ALMA MATER STUDIORUM · UNIVERSITY OF BOLOGNA

FACULTY OF MATHEMATICAL, PHYSICAL AND NATURAL SCIENCES
Undergraduate Degree Course in Computer Science

DATA CLOUD THROUGH GOOGLE CLOUD STORAGE

Thesis in Databases

Supervisor:
Professor
DANILO MONTESI

Presented by:
GINO CAPPELLI

Session III
2010/2011

Abstract

Il *Cloud Storage* è un modello di conservazione dati su computer in rete, dove i dati stessi sono memorizzati su molteplici server, reali e/o virtuali, generalmente ospitati presso strutture di terze parti o su server dedicati. Tramite questo modello è possibile accedere alle informazioni personali o aziendali, siano essi video, fotografie, musica, database o file in maniera "smaterializzata", senza conoscere l'ubicazione fisica dei dati, da qualsiasi parte del mondo, con un qualsiasi dispositivo adeguato. I vantaggi di questa metodologia sono molteplici: infinita capacita' di spazio di memoria, pagamento solo dell'effettiva quantità di memoria utilizzata, file accessibili da qualunque parte del mondo, manutenzione estremamente ridotta e maggiore sicurezza in quanto i file sono protetti da furto, fuoco o danni che potrebbero avvenire su computer locali.

Google Cloud Storage cade in questa categoria: è un servizio per sviluppatori fornito da Google che permette di salvare e manipolare dati direttamente sull'infrastruttura di Google. In maggior dettaglio, Google Cloud Storage fornisce un'interfaccia di programmazione che fa uso di semplici richieste HTTP per eseguire operazioni sulla propria infrastruttura. Esempi di operazioni ammissibili sono: upload di un file, download di un file, eliminazione di un file, ottenere la lista dei file oppure la dimensione di un dato file. Ogniuna di queste richieste HTTP incapsula l'informazione sul metodo utilizzato (il tipo di richista, come GET, PUT, ...) e un'informazione di "portata" (la risorsa su cui effettuare la richiesta). Ne segue che diventa possibile la creazione di un'applicazione che, facendo uso di queste richieste HTTP, fornisce un servizio di Cloud Storage (in cui le applicazioni salvano dati in remoto generalmene attraverso dei server di terze parti).

In questa tesi, dopo aver analizzato tutti i dettagli del servizio Google Cloud Storage, è stata implementata un'applicazione, chiamata *iHD*, che fa uso di quest'ultimo servizio per salvare, manipolare e condividere dati in remoto (nel "cloud"). Operazioni comuni di questa applicazione permettono di condividere cartelle tra più utenti iscritti al servizio, eseguire operazioni di upload e download di file, eliminare cartelle o file ed infine creare cartelle. L'esigenza di un'appliazione di questo tipo è nata da un forte incremento, sul merato della telefonia mobile, di dispositivi con tecnologie e con funzioni sempre più legate ad Internet ed alla connettività che esso offre.

La tesi presenta anche una descrizione delle fasi di progettazione e implementazione riguardanti l'applicazione *iHD*. Nella fase di progettazione si sono analizzati tutti i requisiti funzionali e non funzionali dell'applicazione ed infine tutti i moduli da cui è composta quest'ultima. Infine, per quanto riguarda la fase di implementazione, la tesi presenta tutte le classi ed i rispettivi metodi presenti per ogni modulo, ed in alcuni casi anche come queste classi sono state effettivamente implementate nel linguaggio di programmazione utilizzato.

To my parents

Contents

List of Figures

List of Tables

TRADEMARKS

Chapter 1

Introduction

1.1 What is Google Cloud Storage?

The term “Cloud” refers to the technologies that provide convenient, on-demand network access to a shared pool of computing resources such as networks, servers, storage, applications and services [sye11]. In particular, *Cloud Storage* is a model of networked storage where applications store data in a remote pool of storage (multiple servers) which is generally hosted by third parties. *Google Cloud Storage* [gog01] falls into this category. It is a service for developers provided by Google, that permits to store and access data on the Google’s infrastructure. This means that from any type of device with an internet connection it is possible to store and manage files on the web through Google Storage. This approach is becoming more and more common in recent years due to many factors:

- *Hardware limitations*: The different kind of devices throughout the world, with different software and hardware specification. For instance a smartphone device will provide less memory capacity than a personal computer. The usage of a web service for storing data remotely can overcome these limitations [fur10].
- *Productivity*:
  1. In the corporate world, allowing employees to access their company’s files from their home means that they can be more productive. They can perform work from home instead of having to remain at the office to access files that are on the corporate network [mil09].
  2. Storing files in a remote single location allows to develop advance features such as *file sharing* or *traceability.*
- *Simplicity*: From an user’s point of view, data can be accessed from any part of the globe with just an internet connection. From a developer’s point of view, the

maintenance of a system of file storing it is a difficult issue. Providing a “ready-to-use” interface to access and manage data remotely will simplify all the applications that use this layer.

- *Safety*: Since the data is stored remotely, data is secured from theft, fire or any other damages that might happen to local computer/business [sch12].
- *Costs*:
    1. Companies need only pay for the storage they actually use [shr10].
    2. Storage maintenance tasks, such as backup, data replication, and purchasing additional storage devices are offloaded to the responsibility of a service provider, allowing organizations to focus on their core business [ant10].

Google Cloud Storage provides a RESTful programming interface: applications can use standard **HTTP methods**, such as PUT, GET, POST, HEAD and DELETE to store, share, and manage data [gog02]. Each HTTP request includes the HTTP protocol version (1), a request method (2), a request URI (3), a set of request headers (4) and the body of the request (5). In order to communicate properly with Google Storage applications populate these field in the following manner:

1. *HTTP protocol version*: The Google Cloud Storage supports HTTP/1.1.
2. *request method*: GET, PUT, POST, HEAD or DELETE.
3. *request URI*: all files in the system can be identified by an unique URL—a resource on which you can perform operations with HTTP methods.
4. *request headers*: Google Storage supports the HTTP/1.1 request headers. For example, the “content-length” header is an integer that in a PUT request specifies the size of the file to upload.
5. *request body*: For instance, it can be a file to upload.

These characteristics will be better explained in Chapter 2, which will provide an in-depth view of the Google Cloud Storage developers API.

1.2 Goals of the thesis

This work is conceived mainly for two different purposes:

1. **To explore all the concerns regarding the Cloud Storage**:
   Nowadays, with the spreading of devices with internet capability and the reduction of the internet connection costs, the Cloud Storage model, and more in general the Cloud Computing, represents an important field of the Computer Science.
   The two biggest concerns about Cloud Storage are *reliability* and *security*. Keeping Google Cloud Storage as a reference, we will enhance our understanding on these problems.

2. **To develop a client application for Google Storage**:
   In this thesis will be described the development of a software application written in Java, which is able to communicate with Google Storage through its API. In particular, the main interest is to develop a complete application of Cloud Storage, named **iHD**, that permits to store and manage data "in the Cloud".

1.3 Related work

On the market there are several products of Cloud Storage. The main characteristics that distinguish these products are large availability of remote storage, file sharing, content management interface, file synchronization and integration with other services.

Dropbox [dro01] is a web-based file hosting service operated by *Dropbox, Inc.* that uses cloud storage to enable users to store and share files with others across the Internet using file synchronization. It offers a free account of 2 GB and a paid account of 50 GB, 100 GB, and a team account of 1 TB or more.

Box.net [box01] is an online File Sharing and Cloud Content Management service for enterprise companies from *Alexa Internet, Inc.* Box offers 3 account types: Enterprise, Business and Personal. The company provides 5GB of free storage for personal accounts.

iCloud [icl01] is a cloud storage and cloud computing service from *Apple, Inc.* that allows users to store data on remote computer servers for download to multiple devices such as iOS-based devices and personal computers running Mac OS X or Microsoft Windows. Each iCloud account has 5 GB of free storage.

Windows Live SkyDrive [sky01] is a file hosting service from *Microsoft, Inc.* that allows users to upload files to a cloud storage and access them from a Web browser. The service offers 25 GB of free personal storage, with individual files limited to 100 MB.

iHD [ihd01] is the cloud storage service that will be developed throughout this thesis. It provides a multi-platform client application with a simple interface to manage and share files among users. Will be provided 200MB of free personal storage.

The following table shows the different features of each Cloud Storage service previously exposed.

	iHD	Dropbox	Box.net	iCloud	SkyDrive
Desktop client application	**Yes**	**Yes**	**Yes**	**Yes**	
Web application		**Yes**	**Yes**		**Yes**
File sharing	**Yes**	**Yes**	**Yes**		**Yes**
File synchronization		**Yes**	**Yes**	**Yes**	
Integration with external services			**Yes**	**Yes**	**Yes**
Developers API		**Yes**	**Yes**		

Table 1.1: Comparison between different features of various Cloud Storage services

1.4 Thesis outline

Chapter 2 summarise all the characteristics and details of the Google Cloud Storage service. In particular, the chapter explains all the HTTP methods necessary to establish a complete communication with the Google Cloud Storage server.

Chapter 3 presents the design of the *iHD* application. In this chapter will be evaluated all the requirements on which the iHD application will be developed. Finally, a description of all the modules that compose the application will be provided in order to pave the way to the development process.

Chapter 4 discusses the implementation of the *iHD* application, dwelling on the development issues related to Java and its environment. Moreover, an overview of all the components of the iHD application will be provided, showing in many cases how these components are effectively implemented in Java.

Chapter 5 assesses the contributions of this thesis and highlights directions for future work.

Chapter 2

Google Cloud Storage Technical Background

Google Cloud Storage is an extensive and structured service. Among the many features that will not be viewed in this thesis there are:

- Interoperability with some cloud storage tools and libraries that work with services such as Amazon Simple Storage Service (Amazon S3) [ama01] and Eucalyptus Systems, Inc [euc01].
- Integration with Google's accounts and groups. In particular, it is possible to restrict the access to an object specifying an opportune ACL (access control list) which contains a Google Storage ID (string of 64 hexadecimal digits that identifies a specific Google account holder or a specific Google group).
- Support for the *OAuth 2.0* authentication and authorization to interact with its API. The OAuth 2.0 protocol gives out OAuth tokens. OAuth tokens authenticate tools and applications to access Google Cloud Storage API and also provides the ability to restrict access using scopes.
- Possibility to interact with the Google APIs Console. Google Cloud Storage is available as a service for Google APIs Console projects. It is possible to have many projects and many instances of the Google Cloud Storage service.

In the following chapter will be explored only the characteristics of Google Cloud Storage that are useful to the development of the *iHD* client application.

2.1 Features and Capabilities

2.1.1 High Capacity and Scalability

Google Cloud storage permits to exploit the excellent Google's infrastructure. It provides a scalable storage architecture and a powerful networking infrastructure accessible from two locations: Europe or United States. Google Cloud Storage supports a large number of accounts and objects that can be terabytes in size.

2.1.2 Consistency

Google Cloud Storage provide a strong *read-after-write* consistency for all upload and delete operations. In particular, objects in Google Cloud Storage are either available or not available:

- During upload operations, objects are not available until they are completely uploaded. By extension, uploaded objects are never available for download in a corrupted state or as partial objects. Moreover, when a file is uploaded to Google Cloud Storage, and a success response is received, the file is immediately available for download operations. This is true for download operations and also for the overwriting of an existing file.
- If an object is deleted, an immediate attempt to download the object will result in a *404 Not Found* status code. It is possible to delete only object in an available state. For instance, a delete operation on a file that is being overwritten will result in an error.
- List operations are *eventually* consistent: when a object is created, the newly-created object might not immediately appear in the returned list of objects.

2.1.3 RESTful API (Application Programming Interface)

Google Cloud Storage provides a simple RESTful programming interface which permits to store, share and manage data simply using the standard HTTP methods (GET, PUT, POST, HEAD and DELETE). This method is designed to encapsulate in each request two informations: the type of the request (PUT, GET, ...) and the resource on which perform the request (the URI). The URI is a path to the resource which consists of an object name and a container name. Used together, the object name and the container name create a unique URL to a given resource–a resource on which it is possible to perform operations with HTTP methods.

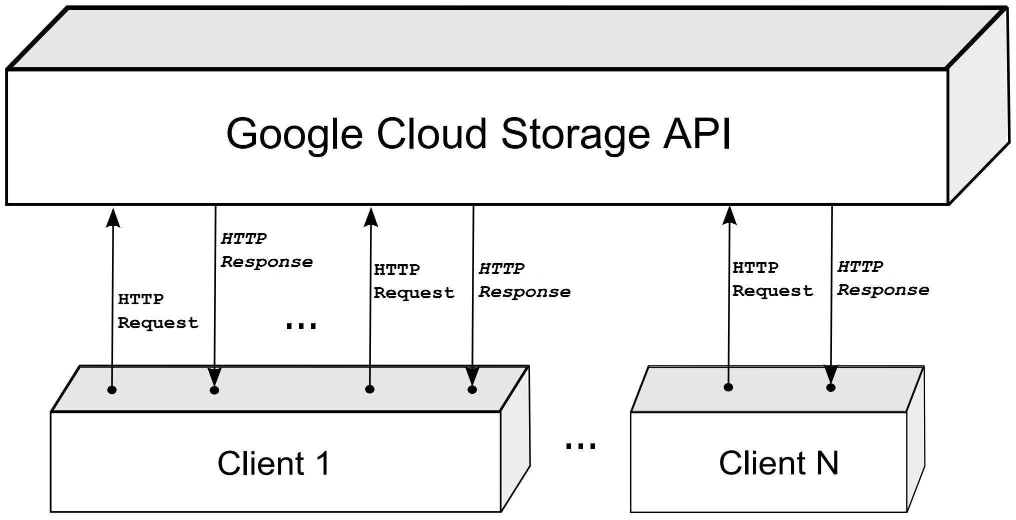


Figure 2.1: Google Cloud Storage HTTP requests and responses

2.1.4 Authentication

In order to interact with the Google Cloud Storage API, each HTTP request must be authenticated through the **HMAC** authentication code. The HMAC (Hash-based Message Authentication Code) is a specific algorithm for calculating a message authentication code (MAC) using a cryptographic **hash function** in combination with a **secret key**. The secret key that will be used is the **Google Cloud Storage developer key**. Developer keys consist of an *access key* and *secret*.

An *access key* is a 20 character alphanumeric string, which is linked to each Google account. It must be used in all authenticated Google Cloud Storage requests, so that the Google Cloud Storage system knows who is making the request. The following is an example of an access key:

GOOGTS7C7FUP3AIRVJTE

A *secret* is a 40 character Base-64 encoded string that is linked to a specific access key. A secret is a pre-shared key that only the owner of an account and the Google Cloud Storage system know. The secret is used to sign all requests as part of the authentication process. The following is an example of a secret:

bGoa+V7g/yqDXvKRqq+JTFn4uQZbPiQJo4pf9RzJ

Through a specific tool (Google Cloud Storage key management tool) it is possible to create and manage up to 5 different developer keys.

2.2 Structure

2.2.1 Projects

All data in Google Cloud Storage belongs to a **Project**, which consists of a set of users, a set of APIs, and billing, authentication, and monitoring settings for those API. The main idea is that it is possible to create various project simultaneously, each one for a different purpose. For instance, it can be useful to create a project for the management of the clients informations of an hotel and another different project for the details of the hotel rooms. Each project will have its own data and own set of users.

2.2.2 Buckets and Objects

In Google Cloud Storage **Buckets** are the basic containers that hold data. Everything that it is stored in Google Cloud Storage must be contained in a bucket. Every bucket must have a unique name across the entire Google Cloud Storage namespace.

Objects are the individual pieces of data that it is possible to store in Google Cloud Storage. Objects have two components: object *data* and object *metadata*. The object data component is usually a file that it is stored in Google Cloud Storage. The object metadata component is a collection of name-value pairs that describe various object qualities. The size of an object or the date of creation are examples of metadata values. Object names can contain any combination of Unicode characters (UTF-8 encoded) less than 1024 bytes in length.

Objects are immutable, which means that an uploaded object cannot change throughout its storage lifetime. An object's storage lifetime is the time between successful object creation (upload) and successful object deletion. In practice, this means that it is not possible to make incremental changes to objects, such as append operations or truncate operations. However, it is possible to overwrite objects that are stored in Google Cloud Storage because an overwrite operation is in effect a delete object operation followed immediately by an upload object operation. So a single overwrite operation simply marks the end of one immutable object's lifetime and the beginning of a new immutable object's lifetime.

2.2.3 Hierarchy

Google Cloud Storage uses a **flat hierarchical structure** to store buckets and objects. All buckets reside in a single flat hierarchy (it is not possible to put buckets inside buckets), and all objects reside in a single flat hierarchy within a given bucket.

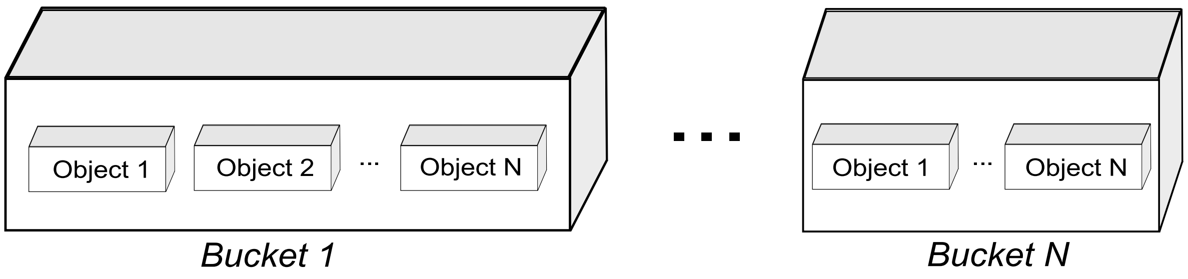


Figure 2.2: Buckets and Objects hierarchy

2.3 API Overview

The Google Cloud Storage API is a RESTful interface that permits to programmatically manage data on Google Cloud Storage. As a RESTful API, the Google Cloud Storage API relies on method information and scoping information to define the operations that can be performed [gog03]. The method information is specified with standard HTTP methods, such as DELETE, GET, HEAD, and PUT. And it is necessary to specify also the scoping information with a publicly-accessible endpoint (URI) and various scoping parameters. In this case, the primary scoping parameter is a path to a resource, which consists of an object name (file name) and bucket name (container name). Used together, the object name, bucket name, and public URI create a unique URL to a given resource.

All the HTTP requests in this section are referred to Google Cloud Storage version 1.0

2.3.1 HTTP Requests

Requests Authentication

All the HTTP requests to Google Cloud Storage must be authenticated. This is done populating the **Authorization** request header of each HTTP request in the following manner:

Authorization: GOOG1 google_storage_access_key:signature

GOOG1 is the **signature identifier** that identifies the signature algorithm and version to be used. The signature identifier for Google Cloud Storage is always GOOG1.

The *google_storage_access_key* is the **user identifier**, a 20 character access key that identifies the person who is making and signing the request. Google Cloud Storage uses the access key to look up the **secret key**, which is a pre-shared secret that only the owner of the account and the Google Cloud Storage system know (see 2.1.4).

The *signature* is a cryptographic hash function of various request headers. The signature is created by using **HMAC-SHA1** as the hash function and the secret key as the cryptographic key (HMAC-SHA1 is a hash-based message authentication code and requires two input parameters, both UTF-8 encoded: a key and a message). The resulting digest is then Base64 encoded.

SecretKey = UTF-8-Encoding-Of(GoogleStorageSecretKey)
Signature = Base64-Encoding-Of(HMAC-SHA1(SecretKey, MessageToBeSigned))

The *MessageToBeSigned* must be constructed by concatenating specific HTTP headers in a specific order. The following notation shows how to construct the message:

MessageToBeSigned = UTF-8-Encoding-Of(CanonicalHeaders + CanonicalExtensionHeaders + CanonicalResource)

CanonicalHeaders is constructed concatenating several header values and adding a newline (U+000A) after each header value. The following notation shows how to do this (newlines are represented by \n):

CanonicalHeaders =
HTTP-Verb + \n +
Content-MD5 + \n +
Content-Type + \n +
Date + \n

The **CanonicalExtensionHeaders** string is formed concatenating all extension (custom) headers that begin with **x-goog-**. In the development of the *iHD* application the CanonicalExtensionHeaders will be often an empty string.

The **CanonicalResource** string is constructed concatenating the resource path (bucket, object and subresource) on which the current request is acting.

Sample Request	**Sample Message To Sign**
PUT /cars/mercedes/classA.jpg HTTP/1.1 Host: mark.commondatastorage.googleapis.com Date: Mon, 20 Feb 2011 20:15:18 GMT Content-Length: 6597 Content-Type: image/jpg Authorization: GOOG1 GOOGT ... FzlAm9ts=	PUT\n \n image/jpg\n Mon, 20 Feb 2011 20:15:18 GMT\n /mark/cars/mercedes/classA.jpg

Table 2.1: Example of a request with its corresponding message to sign

GET Service

GET Service: A GET HTTP request that lists all buckets in a specified project. Request syntax:

```
GET / HTTP/1.1
Host: commondatastorage.googleapis.com
Date: date
Content-Length: 0
Authorization: authentication string
```

Response details: The request returns the list of buckets in an XML document in the response body.

PUT Bucket

PUT Bucket: A PUT HTTP request that creates a new bucket in a specified project. Request syntax:

```
PUT / HTTP/1.1
Host: bucket.commondatastorage.googleapis.com
Date: date
Content-Length: 0
Authorization: authentication string
```

Response details: The response does not include an XML document in the response body.

GET Bucket

GET Bucket: A GET HTTP request that lists the objects that are in a bucket. Request syntax:

```
GET / HTTP/1.1
Host: bucket.commondatastorage.googleapis.com
Date: date
Content-Length: 0
Authorization: authentication string
```

Response details: The request returns the list of objects in an XML document in the response body. Google Cloud Storage does not return lists longer than 1000 objects.

DELETE Bucket

DELETE Bucket: A DELETE HTTP request that deletes an *empty bucket*. This means that if a bucket it is not empty it must be freed before to call this method. Request syntax:

```
DELETE / HTTP/1.1
Host: bucket.commondatastorage.googleapis.com
Date: date
Content-Length: 0
Authorization: authentication string
```

Response details: The response does not include an XML document in the response body.

GET Object

GET Object: A GET HTTP request that permits to download a file. The GET request is scoped to a bucket and object. Request syntax:

```
GET object HTTP/1.1
Host: bucket.commondatastorage.googleapis.com
Date: date
Content-Length: request body length
Authorization: authentication string
```

Response details: The binary data of the file to download.
Request example:

```
GET /friends.doc HTTP/1.1
Host: mark.commondatastorage.googleapis.com
Date: Tue, 15 Feb 2011 15:16:05 GMT
Content-Length: 1500
Authorization: GOOG1 GOOGT ... FzlAm9ts=
```

PUT Object

PUT Object: A PUT HTTP request that uploads or copies an object. The PUT request is scoped with a bucket name and an object's name, and the **object data is inserted into the request body**. Request syntax:

```
PUT /object HTTP/1.1
Host: bucket.commondatastorage.googleapis.com
Date: date
Content-Length: request body length
Content-Type: object MIME type
Content-MD5: object MD5 digest – OPTIONAL
Authorization: authentication string


Request body: ... object data ...
```

Response details: The response does not include an XML document in the response body.
Request example:

```
PUT /mountains.jpg HTTP/1.1
Host: carlos.commondatastorage.googleapis.com
Date: Sat, 11 Feb 2010 12:34:18 GMT
Content-Type: image/jpg
Content-MD5: iB94gawbwUSiZy5FuruIOQ==
Content-Length: 890
Authorization: GOOG1 GOOGT ... FzlAm9ts=
```

DELETE Object

DELETE Object: A DELETE HTTP request that deletes an object. The DELETE method is a DELETE request with bucket and object scope. Request syntax:

```
DELETE /object HTTP/1.1
Host: bucket.commondatastorage.googleapis.com
Date: date
Content-Length: request body length
Authorization: authentication string
```

Response details: The response does not include an XML document in the response body.

HEAD Object

HEAD Object: A HEAD HTTP request that lists metadata for an object. Request syntax:

```
HEAD /object HTTP/1.1
Host: bucket.commondatastorage.googleapis.com
Date: date
Content-Length: 0
Authorization: authentication string
```

Response details: The request can return a variety of response headers depending on the request headers you use. The response body does not include an XML document. The following table shows all the response headers of an HEAD request:

Header	**Description**
Cache-Control	A request and response header that specifies the cache control setting
Content-Length	The length (in bytes) of the request or response body.
Content-Type	The MIME type of the request or response.
Content-Disposition	A request and response header that specifies presentational information about the data being transmitted.
ETag	The entity tag for the response body. Usually a valid MD5 digest.
Last-Modified	The date and time that the object was last modified.

Table 2.2: Response headers of an HEAD request

POST Object

POST Object: A POST HTTP request that uploads objects by **using HTML forms**. HTML form fields:

Field	**Description**
Content-Type	The MIME type of the file uploaded via the form.
file	The file to upload. Must be the last field in the form. It is possible to upload only one object per request.
key	The name of the object to upload.

This method it is not of interest for the development of the iHD application, so it will not further explained.

2.3.2 Resumable uploads

Google Cloud Storage provides a resumable uploads feature that permits to resume upload operations after a communication failure has interrupted the flow of data. Resumable uploads are useful when large file are transferred, because the likelihood of an error during the transmission is high. Furthermore, resuming upload operations can reduce the bandwidth usage (and costs) because it is not necessary to restart file uploads from the beginning. In this section it is shown how to implements the resumable upload feature using the Google Cloud Storage API.

Step 1–Initiate the resumable upload

The first step is to construct a POST request with an empty body, which must contains the following fields:

- A **Content-Type** request header, which contains the content type of the file to upload.
- A **Content-Length** request header, which must be set to 0.
- An **x-goog-resumable** header, which must be set to **start**.

The following example shows a simple POST request that initiate the resumable upload:

```
POST /mozart.mp3 HTTP/1.1
Host: mark.commondatastorage.googleapis.com
Date: Mon, 10 Dec 2011 18:54:10 GMT
Content-Length: 0
Content-Type: audio/mpeg
x-goog-resumable: start
Authorization: GOOG1 GOOGTS ... xs98GTER=
```

Step 2–Process the response

The Google Cloud Storage system will then responds with a 201 Created status message. The response contains also a **Location** header which defines an *upload ID* for the resumeable upload.

The following example shows the response message of the request in the step 1:

```
HTTP/1.1 201 Created
Location: https://mark.commondatastorage.googleapis.com/mozart.mp3/upload_id=thE0RdYnhDa...hJJElop
Date: Mon, 10 Dec 2011 18:54:10 GMT
Content-Length: 0
Content-Type: audio/mpeg
```

Step 3–Upload the file

Once the *upload ID* has been obtained, the next step is to make a PUT request which include the upload ID parameter. In this case the Content-Length header must be set to the dimension of the file to upload.

The following example shows the PUT request for the file mozart.mp3 initiated in step 1:

```
PUT /mozart.mp3?upload_id=thE0RdYnhDa...hJJElop HTTP/1.1
Host: mark.commondatastorage.googleapis.com
Date: Mon, 10 Dec 2011 18:54:10 GMT
Content-Length: 8809876
Authorization: GOOG1 GOOGiT ... leqiklJHl=
```

If the PUT request is not interrupted and the file is completely uploaded, Google Cloud Storage responds with a 200 OK status code. Otherwise, the upload can be resumed performing the steps 4, 5 and 6.

Step 4–Query Google Cloud Storage for the upload status

If the upload operation has been interrupted, it is possible to query Google Cloud Storage for the number of bytes it has received by implementing another PUT request. This PUT request must have an empty body, the upload ID obtained in step 2, and the following headers:

- A **Content-Length** request header, which must be set to 0.
- A **Content-Range** request header, which must be in the following format:

  *Content-Range: */content-length*

  Where content-length is the value of the Content-Length header specified in the PUT request of step 3.

The following example shows how to query the Google Cloud Storage for the interrupted upload of step 3:

```
PUT /mozart.mp3?upload_id=thE0RdYnhDa...hJJElop HTTP/1.1
Host: mark.commondatastorage.googleapis.com
Date: Mon, 10 Dec 2011 18:56:11 GMT
Content-Range: bytes */8809876
Content-Length: 0
Authorization: GOOG1 GOOGT ... grYiUh=
```

Step 5–Process the status response

The system will then respond with a 308 Resume Incomplete status code to the PUT request of step 4. This response contains a **Range** response header, which indicates the amount of bytes the Google Cloud Storage system has received.

The following example shows the response of the PUT request of step 4:

```
HTTP/1.1 308 Resume Incomplete
Range: bytes=0-3826739
Date: Mon, 10 Dec 2011 18:56:11 GMT
Content-Length: 0
Content-Type: audio/mpeg
```

Step 6–Resume the upload

Finally, the upload operation can be resumed implementing a last PUT request. The body of this request must contains the portion of the file not previously uploaded. This portion is calculated subtracting the Range value (obtained in step 5) from the Content-Length (specified in step 3). The PUT request must also contains the *upload ID* of the resumeable upload and the following headers:

- A **Content-Length** request header, which is setted to:

  Content-Length_of_step_3 - Range

- A **Content-Range** request header, which specifies the range of bytes to be uploaded.

The following example shows the PUT request that resumes the upload of the file interrupted in step 3:

```
PUT /mozart.mp3?upload_id=thE0RdYnhDa...hJJElop HTTP/1.1
Host: mark.commondatastorage.googleapis.com
Date: Mon, 10 Dec 2011 18:56:11 GMT
Content-Range: bytes 3826739-8809875/8809876
Content-Length: 4983137
Authorization: GOOG1 GOOGT ... lkYoUg=
```

The steps 4, 5 and 6 can be performed as many times as necessary.

Chapter 3

Design

3.1 A statement of the problem

The goal is to develop a complete application, named **iHD**, of Cloud Storage for the access, management and sharing of files on the web. The end user will simply use the application from any type of device with an internet connection without caring about the storage location of the files. This storage location is provided by the Google Cloud Storage service, which offers a simple programming interface to store and manage data. Through this service, the client application will provide a simple and practical GUI interface to maintain the users data always available “in the Cloud”.

3.2 Requirements analysis

In this section will be analysed all the important characteristics on which the iHD application will be based. These characteristics can be viewed like a starting point from which begin the development of all the various parts of the software.

3.2.1 Functional requirements

Functional requirements define the specific behaviour and functions of an application. The following list specifies the functional requirements for the iHD application:

- The application must permit the download of multiple files simultaneously.
- The application must permit the upload of multiple files simultaneously.
- The application must permit the inclusion of multiple accounts on the same device.
- The application must permit to define a local path, called *workspace*, where to save all the downloaded files.

- The application must permit to share files and folders among the users (registered to the service).
- The application must show a progress bar that indicates the amount of used space for each user.
- The application must show both the traffic in download and in upload measured in Kb/s.
- The application must permits to switch from an user to another without the need of restart.

3.2.2 Non-Functional requirements

Non-functional requirement impose constraints on the design or implementation of an application. The following list defines the non-functional requirements for the iHD application:

- The application shall exhibit a main single interface from which it is possible to access to all the implemented functions.
- The application shall notify an accidental network error and, in this case, it shall interrupt temporarily all the operations.
- The application shall resume the interrupted uploads from the state at which they were arrived.
- The application shall permit to modify some settings like the port numbers, IP addresses, etc.
- All the file transfers shall be secured with SSL.
- All the sensitive data of users saved locally on the device shall be protected using the MD5 algorithm.
- The software shall be portable on all the systems that run a JVM (Java Virtual Machine).

3.2.3 Constraints

The following list defines some constraints related to the development of the iHD application:

- The programming language for the development must be Java.

- The system must be interfaced with the Google Cloud Storage service.
- The RDBMS (relational database management system) must be MySQL.

3.3 Design choices

The iHD application will be divided into macro *components*. Each component will do a single major job. Following this methodology the final structure of the application will be better understandable and the development of the software will be easier.

3.3.1 General overview

The system as a whole is composed by three major parts:

- The *iHD application* developed in Java, distributed from a website on which new users can also crate an account.
- The *MySQL database* that contains the informations of all the registered users.
- The *Google Cloud Storage service* on which the files will be saved.

The following figure shows these three major parts:

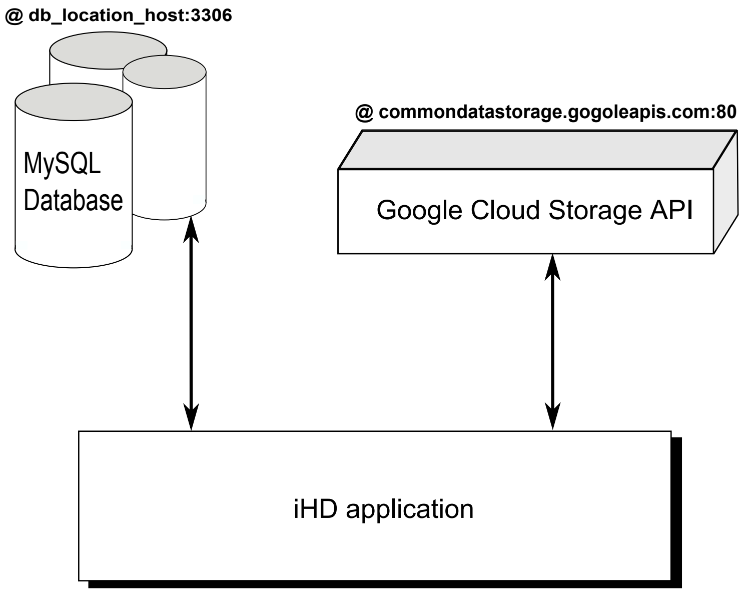


Figure 3.1: iHD application – scheme of communication

As already stated, on the Google Cloud Storage service will be saved the data of different users, so it is necessary to subdivide the available space on Google Cloud Storage among many users. This is done assigning a different bucket to each user, a bucket named with the same username of the correspondent user[1]. For example, if an user called "Steven" subscribes to the service, a new bucket named *Steven* will be created, and all the data of the user Steven will be saved in this bucket. This design choice increments also the security of the iHD application, because a given user will never be able to access the data of another user (saved in fact in another bucket). The following figure shows the buckets management on Google Cloud Storage for the iHD application:

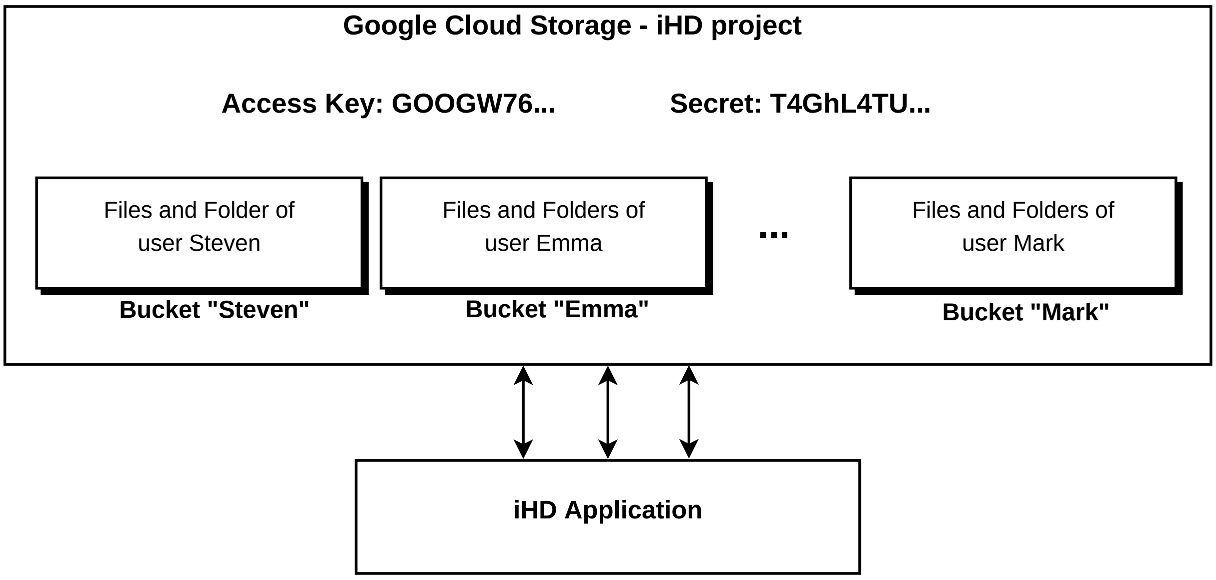

Google Cloud Storage - iHD project
Access Key: GOOGW76...
Secret: T4GhL4TU...
Files and Folder of user Steven
Files and Folders of user Emma
...
Files and Folders of user Mark
Bucket "Steven"
Bucket "Emma"
Bucket "Mark"
iHD Application

Figure 3.2: Buckets management on Google Cloud Storage

Regarding the database, it will contain all the registered users to the iHD service with their username, password and other relevant informations such as the amount of space they are currently using. In more detail, the database will be accessed by the iHD application:

- During login operations, to check the username and password
- To update the amount of used space (in KB) on Google Cloud Storage for a given user. This information changes after upload or delete operations.
- To update the current plan of an user (a plan indicates how many MB's are available for a given user)

[1]Actually Google Cloud Storage provides a single bucket namespace for all its projects, so it could be better to associate each username with a generated bucket name. This feature will be implemented in the next versions of the iHD application.

3.3.2 The Connection Controller

The **Connection Controller** is a *thread* and start its execution immediately after the call of the iHD application. Its job consists to check the connection every x seconds and to notify the application whenever the connection is interrupted. This thread will be useful to interrupt the downloads and uploads on every connection break.

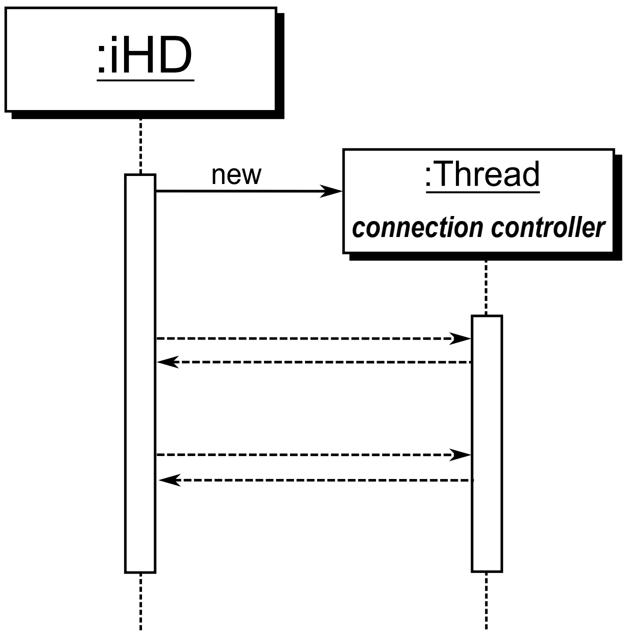


Figure 3.3: The Connection Controller

3.3.3 The Upload Manager

The **Upload Manager** is a component that initiates and manages all the uploads to Google Cloud Storage. Bringing the management of the uploads on this component will simplify the structure of the application. The upload manager is not a thread.

3.3.4 The Download Manager

Like its counterpart, the **Download Manager** is a component the initiates and manages all the downloads from Google Cloud Storage. It is not a thread.

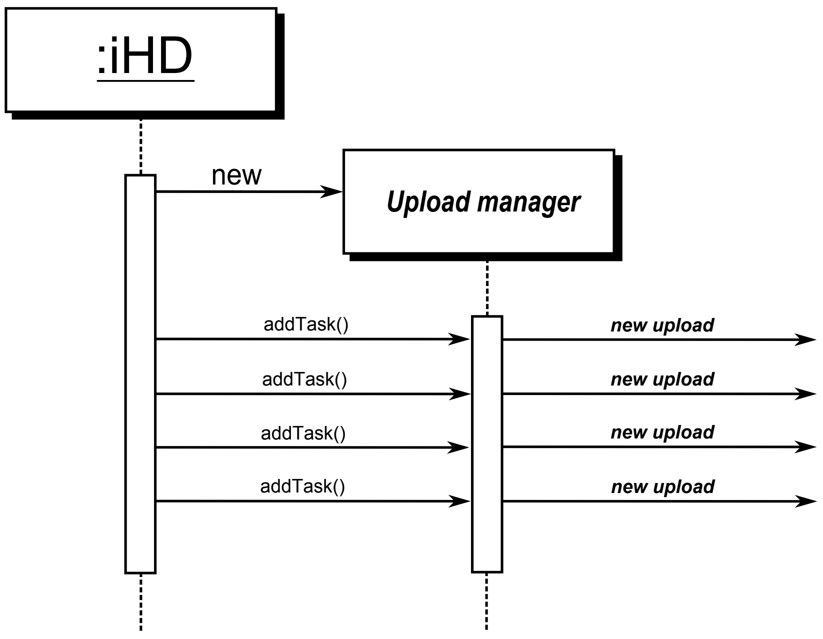


Figure 3.4: The Upload Manager

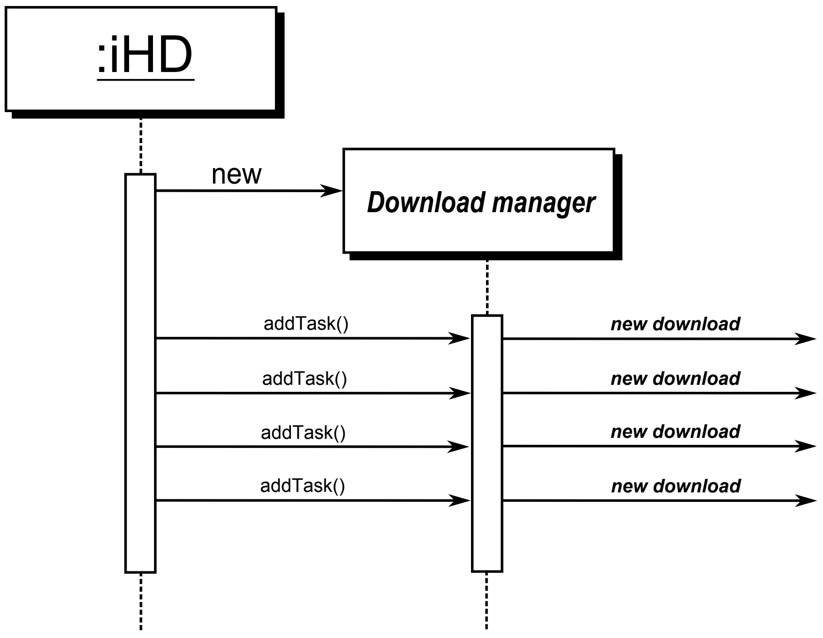


Figure 3.5: The Download Manager

3.3.5 The Database Gateway

The **Database Gateway** is a component that executes all the queries to the MySQL database. It can be viewed like an interface on which any other component must communicate in order to retrieve informations from the database.

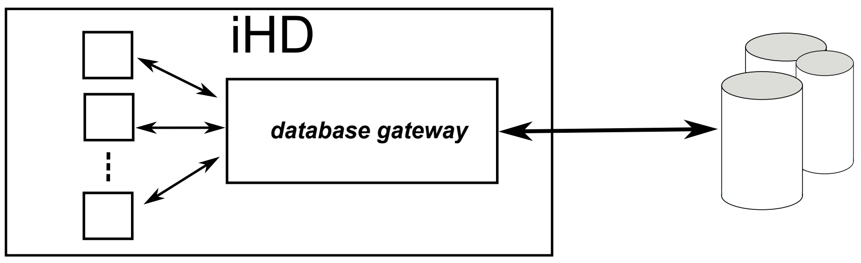


Figure 3.6: The Database Gateway

3.3.6 The Google Storage Gateway

The **Google Storage Gateway** is the component that executes all the HTTP requests to the Google Cloud Storage API. Omitting the *PUT Object* and *GET Object requests* (that are implemented by the Upload and Download manager, respectively), this component implements the following requests:

- PUT Bucket
- GET Bucket
- DELETE Bucket
- DELETE Object
- HEAD Object

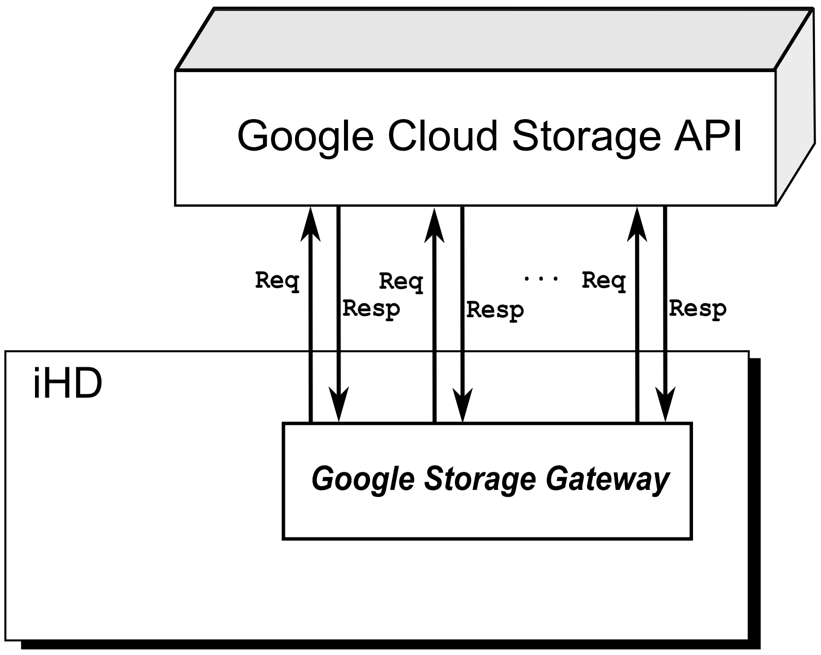


Figure 3.7: The Google Storage Gateway

3.4 The database

The database has been designed following three simple principles:

- **Simplicity**: Only the necessary tables and fields have been implemented, omitting to insert unnecessary or rare values that can be added in future.
- **Atomicity**: The number of fields for each table has been reduced to the minimum. Each field contains only one concept.
- **Compactness**: The number of NULL values has been reduced to the minimum and therefore the opportunity for inconsistency is lower.

The following ER diagram represent the iHD database:

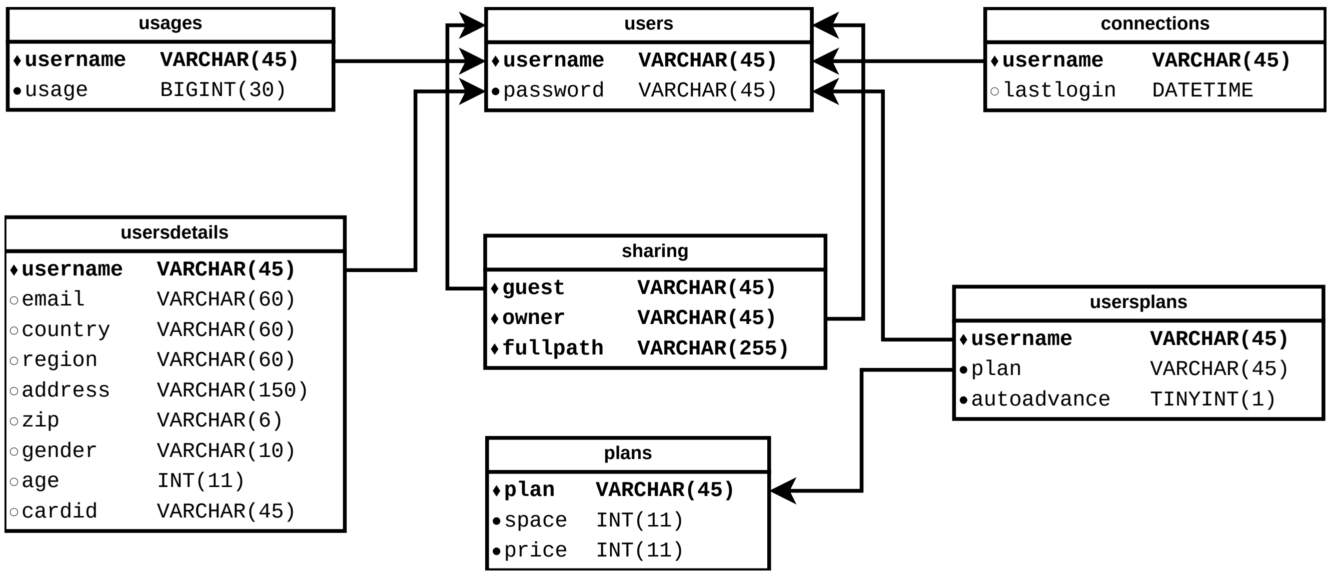


Figure 3.8: The Database ER diagram

users: This table contains all the users with their password (saved through the MD5 algorithm).

usages: This table indicates the amount of space that each user is currently using.

connections: This table indicates for each user when occurred the last login.

plans: This table contains all the available plans with their respective price. A basic free plan of 200MB is offered for the startup of the iHD application.

usersplans: This table associates to each user an available plan from the *plans* table. The *autoadvance* field indicates whether it is possible to automatically pass to the next available plan (with the respective change of the price).

usersdetails: This table contains all the information of the registered users.

sharing: This table permits to share folders among the users. The field *guest* is the invited user, the field *owner* indicates the holder of the folder, while the field *fullpath* represents the path to the folder. In chapter 4 this mechanism will be better explained.

Chapter 4

Implementation

This chapter shows all the major parts of the *iHD* application and how these parts are implemented. The first part of the chapter is focused on the issues regarding the environment on which the application has been developed, while the second part explains how many components of the application are effectively implemented.

4.1 Development Environment and choice of the programming language

In chapter 3 we defined all the functional and not-functional requirements of the iHD application. Among these requirements there are: (I) advanced network capabilities (to enable data transfers over the HTTP protocol), (II) file management (to save users data locally), (III) database management (to communicate with the MySQL database), (IV) multi-threading architecture (to allow the application to accomplish multiple tasks at the same time. An example could be a task that checks the underlying network periodically), (V) portability (availability on different platforms), (VI) security capabilities (to encrypt all the data that transfer over the network and the informations in the database), and (VII) graphic functionalities (the application should exhibit a graphical user interface). It is clear that is necessary to choose a programming language that provides all these capabilities and that is designed to develop large projects. **Java** is surely the best choice. It provides all the necessary libraries to develop the characteristics mentioned above and is an excellent language for developing cross-platform desktop applications. The following list provides the strengths of Java that will be useful to the development of the iHD application:

- Vast array of third-party libraries
- Huge amount of documentation available

- Managed memory
- Native threads
- Excellent performance
- OOP capability
- Portability - it runs on almost every platform
- Well-supported
- Flexibility - does graphics, desktop GUIs and web user interfaces

Regarding the development environment the choice has fallen to **Eclipse**. It is an open-source development platform comprised of extensible frameworks, tools and run-times for building, deploying and managing software across its life-cycle. In particular it provides a software development environment comprising an integrated development environment (IDE) and an extensible plug-in system.

4.2 Libraries

The following list shows the most important java libraries (packages) imported in the iHD application:

- **java.io**: Provides the classes to manage data streams, runtime exceptions and files.
- **java.net/javax.net**: Provides the classes for implementing networking applications. The *URL* and *HttpsURLConnection* are two examples of classes that contain method used to connect with the Google Cloud Storage server.
- **java.awt/javax.swing**: A set of components used for creating user interfaces and for painting graphics.
- **org.w3c/org.xml/javax.xml**: Contains methods and classes used to *parse* the XML responses retrieved from the Google Cloud Storage server (for example the list of buckets or the list of files in a specific bucket).
- **java.util**: Contains useful classes able to handle dates, formats and arrays.
- **java.sql/com.mysql**: All the classes necessary to execute queries to the MySQL databases.

- **Fast MD5 implementation**: It is an open-source library[1] originally written by *Santeri Paavolainen* that implements the MD5 algorithm.

4.3 Classes

Before to show all the classes that compose the iHD application, it is necessary to define the package hierarchy on which these classes are contained. The following figure shows this hierarchy:

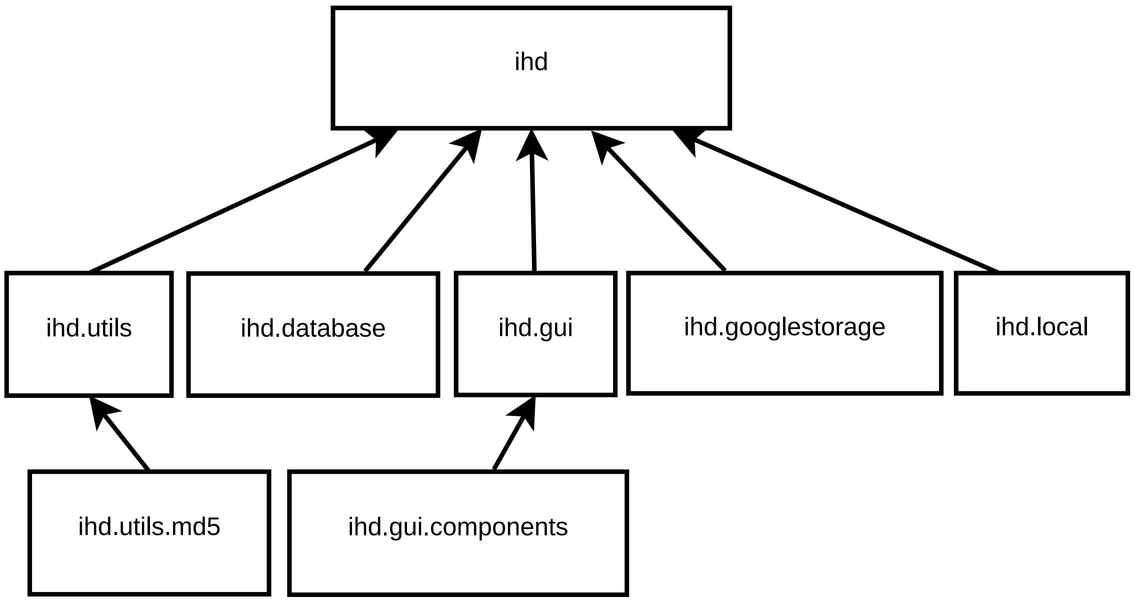


Figure 4.1: The Package Hierarchy

The following list defines for each package all its contained classes:

- **Package ihd**:

  Class **iHD**: This is the main class that contains the main() method. It instantiates the TopInterface class from the ihd.gui package.

- **Package ihd.gui**:

  Class **AboutDialog**: Create a simple window that display all the informations about the iHD application.

  Class **AccountsDialog**: A modal dialog on which it is possible to add/remove/modify an account of the iHD application.

[1]To see more details about the *Fast MD5 implementation* library follow this link: `http://www.twmacinta.com/myjava/fast_md5.php`

Class **ConfigurationDialog**: Whenever the user needs to add a new account to the iHD application (for example through the AccountsDialog class) this modal dialog is shown. It permits to insert all the important informations of the new account and finally save it locally.

Class **TopInterface**: Displays the main window of the iHD application. Through this window it is possible to access all the functionalities of the application.

- **Package ihd.gui.components**:

  Class **FileElement**: Each file is represented through this class. It shows an icon, the name of the file, its size and when it was last modified.

  Class **FileProgressElement**: Every upload or download operation of a file is shown with this class. It contains a progress bar that displays a percentage from 0 to 100. The progress bar is colored with blue for the upload operations and with green for the downloads.

  Class **FileScrollerPanel**: This class creates an area on which is possible to drag&drop files into the application (starting a new upload operation of multiple files) and that displays all the files and folders (as a list). For example it will contain a list of FileElement objects.

  Class **FolderElement**: Each folder is represented through this class. It shows an icon, the name of the folder, the list of the users with which the folder is shared, an add button used to share the folder with other users, and a remove button used to remove the folder and all its content.

  Class **FooterPanel**: Creates an area that displays the state of the connection (connected to the internet or not), the current logged user (or not logged), and the amount of KB/s used in upload and in download.

  Class **HeaderPanel**: Shows a panel with a series of button useful to access many functionalities of the iHD application. In particular there is a download button (download the selected files), an upload button (upload a new file), a new-folder button (create folder), a refresh button (refresh the current folder), a remove button (remove the selected files), a progress bar (amount of used and available space for your account), and a search field (select the files to be shown according to a prefix).

  Class **NavigationPanel**: A panel that displays the sequence of the visited folders. Whenever a new folder is accessed the leaving folder is added to the sequence as the last element. Clicking on a previous folder in the sequence it is possible to going back in the tree.

  Class **NewFolderElement**: Displays a panel that contains an input field. Inserting the name of the folder in the input field and pressing Enter it is possible to create a new folder.

Class **SharingFolderElement**: Displays a panel with a special icon that represent the list of the folders that the other users are sharing with the current user. In order to show all the shared folders is necessary to click on this panel.

Class **WebFolderElement**: Each shared folder is represented through this class (shared means that the folder belongs to another user, but the current user can view it because the folder is shared with him). It shows an icon, the name of the folder and the list of the users with which the folder is shared.

- **Package ihd.databse**:

  Class **DatabaseGateway**: This class contains all the methods necessary to communicate with the iHD database.

- **Package ihd.local**:

  Class **CurrentState**: This class contains all the informations regarding the current behaviour of the iHD application. It contains: the current amount of downloading and uploading KB/s, the logging state (logged/not logged) and the connection state (connected to the network/not connected).

  Class **LocalSettings**: This class contains all the account available on the current machine. It provides many methods to add/remove/edit an account.

- **Package ihd.googlestorage**:

  Class **DownloadManager**: This class manages all the current downloads from the Google Cloud Storage server. Through its multi-threading structure is possible to handle multiple data streams at the same time.

  Class **UploadManager**: This class manages all the current uploads to the Google Cloud Storage server. Through its multi-threading structure is possible to handle multiple data streams at the same time. Whenever an upload is interrupted this class can resume it later without loss of data.

  Class **GoogleStorageGateway**: This class is an interface to the Google Cloud Storage server. It provides all the methods necessary to accomplish operations on the server like: remove file, get file length, get list of files, get list of folders, remove folder, create folder and check user existence.

  Class **GoogleStorageAuthorization**: All the operations to the Google Cloud Storage server require a special authorization string. This class produces for each request to the server a valid authorization string.

- **Package ihd.utils**:

  Class **CommonUtils**: This class provides many useful methods like: get the size of a file in a readable format given its size expressed in bytes, convert dates, get the type of a file given its format.

  Class **ConnectionController**: This class checks every x seconds if the network connection is available.

  Class **GlobalVars**: This is an abstract class (it is not possible to instantiate the class) that contains all the constants of the iHD application: database password, database url location, database user, database port number, Google Cloud Storage access key, Google Cloud Storage secret, Google Cloud Storage host and many informations regarding the speed of upload and download.

  Class **InfiniteProgressPanel**: Display over the main window of the application a loading image, obscuring the underlying components. It can be very useful during operations that take an large amount of time.

- **Package ihd.utils.md5**:

  Class **MD5**: This class is used to crypt a given string using the MD5 hashing algorithm. This package contains also other classes related to the MD5 algorithm but that are not used in the iHD application.

In the following sections many of the viewed classes will be better explained.

4.4 The connection controller

As described in the previous section the **Connection Controller** checks every x seconds if the network connection is available. In order to do this it tries to connect to a specific host (for instance, http://www.google.com) every *timeout* milliseconds, where timeout is a defined constant (e.g. 3000). The following code fragment shows how this is implemented in Java:

```
public class ConnectionController implements Runnable
{
  private boolean      running;

  // ...

  private Thread       t;

  // ...
  public void run()
  {
    while(running)
    {
      try {

        URL url = new URL("http://www.google.com");
        HttpURLConnection huc = (HttpURLConnection)url.openConnection();

        huc.connect();

        // If I arrived here the connect() worked -> connected

        // ...

        state.setConnected(true);

        huc.disconnect();

        try {
          Thread.sleep(timeout);
        } catch (InterruptedException e1) {}

      } catch (IOException e) {

        // huc.connect() not worked -> not connected & exception launched

        // ...

        state.setConnected(false);

        try {
          Thread.sleep(timeout);
        } catch (InterruptedException e1) {}
      }
    }
  }

  public void start()
  {
    t.start();
  }

  // ...
}
```

4.5 The google storage gateway

The **GoogleStorageGateway** class provides a set of methods that allow to accomplish operations on the server (create folder, delete file, ...) and to retrieve informations from the server (list of files, folders, ...). The following list provides all the methods implemented by the GoogleStorageGateway class:

- **public GoogleStorageGateway(String bucket)**: This is the constructor. It assigns a bucket name to this class.
- **public String getBucket()**: Returns the bucket name associated with this class.
- **public boolean checkBucket()**: Returns *true* if the bucket exists on the server (Google Cloud Storage server).
- **public int getFileLength(String filename, String path)**: Returns the size expressed in bytes of a specific file on the server.
- **public int deleteFile(String filename, String path)**: Deletes a specific file on the server.
- **public int removeFolder(String folderPath)**: Removes a specific folder from the server.
- **public int createFolder(String folderName, String path)**: Creates a folder on the server.
- **public ArrayList<String> getFiles(String folderPath)**: Returns the list of files in a specific folder on the server.
- **public ArrayList<String> getFolders(String folderPath)**: Returns the list of folders contained a given folder.

As an example, the following well-commented code shows how the deleteFile() method is implemented:

```
public class GoogleStorageGateway
{
  private String bucket;

  public GoogleStorageGateway(String bucket)
  {
    this.bucket=bucket;
  }
```

```
// ...

public int deleteFile(String filename, String path) throws IOException
{
  int responseCode;

  // Remove all the white-spaces from the path
  // because the HTTP request that we are going to
  // construct does not admit white-spaces.
  path=path.replaceAll("␣", "%20");
  filename=filename.replaceAll("␣", "%20");

  URL myURL=null;

  // Construct the URL with which it is possible to
  // communicate with Google Cloud Storage.

  if(path.matches("/"))
    myURL = new URL("https://"+this.bucket+"."+(GlobalVars.googleHost)
        + "/" + filename);
  else
    myURL = new URL("https://"+this.bucket+"."
        + (GlobalVars.googleHost)
        + path +filename);

  // Connect to the Google Storage server
  HttpsURLConnection conn = (HttpsURLConnection)myURL.openConnection();

  // Our connection can transmit/receive data
  conn.setDoInput(true);
  conn.setDoOutput(true);
  conn.setUseCaches(false);

  // To delete a file use the DELETE HTTP method
  conn.setRequestMethod("DELETE");

  conn.setRequestProperty("Host",this.bucket + "."
                    + (GlobalVars.googleHost));

  // Obtain the current date
  //
  final Date currentTime = new Date();
  final SimpleDateFormat dateFormat =
        new SimpleDateFormat("EEE,␣d␣MMM␣yyyy␣HH:mm:ss␣z");

  dateFormat.setTimeZone(TimeZone.getTimeZone("GMT"));

  // Normalize the date to a standardized model accepted by Google
```

```
    conn.setRequestProperty("Date",dateFormat.format(currentTime));

    // This class it is used to create our authorization string
    // necessary to authorize the HTTP request
    // that we are sendind to Google
    //
    GoogleStorageAuthorization auth = new GoogleStorageAuthorization(
            GlobalVars.accessKey,
            GlobalVars.secret,
            dateFormat.format(currentTime));

    // Set all important information needed to the DELETE request in
    // the GoogleStorageAuthorization class
    //
    auth.setMethod("DELETE");
    auth.setBucketName(this.bucket);
    if(!path.matches("/"))
      auth.setPath(path);
    auth.setFileName(filename);

    // Given the authorization, add it to our request to Google Storage
    conn.setRequestProperty("Authorization",auth.getAuthorizationString());

    // The content-length for a DELETE request is 0
    conn.setRequestProperty("Content-Length","0");

    // Finally send the request to the server (that will delete the file)
    conn.connect();

    // Get the response code by the server
    responseCode=conn.getResponseCode();

    conn.disconnect();

    // Return the response code of the DELETE method
    // to the caller
    return responseCode;
  }
}
```

4.6 The database gateway

The **DatabaseGateway** class provides all the methods necessary to communicate with the MySQL database that contains the informations of the iHD users. In particular, the following methods are provided:

- **public boolean exists(String username)**: Returns *true* if a given username is contained in the database.

- **public boolean checkCorrectness(String username, String password)**: Check if a pair username-password is correct.
- **public int getAvailableStorage(String username)**: Returns the amount of available storage (the maximum amount of space an user can utilize, given by his plan) for a particular user.
- **public int getUsedStorage(String username))**: Returns the amount of non-available storage (used space) for a particular user.
- **public int updateUsedStorage(String username, int amount)**: This method updates the amount of used storage for an user with a quantity equal to *amount* (that can be positive or negative).
- **public boolean shareFolder(String owner, String guest, String fullPath)**: This method permits to the user *owner* to give the access of one of his folders (defined by *fullPath*) to another user *guest*.
- **public boolean removeWebFolder(String guest, String fullPath)**: With this method the user *guest* can remove one of the folders (defined by *fullPath*) that other users are sharing with him.
- **public boolean removeGuestContact(String owner, String guest, String fullPath)**: This method permits to the user *owner* to deny the access of one of his shared folders (defined by *fullPath*) to another user *guest* (which previously obtained the access to the folder through the shareFolder() method).
- **public ArrayList<String> getGuestContacts(String owner, String fullPath)**: Returns the list of users that have the access to a shared folder of the user *owner* (shared folder defined by *fullPath*).
- **public ArrayList<String> getWebFolders(String guest)**: Returns the list of the folders shared with the user *guest*.

It is clear that all the sharing mechanism is implemented through the MySQL database, using the methods shareFolder(), removeWebFolder(), removeGuestContact(), getGuestContacts() and getWebFolders(). Another important characteristic is the management of the users plans. This is done, as can be clearly noticed, through the database, using the methods getAvailableStorage(), getUsedStorage() and updateUsedStorage().

Appendix B shows the full implementation of the DatabaseGateway class. As an example, the following code shows the implementation of the getGuestContacts() method:

```
public class DatabaseGateway
{
  // ...

  public ArrayList<String> getGuestContacts(String owner, String fullPath)
  {
    Connection conn         = null;
    MysqlDataSource dataSource  = new MysqlDataSource();
    ArrayList<String> retList = new ArrayList<String>();

    try
    {
      dataSource.setServerName(GlobalVars.dbLocation);
      dataSource.setPortNumber(GlobalVars.dbPort);
      dataSource.setDatabaseName(GlobalVars.dbName);
      dataSource.setUser(GlobalVars.dbUser);
      dataSource.setPassword(GlobalVars.dbPassword);

      conn = dataSource.getConnection();

      if(conn.isValid(0))
      {
        Statement stat = conn.createStatement();
        String mysqlQuery="SELECT guest " +
                   "FROM sharing " +
                   "WHERE owner='" +
                   owner +
                   "' AND fullpath='" +
                   fullPath + "'";

        ResultSet res=stat.executeQuery(mysqlQuery);

        while(res.next())
          retList.add(res.getString("guest"));

        stat.close();
      }

      conn.close();
    }
    catch(Exception e) {}

    return retList;
  }
}
```

4.7 SSL security

All the files that are transmitted or received from the Google Cloud Storage server must be encrypted in some way. This is done using the SSL (Secure Sockets Layer) protocol that provides communication security over the Internet. The SSL protocol encrypts the segments of network connections above the Transport Layer, using asymmetric cryptography for key exchange, symmetric encryption for privacy, and message authentication codes for message integrity. In Java, as shows the following upload example, this is done with the **HttpsURLConnection** class:

```
// Create a new HttpsURLConnection object
// All the data transmitted through this object will pass under an
// SSL tunnel
//
HttpsURLConnection conn = (HttpsURLConnection)myURL.openConnection();

  // ...
  // Set many properties of the conn object like Content-Length, Host,
  // the request method, etc ...

  conn.setRequestMethod("PUT");
  conn.setRequestProperty("Host",googleHost);
  conn.setRequestProperty("Content-Length",fileLenght);

    // ...

  // ******** Transmit data ********

    // Retrieve an output stream from the connection where to write data
    output = conn.getOutputStream();

    while(/* Data available to write */)
    {
      output.write(data[i]);

      writedBits++;
    }

  // ********     *******     ********

  // Finally close the output stream and the connection
  output.close();
  conn.disconnect();
```

4.8 Multi-threading implementation

In order to permit the upload and download of multiple files at the same time the iHD application must implement a dynamic array (that can expand in size at runtime) of threads. Actually the implemented arrays of threads are two, the first one for the UploadManager class and the second one for the DownloadManager class. Every time a new upload (download) is initiated, a special class called Uploader (Downloader) that implements the Runnable interface is instantiated, and the new created object is added to the respective array. This object will manage the upload (download) transmission, and when its work will be completed the object will be removed from the array. It is important to notice that every object of this type is itself a thread so can follow its execution in parallel with the execution of the iHD application, without stopping the other current operations. This mechanism allows the iHD application to execute multiple downloads and uploads at the same time, as shows the following image:

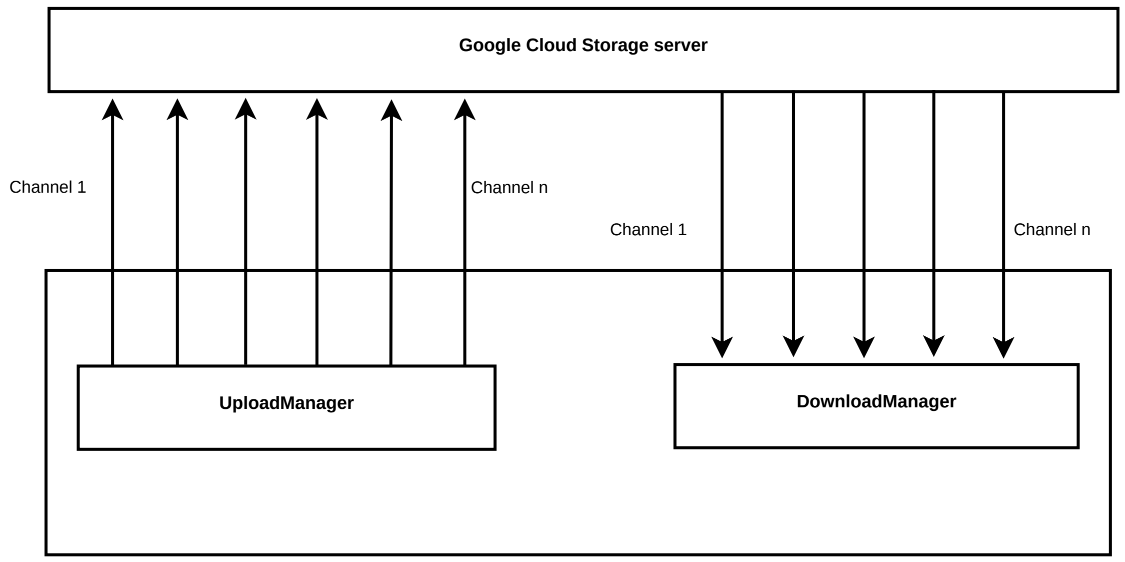


Figure 4.2: Multi-threading implementation of uploads and downloads

4.8.1 The download manager

The **DownloadManager** class handles all the downloads from the Google Cloud Storage server (where the files of the users are located). It provides a single method addTask() to initiate a new download. After this method is called a new thread is executed and the transmission of the file can take place. The following Java code (simplified, and without many parts replaced by a “...”) shows the general behaviour of the DownloadManager class:

```
public class DownloadManager
{
  private ArrayList<Downloader> downloads;

  public DownloadManager()
  {
    downloads = new ArrayList<Downloader>();
  }

  public void addTask(String fileName,
            String filePath,
            String user,
             ...    ) throws IOException
  {
    downloads.add(0,new Downloader(fileName,
                       filePath,
                       user, ... ));
    downloads.get(0).start();
  }

  private class Downloader implements Runnable
  {
    private Thread t;

    private boolean      downloading;

    // ...

    public Downloader(String fileName,
              String filePath,
              String user,
                ...           ) throws IOException
    {
      downloading = true;

      // ...

      t = new Thread(this);

      this.fileName=fileName;
      this.filePath=filePath;
      this.user=user;

      // ...
      this.filePath=this.filePath.replaceAll("␣", "%20");
      this.fileName=this.fileName.replaceAll("␣", "%20");

      // ...
    }

    public void start()
    {
      t.start();
    }

    public boolean isCompleted()
    {
      return !downloading;
    }

    public void run()
    {
      // ...

      try{
        // ...

        HttpsURLConnection conn =
            (HttpsURLConnection)myURL.openConnection();

        // ....

        conn.setRequestMethod("GET");

        // ...

        conn.connect();
        responseCode=conn.getResponseCode();

        if(responseCode==200)
        {
          InputStream in = conn.getInputStream();

          FileOutputStream fileOutput = new FileOutputStream(file);
          OutputStream out = new DataOutputStream(fileOutput);

          // ...

          while((i=in.read())!=-1)
          {
            out.write(i);

            readedBits++;
```

```
            // ...
          }

          // ...

          conn.disconnect();
        }
      } catch(IOException e1) {}

      this.downloading=false;
    }

    // ...
  }
}
```

4.8.2 The upload manager

The **UploadManager** class, like his counterpart, handles all the uploads to the Google Cloud Storage server. It provides a single method addTask() to initiate a new upload. After this method is called a new thread is executed and the transmission of the file can take place. Appendix A shows the full implementation of the UploadManager class. The following Java code (simplified, and without many parts replaced by a **"..."**) shows the general behaviour of the UploadManager class:

```
public class UploadManager
{
  private ArrayList<Uploader> uploads;

  public UploadManager()
  {
    uploads=new ArrayList<Uploader>();
  }

  public void addTask(String fileName,
           String filePath,
           String user,
             ...   ) throws IOException
  {
    uploads.add(0,new Uploader(fileName,
                   filePath,
                   user,
                     ...   ));
    uploads.get(0).start();
  }

  private class Uploader implements Runnable
  {
    private Thread t;

    private Checker      checker;

    private boolean      uploading;
    private boolean      restoring;

    private String       fileName;
    private String       filePath;
    private String       user;

    // ...

    public Uploader(String fileName,
              String filePath,
              String user,
                ...    ) throws IOException
    {
      uploading = true;
      restoring = false;
      writedBits  = 0;

      t = new Thread(this);

      checker = new Checker();

      this.fileName=fileName;
      this.filePath=filePath;
      this.user=user;

      // ...
    }

    public void start()
    {
      t.start();
    }

    public void run()
    {
      // ...

      try{
        // ...
```

```
HttpsURLConnection conn =
  (HttpsURLConnection)myURL.openConnection();

// ...
/******* RESUMABLE UPLOADS - STEP 1 *******/

conn.setRequestMethod("POST");

// ...

conn.setRequestProperty("x-goog-resumable","start");
conn.setRequestProperty("Content-Length","0");

// ...

conn.connect();
responseCode=conn.getResponseCode();

if(responseCode==201)
{

  /******* RESUMABLE UPLOADS - STEP 2 *******/

  uploadId = conn.getHeaderField("Location");

  // ...

  conn.disconnect();

  // ...

  conn = (HttpsURLConnection)myURL.openConnection();

  // ...

  /******* RESUMABLE UPLOADS - STEP 3 *******/

  conn.setRequestMethod("PUT");

  // ...

  conn.connect();

  output = conn.getOutputStream();

  FileInputStream fileInput = new FileInputStream(file);
  InputStream in = new DataInputStream(fileInput);
```

```
            // ...

            checker.start();

            while((i=in.read())!=-1)
            {
              output.write(i);

              writedBits++;

              // ...
            }

            // ...

            checker.stop();

            output.close();
            conn.disconnect();

            this.uploading = false;

          }
        } catch(IOException e1) { ... }
      }

      // ...

      private void restore()
      {
        restoring = true;

        try {
          output.close();
        } catch (IOException e) {}

        // ...

        int responseCode = 0;

        while(responseCode!=308)
        {
          try {
            // ...

            HttpsURLConnection conn =
              (HttpsURLConnection)myURL.openConnection();

            // ...
            /******* RESUMABLE UPLOADS - STEP 4 *******/

            conn.setRequestMethod("PUT");

            // ...

            conn.connect();

            responseCode = conn.getResponseCode();

            /******* RESUMABLE UPLOADS - STEP 5 *******/

            if(responseCode == 308)
              rangeHeader = conn.getHeaderField("Range");

            conn.disconnect();

          } catch(IOException e1) {   ...   }

          // ...
        }

        if(responseCode == 308)
          resume(   ...   );
      }

      private void resume(int alreadyWrited)
      {
        try {
          // ...

          HttpsURLConnection conn =
            (HttpsURLConnection)myURL.openConnection();

          // ...

          /******* RESUMABLE UPLOADS - STEP 6 *******/

          conn.setRequestMethod("PUT");

          // ...

          conn.connect();

          output = conn.getOutputStream();

          FileInputStream fileInput = new FileInputStream(file);
          InputStream in = new DataInputStream(fileInput);
          // ...

          writedBits = alreadyWrited;

          in.skip(alreadyWrited);

          // ...

          restoring = false;

          while((i=in.read())!=-1)
          {
            output.write(i);

            writedBits++;

            // ...
          }

          // ...

          checker.stop();

          output.close();
          conn.disconnect();

          uploading = false;

        } catch(IOException e1) {  ...  }
      }

      private class Checker implements Runnable
      {
        private Thread subT;

        private int lastWrited;

        private boolean running = true;

        // ...

        public Checker()
        {
          subT = new Thread(this);

          lastWrited = 0;
        }
```

```
        public void start()
        {
          subT.start();
        }

        public void stop()
        {
          this.running = false;
        }

        public void run()
        {
          while(running)
          {
            Thread.sleep(timeout);

            if(!restoring && running)
            {
              if(lastWrited >= getWritedBits())
                restore();
              else
                lastWrited = getWritedBits();
            }
          }
        }
      }

      // ...
    }
  }
```

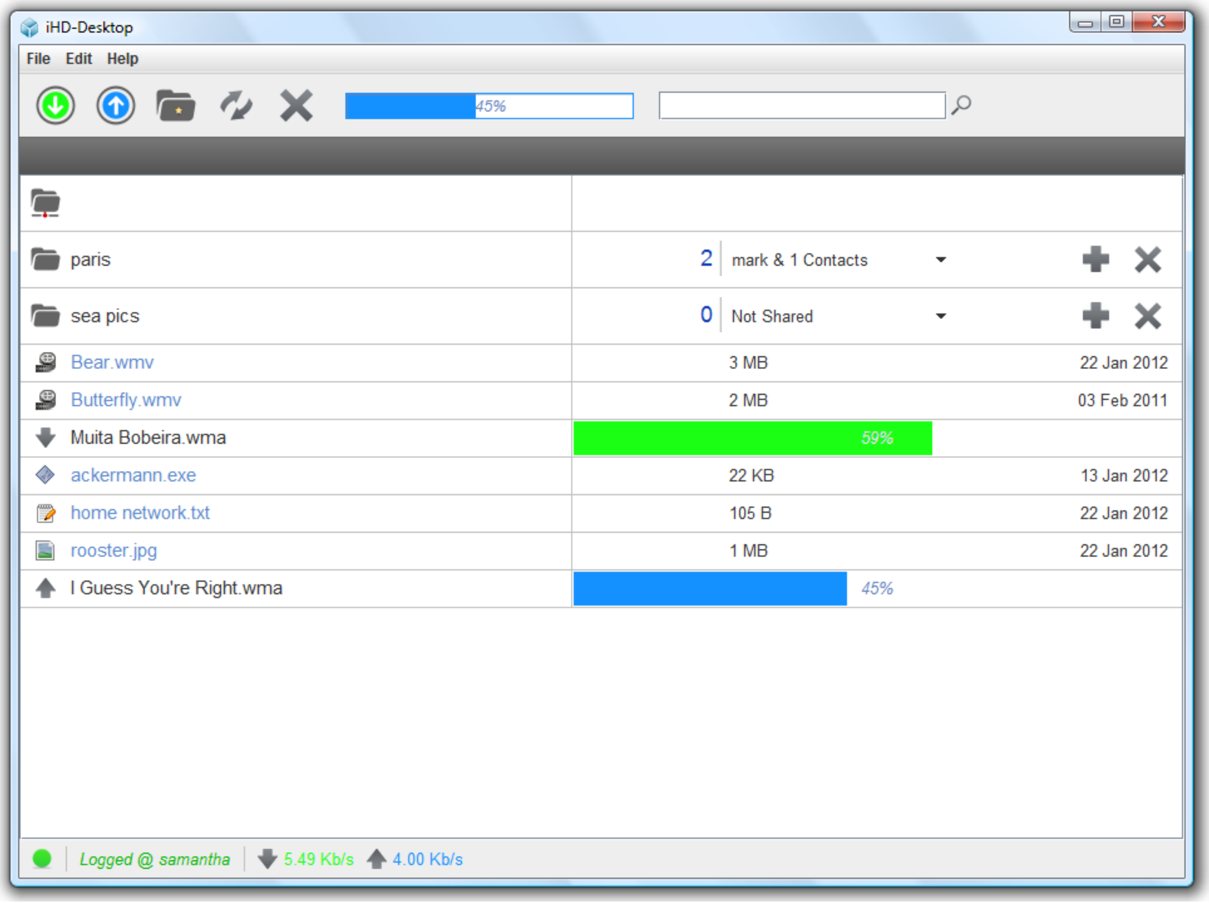
Resumable uploads implementation

As can be seen in code above, the UploadManager class implements the *resumable uploads* (RU) mechanism described in chapter 2 (section 2.3.2). In more detail, this job is done by the Uploader class contained in the UploaderManager, which actually handles the upload operation. Firstly, it makes a POST request in order to initiate the resumable upload (RU - STEP 1), and after that it processes the consequent response to retrieve the *Location* response header (RU - STEP 2). Consequently, it begins to upload the file implementing a PUT request (RU - STEP 3), and if nothing goes wrong the Uploader class finishes his job. Otherwise, a special class that implement the Runnable interface (a thread) called *Checker* notices that the transmission is interrupted and therefore makes a call to a *restore()* method. The restore() method waits for the underlying network connection to be available and then iteratively tries to implement the PUT request to retrieve the interrupted upload status (RU - STEP 4). Once the response to this request is received (it should be a 308 resume-incomplete response. If not, the PUT request will be resubmitted), the *Range* response header is retrieved and a *resume()* method can be called (RU - STEP 5). Finally, the resume() method implements the PUT request (RU - STEP 6) the resumes the upload from the point at which had been interrupted (information given by the *Range* response header). Obviously, the upload operation can be newly interrupted and the restore process can be reinitiated from step 4 (again through the Checker class).

Chapter 5

iHD application - GUI Description

5.1 The Main Window



paris	2	mark & 1 Contacts	
sea pics	0	Not Shared	
Bear.wmv	3 MB		22 Jan 2012
Butterfly.wmv	2 MB		03 Feb 2011
Muita Bobeira.wma	59%		
ackermann.exe	22 KB		13 Jan 2012
home network.txt	105 B		22 Jan 2012
rooster.jpg	1 MB		22 Jan 2012
I Guess You're Right.wma	45%		

Logged @ samantha | 5.49 Kb/s | 4.00 Kb/s

Figure 5.1: The Main Window

5.1.1 Menu bar

File

Name	Description
Exit	Close the software.

Edit

Name	Description
Select All	Select all the files in the main panel.
Configuration Assistant	Shows a dialog that permits to add a new account.
Accounts	Shows the accounts dialog.

Help

Name	Description
Help	Information about how to use the software.
About	Information about the software.

5.1.2 Header panel

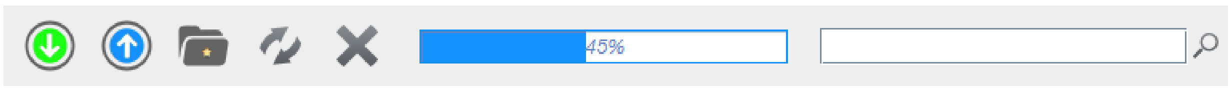


Figure 5.2: Header panel

A panel with a series of components (in order, from left to right):

- A download button (download all the selected files)
- An upload button (upload a new file)
- A new-folder button (create folder)
- A refresh button (refresh the current folder)

- A remove button (remove the selected files)
- A progress bar (percentage of utilized space)
- An input field (shows only the files that begin with a given prefix)

5.1.3 Navigation panel

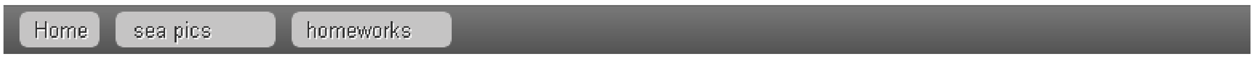


Figure 5.3: Navigation panel

A panel that displays the sequence of the visited folders. Whenever a new folder is accessed the leaving folder is added to the sequence as the last element. Clicking on a previous folder in the sequence it is possible to going back in the tree. Referring to the figure, the current path is "/sea pics/homeworks/**current_folder**" (The "Home" button represents the root).

5.1.4 Scroller panel

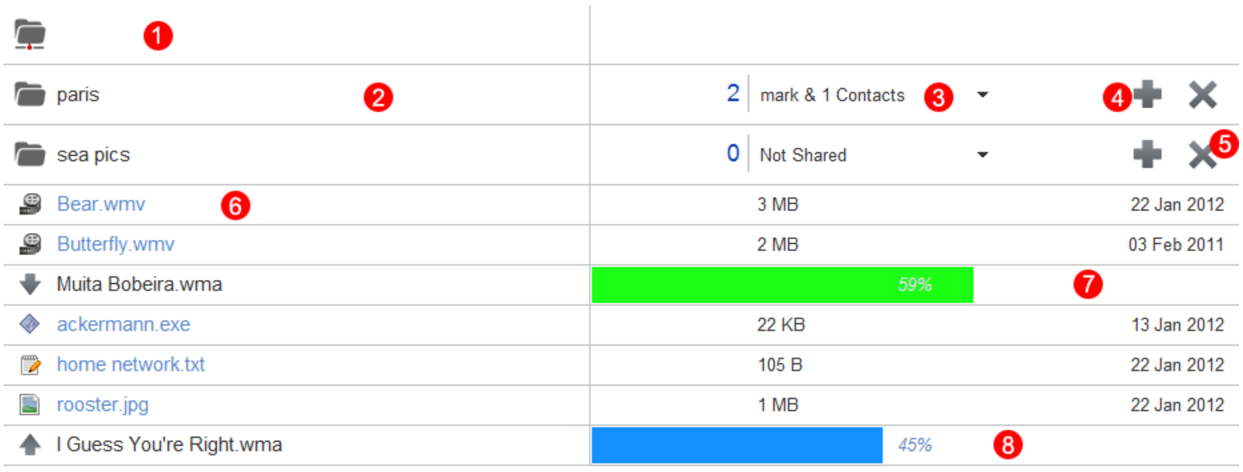


Figure 5.4: Scroller panel

An area on which is possible to drag&drop files into the application (starting a new upload operation of multiple files) and that displays all the files and folders (as a list). In more detail, this area contains:

- **(1) The sharing folder**
  This element is only visible when the current path is the root. Clicking on it the area will be populated with all the shared folders which the user can access.

- **(2) Folders**
  Each folder shows an icon, the name of the folder, the list of the users with which the folder is shared **(3)**, an add button used to share the folder with other users **(4)**, and a remove button used to remove the folder and all its content **(5)**.

- **(6) Files**
  Each file shows an icon, the name of the file, its size and when it was last modified.

- **(7) File downloads**
  Each download is represented with a green progress bar that displays a percentage from 0 to 100.

- **(8) File uploads**
  Each upload is represented with a blue progress bar that displays a percentage from 0 to 100.

5.1.5 Footer panel

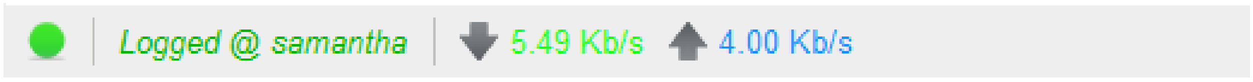
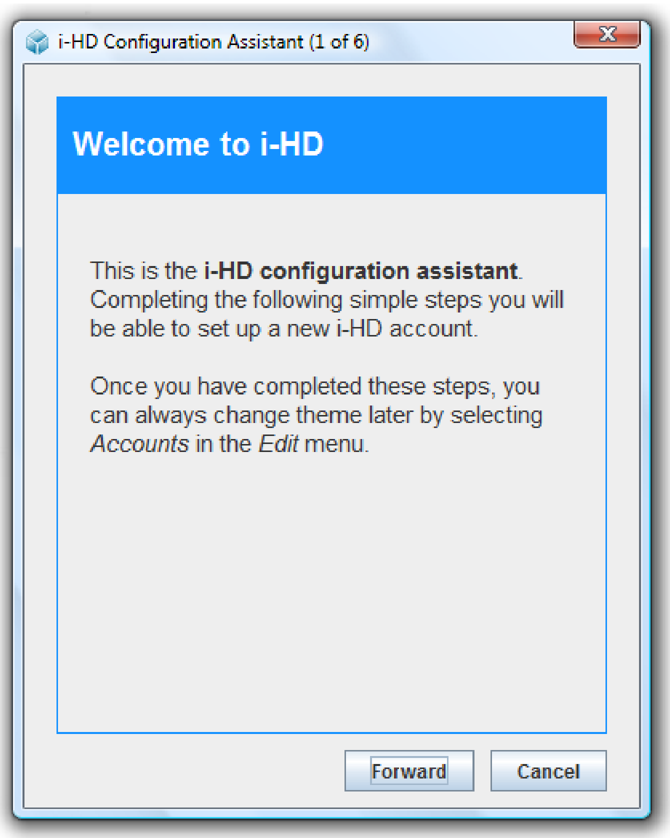


Figure 5.5: Footer panel

An area that displays (in order, from left to right) the state of the connection (connected to the internet or not), the current logged user (or not logged), and the current amount of Kb/s used in download and in upload (respectively).

5.2 The Configuration Assistant Dialog



This is the **i-HD configuration assistant**. Completing the following simple steps you will be able to set up a new i-HD account.

Once you have completed these steps, you can always change theme later by selecting *Accounts* in the *Edit* menu.

Figure 5.6: The Configuration Assistant Dialog

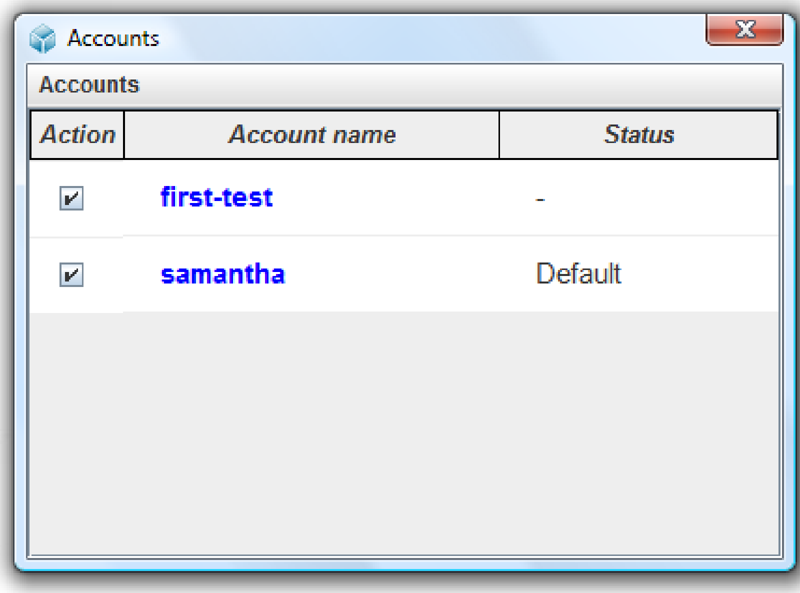
The Configuration Assistant dialog can be called from the Edit menu. It permits to insert all the important informations of a new account and finally save it locally. In particular it shows progressively six steps:

- Step 1: A brief description of the Configuration Assistant behaviour.
- Step 2: Insert the username.
- Step 3: Insert the password.
- Step 4: Insert the workspace, a path that indicated where to download the files from the Google Cloud Storage server.
- Step 5: A checkbox. If enabled it indicates that the auto-advance mode for the current account is activated (auto-advance: whenever the limit space quota is

reached the system automatically upgrades the user plan to the next one in order to increase the available space).

- Step 6: A summary of all the insert informations and a confirmation button.

5.3 The Accounts Dialog



Action	Account name	Status
✓	first-test	-
✓	samantha	Default

Figure 5.7: The Accounts Dialog

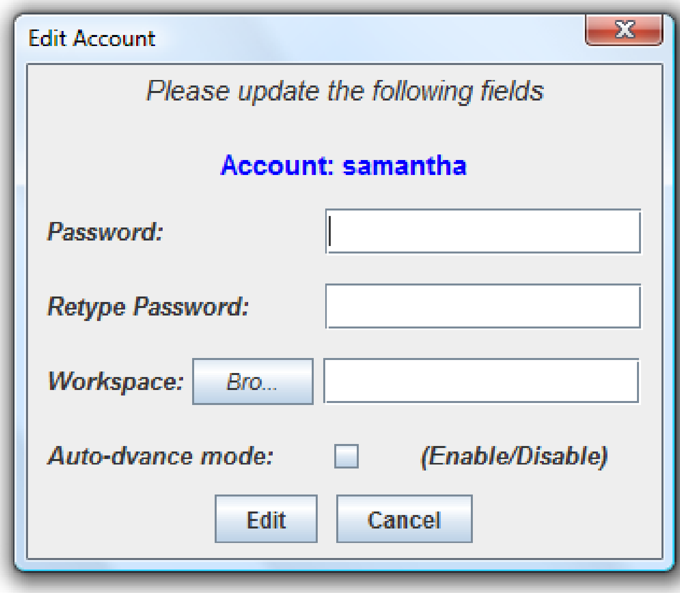
A modal dialog on which it is possible to add/remove/modify an account of the iHD application.

5.3.1 Menu bar

Accounts

Name	Description
Add an account	Shows the Configuration Assistant dialog in order to add a new account.
Edit account	Shows an edit dialog on which it is possible to modify the username, password, workspace and auto-advance mode for a given account.
Remove accounts	Remove the selected accounts.
Close	Closes the Accounts dialog.

5.3.2 Edit Dialog


Edit Account

Please update the following fields

Account: samantha

Password:

Retype Password:

Workspace: Bro...

Auto-dvance mode: ☐ (Enable/Disable)

Edit Cancel

Figure 5.8: The Edit Dialog

An edit dialog on which it is possible to modify the username, password, workspace and auto-advance mode for a given account. It can be selected from the Accounts menu.

Chapter 6

Conclusion

6.1 Thesis summary

This thesis introduced Google Cloud Storage: a service that permits to store data on the Google's infrastructure. It provides a simple programming interface to create applications that store, share, and manage data on Google Cloud Storage. In particular, as seen in chapter 2, the communication between applications and Google Cloud Storage is made using HTTP requests. Each of these requests encapsulates a method information (the type of the request, such as PUT, GET, DELETE, ...) and a scoping information (the resource on which perform the request). It follows that it becomes possible to develop an application that, relying on this API, provides a cloud storage service. This has been done implementing the iHD application, a client that permits to upload and manage files in the Cloud, through the Google Cloud Storage service. About that, chapter 3 showed the design principles on which the iHD application has been developed focusing on the requirements analysis and on the structure of the application. Chapter 4 instead provided an overview of all the components of the iHD application, showing in many cases how these components are effectively implemented in Java. Finally, chapter 5 described the GUI of the iHD application, providing for each element that compose the interface a brief description of its behaviour.

6.2 Future Work

There are a number of features that can be certainly added to the iHD application, among these features there are:

- Interoperability with other services of cloud storage such ad Amazon S3, that provides a simple RESTful programming interface like Google Cloud Storage. In this case, the end-user could choose for each account or for each folder where to save the data. This decision could also be taken by the iHD application in order to balance the storage between different services.

- A mechanism to download entire folders. This is not yet possible through the Google Cloud Storage API but it could be possible to develop it in Java.
- Capacity to rename the folders on the server. This is not yet possible through the Google Cloud Storage API but it could be possible to develop this feature with a series of delete/upload operations.
- Possibility to synchronize a given folder on the local operating system (for instance the workspace folder) with the files on the Google Cloud Storage server. In this case any change on this local folder will lead to a new upload (or delete) operation on the server.
- On the most advanced operating systems, an icon that permits to minimize the iHD application to the System Tray in order to run it in background.
- Possibility to set bandwidth limits. In this way the iHD application will not slow down the entire connection.
- A mechanism to make a specific file on the server completely public, which means that it can be accessed through a generated static URL from every user on the internet. This feature could be implemented quickly because this is an option already available with the Google Cloud Storage service.

6.3 Acknowledgements

Firstly, I gratefully acknowledge my supervisor Danilo Montesi for for his guidance throughout my thesis.

A ton of thanks go out to my family, particularly my mother and father for having believed in me and for putting up with my insistence on exploring the unknown instead of settling down.

I would like to thank all the students of the Computer Science department, and Francesco Orrù, for the many stimulating discussions and enjoyable friendships over the past three years.

Finally I'd like to acknowledge my best friend Andrea Franzoni and his lovely girlfriend for their support.

Appendix A

The UploadManager Class

Below there is the complete implementation of the UploadManager class. The DownloadManager will not be shown because is very similar to the UploadManager.

```
public class UploadManager
{
  private ArrayList<Uploader> uploads;
  private ArrayList<PermissionToken> permissions;

  public UploadManager()
  {
    uploads=new ArrayList<Uploader>();
    permissions=new ArrayList<PermissionToken>();
  }

  public void addTask(String fileName,
            String filePath,
            String localAbsoluteFilePath,
            FileProgressElement fileProgress,
            String user,
            CurrentState currState) throws IOException
  {
    for(int i=0;i<permissions.size();i++)
      permissions.get(i).incrementIndex();

    permissions.add(0, new PermissionToken(fileName, filePath, 0));

    uploads.add(0,new Uploader(fileName,
                  filePath,
                  localAbsoluteFilePath,
                  fileProgress,
                  user,
                  currState,
                  permissions.get(0)));
    uploads.get(0).start();
  }

  public void removeTask(String fileName, String filePath)
  {
    for(int i=0;i<permissions.size();i++)
      if(permissions.get(i).getName().compareTo(fileName)==0 &&
        permissions.get(i).getPath().compareTo(filePath)==0)
      {
        permissions.get(i).removePermission();

        uploads.remove(permissions.get(i).getIndex());

        for(int j=i;j<permissions.size();j++)
          permissions.get(j).decrementIndex();

        permissions.remove(i);

        break;
      }
  }

  private class Uploader implements Runnable
  {
    private Thread t;

    private Checker       checker;

    private boolean       uploading;
    private boolean       restoring;

    public String       fileName;
    public String       filePath;
    private String        user;
    private int         fileLenght;

    private String        uploadId;

    private int         writedBits;
    private OutputStream   output;

    private File          file;
    private FileProgressElement  fileProgress;

    private CurrentState   currState;

    private Timer       time;
    private PermissionToken myPermission;

    public Uploader(String fileName,
             String filePath,
             String localAbsoluteFilePath,
             FileProgressElement fileProgress,
             String user,
             CurrentState currState,
             PermissionToken myPermission) throws IOException
    {
      uploading = true;
      restoring = false;
      writedBits  = 0;

      t = new Thread(this);

      checker = new Checker();

      this.fileName=fileName;
      this.filePath=filePath;
      this.fileProgress=fileProgress;
      this.user=user;

      this.uploadId = new String();

      this.currState = currState;

      this.filePath=filePath.replaceAll(" ", "%20");
      this.fileName=fileName.replaceAll(" ", "%20");

      this.myPermission = myPermission;

      file=new File(localAbsoluteFilePath);

      if(file.exists())
        this.fileLenght=(int)file.length();
    }

    public void start()
    {
      t.start();
    }

    @SuppressWarnings("unused")
    public boolean isCompleted()
    {
      return !uploading;
    }
```

```
@SuppressWarnings("unused")
public String getName()
{
  return this.fileName;
}

@SuppressWarnings("unused")
public String getPath()
{
  return this.filePath;
}

@Override
public void run()
{
  int i;
  int responseCode;

  try {
    URL myURL=null;

    if(filePath.matches("/"))
      myURL = new URL("https://"+user+"."+(GlobalVars.googleHost)
          + "/" + this.fileName);
    else
      myURL = new URL("https://"+user+"."
          + (GlobalVars.googleHost) + this.filePath + this.fileName);

    HttpsURLConnection conn =
      (HttpsURLConnection)myURL.openConnection();

    conn.setDoInput(true);
    conn.setDoOutput(true);
    conn.setUseCaches(false);

    conn.setChunkedStreamingMode(GlobalVars.uploadTransitionWindow);
    conn.setRequestMethod("POST");

    conn.setRequestProperty("Host",user+"."+(GlobalVars.googleHost));

    Date currentTime = new Date();
    SimpleDateFormat dateFormat = new SimpleDateFormat(
      "EEE,␣d␣MMM␣yyyy␣HH:mm:ss␣z",new Locale("en","US"));

    dateFormat.setTimeZone(TimeZone.getTimeZone("GMT"));

    conn.setRequestProperty("Date",dateFormat.format(currentTime));

    conn.setRequestProperty("x-goog-resumable","start");
```

```
        conn.setRequestProperty("Content-Length","0");

        conn.setRequestProperty("Content-Type",
        new MimetypesFileTypeMap().getContentType(this.file));

        GoogleStorageAuthorization auth = new GoogleStorageAuthorization(
            GlobalVars.accessKey,
            GlobalVars.secret,
            dateFormat.format(currentTime));

        auth.setMethod("POST");
        auth.setBucketName(user);
        if(!filePath.matches("/"))
          auth.setPath(filePath);
        auth.setFileName(fileName);
        auth.setContentType(
          new MimetypesFileTypeMap().getContentType(this.file));
        auth.setGoogCustomHeader("x-goog-resumable:start");

        conn.setRequestProperty("Authorization",
                    auth.getAuthorizationString());

        conn.connect();
        responseCode=conn.getResponseCode();

        if(responseCode==201)
        {
          uploadId = conn.getHeaderField("Location");
          uploadId = uploadId.substring(
                uploadId.lastIndexOf("?upload_id=")+11);

          conn.disconnect();

          if(filePath.matches("/"))
            myURL = new URL("https://"+user+"."+(GlobalVars.googleHost)
                + "/" + this.fileName
                + "?upload_id=" + uploadId);
          else
            myURL = new URL("https://"+user+"."
                + (GlobalVars.googleHost)
                + this.filePath + this.fileName
                + "?upload_id=" + uploadId);

          conn = (HttpsURLConnection)myURL.openConnection();

          conn.setDoInput(true);
          conn.setDoOutput(true);
          conn.setUseCaches(false);
          conn.setChunkedStreamingMode(GlobalVars.uploadTransitionWindow);
          conn.setRequestMethod("PUT");
          conn.setRequestProperty("Host",user+"."+(GlobalVars.googleHost));

          conn.setRequestProperty("Date",dateFormat.format(currentTime));

          auth = new GoogleStorageAuthorization(
              GlobalVars.accessKey,
              GlobalVars.secret,
              dateFormat.format(currentTime));

          auth.setMethod("PUT");
          auth.setBucketName(user);
          if(!filePath.matches("/"))
            auth.setPath(filePath);
          auth.setFileName(this.fileName);

          conn.setRequestProperty(
              "Authorization",auth.getAuthorizationString());

          conn.setRequestProperty("Content-Length",
              Integer.toString(this.fileLenght));

          conn.connect();

          output = conn.getOutputStream();

          FileInputStream fileInput = new FileInputStream(file);
          InputStream in = new DataInputStream(fileInput);

          float progress;
          time = new Timer();

          checker.start();
          time.start();

          while((i=in.read())!=-1)
          {
            output.write(i);

            writedBits++;

            if(!myPermission.getPermission())
            {
              this.uploading = false;

              in.close();
              fileInput.close();
              return;
            }

            if((writedBits%GlobalVars.uploadTransitionWindow)==0)
              output.flush();

            progress=(float)(writedBits/(float)fileLenght)*100.0f;
            fileProgress.setValue((int)progress);
          }

          time.stop();

          output.flush();

          checker.stop();

          output.close();
          conn.disconnect();

          this.uploading = false;

          removeTask(fileName, filePath);
        }
      } catch(IOException e1) { time.stop(); }
    }

    private int getWritedBits()
    {
      return this.writedBits;
    }

    private void restore()
    {
      String rangeHeader = null;

      restoring = true;

      try {
        output.close();
      } catch (IOException e) {}

      fileProgress.setInterrupted();

      try {
        Thread.sleep(500);
      } catch (InterruptedException e1) {}

      while(!currState.isLogged())
        while(!currState.isConnected())
```

```
        {
          try {
            Thread.sleep(2000);
          } catch (InterruptedException e1) {}
        }

      int responseCode = 0;

      while(responseCode!=308)
      {
        try {
          URL myURL=null;

          if(filePath.matches("/"))
            myURL = new URL("https://"+user+"."+(GlobalVars.googleHost)
                + "/" + this.fileName
                + "?upload_id=" + uploadId);
          else
            myURL = new URL("https://"+user+"."
                + (GlobalVars.googleHost)
                + this.filePath + this.fileName
                + "?upload_id=" + uploadId);

          HttpsURLConnection conn =
              (HttpsURLConnection)myURL.openConnection();

          conn.setDoInput(true);
          conn.setDoOutput(true);
          conn.setUseCaches(false);

          conn.setChunkedStreamingMode(GlobalVars.uploadTransitionWindow);
          conn.setRequestMethod("PUT");
          conn.setRequestProperty("Host",user+"."+(GlobalVars.googleHost));

          conn.setRequestProperty("Content-Length","0");
          conn.setRequestProperty("Content-Range","bytes␣*/"
            + Integer.toString(this.fileLenght));

          Date currentTime = new Date();
          SimpleDateFormat dateFormat = new SimpleDateFormat(
              "EEE,␣d␣MMM␣yyyy␣HH:mm:ss␣z",new Locale("en","US"));

          dateFormat.setTimeZone(TimeZone.getTimeZone("GMT"));

          conn.setRequestProperty("Date",dateFormat.format(currentTime));

          GoogleStorageAuthorization auth = new GoogleStorageAuthorization(
              GlobalVars.accessKey,
              GlobalVars.secret,
                dateFormat.format(currentTime));

          auth.setMethod("PUT");
          auth.setBucketName(user);
          if(!filePath.matches("/"))
            auth.setPath(filePath);
          auth.setFileName(this.fileName);

          conn.setRequestProperty("Authorization",
                auth.getAuthorizationString());

          conn.connect();

          responseCode = conn.getResponseCode();

          if(responseCode == 308)
            rangeHeader = conn.getHeaderField("Range");

          conn.disconnect();

        } catch(IOException e1) {
            try {
              Thread.sleep(2000);
            } catch (InterruptedException intExcp) {}
          }

        if(responseCode==400)
        {
          checker.stop();
          fileProgress.setBroken();
          restoring = false;
          uploading = false;
          break;
        }

        if(responseCode!=308)
        {
          try {
            Thread.sleep(3000);
          } catch (InterruptedException intExcp) {}
        }

        if(!myPermission.getPermission())
        {
          this.uploading = false;
          return;
        }
      }
      if(responseCode == 308)
        resume(Integer.parseInt(rangeHeader.substring(
                        rangeHeader.lastIndexOf("-")+1)));
    }

    private void resume(int alreadyWrited)
    {
      try {
        URL myURL = null;
        int i;

        if(filePath.matches("/"))
          myURL = new URL("https://"+user+"."+(GlobalVars.googleHost)
              + "/" + this.fileName
              + "?upload_id=" + uploadId);
        else
          myURL = new URL("https://"+user+"."
              + (GlobalVars.googleHost)
              + this.filePath + this.fileName
              + "?upload_id=" + uploadId);

        HttpsURLConnection conn =
          (HttpsURLConnection)myURL.openConnection();

        conn.setDoInput(true);
        conn.setDoOutput(true);
        conn.setUseCaches(false);

        conn.setChunkedStreamingMode(GlobalVars.uploadTransitionWindow);
        conn.setRequestMethod("PUT");
        conn.setRequestProperty("Host",user+"."+(GlobalVars.googleHost));

        Date currentTime = new Date();
        SimpleDateFormat dateFormat = new SimpleDateFormat(
            "EEE, d MMM yyyy HH:mm:ss z",new Locale("en","US"));

        dateFormat.setTimeZone(TimeZone.getTimeZone("GMT"));

        conn.setRequestProperty("Date",dateFormat.format(currentTime));;

        conn.setRequestProperty("Content-Length",
            Integer.toString(this.fileLenght-alreadyWrited));

        conn.setRequestProperty("Content-Range","bytes "+
                            (alreadyWrited+1) + "-"
                            + (this.fileLenght-1)
                            + "/" + this.fileLenght);

        GoogleStorageAuthorization auth = new GoogleStorageAuthorization(
            GlobalVars.accessKey,
            GlobalVars.secret,
            dateFormat.format(currentTime));

        auth.setMethod("PUT");
        auth.setBucketName(user);
        if(!filePath.matches("/"))
          auth.setPath(filePath);
        auth.setFileName(this.fileName);

        conn.setRequestProperty("Authorization",
                  auth.getAuthorizationString());

        conn.connect();

        output = conn.getOutputStream();

        FileInputStream fileInput = new FileInputStream(file);
        InputStream in = new DataInputStream(fileInput);

        float progress;
        time = new Timer();

        writedBits = alreadyWrited;

        in.skip(alreadyWrited);

        fileProgress.setActive();
        restoring = false;

        time.start();

        while((i=in.read())!=-1)
        {
          output.write(i);

          writedBits++;

          if(!myPermission.getPermission())
          {
            this.uploading = false;

            in.close();
            fileInput.close();

            return;
          }

          if((writedBits%GlobalVars.uploadTransitionWindow)==0)
```

```
              output.flush();

            progress=(float)(writedBits/(float)fileLenght)*100.0f;
            fileProgress.setValue((int)progress);
          }

          time.stop();

          output.flush();

          checker.stop();

          output.close();
          conn.disconnect();

          uploading = false;

          removeTask(fileName, filePath);

        } catch(IOException e1) { time.stop(); }
      }

      private class Checker implements Runnable
      {
        private Thread subT;

        private int lastWrited;
        private final int timeout = 9000;

        private boolean running = true;

        public Checker()
        {
          subT = new Thread(this);

          lastWrited = 0;
        }

        public void start()
        {
          subT.start();
        }

        public void stop()
        {
          this.running = false;
        }

        @Override
```

```
    public void run()
    {
      while(running)
      {
        try {
          Thread.sleep(timeout);
        } catch (InterruptedException e1) {}

        if(!restoring && running)
        {
          if(lastWrited >= getWritedBits())
            restore();
          else
            lastWrited = getWritedBits();
        }
      }
    }
  }

  private class Timer implements Runnable
  {
    private Thread timeT;

    private boolean running;

    private int lastWrited;

    public Timer()
    {
      timeT = new Thread(this);

      running    = true;
      lastWrited  = 0;
    }

    public void start()
    {
      timeT.start();
    }

    public void stop()
    {
      running = false;
    }

    @Override
    public void run()
    {
      while(running)
```

```
          {
            currState.addUpRate(writedBits-lastWrited);

            lastWrited = writedBits;

            try {
              Thread.sleep(1000);
            } catch (InterruptedException intExcp) {}
          }
        }
      }
    }

    private class PermissionToken
    {
      private String  fileName;
      private String  filePath;

      private int    index;

      private boolean permission;

      public PermissionToken(String fileName, String filePath, int index)
      {
        this.fileName = fileName;
        this.filePath = filePath;
        this.index = index;

        this.permission = true;
      }

      public void removePermission()
      {
        this.permission = false;
      }

      public boolean getPermission()
      {
        return this.permission;
      }

      public String getName()
      {
        return this.fileName;
      }

      public String getPath()
      {
        return this.filePath;
      }

      public int getIndex()
      {
        return this.index;
      }

      public void incrementIndex()
      {
        this.index++;
      }

      public void decrementIndex()
      {
        this.index--;
      }
    }
  }
```

Appendix B

The DatabaseGateway Class

This appendix shows the full implementation of the DatabaseGateway class.

```
public class DatabaseGateway
{
  public boolean exists(String username)
  {
    Connection conn          = null;
    MysqlDataSource dataSource   = new MysqlDataSource();
    boolean retCode          = false;

    try
    {
      dataSource.setServerName(GlobalVars.dbLocation);
      dataSource.setPortNumber(GlobalVars.dbPort);
      dataSource.setDatabaseName(GlobalVars.dbName);
      dataSource.setUser(GlobalVars.dbUser);
      dataSource.setPassword(GlobalVars.dbPassword);

      conn = dataSource.getConnection();

      if(conn.isValid(0))
      {
        Statement stat = conn.createStatement();
        String mysqlQuery="SELECT * " +
                 "FROM users " +
                 "WHERE username='" +
                 username + "'";

        ResultSet res=stat.executeQuery(mysqlQuery);

        if(res.next())
          retCode=true;
        stat.close();
      }

      conn.close();
    }
    catch(Exception e) {}

    return retCode;
  }

  public boolean checkCorrectness(String username, String password)
  {
    Connection conn          = null;
    MysqlDataSource dataSource  = new MysqlDataSource();
    boolean retCode          = false;

    try
    {
      dataSource.setServerName(GlobalVars.dbLocation);
      dataSource.setPortNumber(GlobalVars.dbPort);
      dataSource.setDatabaseName(GlobalVars.dbName);
      dataSource.setUser(GlobalVars.dbUser);
      dataSource.setPassword(GlobalVars.dbPassword);

      conn = dataSource.getConnection();

      if(conn.isValid(0))
      {
        Statement stat = conn.createStatement();
        String mysqlQuery="SELECT * " +
                   "FROM users " +
                   "WHERE username='" +
                   username +
                   "' AND passwd='" +
                   password + "'";

        ResultSet res=stat.executeQuery(mysqlQuery);

        if(res.next())
          retCode=true;

        stat.close();
      }

      conn.close();
    }
    catch(Exception e) {}

    return retCode;
```

```
}

public int getAvailableStorage(String username)
{
  Connection conn = null;
  MysqlDataSource dataSource = new MysqlDataSource();
  int amount=-1;

  try
  {
    dataSource.setServerName(GlobalVars.dbLocation);
    dataSource.setPortNumber(GlobalVars.dbPort);
    dataSource.setDatabaseName(GlobalVars.dbName);
    dataSource.setUser(GlobalVars.dbUser);
    dataSource.setPassword(GlobalVars.dbPassword);

    conn = dataSource.getConnection();

    if(conn.isValid(0))
    {
      Statement stat = conn.createStatement();
      String generatedQuery="SELECT␣p.space␣" +
                  "FROM␣plans␣as␣p,␣usersplans␣as␣u␣" +
                  "WHERE␣u.username='" +
                  username + "'␣" +
                  "AND␣u.plan=p.planname";

      ResultSet res=stat.executeQuery(generatedQuery);
      res.last();

      amount=res.getInt("space");

      stat.close();
    }

    conn.close();
  }
  catch(Exception e) {}

  return amount;
}

public int getUsedStorage(String username)
{
  Connection conn = null;
  MysqlDataSource dataSource = new MysqlDataSource();
  int amount=-1;

  try
```

```
    {
      dataSource.setServerName(GlobalVars.dbLocation);
      dataSource.setPortNumber(GlobalVars.dbPort);
      dataSource.setDatabaseName(GlobalVars.dbName);
      dataSource.setUser(GlobalVars.dbUser);
      dataSource.setPassword(GlobalVars.dbPassword);

      conn = dataSource.getConnection();

      if(conn.isValid(0))
      {
        Statement stat = conn.createStatement();
        String generatedQuery="SELECT u.usage " +
                    "FROM usages as u " +
                    "WHERE u.username='" +
                    username + "'";

        ResultSet res=stat.executeQuery(generatedQuery);
        res.last();

        amount=res.getInt("u.usage");

        stat.close();
      }

      conn.close();
    }
    catch(Exception e) {}

    return amount;
  }

  public int updateUsedStorage(String username, int amount)
  {
    Connection conn = null;
    MysqlDataSource dataSource = new MysqlDataSource();
    int newAmount=-1;

    try
    {
      dataSource.setServerName(GlobalVars.dbLocation);
      dataSource.setPortNumber(GlobalVars.dbPort);
      dataSource.setDatabaseName(GlobalVars.dbName);
      dataSource.setUser(GlobalVars.dbUser);
      dataSource.setPassword(GlobalVars.dbPassword);

      conn = dataSource.getConnection();

      if(conn.isValid(0))
      {
        Statement stat = conn.createStatement();
        String generatedQuery="SELECT␣u.usage␣" +
                    "FROM␣usages␣as␣u␣" +
                    "WHERE␣u.username='" +
                    username + "'";

        ResultSet res=stat.executeQuery(generatedQuery);
        res.last();

        newAmount=res.getInt("u.usage") + amount;

        generatedQuery="UPDATE␣usages␣as␣u␣" +
                  "SET␣u.usage=" +
                  newAmount + "␣" +
                  "WHERE␣u.username='" +
                  username + "'";

        stat.executeUpdate(generatedQuery);

        stat.close();
      }

      conn.close();
    }
    catch(Exception e) {}

    return newAmount;
  }

  public boolean shareFolder(String owner, String guest, String fullPath)
  {
    Connection conn = null;
    MysqlDataSource dataSource = new MysqlDataSource();
    boolean retVal = false;

    try
    {
      dataSource.setServerName(GlobalVars.dbLocation);
      dataSource.setPortNumber(GlobalVars.dbPort);
      dataSource.setDatabaseName(GlobalVars.dbName);
      dataSource.setUser(GlobalVars.dbUser);
      dataSource.setPassword(GlobalVars.dbPassword);

      conn = dataSource.getConnection();

      if(conn.isValid(0))
      {
        Statement stat = conn.createStatement();
        String mysqlQuery="SELECT * " +
                  "FROM sharing " +
                  "WHERE owner='" +
                  owner +
                  "' AND guest='" +
                  guest +
                  "' AND fullpath='" +
                  fullPath + "'";

        ResultSet res=stat.executeQuery(mysqlQuery);

        if(!res.next())
        {
          stat = conn.createStatement();
          String generatedQuery="INSERT INTO sharing (owner, guest, " +
                      "fullpath, approved) " +
                      "VALUES('" + owner + "'" +
                      ",'" + guest + "'" +
                      ",'" + fullPath + "'" +
                      ",0)";

          stat.executeUpdate(generatedQuery);

          stat.close();

          retVal = true;
        }
      }

      conn.close();
    }
    catch(Exception e) {}

    return retVal;
  }

  public boolean removeWebContacts(String owner, String fullPath)
  {
    Connection conn = null;
    MysqlDataSource dataSource = new MysqlDataSource();
    boolean retVal = false;

    try
    {
      dataSource.setServerName(GlobalVars.dbLocation);
      dataSource.setPortNumber(GlobalVars.dbPort);
      dataSource.setDatabaseName(GlobalVars.dbName);
      dataSource.setUser(GlobalVars.dbUser);
      dataSource.setPassword(GlobalVars.dbPassword);
      conn = dataSource.getConnection();

      if(conn.isValid(0))
      {
        Statement stat = conn.createStatement();
        String generatedQuery="DELETE␣FROM␣sharing␣" +
                   "WHERE␣owner='" + owner + "'␣" +
                   "AND␣fullpath='" + fullPath + "'";

        stat.executeUpdate(generatedQuery);

        stat.close();

        retVal = true;
      }

      conn.close();
    }
    catch(Exception e) {}

    return retVal;
  }

  public boolean removeWebFolder(String guest, String fullPath)
  {
    Connection conn = null;
    MysqlDataSource dataSource = new MysqlDataSource();
    boolean retVal = false;

    try
    {
      dataSource.setServerName(GlobalVars.dbLocation);
      dataSource.setPortNumber(GlobalVars.dbPort);
      dataSource.setDatabaseName(GlobalVars.dbName);
      dataSource.setUser(GlobalVars.dbUser);
      dataSource.setPassword(GlobalVars.dbPassword);

      conn = dataSource.getConnection();

      if(conn.isValid(0))
      {
        Statement stat = conn.createStatement();
        String generatedQuery="DELETE␣FROM␣sharing␣" +
                   "WHERE␣guest='" + guest + "'␣" +
                   "AND␣fullpath='" + fullPath + "'";

        stat.executeUpdate(generatedQuery);
        stat.close();

        retVal = true;
      }

      conn.close();
    }
    catch(Exception e) {}

    return retVal;
  }

  public boolean removeGuestContact(String owner,
                   String guest,
                   String fullPath)
  {
    Connection conn = null;
    MysqlDataSource dataSource = new MysqlDataSource();
    boolean retVal = false;

    try
    {
      dataSource.setServerName(GlobalVars.dbLocation);
      dataSource.setPortNumber(GlobalVars.dbPort);
      dataSource.setDatabaseName(GlobalVars.dbName);
      dataSource.setUser(GlobalVars.dbUser);
      dataSource.setPassword(GlobalVars.dbPassword);

      conn = dataSource.getConnection();

      if(conn.isValid(0))
      {
        Statement stat = conn.createStatement();
        String generatedQuery="DELETE␣FROM␣sharing␣" +
                   "WHERE␣owner='" + owner + "'␣" +
                   "AND␣guest='" + guest + "'␣" +
                   "AND␣fullpath='" + fullPath + "'";

        stat.executeUpdate(generatedQuery);

        stat.close();

        retVal = true;
      }

      conn.close();
    }
    catch(Exception e) {}
    return retVal;
  }

  public boolean approveSharedFolder(String owner,
                    String guest,
                    String fullPath)
  {
    Connection conn = null;
    MysqlDataSource dataSource = new MysqlDataSource();
    boolean retVal = false;

    try
    {
      dataSource.setServerName(GlobalVars.dbLocation);
      dataSource.setPortNumber(GlobalVars.dbPort);
      dataSource.setDatabaseName(GlobalVars.dbName);
      dataSource.setUser(GlobalVars.dbUser);
      dataSource.setPassword(GlobalVars.dbPassword);

      conn = dataSource.getConnection();

      if(conn.isValid(0))
      {
        Statement stat = conn.createStatement();
        String generatedQuery="UPDATE sharing " +
                  "SET approved=1 " +
                  "WHERE owner='" + owner + "' " +
                  "AND guest='" + guest + "' " +
                  "AND fullpath='" + fullPath + "'";

        stat.executeUpdate(generatedQuery);

        stat.close();

        retVal = true;
      }

      conn.close();
    }
    catch(Exception e) {}

    return retVal;
  }

  public ArrayList<String> getGuestContacts(String owner, String fullPath)
  {
    Connection conn         = null;
    MysqlDataSource dataSource  = new MysqlDataSource();
    ArrayList<String> retList = new ArrayList<String>();
    try
    {
      dataSource.setServerName(GlobalVars.dbLocation);
      dataSource.setPortNumber(GlobalVars.dbPort);
      dataSource.setDatabaseName(GlobalVars.dbName);
      dataSource.setUser(GlobalVars.dbUser);
      dataSource.setPassword(GlobalVars.dbPassword);

      conn = dataSource.getConnection();

      if(conn.isValid(0))
      {
        Statement stat = conn.createStatement();
        String mysqlQuery="SELECT guest " +
                   "FROM sharing " +
                   "WHERE owner='" +
                   owner +
                   "' AND fullpath='" +
                   fullPath + "'";

        ResultSet res=stat.executeQuery(mysqlQuery);

        while(res.next())
          retList.add(res.getString("guest"));

        stat.close();
      }

      conn.close();
    }
    catch(Exception e) {}

    return retList;
  }

  public boolean isApproved(String guest, String fullPath)
  {
    Connection conn         = null;
    MysqlDataSource dataSource  = new MysqlDataSource();
    boolean retCode         = false;

    try
    {
      dataSource.setServerName(GlobalVars.dbLocation);
      dataSource.setPortNumber(GlobalVars.dbPort);
      dataSource.setDatabaseName(GlobalVars.dbName);
      dataSource.setUser(GlobalVars.dbUser);
      dataSource.setPassword(GlobalVars.dbPassword);
      conn = dataSource.getConnection();

      if(conn.isValid(0))
      {
        Statement stat = conn.createStatement();
        String mysqlQuery="SELECT␣approved␣" +
                    "FROM␣sharing␣" +
                    "WHERE␣guest='" +
                    guest +
                    "'␣AND␣fullpath='" +
                    fullPath + "'";

        ResultSet res=stat.executeQuery(mysqlQuery);

        while(res.next())
          if(res.getInt("approved")==1)
            retCode = true;

        stat.close();
      }

      conn.close();
    }
    catch(Exception e) {}

    return retCode;
  }

  public ArrayList<String> getWebFolders(String guest)
  {
    Connection conn        = null;
    MysqlDataSource dataSource  = new MysqlDataSource();
    ArrayList<String> retList = new ArrayList<String>();

    try
    {
      dataSource.setServerName(GlobalVars.dbLocation);
      dataSource.setPortNumber(GlobalVars.dbPort);
      dataSource.setDatabaseName(GlobalVars.dbName);
      dataSource.setUser(GlobalVars.dbUser);
      dataSource.setPassword(GlobalVars.dbPassword);

      conn = dataSource.getConnection();

      if(conn.isValid(0))
      {
        Statement stat = conn.createStatement();
        String mysqlQuery="SELECT␣owner,␣fullpath,␣approved␣" +
                 "FROM␣sharing␣" +
                 "WHERE␣guest='" +
                 guest + "'";

        ResultSet res=stat.executeQuery(mysqlQuery);

        while(res.next())
        {
          retList.add(res.getString("owner"));
          retList.add(res.getString("fullpath"));

          if(res.getInt("approved")==0)
            retList.add("false");
          else
            retList.add("true");
        }

        stat.close();
      }

      conn.close();
    }
    catch(Exception e) {}

    return retList;
  }
}
```

Bibliography

[ama01] Amazon.com, Inc., *Amazon S3 homepage.* 2012, February 2012. <http://aws.amazon.com/s3/>

[ant10] Anthony T. Velte, Toby J. Velte, and Robert Elsenpeter. *Cloud Computing: A Practical Approach.* New York, McGraw-Hill, 2010, pp. 29-31.

[box01] Alexa Internet, Inc., *Box.net homepage.* 2012, February 2012. <http://www.box.com>

[dro01] Dropbox, Inc., *Dropbox homepage.* 2012, February 2012. <http://www.dropbox.com>

[euc01] Eucalyptus Systems, Inc., *Eucalyptus homepage.* 2011, February 2012. <http://www.eucalyptus.com>

[fur10] Furht, Borko and, Escalante, Armando. *Handbook of Cloud Computing.* New York, Springer, 2010, pp. 357-375.

[gog01] Google, Inc., *Google Cloud Storage homepage.* 2011, February 2012. <https://developers.google.com/storage/>

[gog02] Google, Inc., *Google Cloud Storage - Developers Guide (API v1.0).* 2011, February 2012. <https://developers.google.com/storage/docs/reference/v1/developer-guidev1>

[gog03] Google, Inc., *Google Cloud Storage - Reference Methods (API v1.0).* 2011, February 2012. <https://developers.google.com/storage/docs/reference-methods>

[icl01] Apple, Inc., *iCloud homepage.* 2012, February 2012. <http://www.icloud.com>

[ihd01] Gino Cappelli, *iHD homepage.* February 2012, February 2012. <http://www.i-hd.eu>

[mil09] Miller, Michael. *Cloud Computing: Web-Based Applications That Change the Way You Work and Collaborate Online.* Indianapolis, Que Publishing, 2009, pp. 80.

[sch12] Schulz, Greg. *Cloud and Virtual Data Storage Networking: Your journey to efficient and effective information services.* New York, Taylor & Francis Group, 2012, pp. 73-89.

[shr10] Shroff, Gautam. *Enterprise Cloud Computing: Technology, Architecture, Applications.* New York, Cambridge University Press, 2010, pp. 64-74.

[sye11] Syed A. Ahson and, Mohammad Ilyas. *Cloud Computing and: Software Services.* New York, Taylor & Francis Group, 2011, pp. 20.

[sky01] Microsoft, Inc., *Windows Live SkyDrive homepage.* 2012, February 2012. <http://skydrive.live.com>