an asynchronous non-blocking one-way invocation where the client publishes a message in the target object request Queue ($Q_{Request}$). By default, the client expects to receive no response and it is even not notified if the message was handled correctly.
- @SyncMethod: This is a synchronous blocking remote call where the client publishes a message in the target object request Queue ($Q_{Request}$), blocking until a response is received in its own client response queue ($Q_{Response}$). This call can be configured with a timeout and a number of retries to trigger the exception if the result does not arrive.
- @MultiMethod: This is a one-to-many invocation from one client to many servers. This call can also be combined with @AsyncMethod or @SyncMethod. The former produces a non-blocking multiple invocation to many servers, whereas the latter produces a blocking multiple invocation that collects the results received from many servers in a determined timeout.

On the one hand, asynchronous invocations fit seamlessly with the underlying asynchronous messaging layer. They reduce the burden of handling messages and queues and do not impose additional overhead in the communication. On the other hand, synchronous invocation imply that the proxy will block during a timeout to wait for the result. Synchronous calls in our model must traverse an intermediary (messaging server) that is not needed in direct client-server models like Java RMI. This may impose a small communication overhead since messages must travel through the queues of server and client objects. The benefit is that we delegate communication to the messaging layer, so the server object in ObjectMQ cannot be saturated like in Java RMI, because our server layer only handles the messages the server can process.

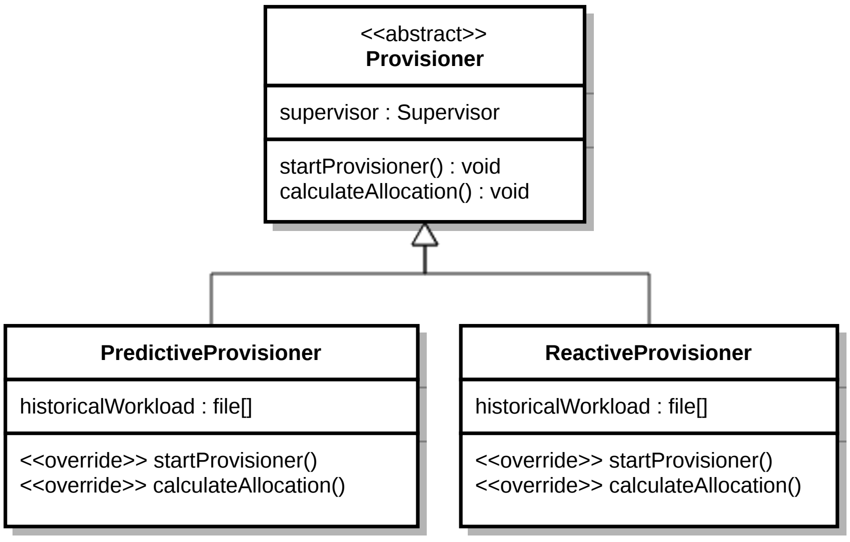

<<abstract>>
Provisioner
supervisor : Supervisor
startProvisioner() : void
calculateAllocation() : void
PredictiveProvisioner
historicalWorkload : file[]
<<override>> startProvisioner()
<<override>> calculateAllocation()
ReactiveProvisioner
historicalWorkload : file[]
<<override>> startProvisioner()
<<override>> calculateAllocation()

Figure 8: Class diagram of the provisioning framework.

We also offer a new one-to-many communication primitive (@MultiMethod). In our case, when many servers listen in the same identifier (queue) they can receive group calls in a Multi queue. Since we rely on the one-to-many communication services of the messaging layer, the system is very efficient and achieves good performance numbers (the ones provided by the messaging layer). This abstraction has proven to be very useful for group communication and it can be combined very easily with the previous abstractions (@AsyncMethod, @SyncMethod).

Finally, an interesting advantage of one-to-many communications is the invocation of methods in a dynamic group of servers that can grow or shrink due to elasticity decisions. Let us explain how our middleware handles programmatic elasticity.

2.4.4 Programmatic Elasticity Framework

We have created an extensible framework that allows third-parties to create their own provisioning policies of server object pools. Our model follows a Master/Slave architecture where the `Supervisor` represents the centralized Master entity that takes care of enforcing `Provisioner`'s policies by launching or removing server objects in `RemoteBroker` Slave servers.

The `Provisioner` interface is the hotspot or extensible hook in our framework that must be inherited to offer a new provisioning policy. For example, in figure 8, we can see the predictive and reactive provisioners that we will explain in the next section. A Provisioner may use information from the HasObjectInfo introspection class to take decisions on server object provisioning. For example, it can observe that messages are not being processed at the adequate speed and ask for another server instance. Or decide that one server is idle and decide to suppress it.

In general, a provisioner will propose a number of server objects required to handle the demand. The `Supervisor` is the responsible entity of enforcing the provisioning policy. It will launch (spawn) or remove (delete) server objects in RemoteBrokers. `RemoteBrokers` are ObjectMQ servers that can launch or shutdown remote object servers. The Supervisor uses multi call abstractions with RemoteBrokers for fault tolerance and introspection information. It periodically ask them about the state of their object servers and it maintains this information updated in HashObjectInfo object for provisioners. If the Supervisor detects that one server failed or that the number of servers is not the one required by the supervisor, it will then spawn or shutdown object servers. Of course, other technologies may be used to spawn or shutdown available servers such as Apache Mesos.

One interesting advantage of our framework is that adhoc policies can be designed depending on the target application. This offers a more fine-grained approach than the traditional coarse grained elasticity offered by cloud providers. We could also use variable like CPU load or memory, but we can also adapt to message processing time in queues offered by our middleware. If we want to enforce a determined processing time per server in an application, we can easily design an adhoc provisioner to this end.

2.4.5 Elastic File Synchronization

Here we show how an application developer can tap into ObjectMQ to enable elastic scaling in file synchronization. Existing commercial Cloud solutions typically fall back on observable resource utilization metrics such as CPU and RAM to drive scaling decisions. For a Personal Cloud system, however, these metrics are ill-suited, as one of the main driving forces behind "live" file synchronization is to guarantee a maximum synchronization time. This requires handling elasticity at the application level by exploiting the knowledge of the application workload, so that it can be utilized a more versatile set of scaling mechanisms.

In what follows, we show how a Personal Cloud system can benefit from the simple programming model of ObjectMQ to achieve elasticity in the file syncing protocol. Particularly, we adopt the model of Urgaonkar et al. [7] for dynamic resource provisioning, though many others could be chosen. The advantage of this model is that it makes use of both a predictive and reactive approach, allowing us to prove the versatility of ObjectMQ. The goal of the predictive method is to allocate resources on large time scales, of the order of days and hours, while the reactive approach is used for shorter time scales, such as seconds and minutes. This allows the system to correct prediction mistakes made by the predictive model, such as unpredictable "flash crowd" patterns. Actually, the predictive method is very useful for online file synchronization. As reported by several independent studies [1, 8, 9] , Personal Cloud systems exhibit strong diurnal and weekly patterns. This allows the predictive provisioning method to allocate servers well ahead of the expected workload peak, and dramatically reduce the odds for clients to experience degraded performance. Indeed, the effectiveness of predictive provisioning will be shown to be very high in our trace-driven experiments with the UB1 cloud-based file syncing service, confirming our intuition that predictive resource provisioning is ideally suited for Personal Cloud systems.

As we set out to enable elasticity for control flows in this work, we assume that the SLA is specified in terms of a suitable high percentile of the response time distribution. We denote this value as d. For instance, a SLA may specify that 95% of the commit requests should incur an end-to-end response time of no more than 5 seconds. As in [7], we assume that all the different instances of the `SyncService` run in homogeneous machines and model each synchronization server as a G/G/1 queuing system, to allow for an arbitrary arrival distribution and arbitrary service times. This enables our elastic scaling scheme to adapt gracefully to changes in the workload intensity caused by time-of-day effects, or even time-of-hour effects. By well-known formulae, the rate δ at which a synchronization server can process commit requests can be simply computed as:

$$\delta \geq \left[s + \frac{\sigma_a^2 + \sigma_b^2}{2(d-s)} \right]^{-1}, \tag{1}$$

where s is the average service time for a commit request, and σ_a^2 and σ_b^2 are the variance of interarrival time and the variance of service time, respectively.

Observe that d is known, while s as well as the variance of interarrival and service time σ_a^2 and σ_b^2 can be monitored online and adjusted correspondingly. By substituting these values into (1), we can obtain a lower bound on the request rate δ that can be serviced by a single server. Once the capacity of a single server is known, the number of required instances η to service a peak request rate of λ can be simply obtained as:

$$\eta = \left\lceil \frac{\lambda}{\delta} \right\rceil. \tag{2}$$

Depending on the value of η and the current number of instances of the `SyncService`, the `Supervisor` will decide to add or remove instances to preserve the performance in response to varying workloads. In practice, the decision of scaling up and down will be performed periodically, once every t time units, to avoid unnecessary oscillations. We will denote by $\lambda_{obs}(t)$ the actual arrival rate seen during the time interval t. Note that the value of $\lambda_{obs}(t)$ can be obtained very easily in our file-sync architecture, since all the commits are queued in a single request queue, as shown in figure. 9.

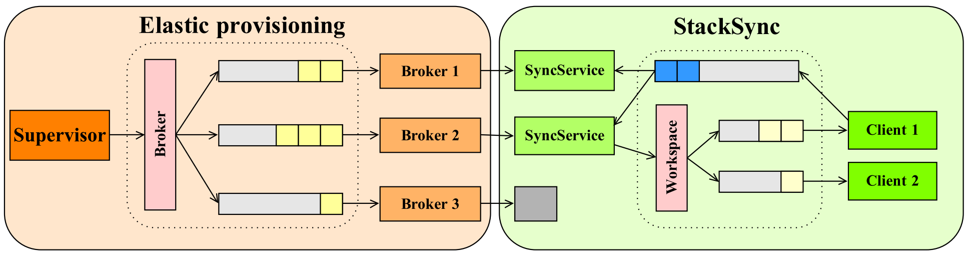


Figure 9: Message broker communication flow.

Predictive Provisioning

This technique uses a workload predictor to anticipate the peak demand over the next time period, and then uses (2) to determine the number of instances that are needed to meet this peak demand. Concretely, the predictor estimates the peak demand that will be seen over the next period of T time units, at the beginning of each period. To do so, it maintains a history of the observed arrival rate for each time period t of duration T over the past several days. From the history, the predictor then derives the probability distribution of the arrival rate for that time period. The peak workload $\lambda_{pred}(t)$ for a particular period t is finally estimated as a high percentile of the arrival distribution for that period.

Reactive Provisioning

Even in the case that predictive provisioning was perfect, sudden spikes or “flash crowds" are unpredictable phenomena. To react to unforeseen events, reactive provisioning acts on shorter time scale to handle short term fluctuations. Basically, it compares the current observed arrival rate $\lambda_{obs}(t)$ over the past few minutes to the predicted rate $\lambda_{pred}(t)$. Specifically if $\frac{\lambda_{obs}(t)}{\lambda_{pred}(t)} > \tau_1$ or drop rate τ_2, then corrective action is necessary. In this case, it recomputes the number of instances by invoking (2).

2.4.6 Conclussions

A core contribution of our architecture is to rely on a lightweight communication framework for providing programmatic elasticity to distributed objects using message queues as their underlying communication middleware. StackSync implements predictive and reactive provisioning policies on top of ObjectMQ that adapt to real traces from the Ubuntu One service. Also, the ObjectMQ unicast and multicast communication primitives have considerably simplified the code of the synchronization protocol. It also enables efficient change notification in a transparent way on top of the underlying messaging service.

StackSync provides a reference implementation and useful tools for rapid prototyping and evaluation. It has been extensively tested using real traces from the Ubuntu One system to validate its elasticity and efficient use of resources. Furthermore, extending an open benchmark [4] of Personal Clouds, we obtained good results comparing our service with Dropbox, Box, and OneDrive. StackSync is a stable open source project after two years of development that is being used in several public institutions and data centers.

Finally, an interesting open question is if our model and invocation abstractions can be generalized for offering programmatic elasticity to cloud applications. Since major cloud providers already offer scalable messaging services, it could be possible for them to offer equivalent programmable middleware and abstractions on top of their infrastructures. Messaging services could then become part of the existing load balancing fabric in the data center.

3 NEC

3.1 Introduction

This section intends to be a detailed guide about the interoperability protocol of CloudSpaces in NEC's personal cloud called Cloud Storage. It also describes the interoperability process through screenshots, examples and all necessary calls with its parameters for its proper operation.

3.2 Interoperability

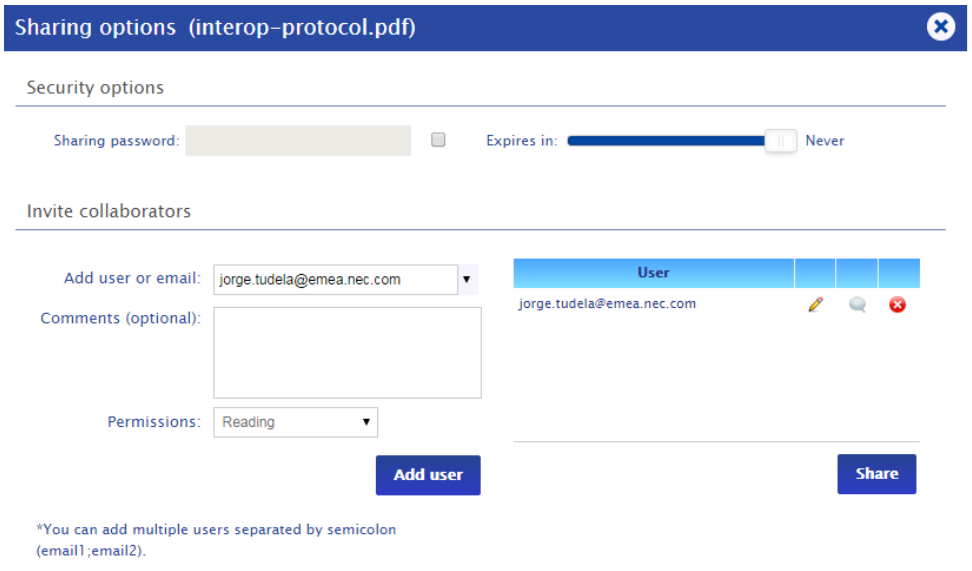
The interoperability protocol used by NEC Cloud Storage is divided into three steps explained below.

- User invitation
- Invitation acceptance
- Access credentials

3.2.1 User invitation

User sends an invitation

A user of NEC Cloud Storage wants to share a file or folder with another user. Therefore, the user logs into its NEC Cloud Storage account, selects the file or folder he/she wants to share, and enters the email of the user.


Sharing options (interop-protocol.pdf)

Security options

Sharing password:

Expires in: Never

Invite collaborators

Add user or email: jorge.tudela@emea.nec.com

Comments (optional):

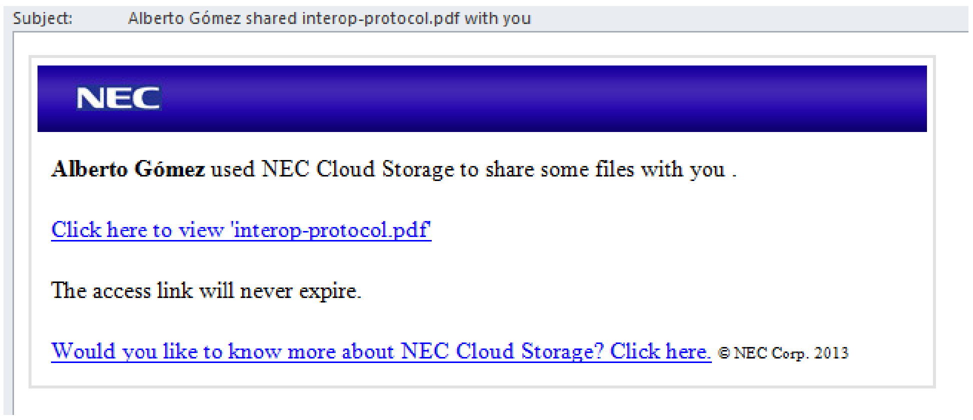
Permissions: Reading

User			
jorge.tudela@emea.nec.com			

*You can add multiple users separated by semicolon (email1;email2).

Figure 10: File sharing options

The invitee will receive an email indicating the intention of the user from NEC Cloud Storage to share a file or folder with him/her and a link to a website located in NEC Cloud Storage.



Subject: Alberto Gómez shared interop-protocol.pdf with you

Alberto Gómez used NEC Cloud Storage to share some files with you .

Click here to view 'interop-protocol.pdf'

The access link will never expire.

Would you like to know more about NEC Cloud Storage? Click here. © NEC Corp. 2013

Figure 11: Example of a mail with the sharing proposal

The recipient selects its Personal Cloud

The user who has received the email clicks on the link and is redirected to a NEC Cloud Storage page (as the sharing proposal was originated by NEC Cloud Storage the link points to this site), where the user is asked to select a Personal Cloud from a list of services that have an agreement with NEC Cloud Storage. In this case, as an example, the user will select StackSync, because it already has an account on this Cloud.

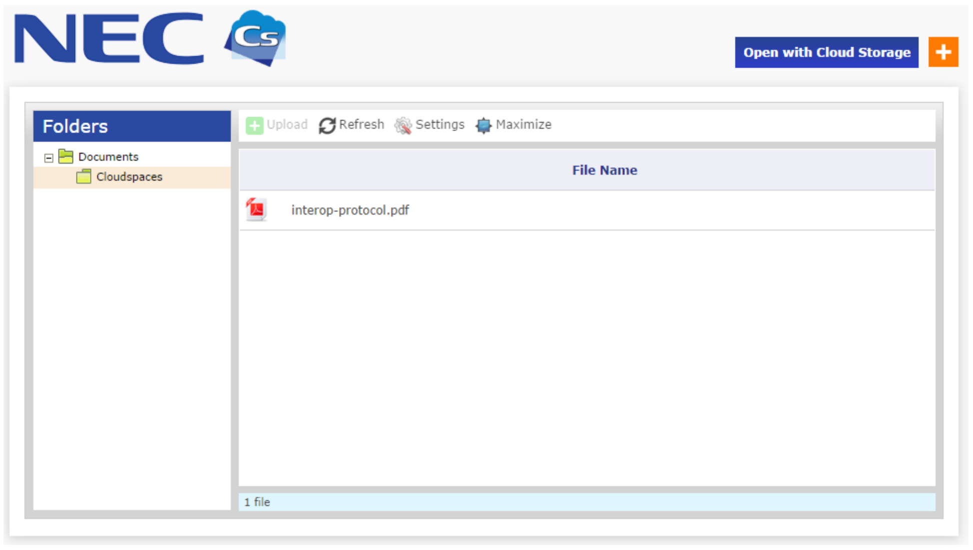


Figure 12: File list screen

In the image above, the user can see the file that someone has shared with him/her. On the right top of the image, the user will be able to choose the Personal Cloud wished.

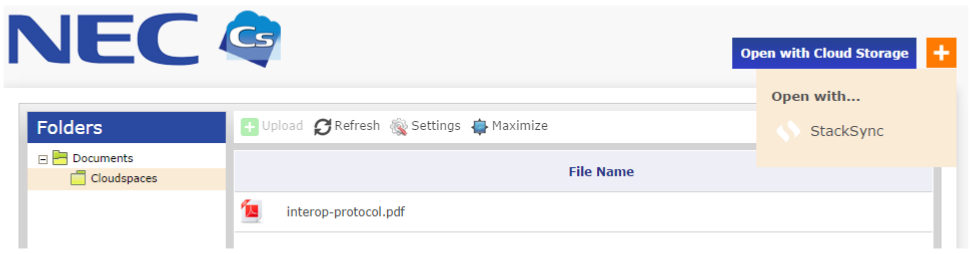


Figure 13: Choosing the preferred personal cloud

Creating the interoperability proposal

At this time, NEC Cloud Storage creates the interoperability proposal by sending an HTTP POST request to StackSync's share URL (in this example case, another instance of NEC Cloud Storage) previously established and that we have saved.

```
POST /AdvancedFeatures/ExternalSharingProposal.aspx
```

Parameter	Type	Description
`share_id`	string	A unique value that identifies the interoperability proposal. This is autogenerated with a GUID format.
`resource_url`	string	An absolute URL to access the shared resource located in NEC Cloud Storage.
`owner_name`	string	The name corresponding to the owner of the folder.
`owner_email`	string	The email corresponding to the owner of the folder.
`resource_name`	string	The name of the folder or file.
`permission`	string	Permissions granted to the recipient. Options are read-only and read-write.
`recipient`	string	The email corresponding to the user who the folder has been shared with.
`callback`	string	An absolute URL to which the destination Private Cloud (e.g. StackSync) will redirect the user back when the invitation step is completed.
`protocol_version`	string	MUST be set to 1.0. Services MUST assume the protocol version to be 1.0 if this parameter is not present.

3.2.2 Invitation acceptance

The user accepts the invitation

StackSync (in this case NEC Cloud Storage) displays the details of the file or folder share invitation to the recipient and asks for its credentials and explicit permission to accept the invitation.

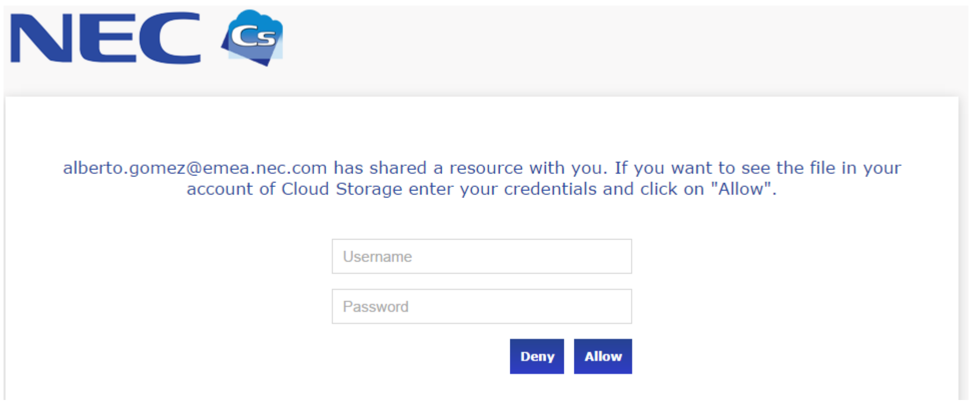

alberto.gomez@emea.nec.com has shared a resource with you. If you want to see the file in your account of Cloud Storage enter your credentials and click on "Allow".

Figure 14: Providing user credentials and accepting/denying the sharing proposal

Returning the proposal response

Once StackSync (2nd instance of NEC Cloud Storage) has obtained approval or denial from user, StackSync will use the callback obtained from request above to inform NEC Cloud Storage about the user decision. Then, StackSync uses the callback to construct an HTTP GET request, and directs the User's web browser to that URL.

```
GET /AdvancedFeatures/ExternalSharingProposal.aspx
```

Parameter	**Type**	**Description**
`share_id`	string	A unique value that identifies the interoperability proposal.
`accepted`	string	A string indicating whether the invitation has been accepted or denied. `true` and `false` are the only possible values.

3.2.3 Access credentials

Granting access to the service

When NEC Cloud Storage receives the proposal result, firstly it validates these data and then generates the access credentials. Finally, it sends an HTTP POST request to StackSync where the resource will be displayed.

NEC Cloud Storage specifies what type of authentication protocol and version must be used to access the resource. In this case, the authentication protocol used is OAuth 1.0a and, at this point, NEC Cloud Storage web module initiates a standard OAuth process to generate the required token that will be sent to the second Personal Cloud. For this purpose a specific module for processing OAuth authentication mechanism has been included which provides both consumer and provider implementations.

To this end, the second Personal Cloud must check the `auth_protocol` and `auth_protocol_version` parameters, and also the specific parameters that will be included in the request. This information will be used when user will try to access to the shared resources.

```
POST /Files.aspx
```

Parameter	Type	Description
`share_id`	string	A unique value that identifies the interoperability proposal.
`auth_protocol`	string	The authentication protocol used to access the shared resource (e.g.oauth).
`auth_protocol_version`	string	The version of the authentication protocol (e.g. 1.0a).

Other authentication-specific parameters are sent together with the above parameters, these parameters may include values like tokens, timestamps or signatures which will be necessary to get and perform the actions with the resource depending on the kind of authentication protocol used.

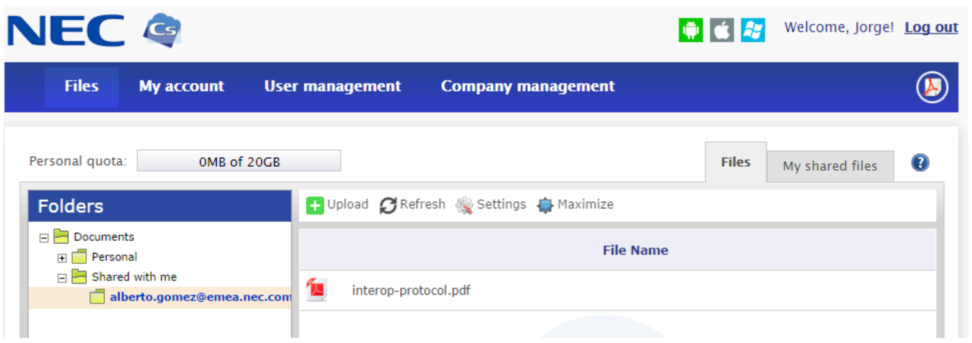


Figure 15: Shared folder appears on user's file list

In the image above, Stacksync (2nd instance of NEC Cloud Storage) shows the shared resource. To be able to access to the resource is necessary to use the tokens sent in last step.

3.3 Sequence diagram

Here we can see a complete workflow of the interoperability protocol where a NEC Cloud Storage user (User A) shares a file or folder with an external Personal Cloud user (User B).

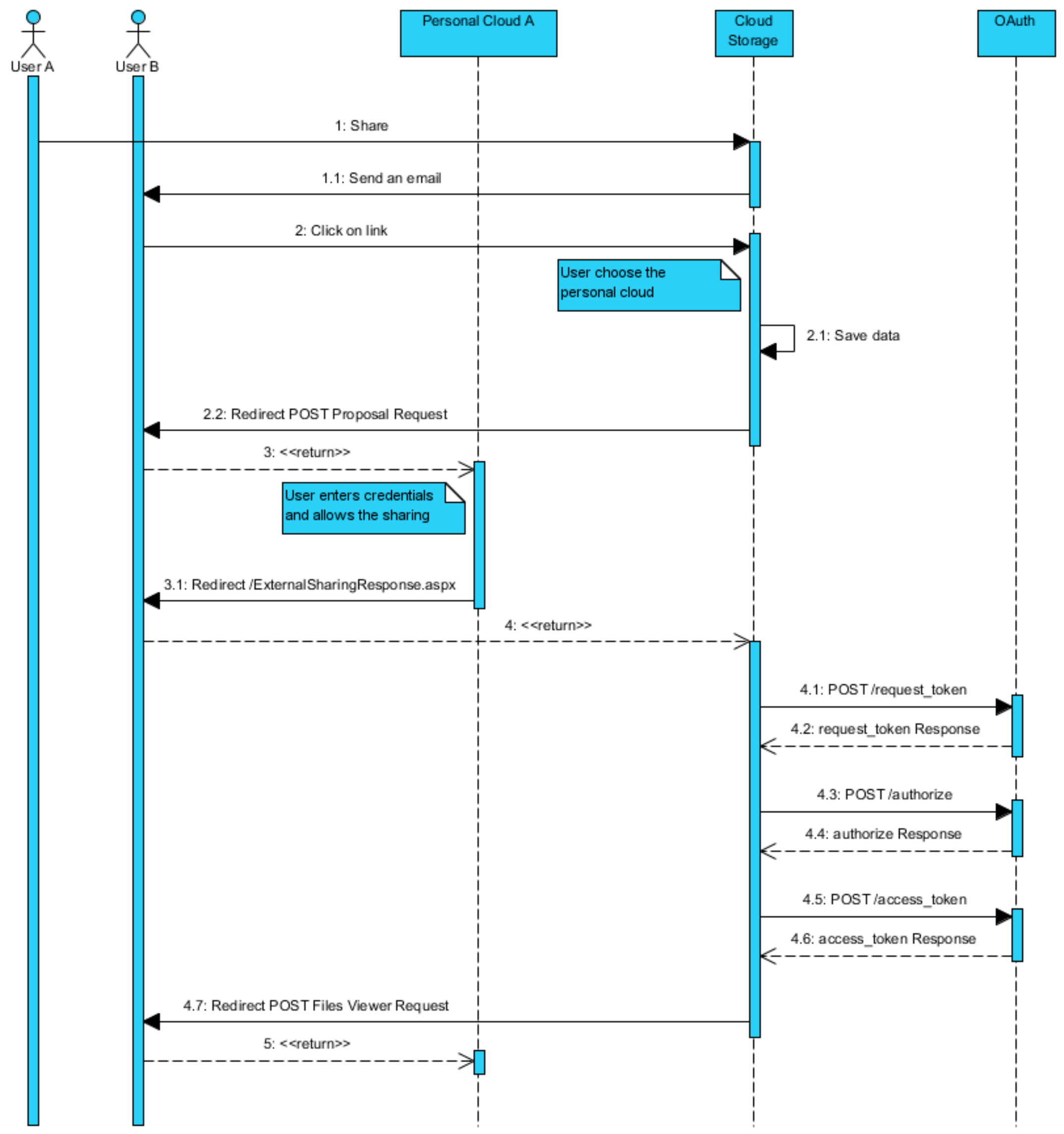

User A
User B
Personal Cloud A
Cloud Storage
OAuth
1: Share
1.1: Send an email
2: Click on link
User choose the personal cloud
2.1: Save data
2.2: Redirect POST Proposal Request
3: <<return>>
User enters credentials and allows the sharing
3.1: Redirect /ExternalSharingResponse.aspx
4: <<return>>
4.1: POST /request_token
4.2: request_token Response
4.3: POST /authorize
4.4: authorize Response
4.5: POST /access_token
4.6: access_token Response
4.7: Redirect POST Files Viewer Request
5: <<return>>

Figure 16: NEC user sharing a folder with an external personal cloud user

Also, we can see an example where an external Personal Cloud user (User A) shares a resource with a NEC Cloud Storage user (User B).

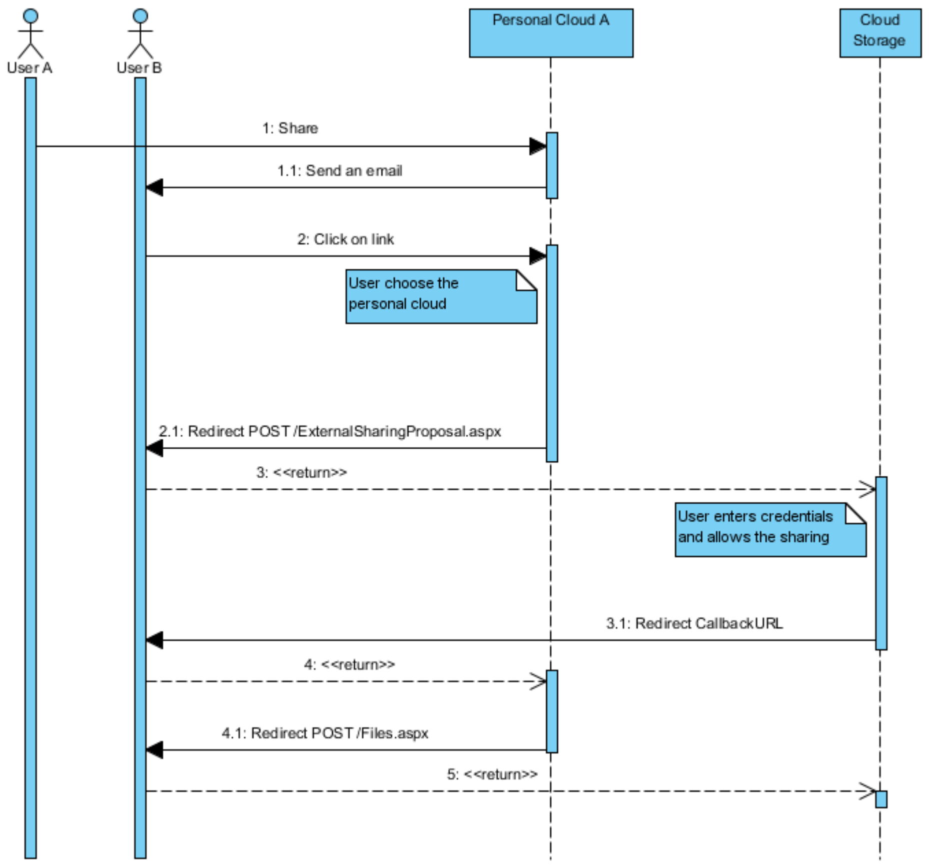

User A
User B
Personal Cloud A
Cloud Storage
1: Share
1.1: Send an email
2: Click on link
User choose the personal cloud
2.1: Redirect POST /ExternalSharingProposal.aspx
3: <<return>>
User enters credentials and allows the sharing
3.1: Redirect CallbackURL
4: <<return>>
4.1: Redirect POST /Files.aspx
5: <<return>>

Figure 17: An external user sharing a folder with a NEC user

4 EyeOS

4.1 Introduction

eyeOS is a web platform that provides a remote virtual desktop for the end user.

The overall user experience is strongly influenced by the classic desktop design, widely known thanks to the most popular operating system on the market. eyeOS Personal Web Desktop includes several features such as: file manager, contacts, groups and other collaborative capabilities. eyeOS Personal Web Desktop is a disruptive technology that fits in perfectly with the CloudSpaces Open Personal Cloud paradigm.

One of the key values that eyeOS provides is the possibility to work directly with files in the cloud. eyeOS does not require users to manually download any files onto their computer nor is it necessary to install anything locally, so the experience is totally transparent: users just log into a website and start working with their files normally.

Furthermore, eyeOS lets you add additional services and applications within the web desktop, so that all the company or organization's web resources are available within a single controlled environment that can be accessed using single sign-on.

By combining eyeOS' web file management capabilities with Personal Cloud, users can access their Personal Cloud contents via web, with a user experience very similar to local desktop environments.

4.2 Authentication

The eyeOS platform uses OAuth authentication in order to interact with the user's protected data stored in Personal Cloud. OAuth is an authorization protocol that enables the user (resource owner) to authorize eyeOS to access the resources on their behalf without giving eyeOS their authentication credentials i.e. user name and password.

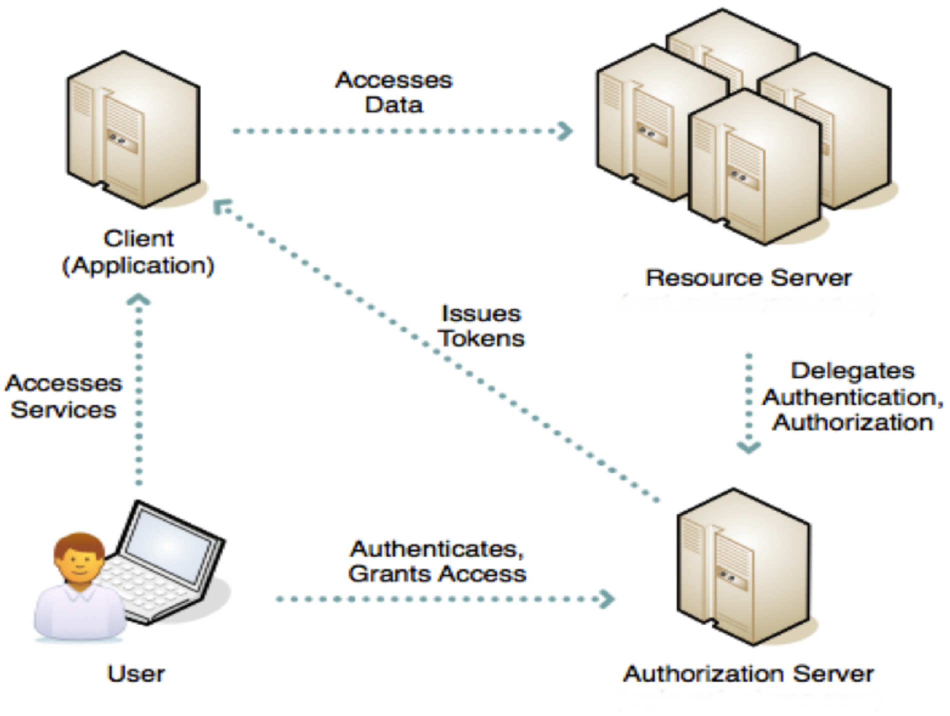


Figure 18: eyeOS authentication

When the eyeOS user accesses the file manager for the first time, a newly developed plugin is used to get a security token with which the keys required for interacting with user data stored in Personal Cloud can be obtained. The Access Token and Token Secret keys are stored in the 'token' table of the relational database management system (RDBMS) based on MySQL. These keys are linked with the user who has logged into the platform, meaning the system can determine the access token for a given user who attempts to use the service at any stage.

The communication flow is as follows:

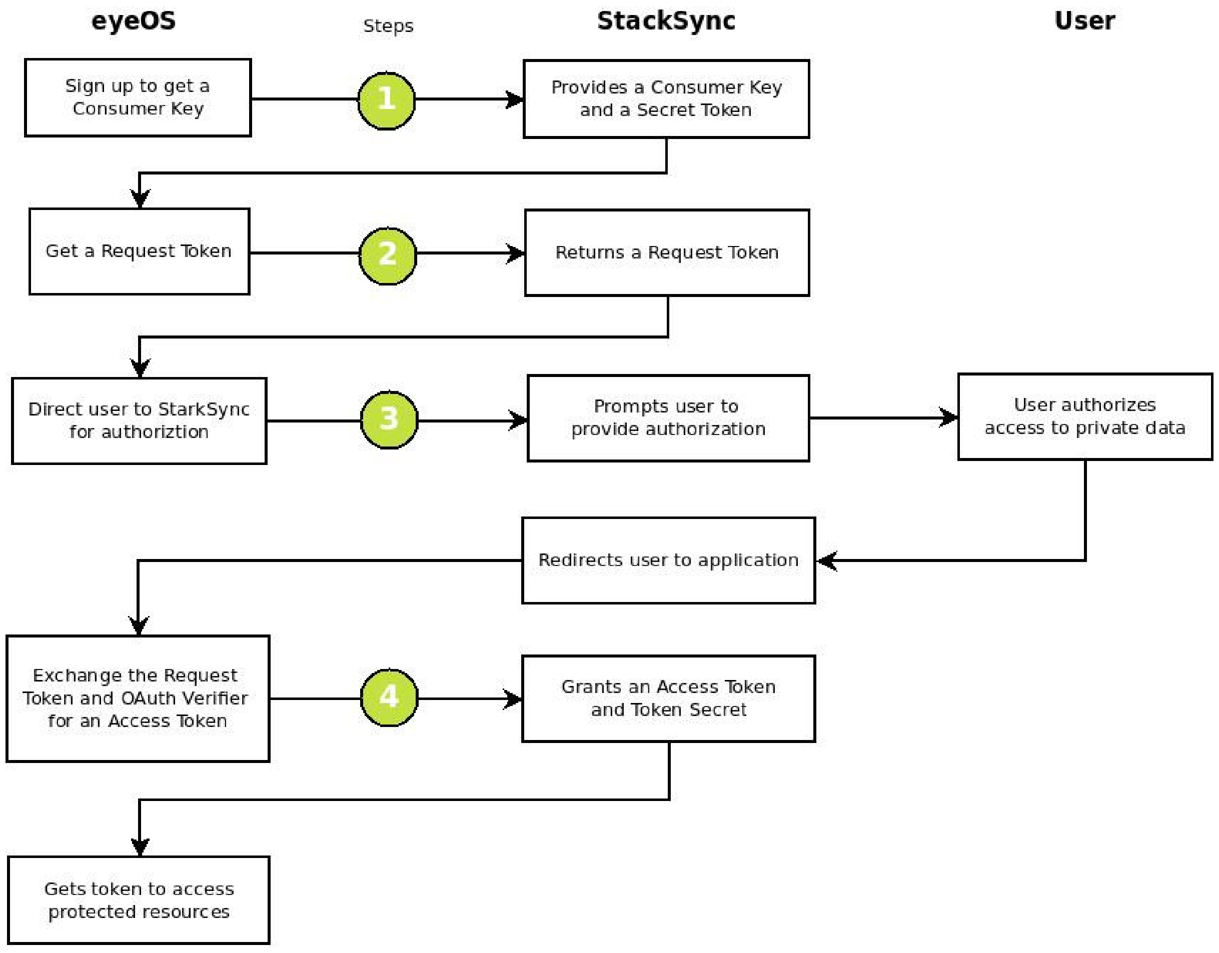

eyeOS
Steps
StackSync
User
Sign up to get a Consumer Key
1
Provides a Consumer Key and a Secret Token
Get a Request Token
2
Returns a Request Token
Direct user to StarkSync for authoriztion
3
Prompts user to provide authorization
User authorizes access to private data
Redirects user to application
Exchange the Request Token and OAuth Verifier for an Access Token
4
Grants an Access Token and Token Secret
Gets token to access protected resources

Figure 19: eyeOS OAuth flow

Step 1:

Request from StackSync the consumer key and secret token that identifies eyeOS as the CloudSpaces resource consumer.This communication is done via email.

Step 2:

Get the request token and provide StackSync with the redirect URL to eyeOS once the user grants authorization. StackSync responds to the previous request by giving a valid request token and an authorization URL.

Step 3:

Redirect the user to the authorization URL where the user grants eyeOS access to their private space. Once StackSync verifies the user, it redirects the user to the eyeOS URL provided in the previous step.

Step 4:

Get the access token and token secret from StackSync, with which eyeOS will identify itself when accessing the user's private space in CloudSpaces.

Authentication is implemented in eyeOS according to the diagram below:

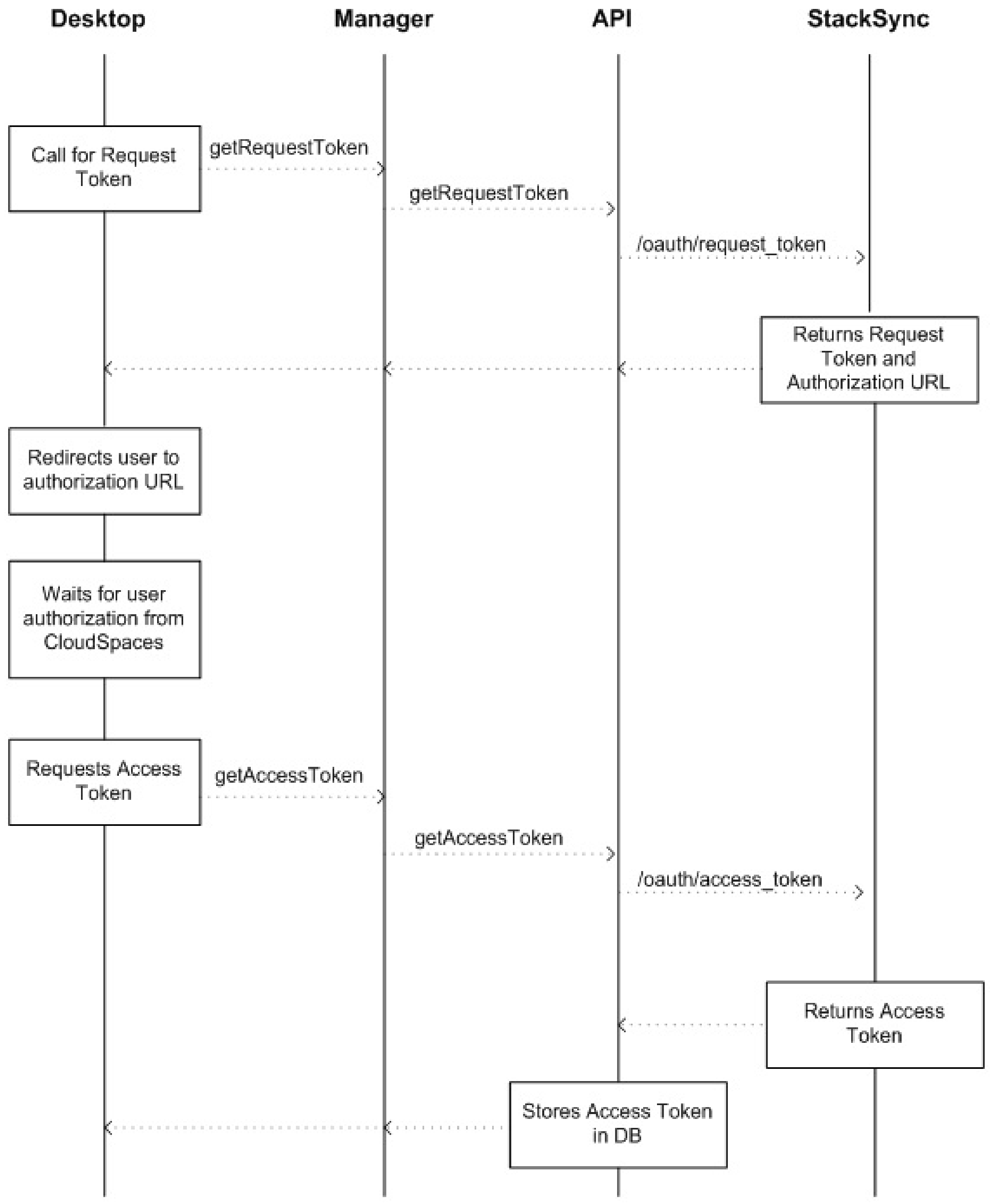


Figure 20: eyeOS authentication implementation

The OAuth Manager and OAuth API functions are given in the GitHub repository [2].

If the user has not previously granted eyeOS access to their CloudSpaces private space, when they access the file manager, the authentication process is initiated, as shown in the following screens:

[2]https://github.com/cloudspaces/eyeos-u1db#implementation-of-stacksync-api-into-eyeos

- The user is asked if they want to grant eyeOS access to their protected data.

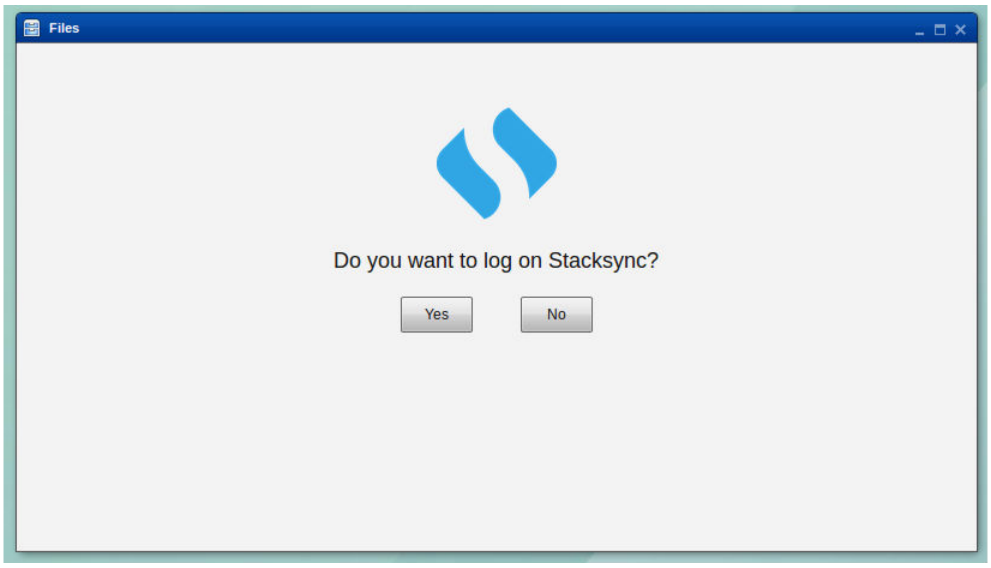


Figure 21: StackSync connect screen

- If the user selects "No", the eyeOS file structure is shown without the StackSync directory, which contains the user's protected data in CloudSpaces.

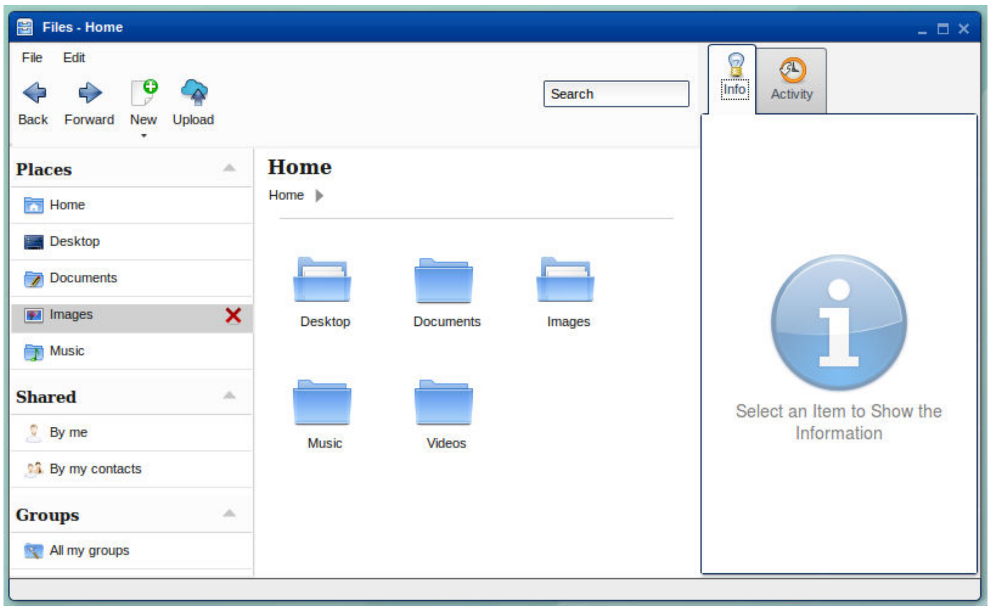


Figure 22: eyeOS file browser

- If the user selects "Yes" in the first screen of the process, communication is established with StackSync to get the access token.

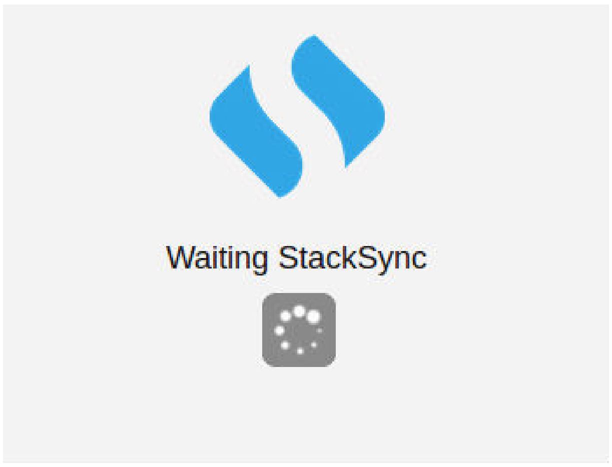
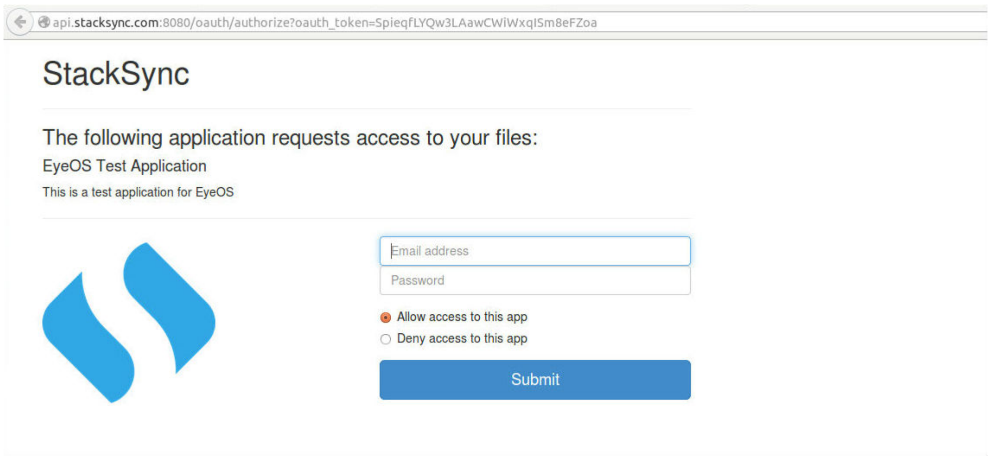


Figure 23: StackSync waiting screen

- A new browser window is pops up in which the user is redirected to the authorization URL received from StackSync. Here the user authorizes eyeOS to access their private space.



api.stacksync.com:8080/oauth/authorize?oauth_token=SpieqfLYQw3LAawCWiWxqISm8eFZoa

StackSync

The following application requests access to your files:

EyeOS Test Application

This is a test application for EyeOS

Email address

Password

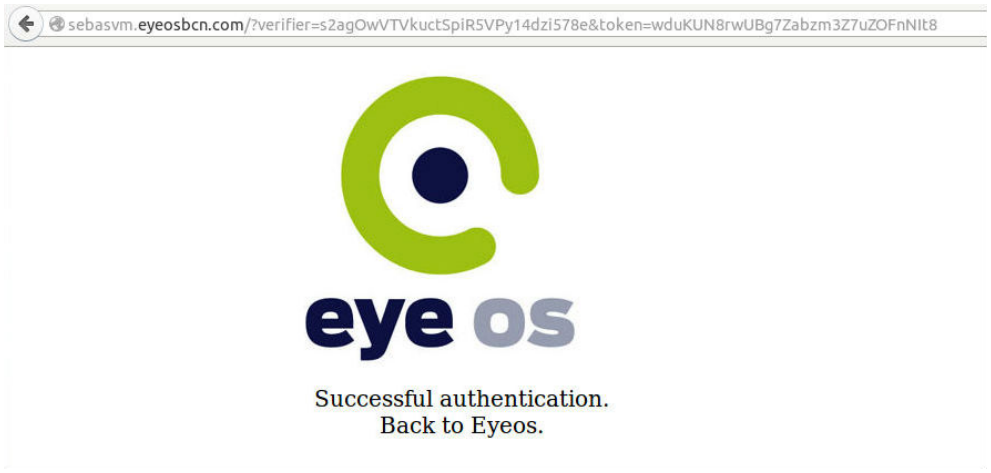
[✓ Allow access to this app]

Deny access to this app

Submit

Figure 24: StackSync login website

- Once access has been authorized, StackSync redirects the user to the URL provided by eyeOS on requesting the request token. This page notifies the user that the process has been completed successfully and that they can return to the eyeOS desktop.



sebasvm.eyeosbcn.com/?verifier=s2agOwVTVkuctSpiR5VPy14dzi578e&token=wduKUN8rwUBg7Zabzm3Z7uZOFnNIt8

Successful authentication.
Back to Eyeos.

Figure 25: eyeOS authentication callback

- The access token for the current eyeOS user is saved. From this moment, the user can access their protected data from the StackSync directory without having to authenticate again.

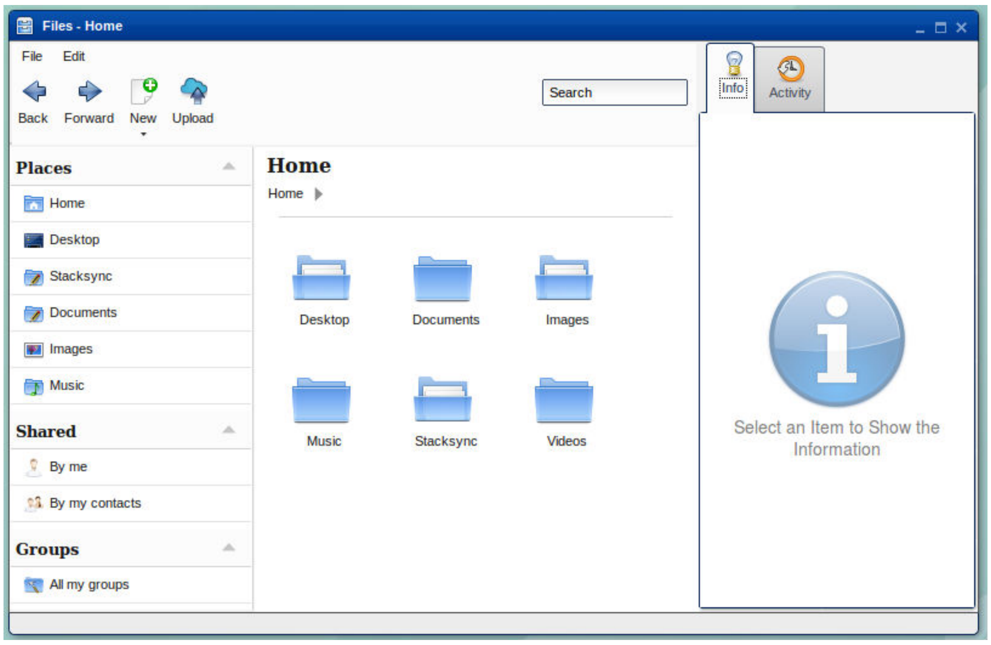
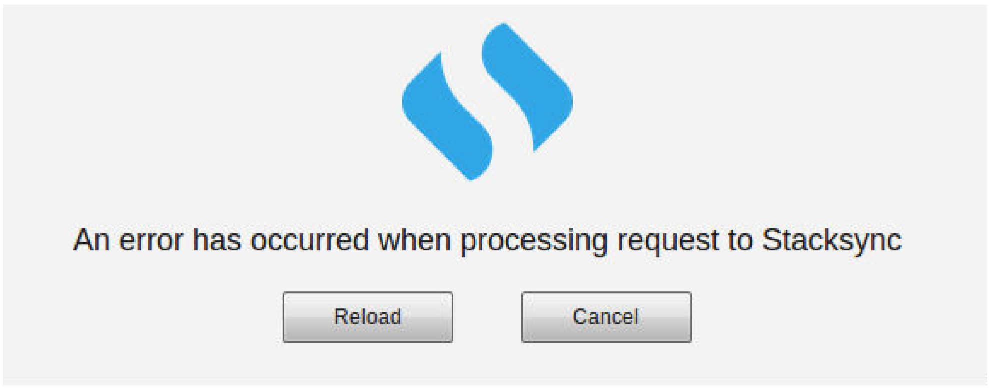


Figure 26: eyeOS file browser with StackSync folder

During the authentication process, various exceptions may be triggered, which interrupt the process to get the access token. These errors are described below:

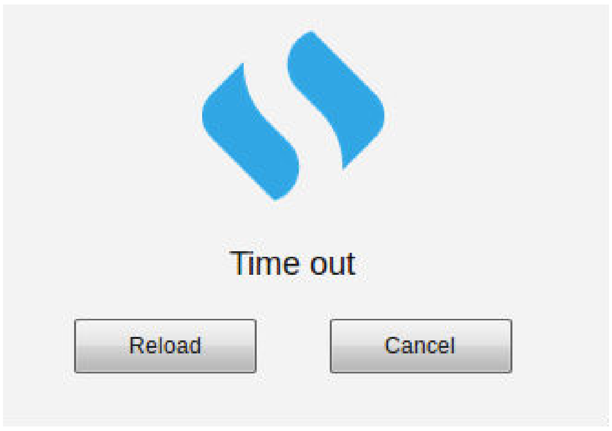
- Communication error. This may occur on sending or receiving data from StackSync.



An error has occurred when processing request to Stacksync

Figure 27: StackSync communication error

- Timeout exceeded for receiving the user's authorization to access their private space. eyeOS establishes a timeout of 1 minute. The process to request the request token can be restarted or interrupted.



Time out

Figure 28: StackSync timeout error

- Access denied as an invalid access token was sent to StackSync. The access token does not expire and is stored permanently in eyeOS. This error occurs when the users deletes the access token from the StackSync portal.

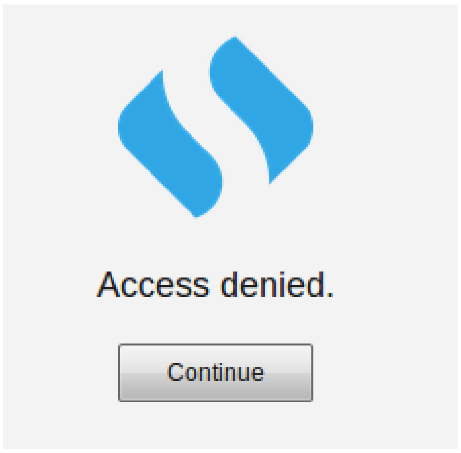


Figure 29: StackSync access denied error

4.3 Integrated File Management

Within the eyeOS Personal Web Desktop, one of the key features of the platform is file management. eyeOS includes a web file manager developed in JavaScript, HTML and CSS that enables users to manage all their files directly from their browser, but with an experience similar to a file manager of any desktop operating system, such as Microsoft Windows™ or GNU/Linux.

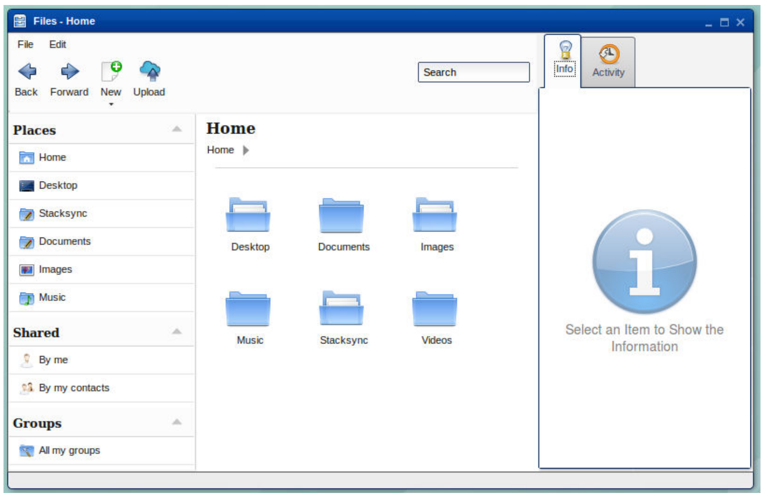


Figure 30: eyeOS file browser

By integrating Personal Cloud services in the eyeOS platform, users can use eyeOS' web file manager and all its features with their Personal Cloud files. For example, users can display online their documents saved in their Personal Cloud, create directories, move files, share documents, etc.

Users access the files in their Personal Cloud using the StackSync directory. To get the file and directory structure of Personal Cloud, a call is made to StackSync's API. This call returns metadata with all the structural information of the files and directories, which eyeOS uses to generate a local replica.

The files and directories are created without content. When the user selects an element and performs an operation on it, i.e. opens, moves or copies it, the element's contents are downloaded. By doing this, the system is not overloaded unnecessarily by retrieving information that the user will not use at that moment. If the content of a file or directory has already been retrieved and there are no changes, it will not be updated.

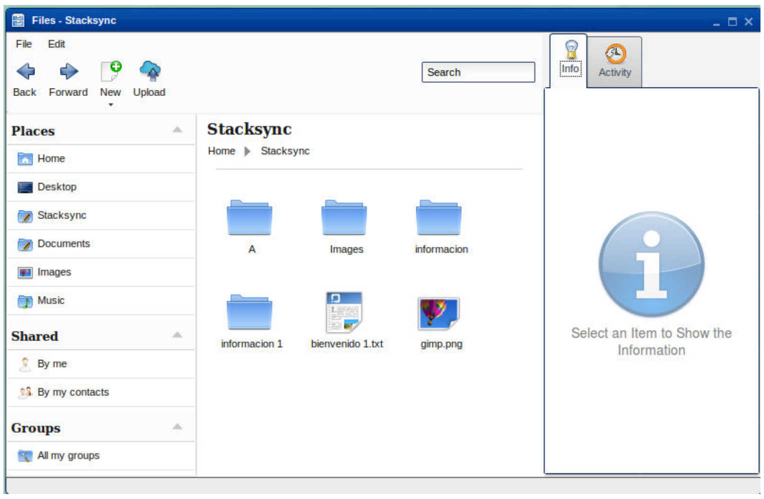
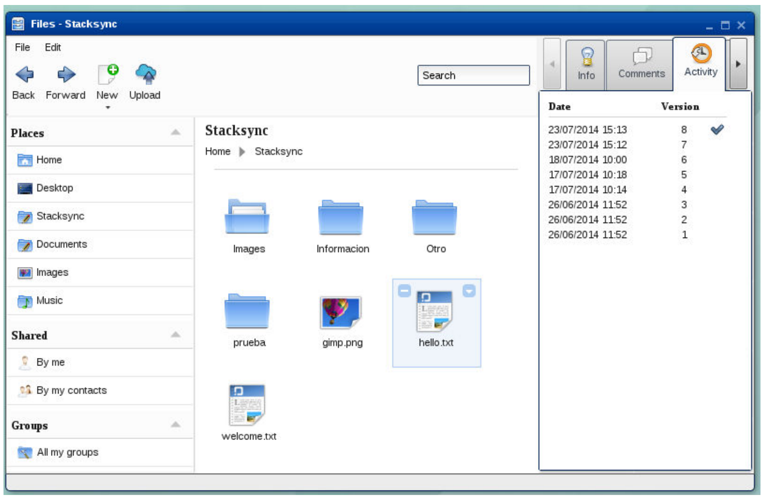


Figure 31: eyeOS StackSync folder

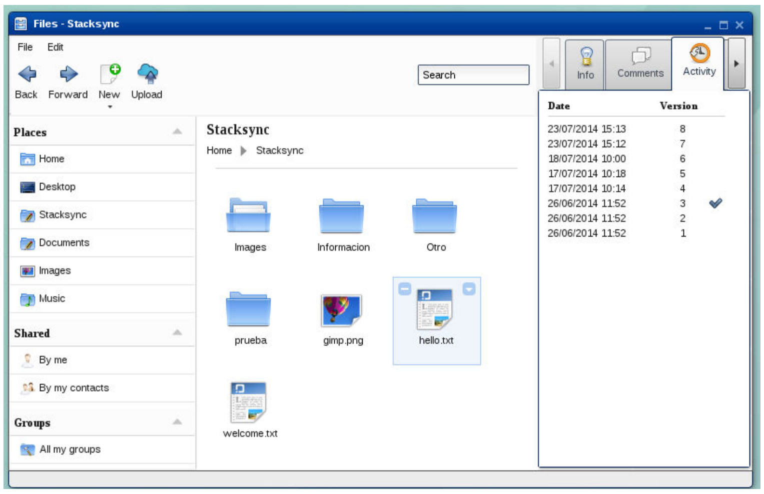
The file manager can retrieve previous versions of a file. It shows a list of all the available versions of the file, letting the user retrieve the contents of the desired version.



Date	Version	
23/07/2014 15:13	8	✓
23/07/2014 15:12	7	
18/07/2014 10:00	6	
17/07/2014 10:18	5	
17/07/2014 10:14	4	
26/06/2014 11:52	3	
26/06/2014 11:52	2	
26/06/2014 11:52	1	

Figure 32: eyeOS file history

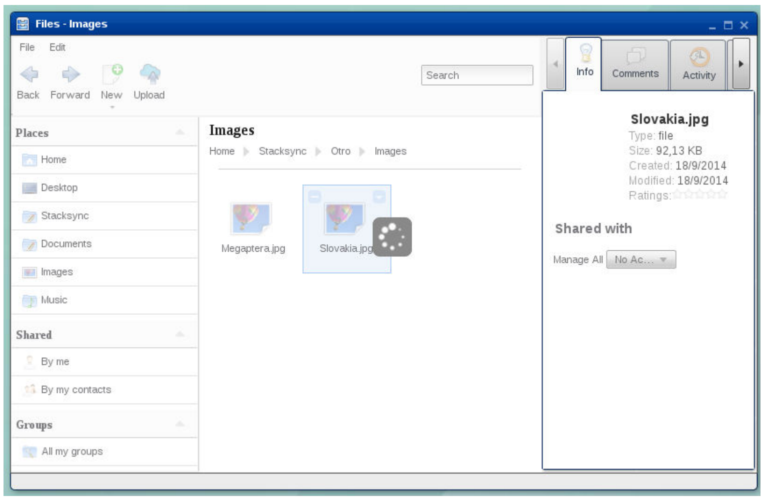
If the user makes changes to a previous version, when those changes are saved, a new version is created in Personal Cloud.



Date	Version
23/07/2014 15:13	8
23/07/2014 15:12	7
18/07/2014 10:00	6
17/07/2014 10:18	5
17/07/2014 10:14	4
26/06/2014 11:52	3 ✓
26/06/2014 11:52	2
26/06/2014 11:52	1

Figure 33: eyeOS retrieve older version

The contents of the current directory are synced with the Personal Cloud directory in a background process, which sends queries every 10 seconds to check whether there are any changes. If there are any changes, the current structure is updated.



Slovakia.jpg
Type: file
Size: 92,13 KB
Created: 18/9/2014
Modified: 18/9/2014
Ratings:

Figure 34: eyeOS folder updates

4.4 Storage

To integrate the Personal Cloud storage service within eyeOS, the Storage Manager and the Storage API have been created. The Storage API communicates with the StackSync API v2, which manages all the requests made from the file manager.

Storage is implemented in eyeOS according to the diagram below:

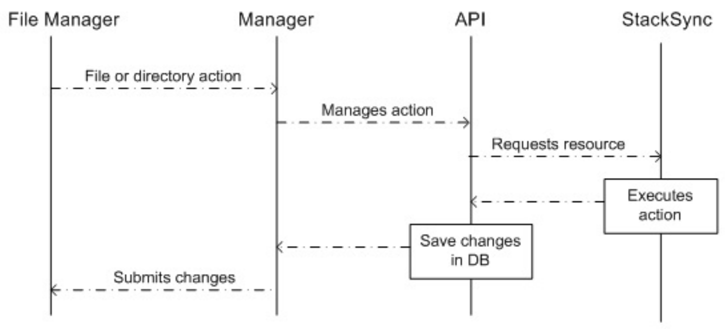


Figure 35: eyeOS storage sequence diagram

The user performs an action in the file manager, such as opening a directory. The Storage Manager using the getMetadata function retrieves the user's access token and id for the directory in StackSync. These values are sent to the API, which is responsible for requesting the resource from StackSync using the getMetadata function. The file structure of the directory is retrieved and stored in the database, and the Manager is notified of the new structure, which will update the eyeOS interface.

The getMetadata functions of the Storage Manager and the Storage API, as well as the other actions performed by the file manager,are described in more detail in the GitHub repository [3].

[3]https://github.com/cloudspaces/eyeos-u1db#implementation-of-stacksync-api-into-eyeos

4.5 Persistence

eyeOS Calendar and the comments tool of the file manager store data in a U1DB database.

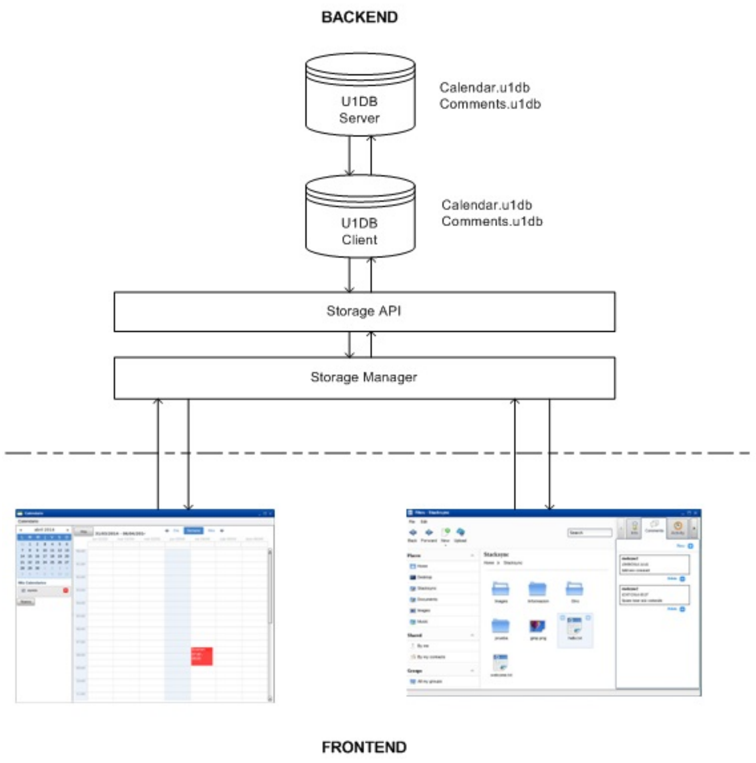


Figure 36: eyeOS persistence architecture

U1DB is a database API to sync JSON documents that was created by Canonical. It lets applications store documents and synchronize them between machines and devices. U1DB is designed to work anywhere, acting as a storage backup for native platform data. This means it can be implemented on any platform, from different languages, providing backup and sync services between platforms.

The U1DB API contains three sections: document storage/retrieval, querying and synchronization. A short description of their operation is given below.

Document storage/retrieval

U1DB stores documents, basically any information that can be expressed in JSON.

Document storage and retrieval functions are described in more detail in the GitHub repository [4].

Querying

Querying in U1DB is done using indexes. To retrieve certain documents from the database according to specific criteria, first you must create an index and then query this index.

[4]https://github.com/cloudspaces/eyeos-u1db#storagerecovering-documents

Querying functions are described in more detail in the GitHub repository [5].

Synchronization

U1DB is a syncable database. Any U1DB database can be synced with a U1DB server. Most U1DB installations can be run as server.

Syncing the server and the client updates both sides so that they contain the same data. Data is saved in local U1DBs, whether online-offline, and then synchronized when online.

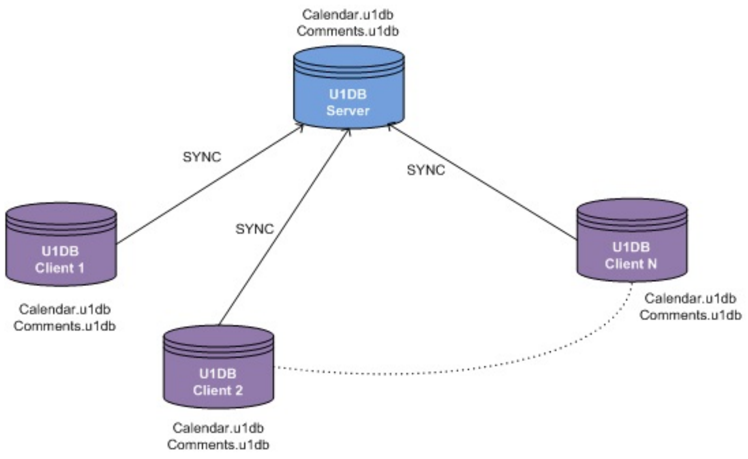


Figure 37: eyeOS persistence communication

Synchronization functions are described in more detail in the GitHub repository [6].

4.5.1 Implementing OAuth

To sync eyeOS Calendar and the comments tool applications, authentication must be done with the server using the OAuth protocol.

The OAuth server provides access to a unique protected resource, the U1DB server. The eyeOS platform implements the Credentials.py script, which contains the APIs required for communicating with the OAuth server.

The communication dialog is as follows:

[5]https://github.com/cloudspaces/eyeos-u1db#queries
[6]https://github.com/cloudspaces/eyeos-u1db#synchronization

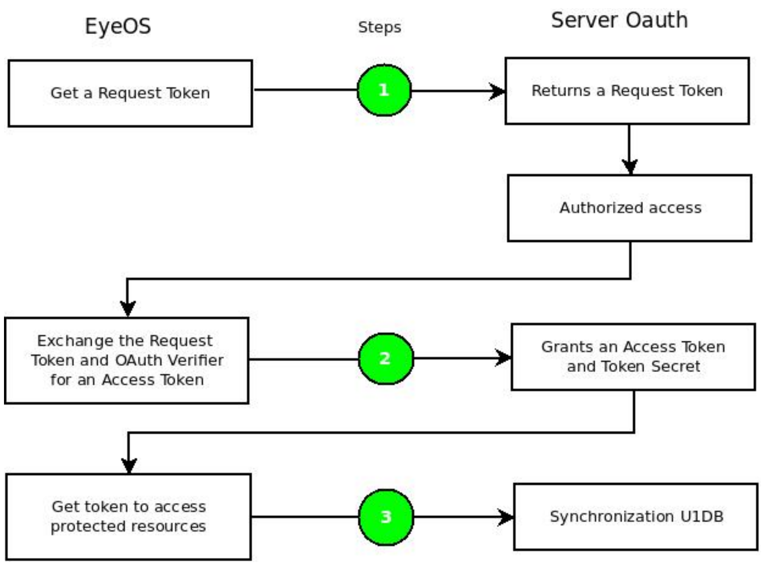

EyeOS
Steps
Server Oauth
Get a Request Token
1
Returns a Request Token
Authorized access
Exchange the Request Token and OAuth Verifier for an Access Token
2
Grants an Access Token and Token Secret
Get token to access protected resources
3
Synchronization U1DB

Figure 38: eyeOS persistence OAuth flow

Step 1:

API getRequestToken() gets the consumer key and consumer secret from the settings. It then makes a call to the OAuth server using the "request_token" URL.

The OAuth server retrieves the consumer key from the database and compares it with the consumer key received. If the keys do not match, it returns an error; otherwise, it performs a new search in the database using the consumer key to get the request token.

The OAuth server responds to the eyeOS call, returning the request token and access verifier to request the access token.

The eyeOS platform stores the request token and the verifier in the session variables so that it does not have to repeat the process in subsequent steps.

Step 2:

API getAccesToken(token,verifier) makes a call to the OAuth server through the “access_token” URL.

The OAuth server retrieves the consumer key and request token from the database and compares them with those received from getAccessToken(). If they do not match, it returns an error; otherwise, it performs a new search in the database with the consumer key and request token to get the access token. If no data is obtained or the data obtained has expired, a new access token must be generated and stored in the database, which means that previous tokens will no longer have access.

The OAuth server responds to the eyeOS call, returning the access token.

Step 3:

API protocol(params) calls the OAuth server via the U1DB synchronization API.

The OAuth server retrieves the access token from the database using the consumer and token received. If they do not match, it returns an error; otherwise, it syncs with the U1DB server.

Step 1 only applies where a request token has not been requested during the eyeOS user session.

Authentication implemented in both applications is requested for each resource request. The request process flow is illustrated in the following diagram:

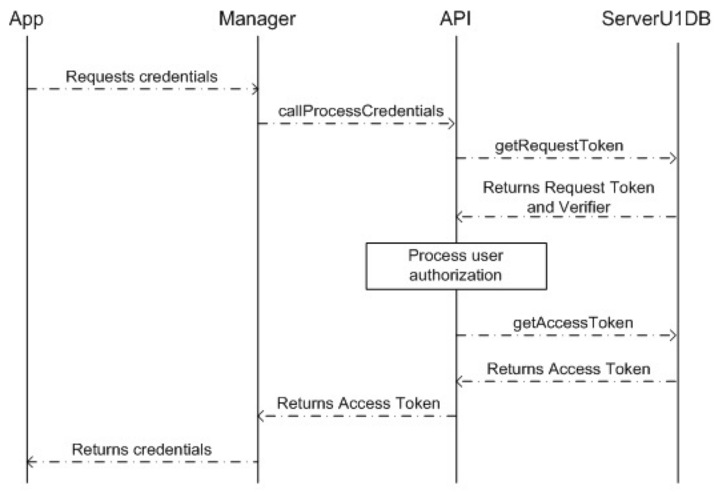


Figure 39: eyeOS persistence sequence diagram

OAuth Manager and OAuth API persistence functions are given in the GitHub repository [7].

4.5.2 Integrating Persistence in eyeOS Calendar

Calendar and event sync processes are performed when eyeOS Calendar is opened.

Calendars are synced every 20 seconds. If any changes are detected, the calendar list is refreshed and the waiting time is reset.

Events are synced every 10 seconds. If any changes are detected, the events of the period shown on screen are refreshed and the waiting time is reset.

Persistence in implemented in eyeOS according to the diagram below:

[7] https://github.com/cloudspaces/eyeos-u1db#implementation-of-u1db-into-eyeos-calendar

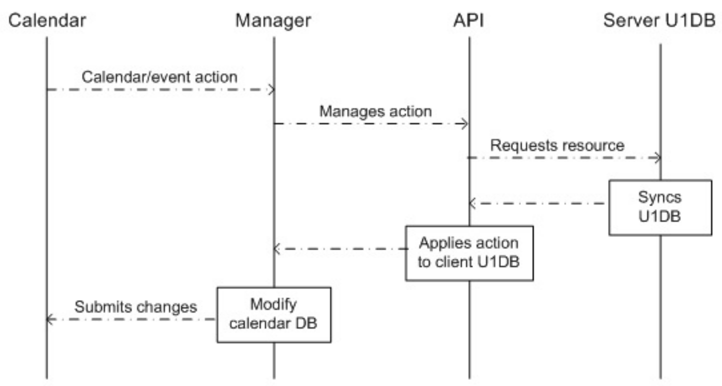


Figure 40: eyeOS persistence calendar sequence diagram

The user performs an action on the calendar, such as adds an event to a specific calendar. The Manager uses the `insertEvent` function to retrieve the user's credentials. These values are sent to the API together with the data of the new event. The API is responsible for requesting the resource from the U1DB server using the `insertEvent` function.

If everything is correct, the U1DB server updates the server database and subsequently notifies other clients of the change. The API receives confirmation of the change and applies it to the client U1DB database. Once the updates have been made, it notifies the Manager, which modifies the local calendar tables and updates the eyeOS interface.

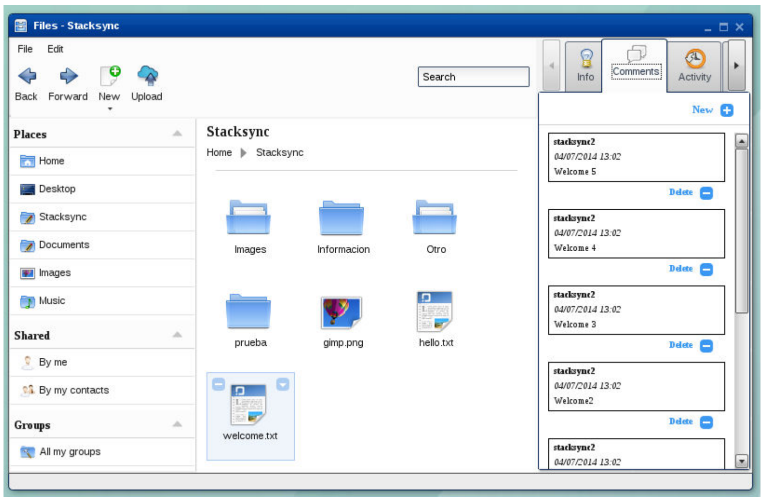
The `insertEvent` functions of the Calendar Manager and the Calendar API, as well as the other actions performed by the calendar, are described in more detail in the GitHub repository [8], respectively.

4.5.3 Integrating Persistence in eyeOS Comments Tool

The comments tool is only valid in the user's Personal Cloud because it is a collaboration resource between StackSync and eyeOS.

The user accesses the comments tool when they select a Personal Cloud file or directory in the file manager and then clicks the "Comments" tab in the right toolbar (social bar).

[8]https://github.com/cloudspaces/eyeos-u1db#implementation-of-u1db-into-eyeos-calendar



stacksync2
04/07/2014 13:02
Welcome 5

stacksync2
04/07/2014 13:02
Welcome 4

stacksync2
04/07/2014 13:02
Welcome 3

stacksync2
04/07/2014 13:02
Welcome2

stacksync2
04/07/2014 13:02

Figure 41: eyeOS file comments

The comments tool lists all comments inserted for a specific element, both those from the element's owner and from other users who have been given privileges to access the element.

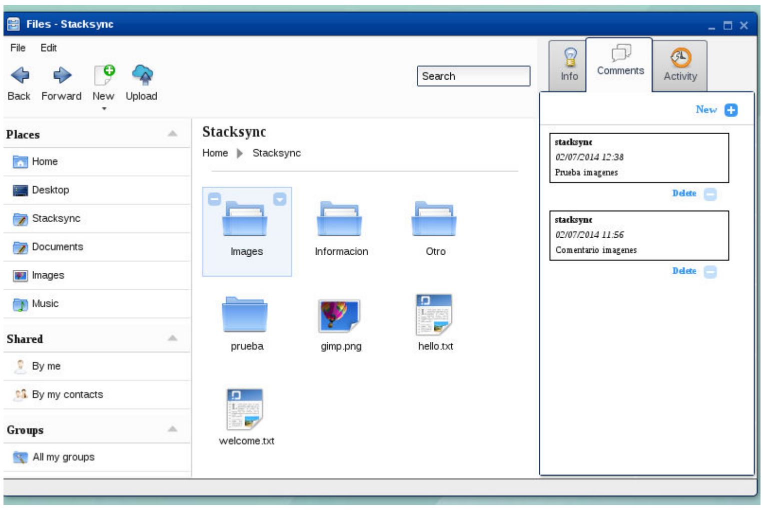
The data given in a list comment is:

- name of the user who made the comment
- creation date/time
- text entered.

Example:

User	stacksync2
Date / time	04/07/2014 13:02
Text	Welcome 3

Other actions that can be performed include insert new comments with the "New" option or delete existing comments with the "Delete" option. The delete option is restricted; only comments made by the user in question can be deleted. Users cannot delete comments made by others.



stacksync
02/07/2014 12:38
Prueba imagenes

stacksync
02/07/2014 11:56
Comentario imagenes

Figure 42: eyeOS file comments

The comments list is not refreshed, as there is no background process that enables new comments to be displayed automatically.

Persistence is implemented in the eyeOS comments tool according to the diagram below:

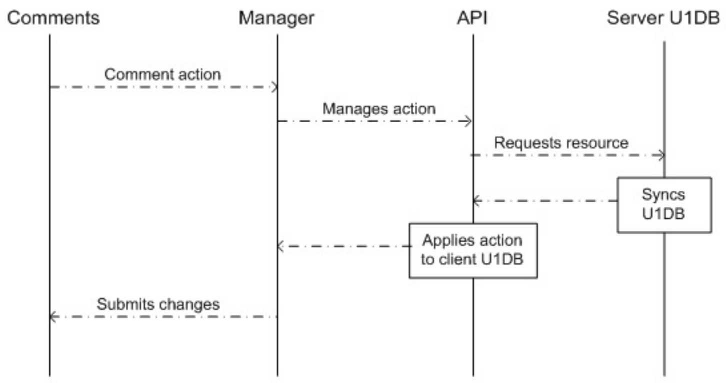


Figure 43: eyeOS comments sequence diagram

The user performs an action, such as inserts a new comment in the list. The Manager using the createComment function retrieves the user's credentials. These values are sent to the API together with the data of the new comment. The API is responsible for requesting the resource from the U1DB server using the createComment function.

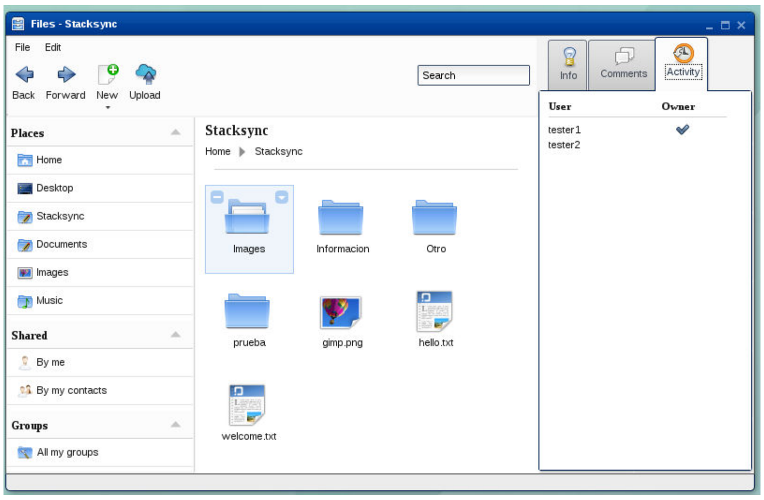
If everything is correct, the U1DB server updates the server database and subsequently notifies other clients of the change. The API receives confirmation of the change and applies it to the client U1DB database. Once the updates have been made, it notifies the Manager, which will update the eyeOS interface.

The createComment functions of the Comment Manager and the Comment API, as well as the other actions performed by the comments tool.

4.6 Share

To share folders and their contents among users of the same or different Personal Clouds, the Share tool is implemented in the file manager.

The user can access the tool by selecting a Personal Cloud directory and then clicking the "Activity" tab in the right toolbar (social bar).

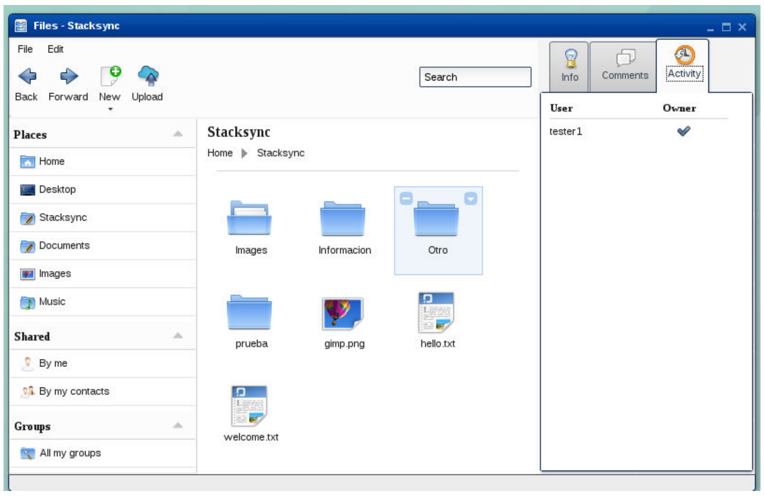


User	Owner
tester1	✓
tester2	

Figure 44: eyeOS sharing feature

This tab lists all users that can access and manage the active directory. Furthermore it indicates who is the owner of the directory.

Where the directory is not shared, only the directory owner is displayed.



User	Owner
tester1	✓

Figure 45: eyeOS not shared folder

If the user wants to share or add more users to the sharing list, they must right click on the directory to open a contextual menu and then select the "Share" option. When they select this option, a form appears in which they need to enter the email addresses of the people with whom they want to share the directory. Once the form has been completed, the data is sent to StackSync. If the operation is done successfully, the form closes. When the user accesses the "Activity" tab of the directory again, they will see the new users added to the list.

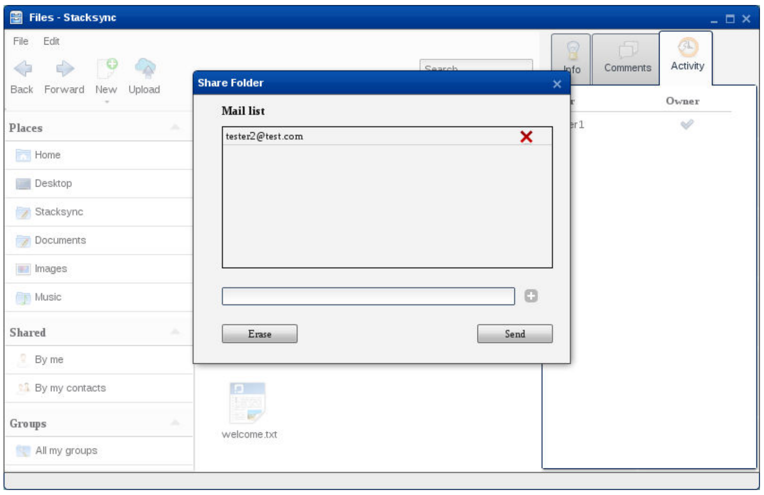
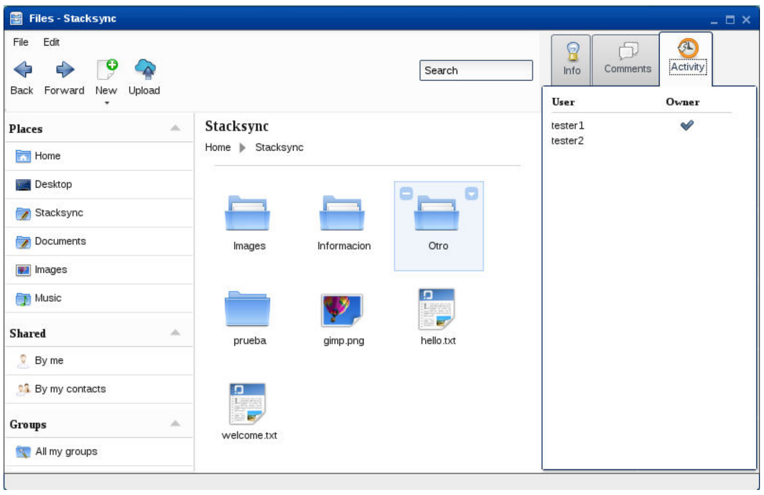


Figure 46: eyeOS sharing dialog



User	Owner
tester1	✓
tester2	

Figure 47: eyeOS shared folder

The list of users sharing the directory is not refreshed, as there is no background process that enables new users to be displayed automatically.

Directory sharing is implemented in eyeOS according to the diagram below:

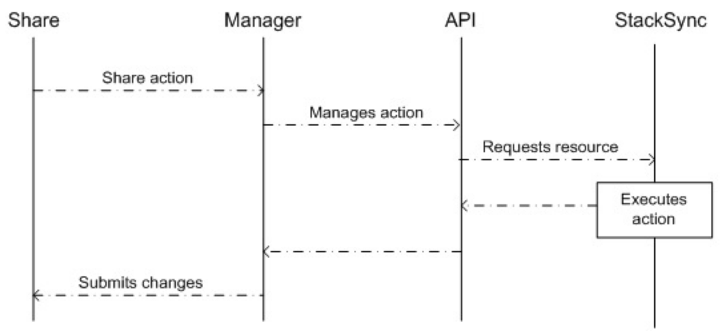


Figure 48: eyeOS sharing sequence diagram

The user performs an action in the file manager, such as lists the users who have access to the directory. The Manager using the `getListUsersShare` function retrieves the user's access token and the id of the directory in StackSync. These values are sent to the API, which is responsible for requesting the resource from StackSync using the getListUsersShare function. It receives the list of users and then it notifies the Manager, which will update the eyeOS interface.

The `getListUsersShare` of the Share Manager and the Share API, as well as the other actions performed by the Share tool.

5 Tissat

Next, we detail the tasks performed by Tissat, which have allowed us to improve safety, efficiency and potential by leveraging the proposed solution on OpenStack.

5.1 Migrating from Keystone v2.0 to v3

As Tissat is a cloud provider, we need to be able to grant administrative privileges to our customers so that they can manage their own resources (i.e. containers, tenants, user accounts).

Unfortunately, on Keystone v2.0, the "admin" role has a global scope that allows a user with admin privileges on a specific tenant to erase other tenants.

Since Tissat has active customers that need to have admin privileges to create their own users, we had to overcome this drawback where a customer could delete other customer's data. Causing an obvious security concern.

Even though this issue was solved in later releases of the Stacksync Server through a change in stacksync's user data mapping scheme. The change involved going from a tuple "tenant: user" to a "stacksync_tenant:user_container" scheme, where all users have a container under the same tenant.

Another challenge that we faced was the fact that system administrators did not want to grant a "stacksync admin user" with a `ResellerAdmin` role. This role was necessary in order to change the quota and other attributes on Stacksync, but it could lead to a situation where an "stacksync admin user" could actually delete other user's data.

At that point, we solved the problem by adopting Keystone v3 for our cloud infrastructure and using a domain-based approach. This way we could differentiate entities where administrative users could not interfere with other resources belonging to other users/customers. So we are currently using Keystone v3 coupled with StackSync Server v0.4.4 on our main environment.

5.2 Secure our Stacksync platform with SSL

Given the fact Stacksync clients transfers potentially private data, we must ensure that data is transmitted safely over an insecure network such as the Internet.

We can identify three phases on the Stacksync workflow where data could be sent in plaintext and therefore exposed to attack. These are: 1) the Keystone authentication process; 2) the Swift file transfer; and 3) the RabbitMQ metadata transfer.

Our solution was to set up SSL on all services that involve data transfers over the Internet. At this moment, we have already configured Keystone and Swift to work over SSL and guarantee our user's privacy.

We are working on the RabbitMQ setup to enable SSL, but it requires some actions to be taken in StackSync to be able to adapt it to this new scenario.

5.3 Development of group-based membership for Stacksync users

The ability to organize users into groups or departments makes it easier to handle the quota management, allowing the existence of administrators group, which can manage users and their available space using StackSync.

For instance, a group URV would manage a limited amount of storage space, and the organization would have an administrative user that would create and delete users and assign quotas to their users.

The following use case shown in figure 49 presents a situation where a "Cloud admin" user creates groups and grants "Group admin" privileges to a certain user.

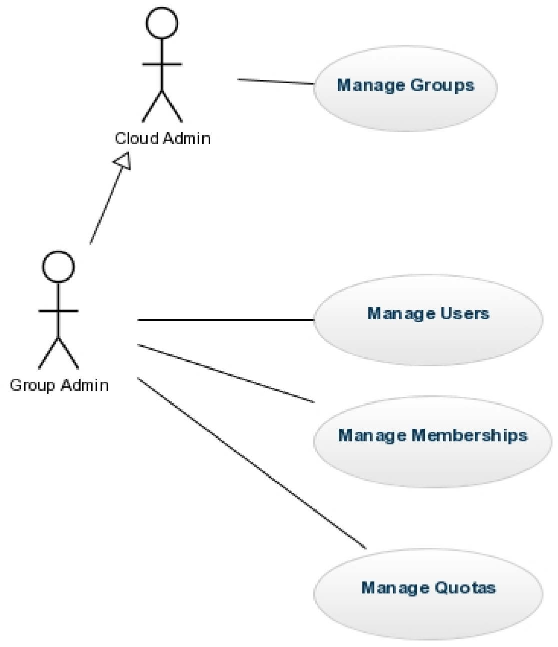


Figure 49: Group permission

Since StackSync is an evolving platform, we tried to decouple the current user scheme as much as possible from our group development. In the following diagram, we present the StackSync group application model.

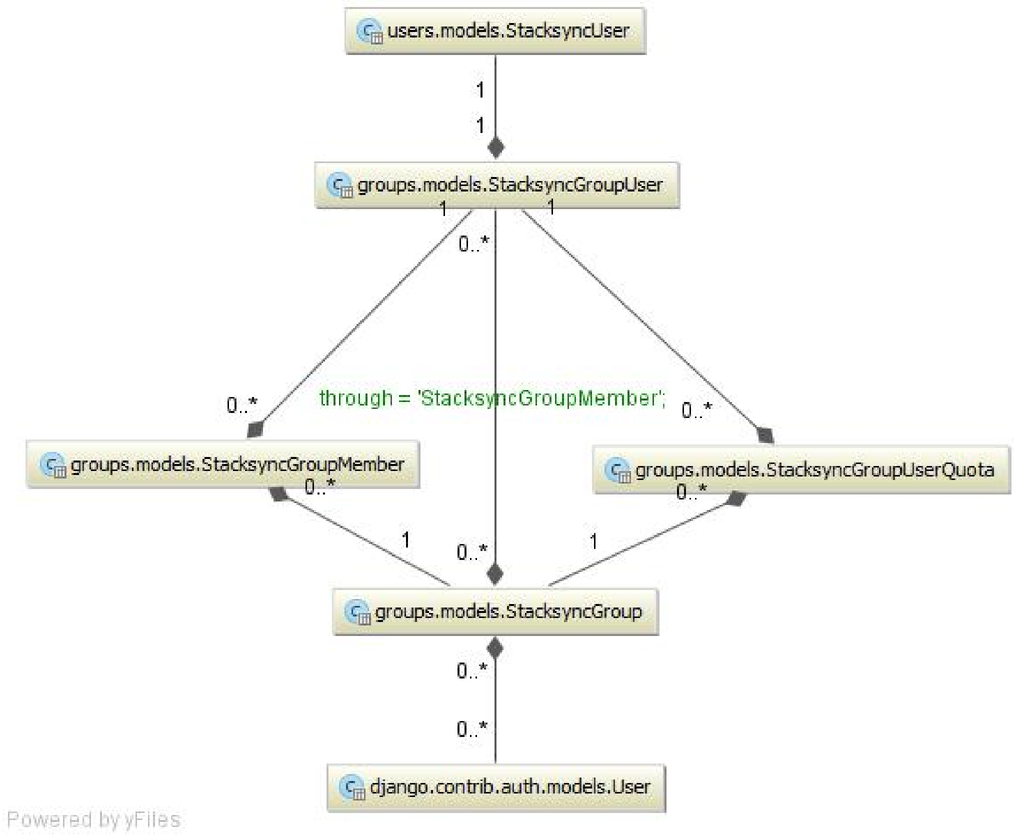

users.models.StacksyncUser
1
1
groups.models.StacksyncGroupUser
1
1
0..*
through = 'StacksyncGroupMember';
0..*
0..*
groups.models.StacksyncGroupMember
0..*
groups.models.StacksyncGroupUserQuota
0..*
1
0..*
1
groups.models.StacksyncGroup
0..*
0..*
django.contrib.auth.models.User
Powered by yFiles

Figure 50: StackSync group application model

5.4 Development of a group-based quota web application

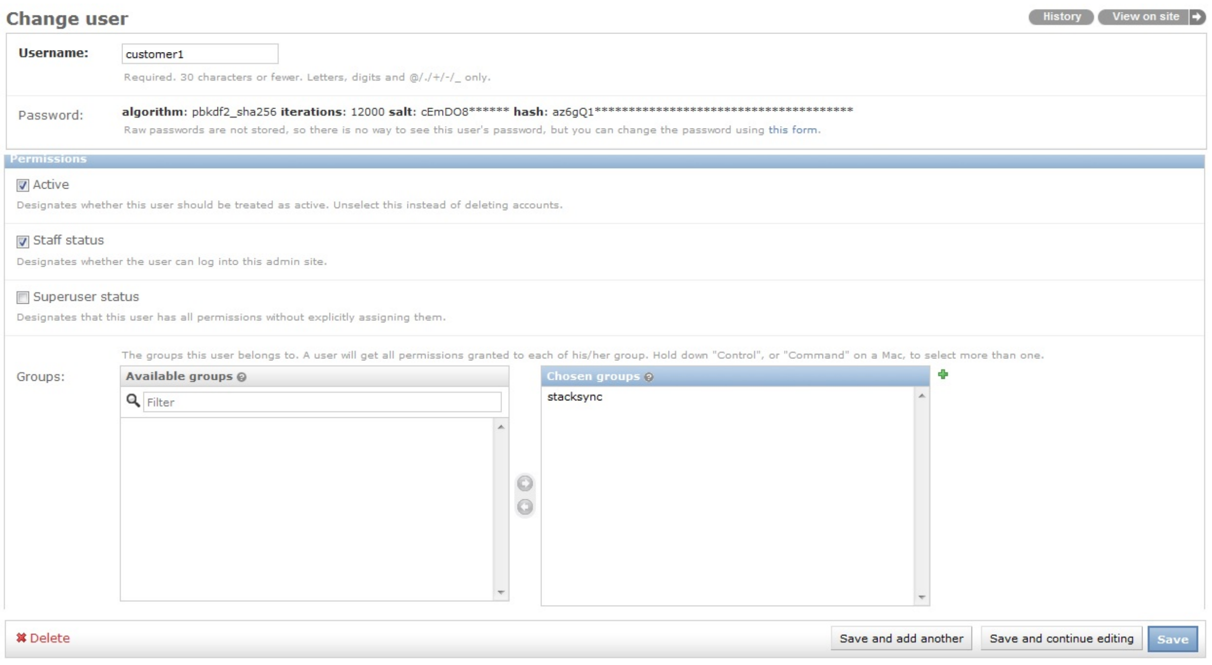
Because generating administration sites for our group managers is a repetitive work, we want to provide an easy way to perform these tasks. For that reason, we have chosen the Django framework to automate the user creation using StackSync through the admin interface.

Our work was mainly focused on restricting that groups could not interfere with other group users.

For each model class we have implemented the corresponding `ModelAdmin` class, in order for the Django admin interface to work properly.

If we go back to our use case where a group/company would use certain amount of storage space. They could have some administrative user to create and delete users, and assign quota to their users. We need to follow these steps to complete these task.

- Create a Group Admin with the StackSync permissions


Change user

History | View on site

Username: customer1
Required. 30 characters or fewer. Letters, digits and @/./+/-/_ only.

Password: **algorithm**: pbkdf2_sha256 **iterations**: 12000 **salt**: cEmDO8****** **hash**: az6gQ1**************************************
Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form.

Permissions

☑ Active
Designates whether this user should be treated as active. Unselect this instead of deleting accounts.

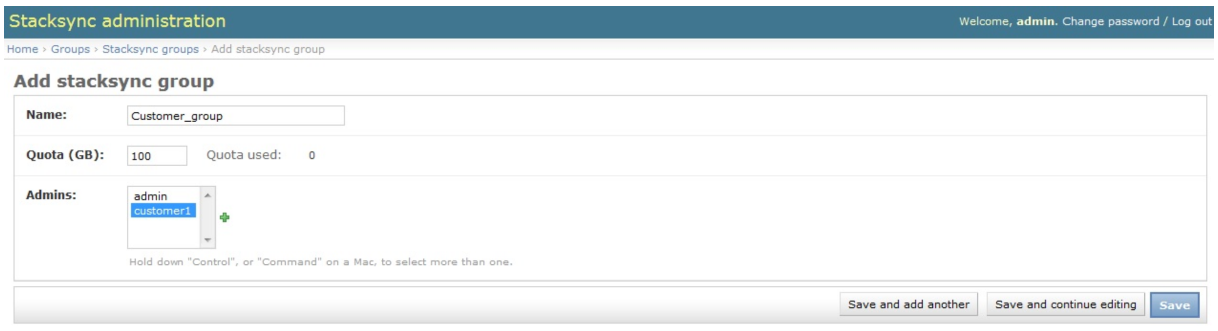
☑ Staff status
Designates whether the user can log into this admin site.

☐ Superuser status
Designates that this user has all permissions without explicitly assigning them.

Groups: The groups this user belongs to. A user will get all permissions granted to each of his/her group. Hold down "Control", or "Command" on a Mac, to select more than one.

Available groups | Chosen groups: stacksync

- Create StackSync Group with a specific quota


Stacksync administration — Welcome, **admin**. Change password / Log out

Home › Groups › Stacksync groups › Add stacksync group

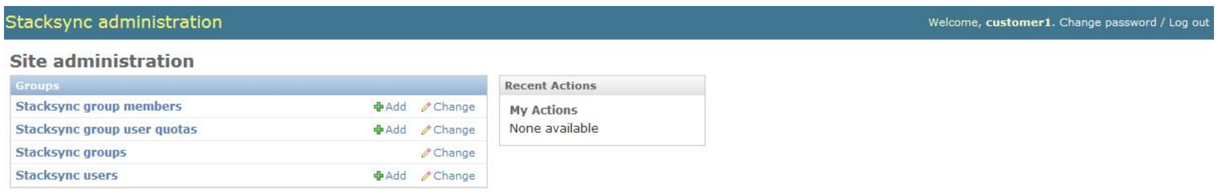
Add stacksync group

Name: Customer_group

Quota (GB): 100 Quota used: 0

Admins: admin, customer1
Hold down "Control", or "Command" on a Mac, to select more than one.

- Now the group/customer manager can access as an admin to create his own users, memberships, and quotas.


Stacksync administration — Welcome, **customer1**. Change password / Log out

Site administration

Groups		
Stacksync group members	Add	Change
Stacksync group user quotas	Add	Change
Stacksync groups		Change
Stacksync users	Add	Change

Recent Actions

My Actions
None available

In this use case, the manager creates a user, then creates a membership and at last creates a quota.

5.5 Migrate web interface back-end from PHP to Python

The initial prototype of the web interface was created in PHP because it was considered appropriate to use a single programming language for all development environments, making it easier for the community by limiting the number of programming languages.

During our development with PHP we encountered issues when having to interact with OpenStack products. We had to use 3rd party framework and connectors that were not widely supported. Therefore, we soon realized that changing the programming language to Python could boost the development progress by using official OpenStack libraries and connectors.

As the application was still in an early stage and the development team had already worked with Python and the Django framework in previous projects, we decided to migrated the application to the new language.

The Django framework enabled us to use a powerful scaffolding system that could help us develop simple applications in a very easy and rapid way.

In order to interact with StackSync, we developed a web client and an administrative interface using freely distributed components.

At this moment, we have an easier codebase with lesser dependencies and more testing since we are using standard and official tools.

5.6 Refactoring of Stacksync web client

In the next figure we show the new user interface, where users can see their storage quota, in case it has been defined.

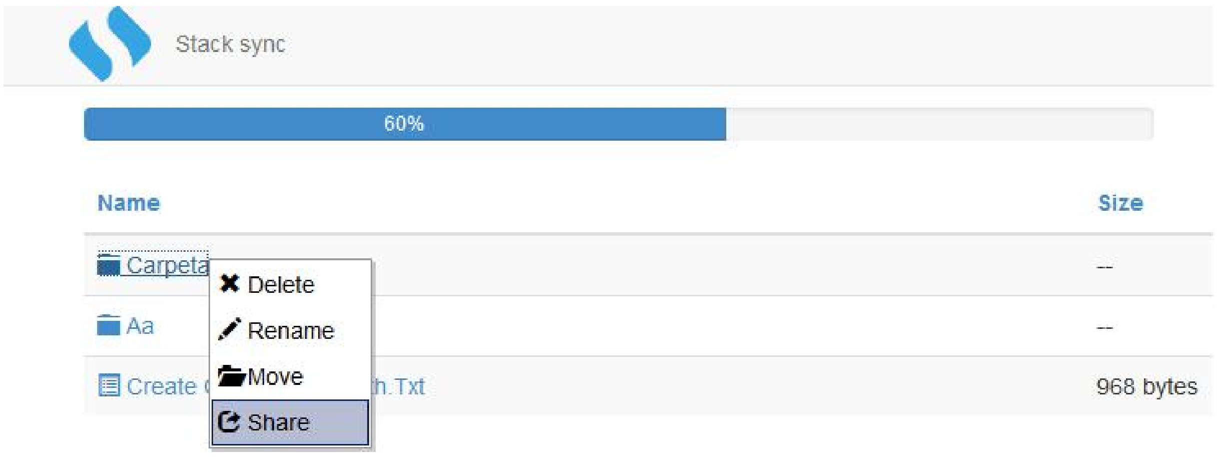
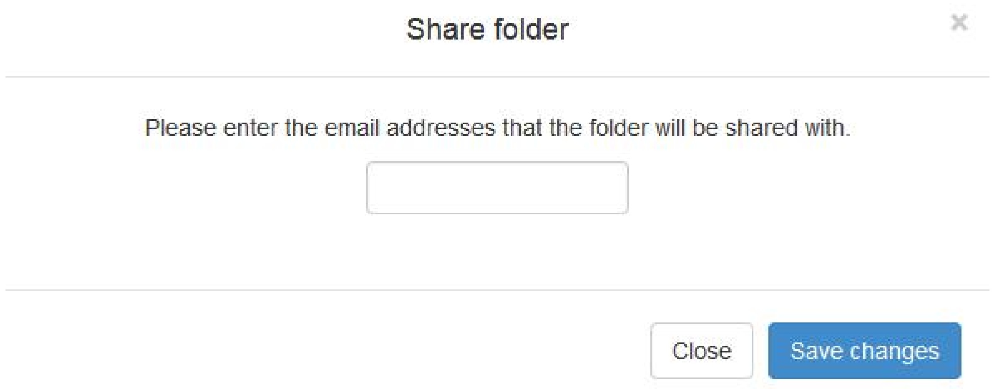


Figure 51: StackSync web client

As seen in figure 51, we have also developed a functionality to share a folder through the web client. A user will be asked to provide the email of the user she wants to invite to the folder.


Share folder

Please enter the email addresses that the folder will be shared with.

Figure 52: StackSync web client share functionality

5.7 StackSync support for ownCloud

We have started the integration studies for sharing between ownCloud and Stacksync platforms.

We have configured an ownCloud server 7.0.2 to use Openstack Swift as storage backend, we tested it in order to analyze if it worked properly with the CloudSpaces infrastructure, and it was determined that in order for ownCloud's synchronization to work properly, we needed to update its database to a development version (daily build).

When using ownCloud, we first uploaded a file to a web server running ownCloud, and after that, the server uploaded the file to the Swift container.

When you downloading a file, ownCloud first retrieves the file from Swift, stores it in a temporary folder in the web server, and then servers it to the user.

In order to create a StackSync implementation, we only need to replace the native upload/download/delete /list file operations for the corresponding StackSync calls.

We have chosen to compare with Opencloud Swift, which is the current Openstack library for OwnCloud. We have documented how to replace regular Swift calls with their StackSync equivalents.

In the following diagram, we compare the current functionalities of Owncloud, and how it would look when running on StackSync.

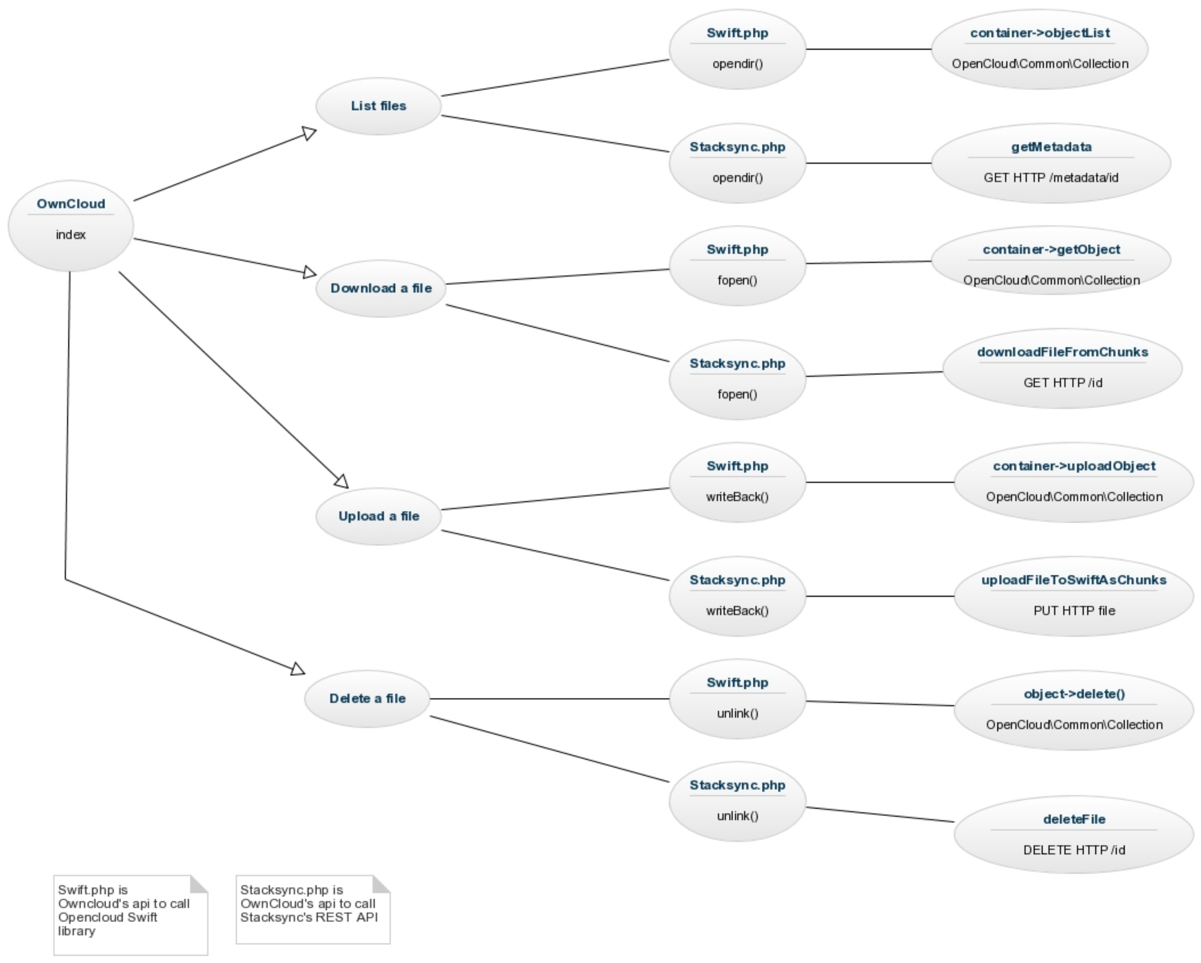
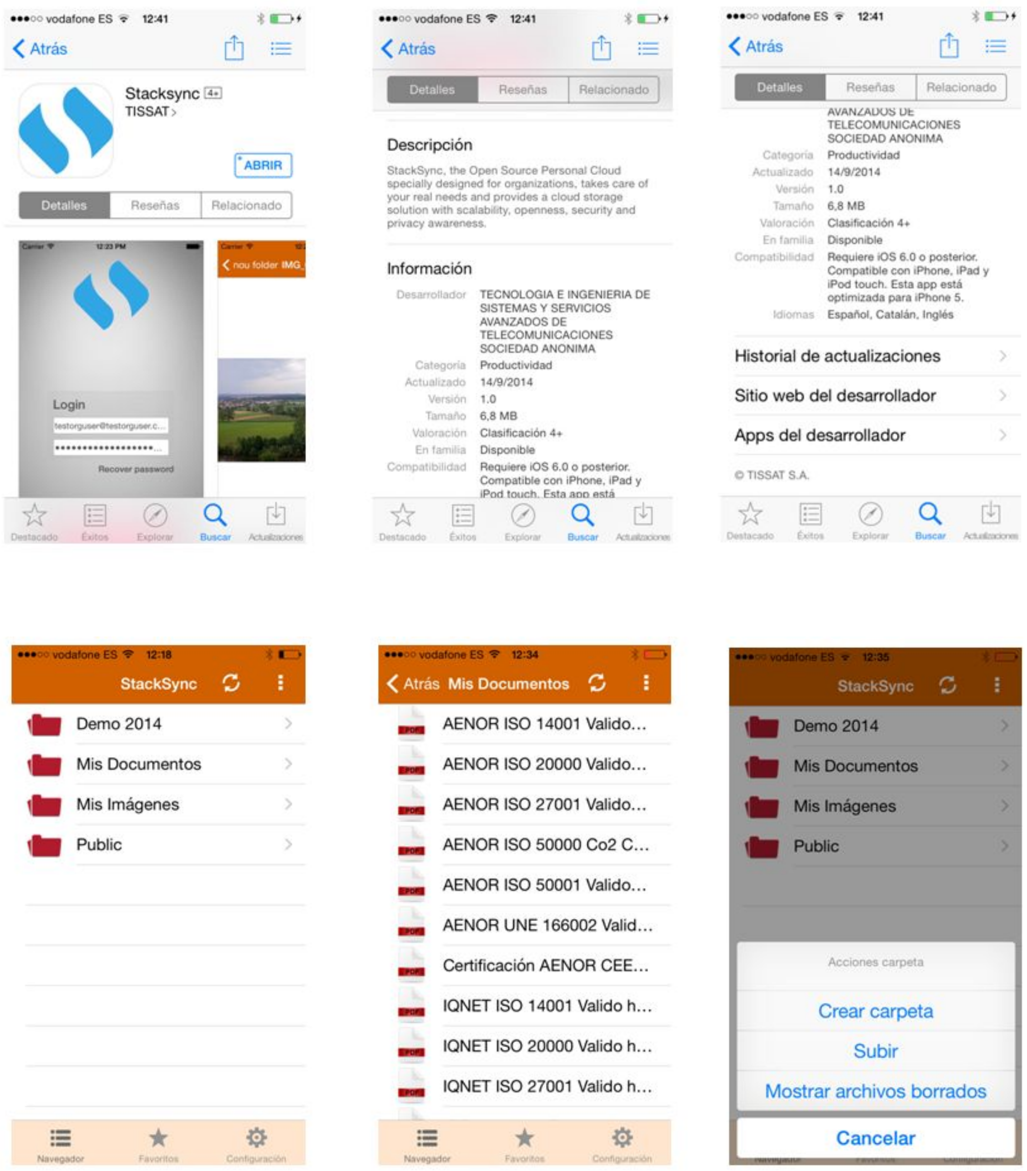


Figure 53: StackSync functionalities

5.8 Development of StackSync iOS app

We have developed and published in the App Store the official StackSync application for iPhone and iPad. So that users of this platform can securely access to their files.



Descripción

StackSync, the Open Source Personal Cloud specially designed for organizations, takes care of your real needs and provides a cloud storage solution with scalability, openness, security and privacy awareness.

Información

Desarrollador	TECNOLOGIA E INGENIERIA DE SISTEMAS Y SERVICIOS AVANZADOS DE TELECOMUNICACIONES SOCIEDAD ANONIMA
Categoría	Productividad
Actualizado	14/9/2014
Versión	1.0
Tamaño	6,8 MB
Valoración	Clasificación 4+
En familia	Disponible
Compatibilidad	Requiere iOS 6.0 o posterior. Compatible con iPhone, iPad y iPod touch. Esta app está optimizada para iPhone 5.
Idiomas	Español, Catalán, Inglés

Figure 54: StackSync iOS app

This prototype application extends the growth potential of the solution, in addition to serve as a basis for developing more advanced versions incorporating file sharing and also new functionalities and advanced services for the platform.

References

[1] I. Drago, M. Mellia, M. Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside dropbox: Understanding personal cloud storage services," in Proc. of ACM Internet Measurement Conference (IMC), 2012, pp. 481–494.

[2] "How we have scaled dropbox," https://www.youtube.com/watch?v=PE4gwstWhmc.

[3] K. Jayaram, "Elastic remote methods," in ACM/IFIP/USENIX Middleware 2013, 2013, pp. 143–162.

[4] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras, "Benchmarking personal cloud storage," in Proc. of ACM Internet Measurement Conference (IMC), 2013.

[5] OASIS, "Amqp: Advanced message queueing protocol," http://www.amqp.org/.

[6] S. C. Kendall, J. Waldo, A. Wollrath, and G. Wyant, "A note on distributed computing," Mountain View, CA, USA, Tech. Rep., 1994.

[7] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," ACM Transactions on Autonomous and Adaptive Systems (TAAS), vol. 3, no. 1, pp. 1:1–1:39, 2008.

[8] R. Gracia-Tinedo, M. Sánchez-Artigas, A. Moreno-Martínez, C. Cotes-González, and P. García-López, "Actively Measuring Personal Cloud Storage," in Proc. of IEEE CLOUD'13, 2013, pp. 301–308.

[9] H. F. Songbin Liu, Xiaomeng Huang and G. Yang, "Understanding data characteristics and access patterns in a cloud storage system," in Proc. of IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGRID), 2013, pp. 327–334.